PUBLISHED BY PROJECT MOSAIC BOOKS

TUFTE-LATEX.GOOGLECODE.COM

Cover inset photo: Richard Marx. Cover photo: Lichens on the North Shore, by the author.

First Corrected Printing, November 2015

Preface

Information is what we want, but data are what we've got. The techniques for transforming data into information go back hundreds of years. A good starting marker is 1592 with the publication of weekly "bills of mortality" in London. These bills were tabulations: a condensing of the data into a form more readily assimilated by the human reader. Constructing such tabulations was a manual operation.

As data became larger, machines were introduced to speed up the tabulations. A major step was Hollerith's development of punched cards and an electrical tabulating system for the US Census of 1890. This was so successful that Hollerith started a company, IBM, that came to play an important role in the development of today's electronic computers.

Also in the late 19th century, statistical methods began to develop rapidly. These methods have been tremendously important in interpreting data, but they were not intrinsically tied to mechanical data processing. Generations of students have learned to carry out statistical operations by hand on small sets of data, typically from a laboratory experiment.

Nowadays, it's common to have datasets that are so large they can be processed only by machine. In this era of "big data," often data are amassed by networks of instruments and computers. The settings for this are diverse: the genome, satellite observations of Earth, entries by web users, sales transactions, etc. With such data, there are new opportunities for finding and characterizing patterns using techniques such as data mining, machine learning, data visualization, and so on. As a result, everyday uses of data involve computer processing of data. This includes data cleaning, combining data from multiple sources, and transformation into a form suitable for input to data-condensing operations like visualization.

In writing this book I hope to help people gain the understanding and skills for data wrangling, a process of preparing data for visualization and other modern techniques of statistical interpretation. Doing so inevitably involves, at the center, using the computer in a sophisticated way.

Need sophisticated computing require extended study of computer programming? My view is that it does not. First, over the last half century, a coherent set of simple data operations have been developed that can be used as building blocks of sophisticated data wrangling processes. The trick is not mastering programming but learning to think in terms of these operations. Much of this book is intended to help you master such thinking. Second, it's possible to use recent developments in software to reduce vastly the amount of programming needed to use the data operations. I've drawn on such software — particularly R and the packages `dplyr` and `ggplot2` developed by Hadley Wickham and the many others who contribute to important open-source projects — to focus on a small subset of functions that accomplish data wrangling tasks in a concise and expressive way. The computer notation is clear enough that, once you have learned to read it, constructing new data wrangling programs can be done by modifying working examples. (Experienced R programmers will note the distinctive style of R statements in this book, including a consistent focus on a small set of key notation.)

I've tried to arrange this book to be accessible to a reader with no technical computing background. The book derives from notes for a course at Macalester College, *Data Computing Fundamentals*, that has no pre-requisites. To be effective, the book should be read along with practice using the computer. Some of this practice involves becoming familiar with the user-interface software. This includes the RStudio environment for computing with R, the use of a text editor, and RMarkdown, the notation that integrates the computer commands with graphical display and explanatory narrative. It's hard to write effective explanations of how to work with user-interfaces; doing it is the best way to learn, watching others do it is also helpful. If you're using this book in a classroom setting, your instructor can provide the demonstrations. If you are reading this book on your own, you can make use of the many videos that introduce using RStudio, its editor, and the RMarkdown document-writing system.

The contributions of many people are behind this book. At a start, this includes the work of the people in the R Core Team and the fantastic community of open-source R developers. Of particular importance to *Data Computing* are Hadley Wickham, Winston Chang, Joe Cheng, Garrett Grolemund, and JJ Allaire at RStudio. Professors Nicholas Horton, Randall Pruim, Elizabeth Shoop and Paul Overvoorde helped in defining the Data Computing Fundamentals course. Collaborators on Project MOSAIC helped encourage the *minimal-R* approach that let's you do a lot with a little bit of R. The many faculty participants in the Computing and Visualization Consortium of small liberal arts colleges provided important feedback to identify

gaps and shortcomings. The careful attention of Jeff Witmer, Craig Ching, Michael Schneider, and Dennis McGuire pointed out many deficiencies in the manuscript. Students in the Data Computing Fundamentals course at Macalester College willingly hiked the initial rough path of the course curriculum. Macalester students Barbara Borges Ribiero, Mengdie Wang, and Jingjing Yang were teaching assistants for the course and spent summer months developing case studies, visualizations, and interactive "apps" for displaying what data verbs do. Andrew Zieffler, Ethan Brown, and Erik Anderson from the University of Minnesota helped in sharpening assessment, developing case studies, and teaching an early version of the course. Prof. Jessen Havill at Denison University put together a Mellon Foundation workshop in 2010 to discuss how to bring computing into the science curriculum; the ideas of a short course and a focus on data emerged from that workshop.

Financial support for the Data Computing Fundamentals course and the Computation and Visualization Consortium was generously provided by the Howard Hughes Medical Institute. The National Science Foundation supported Project MOSAIC (NSF DUE-0920350). Both were instrumental to developing the Macalester course and this book.

My dear wife, Maya, and our beloved daughters Tamar, Liat, and Netta were encouraging (and patient) during many dinner-table discussions of case studies and the many hours I was away writing notes for the Data Computing course and teaching the Wednesday-evening class sessions.

Daniel Kaplan
St. Paul, Minnesota
August 2015

Contents

1
Tidy Data

Data can be as simple as a column of numbers in a spreadsheet file or as complex as the medical records collected by a hospital. A newcomer to working with data may expect each source of data to be organized in a unique way and to require unique techniques. The expert, however, has learned to operate with a small, standard set of tools. As you'll see, each of the standard tools performs a comparatively simple task. Combining those simple tasks in appropriate ways is the key to dealing with complex data.

One of the reason the individual tools can be simple is this: each tool gets applied to data arranged in a simple but precisely defined pattern called *tidy data*. At the heart of tidy data is the *data table*. To illustrate, Table 1.1 a handful of entries from a large US Social Security Administration tabulation of names given to babies. In particular, the table shows how many babies of each sex were given each name in each year.

TIDY DATA: A format for data that provides a systematic approach to computer processing. You will be using tidy data throughout this book and, likely, for the large majority of your professional work with data.

DATA TABLE: A rectangular array of data.

name	sex	count	year
Sherina	F	19	1970
Leketha	F	6	1973
Tahirih	F	6	1976
Radha	F	13	1979
Hatim	M	8	1981
Cissy	F	7	1984
Ahmet	M	6	1986
... and so on for 1,792,091 rows			

Table 1.1: A data table showing how many babies were given each name in each year in the US.

Table 1.1 shows there were 6 boys named Ahmet born in the US in 1986 and 19 girls named Sherina born in 1970. As a whole, the `BabyNames` data table covers the years 1880 through 2013 and includes a total of 333,417,770 individuals, somewhat larger than the current population of the US.

The data in Table 1.1 are "tidy" because they are organized according to two simple rules.

1. The rows, called *case*s, each refer to a specific, unique and similar sort of thing, e.g. girls named Sherina in 1970.
2. The columns, called *variable*s, each have the same sort of value recorded for each row. For instance, `count` gives the number of babies for each case; `sex` tells which gender the case refers to.

CASE: The individual people, objects, events, etc. from which variables are measured.

VARIABLE: The attributes of each case that are measured or observed.

When data are in tidy form, it's relatively straightforward to transform the data into arrangements that are more useful for answering interesting questions. For instance, you might wish to know which

were the most popular baby names over all the years. Even though Table 1.1 contains the popularity information implicitly, some re-arrangement is needed (for instance, adding up the counts for a name across all the years) before the information is made explicit, as in Table 1.2.

The process of transforming information that is implicit in a data table into another data table that gives the information explicitly is called *data wrangling*. The wrangling itself is accomplished by using *data verbs* that take a tidy data table and transform it into another tidy data table in a different form. In the following chapters, you will be introduced to the various *data verbs*.

sex	name	total_births
M	James	5091189
M	John	5073958
M	Robert	4789776
M	Michael	4293460
F	Mary	4112464
... and so on for 102,690 rows		

Table 1.2: The most popular baby names across all years.

	A	B	E	F	G	H	I	J
1			City of Minneapolis Statistics					
2			General Election November 5, 2013					
3	Ward	Precinct	Voters Registering by Absentee	Total Registrations	Voters at Polls	Absentee Voters	Total Ballots Cast	Total Turnout
4	City-Wide Total		708	6,634	75,145	4,954	80,099	33.38%
5								
6	1	1	3	28	492	27	519	27.23%
7	1	2	1	44	836	56	892	31.71%
8	1	3	0	40	905	19	924	38.87%
9	1	4	5	29	768	26	794	36.62%
10	1	5	0	31	683	31	714	37.46%
11	1	6	0	69	739	20	759	32.62%
12	1	7	0	47	291	8	299	15.79%
13	1	8	0	43	415	5	420	30.55%
14	1	9	0	42	596	25	621	25.42%
15	Ward 1 Subtotal		9	373	5,725	217	5,942	30.93%
16								
17	2	1	1	63	1,011	39	1,050	36.42%
18	2	2	5	44	679	37	716	50.39%
19	2	3	4	48	324	18	342	18.88%
20	2	4	0	53	117	3	120	7.34%
21	2	5	2	50	495	26	521	25.49%
22	2	6	1	36	433	19	452	39.10%
23	2	7	0	39	138	7	145	13.78%
24	2	8	1	50	1,206	36	1,242	47.90%
25	2	9	2	39	351	16	367	30.56%
26	2	10	0	87	196	5	201	6.91%
27	Ward 2 Subtotal		16	509	4,950	206	5,156	27.56%
28								
29	3	1	0	52	165	1	166	7.04%

Table 1.3: Ward and precinct votes cast in the 2013 Minneapolis mayoral election.

Table 1.3, in contrast to `BabyNames`, is not in tidy form. True, table 1.3 is attractive and neatly laid out. There are helpful labels and summaries that make it easy for a person to read and draw conclusions. (For instance, Ward 1 had a higher voter turnout than Ward 2, and both wards were lower than the city total.)

Being neat is not what makes data tidy. Table 1.3, however neat it is, violates the rules for tidy data.

- Rule 1: The rows, called *cases*, each must represent the same underlying attribute, that is, the same kind of thing.

  That's not true in Table 1.3. For most of the table, the rows represent a single precinct. But other rows give ward or city-wide totals. The first two rows are captions describing the data, not cases.

- Rule 2: Each column is a variable containing the same type of

value for each case.

That's mostly true in Table 1.3, but the tidy pattern is interrupted by labels that are not variables. For instance, the first two cells in row 15 are the label "Ward 1 Subtotal," different from the ward/precinct identifiers that are the values in most of the first column.

Conforming to the rules for tidy data simplifies summarizing and analyzing data. For instance, in the tidy baby names table, it's easy to find the total number of babies: just add up all the numbers in the `count` variable. It's similarly easy to find the number of cases: just count the rows. And if you want to know the total number of Ahmeds or Sherinas across the years, there's an easy way to do that.

In contrast, it would be more difficult in the Minneapolis election data to find, say, the total number of ballots cast. If you take the seemingly obvious approach and add up the numbers in column I of Table 1.3 (labelled "Total ballots cast"), the result will be *three times* the true number of ballots, because some of the rows contain summaries, not cases.

Indeed, if you wanted to do calculations based on the Minneapolis election data, you would be far better off to put it in a tidy form, like Table 1.4.

ward	precinct	registered	voters	absentee	total.turnout
1	1	28	492	27	0.27
1	4	29	768	26	0.37
1	7	47	291	8	0.16
2	1	63	1011	39	0.36
2	4	53	117	3	0.07
... and so on for 117 rows					

Table 1.4: The Minneapolis election data in tidy form.

The tidy form is, admittedly, not as attractive as the form published by the Minneapolis government. But the tidy form is much easier to use for the purpose of generating summaries and analyses.

Once data are in a tidy form, you can present them in ways that can be more effective than a formatted spreadsheet. Figure 1.1 presents the turnout in each ward that makes it easy to see how much variation there is within and among precincts.

The tidy format also makes it easier to bring together data from different sources. For instance, to explain the variation in voter turnout, you might want to look at variables such as party affiliation, age, income, etc. Such data might be available on a ward-by-ward basis from other records, such as the (public) voter registration logs and census records. Tidy data can be *wrangle*d into forms that can be

WRANGLE: Data wrangling is the process of reforming, summarizing, and combining data to make it more suitable for a given purpose.

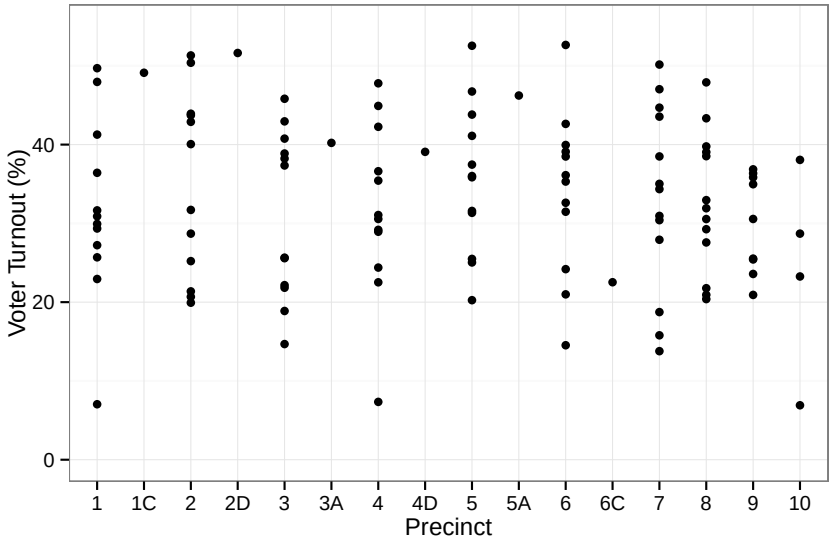


Figure 1.1: A graphical depiction of voter turnout in the different wards.

connected to one another. This would be difficult if you had to deal with an idiosyncratic format for each different source of data.

1.1 *Variables*

In data science, the word *variable* has a different meaning than in mathematics. In algebra, a variable is an unknown quantity. In data, a variable is known; it has been measured. "Variable" refers to a specific quantity or quality that can vary from case to case.

VARIABLE: A quantity or quality that varies from one case to another.

There are two major types of variables: *categorical* and *quantitative* A quantitative variable is just what it sounds like: a number.

A categorical variable tells which category or group a case falls into. For instance, in the baby names data table, `sex` is a categorical variable with two *levels* F and M, standing for female and male. Similarly the `name` variable is categorical. It happens that there are 92,600 different levels for `name`, ranging from Aaron, Ab, and Abbie to Zyhaire, Zylis, and Zymya.

CATEGORICAL: One of the two major kinds of variables. Categorical variables record type or category and often take the form of a word.

QUANTITATIVE: The other of the two major types of variables. A quantitative variable records a numerical attribute.

LEVELS: The individual possibilities for a categorical variable.

1.2 *Cases and what they represent*

As already stated, a row of a tidy data table refers to a case. To this point, you may have little reason to prefer the word *case* to *row*.

When working with a data table, it's important to keep in mind what a case stands for in the real world. Sometimes the meaning is obvious. For instance, Table 1.5 is a tidy data table showing the ballots in the Minneapolis mayoral election in 2013. Each case is an individual voter's ballot. (The voters were directed to mark their

ballot with their first choice, second choice and third choice among the candidates. This is part of a procedure called rank choice voting: `http://vote.minneapolismn.gov/rcv/`.)

Precinct	First	Second	Third	Ward
P-04	undervote	undervote	undervote	W-6
P-06	BOB FINE	MARK ANDREW	undervote	W-10
P-02D	NEAL BAXTER	BETSY HODGES	DON SAMUELS	W-7
P-01	DON SAMUELS	undervote	undervote	W-5
P-03	CAM WINTON	DON SAMUELS	OLE SAVIOR	W-1
... and so on for 80,101 rows				

Table 1.5: Individual ballots in the Minneapolis election. Each voter votes in one ward in one precinct. The ballot marks the voter's first three choices for mayor.

The case in Table 1.5 is a different sort of thing than the case in Table 1.4. In Table 1.4, a case is a ward in a precinct. But in Table 1.5, the case is an individual ballot. Similarly, in the baby names data (Table 1.1), a case is a name and sex and year while in Table 1.2 the case is a name and sex.

When thinking about cases, ask this question: What description would make every case unique? In the vote summary data, a precinct does not uniquely identify a case. Each individual precinct appears in several rows. But each precinct & ward combination appears once and only once. Similarly, in Table 1.1, name & sex do not specify a unique case. Rather, you need the combination of name & sex & year to identify a unique row.

Example: Runners and Races Table 1.6 shows some of the results from a 10-mile running race held each year in Washington, D.C.

name.yob	sex	age	year	gun
jane polanek 1974	F	32	2006	114.50
jane poole 1948	F	55	2003	92.72
jane poole 1948	F	56	2004	87.28
jane poole 1948	F	57	2005	85.05
jane poole 1948	F	58	2006	80.75
jane poole 1948	F	59	2007	78.53
jane schultz 1964	F	35	1999	91.37
jane schultz 1964	F	37	2001	79.13
jane schultz 1964	F	38	2002	76.83
jane schultz 1964	F	39	2003	82.70
jane schultz 1964	F	40	2004	87.92
jane schultz 1964	F	41	2005	91.47
jane schultz 1964	F	42	2006	88.43
jane smith 1952	F	47	1999	90.60
jane smith 1952	F	49	2001	97.87
... and so on for 41,248 rows				

Table 1.6: An excerpt of runners' performance over the years in a 10-mile race.

What's the meaning of a case here? It's tempting to think that a case is a person. After all, it's people who run road races. But notice that individuals appear more than once: Jane Poole ran each year from 2003 to 2007. (Her times improved consistently as she got older!) Jane Smith ran in the races from 1999 to 2006, missing only the year 2000 race. This suggests that the case is a runner in one year's race.

CODEBOOK: A document separate from the data table detailing the meaning of each variable in the table, how levels of categorical variables are encoded, units for quantitative variables, etc.

1.3 *The Codebook*

Data tables do not necessarily display all the variables needed to figure out what makes each row unique. For such information, you sometimes need to look at the documentation of how the data were collected and what the variables mean.

The *codebook* is a document — separate from the data table — that describes various aspects of how the data were collected, what the variables mean and what the different levels of categorical variables refer to. Figure 1.2 shows the codebook for the `BabyNames` data in Figure 1.1.

The word "codebook" comes from the days when data was encoded for the computer in ways that make it hard for a human to read. A codebook should also include information about how the data were collected and what constitutes a case.

For the runners data in Table 1.6, the codebook tells you that the meaning of the `gun` variable is the time from when the start gun went off to when the runner crosses the finish line and that the unit of measurement is "minutes." It should also state what might be obvious: that `age` is the person's age in years and `sex` has two levels, male and female, represented by M and F.

BabyNames (DCF) R Documentation

Names of children as recorded by the US Social Security Administration.

Description

The US Social Security Administration provides yearly lists of names given to babies. These data combine the yearly lists.

`BabyNames` is the raw data from the SSA. The case is a year-name-sex, for example: Jane F 1922. The `count` is the number of children of that sex given that name in that year. Names assigned to fewer than five children of one sex in any year are not listed, presumably out of privacy concerns.

Usage

```
data(BabyNames)
```

Format

`BabyNames` consists of 1,792,091 entries, each of which has four variables:

`name`
: The given name (character string)

`sex`
: F or M (character string)

`count`
: The number of babies given that name and of that sex. (integer)

`year`
: Year of birth (integer)

Source

The data were compiled from the Social Security Administration web site: `http://www.ssa.gov/oact/babynames/names.zip`.

Figure 1.2: The codebook for the `BabyNames` data table.

1.4 *Multiple Tables*

It's often the case that creating a meaningful display of data involves combining data from different sources and about different kinds of things. For instance, you might want your analysis of the runners' performance data in Table 1.6 to include temperature and precipitation data for each year's race. Such weather data is likely contained in a table of daily weather measurements.

In many circumstances, there will be multiple tidy tables, each of which contains information relative to your analysis but has a different kind of thing as a case. Chapter 10 is about techniques combining data from such different tables. For now, keep in mind that being tidy is not about shoving everything into one table.

Exercises

Problem 1.1
Here is an excerpt from the baby-name data table in `"DataComputing::BabyNames"`.

name	sex	count	year
Taffy	F	19	1970
Liliana	F	162	1973
Stan	M	55	1975
Nettie	F	45	1978
Kateria	F	8	1980
... and so on for 1,792,091 rows			

Consider these five entities, that appear in the table shown above (a) through (e):

a) `Taffy` b) `year` c) `sex` d) `name` e) `count`

For each, choose one of the following:

1. It's a categorical variable.
2. It's a quantitative variable.
3. It's the value of a variable for a particular case.

Problem 1.2
What's not tidy about this table?

`president`	`in office`	`number of states`
Lincoln, Abraham	1861-1865	it depends
George Washington	1791-1799	16
Martin Van Buren	1837 to 1841	26

Table 1.7: An untidy table

Problem 1.3
Re-write Table 1.7 in a tidy form. Take care to render the information about years and about the number of states as numbers.

Problem 1.4
Here are three different organizations (**A**, **B**, and **C**) of the same data:

Data Table A

Year	Algeria	Brazil	Columbia
2000	7	12	16
2001	9	14	18

Data Table B

Country	Y2000	Y2001
Algeria	7	9
Brazil	12	14
Columbia	16	18

Data Table C

Country	Year	Value
Algeria	2000	7
Algeria	2001	9
Brazil	2000	12
Columbia	2001	18
Columbia	2000	16
Brazil	2001	14

1. What are the variables in each table?
2. What is the meaning of a case for each table? Here are some possible choices.
   - A country
   - A country in a year
   - A year

Problem 1.5
The codebook for several data tables relating to airports, airlines, and airline flights in the US is published at `https://cran.r-project.org/web/packages/nycflights13/nycflights13.pdf`.

Within that document is the codebook for the data table `airports`.

1. How many variables are there?
2. What do the cases represent?
3. For each variable, make a reasonable guess about whether the values will be numerical or quantitative.

2
Computing with R

Data tables are often large, so it is not possible to undertake paper-and-pencil operations on them. The `BabyNames` data table 1.1 provides a case in point. `BabyNames` has 1,790,091 rows, far too many to carry out by hand even a simple sum or count. Data science relies on computers.

The human role in the process is managerial, to decide what forms of analysis are called for by the purpose at hand and to instruct the computer to carry out the operations needed. The process of instructing a computer is called *computer programming*. Programming is done using a language that is accessible both to humans and the computer; the human writes the instructions and the computer carries them out. These notes use a computer language called R.

COMPUTER PROGRAMMING: The process of writing instructions for a computer. Sometimes the unfortunate word "coding" is used. The point of programming is to make instructions clear, not to obscure them with code.

This chapter introduces concepts that underlie the use of R. The emphasis here will be on the *kinds of things* that R commands involve. Only a few R commands will be covered, but these will illustrate the most important patterns in writing with R. Chapter 3 will start to introduce the commands themselves that are used for working with data.

2.1 R and RStudio

The *R* language includes many useful operations for working with data, drawing graphics, and writing documents, among other things.

R: A language for working with data.

RStudio is a computer application that gives ready access to the resources of R. RStudio provides a powerful way for a person to interact with R. There are other ways to work with R, but for good reason RStudio has become the most popular. RStudio offers facilities both for newbies and for professionals; it's good for everybody.

RSTUDIO: An interface for organizing your work and interacting with R.

RStudio comes in two versions:

1. A *desktop* that runs on the user's own computer.
2. A *server* that runs on a computer in the cloud and is accessed by a

DESKTOP: An application that runs on your own computer.

SERVER: An application that is being run remotely, as a service to many users.

web browser.

You can install R and the desktop version of RStudio on all the most common operating systems: OS-X, Windows, Linux. There are now many text and video guides to installation. See `marin-stats-lectures` for an example.

If you are to use the server version of RStudio, someone has set up a server for you and provided a login ID. You access the server through an ordinary web browser — Chrome, Firefox, Safari, etc. — that may be running on a laptop, a tablet, or even a smartphone.

Setting up an RStudio server? This requires some information technology (IT) knowledge and knowledge of the UNIX operating system. If you're comfortable with such things, there are many guides on the web.

The server version slows down when there are many simultaneous users, as might happen in a classroom.

The two RStudio versions, desktop or server, are almost identical from a user's point of view. The desktop version is often faster but can only be used on one computer; the server version requires almost no set-up and is available from *any* computer browser, but it can be slower.

2.2 *Components of the RStudio application*

PANES: Regions of the RStudio application that are displayed side-by-side.

TABS: Subdivisions of a pane, only one of which can be viewed at one time.

Just as a window in a house has several individual panes of glass, so the RStudio application appears in an graphical interface window divided up into several *panes*. All panes can be viewed at the same time. Each of the panes can contain several *tabs*.

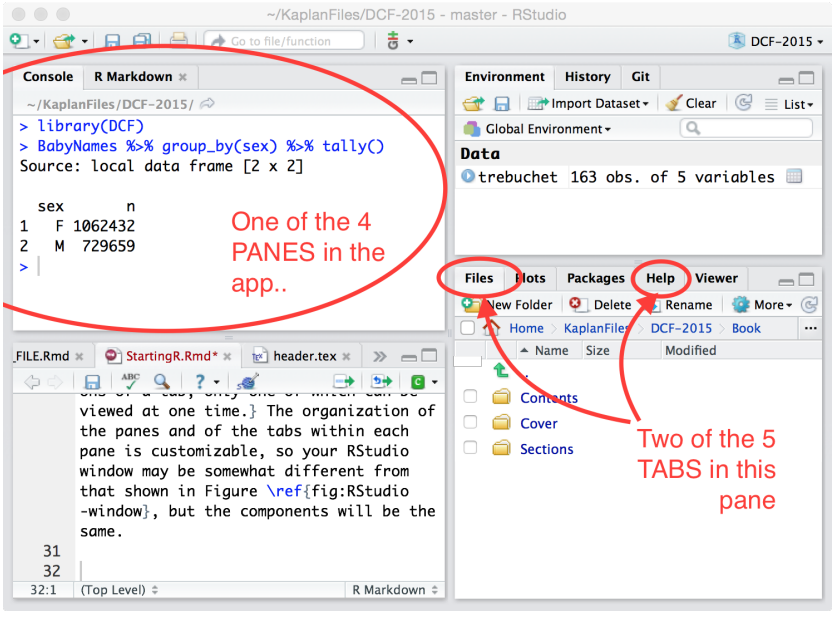


Figure 2.1: The RStudio window is divided into four panes. Each panes can contain multiple tabs.

The organization of the panes and of the tabs within each pane is customizable, so your RStudio window may be somewhat different from that shown in Figure 2.1, but the components will be the same.

You will often be using the Console tab and tabs for editing documents. Each document-editing tab has the name of the document it contains. In Figure 2.1, three such editing tabs are shown, one of which is labeled `StartingR`. Other important tabs: Packages for installing new R software and Help for displaying documentation.

2.3 *Commands and the Console*

This is a good time to start up your version of RStudio. Once RStudio is running:

- Enter R-language commands into the *console*. Find the console tab in your RStudio app and the prompt, >.
- Move the cursor to the console tab so that anything you type shows up there. Type the simple command shown in Figure 2.2.
- After you press *enter*, ↩, R will *execute* your command and print out a response.

Notice that after the response, there is another prompt. You're ready to go again.

PRACTICE. Enter each of the following arithmetic commands at the command prompt, one at a time. Confirm that the response is the right arithmetic answer.

```
16 * 9
sqrt(2)
20 / 5
18.5 - 7.21
```

Figure 2.2: The console tab showing the command prompt > (top), a command composed but not yet entered (middle), and after entering the command. (bottom)

2.4 *Sessions*

Your work in R occurs in *sessions*. A session is a kind of ongoing dialog with the R system.

A new session is begun every time you start RStudio. The session is terminated only when you close or *quit* the RStudio program. If you are using RStudio server (via your browser), the R session will be maintained for days or weeks or months, and will be retained even when you login to the server from a different computer.

When a session is first started, it is in an empty environment. RStudio displays this in the *Environment* tab. (Figure 2.3)

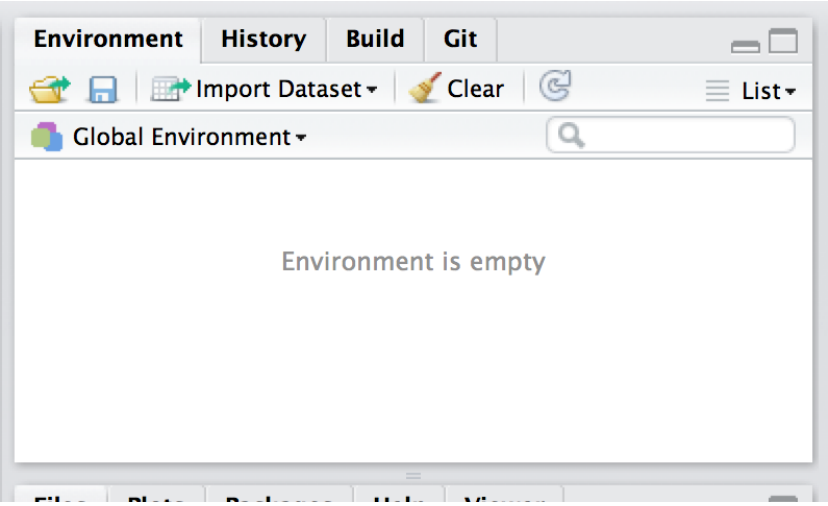


Figure 2.3: The "Environment" tab in RStudio lists the objects in the "session environment." At the start of a session, the environment is empty.

2.5 *Packages*

Many people work to provide and maintain new software and new capabilities for R. Such additions are called *package*s. Among the packages you will be using, the *DataComputing package* contains data for the examples in this book. In order to make these data accessible, you have to *load* the package. The command is:

```
library(DataComputing)
```

Usually you will give this command at the start of a session or at the top of a document.

PACKAGE: The standard way of distributing new software to run within R.

DATACOMPUTING PACKAGE: R software written as a companion to this book. You will need to install it once, then load it each time you start up R to work with the book.

LOAD: Instructing R to refer to commands and data in an R package.

Did something go wrong?

In response to the above command, you might get a response from R like this:

```
Error in library(DataComputing) : there is no package called 'DataComputing'
```

If this happens, it means only that your R account is not up to date with the `DataComputing` package and you will have to *install* the package. To do so, use these two commands in sequence. (Copy the commands verbatim into your R console. The installation may take some minutes depending on the speed of your Internet connection)

INSTALL: A one-time process that copies the contents of a package from its distribution site onto your computer or your account on the RStudio server.

```
install.packages("devtools")
devtools::install_github("DataComputing/DataComputing")
```

Note the repeated `"DataComputing/DataComputing"`.

Then try again with `library(DataComputing)`.

2.6 *Data and Objects*

The data you work with will be organized into *data table*s. Data tables are generally stored in files or database systems or even web pages. To use a data table in R, you need to read the data into your R session. There are many ways to do this. The simplest, and the one we will use in this introduction, is via the `data()` function.

DATA TABLE: A spreadsheet-like configuration of tidy data. Another term used by R users is data frame

You can list the data in a package with this command:

```
data(package = "DataComputing")
```

The command shows a table like the following in an editing tab:

Item	Title
BabyNames	Names of children as recorded by the US Social Security Administration.
CountryGroups	Membership in Country Groups
HappinessIndex	World Happiness Report Data
MedicareCharges	Charges to and Payments from Medicare
	... and so on for 19 rows

To *read* a data table into R from the `DataComputing` package, use the `data()` function with the name of the data table and the name of the package, as in:

READ: To bring a data table into an R session. There are many ways to read data tables; `data()` will suffice for the present.

```
data("NCHS", package = "DataComputing")
```

To take a peak at the data that has already been read into an ongoing session, use `View()`. This will display the object in a new tab as in Figure 2.4.

```
View(NCHS)
```

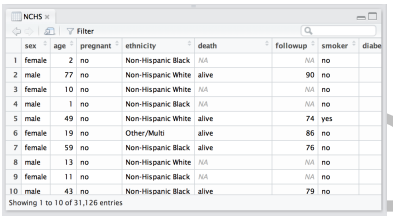


Figure 2.4: The tab opened by RStudio in response to `View(NCHS)`.

Another way to get information about the data is to click on the expand icon, ▶ next to the NCHS listing in the environment tab. You can look at the data table more closely by clicking on the small spreadsheet icon, ▦. Finally, for data tables contained in a package, you can display a narrative description — the codebook — using the `help()` command:

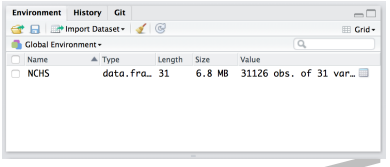


Figure 2.5: The session environment after the NCHS data table has been loaded.

If you haven't already done so, go back to the beginning and type each command into the R console, find the environment tab, and press the spreadsheet icon ▦ and the summary icon ▶.

```
help("NCHS")
```

2.7 *Functions, arguments and commands*

The examples above demonstrate many of the most important components needed to use R. Giving these components names will help in communicating about using R.

FUNCTIONS ARE THE MECHANISM THAT R USES TO CARRY OUT AN OPERATION. Functions take one or more inputs and produce an output. You've seen several functions already: `sqrt()`, `library()`, `data()`, and `help()`. Functions have names. In this text, function names will be written followed by open and close parentheses simply to signal to you, the human reader, that the name refers to a function, as opposed to, say, `NCHS`, which is a data table, not a function.

FUNCTION: The software container for an operation that transforms one or more inputs into an output.

Think of a function as a **verb**. It tells *what* to do.

A COMMAND IS A COMPLETE STATEMENT OF A COMPUTATION. Commands are usually constructed by giving inputs to a function. Think of a command as a **sentence**.

COMMAND: In R, a command is a complete statement of what to do. Commands are made up of functions and arguments.

The grammar of R sentences is straightforward. Follow the *name* of the function with a pair of parentheses. Inside the parentheses, you put the input on which the operation is to be performed.

All of the commands above have this form: a function name followed by parentheses containing an argument.

```
sqrt(2)
library("DataComputing")
library("mosaicData")
data(package="DataComputing")
data(package="mosaicData")
data("NCHS")
help("NCHS")
```

You can also use an object name itself as a command.

```
NCHS
```

This is useful when you want to get a quick display of the value stored under that name.

Behind the scenes: Using an object name as a command is just shorthand for a command using the `print()` function. The explicit command would be `print(NCHS)` in this example.

THE INPUTS TO FUNCTIONS are called *argument*s. Here are some of the arguments in the above examples:

ARGUMENT: An input to a function. In R, the argument is enclosed in parentheses that follow the function name. If there is more than one argument, they are separated by commas, e.g. `arrange(CountryData, gdp)`

```
2
"DataComputing"
package = "DataComputing"
"NCHS"
```

It would be reasonable to call the inputs to functions simply "inputs," but that's not the convention.

The third example, `package="DataComputing"` is called a *named argument*. Named arguments are a way to signal clearly what role the argument is to play. In the command `data(package="DataComputing")` the name of the argument is `package` and the value `"DataComputing"` is the value given to that argument.

NAMED ARGUMENT: An argument whose role in the function is marked by a name. When a function takes named arguments as inputs, the name rather than the order determines how the input is used.

When there is more than one argument to a function, put them all in the same set of parentheses with a comma between the different arguments.

2.8 Objects

An *object* is a packet of information. Just about everything you'll be using in R is an object. There are many different sorts of objects, just like there are many sorts of things in the world.

OBJECT: a packet containing information. The word "object" is not any more descriptive than, say, "thing." At least, "object" makes it clear that you are referring to a thing in R.

It helps to distinguish between the packet and the information contained in it. The information contained in the object is called its *value*. Here are some of the objects and their values that appear in the examples above:

VALUE: The information contained in an object

Name	Value	Kind of object
`NCHS`	data	a data table
`sqrt`	computer commands	a function
	`"DataComputing"`	a string of characters
	`2`	a number
	`"NCHS"`	a string of characters

Most of the objects you will use have names, e.g. `NCHS` or `sqrt`. Sometimes you will use objects that don't have a name, for instance the number `2` or the quoted set of characters `"DataComputing"`.

It will take a while for you to get used to the difference between an object and the *name* of an object. Object names are never in quotes and they never begin with a digit. When quotes are used, it is to identify characters as a string. Strings are used for labels, or to identify something outside of R such as a web location, file name, or caption on a graphic.

Giving Names to Objects

Analyzing or visualizing data often involves several steps, each of which creates a new objects. It's often useful to name these objects so that you can refer to them in the following steps.

Name an object with the `<-` notation. The syntax is simple:

```
name <- value
```

In the jargon of computer programming, naming an object is called *assignment*. You assign a name to an object.

ASSIGNMENT: Storing a value along with a name that can be used later to refer to or access the value

EXAMPLE: RANDOM SAMPLING. It's often useful to take a random subset of the cases in a data table.[1] The function that does this is

[1] For instance, you might want to prototype with a small data table when developing your data wrangling and visualization before tackling the whole data table.

called `sample_n()`. It takes two arguments: the name of the data table from which the subset is to be drawn and a named argument, `size=`, that specifies how many cases should be in the sample. For example:

```
SmallNCHS <- sample_n(NCHS, size=100)
```

By assigning the output of `sample_n()` to a named object, `SmallNHCS` in the statement above, you can access the object when you need it.

You can have as many named objects as you like. RStudio helps you keep track of them by listing them in the Environment tab, as in Figure 2.6.

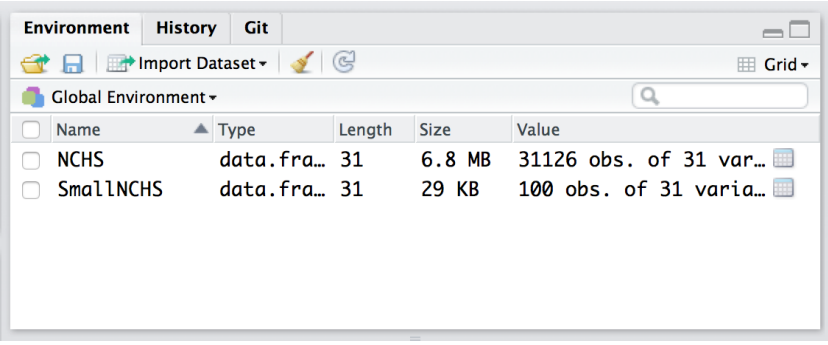


Figure 2.6: When the `SmallNCHS` object is created, it appears in the Environment tab.

THERE ARE A FEW SIMPLE RULES that apply when creating a name for an object:

- The name cannot start with a digit. So ~~`100NCHS`~~ is not allowed, although `NCHS100` is fine. This rule is to makes it easy for R to distinguish between object names and numbers. It also helps you avoid mistakes such as writing `2pi` when you mean `2*pi`.
- The name cannot contain any punctuation symbols (with two exceptions). So ~~`?NCHS`~~ or ~~`N*Hanes`~~ are not legitimate names. The exceptions: You can use `.` and `_` in a name.
- The case of the letters in the name matters. So `NCHS`, `nchs`, `Nchs`, and `nChs`, etc. are all different names that only look similar to a human reader, not to R.

The strikethrough bar, like ~~this~~, is not part of the name. The strikethrough is just a device in the notes to remind you that the name is not allowed.

Occasionally, you will encounter function names like `data.table::fread()`. This refers to a function contained in a specific package. In this case, the name refers to the `fread()` function in the `data.table` package.

EXAMPLE: READING A FILE The following command consists of a function, a quoted *character string* string, the assignment operator,

CHARACTER STRING: Letters, numbers, punctuation, etc. meant to be taken literally as a value. Character strings always start and end with quotes, for instance, `"My name is ..."`. In contrast, quotes are not used with object names.

and an object name.

```
Motors <- read.file("http://tiny.cc/mosaic/engines.csv")
```

The effect of this command is to read some data about internal combustion motors from a web site into an R object called `Motors`. Note that the URL of the data file is a quoted character string, but the function and object names are *not* quoted.

2.9 *Exercises*

Problem 2.1
The following ideas should be meaningful to you from Chapter 2:

package, *function*, *command*, *argument*, *assignment*, *object*, *object name*, *data table*, *named argument*, *quoted character string*, *value*

Construct an example R command that makes use of at least four of the ideas. Label which part of your example R command corresponds to each of those ideas.

Problem 2.2
Which of these kinds of names should be wrapped with quotation marks when used in R?

1. function name
2. file name
3. the name of an argument in a named argument
4. object name

Problem 2.3
Look at the documentation for the `CPS85` data table in the `mosaicData` package. From reading that documentation, what is the meaning of CPS?

Problem 2.4
What's wrong with this statement?

```
help(NHANES, package <- "NHANES")
```

Problem 2.5
Look at the help documentation for the `library()` function.

Without worrying about all the detail, answer these questions simply:

1. What is the other function listed under "Usage"?
2. In the "See Also" section of the documentation, what is the name of the function after `detach()`?

Problem 2.6
Some of these are legitimate object names, others are not. For the ones that are not legitimate, say what is wrong.

1. `essay14`
2. `first-essay`
3. `"MyData"`
4. `third_essay`
5. `small sample`
6. `functionList`
7. `FuNcTiOnLiSt`
8. `.MyData.`
9. `sqrt()`

Problem 2.7

Install the `nycflights13` package into R. (You can use the "Packages" tab which has an "install" button. If you are not using RStudio, given the R command `install.packages("nycflights13")`)

Once the package is installed, you can access the `flights` data table with this command:

```
data(flights, package="nycflights13")
```

The codebook is available with

```
help(flights)
```

Using the codebook and examining the data table with the `View()` command (hint: you'll need to give `flights` as an argument to `View()`), answer these questions:

1. How many variables are there?
2. How many cases are there?
3. What is the meaning of a case? ("Meaning" refers to the kind of entity, for instance, "airport" or "airline" or "date". Hint: the case in `flights` is not any of these things.)
4. For each variable, is the variable quantitative or categorical?
5. For the variables `air_time` and `distance`, what are the units?

Problem 2.8

Consider this list of some possible mistakes in an assignment operation:

1. No assignment operator
2. Unmatched quotes in character string
3. Improper syntax for function argument
4. Invalid object name
5. No mistake

For each of the following assignment statements, say what is the mistake.

a. `ralph <- sqrt 10`
b. `ralph2 <-- "Hello to you!"`
c. `3ralph <- "Hello to you!"`
d. `ralph4 <- "Hello to you!`
e. `ralph5 <- date()`

Problem 2.9

Here are a few characters: `.  , ; _ - ^ [space] ( )`

- Which of those characters can be used in the name of an R object?
- Which of those characters can be used in a quoted character string?

Problem 2.10

These questions should be easy to answer if you use the appropriate commands to load, view, or get documentation on the datasets.

- How many variables are there in `CountryData`?
- What does the variable `tfat` measure in the `NCHS` data table? (in package `DataComputing`)
- How many cases are there in `WorldCities`?
- What's the third variable in `BabyNames`?
- What are the codes for the levels of the categorical variable `party` in the `RegisteredVoters` data table, and what does each code stand for?

3
R Command Patterns

Almost everyone who writes computer commands starts by copying and modifying existing commands. To do this, you need to be able to **read** command expressions. Once you can read, you will know enough to identify the patterns you need for any given task and consider what needs to be modified to suit your particular purpose.

As you read this chapter, you will likely get an inkling of what the commands used in examples are intended to do, but that's not what's important now. Instead, focus on

- Distinguishing between *functions* and arguments.
- Distinguishing between *data tables* and the *variables* that are contained in them.
- Recognizing when a function is being used.
- Identifying the arguments to a function.
- Observing how assignment allows values to be stored and referred to by name.
- Discerning when the output of one function becomes the input to another.

You are **not** expected at this point to be able to *write* R expressions. You'll have plenty of opportunity to do that once you've learned to *read* and recognize the several different patterns used in R data wrangling and visualization commands.

3.1 Language Patterns and Syntax

In a human language like English or Chinese, *syntax* is the arrangement of words and phrases to create well-formed sentences. For example, "Four horses pulled the king's carriage," combines noun phrases ("Four horses", "the king's carriage") with a verb.

Consider this pair of English language sentence patterns, a statement and a question:

1. I did go to school .
2. Did I go to school ?

The content of each of the boxes can be replaced by an equivalent object.

- I can be replaced with "you", "we", "he", "Janice", "the President", and so on.
- go to can be replaced with "stay in", "attend", "drive to", "see", ...
- school can be replaced with "the lake", "the movie", "Fred's parents' house", ...

Such replacements produce sentences like these:

- Did we stay in Fred's parents' house?
- Janice did drive to the lake.
- Did the President see the movie?
- Did he attend school?

Each sentence expresses something different, but they all follow the same patterns.

R HAS A FEW PATTERNS THAT SUFFICE for many data wrangling and visualization tasks. Consider these patterns:

- `object_name` <- `function_name` (`arguments`)
  This is called *function application*. The output of the function will be stored under the name to the left of <-.
- `object_name` <-
  `Data_Table` %>%
  `function_name` (`arguments`)
  This is called *chaining syntax*.
- `object_name` <-
  `Data_Table` %>%
  `function_name` (`arguments`) %>%
  `function_name` (`arguments`)
  This is an extended form of chaining syntax. Such chains can be extended indefinitely.

3.2 *Five kinds of objects*

There are five kinds of objects that you will be working with extensively.

1. Data tables
2. Functions
3. Arguments
4. Variables
5. Constants

Just as it helps in English to know what's a noun and what's a verb, etc., by identifying each kind of object in an expression you'll have an easier time understanding what the expression does.

1. DATA TABLES contain tidy data. A data table comprises one or more variables. It's easy to distinguish functions from data tables and variables: the function name will always be followed immediately by an open parenthesis. Data tables appear at the start of a chain, just before the first %>%.

2. FUNCTIONS are the objects that transform an input into an output. They are easy to spot. Functions are used by applying them to arguments, so you will see an opening parenthesis right after the function name. The corresponding closing parenthesis comes after the arguments.

There are a few functions that have a different syntax, e.g. the simple mathematical functions which are given *between* two arguments, e.g. `3 + 2` or `7 * 8`. This is called *infix* notation and is meant to mimic traditional arithmetic notation. Some of the other infix functions you'll encounter are `%in%`, `==`, `>=`, and so on.

3. ARGUMENTS DESCRIBE THE DETAILS of what a function is to do. They appear between the parentheses that follow a function name. One important exception: data tables are typically presented as an input to a function using the chaining notation %>%.

Many functions take *named arguments* where the name of the argument is followed by a = sign and then the value of that argument. For instance, `by = x` gives `x` as the value of the argument named `by`.

4. VARIABLES ARE THE COLUMNS IN A DATA TABLE. In this book, they will *always* be in function arguments, that is, between parentheses.

5. CONSTANTS ARE SINGLE VALUES, most commonly a number or a character string. Character strings will always be in quotation marks, `"like this."` Numerals are the written form of numbers, for

instance `-42`, `1984`, `3.14159`. Sometimes you'll see numerals written in scientific notation, e.g. `6.0221413e+23` or `6.62606957e-34`.

To help distinguish *data tables* from the *variables* in them, these notes will use a simple naming convention.

- The names of data tables will start with a CAPITAL letter. For instance: `WorldCities`, `NCI60`, `BabyNames`, and so on.
- The names of variables within data tables will start with a **lowercase** letter. For instance: `latitude`, `country`, `population`, `date`, `sex`, `count`, `countryRegion`, `population_density`.

These are *conventions*, not rules enforced by the R language. As you create your own data tables and variables, it is up to you to follow the convention. And, apologies, but even in this book you will encounter data or variables that fail to follow this convention. With time, such situations will be identified and fixed. But they can't be fixed everywhere, since sometimes you rely on resources developed by people or institutions who don't follow the conventions.

EXAMPLE: CLASSIFYING OBJECTS IN AN EXPRESSION. Consider this command:

```
BabyNames %>%
  filter(name == "Arjun") %>%
  summarise(total = sum(count))
```

The statement involves a data table (hints: starts with a capital letter, not followed by a opening parenthesis), three functions (hint: look for the names followed by an opening parenthesis), a named argument `total` (hint: inside the parentheses and followed by a single equal sign). The string `"Arjun"` is a constant. The remaining names — `name` and `count` — are variables. (Hint: they are involved in the arguments to functions).

The `filter()` function is being given the argument `name == "Arjun"`. This can be confusing. Although == somewhat resembles =, the single = is always just punctuation in a named argument. The double == is a function — one of those few infix functions that don't involve parentheses.

By the way, the overall effect of the command is to calculate the total number of individuals named Arjun represented in the `BabyNames` data. The result of the calculation is 5,578. Try it yourself!

NOTE TO EXPERIENCED PROGRAMMERS. You may be wondering where common programming constructs like looping and conditional flow, indexing, lists, and function definition fit in with this book. This book uses a couple of domain-specific, sub-languages, particularly `dplyr` and `ggplot2`. A "sub-language" is a part of a computer language that can be used almost like a language of it's own. The functions in `dplyr` and `ggplot2` already contain within them those programming constructs, so there is no need to use them explicitly. This is analogous to driving a car. The sub-language is the use of

the steering wheel, brake, and accelerator. You can use these to accomplish your task without having to know about how the engine or suspension work.

3.3 *Example Expressions*

These are expressions that you will use frequently, each written in the chaining style. To remind you, `BabyNames` is a data table.

Expressions that give a quick glance at a data table

These functions are generally used interactively, in the R console, to help you when constructing expressions for data wrangling or visualization.

```
BabyNames %>% nrow()     # how many cases in the table
```

```
[1] 1792091
```

```
BabyNames %>% names()    # the names of the variables
```

```
[1] "name"  "sex"   "count" "year"
```

```
BabyNames %>% head(3)    # the first three cases
```

name	sex	count	year
Mary	F	7065	1880
Anna	F	2604	1880
Emma	F	2003	1880

```
BabyNames %>% str()      # another view of the first few cases
```

```
'data.frame':   1792091 obs. of  4 variables:
 $ name : chr  "Mary" "Anna" "Emma" "Elizabeth" ...
 $ sex  : chr  "F" "F" "F" "F" ...
 $ count: int  7065 2604 2003 1939 1746 1578 1472 1414 1320 1288 ...
 $ year : int  1880 1880 1880 1880 1880 1880 1880 1880 1880 1880 ...
```

```
BabyNames %>% glimpse() # just like str()
```

```
Observations: 1792091
Variables:
$ name  (chr) "Mary", "Anna", "Emma", "E...
$ sex   (chr) "F", "F", "F", "F", "F", "...
$ count (int) 7065, 2604, 2003, 1939, 17...
$ year  (int) 1880, 1880, 1880, 1880, 18...
```

Modifying a data table

Sometimes, you will want to modify a data table and store the result under the original name. To illustrate:

```
BabyNames <-
  BabyNames %>%
  filter(year == 1998)
```

Because assignment is being used to capture the value being created by `filter()`, it might look like nothing is being done. But, behind the scenes, `BabyNames` has changed.

```
BabyNames %>% nrow()
```

```
[1] 27890
```

```
BabyNames %>% head(4)
```

Of course, it's not necessarily to use the original name to store the result of the modification. You can use any name that you think appropriate.

name	sex	count	year
Emily	F	26177	1998
Hannah	F	21368	1998
Samantha	F	20191	1998
Ashley	F	19868	1998

Table 3.1: The first 4 rows in the data table produced by filtering `BabyNames` to include only those babies born in 1998.

Named arguments and functions in arguments

The previous example, `BabyNames %>% head(4)` involved an argument to a function; `head()` is given the number 4 to specify how many cases to show. Sometimes the arguments will be the name of a variable or a function applied to a variable. Here's an example:

```
BabyNames %>%
  group_by(sex) %>%
  summarise(total = sum(count))
```

sex	total
F	1765766
M	1910081

Taking apart the above expression, you can see three functions, `group_by()`, `summarise()` and `sum()`. The name of functions is always followed by an open parenthesis. Inside those parentheses are the arguments. The argument to `group_by()` is `sex`, a variable. How do you know? It's evidently not a data table — it's not capitalized. It's neither a character string nor a numerical constant. And it's not a function — `sex` isn't followed by an opening parenthesis. By the process of elimination, this suggests that `sex` is a variable. The argument to `summarise()` is the expression `total = sum(count)`.

The argument to `summarise()` is in named-argument form. Note that `sum()` is a function. You can tell this from the expression: `sum` is followed immediately by an open parenthesis.

The `group_by()` and `summarise()` functions will be described in Chapter 4. They are called *data verbs* because they each take a data table as an input and produce a data table as output.

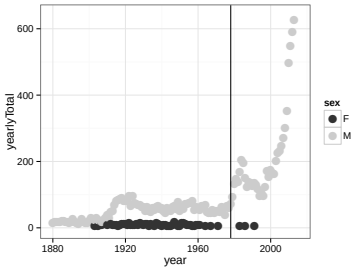


Figure 3.1: The number of babies born each year given the name "Prince".

EXAMPLE: A GENUINE DATA WRANGLING COMPUTATION. What do realistic data wrangling and graphics expressions look like? Figure 3.1 depicts the popularity of the name "Prince" as it varies over the years. (Don't worry for now about what the various functions are doing. You will get to that later.)

```
Princes <-
  BabyNames %>%
  filter(name == "Prince") %>%
  group_by(year, sex) %>%
  summarise(yearlyTotal = sum(count))
# Now graph it!
Princes %>%
  ggplot(aes(x = year, y = yearlyTotal)) +
  geom_point(aes(color = sex)) +
  geom_vline(xintercept = 1978) +
  ylim(0,640) + xlim(1880,2015)
```

Judging from Figure 3.1, the name "Prince" has been increasing in popularity over the last 40 years. One possible explanation is the popularity of the musician, Prince. The vertical line in the graph marks the year that Prince's first album was released: 1978.

3.4 *Constant objects*

Sometimes you will use assignment to store a constant. For instance:

```
data_file_name <- "tiny.cc/mosaic/engines.csv"
age_cutoff <- 21
```

Reminder: The two kinds of constants we will use are quoted character strings and numerals.

Naming constants in this way can help to make your data wrangling expressions more readable. For instance, by using `age_cutoff` in your expressions, you make it easily to update your expressions if you decide to change the age cutoff; just change the 21 to whatever the new value is to be.

3.5 *Chaining syntax*

If you have experience with R, you may never have seen the chaining operator %>% before this book.

Chapter 2 showed several different examples of function application, `function_name ( arguments )`, for instance `sqrt(2)` and `help(CPS85, package="mosaic")`

Chaining syntax is merely another form of function application. The following two patterns accomplish exactly the same thing:

`Data_Table` %>% `function_name` (`arguments`) ..... Chaining pattern.

`function_name` (`Data_Table` , `arguments`) ..... Non-chaining pattern.

In chaining syntax, the value on the left side of %>% becomes the *first argument* to the function on the right side. Note also that %>% is never at the start of a line — it should be placed at the end of any line which is to be followed by another step in a command sequence.

The chaining syntax is a help to the human reading and writing computer commands. Chaining makes more prominent the functions at each step. This is particularly helpful when there are many steps in a data wrangling or visualization task. To illustrate, here's a command sequence used previously in this chapter:

```
Princes <-
  BabyNames %>%
  filter(name == "Prince") %>%
  group_by(year, sex) %>%
  summarise(yearlyTotal = sum(count))
```

This expression can also be written in a non-chaining syntax, for instance:

```
Princes <-
  summarise(
    group_by(
      filter(BabyNames, name == "Prince"),
      year, sex),
    yearlyTotal = sum(count))
```

The chaining syntax takes advantage of the nature of data verbs. Each step in a data wrangling sequence takes a data table along with some other arguments as input and produces a data table as output. The chaining sequence brings the other arguments much closer to the data verb itself, so that you can see at a glance which arguments belong to which data verbs.

3.6 Exercises

Problem 3.1
Using the object name `fireplace`, write different expressions with enough context to be able to identify the name as belonging to

1. a data frame
2. a function
3. the name of a named argument
4. a variable

Write one expression for each of the above.

Problem 3.2
Explain why the following sentence is illegitimate:

```
Result <- %>% filter(BabyNames, name=="Prince")
```

Problem 3.3
What's wrong with this statement?

```
help(NHANES, package <- "NHANES")
```

Problem 3.4
Consider these R expressions. (You don't have to know what the various functions do to solve this problem.)

```
Princes <-
  BabyNames %>%
  filter(name == "Prince") %>%
  group_by(year, sex) %>%
  summarise(yearlyTotal = sum(count))
# Now graph it!
Princes %>%
  ggplot(aes(x = year, y = yearlyTotal)) +
  geom_point(aes(color = sex)) +
  geom_vline(xintercept = 1978)
```

There are several kinds of named objects in the above expressions.

a. function name
b. data table name
c. variable name
d. name of a named argument

Using the naming convention and position rules, identify what kind of object each of the following name is used for. That is, assign one of the types (a) through (d) to each name.

1) `BabyNames` 2) `filter` 3) `name` 4) `==`
2) `group_by` 6) `year` 7) `sex` 8) `summarise`
3) `yearlyTotal` 10) `sum` 11) `count` 12) `ggplot`
4) `aes` 14) `x` 15) `y` 16) `geom_point`
5) `color` 18) `geom_vline` 19) `xintercept`

Problem 3.5
There are several small, example data tables in the `ggplot2` package. Look at the `msleep` data table by using the `View()` function with the name of the object as an argument.

- What is the meaning of the `brainwt` variable?
- How many cases are there?
- What is the real-world meaning of a case?
- What are the levels of the `vore` variable?

Problem 3.6
The data verb functions all take a data table as their first argument and return a data table as their output. The chaining syntax lets the output of one function become the input to the following function, so you don't have to repeat the name of the data frame. An alternative syntax is to assign the output of one function to a named object, then use the object as the first argument to the next function in the computation.

Each of these statements, but one, will accomplish the same calculation. Identify the statement that does not match the others.

a)
```
BabyNames %>%
  group_by( year, sex ) %>%
  summarise( totalBirths=sum(count))
```

b)
```
group_by( BabyNames, year, sex) %>%
  summarise( totalBirths=sum(count) )
```

c)
```
group_by( BabyNames, year, sex ) %>%
  summarise( totalBirths=mean(count) )
```

d)
```
Tmp <- group_by(BabyNames, year, sex)
summarise( Tmp, totalBirths=sum(count) )
```

Problem 3.7
Which characters can be used in an object name?

Problem 3.8
The `date()` function returns an indication of the current time and date.

- What arguments does `date()` take? Use `help()` to find out.
- What *kind* of object is the result from `date()`.

4
Files and Documents

In your work with data, you will be using and creating computer files of various sorts. Of particular importance are three basic roles for files:

1. storing data tables
2. listing R instructions
3. writing reports with narrative, graphics and conclusions

Different file types are used for each of these roles. These types can be referred to by the filename extension of the files.

Table 4.1: Common file types and their uses

File Role	Common file types	Software
Data storage	`.csv`, `.xlsx`, etc.	Spreadsheets
R instructions	`.Rmd`	Text editor, compiler
End reports	`.html`, `.pdf`, `.docx`, etc.	web browser, word processor

Filename Extensions

The *filename extension* is one or more letters following the last period in the name: `.xlsx`, `.docx`, `.html`, `.mpeg`. (Other widely used extensions are `.pdf`, `.zip`, `.png`.) These letters indicate the format of the file and often set which program will be used to open the file: a spreadsheet, a word processor, a browser, or a music player in the examples. Or, in plainer language, the filename extension tells you the *kind* of file.

FILENAME EXTENSION: The letters following the last period in a file name. This suffix indicates the file type, that is, what software to use to interpret the file.

If you are looking at files through RStudio, the filename extension will always be displayed. If you are using your your own computer's file browser, your system may have been set up to hide the extension.

When referring to files within R statements or an Rmd file, you must **always** include the filename extension.

Paths

A filename is analogous to a person's first name. Just as first names are unique within a nuclear family, so filenames must be unique within a folder or directory.

You are probably used to seeing folders and the files they contain organized on your computer as in Figure 4.1. The *file path* is the set of successive folders that bring you to the file.

There is a standard format for file paths. An example:

```
/Users/kaplan/Downloads/0021_001.pdf
```

Here the filename is `0021_001`, the filename extension is `.pdf`, and the file itself is in the `Downloads` folder contained in the `kaplan` folder, which is in turn contained in the `Users` folder. The starting `/` means "on this computer".

The R `file.choose()` — which should be used **only in the console**, not in an Rmd file — brings up an interactive file browser. You can select a file with the browser. The returned value will be a quoted character string with the path name.

```
file.choose() # then select a file
```

```
[1] "/Users/kaplan/Downloads/0021_001.pdf"
```

In RStudio, the *Files* tab will display the path near the top. In Figure 4.2, the ten files lised are all in the same folder, whose path ends with `\ DCF-2014/ReOrganizedJune10/CourseNotes/DataOrganization`.

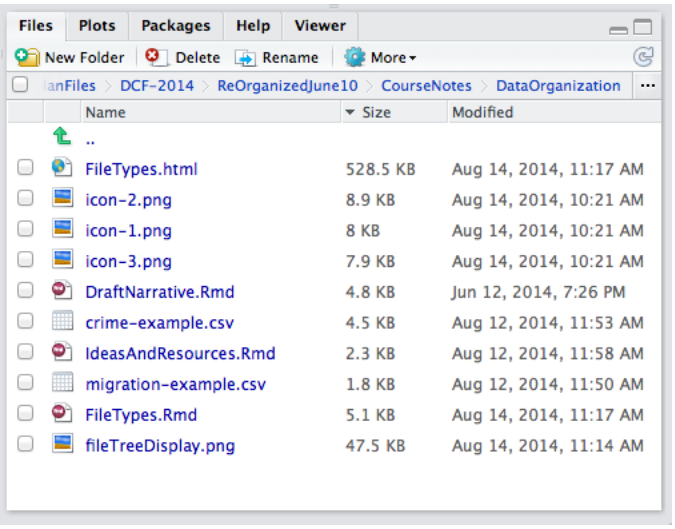


Figure 4.2: Filenames and their file path shown in the Files tab in RStudio. Only part of the path is shown: the folders closest to the files.

To run on with the family metaphor ... You are identified within a household by your first name. The path would tell you which specific nuclear family you belong to, perhaps in the form of your address, like this: *USA/Saint Paul/55105/703 Lincoln Avenue*.

FILE PATH: Information that specifies the location of a file in your file system.

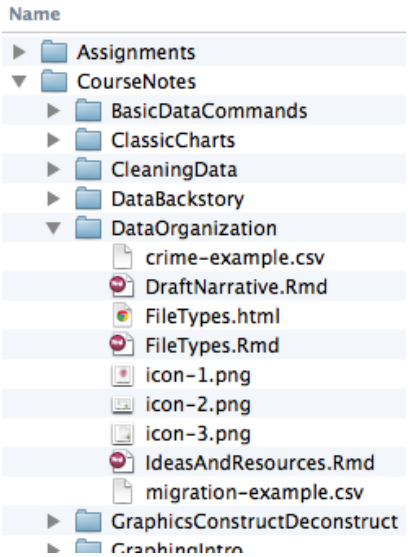


Figure 4.1: Folders contained within folders, as shown by a file browser on Apple OS-X.

QUOTES OR NOT FOR A FILE PATH? When you are referring to a file path in R, it will always be in quotation marks; it's a character

string. Other software such as web browsers don't use quotes when specifying a path.

URLs

You have probably noticed URLs in the locator window near the top of your browser. In Figure 4.3, the URL is:
`http://tiny.cc/dcf/index.html`.

A URL includes in its path name the location of the server on which the file is stored (e.g. `tiny.cc`) followed by the path to the file on that surver. Here, the path is `/dcf` and the file is `index.html`.

You will sometimes need to copy URLs into your work in R, to access a dataset, to make a link to some reference, etc. Remember to copy the entire URL, including the `http://` part if it is there.

Some common filename extensions for the sort of web resources you will be using:

- `.png` for pictures
- `.jpg` or `.jpeg` for photographs
- `.csv` or `.Rdata` for data files
- `.Rmd` for the human editable text of a document
- `.html` for web pages themselves

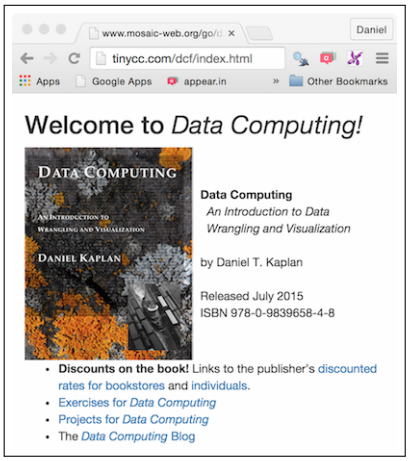


Figure 4.3: A browser directed to the URL `http://tiny.cc/dcf/index.html` .

Data Files

Data tables are often stored individually as files. There are many formats for data files. Among the most common is the `.csv` spreadsheet format, popular because reading it is a standard feature of many data analysis packages (including R).

If you use R extensively, you will also encounter data in `.Rda` (or `.Rdata`), an efficient format for storing data and other information specifically for R. When you get data from an R package, like this:

```
data(CPS85, package="mosaicData")
```

you are in fact reading in an `.Rda` file associated with the package.

There are many other formats for files containing data tables or the information needed to put the contents in data-table format. These are discussed in Chapter 15. Increasingly, data are accessed through *database* systems. In such a case, rather than reading the database as a whole, you make "queries" to access specific data you need.

Files containing data tables are often distributed via the web. These files are not any different than files on your own computer.

DATABASE: A computer system used to store, update, and access data. Relational data bases consist of data tables.

You access such files through a Uniform Resource Locator, better known as a *URL*. For instance, on the *Data Computing* web site, there are a number of `.csv` files. You access them from R with commands like this:

URL: A name for a specific resource, such as a file, on the Internet.

```
Engines <- read.file("http://tiny.cc/mosaic/engines.csv")
Engines
```

Engine	mass	ncylinder	strokes	displacement	bore	stroke	BHP	RPM
Webra Speedy	0.14	1	2	1.80	13.50	12.50	0.45	22000
Motori Cipolla	0.15	1	2	2.50	15.00	14.00	1.00	26000
Webra Speed 20	0.25	1	2	3.40	16.50	16.00	0.78	22000
			... and so on for 39 rows					

So far as accessing data is concerned, there's nothing fundamentally different in reading a file from a URL than reading a file on your own computer.

Documenting your work with Rmd files

The purpose of data wrangling and visualization is communication: condensing and presenting data in a form that conveys information. An important part of communication is documentation and reporting.

Writing is not a linear process. Ideas are presented, revised or abandoned, corrected, re-focussed, and re-arranged. Data-oriented technical reports tie together narrative, graphics and summary tables and are based on potentially complex computer commands. There is an interplay between the computer commands and the narrative. Results from the computer may drive reconsideration of the narrative. Gaps in the narrative may point to shortcomings or omissions in the computer commands. And, always, there is the possibility of errors in writing commands and the need to document commands so that they can be checked and corrected. As well, data are commonly updated, corrected or extended.

The familiar practice of cutting and pasting from the computer console into a word processor does not address these features of technical reports. Cutting and pasting makes it hard to revise or update a report; you've got to cut out the old and paste in the new, figuring out for yourself which is which. This introduces the likelihood of error. And, there's nothing to document the linkages between the computer commands and the word-processed document.

An important concept in data-driven reporting is "reproducibility." The idea is to be able to reproduce your entire document without any manual intervention, and, more important, to be easily able to

generate a new report in response to changes in data or revisions in computer commands. In other words, reproducible reports contain all the information needed to generate a new report. Common document formats such as `.pdf`, `.docx`, or `.html` do not offer support for reproducibility.

In R, reproducible reporting is provided by the `.Rmd` file format and related software. An `.Rmd` file integrates computer commands into the narrative so that, for instance, graphics are produced by the commands rather than being inserted from another source.

YOU CREATE AND MODIFY REPRODUCIBLE REPORTS within RStudio's text *editor*. They contain ordinary text and punctuation: no formatting, color, images, etc. Instead, you use the `.Rmd` file to describe, using ordinary characters, both what you want the eventual format to look like and what R commands you want the computer to carry out in generating the report.

EDITOR: A program for constructing pure-text documents, such as R instructions and Rmd files

Figure 4.4 shows a simple `.Rmd` document, something you might write. The figure also shows the `.html` file that is the result of compiling the `.Rmd`.

```
# My First Document

The text here will be **compiled** into a prettily formatted
document.  This process allows you to:

* Explain and record your work
* Construct and store the R instructions needed for your task.
* Produce formatting appropriate for and end report.

Most important, the compilation process will execute your R
instructions and include the results in the document.

## Example calculation

```{r}
library(DataComputing)
BabyNames %>%
  group_by(year) %>%
  summarise(total_births=sum(count)) %>%
  ggplot( aes(x=year, y=total_births)) +
  geom_line()
```
```

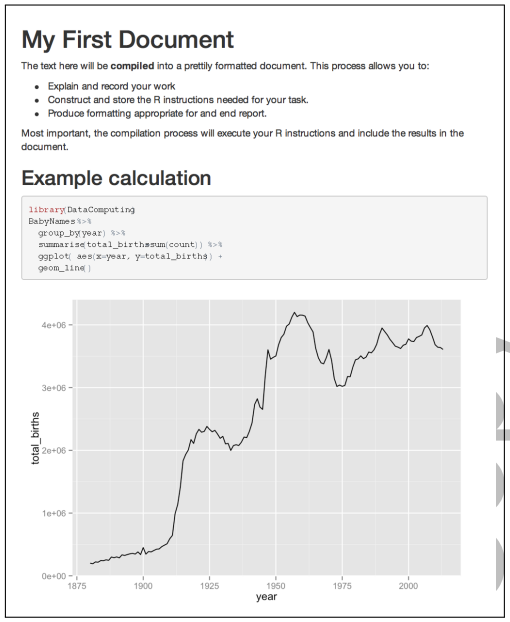


Figure 4.4: The `.Rmd` file on the left consists of text. It is automatically compiled to the formatted `.html` document, with the results of calculations, shown on the right.

COMPILE: The process of a computer translating a file from one format to another.

The Write/Compile Cycle

After you edit an `.Rmd` file, you *compile* the report into a reader-friendly document format such as `.pdf`, `.docx`, or `.html` that can easily be printed or displayed on the report-reader's computer. You

never edit the reader-friendly document; it's created automatically from your `.Rmd` instructions. When you want to update or modify or correct the end report, you edit the `.Rmd` file.

It's a good strategy to compile your `.Rmd` frequently. Start with a small, simple document. Then add a bit more to it: one paragraph or one "chunk" (see below). Compile again. If something goes wrong, you will have a good idea of where the problem lies. Go back and fix things. Make small changes, compile, see if it worked. Repeat. Figure 4.5 shows several steps in such an editing cycle.

It's impossible to avoid errors; even professionals make them. Instead, adopt a process that let's you identify errors quickly so that you can fix them before moving on. The shorter the write/compile cycle, the easier it will be to know when you have erred.

Interactive commands such as `file.choose()` or `scatterGraphHelper()` cannot be used in an an `.Rmd` file; there's no opportunity for interaction in the compilation process. Instead, use the interactive command in the console, get the result, and paste that result into your `.Rmd` file.

Command "Chunks"

The R commands in an `.Rmd` file go into *chunk*s, a range of lines in the documents that are delimited in a special way so that they will be executed as part of the `.Rmd` → `.html` compilation process.[1] The opening delimiter is ```` ```{r} ````. The closing delimiter is simply ```` ``` ````. For example:

````
```{r}
Saltpeter <- read.file("http://tiny.cc/DCF/saltpeter.csv")
```
````

CHUNK: A region in an `.Rmd` file that contains R statements to be executed when the `.Rmd` is compiled.

[1] Exactly the same applies when compiling to `.pdf` of `.docx`.

MOST .RMD FILES WILL DRAW ON A LIBRARY that needs to be loaded into the R session. When you compile `.Rmd` → `.html`, R starts a brand new session that is, initially, empty and with no libraries loaded. When the compilation is complete, that session evaporates, leaving as its only residue the `.html` result.

Both the opening and closing delimiters for a chunk are "back-quotes," a quote character that goes from upper-left to bottom-right. On many keyboards, back-quote is on the same key as tilde (~), like this:

Often, the first chunk in your document will be an instruction to load one or more libraries. Since this will be in just about every `.Rmd` document, it can be called the *boilerplate* chunk. It looks like this: A boilerplate chunk goes at the start of the document. It loads the libraries that the following commands will use.

BOILERPLATE: A standardized piece of text intended for re-use in many documents.

````
```{r include=FALSE}
library(DataComputing)
# and any others that you need, e.g.
library(mosaic)
```
````

The `include=FALSE` chunk argument helps to prettify the document: it is an instruction not to show the contents of the chunk in the output file.

Step 1. Start with a short and simple document.

```
# Saltpeter Logistics

by Abigail Adams

This report compares
production of saltpeter
with the "French method"
to the "Swiss method."
```

Compile to HTML

Step 2. Add the boilerplace chunk and a chunk to your start your wrangling. Test that it works. If not, go back and fix it.

````
# Saltpeter Logistics

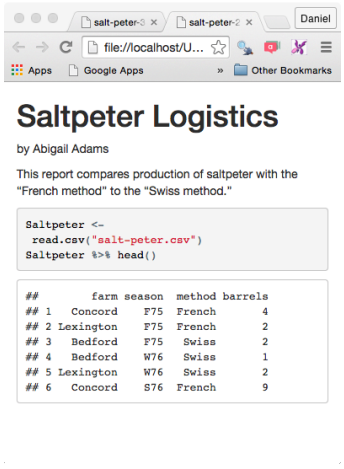
by Abigail Adams

```{r include=FALSE}
library(DataComputing)
```

This report compares
production of saltpeter
with the "French method"
to the "Swiss method."

```{r}
Saltpeter <-
 read.file("saltpeter.csv")
Saltpeter %>% head()
```
````

Compile to HTML



Step 3. Add more to the wrangling chunk. Test that it works.

````
# Saltpeter Logistics

by Abigail Adams

```{r include=FALSE}
library(DataComputing)
```

This report compares
production of saltpeter
with the "French method"
to the "Swiss method."

```{r}
Saltpeter <-
 read.file("saltpeter.csv")
Saltpeter %>%
 group_by(season,method) %>%
 summarise(
  prod = sum(barrels) / n() )
```
````

Compile to HTML

Figure 4.5: Three steps in the write/compile cycle. At each step, the `.Rmd` file (shown in `monospace font`) is compiled into the `.html` format shown underneath.

DISTRIBUTING THE .RMD FILE IS HELPFUL when communicating with a technically savvy audience. A nice strategy for getting the benefits of both the easily-readable .html format and the original .Rmd file is to include the .Rmd file *inside* the .html, much as you might attach a file to an email message. The DataComputing package provides a way to do this easily by including the following chunk in the .Rmd file.

The document template Data Computing simple includes the source document by default.

```
```{r echo=FALSE, results="asis"}
cat("Source Documents: ")
DataComputing::includeSourceDocuments()
```
```

This chunk will embed the source file into your .html. The originating .Rmd file can be extracted by clicking on a link that is contained within the .html file.

4.1 Exercises

Problem 4.1
Markdown provides a simple way to produce section headers and sub-headers, italic and bold text, `monospaced` fonts suitable for computer commands, and even web links. A reference is available in RStudio at the menu HELP/MARKDOWN QUICK REFERENCE.

For each of the following, say how it will be rendered when the Markdown is rendered to HTML. (Hint: You can figure it out by reading the documentation, or you can put the text into an Rmd document and compile it!)

```
*one*

**two**

* three

# Four

`five`

## Six

[seven](http://tiny.cc/dcf/index.html)
```

Problem 4.2
What's wrong with the markup for each of these five chunks:

(a)

```
'''{r}
9+7
'''
```

(b)

```
"""(r)
9+7
"""
```

(c)

```
```{r}
9+7
``
```

(d)

```
. ```{r} 9+7```
```

(e)

```
```{r}
9+7
````
```

**Problem 4.3**
Treat the following lines as an `.Rmd` file which will be compiled to HTML.

```
An Introduction

Arithmetic is *easy*! For instance
```{r}
3 + 2
```
```

Using paper and pencil, sketch out what the HTML document will look like when viewed in a web browser.

**Problem 4.4**
Here is a short list of names:

1. `DataComputing.org`
2. `ahab/whale.Rmd`
3. `ptth://world-bank.org`
4. `http://world-bank.org`
5. `//world-bank.org/index.html`
6. `world-bank.org/index.html`

For each, say whether the name is in the allowed form for a possible URL, a possible file, neither, or both.

**Problem 4.5**
From the RStudio console, load the `DataComputing` package like this:

```
library(DataComputing)
```

Once this is done,

1. Open a new file using the File/New File/R Markdown ... menu item. Select "From Template" and then choose the *DataComputing simple* template.
2. Save the text that appears in the editor tab in a file named `Birds.Rmd`
3. Compile the `Birds.Rmd` file to HTML to verify that the template is working.
4. Edit the `Birds.Rmd` file to include contents that will make the compiled HTML file appear like this:


```

5
Introduction to Data Graphics

Data graphics provide one of the most accessible, compelling, and expressive modes to investigate and depict patterns in data. This chapter presents examples of standard kinds of data graphics: what they are used for and how to read them. To start, you'll make simple examples of graphics using an interactive tool. Later, in Chapter 6, you'll see a unifying framework — a grammar — for describing and specifying graphics, so that you can create custom graphics types that support displaying data in a purposeful way.

5.1 Common Kinds of Graphs

There are many different genres of data graphics, and many different variations on each genre. Here are some commonly encountered kinds.

- **Scatterplots** showing relationships between two or more variables.
- **Displays of distribution**, such as histograms.
- **Bar charts**, comparing values of a single variable across groups.
- **Maps**, showing how a variable relates to geography.
- **Network diagrams**, showing how entities are connected to one another.

Scatter plots

The main purpose of a scatter plot is to show the relationship between two variables across several or many cases. Most often, there is a Cartesian coordinate system in which the x-axis represents one variable and the y-axis the value of a second variable.

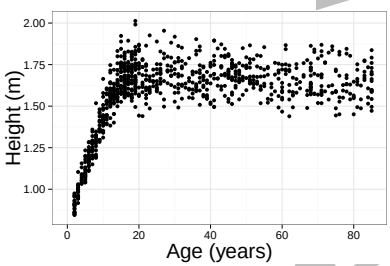


Figure 5.1: A scatter plot.

EXAMPLE: GROWING UP. The NCHS data table gives medical and morphometric measurements of individual people. The scatter plot in Figure 5.1 shows the relationship between two of the variables, height and age. Each dot is one case. The position of that dot signifies the value of the two variables for that case.

Scatterplots are useful for visualizing a simple relationship between two variables. For instance, you can see in the 5.1 the familiar pattern of growth in height from birth to the late teens.

Displays of Distribution

A histogram shows how many cases fall into given ranges of the variable. For instance, Figure 5.2 is a histogram of heights from NCHS. The most common height is about 1.65 m — that's the location of the tallest bar. Only a handful are taller than 2.0 m.

A simple alternative to a histogram is a *frequency polygon*. Frequency polygons let you break things up by other variables. Figure 5.3 shows the distribution of height for each sex, separately.

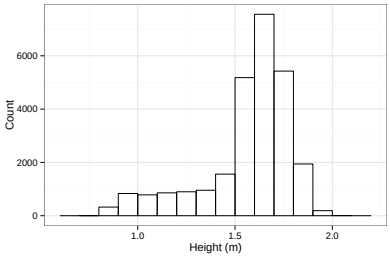


Figure 5.2: A histogram.

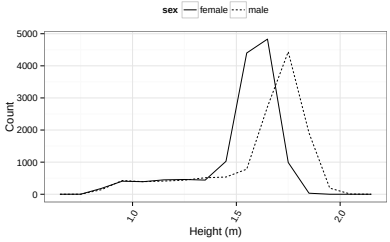


Figure 5.3: A frequency polygon

Bar Charts

The familiar bar chart is effective when the objective is to compare a few different quantities.

EXAMPLE: SMOKING AND DEATH. Based on the NCHS data, how likely is a person to have died during the follow-up period, based on their age and whether they smoke? It's easy to compare bars to their neighbors. From Figure 5.4, for instance, you can see that at each age, non-smokers were more likely to survive.

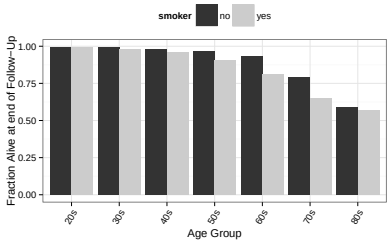


Figure 5.4: A bar chart

Maps

Using a map to display data geographically helps both to identify particular cases and to show spatial patterns and discrepancy. The map in Figure 5.5 shows oil production in each country. That is, the shading of each country represents the variable `oilProd` from `DataComputing::CountryData`. This sort of map, where the fill color of each region reflects the value of a variable, is sometimes called a *choropleth map*.

oil-map-1.pdf

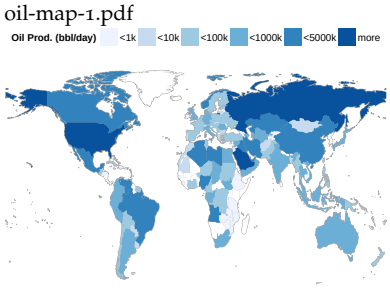


Figure 5.5: A choropleth map

Networks

A *network* is a set of connections, called *edges*, between elements, called *vertices*. A vertex corresponds to a case. The network describes which vertices are connected to other vertices.

The `DataComputing::NCI60` data set is about the genetics of cancer. The data set looks at more than 40,000 probes for the expression of genes, in each of 60 cancers. In the network here, a vertex is a given cell line. Every vertex is depicted as a dot. The dot's color and label gives the type of cancer involved. These are Ovarian, Colon, Central Nervous System, Melanoma, Renal, Breast, and Lung cancers. The edges between vertices show pairs of cell lines that had a strong correlation in gene expression.

The network shows that the melanoma cell lines (ME) are closely related to each other and not so much to other cell lines. The same is true for colon cancer cell lines (CO) and for central nervous system (CN) cell lines.

cancer-network-1.pdf

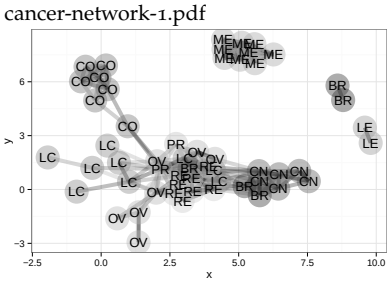


Figure 5.6: A network diagram

5.2 *Constructing Graphics Interactively*

There is a simple pattern to creating a data graphic:

1. Choose or create the glyph-ready data table that will be graphed.
2. Select the kind of graphic: scatterplot, bar chart, map, etc.
3. Decide which variables from the data table will be assigned to which roles in the graphic: *x*- and *y*-coordinates, bar lengths, colors, sizes, etc. This is called *mapping* a variable to a graphical attribute.

MAPPING: specifying which particular graphical attribute is to represent a variable. The word "mapping" is drawn from mathematics, an association between two sets of items, and has nothing to do with geographical maps.

Chapter 8 introduces R commands such as `ggplot()` for drawing graphics. In this chapter, you will use interactive programs to generate the graphics and the corresponding R commands. The commands can be pasted into an R chunk in an `.Rmd` file.[1]

[1] Never put the interactive commands into an `.Rmd` file. There is no one to interact with the session created when compiling. Instead, use the interactive commands in the console to generate the graphics commands, and paste the commands into an R chunk. Instructions for installing `DataComputing` and other packages are posted at `http://Data-Computing.org/` under "Software and Data."

The `DataComputing` package provides several interactive graphing functions. They are interactive in allowing you to use the mouse to specify which variables map to which graphical attributes. The functions are:

- `scatterGraphHelper()`
- `barGraphHelper()`
- `distributionGraphHelper()`
- `WorldMap()`
- `USMap()`

The first argument to each of these functions is a data table whose cases you want to display graphically. The map-making functions also require two more arguments: `key=` specifies the variable in the data table that identifies the country or state for each case. `fill=` specifies the variable to be used for shading each country.

Drawing networks will be introduced later.

Scatter Plots

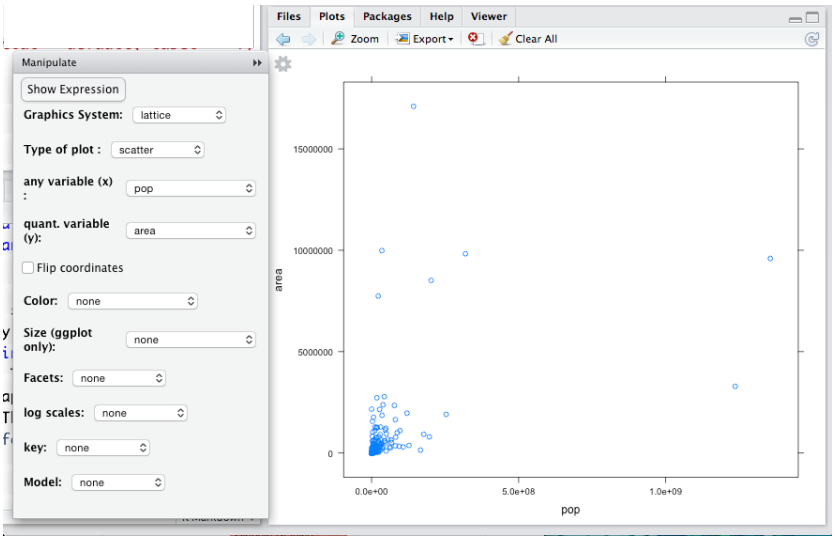


Figure 5.7: The interactive plot tab created using `scatterGraphHelper(CountryData)`. By default, the first two quantitative variables in a data table are mapped to the x- and y-axes. Change this using the drop-down menu to make the scatter plot of interest to you.

Consider the relationship between birth rate and death rate among the countries in `CountryData`. The variables `birth` and `death` that give these rates (in births per 1000 people per year).

An appropriate graphic modality is a scatter plot: birth rate against death rate. To make the graph, use the software appropriate for this modality, namely `scatterGraphHelper()`. As an argument, give the data table from which the variables are taken: `CountryData`.

```
scatterGraphHelper(CountryData)
```

That one simple command gives two essential details: what data table to use and what modality of graph to make. After giving that command, something like Figure 5.7 should appear in your "Plots" tab.

Notice three components of the plots tab:

1. A coordinate grid with dots.
2. A menu for mapping variables to attributes.
3. A small gear icon: 

If you don't see the menu on your system, click on the gear icon. If the menu runs off the bottom of the screen, make the "Plots" tab taller.

By default, the first two quantitative variables, `area` and `pop`, are being used to define the frame. This is not usually what you want.

You can use the `scatterGraphHelper()` menu (Figure 5.8) set the frame to be `death` *versus* `birth`. The overall pattern is U-shaped; both low and high birth rates are associated with high death rates, while birth rates in the middle tend to have lower death rates.

VERSUS: One against the other. The convention is y versus x.

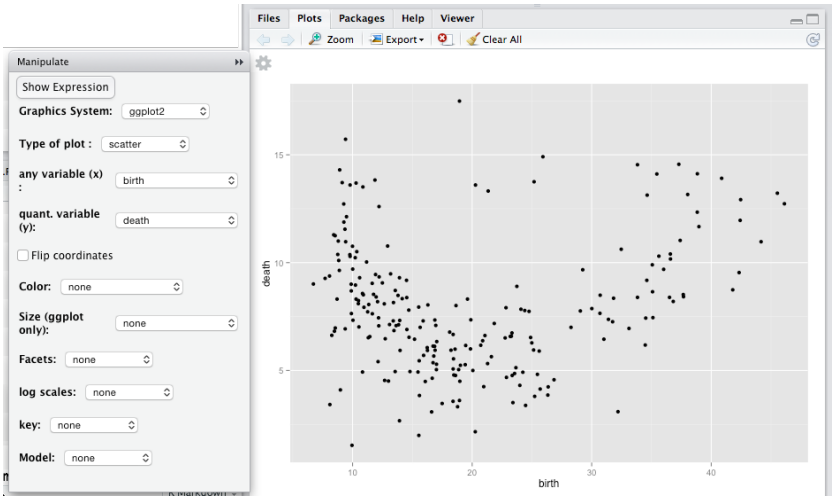
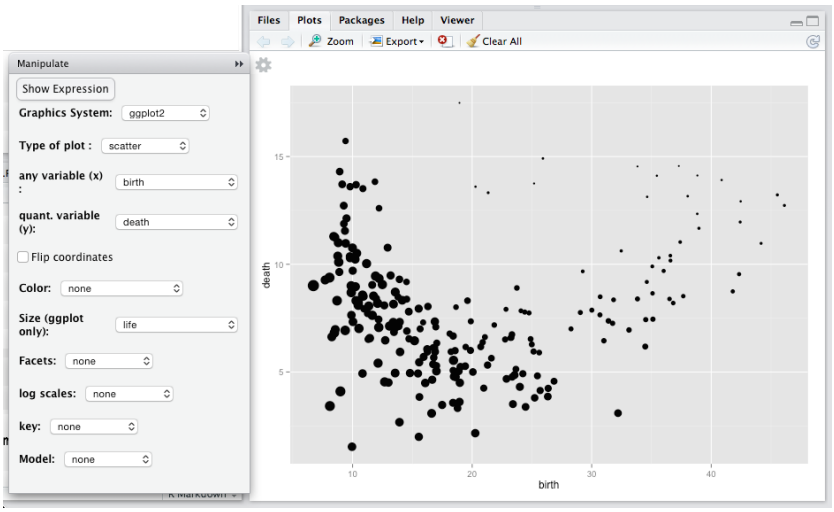


Figure 5.8: Using the `scatterGraphHelper()` menu to set the variables displayed on the x- and y-axes to be birth and rates.

The glyphs in scatterplots — dots here — can have graphical attributes besides from their position in the frame. Standard ones include fill color, shape, size, transparency, and border color. In Figure 5.9, life expectancy is mapped to size. You can see that, for countries with high life expectancy, the death rate is high when the birth rate is low. For countries with low life expectancies, the death rate is high when the birth rate is low.



That's because when life expectancy is high and birth rate is low, the population tends to be older. Older populations have higher death rates.

Figure 5.9: Graphical attributes such as size, shape, and color can be used to represent additional variables. Here, dot size reflects a country's life expectancy.

Distributions

Frequency polygons or histograms are appropriate for showing how the different values are distributed.

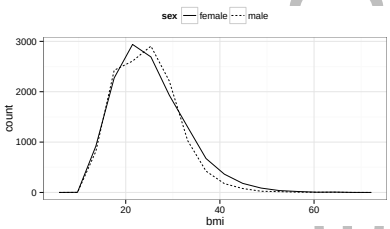


Figure 5.10: The distribution of body mass index shown with a frequency polygon separately for each sex.

```
DistributionGraphHelper(NCHS, format = "frequency polygon")
```

Consider, for instance, how body mass index varies across the subjects in the NCHS data. For a distribution, the measured quantity is mapped to the x-axis. The y-values are set by the number of cases at the corresponding x-value. In the `scatterGraphHelper()` menu shown in Figure 5.11, `sex` has been mapped to line type, producing the chart in Figure 5.10.

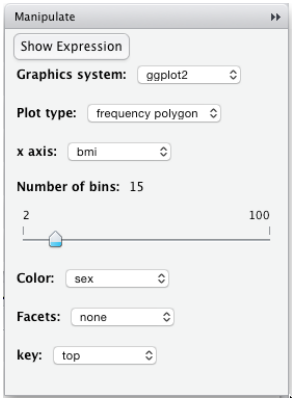


Figure 5.11: Setting the variable mappings for the frequency polygon plot in Figure 5.10 using the `scatterGraphHelper()` menu.

Bar Plots

Bar charts use a glyph whose *length* reflects the value to be presented. Depending on how variables are mapped to graphical attributes, the plots can tell different aspects of the story.

For instance, the individual ballot choices in the 2013 mayoral election in Minneapolis look like this:

Precinct	First	Second	Third	Ward
P-07	DON SAMUELS	BETSY HODGES	DAN COHEN	W-7
P-05	DON SAMUELS	BETSY HODGES	CHRISTOPHER CLARK	W-9
P-01	DON SAMUELS	BETSY HODGES	MARK ANDREW	W-12
P-03A	MARK ANDREW	undervote	undervote	W-10
P-02	BETSY HODGES	MARK ANDREW	DON SAMUELS	W-9
		... and so on for 80,101 rows		

You might be interested to make a bar chart of the number of first-choice votes that each candidate received. In this case, as is typical, a bit of data wrangling is called for to create glyph-ready data. For now, don't worry about the following commands, which will be introduced later.

```
FirstPlaceTally <-
  Minneapolis2013 %>%
  rename(candidate=First) %>%
  group_by(candidate) %>%
  summarise(total = n())
```

candidate	total
ALICIA K. BENNETT	351
BETSY HODGES	28935
BILL KAHN	97
BOB FINE	2094
CAM WINTON	7511

Table 5.1: First place vote tallys in the `Minneapolis2013` data

There were 37 candidates in the election (!), but most got only a small number of votes. The results can be displayed effectively with a bar chart.

```
barGraphHelper(FirstPlaceTally)
```

The chart shows at a glance that there are just a handful of major candidates. For this plot, the `total` variable was mapped to the y-axis, the candidate was mapped to the x-axis, and the candidates were ordered from lowest to highest total of votes. Look closely and you'll see that "undervote" (meaning no candidate was chosen) beat 29 of the candidates.

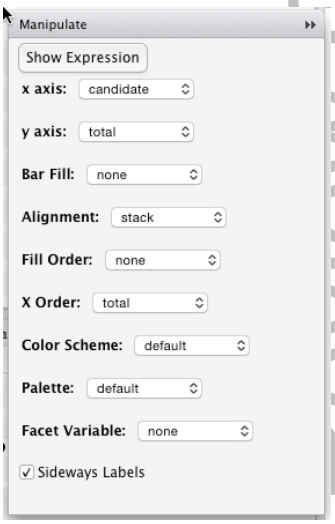


Figure 5.13: The mappings for the vote-tally bar chart in Figure 5.12

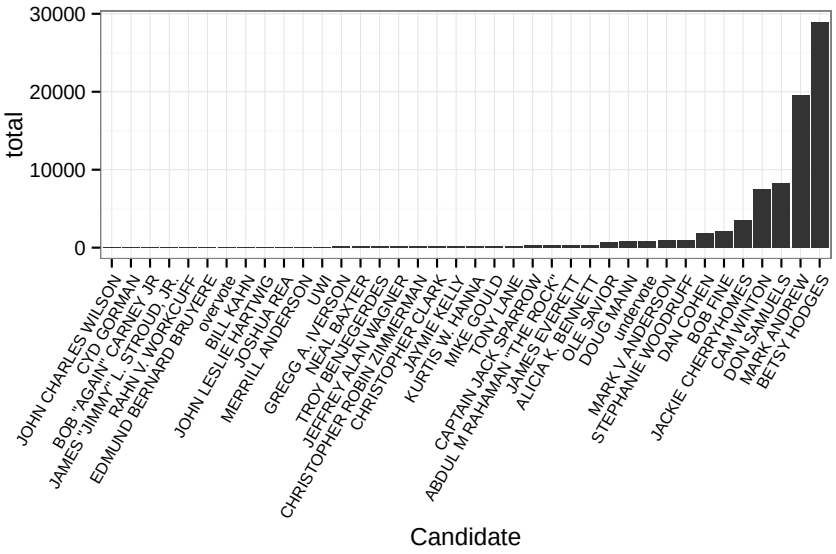


Figure 5.12: A bar chart showing the number of first-place votes given to each candidate.

Making Maps

Showing a variable in a geographical map requires two data tables:

1. A *shape file* giving latitude and longitude of points on the boundaries.

2. The data table for the variable that is to be plotted.

There are shape files for all sorts of geographic entities: countries, states, counties, precincts, and so on. To simplify things, the `DataComputing` package provides two functions: `WorldMap()` with country boundaries and `USMap()` with state boundaries in the US. The shape file is pre-set for these functions; you need only provide a data table with the name of countries (or states) and the variable to be plotted. Figure 5.14 shows how fertility varies from country to country.

```
CountryData %>%
  WorldMap(key="country",fill="fert")
```

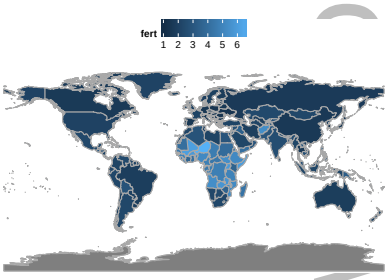
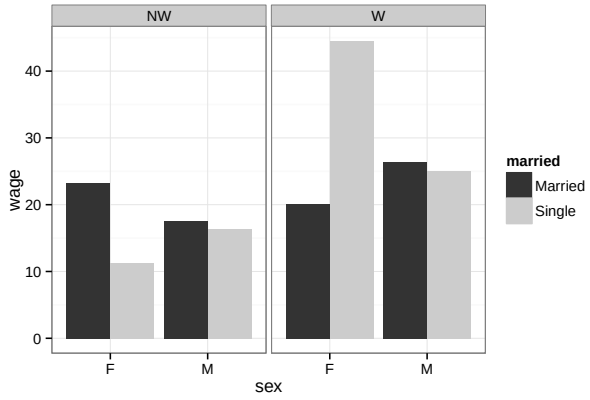


Figure 5.14: A choropleth map of fertility (children born per woman)

5.3 *Exercises*

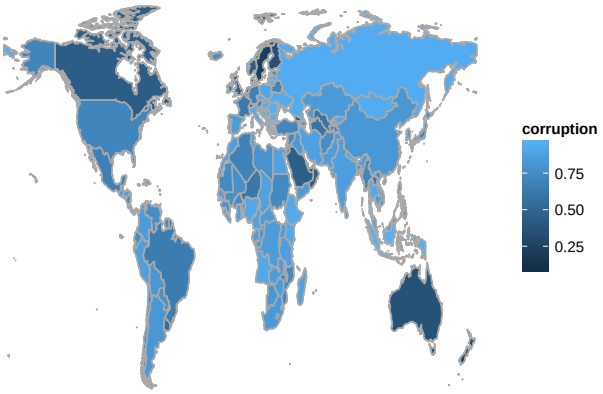
Problem 5.1
Consider this bar graph of the `CPS85` data in the `mosaicData` package:



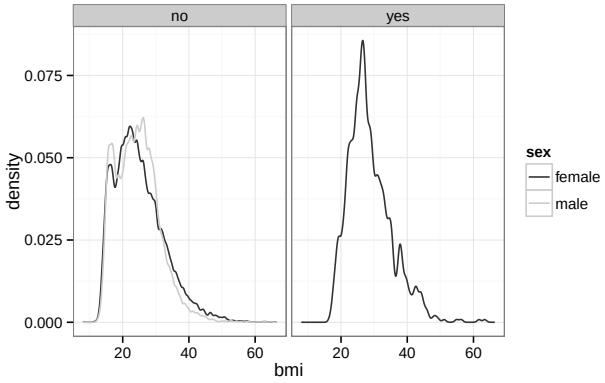
Use `barGraphHelper()` to reconstruct the graph. Start with these commands:

```
library(mosaicData)
library(DataComputing)
barGraphHelper(CPS85)
```

Problem 5.2
Make this map using data from `HappinessIndex` in the `DataComputing` package:

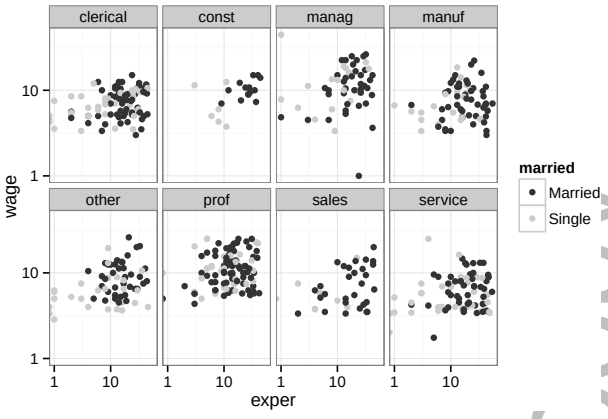


Problem 5.3
Make this graph from the `NCHS` data in the `DataComputing` package.



The "yes" and "no" in the gray bars refer to whether or not the person is pregnant.

Problem 5.4
Using the `CPS85` data table (from the `mosaicData` package) make this graphic:



6
Frames, Glyphs, and other Components of Graphics

Data graphics are built from parts. Chapter 5 showed the parts assembled together. This chapter looks at the parts individually.

Of course, a data table provides the basis for drawing a data graphic. The relationship between a data table and a graphic is simple: Each case in the data table becomes a mark in the graph. The designer of the graphic — you — chooses which variables the graphic will display and how each variable is to be represented graphically: position, size, color, and so on. The marks themselves are called *glyphs*. A data graphic has one glyph for each case in the data table.

Key Graphics Vocabulary

FRAME: The relationship between position and the data being plotted.

GLYPH: The basic graphical "unit" that represents one case. Other terms used include "mark" and "symbol." Variables set graphical attributes of the shape: size, color, shape, and so on. The location of the glyph — location is an important graphical attribute! — is set by the two variables defining the frame.

AESTHETIC: Any graphical attribute of a glyph: size, location, shape, color, etc.

SCALE: The relationship between the value of a variable and the graphical attribute to be displayed for that value.

GUIDE: An indication for the human viewer of the scale, that is, graphics how a variable encodes into its graphical attribute. Common guides are x- and y-axis tick marks and color keys.

6.1 *The Frame*

The frame of a graphic provides the space for drawing glyphs. But there is more to a frame than a blank canvas or piece of paper. The frame defines what position means. Most often, the frame is a rectangular region and position is described in terms of the familiar (x, y) Cartesian coordinate system. In creating a frame, you must decide which variable in your data will correspond to the x coordinate, and which to the y coordinate.

For instance, consider a dataset relevant to economic productivity. Table 6.1 gives per capita GDP for each country as well as some of the explanatory candidates: average educational level in the population, length of roadways per unit area, Internet use as a fraction of the population.

country	gdp	educ	roadways	net_users
Australia	44353.87	5.60	0.11	>60%
Ghana	3509.96	8.10	0.46	>0%
Mozambique	1140.04	5.00	0.04	>0%
	... and so on for 256 rows			

Table 6.1: Data relevant to economic performance. The complete table is available at `http://tiny.cc/dcf/table-6-2.csv`

You define a frame by selecting two variables from the glyph-

ready data table. For instance, Figure 6.1 shows a frame based on GDP and length of roadways. The frame provides the meaning to location in space.

6.2 *Glyphs*

The frame itself doesn't display any of the cases. Instead, the glyphs positioned in the frame represent the cases. There will be one glyph for each case in the data table.

The basic shape used in scatter plots is a simple glyph: a dot, a square, a triangle, an x, and so on. The following graph uses small dots. Since each case is a country, each dot represents one country.

In Figure 6.2 the glyphs are simple. Only position in the frame distinguishes one glyphs from another. The shape, size, etc. of all of the glyphs are identical. There's nothing about the glyph itself which identifies the country. It's possible to use a glyph with several attributes. Figure 6.3 location and label, mapping country name to the label.

But glyphs can have several properties. The aspects of each glyph that we can perceive are called *aesthetics*, or equivalently *graphical attributes*. The word *aesthetics* applied in the context of glyphs is not used in the modern sense. Nowadays, most people associate aesthetics with notions of beauty and artistic taste. The earlier meaning of the word, *properties relating to perception by the senses*, is the one intended when it comes to glyphs.

Location in the frame are the (x, y) aesthetics for a glyph, but other aesthetics can display variables in the data table. For instance, color could be used to show Internet use (as a fraction of the population), as in Figure 6.4. Another aesthetic is size. The size is fixed in 6.4; the same for every country. Figure 6.5 maps the average years of eduction onto the size aesthetic.

6.3 *Scales and Guides*

There are four aesthetics in Figure 6.5. Each of the four aethetics is set in correspondence with a variable; we say the variable is *mapped* to the aesthetic. Length of roadways is being mapped to horizontal position, GDP to vertical position, Internet connectivity to color, and educational attainment to size.

A *scale* is the relationship between a variable and the aesthetic to which it is mapped. For `roadways`, the scale says what value of the variable will correspond to position at the bottom of the frame, what value will correspond to the top of the frame, and where things fall inbetween.

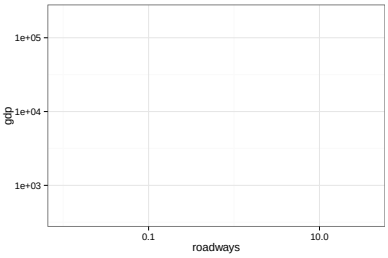


Figure 6.1: A graphics frame set by the `GDP` and `roadway` variables. No glyphs have been set in this frame.

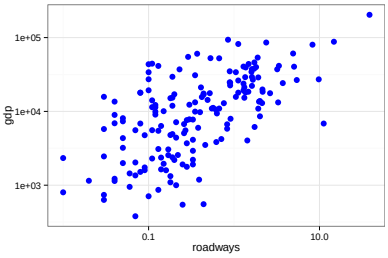


Figure 6.2: Using only position as the aesthetic for glyphs.

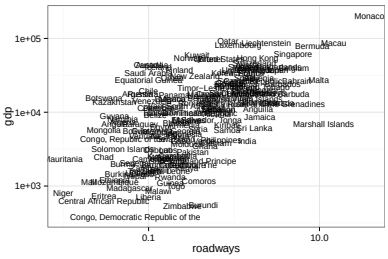


Figure 6.3: Using both location and label as aethetics.

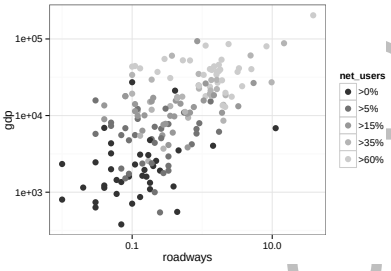


Figure 6.4: `net_users` mapped to color.

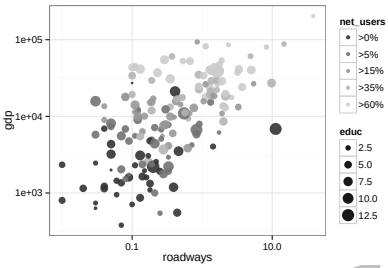


Figure 6.5: `net_users` mapped to color, `educ` mapped to size. Compare this graphic to Figure effig:facet-internet, which shows the same data using facets.

Not all scales are about position. For instance, in Figure 6.5, `net_users` is translated to color. Similarly, average educational attainment (in years) is translated to size: the middle-sized dot corresponds 7½ years of education.

Scales translate values into aesthetic properties. *Guide*s help to human reader to do the back translation. For position aesthetics, the most common sort of guide is the familiar axis with its tick marks and labels. But notice also the guide that tells how dot color corresponds to Internet connectivity. There's still another guide telling how dot size corresponds to education.

GUIDE: a display of a scale

6.4 *Facets*

Using multiple aesthetics such as shape, color, and size to display multiple variables can produce a confusing, hard-to-read graph. *Facets* provides a simple and effective alternative. Figure 6.6 uses facets to show different levels of Internet connectivity, providing a better view than Figure 6.5.

FACETS: Multiple side-by-side graphs used to display levels of a categorical variable.

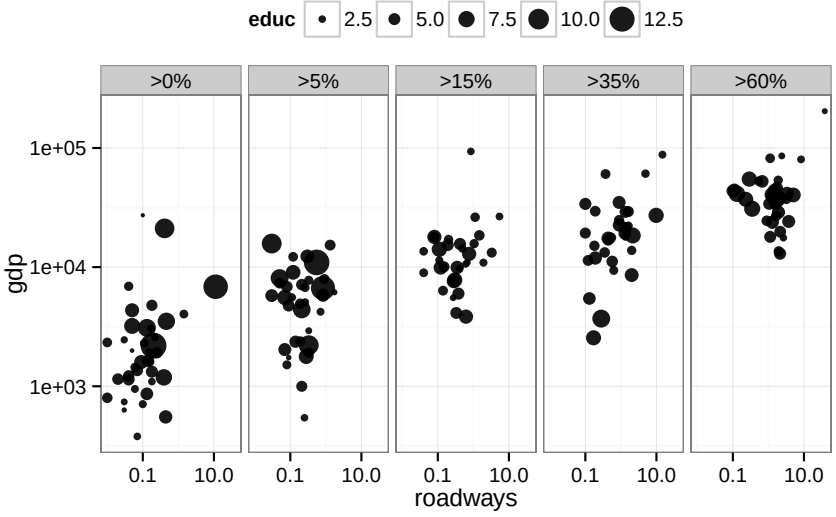


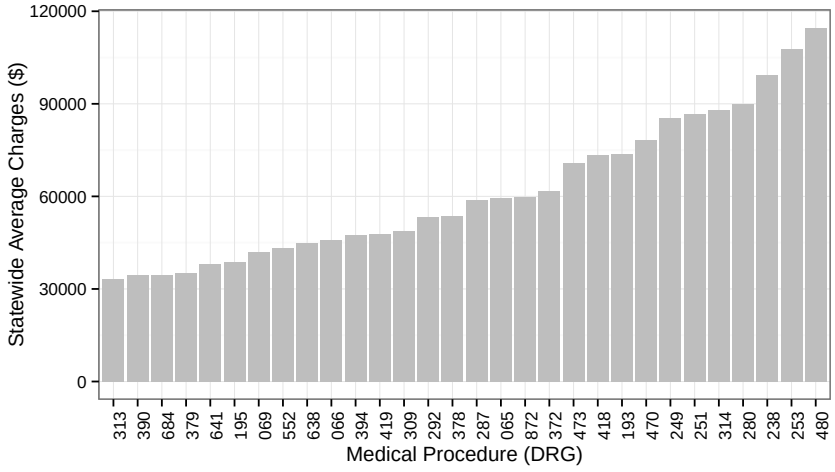
Figure 6.6: Using facets for different ranges of Internet connectivity.

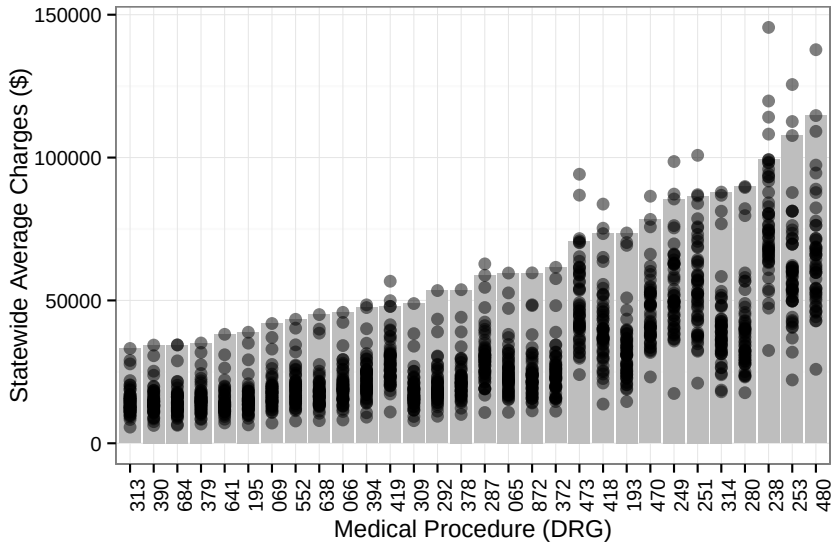
6.5 *Layers*

On occasion, data from more than one data table are graphed together. For instance, suppose you want a display of one state's hospital providers' charges for different medical procedures. The glyph-ready data table for New Jersey looks like Table 6.2. The glyph-ready table can be translated to a chart (Figure 6.7 (top)) using bars to give

a fair impression of the range in charges for different medical procedures in New Jersey.

How do the New Jersey charges compare to those in other states? Tables 6.2 and 6.3 provide relevant data. The two data tables, one for New Jersey and one for the whole country, can be plotted with different types of glyph: bars for New Jersey and dots for the whole country as in Figure 6.8.





With the context provided by the individual states, it's easy to see the charges in New Jersey are among the highest in the country for each medical procedure. (A description of each medical procedure number is given in the data table `DirectRecoveryGroups` in the `DataComputing` package.)

drg	stateProvider	mean_charge
313	NJ	33183.08
390	NJ	34361.27
684	NJ	34396.18
379	NJ	35091.47
641	NJ	38134.32
	... and so on for 100 rows	

Table 6.2: Glyph-ready data for the barplot layer in Figure 6.7

Figure 6.7: Average charges for medical procedures in New Jersey.

drg	stateProvider	mean_charge
039	ME	14480.66
039	KS	25859.60
039	FL	42636.68
039	DC	47369.88
039	MO	26128.30
039	TN	26417.38
057	GA	18508.69
057	SC	21741.92
057	AK	31858.55
057	UT	20247.73
	... and so on for 5,025 rows	

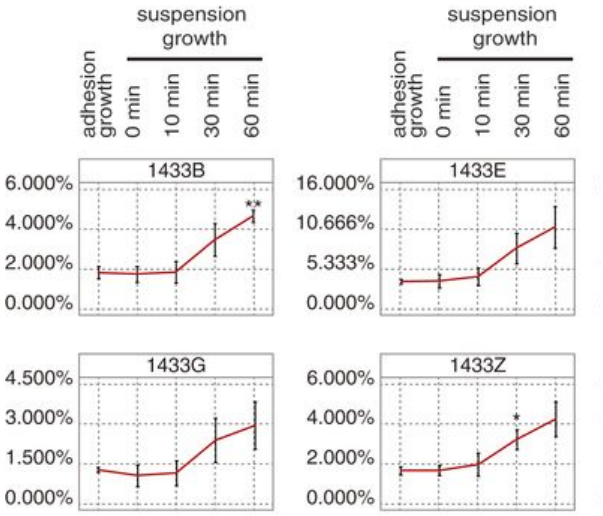
Table 6.3: Glyph-ready data table for the scatter-plot layer in Figure 6.7

Figure 6.8: Adding a second layer to provide a comparison of New Jersey to other states. Average charges for medical procedures in New Jersey.

6.6 *Exercises*

Problem 6.1

The following chart contains four facets. Each shows the amount of a substance in different conditions:

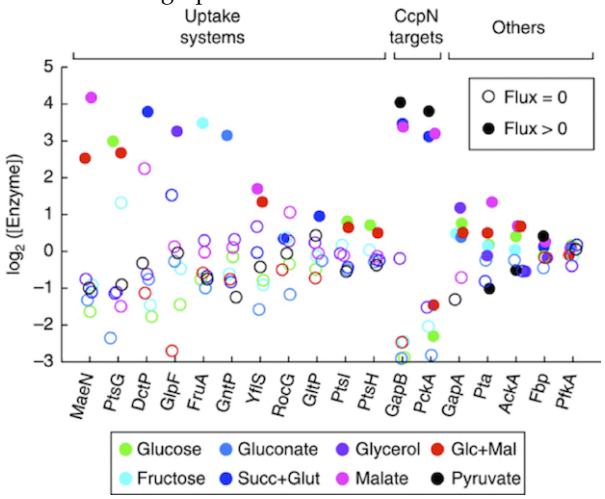


- when the cells are adhering to a surface
- when the cells are growing in suspension for different amounts of time

1. What are the labels/identifiers for the facets?
2. Are the frames the same in each facet?
3. There are three different glyphs shown in the frames. Describe each type in terms of its graphical properties.

Problem 6.2

Consider this graph



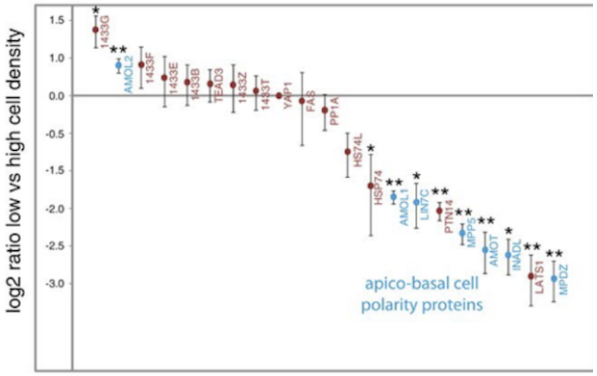
Here are some of the variables and their levels:

- Log enyzme `concentration`: numerical −3 to 5
- `target`: CcpN, Uptake, Other
- `flux`: zero or positive
- `gene`: MaeN, PtsG, DctP, ...
- `molecule`: Glocose, Fructose, Gluconate, ...

1. List all of the **guides** in the graph. For each one, say which variable is being mapped to which graphical attribute.
2. The basic glyph is a dot. Say what are the graphical attributes of the dot (e.g. color, size, ...). For each graphical attribute found in the graph, say which variable is mapped to that attribute.
3. Which two variables set the frame?
4. The scaling of the horizontal variable (e.g. the translation of position to variable levels) is set by a combination of two variables. Which two?

Problem 6.3

Consider this graphic:



Suppose the glyph-ready data underlying the graphic were structured as follows:

protein	center	low	high	polarity	signif
1433G	1.35	1.18	1.54	plus	1
AMOL2	0.78	0.63	1.01	minus	2
1433F	0.79	0.18	1.19	plus	0
1433E	0.42	-0.15	1.01	plus	0
⋮	⋮	⋮	⋮	⋮	⋮

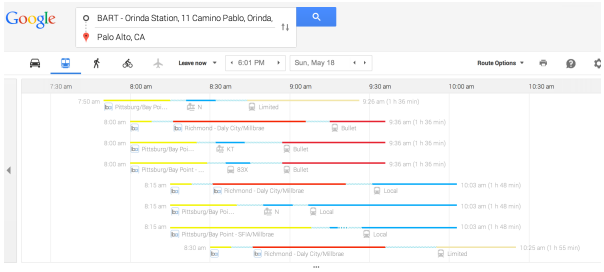
Consider these two kinds of glyph present in the graph:

 and **

1. For each of the two glyphs, list the set of graphical attributes both geometrically (e.g. "dot") and in terms of the variable from the table that is mapped to that attribute (e.g., `polarity`).
2. Which variables define the frame? Give variables for both the horizontal and vertical coordinates.
3. Is color an attribute of the ** glyph?
4. What guides (if any) are displayed?

Problem 6.4

The graph, from Google Maps, shows mass transit options on a Monday morning for getting from Orinda, CA (in the East Bay), to Palo Alto, CA (in the West Bay).



(For a larger version, see `Data-Computing.org/images#C133`.)

1. Considering only that part of the graphic below the blue underlined bus and other modes of transportation, what is the frame?
2. Describe the different types of glyphs used.
3. For each different type of glyph
   - What information is encoded in the shape/style of the glyphs?
   - What information is encoded in the position of the glyph?
4. What guides are there?

Figure 6.9 presents forecasts for the US Senate elections in Nov. 2014. The numbers or words give the forecast probability of one party's candidate — Democrat or Republican — winning. The forecasts are made based on polls up through the end of August 2014. Individual results from several different polling organization are shown. The graphic is an excerpt from the full graphic at `http://www.nytimes.com/newsgraphics/2014/senate-model/comparisons.html`, which shows predictions for all 36 senate seats up for election in 2014. Source: New York Times

Competitive States	NYT Aug 31	538 Aug 4	Cook Aug 22	Roth. Aug 29	Sabato Aug 27	WaPo Aug 29
New Hampshire	84% Dem.	90% Dem.	Leaning	Likely	Likely	>99% Dem.
Michigan	74% Dem.	65% Dem.	Tossup	Leaning	Likely	99% Dem.
Colorado	57% Dem.	60% Dem.	Tossup	Tossup	Leaning	65% Dem.
Iowa	53% Dem.	55% Dem.	Tossup	Tossup	Tossup	63% Rep.
Alaska	52% Dem.	Even	Tossup	Tossup	Tossup	66% Dem.
North Carolina	51% Rep.	Even	Tossup	Tossup	Tossup	91% Dem.
Louisiana	60% Rep.	55% Rep.	Tossup	Tossup	Tossup	51% Dem.
Arkansas	66% Rep.	60% Rep.	Tossup	Tossup	Tossup	65% Rep.
Georgia	82% Rep.	75% Rep.	Tossup	Likely	Leaning	83% Rep.
Kentucky	86% Rep.	80% Rep.	Tossup	Leaning	Likely	94% Rep.

* Rothenberg ratings are converted from a nine-category scale to a seven-category scale to make comparisons easier.

Solid Dem. · Likely Dem. · Leaning Dem. · Tossup · Leaning Rep. · Likely Rep. · Solid Rep.

Figure 6.9: Forecasts made before the Nov. 2014 US Senate elections.

Problem 6.5

In Figure 6.9, what variables define the frame in this graphic?

a. Probability and State.

b. State and Polling Organization.

c. Democrats and Republicans.

d. Just State

e. Just Probability

Problem 6.6

In Figure 6.9, what is the glyph and its graphical attributes?

a. Glyph: names of the states. Graphical attribute: font.

b. Glyph: names of the polling organization. Graphical attribute: the organization's logo.

c. Glyph: Rectangle. Graphical attribute: color.

d. Glyph: Rectangle. Graphical attribute: color and text.

Problem 6.7

In Figure 6.9, what sets the order of the categorical variable in the scale for the vertical variable?

a) State

b) Poll

c) Roth poll probability for the Democratic candidate.

d) NYT poll probability for the Democratic candidate.

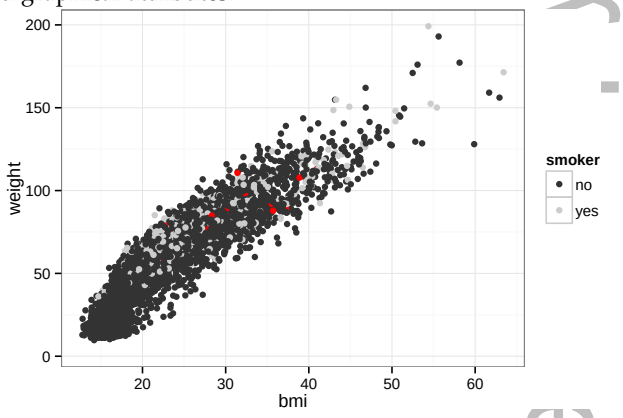
e) Date of the poll.

Problem 6.8

The `NCHS` data (in the `DataComputing` package) has 31126 rows. To speed things up, work with a small subset of `NCHS`:

```
Small <-
  NCHS %>%
  sample_n(size=5000)
```

Using the data in `Small`, make this plot with `scatterGraphHelper()` (in the `DataComputing` package). Then, write down the mapping between variables and graphical attributes.



7
Wrangling and Data Verbs

When data are in glyph-ready form it is straightforward to construct data graphics using the concepts and techniques in Chapters 5. First, you choose an appropriate sort of glyph: dots, stars, bars, countries, etc. Next, select which variables are to be mapped to the various aesthetics for that glyph. Then let the computer do the drawing.

On occasion, data will arrive in glyph-ready form. More typically, you have to *wrangle* the data into the glyph-ready form appropriate for your own purpose.

WRANGLE: Transforming data from one or more data tables into a glyph-read format. The literal meaning of "wrangle" is to herd or round-up animals for some purpose such as bringing them to market. "Data wrangling" is a metaphor for putting data into a form suitable for the data visualization or analysis that's desired.

7.1 Examples of Wrangling

EXAMPLE: COUNTING THE BALLOTS. Table 7.1 records the choice of each of 80101 individual voters in the 2013 mayoral election in Minneapolis, MN, USA.

Precinct	First	Ward
P-10	BETSY HODGES	W-7
P-06	BOB FINE	W-10
P-09	KURTIS W. HANNA	W-10
P-05	BETSY HODGES	W-13
P-01	DON SAMUELS	W-5
... and so on for 80,101 rows		

Table 7.1: The choices of each voter in the mayoral election. See `Minneapolis2013` in the `DataComputing` package.

The primary use for data like Table 7.1 is to determine how many votes were given to each candidate Counting the votes for each candidate is a simple wrangling task that transforms the ballot data (Table 7.1) into a glyph-ready form (Table 7.2 right) that makes the answer obvious. The count information is still latent in the raw ballot data; it is not yet in a form where it is easily seen.

candidate	count
BETSY HODGES	28935
MARK ANDREW	19584
DON SAMUELS	8335
CAM WINTON	7511
JACKIE CHERRYHOMES	3524
... and so on for 38 rows	

Table 7.2: The number of votes for each candidate. This has been wrangled from Table 7.1 into a glyph-ready form, that is, a form in which the sought after information is readily seen.

The instructions for carrying out the wrangling are simple to give in English: Count the number of ballots that each candidate received and sort from highest to lowest. To carry out the process on a computer, you need to express this idea in a form the computer can carry out.

7.2 Planning your wrangling

Before you start wrangling data, it's crucial to envision what the goal is to be. In general, the purpose of wrangling is to take the data you have at hand and put it into glyph-ready form. But glyph-ready for

what? This is something you, the data analyst, need to determine based on your own objectives.

EXAMPLE: DEMOGRAPHIC PATTERNS OF SMOKING Table 7.3 has variables indicating each person's age, sex, and whether he or she smokes..

Suppose you want to use NCHS to explore the links between smoking and age. Often you will have some kind of presentation graphic in mind. Making an informal sketch like Figure 7.1 can be a helpful way to chart a path for data wrangling. The sketch can be based on your imagination of what the data might show. Even so, the sketch contains important information. For instance, from the sketch you can determine what a single glyph represents. As always, each glyph will be one case in the glyph-ready data. In this sketch, the glyph describes a *group* of people — people of one sex in one age group. This differs from the cases in NCHS: individual people.

To make your goal for data wrangling even more explicit, rough out the form of the glyph-ready data table, as in Table 7.4. Don't worry about calculating precise data values; the computer will do that after you have implemented your data wrangling plan.

The glyph-ready data for that graph will has a form like Table 7.4. That's glyph-ready form is your target.

age	sex	smoker
2	female	no
77	male	no
10	female	no
1	male	no
49	male	yes
... and so on for 29,375 rows		

Table 7.3: The NCHS data from the DataComputing package. In NCHS, the case is an individual person.

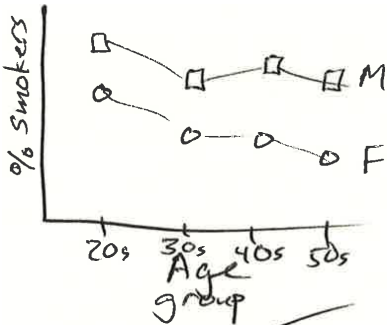


Figure 7.1: A made-up depiction of trends in the fraction of people of different ages who smoke.

ONCE YOU HAVE A TARGET GLYPH-READY FORMAT in mind, use plain English to plan the individual data wrangling steps. For instance, a plan for wrangling Table 7.3 into Table 7.4 might look like this:

1. Turn the age in years into a new variable, the age group (in decades).
2. Drop the people under 20-years old.
3. Divide up the table into separate groups for each age decade and sex.
4. For each of those groups, count the number of people and the number of people who smoke. Divide one by the other to get the fraction who smoke.

Age group	sex	smokers
20s	F	31%
30s	F	15%
40s	M	20%

Table 7.4: A sketch of the form of glyph-ready data corresponding to Figure 7.1. For each aesthetic in Figure 7.1 there is one variable in the glyph-ready table.

Each of these step would be easy enough to do by hand (if you had the time and patience to work through 31126 cases in NCHS). To get the computer to do the work for you, you have to be able to describe each process to the computer. The next sections present a framework for describing wrangling operations that can works for both the human describing the wrangling and the computer that will carry out the calculations.

7.3 A Framework for Data Wrangling

Over the last half century, researchers have identified a small set of patterns that can be used to describe a wrangling process. Because this set is small, it's feasible for you to learn quickly what you need to describe the wrangling you have planned.

As you know, *doing something* in R is accomplished using functions. A function takes one or more inputs (the "arguments") and returns an output. In thinking about data wrangling, it helps to consider these potential forms for inputs and outputs:

- A **data table**, a collection of variables and cases.
- A **variable**, just a single one from the collection in a data table.
- A **scalar**, a single number or character string.

There are three broad families of functions involved in data wrangling. The families differ in what form of input they take in and what form they return.

1. **Reduction functions** take a variable as input and return a scalar.
2. **Transformation functions** take one or more existing variables as input and return a new variable.
3. **Data verbs** take an existing data table as input and return a new data table.

Each step in data wrangling involves a data verb and one or more reduction or transformation functions.

Reduction Functions

Reduction functions summarize or reduce a variable to scalar form. You are likely familiar with several reduction functions:

- `mean()` — find a single typical value
- `sum()` — add up numbers into a total
- `n()` — find how many cases there are

Some other frequently used reduction functions:

- `min()` and `max()` — find the smallest and largest value in a variable
- `median()`, `sd()` and other functions used in statistics
- `n_distinct()` — how many different levels are there among the cases

Transformation Functions

Transformation functions take one or more variables as input and return a new variable. In contrast to reduction functions, which produce a single number — a scalar — by combining all the cases, transformation functions produce a result for each individual case.

Often, transformations are mathematical operations, for instance:

- `weight / height`
- `log10( population )`
- `round( age )`

Other transformation functions include numerical and character comparisons, as in Table 7.5.

Function	Meaning	Example
<	less than	`age < 21`
>	greater than	`age > 65`
==	matches exactly	`age == 39`
%in%	is one of	`age %in% c(18, 19, 20)`
	For character strings	
<	alphabetically before	`name < "ad"`
>	alphabetically after	`name > "ac"`
==	matches exactly	`name == "Jon"`
%in%	is one of	`name %in% c("Abby", "Abe")`

Table 7.5: Some of the functions used to compare values.

Another important operation, `ifelse()` allows you to translate each value in a variable to one of two values, depending on the result of a comparison. For instance

```
ifelse(age >= 18, "voter", "non-voter")
```

Data Verbs

Data verbs carry out an operation on a data table and return a new data table. Some data verbs involve the modification of variables, the creation of new variables, or the deletion existing ones. Other data verbs involve changing the meaning of a case or adding or deleting cases.

In addition to a data table input, each data verb takes additional arguments that provide the specifics of the operation. Very often, these specifics involve reduction and transformation functions.

There are about a dozen data verbs that are commonly used. You can start with these two:

- `summarise()` — turns multiple cases into a single case using reduction functions. "Aggregate" is synonym for "summarise."
- `group_by()` — modifies the action of reduction functions so that they give a single value for different groups of cases in a data table.

To illustrate the uses of `summarise()`, look at the `WorldCities` data table (in the `DataComputing` package) with information about the most populous cities in each country. Table 7.7 is a subset of the variables:

name	population	country	latitude	longitude
Shanghai	14608512	CN	31.22	121.46
Buenos Aires	13076300	AR	-34.61	-58.38
Mumbai	12691836	IN	19.07	72.88
Mexico City	12294193	MX	19.43	-99.13
Karachi	11624219	PK	24.91	67.08
Istanbul	11174257	TR	41.01	28.95
Delhi	10927986	IN	28.65	77.23
Manila	10444527	PH	14.60	120.98
Moscow	10381222	RU	55.75	37.62
Dhaka	10356500	BD	23.71	90.41
	... and so on for 23,018 rows			

Table 7.7: World Cities

A simple summary of the data is a count of the number of cities.

```
WorldCities %>%
  summarise(count = n())
```

count
23018

The reduction verb is `n()`. The expression `count = n()` means that the output data table should have a variable named `count` containing the results of `n()`.

Perhaps you want to know the total population in these cities, or the average population per city or the smallest city in the table, as in Table 7.8.

```
WorldCities %>%
  summarise(averPop  = mean(population, na.rm=TRUE),
            totalPop = sum(population, na.rm=TRUE),
            smallest = min(population, na.rm=TRUE))
```

averPop	totalPop	smallest
112624.76	NA	0

Table 7.8: Summary statistics of the population of world cities.

The average city on the list has about 110,000 people. The total population of people living in these cities is about 2.6 billion — a bit more than one-third of the world population. The smallest city has ... zero people! Evidently, the `WorldCities` data table has cases that are not really cities.

Some things to notice about the use of `summarise()`:

The `n()` function doesn't take any arguments.

- The chaining syntax, %>% has been used to pass the first argument to `summarize()`. This will be useful later on, when there is more than one step in a transfiguration. It's OK to *end* a line with %>%, but **never** start a line with it.
- The output of `summarise()` is a data table. (In this example the output has only one case, but it is still a data table.)
- `summarise()` takes named arguments. The name of an argument is taken as the name of the corresponding variable created by `summarise()`.

group_by

The `group_by()` data verb sets things up so that other data verbs will perform their action on a group-by-group basis. For instance, Figure 7.9 shows the number of cities in `WorldCities` broken down by country.

```
WorldCities %>%
  group_by( country ) %>%
  summarise( count=n() )
```

The `group_by()` verb should always be followed by another verb — `summarise()` in the above example. `group_by()` is the way to indicate to the following verbs that reduction operations should be performed on a group-wise basis.

Using `group_by()` along with `summarise()` and `n()`, provides a basis for counting up the number of cases in groups and subgroups, or for calculating group-wise statistics.

Note that the functions used within arguments to `summarise()` — functions such as `n()`, `max()`, `sum()`, etc. — are **not** data verbs. A data verb takes a data table as input and returns a transfigured data table as output. In contrast, the reduction functions, `n()`, `sum()`, and so on, take a variable as input and return a *single number* as output.

country	count
AD	2
AE	12
AF	50
AG	1
AI	1
... and so on for 243 rows	

Table 7.9: The number of cities in each country listed in `WorldCities`.

Is That All?

The `summarise()` and `group_by()` data verbs are team players; they work best in combination with other data verbs. Once you learn those data verbs, particularly `filter()` and `mutate()`, you'll be able to carry out many more operations. The richness of the data-verb system comes from the ways the different verbs can be combined together.

EXAMPLE: SUMMARIES IN HEALTH-RELATED DATA. `NCHS` contains records of body shape, health, and mortality of 31126 people.

```
data(NCHS, package = "DataComputing")
```

One of the variables in `NCHS` is `hdl` — HDL cholesterol. (HDL is the "good" kind of cholesterol, as opposed to LDL cholesterol, which is reported in the `chol` variable).

Suppose you want to know a typical HDL level for the people enrolled in NCHS. A typical value will be a fair representative of all the cases. It combines together information found in the individual cases. Since all the cases are being combined, `summarise()` is appropriate.

```
NCHS %>%
  summarise(typical = mean(hdl, na.rm = TRUE))
```

typical
52.37

Or, suppose for some reason you want the typical HDL, the tallest height, and the number of cases.

```
NCHS %>%
  summarise(typical = mean(hdl, na.rm = TRUE),
            tallest = max(height, na.rm = TRUE))
```

typical	tallest
52.37	2.04

`na.rm = TRUE` instructs the reduction function to ignore missing data. Table 7.10 shows what happens without `na.rm=TRUE` — the missing data shapes the result and no useful information is produced.

```
NCHS %>%
  summarise(typical = mean( hdl ),
            tallest = max( height ))
```

typical	tallest
NA	NA

Table 7.10: When there is missing data (`NA`), functions like `mean()`, `max()`, etc. will indicate that the result is `NA`. This is rarely what you want, so use the named argument `na.rm` to instruct the functions to ignore the missing data: `na.rm = TRUE`

Using `group_by()` in conjunction with `summarise()` lets you calculate a summary for each of several groups. The groups can be defined by a single variable or by two or more variables. Table 7.11 shows a division into groups by `sex`:

```
NCHS %>%
  group_by(sex) %>%
  summarise(typical = mean(hdl, na.rm = TRUE),
            tallest = max(height, na.rm = TRUE))
```

sex	typical	tallest
female	55.70	1.87
male	48.91	2.04

Table 7.11: Dividing `NCHS` into groups by sex.

In Table 7.12, the division is into smokers and non-smokers of each sex.

```
NCHS %>%
  group_by( sex, smoker ) %>%
  summarise(typical = mean(hdl, na.rm=TRUE),
            tallest = min(height, na.rm=TRUE))
```

sex	smoker	typical	tallest
female	no	56.06	0.80
female	yes	53.51	1.40
female	NA	53.12	0.93
male	no	49.19	0.79
male	yes	47.80	1.52
	... and so on for 6 rows		

Table 7.12: Dividing NCHS into groups by smoker and sex.

EXAMPLE: COUNTING BIRTHS Suppose you want to create a graph like Figure 7.2 to examine the relative number of male and female births over time.

The `BabyNames` data contains this information implicitly. Before the information can be graphed, you need to wrangle the existing data table to one in glyph-ready form.

The particular wrangling needed here is to calculate the sum of `count` for each sex in each year. Here's one way to do this.

```
YearlyBirths <-
  BabyNames %>%
  group_by(year, sex) %>%
  summarise(births = sum(count))
```

`YearlyBirths` is in glyph-ready form. Using `YearlyBirths`, drawing Figure 7.2 is a matter of assigning variables to graphical aesthetics: `year` to the x-axis, `births` to the y-axis, and `sex` to dot color.

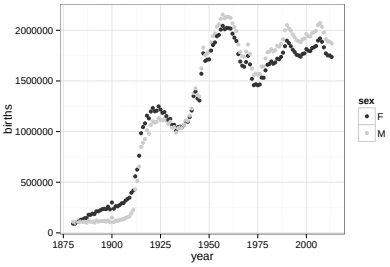


Figure 7.2: Births registered by the US Social Security Administration over the years.

`BabyNames`

name	sex	count	year
Tyshia	F	11	1990
Shacourtney	F	5	1990
Almira	F	7	1990
Elliotte	F	24	2010
Angelmiguel	M	5	2010
... and so on for 1,792,091 rows			

Table 7.13: `BabyNames`

`YearlyBirths`

year	sex	births
1990	F	1897572
1990	M	2052434
2000	F	1814474
2000	M	1962248
2010	F	1771846
... and so on for 268 rows		

Table 7.14: `BabyNames` wrangled into counts of births each year.

7.4 *Exercises*

Problem 7.1
Each of these tasks can be performed using a single data verb. For each task, say which verb it is:

a. Find the average of one of the variables.

b. Add a new column that is the ratio between two variables.

c. Sort the cases in descending order of a variable.

d. Create a new data table that includes only those cases that meet a criterion.

e. From a data table with three categorical variables A, B, & C, and a quantitative variable X, produce an output that has the same cases but only the variables A and X.

f. From a data table with three categorical variables A, B, & C, and a quantitative variable X, produce an output that has a separate case for each of the combinations of the levels of A and B. (Hint: It might be easier to see the answer if the problem statement added, "and gives the maximum value of X over all the cases that have a given combination of A and B.")

Problem 7.2
These questions refer to the `diamonds` data table in the `ggplot2` package. Take a look at the codebook (using `help()`) so that you'll understand the meaning of the tasks. (Motivated by Garrett Grolemund.)

Each of the following tasks can be accomplished by a statement of the form

```
diamonds %>%
  verb1( args1 ) %>%
  verb2( args2 ) %>%
  arrange(desc( args3 )) %>%
  head( 1 )
```

For each task, give appropriate R functions or arguments to substitute in place of `verb1`, `verb2`, `args1`, `args2`, and `args3`.

1. Which color diamonds seem to be largest on average (in terms of carats)?

2. Which clarity of diamonds has the largest average "table" per carat?

Problem 7.3
For each of the operations listed here, say whether it involves a transformation function or a summary function or neither.

a. Determine the 3rd largest.

b. Determine the 3rd and 4th largest values.

c. Determine the number of cases.

d. Determine whether a year is a leap year.

e. Determine whether a date is a legal holiday.

f. Determine the range of a set, that is, the max minus the min.

g. Determine which day of the week (e.g., Sun, Mon, . . .) a given date is.

h. Find the time interval in days spanned by a set of dates.

Problem 7.4
Each of these statements have an error. It might be an error in syntax or an error in the way the data tables are used, etc. Tell what are the error(s) in these expressions.

a)
```
BabyNames %>%
    group_by( "First" ) %>%
    summarise( votesReceived=n() )
```

b)
```
Tmp <- group_by(BabyNames, year, sex ) %>%
    summarise( Tmp, totalBirths=sum(count))
```

c)
```
Tmp <- group_by(BabyNames, year, sex)
summarise( BabyNames, totalBirths=sum(count) )
```

Problem 7.5
Here is a small data table based on `BabyNames`. Take this table as the input.

name	sex	count	year
Christina	M	22	1967
Rotha	F	7	1907
Wayman	M	9	1997
Song	F	11	1994
Julian	M	535	1948
... and so on for 1,792,091 rows			

For each of the following outputs, identify the operation linking the input to the output and write down the details (i.e., arguments) of the operation.

a) Output Table A

name	sex	count	year
Rotha	F	7	1907
Song	F	11	1994
Kalia	F	46	1989
Lissa	F	102	1962
Vicky	F	2945	1957
... and so on for 1,792,091 rows			

b) Output Table B

name	sex	count	year
Rotha	F	7	1907
Song	F	11	1994
Vicky	F	2945	1957
Kalia	F	46	1989
Lissa	F	102	1962
... and so on for 896,046 rows			

c) Output Table C

name	sex	count	year
Christina	M	22	1967
Julian	M	535	1948
... and so on for 416,765 rows			

d) Output Table D

total
333417770

e) Output Table E

name	count
Christina	22
Rotha	7
Wayman	9
Song	11
Julian	535
... and so on for 1,792,091 rows	

Problem 7.6
Using the `Minneapolis2013` data table, answer these questions:

1. How many cases are there?
2. Who were the top 5 candidates in the `Second` vote selections.
3. How many ballots are marked "undervote" in
   - `First` choice selections?
   - `Second` choice selections?
   - `Third` choice selections?
4. What are the top 3 combinations of `First` and `Second` vote selections? (That is, of all the possible ways a voter might have marked his or her first and second choices, which received the highest number of votes?)
5. Which `Precinct` had the highest number of ballots cast?

Problem 7.7
Each of these statements has an error. It might be an error in syntax or an error in the way the data tables are used, etc. Write down a correct version of the statement.

a)
```
BabyNames %>%
  group_by(BabyNames, year, sex) %>%
  summarise(BabyNames, total = sum(count))
```

b)
```
ZipGeography <-
  group_by(State) %>%
  summarise(pop = sum(Population))
```

c)
```
Minneapolis2013 %>%
  group_by(First) ->
  summarise(voteReceived = n())
```

d)
```
summarise(votesReceived = n()) %<%
  group_by(First) <- Minneapolis2013
```

Problem 7.9

a. There's only one data verb that takes a single data table as input and produce an output that (in general) has a different meaning to the case. Which one?

b. There's only one operation that takes two data tables as input rather than just a single data table. Which one?

Problem 7.10

Using the `ZipGeography` data

Find the total land area and population in each state.

- Make a scatter plot showing the relationship between land area and population for each state.
- Make a choropleth map showing the population of each state.
- Make a choropleth map showing the population per unit area of each state.

Problem 7.11

Imagine a data table, `Patients`, with categorical variables `name`, `diagnosis`, `sex`, and quantitative variable `age`.

You have a statement in the form

```
Patients %>%
  group_by(  **some variables** ) %>%
  summarise(count=n(), meanAge = mean(age))
```

Replacing `**some variables**` with each of the following, tell what variables will appear in the output

a. `sex`

b. `diagnosis`

c. `sex, diagnosis`

d. `age, diagnosis`

e. `age`

Problem 7.12

For each of these computations, say what R function is the most appropriate:

1. Count the number of cases in a data table.
2. List the names of the variables in a data table.
3. For data tables in an R package, display the documentation ("codebook") for the data table.
4. Load a package into your R session.
5. Mark a data table as grouped by one or more categorical variables.
6. Add up, group-by-group, a quantitative variable in a data table.

8
Graphics and Their Grammar

Chapter 6 introduced the elements of a grammar for drawing graphics. The grammar helps in thinking about how a graphic is constructed by defining a meaningful taxonomy of component parts: frames, scales, guides, facets, layers, etc.

This chapter introduces an *implementation* of the grammar of graphics. The implementation is provided by an R package, `ggplot2`, that is used throughout this book. that provides functions enabling you to specify graphics in terms of their components and have the machine do the drawing for you.

8.1 Specifying a Graph

Once you know the kinds of components that make up a graphic, you're in a position to describe the graphics you want to create. The process works like this:

1. Describe the frame. That is, state which variables map to the two axes.
2. Describe a graphics layer. This is done by
   a. Choosing a glyph style.
   b. Mapping variables to the aesthetics available for that glyph style, e.g. color, size, shape, ...
3. Repeat (2) for any additional graphics layers.

The `ggplot2` package provides functions for implementing each of the above steps.

1. `ggplot()` creates the frame of the graphic.
2. Glyphs are created by functions starting with `geom_`. There are many kinds of glyphs, so there are many `geom_` functions: `geom_point()`, `geom_line()`, `geom_bar()`, `geom_boxplot()`, `geom_density()`, and so on.
   a. The name of the function indicates the kind of glyph.
   b. The mapping between variables and graphical aesthetics for is specified by the `aes()` function.
3. The + symbol is used to add a layer onto the frame.

Putting graphics together: examples

EXAMPLE: PLOTTING PRINCES. Figure 8.1 depicts the number of babies given the name "Prince" over the years.

`BabyNames`, where the Prince data resides, is not in the form needed for this plot. That is, `BabyNames` is not glyph-ready for this particular graphic; there are entries in `BabyNames` for 92599 other names than Prince. Wrangling `BabyNames` to remove those other names is a straightforward but necessary step to draw the graphic. In this situation, wrangling involves only one step:

```
Princes <-
  BabyNames %>%
  filter(name == "Prince")
```

The graphic's frame is `count` *versus* `year`. Frames are created with `ggplot()`. The mapping of the variables `year` and `count` to the x- and y-axes is specified by the `aes()` function.

```
the_frame <- ggplot(data = Princes,
                    aes(x = year, y = count))
```

There are two layers, one showing the number of Princes each year and one with a vertical line at the year Prince's first album was released. In the number-of-princes layer, shape was used to distinguish between the sexes.

```
layer1 <- geom_point(data = Princes, aes(shape = sex))
```

In the second layer, the glyph is a vertical line. This glyph is provided by the `geom_vline()` function:

```
layer2 <- geom_vline(x = 1978)
```

Finally, put them all together with +

```
the_frame + layer1 + layer2
```

CHAINING IS BUILT IN TO `ggplot()` and the other graphics functions. This provides a way to write your graphics commands without having to specify, over and over again, the source of data and the mappings of aesthetics. To illustrate, here is another way of describing the Prince graphic:

```
Princes %>%
  ggplot(aes(x = year, y = count, shape = sex)) +
  geom_point() +
  geom_vline(x = 1978)
```

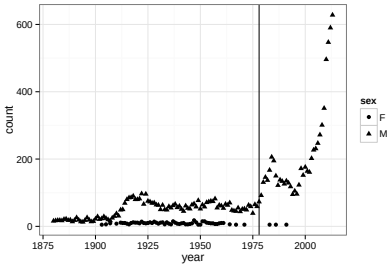


Figure 8.1: The popularity of the name Prince over the years.

As you'll see in Chapter 9, `filter()` is a data verb that passes through only those cases meeting the specified criterion.

name	sex	count	year
Prince	M	49	1912
Prince	M	70	1928
Prince	F	15	1952
Prince	F	9	1956
Prince	M	231	2005
... and so on for 194 rows			

Table 8.1: Glyph-ready data for plotting the popularity of the name "Prince" over time. (Reproduced from Figure 3.1.)

VERSUS: The phrase "A versus B" means that A is on the y-axis and B on the x-axis.

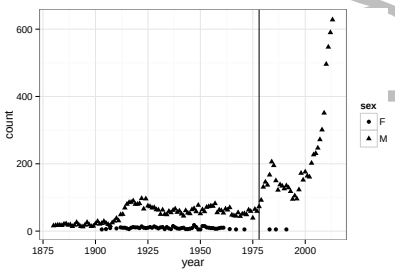


Figure 8.2: Putting together the frame, layer 1, and layer 2 into a complete graphic.

Notice that chaining among the `ggplot()` functions uses the + symbol, rather than `%>%`. It's easy to forget this. When you do, you'll get a cryptic message: `Error: ggplot2 doesn't know how to deal with data of class uneval`. Take this as a hint to look for `%>%` where + ought to be.

EXAMPLE: PEOPLE GROW IN HEIGHT THROUGH CHILDHOOD then stay more or less the same height as adults. Question: Is there systematic change in height during adulthood? Does it depend on smoking?

First, wrangle NCHS into a glyph-ready form to give the mean height at each age, distinguishing between the sexes.

```
Heights <-
  NCHS %>%
  filter( age > 20, ! is.na(smoker)) %>%
  group_by(sex, smoker, age) %>%
  summarise(height = mean(height, na.rm=TRUE))
```

Figure 8.3 has a frame displaying height versus age. Either a point or a line could be used as the glyph. A separate line is made for each group: smokers versus non-smokers. Sex facets the graphic. Line type (dotted or not) distinguishes smokers from non-smokers smokers.

```
Heights %>%
  ggplot(aes(x = age, y = height)) +
  geom_line(aes(linetype = smoker)) +
  facet_wrap( ~ sex)
```

sex	smoker	age	height
female	no	21	1.60
female	no	22	1.62
female	no	23	1.61
female	no	24	1.62
female	no	25	1.63
female	no	26	1.62
... and so on for 260 rows			

Table 8.2: Glyph-ready data for displaying height versus age.

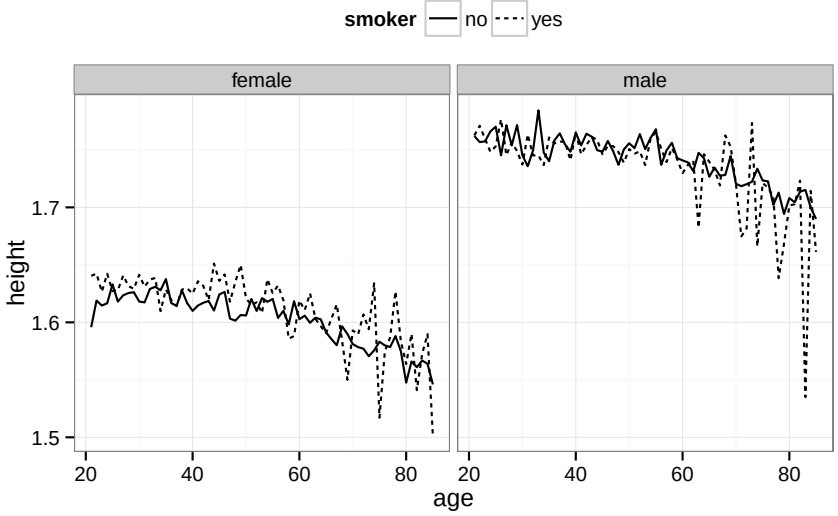


Figure 8.3: Height versus age for smokers and non-smokers.

It looks like both female and male height decline with age. For women, smokers tend to be taller than non-smokers, but not for men.

The statistically savvy reader point out that any causal relationship might go from height to smoking. Perhaps smoking is more common among taller women, rather than non-smoking causing women to be shorter. As well, NCHS does not track individuals over time.[1] It might

[1] NCHS is from a "cross-sectional" study rather than a "longitudinal study."

be that an individual's height doesn't change with age, but the older people in NCHS grew up in a time with poorer nutrition and so are shorter for that reason.

The sharp ups and downs in Figure 8.3 for different ages suggest that there is variation in height that isn't accounted for by age, sex, and smoking status. Such *random variation* can be quantified by a statistical technique called a *confidence band*. These will be introduced in Chapter 14.

8.2 Axes and Labels

By default, the names of the variables used to construct the graphic frame become the labels on the axes. This isn't always desirable. For instance, it's conventional for labels in scientific graphics to include the *units* of the measured quantity. As well, sometimes the default numerical limits are not the best choice: you may want to zoom in on a small region. Or, sometimes, you want to pan out to include anchor values such as zero on the axes.

When the values to be plotted range over several orders of magnitude, it can be appropriate to condense the range using a logarithm transform. Such plots are easier to read when the axis itself has tick marks on a log scale.

With `ggplot()`, such customization is done with plotting commands such as `xlim()` and `ylab()`. Table 8.4 are a few of the most commonly used `ggplot()` functions for refining. Each one of these can be added (using `+`) to your plot, just as you would add another layer.

Function	Purpose	Example
`xlim()`	range of x axis	`+ xlim(30, 75)`
`ylim()`	range of y axis	`+ ylim(1.5, 1.8)`
`xlab()`	x-axis label	`+ xlab("Age (yrs)")`
`ylab()`	y-axis label	`+ ylab("Height (m)")`
`scale_x_log10()`	Log axis for x	`+ scale_x_log()`
`scale_y_log10()`	Log axis for y	`+ scale_y_log()`

Table 8.4: Some of the axis customization functions used with `ggplot()`.

8.3 *Exercises*

Problem 8.1

Here are several functions from the `ggplot2` graphics package used in *Data Computing*.

a)	`geom_point()`	b)	`geom_histogram()`
c)	`ggplot()`	d)	`scale_y_log10()`
e)	`ylab()`	f)	`facet_wrap()`
g)	`geom_segment()`	h)	`xlim()`
i)	`facet_grid()`		

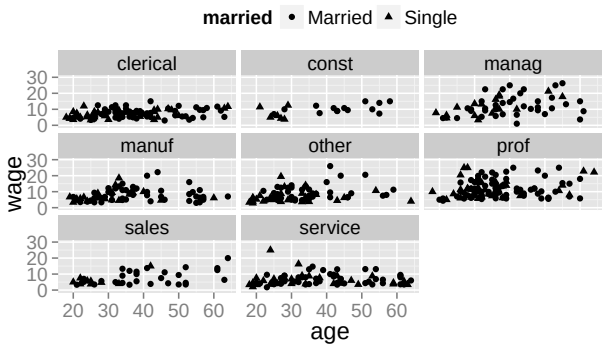
Match each of the functions to the task it performs.

1) Construct the graphics frame
2) Add a layer of glyphs
3) Set an axis label
4) Divide the frame into facets
5) Change the scale for the frame.

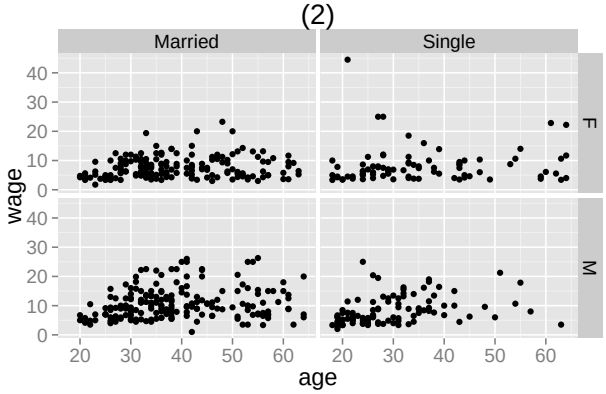
Problem 8.2

Here are two more graphics based on the `mosaicData::CPS85` data table. Write `ggplot()` statements that will construct each graphic.

(1)



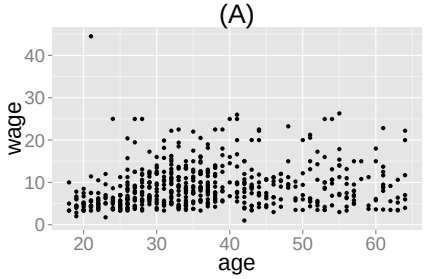
(2)

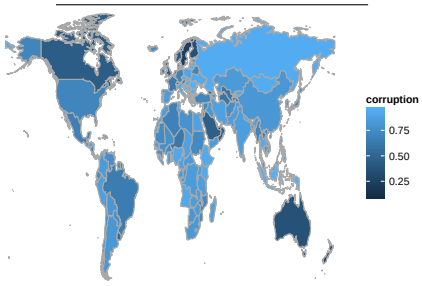


Problem 8.3

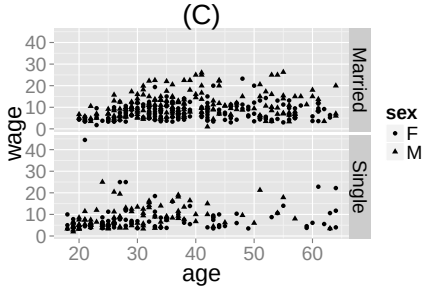
Here are four graphics based on the `mosaicData::CPS85` data table. Write `ggplot()` statements that will construct each graphic.

(A)

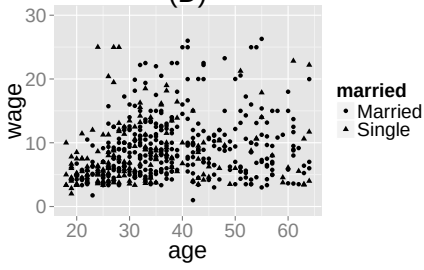




(C)



(D)



9
More Data Verbs

You have already seen two data verbs:

- `summarise()`
- `group_by()`

Although these are being written in computer notation, it's also perfectly legitimate to express actions using them as English verbs. For instance: "Group the baby names by sex and year. Then summarize the groups by adding up the total number of births for each group. This will be the result." That's English. Here's the equivalent statement in computer notation:

```
Result <-
  BabyNames %>%
  group_by( sex, year ) %>%
  summarise( total=sum( count ) )
```

sex	year	total
F	1892	212347
M	1892	122038
F	1935	1048395
M	1935	1040892
F	1978	1531592
... and so on for 268 rows		

Table 9.1: Grouping `BabyNames` by `sex` and `year` then using `summarise()` to add up the total births for each group.

This chapter introduces four new data verbs that, like `summarise()` and `group_by()`, perform their action on a single data table.

- `mutate()`
- `filter()`
- `select()`
- `arrange()`

As with `group_by()` and `summarise()`, each is a standard English word whose action on data is reflected in the colloquial, everyday meaning. And, like English, intricate and detailed statements can be made by combining the words into expressions.

9.1 *Creating variables with mutate()*

The word "mutate" means to change in form or nature. The data verb `mutate()` is more specific: to change a variable or add new variables based on the existing ones. Mutation leaves the cases exactly as they were; the output of `mutate()` will have exactly as many rows as the input data frame.

To illustrate, consider Table 9.2. It has variables `pop` and `area` giving the population and area (in km^2) of each country. Suppose you wanted to know the population *density*, that is, how many people per unit area. Computing the density is a matter of using a transformation verb: divide population by area. The `mutate()` verb let's you carry out this transformation and put the result in a new variable. For instance:

country	pop	area
Afghanistan	31822848	652230
Akrotiri	15700	123
Albania	3020209	28748
Algeria	38813722	2381741
American Samoa	54517	199
... and so on for 256 rows		

Table 9.2: The `Demographics` table created by selection from `CountryData`.

```
Demographics <-
  Demographics %>%
  mutate(pop_density = pop / area)
```

Note that in this R statement `Demographics` is the input to `mutate()` and the result is being stored under the name `Demographics`. This style can be read as "revise the `Demographics` table ...." table. Of course, you could choose to give the result a new name or to pipe the result to another stage of data wrangling, etc.

The arguments to `mutate()` specify what the new variable or variables are to be called (`pop_density` above) and how to calculate the values of those variables. This calculation will *always* involve a transformation variable, just as the `summarise()` verb *always* involves a reduction verb.

9.2 *Removing cases with filter()*

To *filter* means to remove unwanted material. The data verb `filter()` removes unwanted cases, passing through to the result only those cases that are wanted or needed.

```
BabyNames %>%
  filter(sex != "M")
```

Filtering contrasts with selecting. Selecting passes the specified `variables`; filtering passes the specified `cases`.

With filtering, the cases to pass through are specified by one or more criteria or tests. The tests usually involve comparisons between variables or between a variable and a scalar. Table 9.4 gives examples of the comparison functions and their uses.

For instance, here's how you can filter out the boys, producing a result with only the girls' names, shown in Table 9.3.

```
BabyNames %>%
  filter(sex != "M")
```

name	sex	count	year
Mary	F	7065	1880
Anna	F	2604	1880
Emma	F	2003	1880
... and so on for 1,062,432 rows			

Table 9.3: Using `filter()` to exclude the males.

Or, you could filter on the basis of year, for instance, keeping the cases for either sex for babies born after 1990:

```
BabyNames %>%
  filter( year > 1990 )
```

Table 9.4: Examples of the sorts of comparisons used as criteria in `filter()`.

Comparison	Function	Example	Meaning
Exact equality	`==`	`year == 1995`	matches a number
		`name == "Julian"`	matches a character string
Greater than	`>`	`year > 1995`	years after 1995
		`name > "C"`	names starting with "D" or after
Less than	`<`	`year < 1995`	years before 1995
		`name < "C"`	names starting with "A" or "B"
Less than or equal to	`<=`	`year <= 1995`	year is 1995 or before
		`name <= "C"`	names alphabetically before "C"
Greater than or equal to	`>=`		Like <=
Contained in	`%in%`	`year %in% c(1940, 1941)`	matches either 1940 or 1941
		`name %in% c("April","May","June")`	A springtime name

You can specify as many tests as you like. For instance, to keep only the cases for females born after 1990:

```
BabyNames %>%
  filter( year > 1990, sex=="F")
```

The `filter()` function will pass through only those cases that pass *all* the tests.

Sometimes you may want to set "either-or" criteria, say the babies who are female *or* born after 1990. To specify either-or, use the vertical bar | rather than a comma to separate the conditions.

```
BabyNames %>%
  filter(year > 1990 | sex == "F")
```

It's also possible to test for a variable being any of several different values. For instance, Table 9.7 includes just the babies born in any of 1980, 1990, 2000, and 2010:

```
BabyNames %>%
  filter( year %in% c(1980, 1990, 2000, 2010))
```

name	sex	count	year
Aishah	F	17	1992
Derrius	M	22	1997
Bert	M	21	2000
Jadyn	F	1063	2007
Avalon	F	132	2008
... and so on for 697,983 rows			

Table 9.6: Baby names after 1990.

name	sex	count	year
Roma	F	9	1980
Guillermina	F	31	1980
Camellia	F	23	1990
Ryler	M	5	2000
... and so on for 107,935 rows			

Table 9.7: Filtering to keep only babies born in 1980, 1990, or 2000.

9.3 *Choosing variables with select()*

Selecting from a data table means choosing one or more variables from the table, creating a new table that has just a subset of the variables in the original. For now, think of `select()` as enabling you

to simplify a table to eliminate variables that you don't need or to rename a variable.

The use of `select()` is straightforward. It takes a data table as input along with the names of one or more variables. To illustrate, the `Demographics` data table (Table 9.2) was created from a much, much larger table in the `DataComputing` package: `CountryData`. There are 76 variables in `CountryData`.

```
names(CountryData)
```

```
 [1] "country"      "area"         "pop"          "growth"       "birth"
 [6] "death"        "migr"         "maternal"     "infant"       "life"
[11] "fert"         "health"       "HIVrate"      "HIVpeople"    "HIVdeath"
   ... and so on ...
[61] "petroImp"     "gasProd"      "gasCons"      "gasExp"       "gasImp"
[66] "gasRes"       "mainlines"    "cell"         "netHosts"     "netUsers"
[71] "airports"     "railways"     "roadways"     "waterways"    "marine"
[76] "military"
```

By modern standards for computing, 76 is not at all a large number of variables. But for a printed display like Table 9.2, it is ungainly. `Demographics` was derived from `CountryData` like this:

```
Demographics <-
  CountryData %>%
  select(country, pop, area)
```

The `BabyNames` data table provides an illustration. It has four variables (Table 9.8). Selecting just the `name` and `year` variables produces Table 9.9.

```
BabyNames %>%
  select( name, year )
```

If you want to rename a variable, use `rename()`. The arguments to `rename()` specify which variables to rename and what the new name will be. For instance, to change `year` to `when` use the argument `when = year`. (Table 9.10)

```
BabyNames %>%
  select( name, year ) %>%
  rename(when = year)
```

name	sex	count	year
Mary	F	7065	1880
Anna	F	2604	1880
Emma	F	2003	1880
Elizabeth	F	1939	1880
Minnie	F	1746	1880
... and so on for 1,792,091 rows			

Table 9.8: `BabyNames`

name	year
Mary	1880
Anna	1880
Emma	1880
Elizabeth	1880
Minnie	1880
... and so on for 1,792,091 rows	

Table 9.9: Selecting just the `name` and `year` variables from `BabyNames`.

name	when
Mary	1880
Anna	1880
Emma	1880
Elizabeth	1880
Minnie	1880
... and so on for 1,792,091 rows	

Table 9.10: Renaming the `year` variable from Table 9.9.

9.4 *Setting in order with arrange()*

To "arrange" means to set in order. The `arrange()` data verb sorts out the cases in an order that you specify: it might be alphabetical

or numeric, in ascending or descending order. `arrange()` does not change the variables in the data table or filter the cases. Arranging merely sets the order of cases according to some criterion that you specify.

For instance, Table 9.11 shows the first-choices from the Minneapolis mayoral election in 2013 found by counting the ballots:

```
Minneapolis2013 %>%
  group_by( First ) %>%
  summarise( total=n() )
```

First	total
ALICIA K. BENNETT	351
BETSY HODGES	28935
BILL KAHN	97
BOB FINE	2094
CAM WINTON	7511
... and so on for 38 rows	

Table 9.11: Ballot count of first-choice candidates.

The ascending alphabetical order in Table 9.11 might be good for some purposes. But if your goal is to show who won and how they did compared to the other candidates, it's better to arrange the results by `total` in descending descending numeric order as in Table 9.12. By default, `arrange()` puts cases in ascending order: from smallest to largest for numbers and from A to Z for character strings. You can set the order to be descending by using the `desc()` function.

```
Minneapolis2013 %>%
  group_by( First ) %>%
  summarise( total=n() ) %>%
  arrange( desc(total) )
```

First	total
BETSY HODGES	28935
MARK ANDREW	19584
DON SAMUELS	8335
CAM WINTON	7511
JACKIE CHERRYHOMES	3524
... and so on for 38 rows	

Table 9.12: Arranging the candidates in descending order by vote total.

9.5 *Even more data verbs*

A data verb is any function that takes a data table as input and gives a data table as the return value. In addition to the ones already introduced, you may find these useful in your work:

Data Verb	Purpose	Example	Arguments
`head()`	Grab the first few rows	`BabyNames %>% head(5)`	Number of rows to grab
`tail()`	Grab the last few rows	`BabyNames %>% tail(3)`	Number of rows
`transmute()`	Create new variables	Like `mutate()`, returns only new variables	
`rename()`	Rename variables	Keeps unreferenced variables	
`sample_n()`	Take a random rows	`BabyNames %>% sample_n(size=5)`	Size of sample.

9.6 *Summary functions*

To look at a data table, these functions are useful. Each takes the data table as the only argument.

Function	Purpose
`glimpse()`	Quick summary of the table
`str()`	Quick summary
`summary()`	Quick summary
`nrow()`	How many cases in the data table
`ncol()`	How many variables in the data table
`names()`	The variable names
`View()`	Shows the table like a spreadsheet. In RStudio only.

9.7 *Exercises*

Problem 9.1
Identify each of these functions as either a **Data Verb**, a **Transformation**, a **Summary Function**, or a **Quick Presentation** or a **Comparison Expression**. (Hint: If you are unfamiliar with the function, use `help()`.)

a) `str()` b) `group_by()` c) `rank()` d) `mean()` e) `filter()` f) `summary()` g) `summarise()` h) `anti_join()` i) `glimpse()`

Problem 9.2
In the ranked-choice ballot system, once a voter has picked a first choice candidate, there is no advantage in listing that same candidate as second or third choice.

In answering each of the following, you should include three things:

1. A statement in English (using data verbs!) of your strategy for carrying out the calculation.
2. The implementation of the calculation in R.
3. The data table that is the result of the calculation. This will be printed automatically from (2). You **do not** have to format the result beautifully. It's sufficient to show the first few lines in the data table appear. (Remember `head()`).

Here are the questions.

- How many people chose the same candidate for both `First` and `Second` place?
- Of the ballots where the `First` and `Second` place choices are the same, what were the top 3 choices?
- Of the people who selected Ole Savior for `First`, what were the top three `Second` choices?

Problem 9.3
These questions refer to the `diamonds` data table in the `ggplot2` package. Take a look at the codebook (using `help()`) so that you'll understand the meaning of the tasks. (Motivated by Garrett Grolemund.)

Each of the following tasks can be accomplished by a statement of the form

```
diamonds %>%
  verb1( args1 ) %>%
  verb2( args2 ) %>%
  arrange(desc( args3 )) %>%
  head( 1 )
```

For each task, give appropriate R functions or arguments to substitute in place of `verb1`, `verb2`, `args1`, `args2`, and `args3`.

1. Which color diamonds seem to be largest on average (in terms of carats)?
2. Which clarity of diamonds has the largest average "table" per carat?

Problem 9.4
Using the `ZipGeography` data table, answer the following questions. In addition to the answer itself, show the statement that you used and the data table created by your statement that contains the answer.

- How many different counties are there?
- Which city names are used in the most states?
- Which city names with more than 5% of the state population are used in the most states?
- Does any state have more than one time zone?
- Does any city have more than one time zone?
- Does any county have more than one time zone?

10
Joining Two Tables

The data verbs introduced so far — `summarise()`, `mutate()`, `arrange()`, and so on — always take as input a single data table along with other arguments to specify the exact operation to be carried out. This chapter introduces data verbs that take *two tables* as input and produce a single table as output. Such operations enable you to combine information from different tables. With these two-table verbs, you can construct data-wrangling processes that combine *many* data tables.

There are many situations where data from different sources have to be brought together to create a complete picture. For instance, the `CountryData` in Chapter 9 does not include even simple information about the geographic locations of the countries. If you want to plot the demographic data on a map, you will need to provide some geographic location data, as in `CountryCentroids` (from the `DataComputing` package).

Arithmetic operations such as $+$, $-$, $\times$, and $\div$ combine two numbers, e.g. $3 + 7$. Using such simple operations, you can construct more complicated ones that combine many numbers, e.g. $3 * (5/4 + 7)$. The two-table verbs can be used in analogous ways to combine information from multiple tables.

Demographics

country	pop	area
Afghanistan	31822848	652230
Akrotiri	15700	123
Albania	3020209	28748
Algeria	38813722	2381741
American Samoa	54517	199
... and so on for 256 rows		

CountryCentroids

name	iso_a3	long	lat
Afghanistan	AFG	66.17	33.78
Albania	ALB	20.26	41.14
Algeria	DZA	2.83	28.14
American Samoa	ASM	-170.72	-14.30
... and so on for 241 rows			

Table 10.1: Two data tables with information about countries. In order to make a graphic with the demographic information shown on a map, a single glyph-ready table needs to be constructed by joining the information in both tables.

By combining the information in the two tables you can make a plot such as Figure 10.1 showing population density arranged geographically for each country.

Glyph-ready data for the scatter-plot component of Figure 10.1 has the form of Table 10.2. This is a combination of the tables from

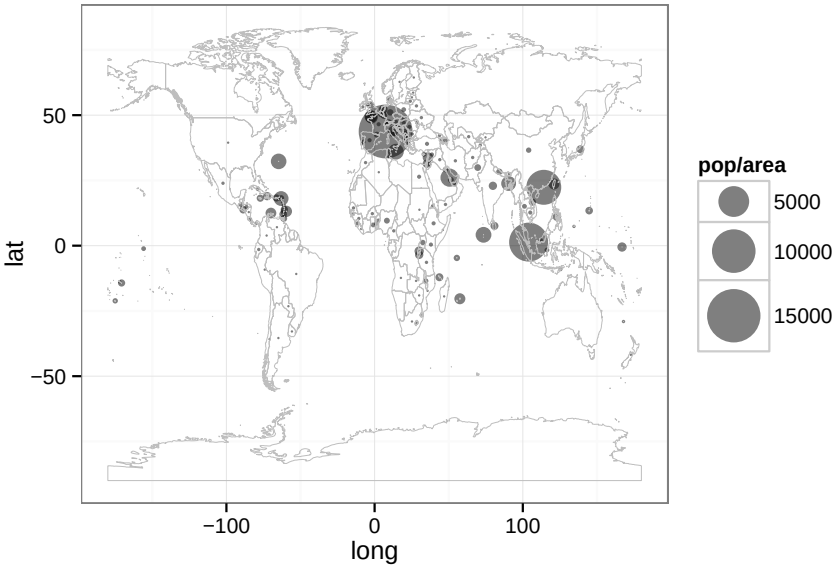


Figure 10.1: Population density shown as dot area. Country borders are a graphics layer produced by `mWorldMap()`.

`Demographics` and `CountryCentroids`. But what kind of combination is it?

country	iso_a3	long	lat	pop	area
Afghanistan	AFG	66.17	33.78	31822848	652230
Albania	ALB	20.26	41.14	3020209	28748
Algeria	DZA	2.83	28.14	38813722	2381741
American Samoa	ASM	-170.72	-14.30	54517	199
Andorra	AND	1.53	42.53	85458	468
		... and so on for 188 rows			

Table 10.2: Glyph-ready data for Figure 10.1.

Note first that `Demographics` and `CountryCentroids` have different numbers of rows, 256 and 241 respectively. Also note that the second row in `Demographics` is a different case entirely from the second row in `CountryCentroids`. So the glyph-ready data is not simply a cut-and-paste job of the `long` and `lat` variables of `CountryCentroids` as new variables conjoined to `Demographics`.

In combining `Demographics` and `CountryCentroids` to construct the glyph-ready data, a match is made between corresponding rows, even though they be in different positions in the two tables. For instance, row 7 in `Demographics` has been matched to row 4 in `CountryCentroids`. Of course, what these two rows have in common is that they both refer to Angola.

To *join* is a data verb that looks for information to match rows in one table to zero or more rows in a second table. Conventionally, the two tables are referred to as the *left* and the *right* table. The R/dplyr

JOIN: A data verb that finds matching rows in each of two data tables — referred to as the left and the right tables — and constructs an output that combines the data from the two tables accordingly.

syntax is:

```
LeftTable %>%
  joinOperation(RightTable, vars_for_matching)
```

10.1 Variables for Matching

What does it mean for a case in the Left table to match a case in the Right table? A match is defined in terms of a set of variables that appear in both the Left and the Right. When a case in the Left has values for those variables that are identical to the values for those variables in the Right, there is a match between the Left case and the Right case.

Table 10.3 displays data on human migration between countries from `DataComputing::MigrationFlows`. Each case in `MigrationFlows` registers migration from an *origin* country to a *destination* country.

Suppose your interest in these data is to understand the forces that shape migration. Variables from `DataComputing::CountryData`, could be useful here. Table 10.4 selects a few health measures for each country: life expectancy, infant mortality, and fraction of the population considered underweight.

Consolidating the health-related data to the migration data is done with a join. For instance, suppose you want to bring side-by-side the health data for the "origin" country and the migration data. To do this, you'll have to match the `origincode` variable in `MigrationFlows` to the `country` variable in `CountryData`. It's easy enough to do this for those countries whose code you can figure out, e.g. `FRA` for France, `AFG` for Afghanistan, etc. Common sense tells you to look up those codes you don't already know, e.g. `AUT` for Austria and `AUS` for Australia.

Data-wrangling software will construct a match only when the information is in exactly the same form. So, while `AUT` is equivalent to `Austria`, you need to demonstrate this equivalence. In other words, you'll need to find or construct a data table that contains the translation from country code to country name. This sometimes requires creativity; it certainly requires a knowledge of what data are available. For instance, the `DataComputing::CountryCentroids` data table (Table 10.5) lists *both* the country and the country code:

```
CountryCentroids %>%
  select(name, iso_a3)
```

To translate the `country` variable in `CountryData` to the `iso_a3` code, join `CountryCentroids` to `CountryData`, producing 10.6.

destcode	origincode	Y2000
FRA	AFG	923
FRA	DZA	425229
FRA	AUS	9168
... and so on for 107,184 rows		

Table 10.3: The first several cases in the `MigrationFlows` data table. (Not all variables are shown.) Each case is about migration from one country to another: the "origin" and "destination" countries. Country names are given using the three-letter ISO-A3 code. For instance, the migration to France (FRA) from Algeria (DZA) is conspicuously large.

country	life	infant
Afghanistan	50.49	117.23
Akrotiri	NA	NA
Albania	77.96	13.19
Algeria	76.39	21.76
American Samoa	74.91	8.92
... and so on for 256 rows		

Table 10.4: `CountryData`: Health-related data from the CIA World Factbook data. Note that the country is identified by its common name.

name	iso_a3
Afghanistan	AFG
Aland	ALA
Albania	ALB
Algeria	DZA
... and so on for 241 rows	

Table 10.5: Two variables selected from `CountryCentroids`

```
InfantMortality <-
  CountryCentroids %>%
  select(name, iso_a3) %>%
  left_join(CountryData %>% select(country, infant),
            by = c("name" = "country"))
```

name	iso_a3	infant
Afghanistan	AFG	117.23
Aland	ALA	NA
Albania	ALB	13.19
Algeria	DZA	21.76
... and so on for 241 rows		

Table 10.6: `InfantMortality` table produced by joining `CountryCentroids` to `CountryData`.

Note that the `by` argument to `left_join()` specifies that the case matching is to be accomplished by comparing the `country` variable in the Right table to the `name` variable in the Left table. Now you are in a position to join the infant mortality data to the migration data. The variables for matching will be `country` in `InfantMortality` and `destcode` in `MigrationFlows`:

```
MigrationFlows %>%
  left_join(InfantMortality, by = c("destcode" = "iso_a3"))
```

sex	destcode	origincode	Y2000	name	infant
Male	FRA	AFG	923	France	3.31
Male	FRA	DZA	425229	France	3.31
Male	FRA	AUS	9168	France	3.31
Male	FRA	AUT	7764	France	3.31
Male	FRA	AZE	118	France	3.31
... and so on for 107,184 rows					

Often, the tables you are joining have the same name for corresponding variables. A *natural join* defines the matching cases by *all* the same-named pairs of variables. In other situations, the corresponding variables may have different names, or you may not want to join based on all the same-named pairs. In these situations, you can explicity identify the corresponding variables with the `by` argument.

NATURAL JOIN: The selection of variables that match the cases in one table to another based simply on variables with the same name that appear in both tables.

Or, you could bring in infant mortality for the origin country:

```
MigrationFlows %>%
  left_join(InfantMortality, by=c("origincode"="iso_a3"))
```

sex	destcode	origincode	Y2000	Y1990	Y1980	Y1970	Y1960	name	infant
Male	FRA	AFG	923	91	55	29	1471	Afghanistan	117.23
Male	FRA	DZA	425229	861691	794288	723746	521679	Algeria	21.76
Male	FRA	AUS	9168	903	1483	1906	14614	Australia	4.43
Male	FRA	AUT	7764	2761	4686	4861	12375	Austria	4.16
Male	FRA	AZE	118	12	20	4	188	Azerbaijan	26.67
... and so on for 107,184 rows									

10.2 Different Types of Join

You've seen the role of matching variables in a join operation. When there is a match between a case in the Left table and a case in the Right table, the remaining variables from the Right table are concatenated onto the case from the Left table.

Some questions remain:

- What happens when a case in the Right table has no matches in the Left?
- What happens when a case in the Left table has no matches in the Right?
- What happens when there are multiple cases in the Left that match a case in the Right?

There are different answers that are appropriate in different situations. For that reason, there are a variety of join operators that work in different ways.

- `left_join()` — the output has *all* the cases from the Left, even if there is no match in the right.
- `inner_join()` — the output has *only* the cases from the Left with a match in the Right.
- `full_join()` — the output will have all the cases from both the Left and the Right.

All of these join operators behave the same when there are multiple matches in the Right table for a case in the Left table: Each of these multiple matches will produce a case in the output.

10.3 Common Uses for Joins

Mutating

Mutating means to add new variables to a data table. The `mutate()` data verb lets you add new variables constructed by transformation operations on existing variables. The join verbs can do something similar: constructing new variables by pulling them out of a different data table.

Translating

One common sort of mutation deserves special attention. Often, you will need to translate levels in one variable into a different set of levels. You've already seen this in the `MigrationFlows` and `CountryData`

task; `Migration` data had countries listed in ISO-code form, whereas `CountryData` has countries listed by name. The translation from one form to another happened to be described by the `CountriesCentroid` data.

Filtering

You can use the `filter()` function to extract those cases that meet a given criterion. Sometimes you may have a list of the cases you want. `inner_join()` lets you choose the cases you want in that list. For example, Table 10.7 lists the G8 countries. To extract the G8 cases from another table, use an inner join between that table and G8 country table

```
G8Countries <-
  CountryGroups %>%
  filter( G8 ) %>%
  select( country )


G8CountryData <-
  CountryData %>%
  inner_join(G8Countries)
```

country
Canada
France
Germany
Italy
Japan
... and so on for 8 rows

Table 10.7: The G8 countries.

country	GDP	pop
Canada	1.518e+12	34834841
France	2.276e+12	66259012
Germany	3.227e+12	80996685
Italy	1.805e+12	61680122
Japan	4.729e+12	127103388
... and so on for 8 rows		

Table 10.8: Information about G8 countries extracted using an inner join.

10.4 Exercises

Problem 10.1
Most data verbs, when used with the chaining syntax `%>%`, have arguments that consist only of reduction and transformation functions, constants, and variables. For instance:

```
BabyNames %>%
  group_by(year) %>%
  summarise(total = sum(count))
```

In contrast, the join family of data verbs — `inner_join()`, `left_join()`, etc. — always have a data table as one of the arguments inside the parentheses. Explain why.

Problem 10.2
Consider these two tables containing demographic and geographic information about countries.

`Demographics`

country	pop	area
Afghanistan	31822848	652230
Akrotiri	15700	123
Albania	3020209	28748
Algeria	38813722	2381741
American Samoa	54517	199
... and so on for 256 rows		

`CountryCentroids`

name	iso_a3	long	lat
Afghanistan	AFG	66.17	33.78
Aland	ALA	19.97	60.20
Albania	ALB	20.26	41.14
Algeria	DZA	2.83	28.14
American Samoa	ASM	-170.72	-14.30
... and so on for 241 rows			

Explain why the information in the two tables cannot be successfully combined by laying the two tables side by side into a single table, that is, by simply copying the `long` and `lat` variables from one table and pasting them alongside the `country`, `pop` and `area` variables in the other table.

Problem 10.3
Here are three tables, **A**, **B**, and **C**, with different organizations of the same data:

Data Table A

Year	Algeria	Brazil	Columbia
2000	7	12	16
2001	9	14	18

Data Table B

Country	Y2000	Y2001
Algeria	7	9
Brazil	12	14
Columbia	16	18

Data Table C

Country	Year	Value
Algeria	2000	7
Algeria	2001	9
Brazil	2000	12
Columbia	2001	18
Columbia	2000	16
Brazil	2001	14

1. Which table format do you think would make it easiest to find the change from 2000 to 2001 for each country. How would you do it?
2. Suppose you have another table, `ContinentData`, which gives the continent that each country is in. Which table format do you think would make it easiest to find the sum of the values for each continent for each of the years? How would you do it?

11

Wide versus Narrow Data Layouts

Each row of a data table is an individual case. Often it's useful to re-organize the same data in a different way with different meanings to a case. This can make it easier to perform wrangling tasks such as comparisons, joins, and the inclusion of new data.

Consider the format of `BP_wide` shown in Table 11.1, where the case is a research study subject and there are separate variables for the measurement of systolic blood pressure (SBP) before and after exposure to a stressful environment.

Exactly the same data can be presented in the format of the `BP_narrow` data table (Table 11.2), where the case is an individual occasion for blood-pressure measurement.

Each of the formats `BP_wide` and `BP_narrow` has its advantages and its deficits. For example, it's easy to find the before-and-after change in blood pressure using `BP_wide`:

```
BP_wide %>%
  mutate(change = after - before)
```

On the other hand, a narrow format is more flexible for including additional variables, for example the date of the measurement or the diastolic blood pressure as in Table 11.3. Narrow format also makes it feasible to add in additional measurement occasions. For instance, Table 11.3 shows several "after" measurements for subject WJC. (Such "repeated measures" are a common feature of scientific studies.)

subject	before	after
BHO	120	160
GWB	115	135
WJC	105	145

Table 11.1: `BP_wide`: A wide format

subject	when	sbp
BHO	before	160
GWB	before	115
WJC	after	145
GWB	after	135
WJC	before	105
BHO	after	160

Table 11.2: `BP_narrow`: A narrow format

subject	when	sbp	dbp	date
BHO	before	160	69	2007-06-19
GWB	before	115	54	1998-04-21
BHO	before	155	65	2005-11-08
WJC	after	145	75	2002-11-15
WJC	after	NA	65	2010-03-26
WJC	after	130	60	2013-09-15
GWB	after	135	NA	2009-05-08
WJC	before	105	60	1990-08-17
BHO	after	160	78	2017-06-04

Table 11.3: A data table extending the information in Tables 11.2 and 11.1 to include additional variables and repeated measurements. The narrow format facilitates including new cases or variables.

A simple strategy allows you to get the benefits of either format: convert from wide to narrow or from narrow to wide as suits your purpose.

11.1 *Data verbs for converting wide to narrow and* vice versa

Transforming from wide to narrow is the action of a data verb: a wide data table is the input and a narrow data table is the output. The reverse task, transforming from narrow to wide, involves another data verb. These notes use *spread* and *gather.* They are implemented by `spread()` and `gather()` in the `tidyr` package.

Different authors use different names for these verbs: e.g., casting versus melting, stacking versus unstacking, spread versus gather.

`tidyr` is a close relation to `dplyr`, the main data wrangling package used in this book. Both were created by Hadley Wickham.

Spreading

The `spread()` function converts from narrow to wide. Carrying out this operation involves specifying some information in the arguments to the function. The *value* is the variable in the narrow format which is to be divided up into multiple variables in the resulting wide format. The *key* is the name of the variable in the narrow format that identifies for each case individually which column in the wide format will receive the value.

For instance, in the narrow form `BP_narrow` (Table 11.2) the *value* variable is `sbp`. In the corresponding wide form, `BP_wide` (Table 11.1), the information in `sbp` will be spread between two variables: `before` and `after`. The *key* variable in `BP_narrow` is `when`. Note that the different categorical levels in `when` set which variable in `BP_wide` will be the destination for the `sbp` value of each case.

Only the *key* and *value* variables are involved in the transformation from narrow to wide. Other variables in the narrow table, such as `subject` in `BP_narrow`, are used to define the cases.

Translating `BP_narrow` to `BP_wide` can be done like this:

```
BP_narrow %>%
  spread(key = when, value = sbp)
```

Gathering

Now consider how to transform `BP_wide` into `BP_narrow`. The names of the variables to be gathered together, `before` and `after`, will become the categorical levels in the narrow form. That is, they will make up the *key* variable in the narrow form. The data analyst has to invent a name for this variable. There are all sorts of sensible possibilities, for instance `before_or_after`. In gathering `BP_wide` into `BP_narrow`, the concise variable name `when` was chosen.

Similarly, a name must be invented for the variable that is to hold the values in the variables being gathered. Again, there are many reasonable possibilities. It's sensible to choose a name that reflects the kind of thing those value are, in this case systolic blood pressure. So, `sbp` is a good choice.

Finally, the analyst needs to specify which are the variables to be gathered. For instance, it hardly makes sense to gather `subject` with the other variables; it will remain as a separate variable in the narrow result. Values in `subject` will be repeated as necessary to give each case in the narrow format its own correct value of `subject`.

Here's how to convert `BP_wide` into `BP_narrow`:

```
BP_wide %>%
  gather( key=when, value=sbp, before, after )
```

Again, the names of the key and value arguments are given as arguments. These are the names invented by the data analyst; those names are not part of the wide input to `gather()`. Arguments after the key and value are the names of the variables to be gathered.

11.2 *Example: Gender-neutral names*

In "A Boy Named Sue", country singer Johnny Cash famously told the story of a boy toughened in life by being given a girl's name. Indeed, Sue is given to about 300 times as many girls as boys.

```
BabyNames %>%
  filter(name == "Sue") %>%
  group_by(name, sex) %>%
  summarise(total = sum(count))
```

name	sex	total
Sue	F	144410
Sue	M	519

Table 11.5: 99.6% of Sues are female.

```
BabyNames %>%
  filter(name == "Robin") %>%
  group_by(name, sex) %>%
  summarise(total = sum(count))
```

name	sex	total
Robin	F	288396
Robin	M	43829

Table 11.6: 15% of Robins are male.

On the other hand, Batman's partner, Robin, had no such problems. About 15% of Robins are male.

This technique works well if you want to look at gender balance in one name at a time, but suppose you want to find the most gender-neutral names from all 92,600 names in `BabyNames`? For this, it would be useful to have the results in a wide format, like Table 11.7.

To start, add up the counts in `BabyNames` over the years and convert to a wide format as in Table 11.7. `sex` is the key variable in the conversion.

name	F	M
Leslie	263049	112463
Robin	288396	43829
Sue	144410	519
... and so on for 92,600 rows		

Table 11.7: A wide format facilitates examining gender balance in `BabyNames`.

```
BabyWide <-
  BabyNames %>%
  group_by(sex, name) %>%
  summarise(total = sum(count)) %>%
  spread(key=sex, value = total, fill = 0)
```

A fill of zero is appropriate here: for a name like Ab or Abe, where there are no females, the entry for F should be zero.

One way to define "approximately the same" is to take the smaller of the ratios M/F and F/M. If females greatly outnumber males, then F/M will be large, but M/F will be small. If the sexes are about equal, then both ratios will be near 1. The smaller will never be greater than 1, so the most balanced names are those with the smaller of the ratios near 1.

Here's a statement to find the most balanced gender-neutral names out of the names with more than 50,000 babies of each sex. Remember, a ratio of 1 means exactly balanced; a ratio of 0.5 means two to one in favor of one sex; 0.33 means three to one. (The `pmin()` transformation function returns the smaller of the two arguments for each individual case.)

```
BabyWide %>%
  filter(M > 50000, F > 50000) %>%
  mutate(ratio = pmin(M / F, F / M) ) %>%
  arrange(desc(ratio))
```

name	F	M	ratio
Riley	76811	85039	0.90
Jackie	90217	78061	0.87
Casey	74699	108072	0.69
Jessie	165106	109031	0.66
Angel	90531	203376	0.45
Leslie	263049	112463	0.43
Marion	187647	71662	0.38
Jordan	125588	345635	0.36
Taylor	304305	107264	0.35
Willie	146124	447403	0.33
... and so on for 19 rows			

Table 11.8: The most gender-balanced common names

11.3 Exercises

Problem 11.1

Here are three data tables with the same information:

Version One

name	sex	year	nbabies
Harrison	F	2012	15
Harrison	M	1912	170
Harrison	M	2012	2120
Roderick	M	1912	46
Roderick	M	2012	202
... and so on for 9 rows			

Version Two

name	year	F	M
Harrison	1912	NA	170
Harrison	2012	15	2120
Roderick	1912	NA	46
Roderick	2012	NA	202
Terry	1912	17	49
... and so on for 6 rows			

Version Three

name	sex	1912	2012
Harrison	F	NA	15
Harrison	M	170	2120
Roderick	M	46	202
Terry	F	17	17
Terry	M	49	479

a. What is the meaning of a case in each of the tables?
   - Version One
   - Version Two
   - Version Three

b. Comparing Version One to Version Two, which table is narrow and which one is wide?

c. What "key" variable from the narrow table is being used?

d. There are no `NA`s in Version One, but there are in Versions Two and Three. Why?

e. Version Two has 6 cases, while Version 3 has only 5 cases. How can they contain the same information?

f. Version Three was "spread" from Version One. What variable was used to denote the spread columns?

g. Version One can be created by gathering columns from Version Two.
   - Which variables from Two were gathered into One?
   - What "key" variable, not explicitly named in Version Two, does appear in Version One?
   - Where were the values taken from Version Two to use as levels in the key variable created for Version One?

Problem 11.2

- Suppose you want to create the following table with the most popular name of either sex each year

name	sex	year	nbabies
Harrison	F	2012	15
Roderick	M	1912	46
Roderick	M	2012	202
Terry	F	1912	17

What should the chain of commands look like to make this from the data table "Version One" in the previous exercise?

- Suppose you want to calculate the ratio of male to female in each name in each year. Like this:

name	year	ratio
Harrison	1912	NA
Harrison	2012	0.01
Roderick	1912	NA
Roderick	2012	NA
Terry	1912	0.35
... and so on for 6 rows		

- Would you rather start from "Version Two" or "Version Three"?
- If you were given "Version One", would you rather work directly on that with the data verbs or, first, translate to one of the other forms?

Problem 11.3

Comparing each of the following *pairs* of tables, say which one is wide and which one is narrow. a. **A** versus **C** b. **B** versus **C** c. **A** versus **C**

Data Table A**

Year	Algeria	Brazil	Columbia
2000	7	12	16
2001	9	14	18

Data Table B

Country	Y2000	Y2001
Algeria	7	9
Brazil	12	14
Columbia	16	18

Data Table C

Country	Year	Value
Algeria	2000	7
Algeria	2001	9
Brazil	2000	12
Columbia	2001	18
Columbia	2000	16
Brazil	2001	14

Problem 11.4

Consider the data table `BP_wide` in Table 11.1. Using paper and pencil, not the computer, sketch out what will be the result of this (unfortunate) conversion from wide to narrow:

```
BP_wide %>%
  gather(key = when, value = sbp,
         subject, before, after)
```

Hint: The name `when` for the key, and `sbp` for the value don't reflect what's actually in the result.

Problem 11.5

Here are two tables containing information relevant to Table 11.3 in the text.

`Measurements`

subject	what	value	date
BHO	sbp	160	2007-06-19
GWB	sbp	115	1998-04-21
BHO	sbp	155	2005-11-08
WJC	sbp	145	2002-11-15
WJC	sbp	130	2013-09-15
... and so on for 16 rows			

`Treatments`

subject	treatment_date
BHO	2012-08-05
GWB	2005-11-14
WJC	1998-09-30

These are concise forms for organizing the data with several advantages:

1. The use of `treatment_date` avoids possible areas when converting the measurement date to "before" or "after".
2. Additional measurements (e.g. respiration rate, white blood cell count, ...) can easily be included.
3. Addition information about each measurement (e.g. the name of the technician performing the measurement) can be easily included.

You can access the complete tables with this statement, which will create `Measurements` and `Treatments`.

```
"http://tiny.cc/dcf/MeasTreatTables.rda" %>%
  url() %>% load()
```

The `when` variable in Table 11.3 describes whether the measurement date was before or after the treatment date.

Your task: Using the `Measurements` and `Treatment` tables, reconstruct Table 11.3. (Hint: the dates in both `Measurements` and `Treatments` are stored in a way that you can use the numerical comparison function > to determine whether one date is after another.)

12
Ranks and Ordering

Many questions take forms such as these:

- "Find the largest . . . "
- "Find the three largest . . . "
- "Find the smallest within each group . . . "

The functions `min()` and `max()` are reasonable candidates for carrying out such tasks, but they are limited. For instance, to find the name whose yearly count was the highest:

```
BabyNames %>%
  filter(count == max(count))
```

name	sex	count	year
Linda	F	99674	1947

`Filter()` needs a criterion. The criterion `count == max( count )` (with the double equals sign ==) passes through the case where the value of `count` matches the largest value of `count`. That will be the biggest case.

It's also possible to ask for the cases that are almost as popular as the biggest, e.g. at least 90% as popular.

```
BabyNames %>%
  filter(count > 0.90 * max(count))
```

name	sex	count	year
Linda	F	99674	1947
James	M	94758	1947
Robert	M	91652	1947
Linda	F	96210	1948
Linda	F	90994	1949
... and so on for 8 rows			

Frequently the question will be framed in terms of the n biggest or smallest values, not as a fraction of the largest. To perform such tasks, a new transformation verb is helpful: `rank()`. The `rank()` function does something simple but powerful: it replaces each number in a set with where that number stands with respect to the others. For instance, look at the tiny data table `Set` shown in Table 12.1. What's the rank of the number 5 in the `numbers` variable.

`Set`

numbers
2
5
4
7
2
9
9
8

Table 12.1: `Set`, a data table with one variable.

The rank of the number 5 in the set is four. Why? 5 is the fourth smallest number in the set. (Three of the numbers in the set 2, 2, 4 are smaller than 5.) The smallest number in a set of n numbers will

have rank 1; the largest will have rank n.[1] Table 12.2 shows the rank of each number in the set:

[1] Unless there are ties.

```
Set %>% mutate(the_rank = rank(numbers))
```

numbers	the_rank
2	1.50
5	4.00
4	3.00
7	5.00
2	1.50
9	7.50
9	7.50
8	6.00

Table 12.2: The ranks of the values in `numbers`.

Note that the two 2's have the same rank, as do the two 9's. They're tied.

Suppose you want to find the 3rd most popular name of all time. Use `rank()`.

```
BabyNames %>%
  group_by(name) %>%
  summarise(total = sum(count)) %>%
  filter(rank(desc(total)) == 3)
```

name	total
Robert	4809858

Table 12.3: The third most popular name of all time

Or, to find the top three most popular names, replace == in the above by <=.

```
BabyNames %>%
  group_by(name) %>%
  summarise(total=sum(count)) %>%
  filter( rank( desc(total) ) <= 3)
```

name	total
James	5114325
John	5095590
Robert	4809858

Table 12.4: The all-time top three most popular names in `BabyNames`.

When applied to grouped data, `rank()` will be calculated separately within each group. That is, the rank of a value will be with respect to the other cases in that group. For instance, here's the third most popular name each year.

```
BabyNames %>%
  group_by( year ) %>%
  filter( rank( desc(count) ) == 3)
```

name	sex	count	year
Mary	F	7065	1880
Mary	F	6919	1881
Mary	F	8148	1882
Mary	F	8012	1883
William	M	8897	1884
... and so on for 134 rows			

Table 12.5: The third most popular name in each year

SOMETIMES, TWO OR MORE NUMBERS ARE TIED IN RANK. The `rank()` function deals with these by assigning all the tied values the same rank, which is the mean of the ranks those values would have had if they were even slightly different. There are other rank-like transformation verbs that handle ties differently. For instance, `row_number()` breaks ties in favor of the first case encountered. (See Table 12.6.)

```
Set %>%
  mutate(the_rank = rank(numbers),
         ties_broken = row_number(numbers))
```

numbers	the_rank	ties_broken
2	1.50	1
5	4.00	4
4	3.00	3
7	5.00	5
2	1.50	2
9	7.50	7
9	7.50	8
8	6.00	6

Table 12.6: `row_number()` breaks ties in rank.

12.1 Exercises

Problem 12.1
For each sex, find the 5 most popular names in `BabyNames` adding up over all the years.

Problem 12.2
Using `BabyNames`, for each year, find the fraction of all babies born in that year who were given a name in the top 100 for that year. Make a graph showing how this fraction has changed over the years.

To start, produce a data table that looks like this:

year	frac_in_top_100	total
1880	FALSE	68667
1880	TRUE	132817
1881	FALSE	66340
1881	TRUE	126360
1882	FALSE	77258
... *and so on for 268 rows*		

Then you can use `spread()` and `mutate()` to find the fraction of babies with names in the top 100 each year.

```
GlyphReady <-
  PopularCounts %>%
  spread(frac_in_top_100, total) %>%
  mutate(frac_in_top_100 = `TRUE` / (`TRUE` + `FALSE`))
```

Problem 12.3
For each of the operations listed here, say whether it involves a transformation function or a summary function or neither.

a. Determine the 3rd largest.
b. Determine the 3rd and 4th largest values.
c. Determine the number of cases.
d. Determine whether a year is a leap year.
e. Determine whether a date is a legal holiday.
f. Determine the range of a set, that is, the max minus the min.
g. Determine which day of the week (e.g., Sun, Mon, ...) a given date is.
h. Find the time interval in days spanned by a set of dates.

13
Networks

13.1 Vertices and edges

A network is a set of connections between objects. A network combines information about two different kinds of things:

1. The objects themselves, generally called *vertices*.
2. The connections — called *edges* — between pairs of vertices.

VERTICES: The objects that are being connected in a network. Sometimes called "nodes"

EDGE: The connections between pairs of vertices in a network

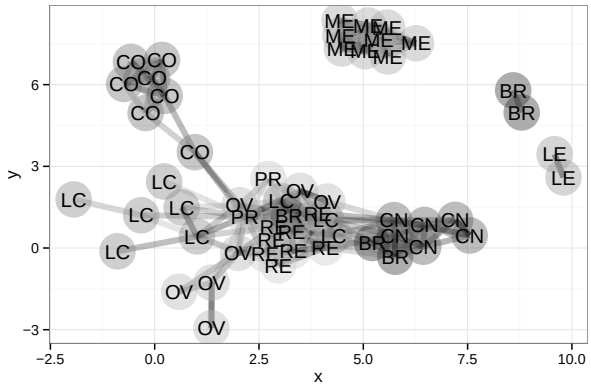


Figure 13.1: A network of relationships between the genes expressed in cancer cells from different people. (Reproduced from Figure 5.6.)

In the previous examples in this book, the end-result of data wrangling has been a single, glyph-ready data table. With networks, things are different. A network always involves two different kinds of things — vertices and edges. Accordingly, representing a network with data always involves two different data tables:

- A data table for the vertices, giving the properties of each vertex. In Figure 13.1 the vertices are samples of cancer cells and are shown as large dots with colors and labels identifying the type of cancer. A data table representing the vertices might look like Table 13.1.

ID	cancerType
TK_10	renal
MDA_MB_35	melanoma
MCF7	breast
... and so on for 60 rows	

Table 13.1: A data table representing the vertices in Figure 13.1.

- A different data table for the edges. Each case in this table has at least two attributes: the origin vertex of the edge and the des-

tination vertex of the edge. The data table might look like Table 13.2.

The following sections show some of the ways that data wrangling can be used in determining what are the edges and what determines the *location* of each vertex in the graphic.

origin	dest
MCF7	LOXIMV1
MCF7	UO_31
LOXIMVI	HL_60
SK_MEL_5	NCI_H226
SK_OV_3	LOXIMV1
... and so on for 241 rows	

Table 13.2: A data table representing the edges in Figure 13.1.

13.2 *Example: Migration Networks*

The `MigrationFlows` data table presents how many people migrate between one country and another. Table 13.3 shows `Bilateral`, a wrangled form of `MigrationFlows` listing how many women migrated from one country to another and *vice versa*.

CountryA	CountryB	BtoA	AtoB
POL	DEU	2961	1112903
PAK	IND	2923092	3995350
GBR	CAN	32687	487980
UKR	POL	198293	634186
ITA	ARG	232	407123
... and so on for 53,592 rows			

Table 13.3: `Bilateral`, which shows emigration and immigration as different variables.

`Bilateral` describes edges: the links between countries. In a data table describing edges, the case will be an edge. The minimal information needed to describe an edge is the identities of the two vertices connected by the edge.

To describe the vertices, a different data table is required, one where each case is one vertex. The vertex data table can contain other information about each vertex. For countries, that information might be the population or land area or GPD of each country. The glyph-ready vertex data table should also describe where each vertex is to be located on the graph. `CountryCentroids` is a vertex data table, giving the latitude and longitude of each country.

To show the vertices graphically, the information in the vertex data table can be used to generate a scatter plot like Figure 13.2.

`Bilateral` lists the edges, which can be drawn as line segments. But `Bilateral` is not glyph-ready for drawing the edges. Drawing an edge requires four location attributes: x and y of the vertex at one end of the edge, and x and y of the vertex at the other end. The location of each vertex is contained in the vertex data table, `CountryCentroids`. In order to wrangle `Bilateral` into a glyph-ready form, the location information from `CountryCentroids` is joined to it. This requires two join steps: joining the location of the vertex for `countryA` and similarly joining the location of the vertex for `countryB`. The result is the `Edges` data table (Table 13.4).

`CountryCentroids`

name	iso_a3	long	lat
Afghanistan	AFG	66.17	33.78
Aland	ALA	19.97	60.20
Albania	ALB	20.26	41.14
Algeria	DZA	2.83	28.14
... and so on for 241 rows			

Figure 13.2: Locations of the vertices described by `CountryCentroids`. Another layer shows country boundaries.

CountryA	CountryB	BtoA	AtoB	longA	latA	longB	latB
FRA	DZA	201387	78867	2.72	46.53	2.83	28.14
FRA	AUS	8385	1925	2.72	46.53	134.61	-25.85
FRA	AUT	7100	7460	2.72	46.53	14.32	47.60
FRA	BEN	114	142	2.72	46.53	2.55	9.62
FRA	AND	111	111	2.72	46.53	1.53	42.53
		... and so on for 836 rows					

Table 13.4: `Edges`, a data table based on `Bilateral` wrangled into a form that is glyph-ready for drawing network edges.

Drawing the edges can be done in the usual way. The "segment" geom is an appropriate glyph:

```
ggplot(Edges, aes(x=longA, y=latA, xend=longB, yend=latB)) +
  geom_segment(alpha=0.2)
```

Figure 13.3: The edges in `Bilateral` using latitude and longitude to mark the location of each vertex.

13.3 *Vertex Location from the Edges*

It's perfectly sensible to show the migration network using latitude and longitude for the vertex location. But many networks, for instance the gene expression network in Figure 13.1, have vertices with no natural location. Indeed, even when there is a natural location for each vertex, that might not be the best way to draw the network. Looking at Figure 13.3 critically, you can see that the graphic is dominated by the country pairs that are far apart, e.g. China and the US, Japan and Brazil, the UK and Australia. Migration among European countries, on the other hand, is compressed into a small part of the graphic. There are many crossings where the person viewing the graph can get confused.

Sometimes displays of networks are better when the vertices are placed to put them close to the vertices to which they are linked. Instead of identifying an attribute of the vertices, the location is set by the edges in a way that minimizes crossings or by some other criterion.

It's challenging to figure out the best places for vertices to minimize the edge crossings and to keep connected vertices together. Several different methods have been described in the network-drawing literature. To simplify things, the `DataComputing` package provides two functions, `edgesToVertices()` and `edgesForPlotting()` to do this automatically.

The `igraph` package provides the power behind the location algorithms.

To illustrate, consider a network to display cultural affinity between countries. As a simplistic measure of "cultural affinity," use the balance between inbound and outbound migration between two countries and the number of people migrating. Here's one way:

```
CulturalAffinity <-
  Bilateral %>%
  mutate(balance = pmin(AtoB / BtoA, BtoA / AtoB))
```

`CulturalAffinity` contains information about edges, but it has an edge even for country pairs with unbalanced migration. To show strong cultural affinity, eliminate the country pairs with unbalanced or small migration.

```
BigAffinity <-
  CulturalAffinity %>%
  filter(balance > 0.5, AtoB > 500 | BtoA > 500)
```

There are 158 edges in `BigAffinity`. Use `edgesToVertices()` to position the vertices in a way that minimizes edge crossing.

```
Vertices <-
  BigAffinity %>%
  edgesToVertices(from=CountryA, to=CountryB)
```

ID	x	y
FRA	1.25	2.24
GHA	7.67	1.29
HKG	4.76	3.51
ISL	4.99	-4.40
AZE	11.15	6.19
... and so on for 125 rows		

Table 13.5: Vertex location derived from `CulturalAffinity` using `edgesToVertices()`

Now that the vertex location has been determined, that location can be joined with the edge data in `CulturalAffinity`. This will enable the edges to be drawn connecting the vertices.

```
Edges <-
  Vertices %>%
  edgesForPlotting(ID=ID, x, y,
                   Edges=BigAffinity,
                   from=CountryA, to=CountryB )
```

Finally, a plot of the network:

```
ggplot(Edges, aes(x = x, y = y)) +
  geom_segment(aes(xend = xend, yend = yend, alpha = balance)) +
  geom_point(data=Vertices, size = 5, color = "white", aes(x = x, y = y)) +
  geom_text(data=Vertices, size = 3, aes(x = x, y = y, label=ID))
```

Even though the countries are not presented by geographic location, the network helps to illustrate the patterns of affinity between countries. Great Britain (GBR), the USA, and France (FRA) each are hubs having affinities with several countries. Egypt (EGY) is the center of an Arabic network connecting Saudi Arabia, Yemen, Kuwait, Palestine, Syria, and Tunisia. Another cluster centered on Russia (RUS) connects countries formed after the break-up of the Soviet Union.

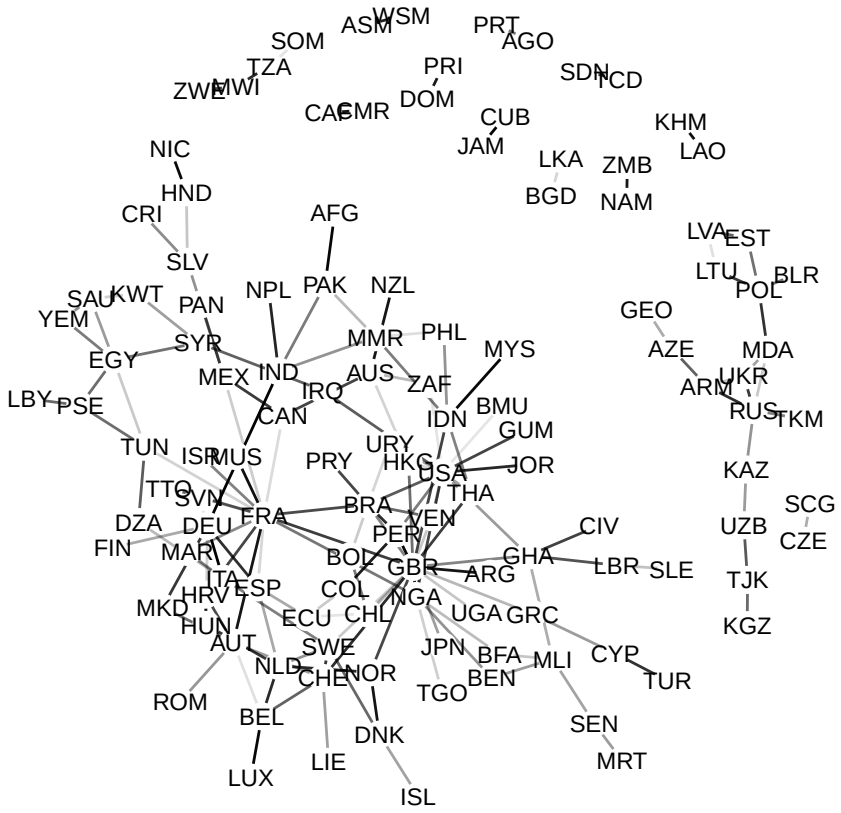


Figure 13.4: Links between countries with balanced two-way migration.

13.4 *Exercises*

Problem 13.1

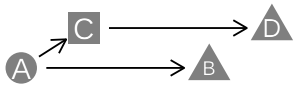
Consider this table of edges:

from	to
China	Japan
USSR	Japan
Germany	USA
France	Germany
Italy	France
Germany	UK
Japan	UK
USA	Japan
USSR	Germany

1. How many distinct vertices are there?
2. How many edges are there?

Problem 13.2

For this network . . .



1. What are the vertices?
2. Which of these tables shows the edges correctly?

Table 1

A	C
B	C
A	B

Table 2

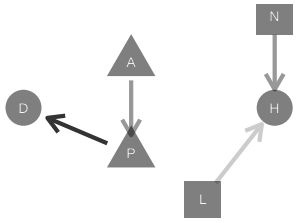
A	C
A	B
C	D

Table 3

A	C	D
A	B	

3. Explain what's wrong with each of the other tables.

Problem 13.3



Which table gives the information needed to draw the edges in the graph?

Table 1

From	To
A	P
H	D
L	H
A	N

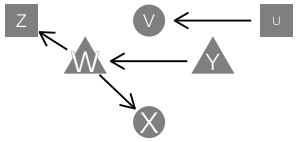
Table 2

From	To
A	P
P	D
L	H
N	H

Table 3

From	To
P	A
P	N
H	L
N	H

Problem 13.4



Here is a table of the vertices in the network above:

country	pop	area	exports	roads
U	3	7	1	3
V	1	4	2	2
W	2	2	4	1
X	1	1	4	3
Y	2	1	3	4
Z	3	3	2	5

1. Which variable is mapped to the size of the letter?
2. Which variable is mapped to the shape of the gray symbol?

14
Collective Properties of Cases

Up to now, the emphasis has been on individual cases. Data graphics, for instance, translate each case into a representation in terms of a glyph's graphical attributes: position, color, size, shape, etc.

There are many occasions where a different emphasis is called for: representing the *collective properties* of cases. You've seen simple collective properties calculated with *summary functions*: mean, median, max, etc. The concern here is detailed collective descriptions, aspects of the collective that cannot be summed up in a single number.

COLLECTIVE PROPERTIES: Features of groups of cases.

14.1 Statistics

Statistics is the area of science concerned with characterizing case-to-case variation and the collective properties of cases. Statistics provides many important insights into the collection and interpretation of data. The new field of "data science" is a collaboration between statistics and computer science. Although this book is mainly about computing, a solid grasp of statistics is an important foundation for effective work with data.

A central topic in statistics is the precision with which quantities can be calculated from data. This is not precision in the sense of the correctness of calculations, but has to do with the randomness that inevitably comes into play when selecting a sample of cases from a larger set of possibilities.

This chapter will touch on some important ideas from statistics, especially "confidence intervals" and "confidence band." The techniques by which computers can learn from data, presented in Chapter 17, have embedded in them considerable statistical theory, although statistics itself is not the subject of that chapter.

The main emphasis of this chapter will be the statistical graphics used to display variation among cases, particularly the distribution of values of one variable or several of variables.

14.2 *Density distribution of a variable*

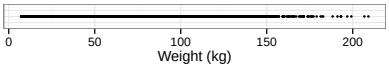


Figure 14.1: A distribution shown as the density of points on a line. This is not effective.

To start, consider the `weight` variable in the `NCHS` data. Of course, weight varies from person to person; there is a distribution of weights from case to case. Sometimes it's helpful to show this distribution, what values are common, what values are rare, what values never show up.

Naïvely, this might be done by drawing a weight axis and putting on it a point for each case, as in Figure 14.1. This graph shows some familiar facts: humans tend not to be lighter than a few kilograms, and tend not to be heavier than about 150 kg. Between these limits, there seems to be no detail.

JITTER: Moving the position of cases by a small random amount, to alleviate overplotting of multiple cases which otherwise would have the same position.

This is misleading, the result of having so many cases (29375) to plot. All those cases lead to ink saturation. One way to overcome this problem is by using transparency, another is to *jitter* the individual points. Figure 14.2 shows the result.

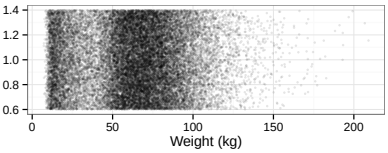


Figure 14.2: The points which were over-crowded onto a line in Figure 14.1 have been spread out: jittered

Jittering produces a much more satisfactory display; the ink saturation has been greatly reduced and there is more detail. For instance, the most common weights are between 55 and 90 kg, and that weights less than 20 kg are also pretty common. There aren't so many people with weights near 30kg, and few over 120 kg.

The visual impression is due to collective properties of the data. Every individual glyph is identical, but for weights where the glyphs have a high *density*, the impression the glyphs collectively give is of darkness. It's easy to see how the density of glyphs varies at different weights. This is called the *density distribution*.

DENSITY: How tightly cases are bunched together.

DENSITY DISTRIBUTION: The variation in density of cases over different values of a variable

The collective property of density is conventionally displayed as a another kind of glyph, as in Figure 14.3, that displays the collective density property using the height of a function instead of the shade of darkness.

```
ggplot(NCHS, aes(x = weight) ) +
  geom_density(color = "gray", fill = "gray", alpha = 0.75) +
  xlab("Weight(kg)") + ylab("Density (1/kg)")
```

In this format, the *height* of the band indicates the density of cases. You can't see individual cases, just the collective property of the density at each weight.

To help you see the translation between the density as vertical position and the density as darkness, Figure 14.4 overlays the two depictions of density in Figures 14.2 and 14.3.

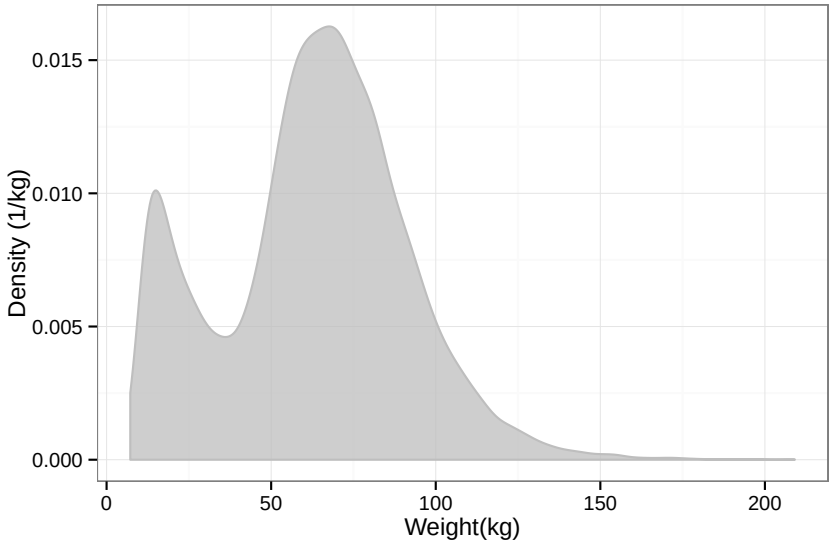


Figure 14.3: A density distribution plot. The scale of the y axis may seem odd; it has units of "inverse kilograms." This scale is arranged so that the area under the entire density curve is 1. This convention facilitates densities for different groups, e.g. male versus female. It also means that narrow distributions tend to have high density.

Graphs of the distribution of density can be used to compare different groups. You need merely indicate which variable should be used for grouping. For instance, Figure 14.5 displays the distribution density of body weight for each sex:

```
ggplot(NCHS, aes(x = weight, group = sex) ) +
  geom_density(aes(color = sex, fill = sex), alpha = 0.5) +
  xlab("Weight (kg)") + ylab("Density (1/kg)")
```

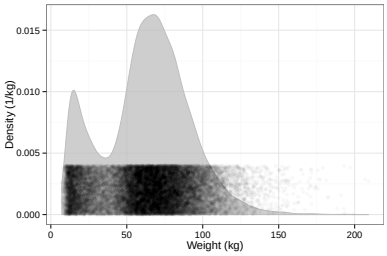


Figure 14.4: The density distribution function is high where the points are dense.

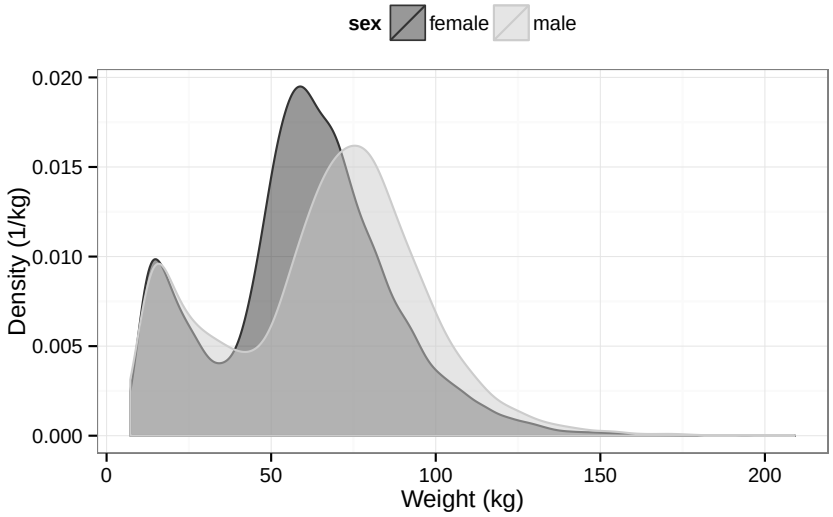


Figure 14.5: Comparing the distributions for body weight between males and females in the NCHS data.

GRAPHS OF DENSITY DISTRIBUTIONS OFTEN HAVE A SIMPLE SHAPE. This allows you to place the distributions side by side for comparison. For instance, suppose you arrange to divide people into age groups.

```
NCHS <-
  NCHS %>%
  mutate( ageGroup=
      mosaic::ntiles(age, n=6, format="interval") )
```

The age group can be used to define facets for a graphic, as in Figure 14.6.

```
NCHS %>%
  ggplot( aes(x=weight, group=sex) ) +
  geom_density(aes(color=sex, fill=sex), alpha=.75) +
  xlab("Weight (kg)") + ylab( "Density (1/kg)") +
  facet_wrap( ~ ageGroup, nrow=1 )
```

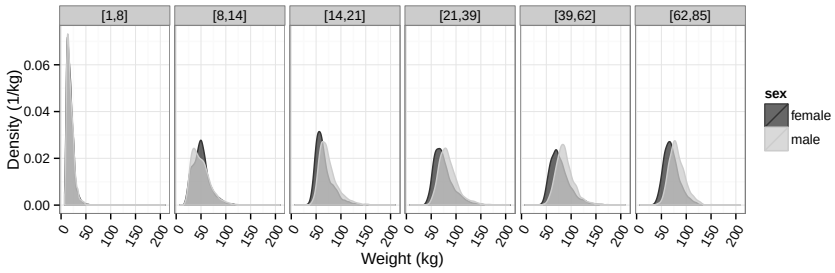


Figure 14.6: Using faceting to compare the weight distribution of different age groups.

Figure 14.6 tells a rich story. For young children the two sexes have almost identical distributions. For kids from 5 to 13 years, the distributions are still similar, though you can see that some girls have grown faster, producing a somewhat higher density near 50 kg. From the teenage years on, the distribution for men is shifted to somewhat higher weight than for women. This shift stays pretty much constant for the rest of life.

14.3 *Other Depictions of Density Distribution*

The detail in the density curves can be visually overwhelming: there's a lot of detail in each density curve. To simplify the graph, sometimes a more stylized depiction of the distribution is helpful. This makes it feasible to put the distributions side-by-side for comparison. (Figure 14.7) The most common such depiction is the *box-and-whisker* glyph.

BOX-AND-WHISKER: A simple presentation of a distribution showing the median, extremes, and intermediate values (25th, 75th percentiles) of a distribution.

The box-and-whisker glyph shows the extent of the middle 50% of the distribution as a box. The whiskers show the range of the top and bottom quarter of the distribution. When there are uncommon, extremely high or low values — called "outliers" — they are displayed as individual dots for each of the outlier cases.

```
NCHS %>%
  ggplot(aes(y = weight, x = ageGroup)) +
  geom_boxplot(
    aes(color = ageGroup, fill = ageGroup),
    alpha = 0.25, outlier.size = 2, outlier.colour="gray") +
  facet_wrap( ~ sex ) +
  ylab("Weight (kg)") + xlab( "Age Group (yrs)")
```

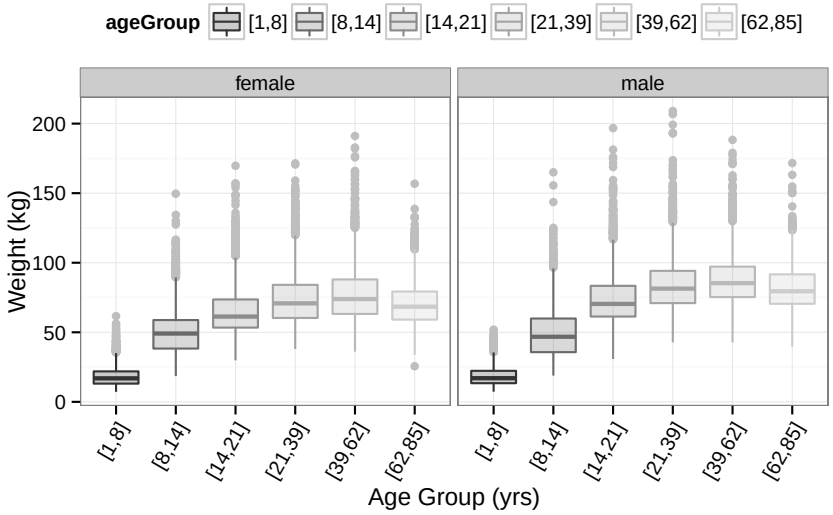


Figure 14.7: Box and whisker glyphs used to show the distribution of weight for different age groups. Faceting by sex.

The *violin glyph* provides a similar ease of comparison to the box-and-whisker glyph, but shows more detail in the density. (Figure 14.8)

VIOLIN GLYPH: Shows distributions like a box and whisker glyph but smoother.

```
NCHS %>%
  ggplot( aes(y=weight, x=ageGroup) ) +
  geom_violin(fill = "gray") +
  facet_wrap( ~ sex ) +
  ylab("Weight (kg)") + xlab( "Age Group (yrs)")
```

Both the box-and-whisker and violin glyphs use ink efficiently. This allows you to show other data for comparison. For instance, Figure 14.9 compares the weight of people with and without diabetics, broken down by age and sex. You can see that diabetics tend to be heavier than non-diabetics, especially for ages 19+.

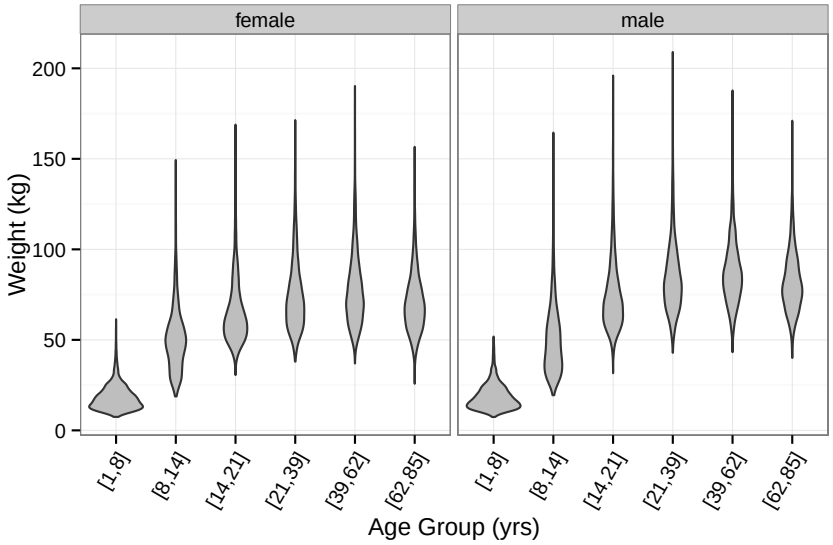


Figure 14.8: A violin plot of the distribution of weights.

```
NCHS %>%
  ggplot(aes(y = weight, x = ageGroup )) +
  geom_boxplot(aes(fill = diabetic),
               position = position_dodge(width=0.8),
               outlier.size=1, outlier.colour="gray", alpha=.75) +
  facet_grid( sex ~ .) +
  ylab("Weight (kg)") + xlab( "Age Group (yrs)")
```

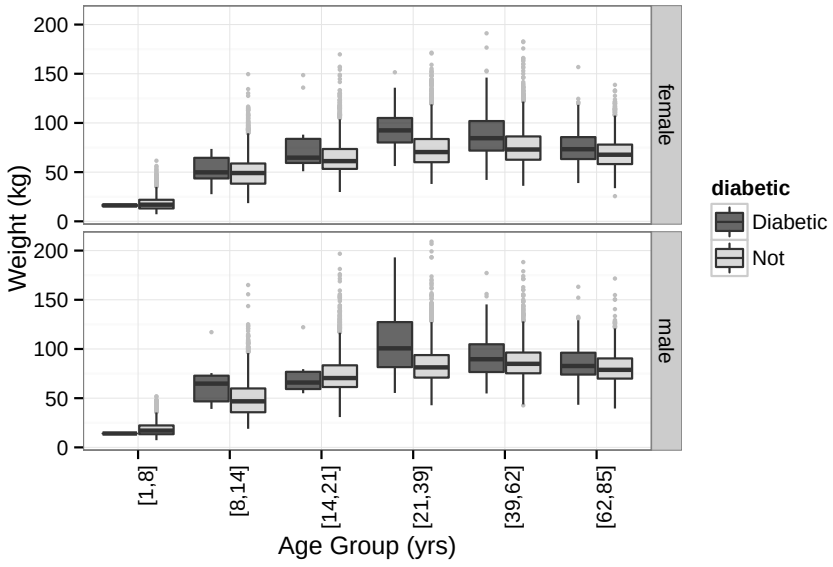


Figure 14.9: Box and whisker plot comparing the weight distribution of diabetics (in red) to the rest of the cases.

14.4 *Confidence Intervals*

Statistical inference refers to an assessment of the *precision* or *strength of evidence* for a claim. For instance, in interpreting the plot above, it was claimed that diabetics tend to be heavier than non-diabetics. Perhaps this is just an accident. Might it be that there are so few diabetics that their weight distribution is not well known?

The *confidence interval* of a statistic tells how much uncertainty there is due to the size of the sample. Smaller-sized samples provide weaker evidence and their confidence intervals are broader than those produced by larger-sized samples.

A box-and-whiskers glyph can show the confidence interval of the median by means of a "notch". (Figure 14.10) Overlap between the notches in two boxes indicates that the evidence for a difference is weak.

```
NCHS %>%
  ggplot(aes(y = weight, x = ageGroup )) +
  geom_boxplot(aes(fill = diabetic),
               notch = TRUE,
               position = position_dodge(width=0.8),
               outlier.size=1, outlier.colour="gray") +
  facet_grid( sex ~ .) +
  ylab("Weight (kg)") + xlab( "Age Group (yrs)")
```

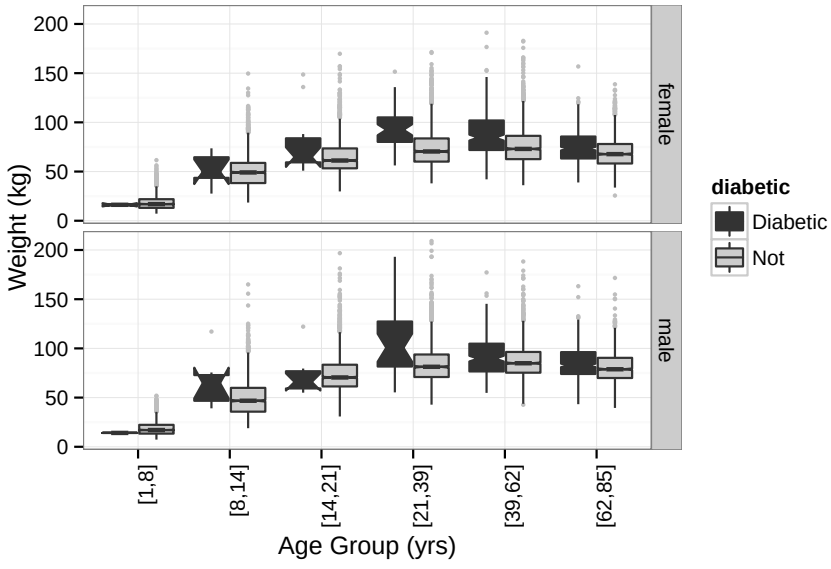


Figure 14.10: A notched box and whisker plot. The notches show the uncertainty in the median weight for each group.

14.5 *Model Functions*

A *function* is one way of describing a relationship between input and output. Often it's helpful to use a function estimated from data.

For example, consider the relationship between weight and age in the NCHS data. Is it the same for diabetics and non-diabetics?

```
NCHS %>%
  ggplot(aes(x = age, y = weight, color = diabetic)) +
  stat_smooth(se = FALSE) + geom_point(alpha = 0.02)
```

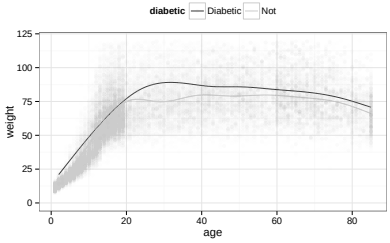


Figure 14.11: Smoothers showing typical weight as a function of age.

There are two functions in Figure 14.11, each calculated by `stat_smooth()` from the age and weight variables. One function is for diabetics, the other for non-diabetics. Each function gives just one weight value for each age, rather than the distribution seen in the raw data. That value is more or less the mean weight at each age. From the functions, it seems that the diabetics are consistently heavier than the non-diabetics, at least on average. At age 30, that average difference is more than 20kg; it is less for people near 80.

On it's face, Figure 14.11 suggests that, for any age, the mean weight of diabetics is higher than that of non-diabetics. Before drawing such a conclusion, however, it's important to know how precisely the functions can be estimated from the data.

The calculation of the functions are exact, but the specific function produced depends on the particular set of cases in the data. Imagine that the original data table consisted of cases that had been randomly selected from all possible cases. Suppose a new set of cases were collected, also randomly selected. The function calculated based on that new set would likely differ *somewhat* from the function calculated from the original data table. The point of a statistical *confidence interval* is to provide a quantitative meaning to "somewhat."

When applying the idea of confidence intervals to a function, one arrives at bands around the function: the *confidence band*s. Confidence bands are constructed so that they *almost always* contain the function that would have been found if the *entire* population were used. In this sense, the confidence bands show how much variation in the shape of the function is due to the random selection of the cases that ended up in the original data table.

The variation that stems from random sampling of cases is so important to interpreting the graphic that the default in `stat_smooth()` is to calculate and show the confidence band unless the user specifically asks otherwise.

CONFIDENCE BAND: A confidence interval around a function

ALMOST ALWAYS: With high probability. The value of this probability, called the "confidence level," is by convention arranged to be 95 percent.

```
NCHS %>%
  ggplot( aes(x=age, y=weight, color=diabetic ) ) +
```

```
stat_smooth( ) +
geom_point(alpha = 0.02)
```

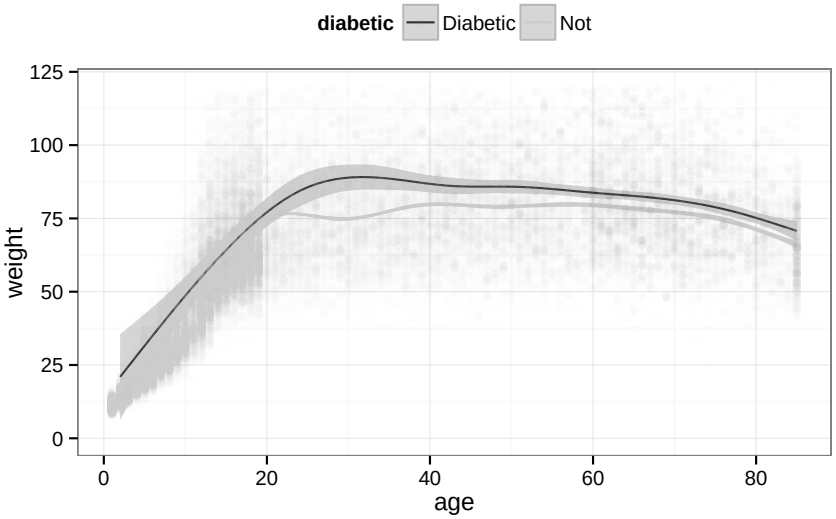


Figure 14.12: The functions in Figure 14.11 with confidence bands drawn around each.

Confidence intervals and bands are widely mis-interpreted. The confidence band does not show the range of cases, it shows the precision with which the statistic or model function is estimated. So it's wrong to say something like 'diabetics are heavier than non-diabetics'. Instead, the statement justified by the separation between the confidence bands is 'the mean weight of diabetics (taking into account age) is greater than that of non-diabetics.' Long winded? Yes. Maybe that's why confidence intervals and bands are so widely mis-interpreted.

The function for non-diabetics has a narrow confidence band; you have to look hard to even see the gray zone. The confidence band for diabetics, however, is noticeably wide. For ages below about 20, the non-diabetic and diabetic confidence bands overlap. This means that there is only weak evidence that diabetics as a group in this age group differ in weight from non-diabetics. Around age 20, the two functions' confidence bands do not overlap. This indicates stronger evidence that the groups in the 20+ age bracket do differ in **average** weight.

You might ask why the confidence band for the diabetics is so much wider than for the non-diabetics? And why is the diabetic confidence band broad in some places and narrow in others?

The width of a confidence band depends on three things:

- the amount of variation in the value of y at or near the x value.
- the number of cases close to the x value
- the extent to which the function is an extrapolation.

The functions in Figures 14.11 and 14.12 are called *smoother*s. A smoother bends with the data collectively, but doesn't try to jump sharply to match individual cases. The smoother is a compromise between matching the data exactly, and remaining smooth. Think of it as a wooden strip that is somewhat stiff, but can still be bent gently.

SMOOTHER: A function that can bend with the data.

A much more commonly used function — the "linear" function — is utterly stiff; it can't bend at all. You can construct this sort of function from the data with `method=lm` as an argument to `stat_smooth()`.

```
NCHS %>%
  ggplot(aes(x = age, y = weight,
             color = diabetic, fill = diabetic )) +
  stat_smooth(method = lm) +
  geom_point(alpha = 0.02)
```

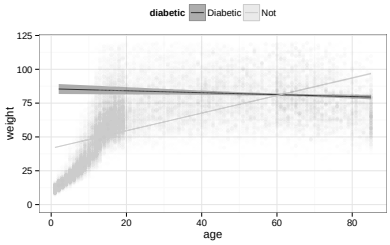


Figure 14.13: A straight-line model is too stiff and misleadingly suggests a large difference in weight for young people between diabetics and non-diabetics.

In this situation, a linear function is too stiff. The function for non-diabetics slopes up at high ages not because older non-diabetics in fact weigh less than middle-aged diabetics, but because the line needs to accommodate both the children (light in weight) and the adults (who are heavier).

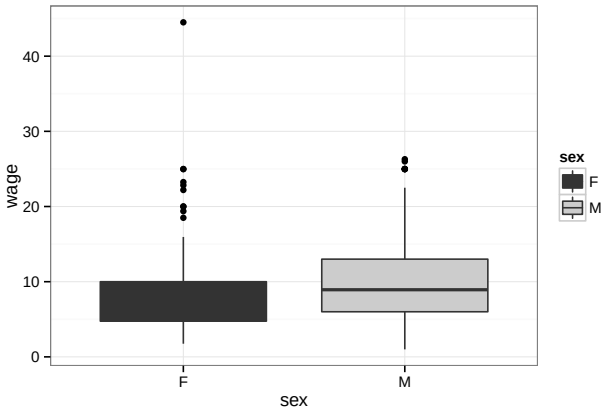
You'll often see linear functions in the scientific literature. This is for several reasons, some good and some bad:

- Linear functions generally have tighter confidence bands than smoothers. (A good reason)
- People are familiar with straight-line functions from high school. (A bad reason)
- Many people don't know about non-linear functions such as a smoother. (A bad reason)

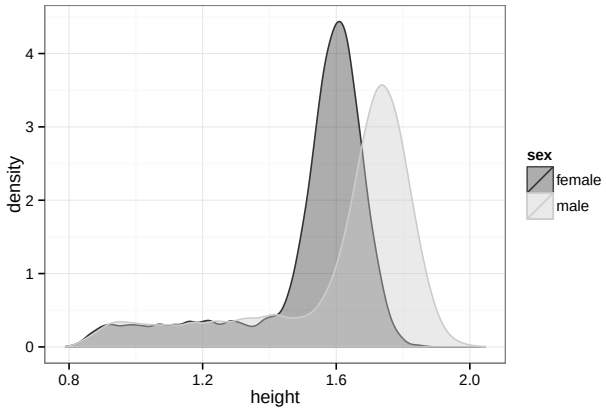
A simple rule of thumb: When in doubt, use a smoother.

14.6 Exercises

Problem 14.1
Reconstruct this graphic using the `ggplot()` system and the `mosaicData::CPS85` data table.

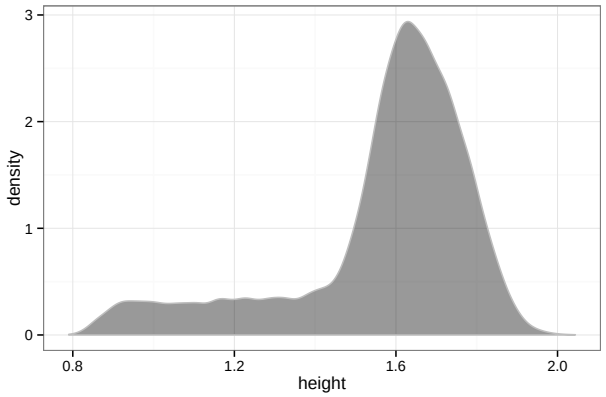


Problem 14.3
Based on the following graph, what's the most likely height for women? For men?

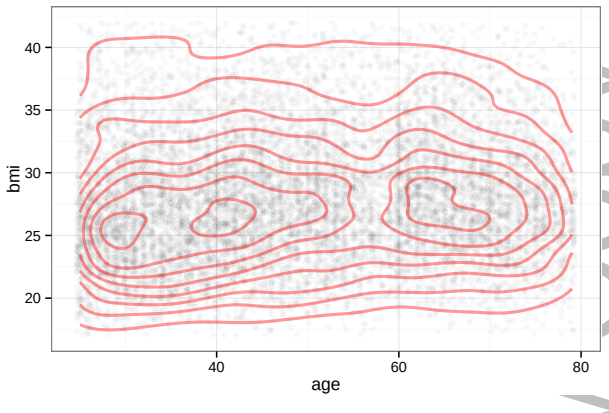


This and several following problems concern the `NCHS` data in the `DataComputing` package.

Problem 14.2
Judging from the graph, what's the most likely height among the `NCHS` people? (FYI: The heights are in meters.)



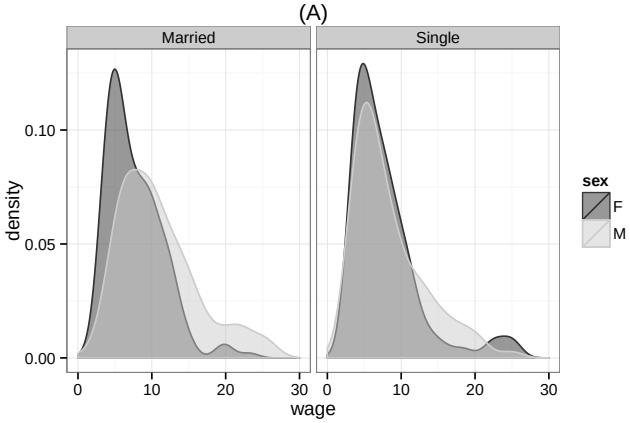
Problem 14.4
Below is a plot of the density for `bmi` and `age` simultaneously. The curved lines play the role of contours on a contour map of geography. Based on this graph, what's the most likely BMI for a 40-year old? For a 70-year old?



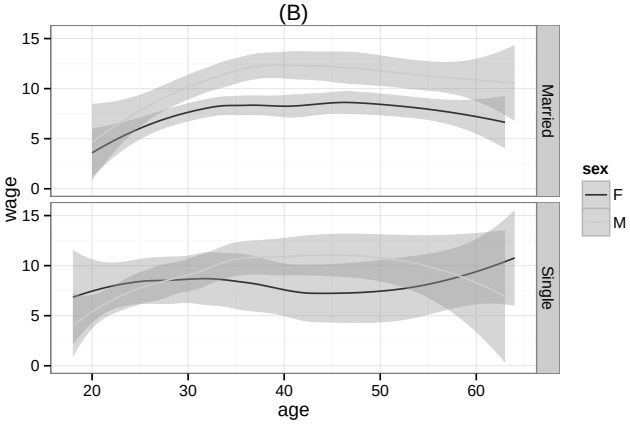
Problem 14.5

Here are several graphics based on the `mosaicData::CPS85` data table. Write `ggplot2()` statements that will construct each graphic.

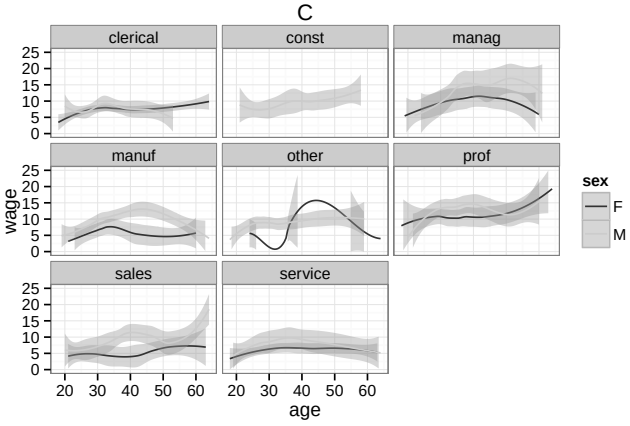
(A)



(B)

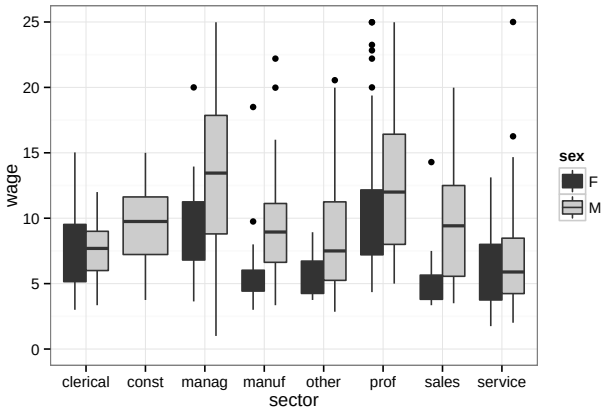


C



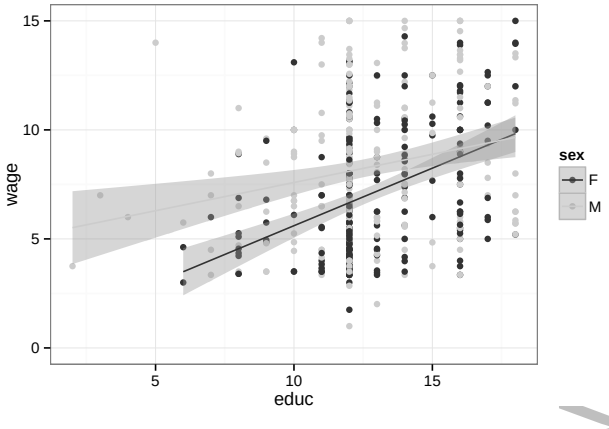
Problem 14.6

Reconstruct this graphic using the `ggplot()` system and the `mosaicData::CPS85` data table.



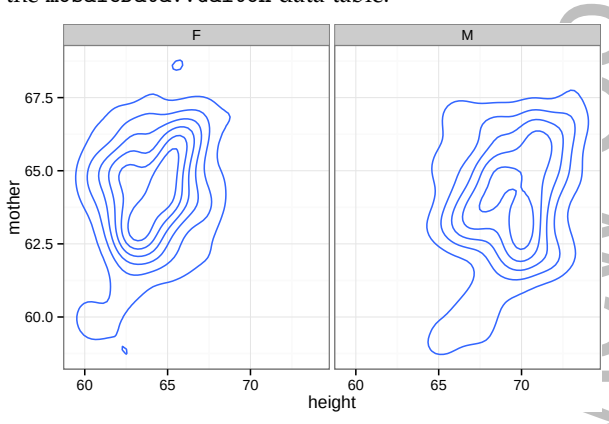
Problem 14.7

Reconstruct this graphic using the `ggplot()` system and the `mosaicData::CPS85` data table.



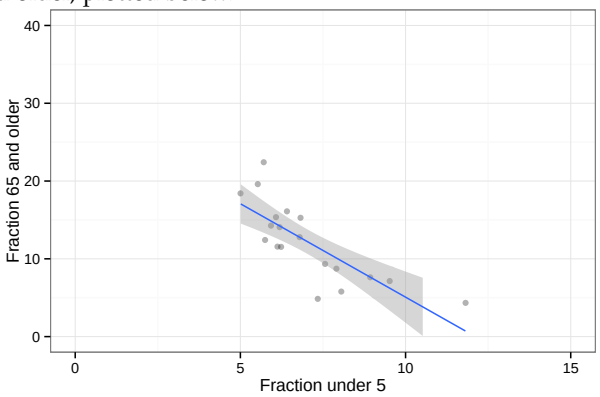
Problem 14.8

Reconstruct this graphic using the `ggplot()` system and the `mosaicData::Galton` data table.



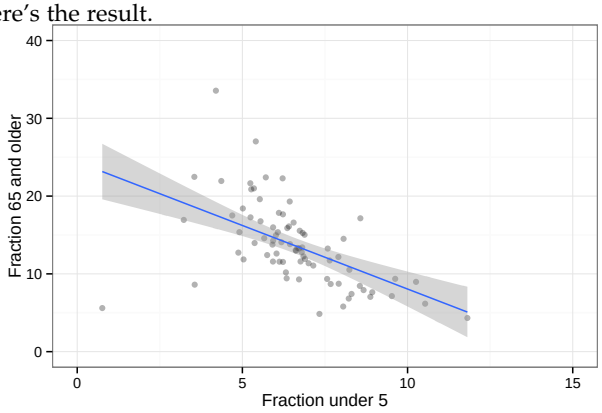
Problem 14.9

You're doing a study of whether grandparents go where the grandkids are. As a first attempt, you look at 20 ZIP codes. You find the relationship between the fraction of people in a ZIP code under 5 years old and the fraction 65 and older, plotted below.



- Do the data indicate that ZIP codes with high elderly populations tend to have high child populations?
- Looking at the confidence bands, is your data possibly consistent with there being no relationship (that is, a level line) between elderly population and child population?

You decide to get more data: study 80 ZIP codes. Here's the result.



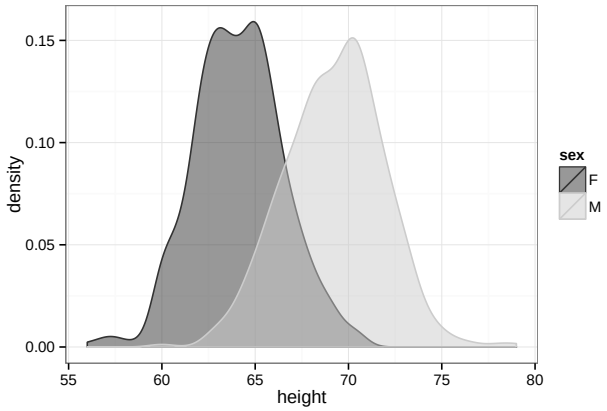
a. Is a flat line consistent with the data?

b. Compare the height of the confidence band with 20 ZIP codes to the height of the band with 80 ZIP codes: 4 times as much data in the larger sample than the smaller. Roughly, what's the ratio of confidence band heights?

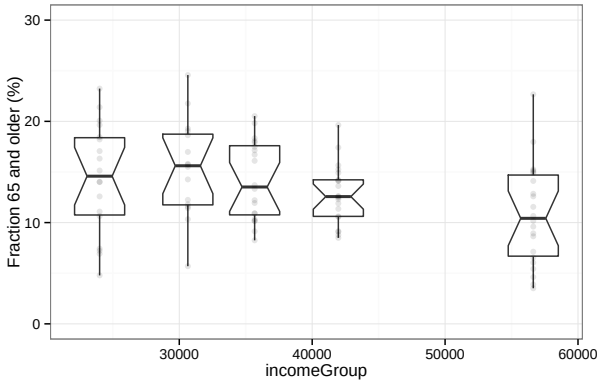
c. Statistical theory indicates that the width of a confidence band based on n points goes as $1/\sqrt{n}$. Does this seem about right?

Problem 14.10

Reconstruct this graphic using the `ggplot()` system and the `mosaicData::Galton` data table.

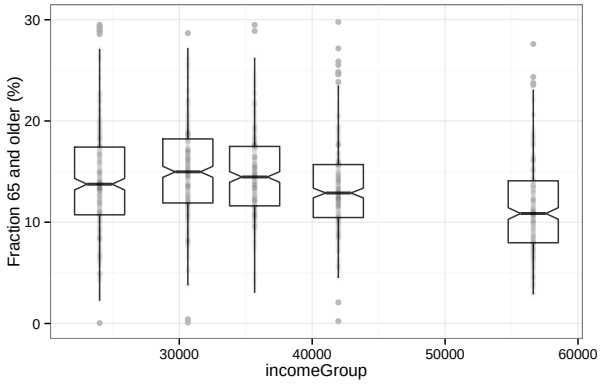


Problem 14.11
Studying a sample of 100 ZIP codes, the following graphic divides ZIP codes into 5 income groups, looking at the fraction of the population that is 65 and older.



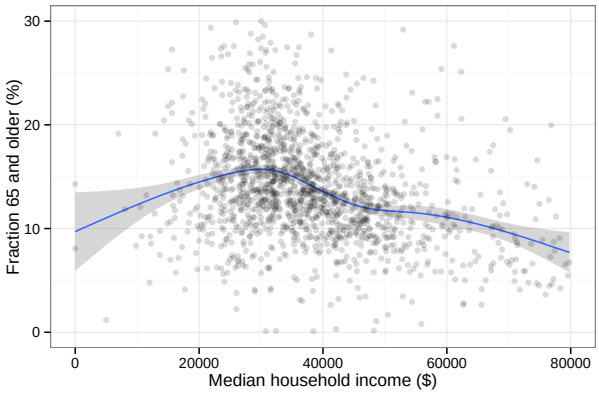
(a) What relationship do you see between income and the fraction of the population greater than 65? (The notch shows the confidence interval on the median population.)

The same analysis, but for a sample 16 times as big as before.



(b) Do the notch sizes (confidence intervals) follow the $1/\sqrt{n}$ rule?

Extra information: Often, quantitative variables such as income are divided into groups. Sometimes this reflects a lack of awareness by the data analyst about other possibilities. For instance, smoothers can provide an indication of general trends in the data without dividing into groups. For instance, this graph has `income` rather than `incomeGroup` on the x-axis.



Problem 14.12
One way to measure the "spread" of values is with the "standard deviation." The higher the standard deviation, the more spread out the values are. Use `sd()`, a summary function, to compute the standard deviation.

In the `MedicareCharges` data table, the `drg` variable describes billing codes for different types of medical conditions, called "Direct Recovery Groups."

- Taking all the hospitals together, which DRG has the highest standard deviation in Medicare charges.
- Taking all the DRGs together, which hospital has the highest standard deviation in Medicare charges.
- Medicare typically pays the hospital less than was charged by the hospital. Calculate the proportion of the charge that was paid (hint: a simple ratio), then find,
  - across all hospitals, which DRG has the
    - lowest proportion
    - highest proportion
  - across all DRGs, which hospital has the
    - lowest proportion
    - highest proportion

Problem 14.13
Figure 8.3 shows height versus age for smokers and non-smokers in the `NCHS` data.

Make a similar graphic that shows confidence bands of a smoother. Interpret the graph to conclude at what ages the smoker and non-smoker groups differ systematically in height.

15
Scraping and Cleaning Data

"Every easy data format is alike. Every difficult data format is difficult in its own way." — inspired by Leo Tolstoy

The wrangling and visualization techniques in this book are designed for working with data in a data-table format. Often, the data you encounter are in some other format. An early step in working with such formats is to translate them into one or more data tables. This process is called *data scraping*.

Likewise, data often have errors that stem from blunders in data entry or from deficiencies in the way data is stored or coded. Correcting such errors is called *data cleaning*.

DATA SCRAPING: Gathering data from sources that are not already in data table format. It usually refers to translating from formats used by web browsers to data-table form. Here it's used in a more general sense.

DATA CLEANING: Correcting errors or deficiencies in the way data is stored or coded.

15.1 Data-Table Friendly Formats

Many formats for data are essentially equivalent to data tables. When you come across data in a format that you don't recognize, it's worth checking whether it's one of the data-table friendly formats. Sometimes the filename extension provides an indication. Here are several, each with a brief description:

- CSV file. A non-proprietary text format that is widely used for data exchange between different software packages.
- Data tables in a technical software-package specific format:
  - Octave (and through that, MATLAB) widely used in engineering and physics.
  - Stata, commonly used for economic research
  - SPSS, commonly used for social science research
  - Minitab, often used in business practices
  - SAS, often used for large data sets

  - Epi, used by the Centers for Disease Control (CDC) for health and epidemiology data.

- Relational databases. This is the form that much of institutional, actively updated data are stored in. This includes business transaction records, government records, and so on.
- Excel, a set of proprietary spreadsheet formats heavily used in business. Watch out, though. Just because something is stored in an Excel format doesn't mean it's a data table. Excel is sometimes used as a kind of table cloth for writing down data with no particular scheme in mind.
- Web-related
  - HTML `<table>` format
  - JSON
  - Google spreadsheets published as HTML
  - Application Programming Interfaces (APIs) such as SODA

The particular software and techniques for reading data in one of these formats, varies depending on the format. For Excel or Google spreadsheet data, it's often sufficient to use the application software to export the data as a CSV file. There are also R packages for reading directly from an Excel spreadsheet, which is useful if the spreadsheet is being updated frequently.

For the technical software package formats, the `foreign` R package provides useful reading and writing functions. For relational databases, even if they are on a remote server, there are several useful R packages, including `dplyr` and `data.table`.

CSV and HTML `<table>` formats are frequently encountered. The next subsections give a bit more detail about how to read them into R.

CSV

CSV stands for comma-separated values. It's a text format that can be read with a huge variety of software. It has a data table format, with the values of variables in each case separated by commas. Here's an example of the first several lines of a CSV file:

```
"name","sex","count","year"
"Mary","F",7065,1880
"Anna","F",2604,1880
"Emma","F",2003,1880
"Elizabeth","F",1939,1880
```

The top row usually (but not always) contains the variable names. Quotation marks are often used at the start and end of character strings; these quotation marks are not part of the content of the string. CSV files are often named with the `.csv` suffix, it's also common for them to be named with `.txt` or other things. You will also see characters other than commas being used to delimit the fields: tabs and vertical bars are particularly common.

Since reading CSV spreadsheet data is so common, many package authors have provided functions for this task. `read.csv()` in the `base` package is perhaps the most widely used, but not the best.

Function	Package	Adapts to delimiter	Web URL	Fast
`read.csv()`	`base`	no	no	no
`read_csv()`	`readr`	no	yes	yes
`fread()`	`data.table`	yes	yes	yes
`read.file()`	`mosaic`	yes	yes	yes

Figure 15.1: Several functions for reading `.csv` files. To the user, they differ in flexibility in responding to the details of data file formats, whether they read Internet files directly, and speed.

`read.file()` is the function used in this book for reading CSV and some other kinds of files. It is a repackaging of `data.table::fread()` and `readr::read_csv()` the in the `data.table` package. `fread()` works for both files on your computer and those available on the Internet *via* a URL.fread@`data.table::fread()` For instance, here's a way to access a `.csv` file over the Internet.

```
# Remember the quotes around the character string.
myURL <- "http://tiny.cc/dcf/houses-for-sale.csv"
MyDataTable <- read.file(myURL)
```

price	bedrooms	bathrooms	fuel	air_cond	construction
132500	2	1.00	3	0	0
181115	3	2.50	2	0	0
109000	4	1.00	2	0	0
155000	3	1.50	2	0	0
86060	2	1.00	2	1	1
... and so on for 1,728 rows					

Table 15.2: The data table at `tiny.cc/dcf/houses-for-sale.csv`.

Just as reading a data file from the Internet uses a URL, reading a file on your computer uses a complete name, including the path to the file. Most people are used to using a mouse-based selector to access their files. The `file.choose()` function enables you to select a file in the familiar way, returning a character string with the file and path name. You can then call `fread()` with that character string. To illustrate:

```
file_name <- file.choose() # then navigate and click on your file
MyDataTable2 <-
  data.table::fread(file_name)
```

Useful arguments to `fread()`:

- `nrows=0` — just read the variable names. This is helpful when you are checking into the format and variable names of a data table. Of course, you might also want to look at a few rows of data, by setting `nrows` to a small positive integer, e.g. `nrows=3`.
- `select=c(1,4,5,10)` allows you to specify the variables you want to read in. This is useful for large data files with extraneous information.
- `drop=c(2,3,6)` is like `select`, but drops the specified columns.

A data table read in with `fread()` is not directly compatible with the join functions used in `dplyr`. This is easy to fix by converting them to "data frames," like this:

```
MyDataTable2 <-
  data.table::fread(file_name) %>%
  as.tbl()
```

Regretably, this adds a layer of complexity to working with `fread()`. For this reason, this book often uses `mosaic::read.file()`. (See below.)

HTML Tables

Web pages are usually in HTML format, which is then translated by your browser to the formatted content you see. HTML markup includes facilities for presenting tabular content. This HTML `<table>` markup is often the way human-readable data is arranged, as in the page shown in Figure 15.2.

When you have the URL of a page containing one or more tables, it is sometimes easy to read them in to R as data tables. Rather than using a CSV file-reading function like `fread()`, use the general purpose `read_html()`. Once you have the content of the web page, you can translate any tables in the page from HTML to data table format.

```
library(rvest)
web_page <- "http://en.wikipedia.org/wiki/Mile_run_world_record_progression"
SetOfTables <- web_page %>%
  read_html() %>%
  html_nodes(xpath = '//*[@id="mw-content-text"]/table') %>%
  html_table(fill = TRUE)
```

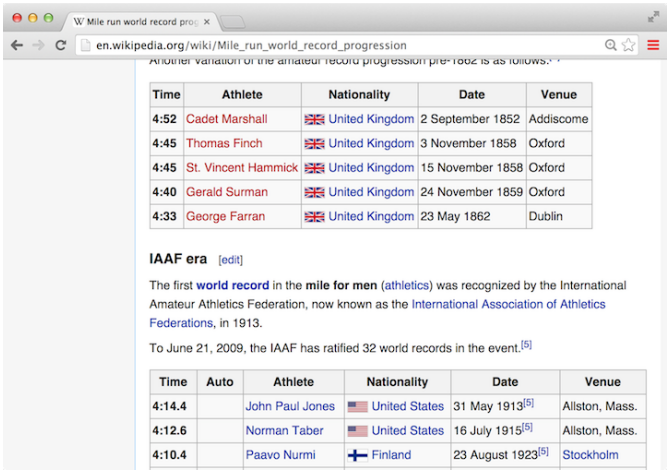


Figure 15.2: Part of a page on mile-run world records Wikipedia. Two separate data tables are visible. You can't tell from this small part of the page, but there are eight tables altogether on the page. These two tables are the third and fourth in the page.

The result, `SetOfTables`, is not a data table. Instead, it is a *list* of the tables found in the web page. Use `length()` to find how many items there are in the list of tables.

LIST: An R object used to store a collection of other R objects.

```
length(SetOfTables)
```

```
[1] 7
```

You can access any of those 7 tables. The first table is `SetOfTables[[1]]`, the second table is `SetOfTables[[2]]`, and so on. As it happens, the tables shown in Figure 15.2 are the second and fourth in the running-records web page.

Note the use of double square brackets.

```
Table2 <- SetOfTables[[2]]
```

Time	Athlete	Nationality	Date	Venue
4:55	J. Heaviside	United Kingdom	1 April 1861	Dublin
4:49	J. Heaviside	United Kingdom	27 May 1861	Dublin
4:46	Matthew Greene	United Kingdom	27 May 1861	Dublin
		... and so on for 16 rows		

Table 15.3: The third table embedded in the Wikipedia page on running records.

```
Table4 <- SetOfTables[[4]]
```

Table 15.4: The fourth table from the Wikipedia page

Time	Auto	Athlete	Nationality	Date	Venue
4:14.4		John Paul Jones	United States	31 May 1913[5]	Allston, Mass.
4:12.6		Norman Taber	United States	16 July 1915[5]	Allston, Mass.
			... and so on for 32 rows		

Of course, you might prefer to use names that are more descriptive than `Table2` and `Table4`.

15.2 Cleaning Data

A person somewhat knowledgable about running would have little trouble interpreting the Tables 15.3 and 15.4 correctly. The `Time` is in minutes and seconds. The `Date` gives the day on which the record was set. But when the data table is read into R, both `Time` and `Date` are stored as character strings. Before they can be used, they have to be converted into a format that the computer can process like a date and time. Among other things, this requires dealing with the `[5]` at the end of the date information.

Data cleaning refers to taking the information contained in a variable and transforming it to a form in which that information can be used.

Recoding

Table 15.5 displays a few variables from Table 15.2 describing 1728 houses houses for sale in Saratoga, NY.[1] The full table includes additional variables such as `living_area`, `price`, `bedrooms`, and `bathrooms`. The data on house systems, such as `sewer_type` and `heat_type` have been stored as numbers, even though they are really categorical.

[1] The example comes from Prof. Richard DeVeaux at Williams College.

```
Houses <-
  read.file("http://tiny.cc/dcf/houses-for-sale.csv")
```

fuel	heat	sewer	construction
3	4	2	0
2	3	2	0
2	3	3	0
2	2	2	0
2	2	3	1
... and so on for 1,728 rows			

Table 15.5: Four of the variables from the `houses-for-sale.csv` file giving features of the Saratoga houses stored as integer codes. Each case is a different house.

There's nothing fundamentally wrong with using integers to encode, say, fuel type. But it can be confusing to interpret results. Worse, the numbers imply a meaningful order to the categories when there is none.

To translate the integers to a more informative coding, you first have to find out what the various codes mean. Often, this information comes from the codebook, but sometimes you will need to contact the person who collected the data.

Once you know the translation, you can use spreadsheet software to enter them into a data table, like this one for the houses:

```
Translations <-
  read.file("http://tiny.cc/dcf/house_codes.csv")
```

code	system_type	meaning
0	new_const	no
1	new_const	yes
1	sewer_type	none
2	sewer_type	private
3	sewer_type	public
0	central_air	no
1	central_air	yes
2	fuel_type	gas
... and so on for 13 rows		

Table 15.6: Codes for the house system types.

`Translations` describes the codes in a format that makes it easy to add new code values as the need arises. The same information can also be presented a wide format as in Table 15.7.

```
CodeVals <-
  Translations %>%
  spread(key = system_type, value = meaning, fill = "invalid")
```

CodeVals

code	central_air	fuel_type	heat_type	new_const	sewer_type
0	no	invalid	invalid	no	invalid
1	yes	invalid	invalid	yes	none
2	invalid	gas	hot air	invalid	private
3	invalid	electric	hot water	invalid	public
4	invalid	oil	electric	invalid	invalid

Table 15.7: `Translations` rendered in a wide format.

In `CodeVals`, there is a column for each system type that translates the integer code to a meaningful term. In cases where the integer has no corresponding term, `invalid` has been entered. This provides a quick way to distinguish between incorrect entries and missing entries.

To carry out the translation, join each variable, one at a time, to the data table of interest. Note how the `by` value changes for each variable:

```
Houses <-
  Houses %>%
  left_join(CodeVals %>% select(code, fuel_type),
            by = c(fuel = "code")) %>%
  left_join(CodeVals %>% select(code, heat_type),
            by = c(heat = "code")) %>%
  left_join(CodeVals %>% select(code, sewer_type),
            by = c(sewer = "code"))
```

Table 15.8 shows the re-coded data.

fuel_type	heat_type	sewer_type
electric	electric	private
gas	hot water	private
gas	hot water	public
gas	hot air	private
gas	hot air	public
... and so on for 1,728 rows		

Table 15.8: The `Houses` data with re-coded categorical variables.

From Strings to Numbers

You've seen two major types of variables: quantitative and categorical. You're used to using quoted character strings as the levels of categorical variables, and numbers for quantitative variable.

Often, you will encounter data tables that have variables whose meaning is numeric but whose representation is a character string. This can occur when one or more cases is given a non-numeric value, e.g., `not available`.

The `as.numeric()` function will translate character strings with numerical content into numbers. `as.character()` goes the other way.

For example, in the `OrdwayBirds` data, the `Month`, `Day` and `Year` variables are all being stored as character string, even though their evident meaning is numeric. Convert the strings to numbers like this:

```
OrdwayBirds <-
  OrdwayBirds %>%
```

```
  mutate(Month=as.numeric(Month),
         Year=as.numeric(Year),
         Day=as.numeric(Day))
```

If the numerical strings have punctuation, e.g. `"¥2,540,937"`, the `tidyr::extract_numeric()` function is effective.

Dates

Dates are generally written down as character strings, for instance, "29 October 2014". Dates have a natural order. When you plot values such as *16 December 2015* and *29 October 2016*, you expect the December date to come after the October date, even though this is not true alphabetically of the string itself.

When plotting a value that is numeric, you expect the axis to be marked with a few round numbers. A plot from 0 to 100 might have ticks at 0, 20, 40, 60, 100. It's similar for dates. When you are plotting dates within one month, you expect the day of the month to be shown on the axis. But if you are plotting a range of several years, you it would be appropriate to show only the years on the axis.

When you are given dates stored as a character string, it can be useful to convert them to a computer format designed specifically for dates. For instance, in the `OrdwayBirds` data, the `Timestamp` variable refers to the time the data were transcribed from the original lab notebook to the computer file. You can translate the character string into a genuine date using functions from the `lubridate` package. Figure 15.9 shows a few of the date character strings from the `Timestamp` variable in `OrdwayBirds`.

These dates are written in a format showing month/day/year hour:minute:second. The `mdy_hms()` function from the `lubridate` package converts strings in this format to a date. As an example, suppose you want to examine when the entries were transcribed and who did them. You might create a data table and plot, as in Figure 15.3 like this.

```
library( lubridate )
WhenAndWho <-
  OrdwayBirds %>%
  select(Who = DataEntryPerson, When = Timestamp) %>%
  mutate(When = mdy_hms(When))

WhenAndWho %>%
  ggplot(aes(When, Who)) +
  geom_point(alpha = 0.2)
```

Timestamp
1/31/2011 19:12:42
5/26/2010 11:44:32
4/8/2011 11:08:31

Table 15.9: A few timestamp strings from `OrdwayBirds`

`WhenAndWho`

Who	When
Jerald Dosch	2010-04-14 13:20:56
Caitlin Baker	2010-05-13 16:00:30
Caitlin Baker	2010-05-13 16:02:15
Caitlin Baker	2010-05-13 16:03:18
Caitlin Baker	2010-05-13 16:04:23
... and so on for 15,825 rows	

Table 15.10: The times at which `OrdwayBirds` data were transcribed.

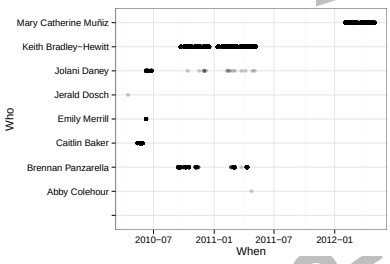


Figure 15.3: The transcribers of `OrdwayBirds` from lab notebooks worked at different times of day.

Many of the same operations that apply to numbers can be used on dates. For example, the date range worked by various transcribers can be calculated as a difference in times. (Table 15.11)

```
WhenAndWho %>%
  group_by(Who) %>%
  summarise(start = min(When,na.rm=TRUE),
            finish = max(When, na.rm=TRUE))
```

There are many similar `lubridate` functions for converting into dates strings in different formats, e.g. `ymd()`, `dmy()`, and so on. There are also functions like `hour()`, `yday()`, etc.

Who	start	finish
Abby Colehour	2011-04-23	2011-04-23
Brennan Panzarella	2010-09-13	2011-04-10
Caitlin Baker	2010-05-13	2010-05-28
Emily Merrill	2010-06-08	2010-06-08
Jerald Dosch	2010-04-14	2010-04-14
... and so on for 8 rows		

Table 15.11: Starting and ending dates for each transcriber involved in the `OrdwayBirds` project.

15.3 *Factors or Strings?*

R was designed with a special type for holding categorical data: "factors". Factors store categorical data efficiently and provide a means to put the categorical levels in whatever order is desired. Unfortunately, factors also make cleaning data more confusing. The problem is that it's easy to mistake a factor for a character string, but they have different properties when it comes to conversion to numeric or date form and especially when using the character processing techniques in Chapter 16.

By default, `read.file()`, `readr::read_csv()`, and `data.table::fread()` will interpret character strings as just that. Other functions such as `read.csv()` convert character strings into factors. Cleaning such data often requires converting it back to character-string format using `as.character()`. Failing to do this when needed can result in completely erroneous results without any warning.

For this reason, the data tables used in this book have been stored with categorical or text data in character-string format. Be aware that data provided by other packages do not necessarily follow this convention. So, if you get mysterious results when working with such data, consider the possibility that you are working with factors rather than character strings.

CSV files in this book are typically read with `read.file()`. If, for some reason, you prefer to use the `read.csv()` function, make sure to give argument `stringsAsFactors = FALSE` to insist that text data be stored as character strings.

15.4 Exercises

Problem 15.1
Here are some character strings containing times or dates written in different formats.

For each date, choose an appropriate function from the `lubridate` package to translate the character string into a date-time object. Then, using subtraction, find the number of days between that date and your birthday.

- `"April 30, 1777"` Johann Carl Friedrich Gauss
- `"06-23-1912"` Alan Turing's birthday
- `"3 March 1847"` Alexander Graham Bell's birthday
- `"11:00 am on Nov. 11th, 1918 at 11:00 am"` Armistice ending World War I on the Western Front.
- `"July 20, 1969"` First manned moon landing

Example:

```
lubridate::ymd("1941-09-01") -
  lubridate::mdy("July 28th, 1914")
```

```
Time difference of 9897 days
```

Problem 15.2
Here are some strings containing numerical amounts. For which ones does `as.numeric()` work correctly? How about `tidyr::extract_numeric()`?

a. `"42,659.30"`
b. `"17%"`
c. `"Nineteen"`
d. `"£100"`
e. `"9.8 m/seconds-square"`
f. `"9.8 m/s^2"`
g. `"6.62606957 × 10^-34 m2 kg / s"`
h. `"6.62606957e-34"`
i. `"42.659,30"` (A European style)

Problem 15.3
Grab Table 4 (or another similar table) from the Wikipedia page on world records in the mile (or some similar event). Make a plot of the record time versus the date in which it occurred. For extra credit, mark each point with the name of the athlete written above the point. (Hint: Use `geom_text()`)

Some tips To convert time entries such as "4:20.5" into seconds, use the `lubridate` package's `as.duration(ms("4:20.5"))`.

You can get rid of the footnote markers such as `[5]` in the dates with a statement like this:

```
T4 <-
  T4 %>%
  mutate(Date = gsub("\\[.\\]$", "", Date))
```

The `gsub()` transformation function replaces the characters identified in the first argument with those in the second argument. The string `"\\[.\\]$"` is an example of a "regular expression" which identifies a pattern of characters, in this case a single character in square brackets just before the end of the string. (Chapter 16 describes regular expressions in more detail.)

16
Using Regular Expressions

A "regular expression," often called *regex*, is a way of describing patterns in strings of characters. What is such a pattern? A few examples:

- Telephone numbers. Although called "numbers," they are often written in a nonstandard numerical format, e.g.,
  - `(651) 696-6000` `+1 651.696.6000`
- Times of day are written according to a pattern, e.g.
  10:30 AM, 9 o'clock, 13:15.54
  Durations are also written in a similar way. The world class time in the marathon is 2:02:57.
- POSTAL CODES. In the US, postal codes are written in several formats:
  - five digits, e.g., 55105,
  - Zip + 4, 9 digits written 55105-1362
- Leading and trailing spaces. Sometimes the content of a string is preceeded or followed by blank statements, like this:
  " henry "

Regular expressions are used for several purposes:

- to detect whether a pattern is contained in a string. Use `filter()` and `grepl()`.
- to replace the elements of that pattern with something else. Use `mutate()` and `gsub()`.
- to extract a component that matches the patterns. Use `extract()` from the `DataComputing` package.

To illustrate, consider the baby names data, summarised to give the total count of each name for each sex.

```
NameList <- BabyNames %>%
  group_by( name, sex ) %>%
  summarise( total=sum(count) ) %>%
  arrange( desc(total))
```

Here are some examples of patterns in names and the use of a regular expression to detect them. The regular expression is the string in quotes. `grepl()` is a function that compares a regular expression to a string, returning TRUE if there's a match, FALSE otherwise.

- The name contains "shine", as in "sunshine" or "moonshine" or "Shinelle"

```
NameList %>%
  filter(grepl("shine", name, ignore.case = TRUE))
```

name	sex	total
Rashine	M	11
Shine	F	95
... and so on for 16 rows		

- The name contains three or more vowels in a row.

```
NameList %>%
  filter(grepl("[aeiou]{3,}", name, ignore.case = TRUE))
```

name	sex	total
Aaidan	M	38
Aaiden	M	472
... and so on for 1,933 rows		

- The name contains three or more consonants in a row.

```
NameList %>%
  filter(grepl("[^aeiou]{3,}", name, ignore.case = TRUE) )
```

name	sex	total
Aadarsh	M	140
Aadhya	F	390
... and so on for 16,767 rows		

- The name contains "mn"

```
NameList %>%
  filter(grepl("mn", name, ignore.case = TRUE))
```

name	sex	total
Aamna	F	181
Amna	F	1099
... and so on for 51 rows		

- The first, third, and fifth letters are consonants.

```
NameList %>%
  filter(grepl("^[^aeiou].[^aeiou].[^aeiou]", name,
           ignore.case = TRUE))
```

name	sex	total
Babacar	M	124
Babajide	M	44
... and so on for 28,747 rows		

16.1 Regex basics

There are simple regular expressions and complicated ones. All of them look foreign until you learn how to read them.

Regexes encode patterns. The simplest patterns can be taken literally. For instance, `"able"` will match any string with "able" in it,

e.g., "You are able . . . ", "She is capable", "The motion is tabled". But `"able"` will not match "Able to do what?" Capital letters are distinct from lower case, although this can be turned off in many functions with the argument `ignore.case = TRUE`, as used in the previous examples based on `BabyNames`.

More complicated patterns involve punctuation that has a special meaning. These characters have a special meaning: `.`, `^`, `$`, `[`, `|`, `]`, `{`, `}`

For example, `".ble"` is a regex that matches any set of four letters ending with "ble". This will match all the strings matched by `"able"` but others as well: "sugar is soluble in water", "money is fungible", "Bible reading for today".

- Simple patterns:
  - A single `.` means "any character."
  - A character, e.g., `b`, means just that character.
  - Characters enclosed in square brackets, e.g., `[aeiou]` means any one of those characters. (So, `[aeiou]` is a pattern describing a vowel.)
  - The `^` inside square brackets means "any except these." So, a consonant is `[^aeiou]`
- Alternatives. A vertical bar means "either." For instance, the regex `"rain|snow|sleet|hail"` will match any of those four forms of precipitation.
- Two simple patterns in a row, means those patterns consecutively. Example: `"M[aeiou]"` means a capital M followed by a single lower-case vowel.
- Repeats
  - A pattern followed by a `+` means "zero or more times." So, "'M[aeiou]+" matches "Miocene", "Mr.", "Ms", "Mrs.", "Miss", and "Man", among many others.
  - A simple pattern followed by a `?` means "zero or one times." So, `"M[aeiou]?[^aeiou]"` will not match "Miocene". (The regex means: M followed by at most one vowel, followed by a consonant. "Miocene" has two vowels following the M.)
  - A simple pattern followed by a `*` means "one or more times." `"M[aeiou]*"` will match "Miocene" and "Miss", but not "Mr."
  - A simple pattern followed by `{2}` means "exactly two times." Similarly, `{2,5}` means between two and five times, `{6,}` means six times or more. For instance, `[aeiou]{2}` means "exactly two vowels in a row" and will not match "Miss" or "Mr" or "Madam".

- Start and end of strings.
  - `^` at the beginning of a regular expression means "the start of the string"
  - `$` at the end means "the end of the string."
- Extraction: Some functions that process regexes will extract the specific character sequence that matches the specified sequence. To specify which part of the match is to be extracted, put it inside open and close parentheses. For example, the regex `"([aeiouAEIOU]{4})"` in following statement will match all names with 4 consecutive vowels of either upper or lower case and extract those vowels as a new variable.

```
BabyNames %>%
  group_by(name) %>%
  summarise(total = sum(count)) %>%
  extractMatches("([aeiouAEIOU]{4})", name) %>%
  filter( ! is.na(match1)) %>%
  arrange(desc(total))
```

name	total	match1
Louie	29293	ouie
Sequoia	3275	uoia
Gioia	480	ioia
Keiaira	81	eiai
Zoiee	77	oiee
... and so on for 39 rows		

Table 16.1: Baby names with 4 consecutive vowels captured by the regex `"([eaiouAEIOU]{4})"`

On occasion, you will want to use one of the regex control characters in a literal sense, for example `.` to mean "period" and not "any one character." You can do this by "escaping" that character by preceeding it with a double `\\`. For instance, in the regex `"^\\$|¥|£|¢$"` about currency symbols, the first `$` is intended to refer to the dollar currency sign. But on its own in a regex, `$` refers to the end of a character string. In order to signal that the first `$` is to be taken literally as a currency symbol, it's been escaped with `\\`. To create a regex involving the period found at the end of sentences, you need to escape the `.`, like this: `"\\.$"`

16.2 *Example tasks with regular expressions.*

Get rid of percent signs and commas in numerals

Numbers often come with comma separators or unit symbols such as % or $. For instance, Table 16.2 shows part of a table about public debt from Wikipedia. To use these numbers for computations, they must be cleaned up.

```
Debt %>%
  mutate( debt=gsub("[$,%]|billion","",debt),
          percGDP=gsub("[,%]", "", percGDP))
```

country	debt	percGDP
World	$56,308 billion	64%
United States*	$17,607 billion	73.60%
Japan	$9,872 billion	214.30%
China	$3,894 billion	31.70%
Germany	$2,592 billion	81.70%
... and so on for 28 rows		

Table 16.2: Public debt (from Wikipedia)

Remove a currency sign

```
gsub("^\\$|€|¥|£|¢$","", c("$100.95", "45¢"))
```

```
[1] "100.95" "45"
```

Remove leading or trailing spaces

```
gsub("^ +| +$", "", "   My name is Julia     ")
```

```
[1] "My name is Julia"
```

How often do names end in vowels?

```
NameList %>%
  filter( grepl( "[aeiou]$", name ) ) %>%
  group_by( sex ) %>%
  summarise( total=sum(total) )
```

sex	total
F	96702371
M	21054791

Table 16.4: The number of babies given names ending in a vowel.

Girls' names are almost five times as likely to end in vowels as boys' names.

What are the most common end vowels for names?

To answer this question, you have to extract the last vowel from the name. The `extractMatches()` transformation function can do this.[1] The regex `"([aeiou])$"` used with `extractMatches()` means "any of aeiou immediately before the end of the string. The parentheses in the regex instruct `extractMatches()` to pull out the part of the match with the regex corresponding to the parentheses's contents. In Table 16.5, the last entry is for `NA`. When there is no match — in this context, when the name ends in a consonant — `extractMatches()` returns `NA`.

[1] `extractMatches()` is in the `DataComputing` package.

You can see that a name ending in "a", "e", or "i" suggests strongly that it's a girl's name.

```
NameList %>%
  extractMatches("([aeiou])$", name, vowel=1 ) %>%
  group_by( sex, vowel ) %>%
  summarise( total=sum(total) ) %>%
  arrange( desc(total) ) %>%
  spread(key=sex, value=total)
```

vowel	F	M
a	56088501	1844041
e	36432218	14341114
i	3693024	753311
o	403120	4041190
u	85508	75135
... and so on for 6 rows		

Table 16.5: The count of babies with names ending in each vowel.

16.3 *Exercises*

country	debt	percGDP
World	56308	64
United States*	17607	73.60
Japan	9872	214.30
China	3894	31.70
Germany	2592	81.70
... and so on for 28 rows		

Table 16.3: Remove comma separators, etc. to create a pure number.

Problem 16.1

Using the `BabyNames` data table, find the 10 most popular

1. Boys' names ending in a vowel.
2. Names ending with "joe" or "jo" (like BettyJoe)

Problem 16.2

Here is a character string with a regular expression:

```
"([2-9][0-9]{2})[- .]([0-9]{3})[- .]([0-9]{4})"
```

To explain the first bit . . .

- `[2-9]` means "one digit from 2 to 9.
- `[0-9]` refers to one digit from 0 to 9.
- `[0-9]{2}` refers to two consecutive digits, 0 to 9.
- `[2-9][0-9]{2}` means one digit 2 to 9 followed by two digits 0 to 9
- `[- .]` means "any of the characters dash, space, period, just once.
- The parentheses refer to the matching contents to be extracted. The whole expression has the structure `(stuff)[- .](more stuff)[- .](still more stuff)`. The three sets of parentheses mean to extract those three pieces from strings thatmatch.

Explain what familiar kinds of strings the entire general expression would match. (Hint: Call me maybe.) What components of those strings are being extracted?

Problem 16.3

Consider this regex:

```
(A[LKSZRAP]|C[AOT]|D[EC]|F[LM]|G[AU]|HI|I[ADLN]|
K[SY]|LA|M[ADEHINOPST]|N[CDEHJMVY]|O[HKR]|P[ARW]|
RI|S[CD]|T[NX]|UT|V[AIT]|W[AIVY])
```

(Ignore the line breaks)

1. How long will the strings be that match the pattern?
2. How many different strings will match?
3. People living in the United States may be able to figure out what the pattern is meant to express. Give it a try.

Problem 16.4

A list of names from the Bible can be accessed like this:

```
BibleNames <-
  read.file("http://tiny.cc/dcf/BibleNames.csv")
```

Using the names in `BibleNames`,

1. Which names have any of these words in them: "bar", "dam", "lory"?
2. Which names *end* with those words?

You need only show a few.

Problem 16.5

The city of Boston publishes various public-safety data online. A data table listing almost 300,000 crime reports from Feb. 6, 2012 up through the present is available via `https://data.cityofboston.gov/Public-Safety/Crime-Incident-Reports/7cdf-6fgx`

A small, convenient extract of the Boston crime data is available to you:

```
CrimeSample <-
  read.file("http://tiny.cc/dcf/Boston-Crimes-50.csv")
```

The `Location` variable contains information about latitude and longitude. Each of these is a number, but they are represented in `Location` as a formatted character string.

Write a regular expression that will extract the latitude and longitude as numbers into separate variables. To do the extraction you can use `tidyr::extract()`, e.g.

```
my_regex <- # Your regular expression goes here
CrimeSample %>%
  tidyr::extract("Location", into=c("lat", "long"),
                 regex = my_regex,
                 convert = TRUE)
```

Some hints:

- You'll need the extraction parentheses, written as plain parens: `( )`. If you want to refer to the parentheses characters, not as extraction markers but as plain text, you need to "escape" them with backslashes, e.g. `"\\(some pattern in parens\\)"` The two backslashes are needed so that R realizes that you are escaping the character that follows.
- The regex symbol for a digit is [0-9].
- A regex that will extract a single floating-point number surrounded by parentheses is: `"\\([+-]*[0-9]*[0-9\\.][0-9]*\\)"`.

17
Machine Learning

Up to now, the representation of data has been in the form of graphics. Graphics are often easy to interpret (but not always). They can be used to identify patterns and relationships — this is termed *exploratory analysis*. They are also useful for telling a story — presenting a pattern or relationship that is important for some purpose.

But graphics are not always the best way to explore or to present data. Graphics work well when there are two or three or even four variables involved in the patterns to be explored or in the story to be told. Why two, three or four? Two variables can be represented with position on the paper or screen or, ultimately, the eye's retina.

To represent a third variable, color or size can be used. In principle, more variables can be represented by other graphical aesthetics: shape, angle, color saturation, opacity, facets, etc. Doing so raises problems for human cognition; people simply don't do well at integrating so many graphical modes into a coherent whole. The 4-variable graphic in Figure 17.1 is practically incomprehensible.

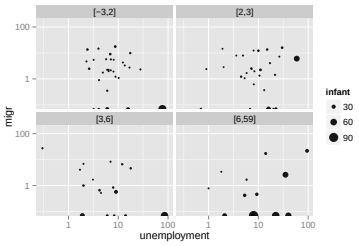


Figure 17.1: A display of four variables from `CountryData`: unemployment, migration, infant mortality, and monetary inflation.

MACHINE LEARNING: Using computer algorithms to search for and present patterns in data.

As an alternative to graphics, *machine learning* can be use to identify and describe useful patterns in data.The term machine learning was coined in the 1990s to label a set of inter-related algorithmic techniques for extracting information from data. In the days before computers were invented, mathematical approaches, particularly linear algebra, were used to construct model formulas from data. These pre-computer techniques, generally called *regression techniques*, ought now be included as machine learning. Many of the important concepts addressed by machine-learning techniques emerged from the development of regression techniques. Indeed, regression and related statistical techniques provide an important foundation for understanding machine learning.

REGRESSION TECHNIQUES: Long-standing ways to fit model formulas to data.

There are two main branches in machine learning: *supervised learning* and *unsupervised learning*. In supervised learning, the data being studied already include measurements of outcome variables. (Regression techniques are a form of supervised learning.) For instance,

SUPERVISED LEARNING: Finding and describing a relationship among measured variables.

UNSUPERVISED LEARNING: Finding ways to represent data in a compact form. This is often presented as finding a relationship between data and otherwise *unmeasured features of the cases.*

in the NCHS data, there is already a variable indicating whether or not the person has diabetes. Building a model to explore or describe other variables related to diabetes (weight? age? smoking?) is supervised learning.

In unsupervised learning, the outcome is unmeasured. For instance, assembling DNA data into an evolutionary tree is a problem in unsupervised learning. However much DNA data you have, you don't have a direct measurement of where each organism fits on the "true" evolutionary tree. Instead, the problem is to create a representation that organizes the DNA data themselves.

Supervised Learning

A basic technique in supervised learning is to search for a function that describes how different measured variables are related to one another.

A *function*, as you know, is a relationship between inputs and an output. Outdoor temperature is a function of season: season is the input; temperature is the output. Length of the day — i.e. how many hours of daylight — is a function of latitude and day of the year: latitude and day-of-the-year (e.g. March 22) are the inputs; day length is the output. But for something like the risk of developing colon cancer or diabetes, what should that be a function of?

There is a new bit of R syntax to help with defining functions: the tilde. The tilde is used to indicate what is the output variable and what are the input variables. You'll see expressions like this:

```
diabetic ~ age + sex + weight + height
```

Here, the variable `diabetic` is marked as the output, simply because it's to the left of the tilde. The variables `age`, `sex`, `weight`, and `height` are to be the inputs to the function. You'll also see the form `diabetic ~ .` as a syntax. The dot to the right of the tilde is a shortcut: "use all the available variables (except the output)."

There are several different goals that might motivate constructing a function:

- Predict the output given an input. It's February, what will the temperature be? On June 15 in Saint Paul, Minnesota, USA (latitude 45 deg N), how many hours of daylight will there be.

- Determine which variables are useful inputs. We all know that outdoor temperature is a function of season. But in less familiar situations, e.g. colon cancer, the relevant inputs are uncertain or unknown.

- Generating hypotheses. For a scientist trying to figure out the causes of colon cancer, it can be useful to construct a predictive model, then look to see what variables turn out to be related to risk of colon cancer. For instance, you might find that diet, age, and blood pressure are risk factors for colon cancer. Knowing some physiology suggests that blood pressure is not a direct cause of colon cancer, but it might be that there is a common cause of blood pressure and cancer. That "might be" is a hypothesis, and one that you probably would not have thought of before finding a function relating risk of colon cancer to those inputs.
- Understand how a system works. For instance, a reasonable function relating hours of daylight to day-of-the-year and latitude reveals that the the northern and southern hemisphere have reversed patterns: long days in the southern hemisphere will be short days in the northern hemisphere.

Depending on your motivation, the kind of model and the input variables will differ. In understanding how a system works, the variables you use should be related to the actual, causal mechanisms involved, e.g. the genetics of cancer. For predicting an output, it hardly matters what the causal mechanisms are. Instead, all that's required is that the inputs are known at a time *before* the prediction is to be made.

Consider the relationship between age and colon cancer. As with many diseases, the risk of colon cancer increases with age. Knowing this can be helpful in practice, for instance, helping to design an efficient screening program to test people for colon cancer. On the other hand, age does not suggest a way to avoid colon cancer: there's no way for you to change your age. You can, however, change things like diet, physical fitness, etc.

The *form* or *architecture* of a function refers to the mathematics behind the function and the approach to finding a specific function that is a good match to the data. Examples of the names of such forms are *linear least squares*, *logistic regression*, *neural networks*, *radial basis functions*, *discriminant functions*, *nearest neighbors*, etc.

These notes will focus on a single, general-purpose function architecture: *recursive partitioning*. Software for constructing recursive partitioning function is in the `party` package:

```
library(party)
```

To start, take a familiar condition: pregnancy. You already know what factors predict pregnancy, so you can compare the findings of the machine learning technique of recursive partitioning to what you already know.

The `party::ctree()` function builds a recursive-partitioning function. The `NCHS` data has a variable `pregnant` that identifies whether the person is pregnant.

```
whoIsPregnant <-
  ctree(pregnant == "yes" ~ age + income + sex, data=NCHS)
```

There are different ways to look at the function contained in `whoIsPregnant`. An important one is the output produced for any given input. Here's how to calculate that output when the inputs are an age of 25 (years), an income of 6 (times the poverty threshold), and sex female:

```
f <- funCT(whoIsPregnant) # get the function
f(age = 25, income = 6, sex = "female")
```

age	income	sex	prob
25	6	female	0.39

Table 17.1: According to the `whoIsPregnant` model, there is a 39 percent probability that a randomly selected 25-year old female in income group 6 is pregnant.

The input values are repeated to make it easier to compare inputs to outputs. Since the output is a yes-or-no, the output is given as the probability of each of these. So, for a 25-year-old woman with an income level of 6, the probability of being pregnant is 38.6%.

Another way to look at a recursive partitioning function is as a "decision tree".

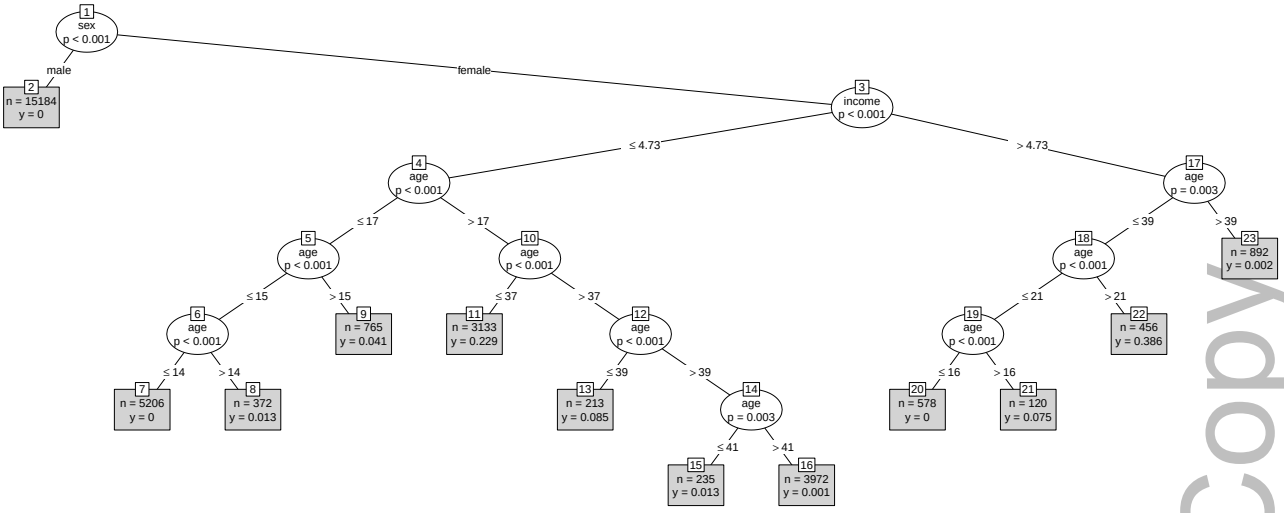


Figure 17.2: The decision tree from the `whoIsPregnant` model.

To read the tree in Figure 17.2, start at the top. The top vertex, drawn as an oval, specifies the variable `sex`. From that oval there are two branches: one labeled "male" and the other "female." This is the partitioning of the vertex from which the branches stem. As you follow a branch, you may come to another vertex, which itself branches, or you may come to a terminal (shown in the gray box). The terminal gives the output of the model.

For the 26-year-old female with an income level of 6, the function output can be read off by following the appropriate vertices. Starting at the top, follow the "female" branch. The vertex that leads to is about the `income` variable; the two branches from that vertex indicate the criterion for going left or right. In this case, the threshold for branching right is an income greater than 4.73. This indicates that to find the model output for an income level of 6, branch right.

The right branch from income leads to a vertex involving `age`. The threshold for branching right is an age greater than 39 years. The 26-year-old does not reach the threshold, so branch left.

That left branch leads to another vertex involving `age`. The threshold for branching right is 21 years old. The 26-year-old meets that threshold, so branch right. The right branch leads to a terminal: the gray box marked "22". The output for the example person — income 6, age 25, female — is 0.386. Admittedly, the format of the terminal is a bit obscure, but the information is there.

Within each vertex there is a notation like $p < 0.001$. This is not part of the function itself. Instead, it a statistical quantity calculated during the construction of the function and was used to inform the decision about whether there is enough data to be confident that the split at this node isn't just an artifact of the data sampling process. Low values indicate strong confidence.[1]

What has been learned in constructing this function? First, without having been told that `sex` is related to pregnancy, the recursive partitioning has figured out that sex makes a difference; only females get pregnant. Of course you knew that already, but the machine learned that fact from the data without any other experience with biology. The machine also identified the fact that females under 14 years old are not likely to be pregnant, and women over 40 are also unlikely to be pregnant. Again, you knew that already, but the machine learned it from the data. The machine also learned something that might be unfamiliar to you, that age patterns of pregnancy are different for low-income and high-income people.

It might be easier to see the income-related patterns by looking at the specific model outputs. Table 17.2 shows the function's output for low-income females from 15 to 25 years old:

```
f(age = c(15,20,25), income = 3, sex = "female")
```

And for high-income females:

```
f(age = c(15,20,25), income = 6, sex = "female")
```

Part of the machine learning is finding out which variables appear to be useful in matching the patterns of pregnancy in the NCHS data. For instance, the following recursive-partitioning tree can make use

[1] Somewhat more technically, p is the p-value from a hypothesis test. The hypothesis being tested with the data is that the observed difference between the left and right branches is merely the result of random, meaningless chance, and likely wouldn't show up if the NCHS data were collected on a new group of people.

age	income	sex	prob
15	3	female	0.01
20	3	female	0.23
25	3	female	0.23
15	6	female	0.00
20	6	female	0.07
25	6	female	0.39

Table 17.2: Probabilities of pregnancy from the `whoIsPregnant` model.

of several more variables. To avoid rediscovering the obvious, only females will be considered.

```
whoIsPregnant2 <-
  ctree(pregnant == "yes" ~
          age + income + ethnicity +
          height + smoker + bmi,
        data=NCHS %>% filter(sex == "female") )
```

Figure 17.3 presents the resulting tree. The variables `ethnicity`, `height`, `smoker`, and `bmi` have been added to the model. A reasonable person might question why these should be associated with becoming pregnant. Nonetheless, the model indicates that the variables have some predictive value.

BMI is the body mass index, a commonly used indicator of overweight and underweight.

Unsupervised Learning

Figure 17.4 shows an evolutionary tree of mammals. We humans (hominidae) are on the far left of the tree. The numbers at the branch points are estimates of how long ago — in millions of years — the branches separated. According to the diagram, rodents and we primates diverged about 90 million years ago.

How do evolutionary biologists construct a tree like this? They look at various traits of different kinds of mammals. Two mammals that have similar traits were deemed closely related. Animals with dissimilar traits are distantly related. Putting together all this information about what is close to what results in the evolutionary tree.

The tree, sometimes called a **dendrogram**, is a nice organizing structure for relationships. Evolutionary biologists imagine that at each branch point there was an actual animal whose descendents split into groups that developed in different directions. In evolutionary biology, the inferences about branches come from comparing existing creatures to one another (as well as creatures from the fossil record). Creatures with similar traits are in nearby branches, creatures with different traits are in further branches. It takes considerable expertise in anatomy and morphology to know what similarities and differences to look for.

But there is no outcome variable, just a construction of what is closely related or distantly related to what.

Trees can describe degrees of similarity between different things, regardless of how those relationships came to be. If you have a set of objects or cases, and you can measure how similar any two of the objects are, you can construct a tree. The tree may or may not reflect some deeper relationship among the objects, but it often provides a simple way to visualize relationships.

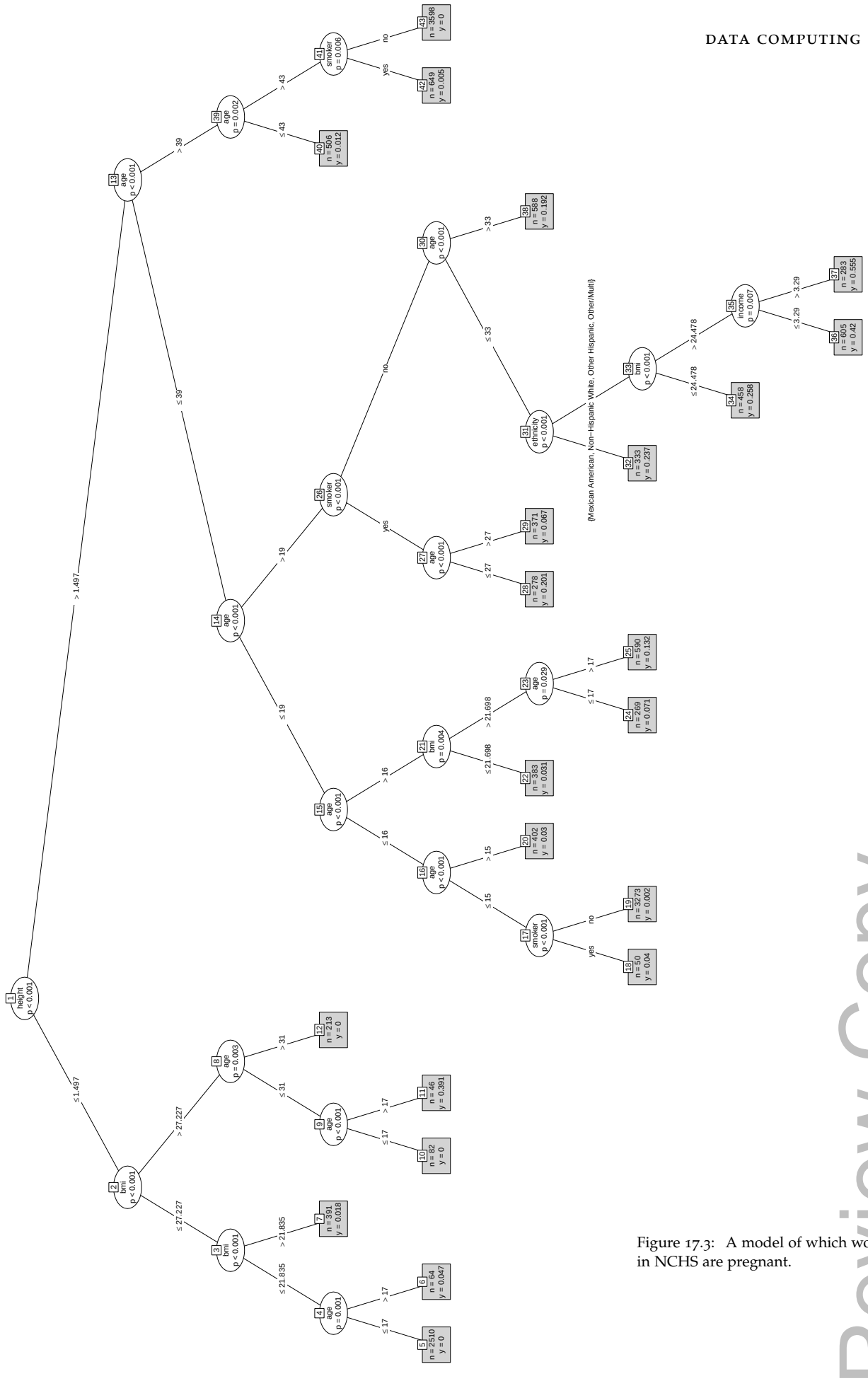


Figure 17.3: A model of which women in NCHS are pregnant.

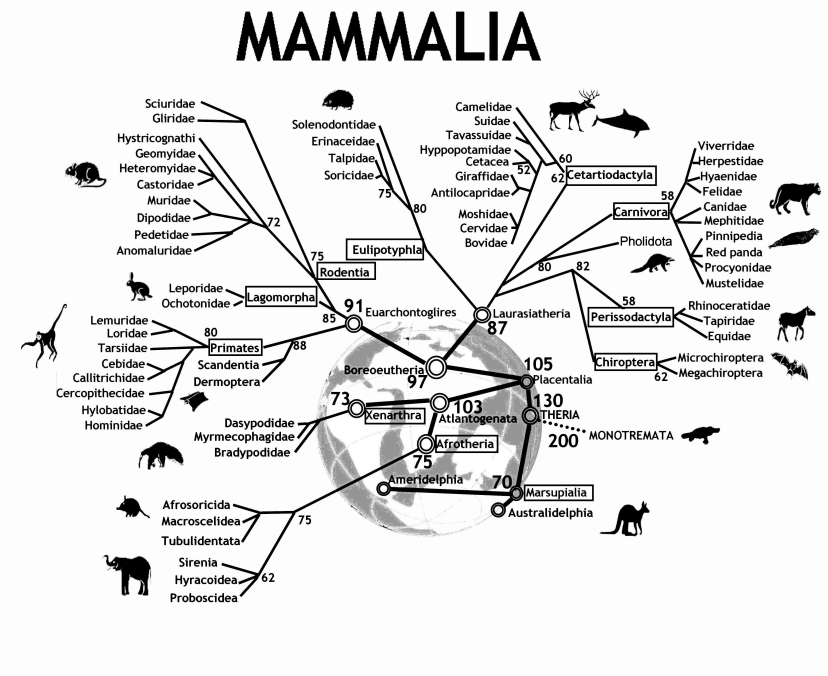


Figure 17.4: An evolutionary tree for mammals. Source [wiki-mammals]

Trees from Data

When the description of an object consists of a set of numerical variables, there are two main steps in constructing a tree to describe the relationship among the cases in the data:

1. Represent each case as a point in a Cartesian space.
2. Make branching decisions based on how close together points or clouds of points are.

To illustrate, consider the unsupervised learning process of identifying different types of cars. The `mtcars` data contains automobile characteristics: miles-per-gallon, weight, number of cylinders, horse power, etc.

For an individual quantitative variable, it's easy to measure how far apart any two cars are: take the difference between the numerical values. The different variables are, however, on different scales. For example, `gear` ranges only from 3 to 5, while `hp` goes from 52 to 355. This means that some decision needs to be made about rescaling the variables so that the differences along each variable reasonably reflect how different the respective cars are. There is more than one way to do this. The `dist()` function takes a simple and pragmatic point of view: each variable is equally important.

The output of `dist()` gives the *distance* from each individual car to every other car.

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear
30.40	4	75.70	52	4.93	1.61	18.52	1	1	4
18.10	6	225.00	105	2.76	3.46	20.22	1	0	3
15.80	8	351.00	264	4.22	3.17	14.50	0	1	5
32.40	4	78.70	66	4.08	2.20	19.47	1	1	4
19.20	6	167.60	123	3.92	3.44	18.30	1	0	4
				... and so on for 32 rows					

Table 17.3: The mtcars data table

```
carDiffs <- dist(mtcars)
```

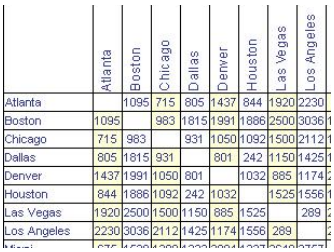
car_model	Honda Civic	Valiant	Ford Pantera L	Fiat 128	Merc 280	Camaro Z28
Honda Civic	0	170	360	15	120	350
Valiant	170	0	210	160	63	200
Ford Pantera L	360	210	0	350	240	20
Fiat 128	15	160	350	0	110	340
Merc 280	120	63	240	110	0	230
Camaro Z28	350	200	20	340	230	0

This point-to-point distance table is analogous to the tables that used to be printed on road maps giving the distance from one city to another, like Figure 17.5.

Notice that the distances are symmetrical: it's the same distance from Boston to San Francisco as from San Francisco to Boston (3043 mi, according to the table).

Knowing the distances between the cities is not the same thing as knowing their locations. But the set of mutual distances is enough information to reconstruct the relative positions.

Cities, of course, lie on the surface of the earth. That need not be true for the abstract distance between automobile types. Even so, the set of mutual distances provides information equivalent to knowing the relative positions. This can be used to construct branches between nearby items, then to connect those branches, and so on to construct an entire tree. The process is called "hierarchical clustering." Figure 17.6 shows a tree constructed by hierarchical clustering to relate car models to one another.



	Atlanta	Boston	Chicago	Dallas	Denver	Houston	Las Vegas	Los Angeles
Atlanta		1095	715	805	1437	844	1920	2230
Boston	1095		983	1815	1991	1886	2500	3036
Chicago	715	983		931	1050	1092	1500	2112
Dallas	805	1815	931		801	242	1150	1425
Denver	1437	1991	1050	801		1032	885	1174
Houston	844	1886	1092	242	1032		1525	1556
Las Vegas	1920	2500	1500	1150	885	1525		289
Los Angeles	2230	3036	2112	1425	1174	1556	289	

Figure 17.5: Distances between some US cities.

```
hc <- hclust(carDiffs)
plot(hc, hang=-1)
```

Another way to group similar cases considers just the cases, without constructing a heirarchy. The output is not a tree but a choice of which group each case belongs to.[2]

As an example, consider the cities of the world (in `WorldCities`). Cities can be different and similar in many ways: population, age structure, public transport and roads, building space per person, etc. The choice of features depends on the purpose you have for making the grouping.

My purpose is to show you that clustering via machine learning can actually learn genuine patterns in the data. So I choose features that are utterly familiar: the latitude and longitude of each cities.

Now, you know about the location of cities. They are on land. And you know about the organization of land on earth; most land falls in one of the large clusters called continents. But the `WorldCities` data

[2] There can be more detail than this, for instance a probability for each group that a specific case belongs to the group.

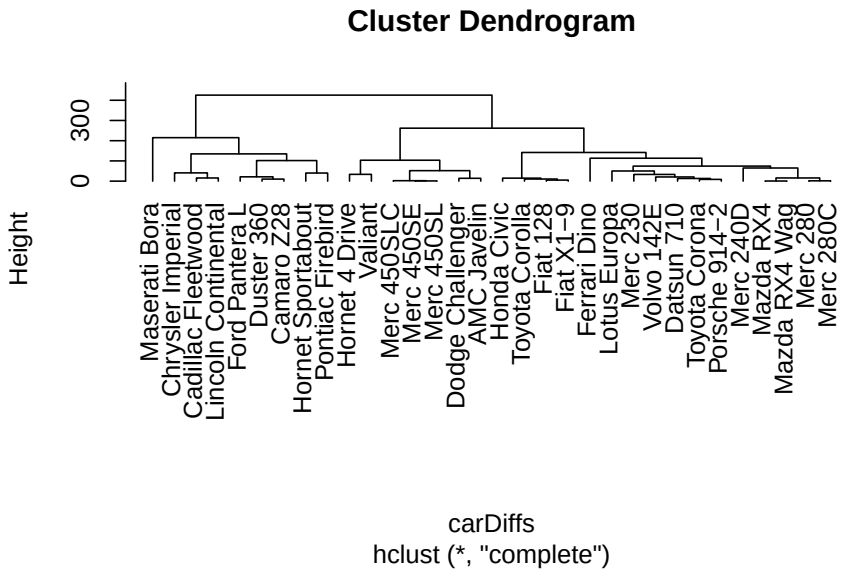


Figure 17.6: The dendrogram constructed by hierarchical clustering from car-to-car distances implied by the mtcars data table.

doesn't have any notion of continents. Perhaps it's possible that this feature, which you are completely aware of but isn't in the data, can be learned by a computer that never took grade-school geography.

For simplicity, consider the 4000 biggest cities in the world and their longitudes and latitudes.

```
BigCities <-
  WorldCities %>%
  arrange(desc(population)) %>%
  head( 4000 ) %>% select(longitude, latitude)
clusts2 <-
  BigCities %>%
  kmeans(centers = 6) %>%
  fitted("classes") %>% as.character()
BigCities <-
  BigCities %>% mutate(cluster = clusts2)
BigCities %>%
  ggplot(aes(x=longitude, y=latitude)) +
  geom_point(aes(color=cluster, shape=cluster) )
```

As shown in Figure 17.7, the clustering algorithm seems to have identified the continents. North and South America are clearly distinguished, as well as most of Africa. (The cities in North Africa are matched to Europe.) The distinction between Europe and Asia is essentially historic, not geographic. The cluster for Europe extends a ways into what's called Asia. East Asia and central Asia are marked as distinct. To the algorithm, the low population areas of Tibet and

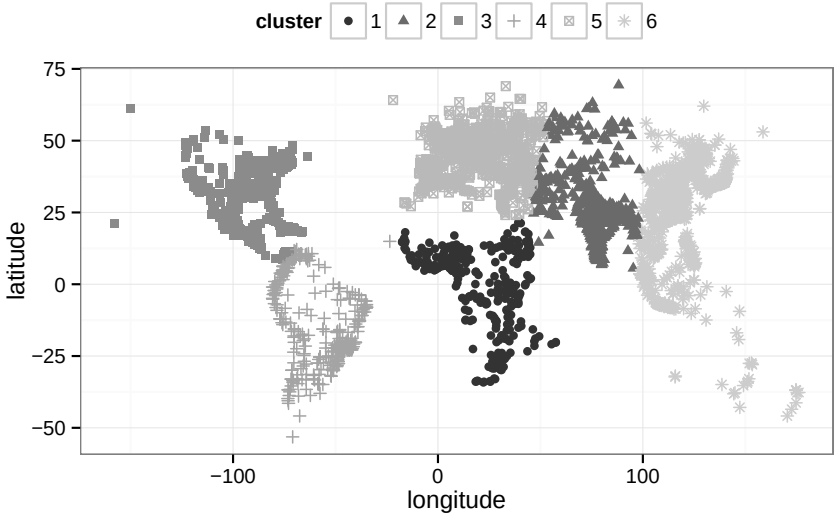


Figure 17.7: Clustering of world cities by their geographical position.

Siberia look the same as a major ocean.

Dimension Reduction

It often happens that a variable carries little information that's relevant to the task at hand. Even for variables that are informative, there can be redundancy or near duplication of variables. That is, two or more variables are giving essentially the same information; they have similar patterns across the cases.

Such irrelevant or redundant variables make it harder to learn from data. The irrelevant variables are simply noise that obscures actual patterns. Similarly, when two or more variables are redundant, the differences between them may represent random noise.

It's helpful to remove irrelevant or redundant variables so that they, and the noise they carry, don't obscure the patterns that machine learning could learn.

As an example of such a situation, consider votes in a parliament or congress. This section explores one such voting record, the Scottish Parliament in 2008. The pattern of ayes and nays may indicate which members are affiliated, i.e. members of the same political party. To test this idea, you might try clustering the members by their voting record.

Table 17.4 shows a small part of the voting record. The names of the members of parliament are the cases. Each ballot — identified by a file number such as "S1M-4.3" — is a variable. A 1 means an aye vote, −1 is nay, and 0 is an abstention. There are more than 130 members and more than 700 ballots. It's impractical to show all of the 100,000+ votes in a table. But there are only 3 levels for each

S1M.1	S1M.4.1	S1M.4.3
1	0	0
1	1	0
1	-1	-1
1	-1	-1
-1	-1	-1
... and so on for 10 rows		

Table 17.4: A few cases and variables from the Scottish Parliament voting data. There are 134 variables, each corresponding to a different ballot.

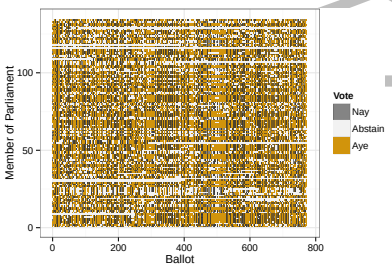


Figure 17.8: The values from the Scottish Parliament voting record displayed using a color code. Each dot refers to one member's vote on one ballot. Order is alphabetical by member, chronological bybBallot.

variable, so displaying the table as an image might work. (Figure 17.8)

It's hard to see much of a pattern here, although you may notice something like a tartan structure. The tartan pattern provides an indication to experts that the data could be re-organized in a much simpler way.[3]

As a start, Figure 17.10 shows the ballot values for all of the members of parliament for two randomly selected ballots. (Jittering is used to give a better idea of the point count at each position. The red dots are the actual positions.)

Each point is one member of parliament. Similarly aligned members are grouped together at one of the nine possibilities marked in red: (Aye, Nay), (Aye, Abstain), (Aye, Aye), and so on through to (Nay, Nay). In these two ballots, eight of the nine are possibilities are populated. Does this mean that there are 8 clusters of members?

Intuition suggests that it would be better to use *all* of the ballots, rather than just two. In Figure 17.11, the first 336 ballots have been added together, as have the remaining ballots. This graphic suggests that there might be two clusters of members who are aligned with each other. Using all of the data seems to give more information than using just two ballots.

You may ask why the choice was made to add up the first 336 ballots as x and the remaining ballots as y. Perhaps there is a better choice to display the underlying patterns, adding up the ballots in a different way.

In fact, there is a mathematical approach to finding the *best* way to add up the ballots, called "singular value decomposition" (SVD). The mathematics of SVD draw on a knowledge of matrix algebra, but the operation itself is readily available to anyone.[4] Figure 17.12 shows the position of each member on the best two ways of summing up the ballots.

Figure 17.12 shows, at a glance, that there are three main clusters. The red circle marks the "average" member. The three clusters move away from average in different directions. There are several members whose position is in-between the average and the cluster to which they are closest.

For a graphic, one is limited to using two variables for position. Clustering, however, can be based on many more variables. Using more SVD sums enables may allow the three clusters to be split up further. The color in Figure 17.12 above shows the result of asking for 6 clusters using the 5 best SVD sums. Table 17 compares the actual party of each member to the cluster memberships.

Figure 17.9: A tartan pattern, in this case the official tartan of West Virginia University.

[3] For those who have studied linear algebra: "Much simpler way" means that the matrix can be approximated by a matrix of low-rank.

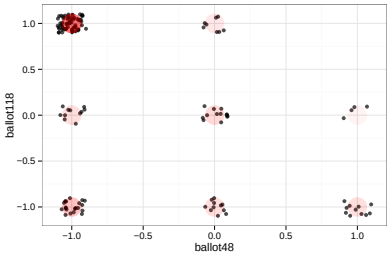


Figure 17.10: Positions of members of parliament on two ballots.

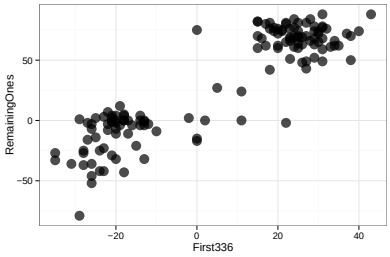


Figure 17.11: Positions of members of parliament on two ballot indices made up by the sum of groups of ballots.

[4] In brief, SVD calculates the best way to add up (i.e. linearly combine) the columns and the rows of a matrix to produce the largest possible variance. Then SVD finds the best way to add up the what's left, and so on.

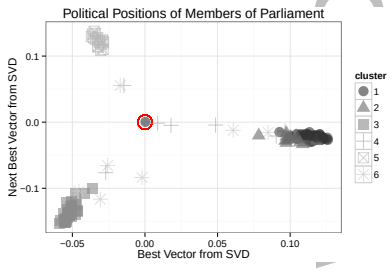


Figure 17.12: The position of each member of Parliament using the two 'best' ways of summing the ballots.

party . . . . . cluster	1	2	3	4	5	6
Member for Falkirk West	0	0	0	0	0	1
Scottish Conservative and Unionist Party	0	0	0	1	18	1
Scottish Green Party	0	0	0	0	0	1
Scottish Labour	48	3	0	5	0	2
Scottish Liberal Democrats	1	16	0	0	0	0
Scottish National Party	0	0	34	1	0	1
Scottish Socialist Party	0	0	0	0	0	1

The correspondence between cluster membership and actual party affiliation.

How well did clustering do? The party affiliation of each member of parliament is known, even though it wasn't used in finding the clusters. For each of the parties with multiple members, the large majority of members are placed into a unique cluster for that party. In other words, the technique has identified correctly that there are four different major parties.

There's more information to be extracted from the ballot data. Just as there are clusters of political positions, there are clusters of ballots that might correspond to such factors as social effect, economic effect, etc. Figure 17.13 a shows the position of each individual ballot, using the best two SVD sums as the x- and y-coordinates.

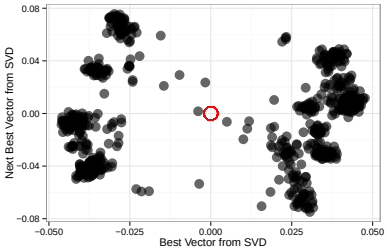


Figure 17.13: Orientation of issues among the ballots.

There are obvious clusters in Figure 17.13. Still, interpretation can be tricky. Remember that, on each issue, there are both aye and nay votes. This is what accounts for the symmetry of the dots around the center (indicated by an open circle). The opposing dots along each angle from the center might be interpreted in terms of *socially liberal* vs *socially conservative* and *economically liberal* versus *economically* conservative. Deciding which is which likely involves reading the bill itself.

Finally, the "best" sums from SVD can be used to re-arrange cases and separately re-arrange variables while keeping exactly the same values for each case in each variable. (Figure 17.14.) This amounts simply to re-ordering the members in a way other than alphabetical and similarly with the ballots. This dramatically simplifies the appearance of the data compared to Figure 17.8. With the cases and rows arranged as in Figure 17.14, it's easy to identify blocks of members and which ballots are about issues for which the members vote *en bloc*.

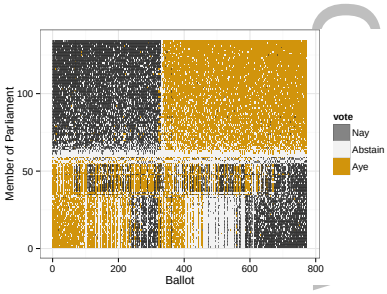


Figure 17.14: Re-arranging the order of the variables and cases in the Scottish parliament data simplifies the appearance of the data table.

17.1 Exercises

Problem 17.1

A risk factor is a characteristic associated with an increased likelihood of developing a disease or injury. In many situations, identifying risk factors provide an important means to screen those at high risk. In this exercise, you'll use the `NCHS` data to look for risk factors of diabetes.

In a large dataset such as `NCHS` with many variables, there are often some variables with much missing data. For instance, with `NCHS`, there are `nrow(NCHS)` cases, but many fewer that have no missing values. The `na.omit()` data verb filters out only those cases that have no missing values:

```
NCHS %>%
  na.omit() %>%
  nrow()
```

```
[1] 12013
```

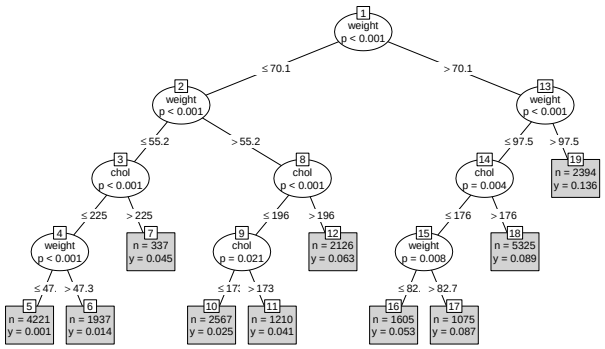
If only a few variables are of interest, it can be helpful to select out those variables when filtering out missing data.

```
CompleteCases <-
  NCHS %>%
  select(diabetic, weight, age,
         bmi, chol, smoker) %>%
  na.omit()
CompleteCases %>% nrow()
```

```
[1] 22797
```

Here is one model that considers weight and cholesterol levels as risk factors for diabetes:

```
mod1 <- party::ctree(
  diabetic ~ weight + chol,
  data=CompleteCases)
plot(mod1, type="simple")
```



In interpreting this tree, note that the quantity y indicates the probability of people in the group having diabetes, while n gives the size of the group.

A useful risk factor splits the population into groups with very high and with very low probabilities of diabetes. For instance, node 19 marks a group with a 13.6% risk of diabetes, substantially larger than other groups. Using other variables in the model, such as `smoker` or `age` might do a better job of dividing the cases into groups with high and low risk.

One way to measure the effectiveness of the model is with a quantity called the "log likelihood." The log likelihood compares, for all the cases individually, the actual outcome against the probability predicted by the model. A high log likelihood indicates a more successfully predictive model. Without going into the theory behind log likelihood, you can still calculate the value.

```
CompleteCases %>%
  mutate(probability = as.numeric(predict(mod1)),
         likelihood =
           ifelse(diabetic,
                  probability,
                  1-probability)) %>%
  summarise(log_likelihood = sum(log(likelihood)))
```

log_likelihood
-4450.64

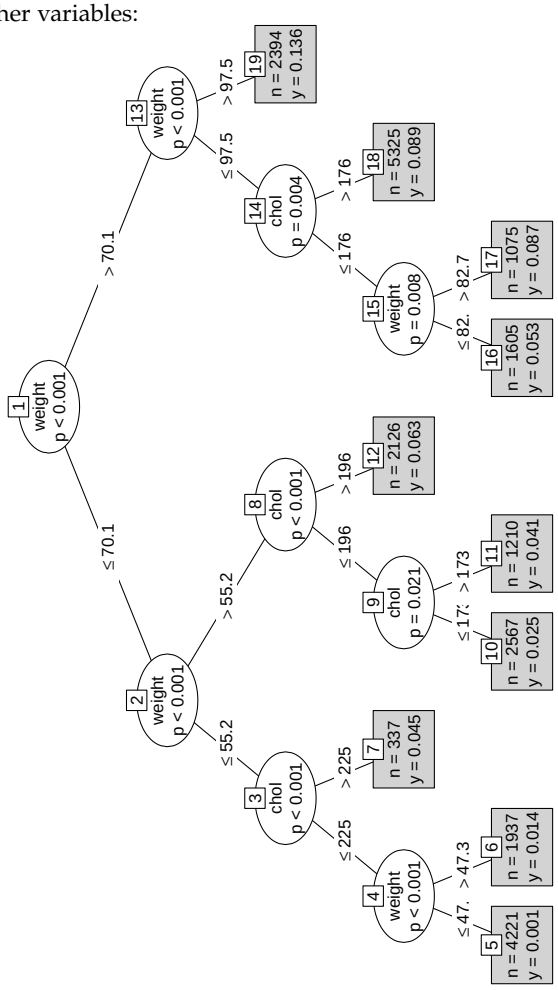
The number is meaningful only in comparison to the log likelihood for other models. Explore other models for diabetes using different combinations of potential risk factors. Find one with a higher log likelihood than `mod1`. Remember that the log likelihood number will always be negative, so -3000 is much bigger than -4000.

Problem 17.2

Consider these data on houses for sale:

```
Houses <-
  read.file("http://tiny.cc/dcf/houses-for-sale.csv")
```

Here is a model of house price as a function of several other variables:



In the model tree, the terminal node, shown as a rectangle, gives information about n the number of houses included in that group and y the mean price of the houses in that group. Based on the model tree, answer these questions:

1. What variables are in the model?
2. For houses with a living area less than 1080 square feet, does the number of bathrooms make a difference in price?
3. For houses with a living area between 1080 and 1483 square feet, how much is the typical difference in price between houses with 1 1/2 baths and houses with 2 bathrooms?
4. Is having a fireplace associated with a higher house price? What other variables does this depend on?

A
Projects

These projects involve a range of topics. Each contains a set of specified tasks, but the best way to learn is to ask your own questions and compute your own answers.

Suitable after the first 10 chapters:

- Popular names
- Bird species
- World cities
- Stocks and dividends

After chapters 11, 12, and 14:

- Statistics of gene expression
- Bicycle sharing
- Inspecting restaurants

Popular Names

Over the years and decades, names come and go in popularity. `BabyNames` provides a lot of information about this, but it is not in glyph-ready form.

Task:

Create a graph (like Figure A.1) showing the ups and downs in the popularity of names of interest to you. using names of interest to you. The raw material you have is the `BabyNames` data table in the `DataComputing` package.

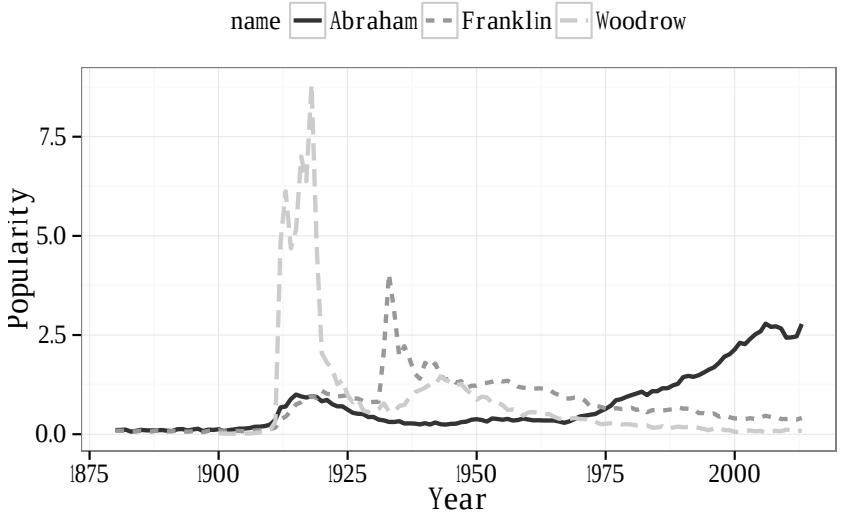


Figure A.1: A sketch of the popularity over time of a few names.

Developing a plan for wrangling

Step 1: Examine the data you have at hand — for this project, the data are the one table `BabyNames` — to find out what variables are available and what is the meaning of a case.

Step 2: Imagine what your end report will look like and sketch out your idea. Here, Figure A.1 will serve as the sketch of the goal.

Step 3: Analyze the graphic to figure out what a glyph-ready data table should look like. Mostly, this involves figuring out what variables are represented in the graph. Write down a small example of a glyph-ready data frame that you think could be used to make something in the form of the graphic.

- What variable(s) from the raw data table do not appear at all in the graph?
- What variable(s) in the graph are similar to corresponding variables in the raw data table, but might have been transformed in some way?

Step 4: Consider how the cases differ between the raw input and the glyph-ready table.

- Have cases been **filtered** out?
- Have cases been grouped and summarized within groups in any way?
- Have any new variables been introduced? If so, what's the relationship between the new variables and existing variables?

Step 5: Using English, write down a sequence of steps that will accomplish the wrangling from the raw data table to your hypothesized glyph-ready data table.

Step 6: Using paper and pen, translate your design, step by step, into R.

Step 7: Implement, test, and revise. Place your commands from Step (6) into a suitable document. Run them in R.

It's usual, particularly for beginners, but even for experts, that the wrangling sequence as first written won't work exactly as you anticipated. Among other reasons, this might be due to syntax errors in your R commands, or mis-matches when referring to variables, or because you unintentionally left out some or another data wrangling action.

Expect to follow a cycle of testing, diagnosing problems, revising, testing again, diagnosing, revising again, ... One effective strategy is divide and conquer; examine the output from early wrangling actions before adding in the actions to follow.

Once you have your glyph-ready data, make your graphic. You may find this template `ggplot()` command useful:

```
GlyphReadyForm %>%
  ggplot(aes(x = year, y = total, group = name)) +
  geom_line( size = 1, alpha = 0.5, aes(color = name)) +
  ylab("Popularity") + xlab("Year")
```

Bird Species

The `OrdwayBirds` data table is a historical record of birds captured and released at the Katharine Ordway Natural History Study Area, a 278-acre preserve in Inver Grove Heights, Minnesota, owned and managed by Macalester College. Originally written by hand in a field notebook, the entries have been transcribed into electronic format under the supervision of Jerald Dosch, Dept. of Biology, Macalester College.

Due to mistakes in data entry, the `SpeciesName` variable needs some fixing. `SpeciesName` is intended to identify the species of each of the birds, but the spelling often varies among birds of the same biological species. This leads to mis-classification of birds. There are also problems with the `Month` and `Day` variables; they are supposed to be numerical, but mistakes prevent them from being correctly identified as such.

Fortunately, all these errors are easy to correct. The data table `OrdwaySpeciesNames` collects together all the variant spellings. Entry by entry, each mis-spelling was translated (by a human) into a standardized spelling. Thus, `join()` can be used to correct the mis-spellings in the `OrdwayBirds` table.

You are going to look at the month-to-month presence of different species. Think of your assignment as creating a manual for birders to guide them to the correct time of year to visit Ordway to see a particular species.

Some simple data cleaning

There are many variables that you won't need for this activity, and you still have to fix the `Month` and `Day` variables. To keep things simple, cut and paste this command into a chunk at the start of the document.

Step 0 *Get the data table*

```
OrdwayBirds <-
  OrdwayBirds %>%
```

```
  select( SpeciesName, Month, Day ) %>%
  mutate( Month=as.numeric(as.character(Month)),
          Day=as.numeric(as.character(Day)))
```

The `mutate()` step is part of the data cleaning process, arranging `Month` and `Day` as numerical variables as originally intended by the folks entering the data.

Task 1. Including mis-spellings, how many different species are there in the `OrdwayBirds` data?

Make a data table that gives the number of distinct species in the `SpeciesNameCleaned` variable in `OrdwaySpeciesNames`.
You will find it helpful to use `n_distinct()` a reduction function, which counts the number of unique values in a variable.

Task 2 Use the `OrdwaySpeciesNames` table to create a new data table that corrects the mis-spellings in `SpeciesNames`. This can be done easily using the `inner_join()` data verb.

```
Corrected <-
  OrdwayBirds %>%
  inner_join( OrdwaySpeciesNames ) %>%
  select( Species=SpeciesNameCleaned, Month, Day ) %>%
  na.omit() ## cleaned up the missing ones
```

Look at the names of the variables in `OrdwaySpeciesNames` and `OrdwayBirds`.

- Which variable(s) was used for matching cases.
- What were the variable(s) that will be added .

Task 3 Count how many bird captures there are of each of the (corrected) species? You can call the variable that contains the count `count`. Arrange this into descending order from the species with the most birds, and look through the list.

Define for yourself a "major species" as a species with more than a particular threshold count. Set your threshold so that there are 5 or 6 species designated a major.

Filter to produce a data table with only the birds that belong to a major species. Save the output in a table called `Majors`.

Task 4 When you have correctly produced `Majors`, write a command that produces the month-by-month count of each of the major species. Call this table `ByMonth`.

Hint: Remember `n()`. Also, one of the arguments to one of the data verbs will be `desc(count)` to arrange the cases into descending order. Display the top 10 species in terms of the number of bird captures.

Hint: Remember that summary functions can be used case-by-case when filtering or mutating a data table that has been grouped.

Display this month-by-month count with a bar chart arranged in a way that you think tells the story of what time of year the various species appear. You can use `barGraphHelper()` to explore different possibilities. Use the "Show Expression" button in `mBar()` to create an expression that you can cut and paste into a chunk in your Rmd document, so that the graph gets created when you compile it.

Warning: `barGraphHelper()` should never be a statement in your Rmd file, it needs to be used interactively from the console.

Once you have the graph, use it to answer these questions:

1. Which species are present year-round?
2. Which species are migratory, that is, primarily present in one or two seasons?
3. What is the peak month for each major species?
4. Which major species that are seen in good numbers for at least 6 months of the year? (Hint: `n_distinct()` and `>= 6`.)

World Cities

Figure A.2 has two layers:

1. Country borders
2. A scatter plot mapping the urban population of each country to the area of a dot.

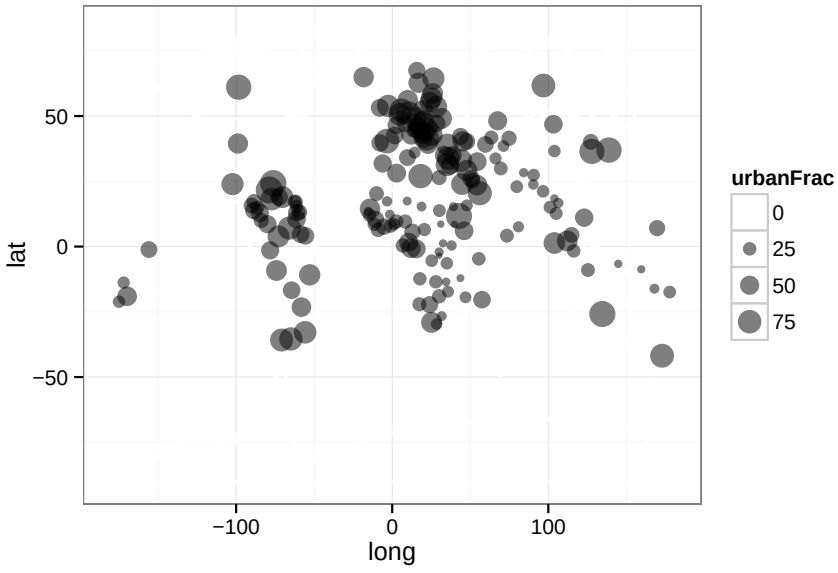


Figure A.2: The end result for this project: each country's urban population fraction shown geographically.

The data behind the graphic in Figure A.2 comes from distinct data tables.

- `WorldCities` gives the population of each city.
- `CountryData` gives, among other things, the population of each country.
- `CountryCentroids` gives the latitude and longitude of the center of each country.

`WorldCities`

code	country	population
3040051	AD	15853
3041563	AD	20430
290594	AE	44411
291074	AE	115949
291696	AE	33575
... and so on for 23,018 rows		

`CountryData`

country	area	pop
Afghanistan	652230	31822848
Akrotiri	123	15700
Albania	28748	3020209
Algeria	2381741	38813722
American Samoa	199	54517
... and so on for 256 rows		

- `Codes` translates between ISO-A2 and ISO-A3 codes.

Your task is to join these sources of information together to create a glyph-ready data table suited for making the scatterplot layer of Figure A.2.

Envision the Output

Analyze Figure A.2 to determine a suitable form for glyph-ready data:

- What variables form the frame?
- What is the glyph? (Ignore the country borders, which are really a guide, not a glyph.)
- What graphical properties does the glyph have? Which variables are mapped to those properties?

Sketch out the form of a glyph-ready data table.

- What are the variables?
- What is the physical meaning of a case?

Write the variable names and a case or two of made-up data in the usual rectangular format.

Origins of the variables

Each of the variables in the glyph-ready table originates in one of the four "raw" tables. Determine which table or tables contains the information needed to generate each variable in the glyph-ready table.

The meaning of a case

The meaning of "case" in the glyph-ready data matches that of three of the four original tables. Which table doesn't have the same meaning for case as the glyph-ready data?

Joining the tables

You're starting with four tables. To end up with the single glyph-ready data, you'll need to perform several joins, each of which involves two tables. For each of the joins,

- Specify the two tables to be involved.
- Name the variables from each table to be used for matching.

Give a name to the output table. (They are labelled A, B, and C below.)

`CountryCentroids`

name	iso_a3	long	lat
Afghanistan	AFG	66.17	33.78
Aland	ALA	19.97	60.20
Albania	ALB	20.26	41.14
Algeria	DZA	2.83	28.14
American Samoa	ASM	-170.72	-14.30
... and so on for 241 rows			

`Codes`

ISO2	ISO3
AF	AFG
AL	ALB
DZ	DZA
AD	AND
AO	AGO
... and so on for 194 rows	

Output Name	Table 1 Name	Table 2 Name	Variables to match
A			
.			
B			
.			
C			

Stocks and Dividends

Many companies are publicly traded. This means that the company issues stock certificates which can be bought and sold on an exchange. Investors buy stock certificates mainly because they can sell them in the future, perhaps making a profit, and because companies pay dividends, a share of the company's profit, to each shareholder.

Figures A.3 and A.4 show the scale of price fluctuations and of accumulated dividends.

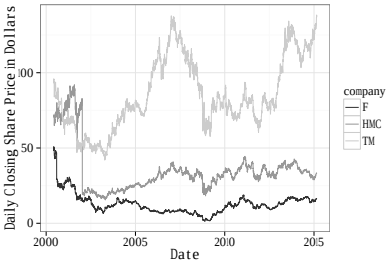


Figure A.3: Stock prices for Ford, Honda, and Toyota

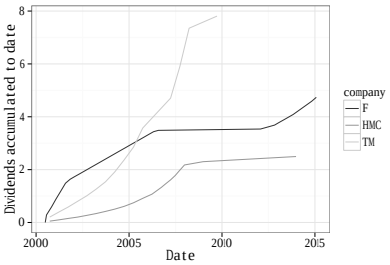


Figure A.4: Accumulated dividends (per share) paid by Ford, Honda, and Toyota

Task

Compare the income (or loss) that comes from buying and selling a stock certificate to the income that comes from dividends over the same period. Answer these questions:

- Which source of income is bigger?
- Is there any correlation between the income from dividends and the income (or loss) from buying and selling?

Getting Price Data

Sites such as `finance.yahoo.com` collect and distribute information about individual companies. You can use the `readStockPrices()` function (in the `DataComputing` package) to read such data directly from Yahoo into R.

For example, here are some automotive stocks and their daily prices from 2010 to 2015.

```
companies <- c("F", "TM", "HMC")
Prices <-
  read_stock_prices(companies, what="daily", start_year=2000, end_year=2015)
```

- Choose a few companies of interest to you. You can find stock company symbols at `finance.yahoo.com`. (Suggestion: pick a sector of the economy, e.g. energy, high-tech, consumer products, etc. and use companies from that sector.)

- Plot out the "closing price" (`Close`) versus `date` to get a graphic like Figure A.3.

Buy/Sell Profit

Pick a buy date and a sell date. You can use a command like this to create a Table like that shown in Table A.2.

```
Actions <-
  data.frame(
    action = c("buy", "sell"),
    date = ymd(c("2006-01-03", "2014-12-30")))
```

`Actions`

action	date
buy	2006-01-03
sell	2014-12-30

Table A.2: A suggested table format for the buy and sell dates.

Combine `Prices` and `Actions` to produce a table like `SalesDifference` in Table A.3:

> *Hints*: (1) What kind of join should you use so that you get only those cases that match one of the dates in the `Actions` table? (2) The wide-vs-long techniques in Chapter 11 will be useful.

From the data table with buy and sell prices, calculate the dollar amount of profit (or loss) and the percentage change, as in Table A.4.

`SalesDifference`

company	buy	sell
F	7.83	15.50
HMC	29.36	29.60
TM	106.85	125.91

Table A.3: Profits from buying at the start of 2006 and selling at the end of 2014. The dollar amount is profit per share.

Indexing Prices

Since stock prices vary markedly from one company to another, a common practice is to "index" the price to a particular date as in Figure A.5. (Question: In the graph, roughly which date was used for the reference?)

- Pick a single date of your choice and extract the stock price information for each company on that date. In the result, there should be one case for each company. Select just the `date`, `company`, and `close` variables, renaming `close` as `standard`. Call the resulting data frame `Reference`.

```
ref_date <- ymd("2005-01-03")
Reference <-
  Prices %>%
  filter(date==ref_date) %>%
  select(company, standard=close)
```

company	profit	percent
F	7.67	98.00
HMC	0.24	0.82
TM	19.06	17.80

Table A.4: Profit from each of the stocks.

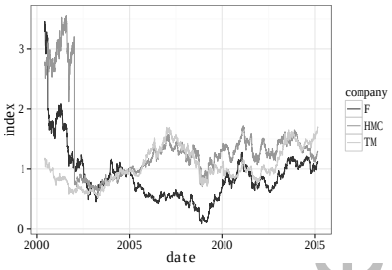


Figure A.5: Indexed stock prices for Ford, Honda, and Toyota

- You now need to combine the `Reference` with each day's price data for that company. You'll find the standardized price on each day by creating a new variable which is the ratio of the day-to-day price (use `Close`) to the `standard` for that company. Before you can do this, you'll need to combine the `Prices` and `Reference`

data tables. You'll use a *join* verb to do this. In order to check your results, sketch out what you think the result *should* be before you do the join.

Dividends

You can read in dividend data like this:

```
Dividends <-
  read_stock_prices(companies, what="dividends")
```

Once you have the dividend data, extract out the dividends for all dates between your buy and sell dates. (Hint: Join `Dividends` to `Actions` using `company` to match. The result will have two. When)

- The dividend amount is actually a rate: the dividend paid (in dollars) divided by the stock price. Find the dollar amount of each dividend payment for one *share* of stock rather than one *dollar* of stock. This involves multiplying the dividend rate by the stock price on that date.
- Find the total amount of dividends for each company during the period of interest. Compare this amount to the profit (or loss) from buying and selling the stock certificates. For the car companies, the result for the period 2005-01-01 though 2014-12-31 is shown in Table A.5.

company	total_dividend
F	1.75
HMC	1.89
TM	5.44

Table A.5: Dividends (in dollars per share) paid out by each company over the interval 2005 through 2014.

Project: Statistics of Gene Expression

Simple Graphics for Gene Expression

In the 1980s, the National Cancer Institute developed a set of 60 cancer cell lines, called NCI60. The original purpose was for screening potential anti-cancer drugs. Here you will examine gene expression in these cell lines. More than 41,000 probes were used for each of the 60 cell lines. For convenience, the data are provided by the `DCI` package in two data tables `NCI60` and `NCI60cells`.

`NCI60` is somewhat large — 41,078 probes by 60 cell lines. Each of these 2,454,680 entries is a measure of how much a particular gene was expressed in one cell line.

One clue that a gene is linked to cancer is differences in expression of that gene from one cancer type to another. Figure A.6 shows the average expression of TOP3A for the different cancer types. Here's the wrangling involved in extracting the expression of TOP3A for each cell line.

```
Narrow <-
  NCI60 %>%
  tidyr::gather(cellLine, expression, -Probe)
CellTypes <-
  NCI60cells %>%
  select(cellLine, tissue) %>%
  mutate(cellLine = gsub("\\:",".", as.character(cellLine)))
Narrow <-
  Narrow %>%
  inner_join(CellTypes)

Probe_TOP3A <-
  Narrow %>%
  filter(Probe=="TOP3A")
```

Now, the mean expression of TOP3A in each cancer tissue type:

```
SummaryStats <-
  Probe_TOP3A %>%
```

Probe	cellLine	expression	tissue
AT_D_3	BR.MCF7	-7.45	Breast
AT_D_5	BR.MCF7	-7.05	Breast
AT_D_M	BR.MCF7	-7.05	Breast
AT_L_3	BR.MCF7	-7.32	Breast
AT_L_5	BR.MCF7	-7.38	Breast
... and so on for 2,464,680 rows			

Table A.6: A narrow form of the expression data.

Probe	cellLine	expression	tissue
TOP3A	LE.SR	0.53	Leukemia
TOP3A	ME.M14	-0.93	Melanoma
TOP3A	BR.MCF7	-0.37	Breast
TOP3A	BR.T47D	-0.31	Breast
TOP3A	CO.HT29	-1.72	Colon
... and so on for 60 rows			

Table A.7: Expression in each cell line of the TOP3A Probe

```
  group_by(tissue) %>%
  summarise(mn_expr = exp(mean(expression, na.rm=TRUE)))
```

Figure A.6 shows the mean expression data graphically. This sort of bar graph is often seen in the scientific literature, but that does not mean it is an effective presentation.

```
SummaryStats %>%
  ggplot(aes(x = tissue, y = mn_expr)) +
  geom_bar(stat = "identity") +
  theme(axis.text.x = element_text(angle = 30, hjust=1))
```

To judge from the figure, TOP3A is expressed more highly in breast cancer than in other cancer tissue types.

But don't jump to conclusions. Expression differs even from one cell line to another of the same tissue type, as in Figure A.7:

```
Probe_TOP3A %>%
ggplot(aes(x=tissue, y=exp(expression))) +
  geom_point() +
  theme(axis.text.x = element_text(angle = 30, hjust=1))
```

When looking at the individual cell lines, it's not so clear that TOP3A is expressed differently in breast cancer compared to the other.

Before going on, decide what you like and don't like about Figure A.6. Write it down before you continue.

Remember, you shouldn't continue until you write down your opinions about Figure A.6.

... Really!

... Are you ready now?

... You've really written something?

... Then go on.

There are several bad features of this plot:

- Too much ink.
- The order of the levels of `tissue` is alphabetical. It's unlikely that the mechanisms of cancer consider what we call different types of cancer in English. So the x-axis order is being wasted.
- The precision of the feature (mean expression) is not shown. How would the viewer know whether this spread is just the result of random variation?

tissue	mn_expr
Breast	0.91
CNS	0.53
Colon	0.35
Leukemia	0.66
Melanoma	0.57
... and so on for 9 rows	

Table A.8: Mean expression of TOP3A in the NCI60 cell lines.

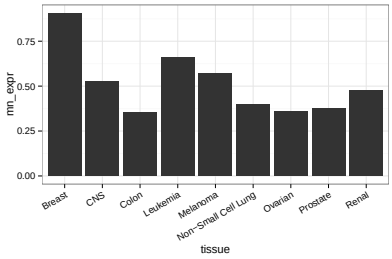


Figure A.6: A bar chart comparing TOP3A expression in the different tissue types.

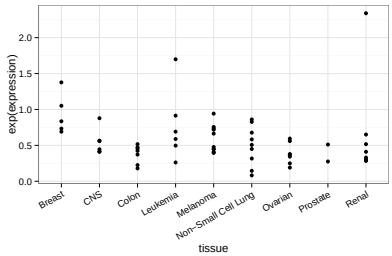


Figure A.7: TOP3A expression in the individual cells.

Here are several suggestions for improving the graphic:

1. Lighten up on the color. Perhaps `alpha=.2`. Or perhaps a dot plot rather than a bar chart.

2. Reorder the tissue types.

   One way to do this is to pick a quantity that you want to dictate the order of groups. For instance, the mean expression.

   - Create a data table of that quantity for each group (as with `SimpleStats`).
   - Reorder the categories based on the quantity:

     ```
     SummaryStats <-
       SummaryStats %>%
       mutate(tissue = reorder(tissue, mn) )
     ```

     Use `disc()` to order the other way.

3. Show a statistical measure of the variation.

   Without going into details of statistical method, a common way to present the imprecision in an estimated quantity is with a "standard error." For quantities such as the mean, the standard error has a straightforward mathematical form that is a staple of introductory statistics textbooks.

```
SummaryStats <-
  Probe_TOP3A %>%
  group_by(tissue) %>%
  summarise(mn = mean(expression,  na.rm=TRUE),
            se = sd(expression, na.rm=TRUE) / sqrt(n()))
```

4. Show the expression value for each of the individual cases in `MyProbe`.

5. Use a different modality, e.g. a dot plot, a box-and-whiskers plot (with `notch=TRUE`), a violin plot.

On occasion, you will want to generate two or more layers in a graphic that are based in more than one data table. You can do this within `ggplot()` by specifying a `data=` argument for the appropriate geom. For example, Figure A.8 is a bar plot of group means in one layer and a plot of individual cell lines in another layer:

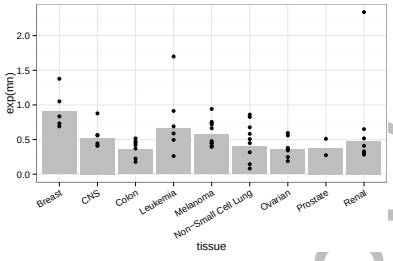


Figure A.8: A graphic with two layers. The bar chart shows the `SummaryStats` data table while the scatter plot shows the `Probe TOP3A` data table. Within a geom, the `data=` argument is used to specify the data set to use in drawing the geom. By default, this is the data table handed to `ggplot()` but it's easy to specify something different.

```
SummaryStats %>%
  ggplot(aes(x = tissue, y = exp(mn))) +
  geom_bar(stat = "identity", fill="gray", color=NA) +
  geom_point(data = Probe_TOP3A, aes(x=tissue, y=exp(expression))) +
  theme(axis.text.x = element_text(angle = 30, hjust=1))
```

The mean reflects the TOP3A expression collectively within each tissue type. As such, the mean is a *statistic*. In order to compare different tissues to one another, some indication must be given for the precision of the mean. Classically, this indication is the *confidence interval* at the 95-percent level. Without going into detail, here's a calculation and presentation of the confidence interval.

STATISTIC: A quantity giving a collective property for a set of cases.

```
SummaryStats <-
  SummaryStats %>%
  mutate(top = mn + 2 * se,
         bottom = mn - 2 * se)
SummaryStats %>%
  ggplot(aes(x = tissue, y = exp(mn))) +
  geom_bar(stat = "identity", alpha=0.2) +
  geom_errorbar(aes(x = tissue,
                    ymax = exp(top),
                    ymin =exp(bottom)), width=0.5) +
  theme(axis.text.x = element_text(angle = 30, hjust=1))
```

Statisticians refer to such things as "dynamite plots," in a way intended to be pejorative. You can find a better way to present these data.[1] One complaint is that the bars command too much attention. Another is that it's helpful to show the individual measurements along with the statistic, as in Figure A.10.

Your turn: Pick your own probe and make a figure like that of Figure A.10.

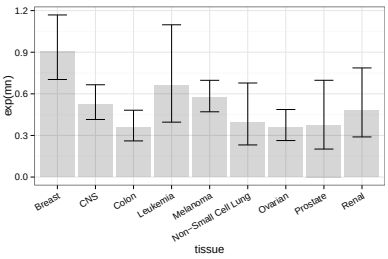


Figure A.9: A dynamite plot. You can do better!

[1] See Classic: Belia, et al. "Researchers Misunderstand Confidence Intervals and Standard Error Bars", Psych Methods, 2005 and Lane and Sandor: "Designing Better Graphs by Including Distributional Information and Integrating Words, Numbers, and Images", Psych Methods, 2009

Probing for a probe

There are 32,344 distinct probes in `NCI60`. TOP3A in the above example was selected literally at random. In this section, you'll identify candidate probes that might be more closely related to tissue type than TOP3A.

It's impractical to look through 32,344 different plots like Figure A.10. You need a way for the computer to evaluate each probe on its own.

The R^2 (pronounced "R-squared") statistic provides a measure of how much of the variation in one variable, called a response variable, is accounted for other, explanatory variables. R^2 is always between zero and one. Zero means that the explanatory variables account for nothing. One means that the explanatory variables account for all of the response.

You can use the tissue type as an explanatory variable, and use it to account for the level of expression. The function `r2()` calculates R^2.

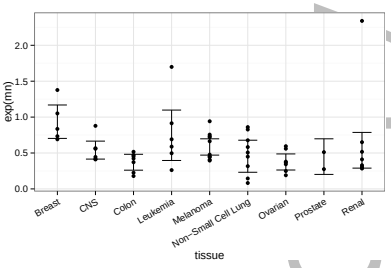


Figure A.10: Better than dynamite!

```
r2 <- function(data) {
  mosaic::rsquared(lm(data$expression ~ data$tissue))
}
```

One strategy for finding probes that are strongly connected with tissue type is to find the R^2 for each probe. This involves a few operators you have not seen. The `do()` in the following is analogous to `summarise()`. The `unlist()` function does a simple translation to put the results of `do()` in a form that can be graphed in the usual way.

```
ProbeR2 <-
  Narrow %>%
  group_by(Probe) %>%
  do(r2 = r2(.)) %>%
  mutate(r2 = unlist(r2))
```

Next, here are statements to pull out the 30 probes with the largest R^2 and order them from largest $R2$ to smallest.

```
Actual <-
  ProbeR2 %>%
  arrange(desc(r2)) %>%
  head(30) %>%
  mutate(Probe = reorder(Probe, desc(r2)))
```

Finally, the R^2 can be graphed, as in Figure A.11

```
Actual %>%
  ggplot(aes(x=Probe, y=r2)) +
  geom_point() +
  theme(axis.text.x = element_text(angle = 45, hjust=1))
```

Your turn: Choose one of the probes with high R^2. Plot out expression versus tissue type, just as Figure A.10 shows it for TOP3A. Do you see a qualitative difference between the graph of your high R^2 probe and Figure A.10?

False discoveries

A major concern when selecting results from tens of thousands of possibilities (or even tens of possibilities) is the possibility of false discovery. Even if the probe expression is unrelated to tissue type, examining the probes with the highest R^2 will select out those that, perhaps by chance, have a relationship. How to determine if the high R^2 is due to chance selection?

You can get an idea of what sort of role chance plays by examining a situation where only chance is at play. In statistics, this sort of

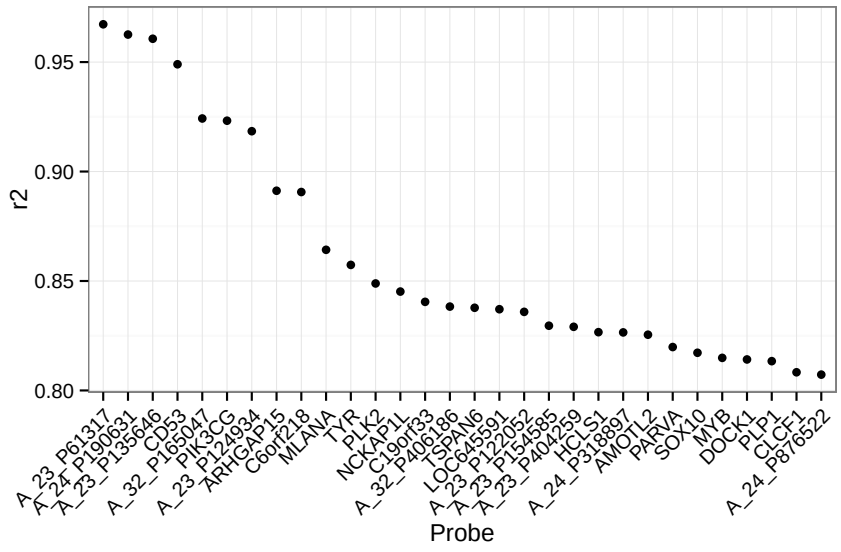


Figure A.11: Probes with the largest R^2 for expression level explained by tissue type.

situation is called the *Null Hypothesis*. By comparing the actual statistic (in this example, $R^2) to that from the Null Hypothesis, you can determine whether the observed statistic is likely to have arisen in a world where the Null Hypothesis is true. This probability is called the *p-value*. Only when this probability is small can you "reject the Null Hypothesis."

NULL HYPOTHESIS: A hypothetical setting where fluctuation in the response variable is due to chance.

P-VALUE: In a world where the Null Hypothesis is true, the probability of seeing a statistic as large as you observed in the actual data.

It's often believed that any p-value less than 0.05 calls for rejecting the Null Hypothesis. But when you are examining multiple possibilities, such as the 32,344 R^2 values for the different probes, 0.05 is a poor guide. Instead, statistical tests for "multiple comparisons" need to be performed.

The first step in finding a p-value is to create a world in which the Null Hypothesis is true. For this gene-expression example, an appropriate Null Hypothesis is that gene expression is unrelated to tissue type. Of course, we aren't able to create cancer cells where this is true, but there is a trick. Using the data at hand — which might or might not be consistent with a Null Hypothesis — you can generate new data which *must* be consistent with the Null. You do this by shuffling the data so that the expression levels are unrelated with tissue type.

Here's a set of statements for shuffling the expression data (that's what `mosaic::shuffle()` is doing) and finding the R^2 that would be found in the Null Hypothesis world.

```
NullR2 <-
  Narrow %>%
  group_by(Probe) %>%
```

```
  mutate(expression = mosaic::shuffle(expression)) %>%
  group_by(Probe) %>%
  do(r2 = r2(.)) %>%
  mutate(r2 = unlist(r2))
```

By comparing the distributions of R^2 from the actual data to the shuffled data, you can get an idea of the extent to which the actual data is different.

```
ProbeR2 %>%
  ggplot(aes(x=r2)) +
  geom_density(fill="gray30", color=NA) +
  geom_density(data=NullR2, aes(x=r2),
               fill="gray80", alpha=.75, color=NA)
```

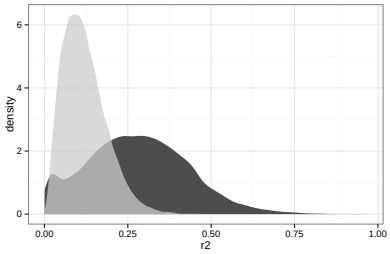


Figure A.12: Comparing the distribution of R^2 for the actual data (dark gray) to that for the null hypothesis data (light gray).

You can see from Figure A.12 that there are hardly any null-hypothesis probes with R^2 greater than 0.30. A conventional p-value would be misleadingly small for any such R^2. What's at issue is not whether the Null Hypothesis can produce $R^2 > 0.30$, but what the highest R^2 values will be when selected from 32,344 candidates. That sort of question can be answered by comparing the very highest R^2 probes from the Null to the highest R^2 from the actual data:

```
Null <-
  NullR2 %>%
  arrange(desc(r2)) %>%
  head(30)
Actual$null <- Null$r2
Actual %>%
  ggplot(aes(x=Probe, y=r2)) +
  geom_point() +
  geom_point(aes(y=null), color="gray50") +
  theme(axis.text.x = element_text(angle = 45, hjust=1))
```

You can see that none of the top 30 R^2 values for the actual data lie anywhere near those from the Null Hypothesis.

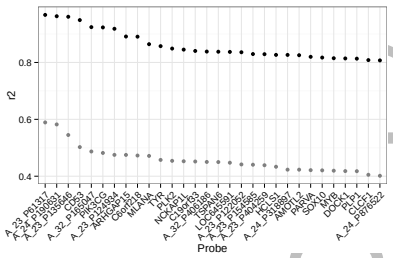


Figure A.13: Comparing the highest R^2 from the actual data and the Null.

Project: Bicycle Sharing

Capital BikeShare is a bicycle-sharing system in Washington, D.C. At any of about 400 stations, a registered user can unlock and check out a bicycle. After use, the bike can be returned to the same station or any of the other stations.

Figure A.14: Washington, D.C.

Such sharing systems require careful management. There need to be enough bikes at each station to satisfy the demand, and enough empty docks at the destination station so that the bikes can be returned. At each station, bikes are checked out and are returned over the course of the day. An imbalance between bikes checked out and bikes returned calls for the system administration to truck bikes from one station to another. This is expensive.

In order to manage the system, and to support a smart-phone app so that users can find out when bikes are available, Capital BikeShare collects real-time data on when and where each bike is checked out or returned, how many bikes and empty docks there are at each station. Capital BikeShare publishes the station-level information in real time. The organization also publishes, at the end of each quarter of the year, the historical record of each bike rental in that time.

You can access the data from the Capital Bikeshare web site. Doing this requires some translation and cleaning, skills that are introduced in Chapter (15). For this project, however, already translated and cleaned data are provided in the form of two data tables:

- `Stations` giving information about location of each of the stations in the system.

  ```
  Stations <- mosaic::read.file("http://tiny.cc/dcf/DC-Stations.csv")
  ```

- `Trips` giving the rental history over the last quarter of 2014.

  ```
  data_site <- "http://tiny.cc/dcf/2014-Q4-Trips-History-Data-Small.rds"
  Trips <- readRDS(gzcon(url(data_site)))
  ```

  The `Trips` data table is a random subset of 10,000 trips from the full quarterly data. Start with this small data table to develop your analysis commands. When you have this working well, you can

access the full data set of more than 600,000 events by removing `-Small` from the name of the `data_site`.

In this activity, you'll work with just a few variables:

- From `Stations`:
  - the latitude and longitude of the station (`lat` and `long`, respectively)
  - `name`: the station's name
- From `Trips`:
  - `sstation`: the name of the station where the bicycle was checked out.
  - `estation`: the name of the station to which the bicycle was returned.
  - `client`: indicates whether the customer is a `"regular"` user who has paid a yearly membership fee, or a `"casual"` user who has paid a fee for five-day membership.
  - `sdate`: the time and date of check-out
  - `edata`: the time and date of return

Notice that the location and time variable start with an "s" or an "e" to indicate whether the variable is about the start of a trip or the end of a trip.

Time/dates are stored as a special kind of number: a *POSIX date*. You can use `sdate` and `edate` in the same way that you would use a number. For instance, Figure A.15 shows the distribution of times that bikes were checked out.

POSIX DATE: A representation of date and time of day that facilitates using dates in the same way as numbers, e.g. to find the time elapsed between two dates.

```
Trips %>%
  ggplot(aes(x=sdate)) +
  geom_density(fill="gray", color=NA)
```

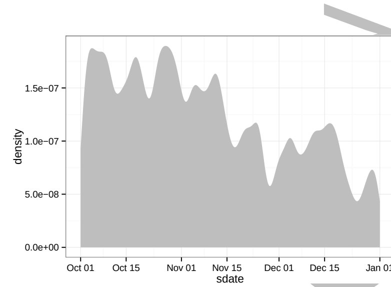


Figure A.15: Use of shared bicycles over the three months in Q3.

A.1 How long?

Your Turn: Make a box-and-whisker plot, like Figure A.16 showing the distribution of the duration of rental events, broken down by the `client` type. The duration of the rental can be calculated as `as.numeric(edate - sdate)`. The units will be in either hours, minutes, or seconds. You figure it out.

When you make your plot, you will likely find that the axis range is being set by a few outliers. These may be bikes that were forgotten. Arrange your scale to ignore these outliers, or filter them out.

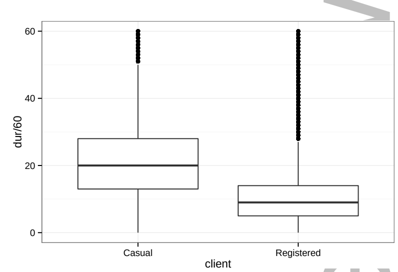


Figure A.16: The distribution of bike-rental durations as a box-and-whisker plot.

A.2 When are bikes used?

The `sdate` variable in `Trips` indicates the date and time of day that the bicycle was checked out of the parking station. `sdate` is stored as a special

Often, you will want discrete components of a date, for instance:

- The day of the year (1 to 365): use `lubridate:yday(sdate)`
- The day of the week (Sunday to Saturday): use `lubridate::wday(sdate)`
- The hour of the day: use `lubridate::hour(sdate)`
- The minute in the hour: use `lubridate::minute(sdate)`

Your Turn: Make histograms or density plots of each of these discrete components. Explain what each plot is showing about how bikes are checked out. For example, Figure A.17 shows that few bikes are checked out before 5am, and that there are busy times around the rush hour: 8am and 5pm.

```
Trips %>%
  mutate(H = lubridate::hour(sdate)) %>%
  ggplot(aes(x = H)) +
  geom_density(fill="gray", adjust=2)
```

A similar sort of display of events per hour can be accomplished by calculating and displaying each hour's count, as in Figure A.18.

```
Trips %>%
  mutate(H = lubridate::hour(sdate)) %>%
  group_by(H) %>%
  summarise(count = n()) %>%
  ggplot(aes(x=H, y=count)) +
    geom_point() + geom_line()
```

The graphic shows a lot of variation of bike use over the course of the day. Now consider two additional variables: the day of the week and the "client" type.

Your Turn: Group the bike rentals by three variables: hour of the day, day of the week, and client type. Find the total number of events in each grouping and plot this count versus hour. Use the `group` aesthetic to represent one of the other variables and faceting to represent the other. (Recall that faceting is done by adding `facet_wrap( ~var)` to the plotting commands.)

Your Turn (optional): Make the same sort of display of how bike rental vary of hour, day of the week, and client type, but use `geom_density()` rather than grouping and counting. Compare the two displays — one of discrete counts and one of density — and describe any major differences.

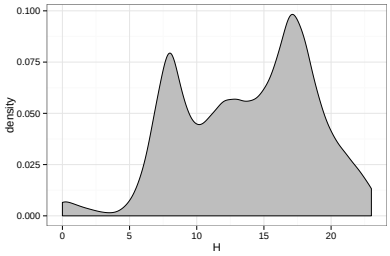


Figure A.17: Distribution of bike trips by hour of the day

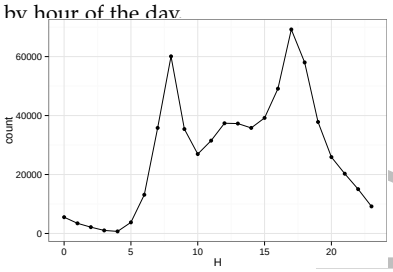


Figure A.18: The number of events in each hour of the day. Compare to the scale in Fig. effig:bikes-over-hours. Why are the scales different?

A.3 How far?

Find the distance between each pair of stations. You know the position from the `lat` and `long` variables in `Stations`. This is enough information to find the distance. The calculation has been implemented in the `haversine()` function:

```
source("http://tiny.cc/dcf/haversine.R")
```

`haversine()` is a transformation function. To use it, create a data table where a case is a *pair* of stations and there are variables for the latitude and longitude of the starting station and the ending station. To do this, join the `Station` data to itself. The following statements show how to create appropriately named variables for joining.

```
Simple <-
  Stations %>%
  select(name, lat, long) %>%
  rename(sstation=name)
Simple2 <-
  Simple %>%
  rename(estation=sstation, lat2=lat, long2=long)
```

Look at the `head()` of `Simple` and `Simple2` and make sure you understand how they are related to `Stations`.

The joining of `Simple` and `Simple2` should match every station to every other station. This sort of matching does not make use of any matching variables; everything is matched to everything else. This is called a *full outer join*.

Since a ride can start and end at the same station, it also makes sense to match each station to itself.

FULL OUTER JOIN: A join which matches every case in the left table to each and every case in the right table.

First, try the full outer join of just a few cases from each table, for example 4 from the left and 3 from the right.

```
merge(head(Simple, 4), head(Simple2, 3), by=NULL)
```

Make sure you understand what the full outer join does before proceeding. For instance, you should be able to predict how many cases the output will have when the left input has n cases and the right has m cases.

- There are 347 cases in the `Stations` data table. How many cases will there be in a full outer join of `Simple` to `Simple2`?

It's often impractical to carry out a full outer join. For example, joining `BabyNames` to itself with a full outer join will generate a result with more than three-trillion cases.

There are only a few computers in the world that are be able to hold three trillion cases from `BabyNames`. It's the equivalent of about 5 million hours of MP3 compressed music. A typical human lifespan is about 0.6 million hours.

Perform the full outer join and then use `haversine()` to compute the distance between each pair of stations.

```
StationPairs <- merge(Simple, Simple2, by=NULL)
```

Check your result for sensibility. Make a histogram of the station-to-station distances and explain where it looks like what you would expect. (Hint: from one end of Washington, D.C. to the other is about 14.1 miles.)

```
PairDistances <-
  StationPairs %>%
  mutate(distance = haversine(lat, long, lat2, long2)) %>%
  select(sstation, estation, distance)
```

Once you have `PairDistances`, you can join it with `Trips` to calculate the start-to-end distance of each trip.

Of course, a rider may not go directly from one station to another.

- Look at the variables in `Stations` and `Trips` and explain why `Simple` and `Simple2` were given different variable names for the station.

An `inner_join()` is appropriate for finding the distance of each ride.

```
RideDists <-
  Trips %>%
  inner_join(PairDistances)
```

Watch out! `Trips` and `PairDistances` are large enough that the join is expensive: it takes about a minute.

Your turn: Display the distribution of the ride distances of the rides. Compare it to the distances between pairs of stations. Are they similar? Why or why not?

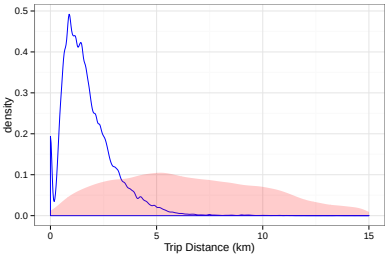


Figure A.19: The distribution of trip lengths compared to the distribution of distances between pairs of stations (shaded).

A.4 Mapping the Stations

You can draw detailed maps with the `leaflet` package. You may need to install it.

```
devtools::install_github("rstudio/leaflet")
```

`leaflet` works much like `ggplot()` but provides special facilities for maps. Here's how to make the simple map:

```
library(leaflet)
stationMap <-
  leaflet(Stations) %>%    # like ggplot()
  addTiles() %>%           # add the map
  addCircleMarkers(radius=2, color="red") %>%
  setView(-77.04, 38.9, zoom=12)
```

To display the map, use the object name as a command: `stationMap`

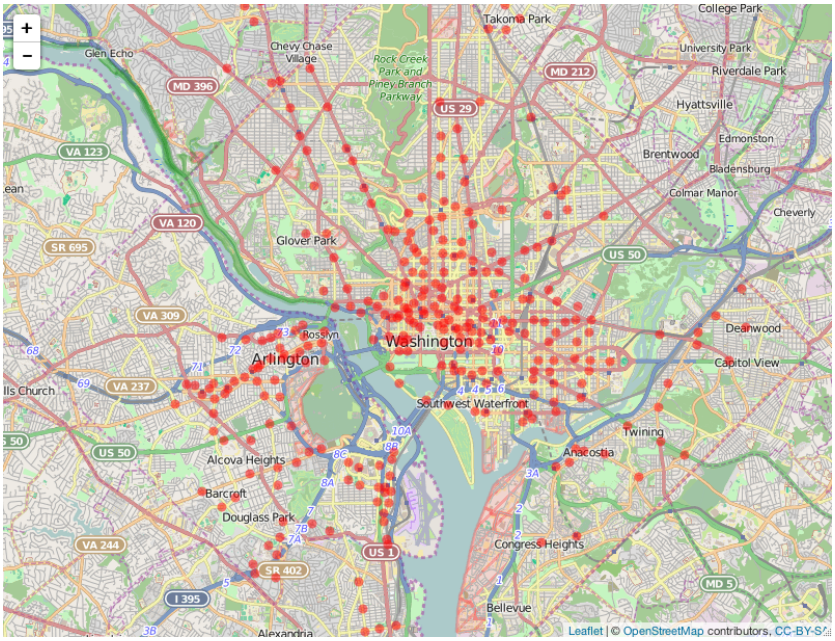


Figure A.20: A map of station locations drawn with the `leaflet()` function in the `leaflet` package.

A.5 Long-distance stations

Your Turn (optional, but cool): Around each station on the map, draw a circle whose radius reflects the median distance covered by rentals starting at that station. To draw the circles, use the same `leaflet` commands as before, but add in a line like this.

```
addCircles(radius = ~ mid, color="blue", opacity=0.001)
```

For `addCircles()` to draw circles at the right scale, the units of the median distance should be presented in meters rather than kilometers. This will create too much overlap, unfortunately. So, set the radius to be half or one-third the median distance in meters. From your map, explain the pattern you see in the relationship between station location and median distance.

Project: Inspecting Restaurants

Restaurants are generally inspected for health-related practices: food handling and temperature, personal hygiene of the workers, and control of cockroaches and other vermin. Based on the inspection results, local governments may mark the restaurant as passing, put the restaurant on notice, or even close the restaurant.

In New York City, such inspections are conducted and recorded by the Department of Health and Mental Hygiene, which provides a letter grade to each restaurant based on the number and type of violations.

In the era of big data, governments sometimes post their inspection results. In New York City, the record published at `https://data.cityofnewyork.us` includes each individual health violation.

Figure A.21: Orange County, California, restaurant inspection notifications.

Figure A.22: A New York City restaurant displaying its health inspection grade.

Accessing the NYC restaurant inspections data

Execute the commands in this section now, before reading on. It will take about 2 minutes for the last one. Then, while you're waiting, read the rest of this activity. You'll know it's working if `Violations`, `Cuisines`, `ViolationCodes` show up in your account.

You will use four data tables in this exercise:

1. `ZipGeography` in the `DataComputing` package.
2. `Violations` which you must load into R.

```
load(url("http://tiny.cc/dcf/Violations.rda"))
```

3. `Cuisines` ..... ditto ...

```
load( url( "http://tiny.cc/dcf/Cuisines.rda" ) )
```

4. `ViolationCodes` ..... ditto ...

```
load(url("http://tiny.cc/dcf/ViolationCodes.rda"))
```

The data table `Violations` contains information about each health violation.

- `CAMIS` a file code assigned to each restaurant
- `DBA` name of the restaurant.
- `BORO` Which of the five boroughs of New York City the restaurant is located in — Brooklyn, Bronx, Manhattan, Queens, Staten Island — coded as a number 1 to 5.
- `BUILDING`, `STREET`, `ZIPCODE` components of the address of the restaurant.
- `CUISINECODE` the sort of food served by the restaurants. (See the `Cuisines` data set.)
- `INSPDATE` the date of the inspection
- `CURRENTGRADE` the restaurant's grade *after* the inspection.
- `GRADEDATE` the date when the grade was issued
- `VIOLCODE` the violation being recorded. See `ViolationCodes` for explanations.
- `SCORE` the number of demerit points given for the violation.

Quick questions about the data

- What is the meaning of a case in this data table? You can look at the data table with this command: `View( Violations )`.
- How many cases are there?
- How many distinct restaurant names are there?
- Are there any restaurants with the same name but with multiple branch locations? Select one or more variables that plausibly identify a unique branch.
- Which restaurants (individual branches) have the most violations?
- For each restaurant, how many critical violations were reported? Non-critical?
- Make three related charts:

1. the distribution across restaurants of total scores
2. the same, but consider only the score for critical violations.
3. the same, but just for non-critical violations.

In each of the charts, display the score distribution separately for restaurants with different grades.

- Make an interactive map showing the location of restaurants with poor grades. To specify the location, use the restaurant's ZIP code. Get the latitude and longitude of each zip code from `ZipGeography`.

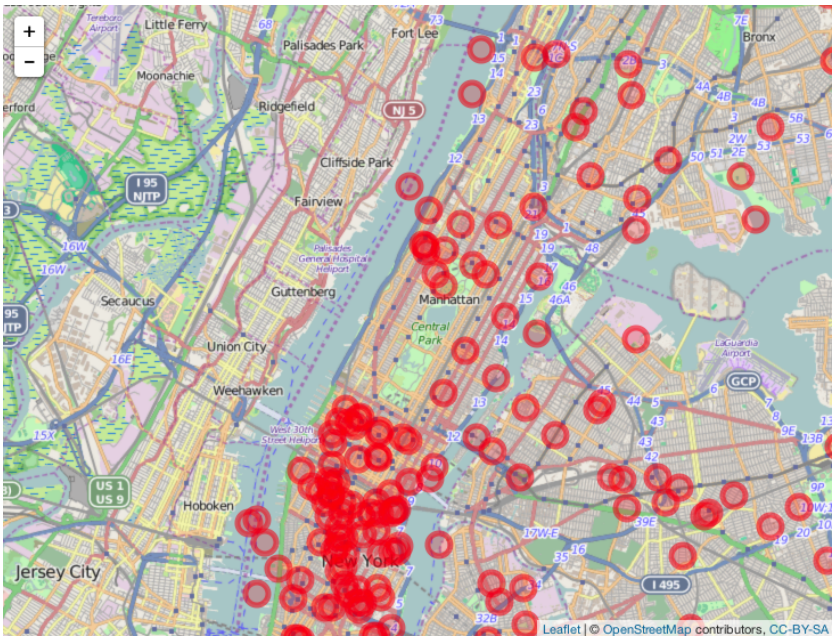


Figure A.23: Locations of C-graded restaurants.

A World of Cuisines

The data table `Cuisines` details the code for each restaurant's cuisine type.

- Judging from the number of restaurants, what's the most common cuisine type in the whole city? In each borough? In each zip code?
- Make an interactive map showing the locations (by ZIP code) of restaurants with a cuisine you select.

Street or Road?

address
PO BOX 74
16 COVEWOOD ROAD
PO BOX 573
103 DANBURY ST SW
PO BOX 933
NCSU BOX 22425
PO BOX 37322
PO BOX 1125
PO BOX 2362
NCSU BOX 15644
NCSU BOX 03349
209 ODUM AVE
P O BOX 43
PO BOX 4202
NCSU BOX 15102
... and so on for 15,483 rows

People's addresses involve streets, lanes, courts, avenues, and so on. How many such road-related words are in common use?

In answering this question, you would presumably want to look at lots of addresses and extract the road-related term. You could do this by eye, reading down a list of a few hundred or thousand addresses. But if you want to do it on a really large scale, a city or state or country, you would want some automated help, for instance, a computer program that discards the sorts of entries you have already identified to give a greater concentration of unidentified terms. In this activity, you're going to build such a program.

Some resources:

1. The file `http://tiny.cc/dcf/street-addresses.csv` contains about 15000 street addresses of registered voters in Wake County, North Carolina.

2. The file "`http://tiny.cc/dcf/CMS_ProvidersSimple.rds`" has street address of about 900,000 medicare service providers. Download the file to save it on your own system, then read it in under a convenient name.

```
download.file(url="http://tiny.cc/dcf/CMS_ProvidersSimple.rds",
              destfile = "YourNameForTheFile.rds")
DataTable <- readRDS("YourNameForTheFile.rds")
```

To solve such problems, start by looking at a few dozen of the addresses to familiarize yourself with common patterns. With those few dozen

1. In everyday language, describe a pattern that you think will identify the information you are looking for.

2. Translate (1) into the form of a regular expression.

3. Filter to retain the cases that match the expression. Hint: `filter()` and `grepl()` are useful for this.

4. Filter to retain the cases that do not match the expression.
5. Examine the results of (2) and (3) to identify shortcomings in your patterns.
6. Improve or extend the pattern to deal with the mistaken cases.
7. Repeat until satisfied.
8. Put extraction parentheses around the parts of the regular expression that contain the info you want.

Solved Example:

Suppose you wanted to extract the PO Box number from an address.

Read the street address data and pull out a sample of a few dozen cases.

```
Addresses <- read.file("http://tiny.cc/dcf/street-addresses.csv")
Sample <- Addresses %>%
  sample_n(size = 50)
```

Following each of the steps listed above:

1. The PO Box cases tend to have a substring "PO".
2. The regular expression for "PO" is simply `"PO"`.
3. Find some cases that match:

```
Matches <-
  Sample %>%
  filter(grepl("PO", address))
```

address
PO BOX 74
PO BOX 573
PO BOX 933
PO BOX 37322
PO BOX 1125
... and so on for 22 rows

4. Find cases that don't match:

```
Dont <-
  Sample %>%
  filter( ! grepl("PO", address))
```

address
16 COVEWOOD ROAD
103 DANBURY ST SW
NCSU BOX 22425
NCSU BOX 15644
NCSU BOX 03349
... and so on for 28 rows

5. Find any cases in the `Matches` that shouldn't be there. (None are seen in this example.) Find any cases in `Dont` that should have matched. (There are several, three of which appear in the table to the right.)
6. It looks like "BOX" is a better pattern. Since the box number is wanted, the regex should include an identifier for the number inside extraction parentheses. So `"BOX\\s+(\\d+)"`.

Note the double slashes, `\\`, in `\\s` and `\\d`. Ordinarily, `\` is a special character in R character strings used to designate special characters like new-line `\n` or tab `\t`. The double `\\` means, "just an ordinary slash, please." Confusing. But whenever characters are used to signal something special, you have to take an extra step to say that you don't want the special meaning.

```
pattern <- "BOX\\s+(\\d+)"
Matches <-
  Sample %>%
  filter(grepl(pattern, address))
Dont <-
  Sample %>%
  filter( ! grepl(pattern, address))
```

address
16 COVEWOOD ROAD
103 DANBURY ST SW
209 ODUM AVE
6221-104 ST REGIS CIRCLE
233 RIVERBEND RD
... and so on for 16 rows

The result seems satisfactory.

So, use `tidyr::extract()` to pull out the part of the pattern identified by extraction parentheses.

```
BoxNumbers <-
  Sample %>%
  filter(grepl(pattern, address)) %>%
  tidyr::extract(address, into="boxnum", regex=pattern)
```

boxnum
74
573
933
22425
37322
... and so on for 34 rows

Note that `tidyr::extract()` should be given only those cases that match the regular expression, so `filter()` is applied before `tidyr::extract()`

Back to the Streets

Street endings (e.g. "ST", "LANE") are often found at the end of the address string. Use this as a starting point to find the most common endings.

Once you have a set of specific street endings, you can use the regex "or" symbol, e.g. `"(ST|RD|ROAD)"`. The parentheses are not incidental. They are there to mark a pattern that you want to extract. In this case, in addition to knowing that there is a `ST` or `RD` or `ROAD` in an address, you want to know which one of those possibilities it is so that you can count the occurance of each of the possibilities.

To find street endings that aren't in your set, you can filter out the street endings or non-street addresses you already know about.

Your turn: Read the following R statements. Next to each line, give a short explanation of what the line contributes to the task. For each of the regexes, explain in simple everyday language what pattern is being matched.

```
pattern <- "(ST|RD|ROAD)"
LeftOvers <-
  Addresses %>%
  filter( ! grepl(pattern, address),
          ! grepl("\\sAPT|UNIT\\s[\\d]+$", address),
          ! grepl(" BOX ", address)
  )
```

address
2117 MARINER CIRCLE
101 EPPING WAY
04-I ROBIN CIRCLE
NCSU BoX 15637
4719 BROWN TRAIL
... and so on for 2,411 rows

For each set of patterns that you identify, compute the `LeftOvers`. Examine them visually to find new street endings to add to the pattern, e.g. `LANE`.

When you have this working on the small sample, use a larger sample and, eventually, the whole data set. It's practically impossible to find a method that will work perfectly on new data, but do the best you can.

Your turn: In your report, implement your method and explain how it works, line by line. Present your result: how many addresses there are of each kind of road word.

For the professional . . .

Breaking addresses into their components is a common task. People who work on this problem intensively sometimes publish their regular expressions. Here's one from Ross Hammer published at `http://regexlib.com/Search.aspx?k=street`

```
^\s*((?:(?:\d+(?:\x20+\w+\.?)+(?:(?:\x20+STREET|ST|DRIVE|DR|AVENUE|AVE|ROAD|RD|LOOP|COURT
|CT|CIRCLE|LANE|LN|BOULEVARD|BLVD)\.?)?)|(?:(?:P\.\x20?O\.|P\x20?O)\x20*Box\x20+\d+)|
(?:General\x20+Delivery)|(?:C[\\\/]O\x20+(?:\w+\x20*)+))\,?\x20*(?:(?:(?:APT|BLDG|DEPT|
FL|HNGR|LOT|PIER|RM|S(?:LIP|PC|T(?:E|OP))|TRLR|UNIT|\x23)\.?\x20*(?:[a-zA-Z0-9\-]+))|
(?:BSMT|FRNT|LBBY|LOWR|OFC|PH|REAR|SIDE|UPPR))?)\,?\s+((?:(?:\d+(?:\x20+\w+\.?)+
(?:(?:\x20+STREET|ST|DRIVE|DR|AVENUE|AVE|ROAD|RD|LOOP|COURT|CT|CIRCLE|LANE|LN|BOULEVARD|
BLVD)\.?)?)|(?:(?:P\.\x20?O\.|P\x20?O)\x20*Box\x20+\d+)|(?:General\x20+Delivery)|
(?:C[\\\/]O\x20+(?:\w+\x20*)+))\,?\x20*(?:(?:(?:APT|BLDG|DEPT|FL|HNGR|LOT|PIER|RM|
S(?:LIP|PC|T(?:E|OP))|TRLR|UNIT|\x23)\.?\x20*(?:[a-zA-Z0-9\-]+))|(?:BSMT|FRNT|LBBY|
LOWR|OFC|PH|REAR|SIDE|UPPR))?)?\,?\s+((?:[A-Za-z]+\x20*)+)\,\s+(A[LKSZRAP]|C[AOT]|
D[EC]|F[LM]|G[AU]|HI|I[ADLN]|K[SY]|LA|M[ADEHINOPST]|N[CDEHJMVY]|O[HKR]|P[ARW]|RI|
S[CD]|T[NX]|UT|V[AIT]|W[AIVY])\s+(\d+(?:-\d+)?)\s*$
```

Scraping Nuclear Reactors

In this project,[2] you're going to look at data about nuclear reactors. Let's use Japan as an example. Often, when you are doing a quick project, sources like Wikipedia are useful.

Go to the page `http::://en.wikipedia.org/wiki/List_of_nuclear_reactors"`. Find the reactor list for Japan. Figure A.24 shows part of the list[3] as a cut-and-paste image from a web browser.

[2] Devised initially by Prof. Nicholas Horton, Amherst College

[3] on March 23, 2015

Name	Reactor	Reactor		Status	Capacity in MW		Construction Start Date	Commercial Operation Date	Closure
		Type	Model		Net	Gross			
Fukushima Daiichi	1	BWR	BWR-3	Shutdown	439	460	25 Jul, 1967	26 Mar, 1971	19 May 2011
Fukushima Daiichi	2	BWR	BWR-4	Shutdown	760	784	09 Jun, 1969	18 Jul, 1974	19 May 2011
Fukushima Daiichi	3	BWR	BWR-4	Shutdown	760	784	28 Dec, 1970	27 Mar, 1976	19 May 2011
Fukushima Daiichi	4	BWR	BWR-4	Shutdown	760	784	12 Feb, 1973	12 Oct, 1978	19 May 2011

Figure A.24: Part of the Wikipedia table describing nuclear reactors in Japan.

Unfortunately, it is not a matter of cut-and-paste to get the tables in Wikipedia into the form of a data table in R. The tables often have a complex, non-tidy form. In addition, the tables are written using HTML tags, which can have be confusing. For instance, here a bit of the HTML behind the table of reactors in Japan.

```
<table class="wikitable sortable">
<tr>
<th rowspan="2" style="background:#FFDEAD;">Name</th>
<th rowspan="2" style="background:#FFDEAD;">Reactor</th>
<th colspan="2" style="background:#FFDEAD;">Reactor</th>
<th colspan="2" style="background:#FFDEAD;">Capacity in MW</th>
...
</tr>
<tr>
<td>Fukushima Daiichi</td>
<td>1</td><td>BWR</td><td>BWR-3</td><td>Shutdown</td>
```

```
<td>439</td><td>460</td><td>25 Jul, 1967</td>
<td>26 Mar, 1971</td><td>19 May 2011</td>
<tr>
```

Compare the human-readable version of the table with the HTML markup. You'll see that the data is there, but there is a lot of extraneous material and the arrangement is set not by position in a spreadsheet layout but by *HTML tag*s like `<td>` and `<tr>`.

HTML TAG: A markup indicator, analogous to `*` or `###` or `[text](line)` in Markdown.

```
library(rvest)
library(lubridate)
page <- "http://en.wikipedia.org/wiki/List_of_nuclear_reactors"
xpath <- '//*[@id="mw-content-text"]/table'
table_list <- page %>%
  read_html() %>%
  html_nodes(xpath = xpath) %>%
  html_table(fill = TRUE)
```

The `result` object is not a data table; it is a *list* of data tables. Here are some of the operations you can apply to lists:

Description	Syntax	Example
How many elements in the list	`length(`*table*`)`	`length(tableList)`
Grab a single element	*table*`[[`*element number*`]]`	`tableList[[20]]`

1) Find the table element

Start with `head(tableList[[5]])` and go down the list until you find the table for Japan. The tables are listed by number in the same order that they appear on the page. As of the time of this writing,[4] `tableList[[5]]` is for Austria, so you'll have to go a good distance down the table to get to Japan.

[4] Wikipedia articles are works in progress. Over a period of even a few days they may have been modified substantially.

2) The table will look like this:

Your turn: In what ways is the table tidy? How is it not tidy? What's different about it from a tidy table?

Once you've answered the above questions ... and only then ... continue reading.

Among other things, two of the variables names are missing and others have multiple words separated by spaces. You can rename them using the data verb `rename()`, finding the names from the Wikipedia table. Another problem is that the first row is not data but a continuation of the variable names. So row number 1 should be dropped.

```
names(Japan)[c(4,7)] <- c("model", "grossMW")
Japan <-
  Japan %>%
  filter(row_number() > 1) %>%
  rename(name = Name, reactor = `Reactor No.`,
         type = Reactor,
         status = Status, netMW = `Capacity in MW`,
         construction = `Construction Start Date`,
         operation = `Commercial Operation Date`, closure = Closure)
```

This sort of variable-name cleaning is common. But it's not the only sort of reformatting that's needed here. Look at each of the variables and decide what the data type is: character, numerical, date, etc. Now use `str()` to see how the variable is typed in the data table itself.

You are going to need to `mutate()` the variables that are not in the right type. Some suggestions:

1. To convert a character string of digits into a number, use `as.numeric()` or `as.integer()`.

2. The `lubridate` package functions can be used to turn character string dates into a *POSIXct date object*. Identify what the format of the date is. The `lubridate` translation functions are `mdy()`, `mdyhms()`, `dmy()`, and so on.

POSIXct date object: A type of R object representing points in time and allowing plotting, mathematical operations and extraction of components (such as the year or day of the week).

Your turn: Your cleaned data, make a plot of net generation capacity versus date of construction. Color the points by the *type* of reactor, e.g., BWR, PWR, or FBR.[5] In addition to your plot, give a sentence or two of interpretation; what patterns do you see?

[5] Boiling water reactor, pressurized water reactor, fast breeder reactor, respectively

Your turn: Carry out the same cleaning process for the China reactor table and append it with the Japan data. (Hint: You'll want one of the functions `cbind()` or `rbind()`.) You'll also want to add a variable to each table that has the name of the country.

Collating the data for all countries is a matter of repeating this process over and over. (You don't have to do this.) Inevitably, there are inconsistencies. For example, the US data is somewhat different in format than Japan or China.

Your turn: Read in to R the table on the Wikipedia page for US reactors. What is the physical meaning of a case in the US table? How does it compare to the meaning of a case for the Japan or China data.

Your turn: Make an informative graphic similar to Figure A.25 that shows how long it took between start of construction and commissioning for operation of each nuclear reactor in Japan (or another

country of your choice). One possibility: use reactor name vs date as the frame. For each reactor, set the glyph to be a line extending from start of construction to commissioning. You can do this with `geom_segment()` using name as the y coordinate and time as the x coordinate.

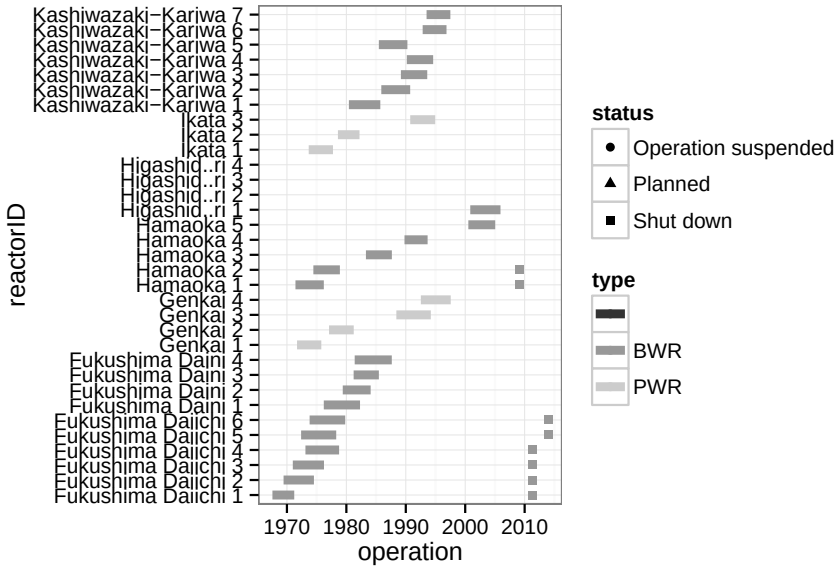


Figure A.25: Time interval from start of construction to operation.

B
Solutions to Selected Exercises

Chapter 1

Problem 1.2 Table 1.7 is not so tidy.

1. The values of the `president` variable are not all in the same form. The entry for Lincoln is in the form last-name, first-name, while the other values are not. It might be appropriate to divide the name into two variables: given name and surname. For instance, "Van" is not the middle name of Martin Van Buren; it's part of his surname "Van Buren."
2. The `in office` variable contains two numbers, with different punctuation between them. It would be appropriate to split `in office` into two variables, one for the start year and the other for the end year.
3. The `number of states` is not a number in Abraham Lincoln's case. (The situation changed rapidly during the US Civil War.) All the values in a variable should be the same kind of thing.

A bit more detail — one rule for tidy data is that all of the values for a variable should be the same kind of thing. You could argue that "1861-1865" and "1837 to 1841" are the same kind of thing: a set of characters that can be interpreted by a person. Don't buy it. It's much better, in general, if numerical quantities (like the year) are represented as numbers.

Problem 1.4

1. The variables are:
   - Table **A**: Year, Algeria, Brazil, Columbia
   - Table **B**: Country, Y2000, Y2001
   - Table **C**: Country, Year, Value
2. The cases are:
   - Table **A**: a year
   - Table **B**: a country
   - Table **C**: a country in a year

Problem 1.5
Each case is an airport. There are seven variables: `faa` (categorical), `name` (categorical), `lat` (numerical), `lon` (numerical), `alt` (numerical), `tz` (more or less numerical: data about times can be complicated), `dst` (categorical)

Chapter 2

Problem 2.1
The R expression is:

```
Engines <-
  data.table::fread("http://tiny.cc/mosaic/engines.csv")
```

Here's an example of an answer that uses the words listed (shown in *italics*) to describe the expression.

> `data.table` is a *package* that defines the *function* `fread()`. The function takes as an *argument* a *quoted character string* identifying the file to be read. The *object* created by `fread()` is a *data table* that is being *assigned* the *object name* `Engines`.

Problem 2.3
Current Population Survey

Problem 2.6

1. `essay14`: no problems
2. `first-essay`: a dash (`-`) is not one of the allowed punctuation marks in an object name.
3. `"MyData"`: Being in quotes, `"MyData"` is a constant, not an object name.
4. `third_essay`: no problems. An underscore is legitimate in an object name
5. `small sample`: a space is not allowed in an object name
6. `functionList`: no problems

7. `FuNcTiOnLiSt`: no problems. Admittedly, it's a perverse and hard to type name, but it's legal
8. `.MyData.`: no problems. Periods are allowed in a function name. It doesn't matter where they occur. It would even be legal to use and name like this: `.....<- 7`. But this is bad style!
9. `sqrt()`: parentheses are not allowed in function names. In the text of this book, the author uses parentheses when referring to a function. That's just to help remind you that the object name, in this case `sqrt` is referring to a function as opposed to a data table or variable.

Chapter 3

Problem 3.1

1. a data frame: Put it at the start of a chain, e.g. `fireplace %>% nrow()`
2. a function: Follow it by an open parenthesis, `fileplace(`
3. the name of a named argument. Follow it by = inside the parentheses of a function, e.g., `fun(fileplace = 7)`
4. a variable: place it inside the parentheses of a function, but not in the position of the name of a named argument, e.g., `fun(fileplace)` or `fun(x = fireplace)`

Although we encourage you to start names of data table with a capital letter, this is not a requirement of R syntax.

Problem 3.3
The named argument `package` should be used with the equal sign (=) as in `package = "NHANES"`

Problem 3.5

a) mistake 3; b) mistake 1; c) mistake 4; d) mistake 2; e) no mistake

Problem 3.6
Statement (c) is different. Rather than calculating the total number of births (using `sum()`), it finds the mean number of births across all the cases in each year/sex group.

Chapter 4

Problem 4.1

1. Put the word "one" in italics.
2. Put the word "two" in bold face.
3. An item in a bullet point list
4. A first-level header
5. The word "five" in a computer font, that is, `five`
6. A second-level header
7. A web link showing the word "seven" and linking to the `tiny.cc` URL for this book.

Problem 4.2

a. This uses the apostrophe rather than the back-quote.
b. Two mistakes. 1) Uses the quotation mark rather than the back-quote. 2) Curly braces rather than parentheses are needed.
c. There are only two back-quotes in the closing fence.
d. The command and the fences must be on separate lines.
e. Four back-quotes in the closing fence.

Problem 4.3
The browser window will render the HTML to look like this:

An Introduction

Arithmetic is *easy*! For instance

```
3 + 2
```

```
[1] 5
```

Problem 4.4

1. `DataComputing.org`: both. It's a proper URL but it can also be the name of a file.
2. `ahab/whale.Rmd`: a file only. The relative path is `ahab` with the file `whale.Rmd` in the `ahab` folder
3. `ptth://world-bank.org`: neither a valid file name nor a URL. `ptth:` is not a recognized prefix.
4. `http://world-bank.org`: a valid URL. It cannot be a file name since `//` is not an allowed fragment of a path.
5. `//world-bank.org/index.html`: most browsers will not recognize the leading `//` as valid in a URL. It's not valid in a file name either.
6. `world-bank.org/index.html`: a valid file name (`index.html` in the `world-bank.org` folder) or a valid URL.

Chapter 5

Problem 5.1

- `sex` is mapped to the x-axis.
- `wage` is mapped to the y-axis.
- `married` is mapped to the fill color.
- The different fill colors are positioned to be "dodged" rather than "stacked."
- The graph is facetted based on `race`.

Problem 5.3
This is a density plot made with

```
distributionGraphHelper(NCHS)
```

- `bmi` is mapped to the x axis
- The y-axis is set by the plot type, "density"
- `sex` determines the color (shown as level of gray in the printed version)
- facetting is based on `pregnant`

Problem 5.4
This is a scatterplot made with the help of `scatterGraphHelper()`.

- `exper` is mapped to the x axis
- `wage` is mapped to the y axis
- `married` sets color (shown as grayscale in the printed version)
- faceting is based on `sector`
- both the x- and y- axes have been set to logarithmic scales.

Chapter 6

Problem 6.1

1. Facet labels: 1433B,1433E, 1433G, 1433Z
2. No. The scale of the vertical axis differs among the frames.
3. The glyph types are:
   - A vertical "error bar"
   - A path connecting the mid-points of the error bars.
   - Single and double stars.

Comment: The usual scientific vernacular is that the error bars show the spread of several measurements and the stars indicate whether there is a statistically "significant" difference between the measurements marked and some reference condition. Critique: It might be better to indicate the individual measurements rather than summarizing them using an error bar. There's no particularly good reason for the path. The reference condition isn't identified in the graph.

Problem 6.3

1. The star glyph has these attributes: y-position, x-position, number of stars — 0, 1, or 2, to judge from the graph. The "error bar" glyph has the y-position of the center dot, the x-position of protein, the length of the bars reaching up and down, the label naming the protein, and color indicating the polarity. (This last is hard to see.)
2. The y-axis is labelled as cell density. In the data, it corresponds to the `center` variable. The x-axis is `protein`, a categorical variable.
3. Color is *not* an attribute of the ** glyph.
4. Guides: the tick marks on the y-axis. There is no guide right on the x-axis. The labels on the error bar glyphs are effectively a guide to the x-axis.

Problem 6.5
State (e.g. New Hampshire) on the vertical axis, Polling organization (e.g. NYT) on the horizontal axis.

Chapter 7

Problem 7.1

a) `summarise()` b) `mutate()` c) `arrange()` d) `filter()` e) `select()` f) `group_by()` and `summarise()`

Problem 7.3

a. summary function
b. neither a summary nor a transformation. It is a pair of values rather than a single value as required for a summary function.
c. summary function
d. transformation function
e. transformation function
f. summary function
g. transformation function
h. summary function: an entire set of dates is being turned into a single number.

Problem 7.5

a. `arrange(sex, count)`
b. `filter(sex == "F")`
c. `filter(sex == "M", count > 10)`
d. `summarise(total = sum(count))`
e. `select(name, count)`

Problem 7.7

a) `BabyNames` is pushed twice as an input to `group_by()` and `summarise()`, once by the chaining syntax, once as the first object listed after the parentheses.

```
BabyNames %>%
  group_by(year, sex) %>%
  summarise(total = sum(count))
```

b) Rather than `ZipGeography` being the first input to `group_by()`, it's being used to store the result. Alas, there won't be any result, because there is no data table input to the `group_by()` data table.

```
ZipGeography %>%
  group_by(State) %>%
  summarise(pop = sum(Population))
```

c) The -> after `group_by()` should be the chaining `%>%`

d) It's more or less upside down.

```
Minneapolis2013 %>%
  group_by(First) %>%
  summarise(votesReceived = n())
```

First	votesReceived
ABDUL M RAHAMAN "THE ROCK"	338
ALICIA K. BENNETT	351
BETSY HODGES	28935
BILL KAHN	97
BOB "AGAIN" CARNEY JR	56
... and so on for 38 rows	

Problem 7.9

a) `summarise()` b) join, whether it be `inner_join()`, `left_join()`, etc.

Problem 7.11

The variables given as arguments to `group_by()` will always appear in the output, along with whatever variables are created by `summarise()`.

a. `sex, count, meanAge`
b. `diagnosis, count, meanAge`
c. `sex, diagnosis, count, meanAge`
d. `age, diagnosis,count, meanAge`
e. `age, count, meanAge`

Problem 7.12

1) `nrow()` 2) `names()` 3) `help()` 4) `library()`
2) `group_by()` 6) `summarise()`

Chapter 8

Problem 8.3

All of the graphics have the same frame:

```
Frame <- CPS85 %>% ggplot(aes(x=age, y=wage))
```

Using that frame, saved in an object called `Frame` ...

- Graph (A)

  ```
  Frame + geom_point()
  ```

- Graph (B)

  ```
  Frame + geom_point(aes(shape=sex))
  ```

- Graph (C)

  ```
  Frame + geom_point(aes(shape=sex)) +
    facet_grid( married ~ .)
  ```

- Graph (D)

  ```
  Frame + geom_point(aes(shape=married)) +
    ylim(0,30)
  ```

Chapter 9

Problem 9.1

- `str()` Quick presentation
- `group_by()` Data verb
- `rank()` Transformation
- `mean()` Summary Function
- `filter()` Data Verb
- `summary()` Quick presentation
- `summarise()` Data verb
- `merge()` Data verb
- `glimpse()` Quick presentation

Problem 9.3

1. Which color diamonds are largest on average?

```
diamonds %>%
  group_by(color) %>%
  summarise(size=mean(carat, na.rm = TRUE)) %>%
  arrange(desc(size)) %>%
  head(1)
```

color	size
J	1.16

2. Clarity with the largest average "table."

```
diamonds %>%
  group_by(clarity) %>%
  summarise(
    ave_table = mean(table, na.rm = TRUE)) %>%
  arrange(desc(ave_table)) %>%
  head(1)
```

clarity	ave_table
I1	58.30

Chapter 10

Problem 10.1

Most of the data verbs operate on a single data table as input. The join verbs, however, combine *two* data tables. Since only one can be passed into the verb by the chaining syntax, the other must be included as an argument inside the parentheses.

Problem 10.2

Not all the cases in the demographics table are contained in the geographic table, and there are cases in the geographic table that are not in the demographic table. For instance, Åland is in the geographic table but not in demographics. Akrotiri is in the demographics but not in the geographic table. You wouldn't want to combine the location of Åland with the demographics of Akrotiri!

One of the purposes of the join family of data verbs is to handle such missing or extra cases in the tables being combined.

Problem 10.3

1) Table **B** is in a format that makes it easy to find the change between years for each country.

```
tableB %>%
  mutate(diff = Y2001-Y2000)
```

2). Table **C** would make it easy to `join()` the country/continent data. Then finding the sum for each continent in each year could be accomplished with `group_by()` and `summarise()`

```
tableC %>%
  left_join( ContinentData ) %>%
  group_by( Continent, Year ) %>%
  summarise( total = sum( Value ) )
```

Chapter 11

Problem 11.3

- Table **A** versus **C**: A is wide, C is narrow.
- Table **B** versus **C**: B is wide, C is narrow.
- Table **A** versus **B**: They are both the same relative to each other. In a narrow format of data, the cases are more detailed than in the wide format. For instance, table **C** has cases that are "a country in a year," which is more detailed than either "year" or "country."

Problem 11.4

when	sbp
subject	BHO
subject	GWB
subject	WJC
before	120
before	115
before	135
after	160
after	135
after	145

Chapter 12

Problem 12.1

```
BabyNames %>%
  group_by(sex, name) %>%
  summarise(count = sum(count)) %>%
  mutate(popularity = rank(desc(count))) %>%
  filter(popularity <= 5)
```

sex	name	count	popularity
F	Elizabeth	1591439	2
F	Jennifer	1461186	4
F	Linda	1450328	5
F	Mary	4112464	1
F	Patricia	1570135	3
	... and so on for 10 rows		

Problem 12.2

```
PopularCounts <-
  BabyNames %>%
  group_by(year, name) %>%
  summarise(total = sum(count)) %>%
  mutate(ranking =
           ifelse(rank(desc(total)) <= 100,
                  "Top_100", "Below" )) %>%
  group_by(year, ranking) %>%
  summarise(total = sum(total))
GlyphReady <-
  PopularCounts %>%
  spread(ranking, total) %>%
  mutate(frac_in_top_100 =
           Top_100 / (Top_100 + Below))
GlyphReady %>%
  ggplot(aes(x=year, y=frac_in_top_100)) +
  geom_line() +
  ylim(0, NA)
```

Problem 12.3

a. summary function
b. neither a summary nor a transformation. It is a pair of values rather than a single value as required for a summary function.
c. summary function
d. transformation function
e. transformation function
f. summary function
g. transformation function
h. summary function: an entire set of dates is being turned into a single number.

Chapter 13

Problem 13.1

1. There are 7 distinct vertices: China, France, Germany, Italy, UK, USA, USSR
2. There is one row for each edge, so 9 edges.

Problem 13.3

NA

Chapter 14

Problem 14.1

```
mosaicData::CPS85 %>%
  ggplot(aes(x = sex, y = wage)) +
  geom_boxplot(aes(fill=sex))
```

Problem 14.2
About 1.63 meters.

Problem 14.3
For women, a litle less than 1.60 meters. For men, about 1.75 meters.

Problem 14.5

- Graph (A)

```
CPS85 %>%
  ggplot(aes(x=wage)) +
  geom_density(aes(fill=sex,color=sex),
        alpha=0.5) +
  facet_grid( . ~ married) +
  ggtitle("(A)") + xlim(0,30)
```

- Graph (B)

```
CPS85 %>%
  ggplot(aes(x=age, y=wage)) +
  geom_smooth(aes(color=sex)) +
  ggtitle("(B)") +
  facet_grid( married ~ .) + ylim(0,NA)
```

- Graph (C)

```
CPS85 %>%
  ggplot(aes(x=age, y=wage)) +
  ggtitle("(C)") +
  geom_smooth(aes(color=sex)) +
  facet_wrap( ~ sector) + ylim(0,25)
```

Problem 14.7

```
mosaicData::CPS85 %>%
  ggplot(aes(x = educ, y = wage,
       color = sex)) +
  geom_point() +
  stat_smooth(method="lm") +
  ylim(0,15)
```

Problem 14.8

```
mosaicData::Galton %>%
  ggplot(aes(x = height, y = mother)) +
  geom_density2d() +
  facet_grid( ~ sex)
```

Problem 14.11

a. The fraction of people aged 65 and older is larger at the lowest income groups. But since the confidence intervals overlap among all the income groups, this relationship might be illusory: the medians might be roughly the same for all groups.
b. With more data, the notches are smaller. According to the $\sqrt{n}$ rule, with 16 times as much data, the notches should be about 1/4 the size of those in the smaller sample. Measuring the notches with a ruler confirms this in the graph.

Chapter 15

Problem 15.1

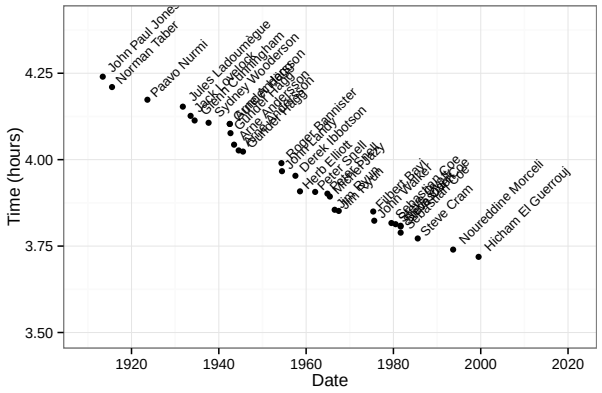
- `"April 30, 1777"` — `mdy()`
- `"06-23-1912"` — `mdy()`
- `"3 March 1847"` — `dmy()`
- `"11:00 am on Nov. 11th, 1918 at 11:00 am"` — `mdy_hm()`
- `"July 20, 1969"` — `mdy()`

Problem 15.3

```
library(rvest)
library(RCurl)
web_page <-
 "http://en.wikipedia.org/wiki/no break here
 Mile_run_world_record_progression"
SetOfTables <-
  web_page %>%
  rvest::html() %>%
  readHTMLTable(stringsAsFactors=FALSE)
T4 <- SetOfTables[[4]]

T4 %>%
  mutate(Date = gsub("\\[.\\]$", "", Date),
         Time = lubridate::as.duration(ms(Time))/60,
         Date = dmy(Date)) %>%
  ggplot(aes(x=Date, y=Time)) +
  geom_point() +
  geom_text(aes(label=Athlete),
     size=3, angle=45,hjust=-0.1) +
```

```
  ylim(3.5,4.4) + ylab("Time (hours)") +
  xlim(ymd("1910-01-01"), ymd("2021-01-01"))
```




```
BabyNames %>%
  #    Add up across all years for each name
  group_by(name) %>%
  summarise(total = sum(count)) %>%
  #    Filter to find names ending in [jo|joe]
  #    The $ is the end-of-string marker
  filter(grepl(".*(jo$|joe$)", name)) %>%
  #    Grab the top 10.
  arrange(desc(total)) #%>%
```

name	total
Maryjo	6973
Billiejo	1455
Marijo	1275
Bobbijo	1230
Bobbiejo	1009
... and so on for 72 rows	

```
  head(10)
```

```
[1] 10
```

Chapter 16

Problem 16.1

1) Boys' names ending in a vowel.

```
BabyNames %>%
  #    Filter to keep only the boys.
  filter(sex == "M") %>%
  #    Add up across all years for each name
  group_by(name) %>%
  summarise(total = sum(count)) %>%
  #    Filter to find names ending in a vowel
  #    The $ is the end-of-string marker
  filter(grepl("[aeiou]$", name)) %>%
  #    Grab the top 10.
  arrange(desc(total)) %>%
  head(10)
```

name	total
George	1451408
Joshua	1163815
Jose	539924
Kyle	469045
Lawrence	454323
... and so on for 10 rows	

2) Names ending in "jo" or "joe". This is much the same as (1), without needing to filter by sex. The regex uses the "or" (vertical bracket).

Problem 16.3

1. Each matched string will be two letters, e.g. `AL`, `AK`, `AS`.
2. For a pattern like `A[LKSZRAP]`, there are seven strings that match. For an pattern like `D[EC]`, there are two strings that match: `DE` and `DC`. For a pattern like `RI`, there is only one string that matches. Altogether, 61 patterns.
3. These are appreviations for the US states and territories.

Problem 16.5

```
my_regex <-
  "\\(\\s*([+-]*[0-9]*[0-9\\.][0-9]*)no break here
  \\s*,\\s*([+-]*[0-9]*[0-9.][0-9]*)\\s*\\)"
Result <-
  CrimeSample %>%
  extract(Location,
          into=c("lat", "long"),
          regex = my_regex,
          convert = TRUE)
```

lat	long
42.34	-71.11
42.26	-71.16
42.33	-71.08
... and so on for 50 rows	

Chapter 17

Problem 17.2

1. Price is the variable to be explained, called the response variable. The explanatory variables included in the model are `living_area`, `bathrooms`, `bedrooms` and `fireplaces`.
2. No. Those are the houses to the left in node 3. For those houses, the `bathrooms` variable does not enter into the model.
3. Of those houses (in node 5), those with 1 1/2 or fewer bathrooms have a typical price of $151,000 while those with 2 or more bathrooms have a typical price of $180,000.
4. The `fireplaces` variable enters into the model only for houses larger than 2816 square feet. For those houses, having more than 1 fireplace is associated with a much larger price: $502,000 compared to $385,000.

Keep in mind that the model shows associations between variables. This is not at all the same thing as causal relationships. For example, in question (4), it might be that very fancy houses have much higher prices and they tend to have more than one fireplace. Perhaps it's the overall fanciness rather than just the existence of a second fireplace that influences the price.

C
Readings and Notes

Chapter 1: Tidy Data

The tabular presentation of data appears in cuneiform tablets written as long ago as 2500 BCE. (Robson et al. 2003) The first formal expression of principles of tidy data is startlingly recent. It was developed by computer scientist Edgar F. Cobb (Cobb 1970) who coined the term "relational database." The term "tidy data" comes from Hadley Wickham (Wickham 2013b). His `reshape2` R package implementing data tidying operations is among the ten most frequently downloaded packages. The `gather()` and `spread()` functions, used in this book, come from the `tidyr` package, a recent updating and streamlining of `reshape2`.

The Minneapolis mayoral election data in Table 1.3 is from `http://vote.minneapolis.mn.gov/www/groups/public/@clerk/documents/webcontent/wcms1p-126724.xlsx`

Chapter 2: Computing with R

Videos provide a valuable resource to guide a user through installation of R and RStudio as well as the various capabilities of RStudio. Of course, some videos are better than others. A nice video for beginners about installation is (Marin 2013).

Interviews with RStudio founder JJ Allaire (Arino 2014a) and Joe Cheng (Arino 2014b) illustrate that there are actual people behind the ideas.

Chapter 3: R Command Patterns

There are many introductions to R. Often, these are given in a specific context such as elementary statistics (Verzani 2014), ecology (Stevens 2009), introductory calculus (D. T. Kaplan 2013a), or any of a number of other areas: quantitative economics, numerical analysis, finance, forestry, and a host of others. For R programming, see (Matloff 2011), (Wickham 2015a), and (Chambers 2008).

Of course, the context in which R is presented in this book is data wrangling and visualization, so you may find introductions in another context to be very distant from *Data Computing* in terms of R style, the functions used, and the kinds of R objects that are dealt with. This book takes a "minimal R" approach [Pruim, Horton, and Kaplan (2015). Rather than using general programming constructs, *Data Computing* uses the "mini-languages" provided by the `dplyr` and `ggplot2` packages. In this text you won't see `$`, `[]` and `[[]]`, loops such as `for` and `while`, conditionals such as `if`, and object classes such as lists. Another example of a "minimal R" approach is given by the `mosaic` package for introductory statistics.(Pruim, Kaplan, and Horton 2011,Pruim, Horton, and Kaplan (2015), D. T. Kaplan (2013b)) In `mosaic`, the `do()` function lets the user carry out the repetitive calculations involved with bootstrapping, etc., without requiring loops or indexing.(Kaplan, Pruim, and Horton 2012)

Chapter 4: Files and Their Types

The R/Markdown `.Rmd` format is part of a wide movement in statistics toward "reproducible research." The idea is that scientific publications and other reports should include with them "the full computational environment used to produce the results in the [report] such as the code, data, etc. that can be used to reproduce the results and create new work based on the research."(Wikipedia 2015)

Ability to use reproducible research techniques is considered to be a foundational skill for data scientists. (Peng, Leek, and Caffo 2015) But it is not a narrow technical skill. Anyone using data to draw responsible conclusions benefits from being able to reproduce and update her analysis. John Chambers has described this as an ethical responsibility for statistical programmers. His *Prime Directive* states, "The computations and the software for data analysis should be trustworthy: they should do what they claim and be seen to do so." (Chambers 2008)

The implementation of reproducible research techniques using `knitr` and Markdown, and the support for these in RStudio, has made doing the right thing relatively easy. This ease of use — its being the right thing! — are supporting an uptake of the techniques by statistical educators and their students.(Baumer et al., n.d.)

There are book-length treatments that cover the several systems widely used in reproducible research with R. The author of `knitr`, Yihui Xie describes its workings in (Xie 2013). Techniques for producing variety of publication formats: HTML, PDF, presentation slides, etc. are described in (Gandrud 2013). Notably, there is even a system, called `shiny`, developed by RStudio's Joe Cheng for constructing interactive web "apps" that can be integrated with R/Markdown documents.(RStudio 2015)

Chapter 5: Introduction to Data Graphics

Many of the types of graphics presented in this chapter are topics of mainstream introductory statistics texts. Two particularly distinctive books on data graphics are Tufte's beautiful and famous *Visual Display of Quantitative Information* (Tufte 2001) and and Cleveland's less-well known classic. (Cleveland 1993) These books are not about software. Instead, they are about principles for conveying information graphically.

Hadley Wickham, the author of the `ggplot2` package used in this book, provides a very concise overview of the capabilities and limitations of human visual perception and the consequences of this form making informative statistical graphics.(Wickham 2013a) This is based on much earlier work spanning more than a half century; see the references in Wickham (2013a) for a guide.

The FlowingData blog, `flowingdata.com`, is a particularly good site for browsing innovative visualizations and getting tips on advanced techniques.

Chapter 6: Frames, Glyphs and other Components of Graphics

This chapter is a lead-in to Chapter 8; see the references there. The word "glyph" is used in this book and appears in Wilkinson's *Grammar of Graphics* (Wilkinson 2005); other authors use other terms: shape, mark, geom, and so on. In the view of this author, "glyph" has considerable merit: it's short and easily pronounced, it's uncommon and so can be easily associated with graphics. ("Mark", for instance, has at least five distinct meanings as a noun and six as a verb.) It lends itself to a useful adjective: glyph-ready.

Chapter 7: Wrangling and Data Verbs

The actual wrangling task for the Minneapolis mayoral ballots was more complicated than a simple count. The election was held under a "rank choice" system. There was a succession of counts. In each count, the lowest-ranked candidate was dropped. For the voters who made a first choice of that candidate, the second choice candidate was used in subsequent counting. This cycle of counting went on until there was one candidate with an absolute majority of votes.

Figure 7.2 shows a pattern that may be startling at first glance. Why the low number of births before

1920? This is not because of a low birthrate. Rather, it is an artifact of the manner in which the data were collected three-quarters of a century ago. Social Security started registering people in 1935 when they became eligible for participation. This included only a small fraction of the people born before 1920.

"Data wrangling" is a term of growing use. Mainstream terminology includes "data processing" or "data manipulation". These are legitimate but carry unfortunate connotations. "Data processing" suggests a routine and pre-defined task, not the creative interplay between insight and data. "Data manipulation" may suggest an unfair or unscrupulous process that imposes a conclusion on the data rather than honestly bringing out the information latent in the data.

Chapter 8: Graphics and their Grammar

The idea that graphics have their own grammar was presented by Wilkinson. (Wilkinson 2005) Hadley Wickham used Wilkinson's grammar with some modifications (Wickham 2010a), as the organizing theme for the graphics commands in the `ggplot2` R package . Wickham's initial book (Wickham 2010b) is now dated in terms of the commands themselves, but is a good way to see how the grammar applies to the variety of graphical modes. `ggplot2` is one of the top few most downloaded "contributed" R packages.

There are two other user-level graphics systems in R. "Base graphics" (with commands like `plot()`, `hist()`, `points()`, `lines()`, `barplot`) was the first available. Subsequently "lattice graphics" (some commands: `xyplot()`, `histogram()`, `barchart()`) offered a more sophisticated appearance and high-level support for faceting, color to indicate groups, etc. Lattice graphics uses a *formula notation* which was pioneered by R for the purpose of constructing statistical models, as in Chapter 17. The `mosaic` package (Pruim, Kaplan, and Horton 2011) extends the formula notation to a wide variety of data-analysis tasks and even to calculus (D. T. Kaplan 2013a). The `ggplot2` package is another graphics system particularly admired for the elegance of the graphics it produces. In this book, `ggplot2` is used because it parallels the sort of constructions used in data wrangling with the `dplyr` package.(Wickham and Francois 2014)

Due to the popularity of `ggplot2` there are many online and printed manuals. A web search such as "ggplot2 change legend position" will often tell you what you need to know to modify a graphics template. A particularly useful printed (and online) resource is the "Graphics Cookbook" by Winston Chang (Chang 2012).

As the domain of statistical graphics expands to include interaction and dynamic views, the terminology needed to describe graphics will change. Already, the `shiny` package for web apps includes terms such as "reactive." The `ggvis` package, seen by some as the successor to `ggplot2`, has a somewhat different terminology than `ggplot2`. Additional kinds of graphics, e.g. interactive street maps on which other data can be overlaid, introduce still more ideas. See for example the `leaflets` package.

Chapter 9: More Data Verbs

The `dplyr` package includes a helpful introductory vignette.(Wickham 2015b) ("Vignette" is the standard term in R for narrative- or tutorial-form documentation.) A short video introduction to `dplyr`, given by Hadley Wickham, is aimed at those using other data wrangling systems. (Wickham 2014c). A short online course by RStudio's Garrett Grolemund is online at DataCamp, a platform that integrates the R console into the tutorial narrative.(Grolemund 2015) Datacamp (`https://www.datacamp.com/`) offers several other courses using R for data science.

The `data.table` R package by Matt Dowle *et al.* provides powerful and fast data wrangling capabilities.(Dowle 2014) The tight and concise symbol-based notation in `data.table` is appreciated by some users. Others prefer the more sequential and name-based form of `dplyr`.

The use of SQL for data wrangling and managing database systems is practically universal. There is a host of tutorials, textbooks, manuals, cookbooks,

quick starts, blogs, etc. on SQL, far too many to represent with a handful of citations. By browsing the web, you'll be able to find an introduction that suits your own style.

Here is a very simple data wrangling operation in each of the three systems: SQL, `data.table`, and `dplyr`.

- `dplyr` package

```
BabyNames %>%
  group_by(year, sex) %>%
  summarise(nNames = n())
```

- SQL

```
SELECT year, sex, SUM(count) as nNames
  FROM BabyNames  GROUP_BY year sex;
```

- `data.table` package

```
BabyNames[ , length(count),
          by = c("sex", "year") ]
```

Chapter 10: Join

As the only kind of data verb to bring together two data tables, joins are essential, even if they are unfamiliar to newcomers. There are many online tutorials about the various kinds of joins. These are easily found by a web search. The `dplyr` package contains a tutorial vignette entitled "Two table verbs." (Wickham 2014b) The vast majority of join tutorials are written in SQL, which is by far the dominant language used for processing data tables.

Chapter 11: Wide versus Narrow

The `spread()` and `gather()` data verbs used in this book are from the `tidyr` package. The predecessor to `tidy` was `reshape2`.(Wickham 2014a) When using `dplyr`, it's best to use `tidyr` rather than `reshape2`. But since `tidyr` is so new, most of the introductions and tutorials on wide vs narrow data are based in the notation of `reshape2`. See, for example (Anderson 2013) (Wickham 2013b)

Chapter 12: Ranks

Something about the different sorts of ranks.

Non-parametric statistics and rank.

Chapter 13: Networks

This book uses `ggplot2` graphics to draw networks. Much more sophisticated systems for plotting and analyzing networks are available, such as the `igraph` package. (Igraph 2013) This packge is an implementation in R of the more general igraph system. (Csardi and Nepusz 2006)

Chapter 14: Statistics

This chapter covers three principle statistical topics: distributions, confidence intervals (and bands), and the smoother as a function approximation to data. A typical introductory statistics textbook will not mention smoothers, and will feature statistical tests such as the t-test, the chi-squared test, etc. These tests, and the formulas behind them, were developed in the decades around 1900. A more modern approach to statistics is based in models and randomization.

There are many book-length presentations of statistical modeling in R. The earliest is Chamber's and Hastie's presentation using S.(Chambers and Hastie 1992) (R is an open-source remake and extension of S.) A well-respected contemporary text is Gelman and Hill (2006). An introduction to statistical models is provided by D. T. Kaplan (2013b).

The `ggplot2` package makes a useful distinction between "geoms" and "stats". Many of the functions used for drawing graphics start with `stat_`, and many of the geoms perform (e.g. `geom_density()`, `geom_bar()`) do statistical calculations as part of their operation.

Chapter 15: Data Scraping and Cleaning

More guidance on reading data into R from a variety of external formats is provided in a DataCamp tutorial. (DataCamp 2015) For reading directly from Google Docs, see the `googlesheets` package. (Bryan and Zhao 2015)

Chapter 16: Using Regular Expressions

There are many tutorials about regular expressions on the Internet, including interactive ones such as http://regexone.com/.

Chapter 17: Machine Learning

An excellent introduction to machine learning from a statistical perspective is James et al. (2013). There are several R-based tutorials, such as Lantz (2013) or Yu-Wei (2015).

Although machine learning is often presented as if it were a discontinuous break with the past, in fact one easily see the influence of earlier statistical modeling techniques. The author's previous book, *Introduction to Statistical Modeling*, (D. T. Kaplan 2013b) highlights such statistical techniques. Disciplinary boundaries, such as those between statistics and computer science, are at best vague regions. At worst, they mislead students into ignoring topics that may be critical to their broader understanding.

The Scottish Parliament example was initially constructed by then-student Caroline Ettinger and her faculty advisor, Andrew Beveridge, at Macalester College, and presented in Ms Ettinger's senior capstone thesis.

Source of Figure **??**: Richinstead at Wikimedia. CC-BY-SA-3.0.)

Projects

NCI60 and NCI60cells are lightly re-organized from the form provided by Staunton *et al.* at http://www.broadinstitute.org/mpr/NCI60/NCI60.html.

An overview in *Nature Cancer Reviews* [Shoemaker 2006] describes some of the ongoing work with these cell lines.

Expression data described in [Staunton 2001]

NCI60 and NCI60cells are lightly re-organized from the form provided by Staunton *et al.* at http://www.broadinstitute.org/mpr/NCI60/NCI60.html.

Figure A.22 source: https://nyoobserver.files.wordpress.com/2011/06/restaurant_0.jpg

Figure A.21 source: http://barfblog.com/wp-content/uploads/2014/12/OC.color_.grades.jpeg

The restaurant violation data tables are published by the New York City government https://data.cityofnewyork.us/Health/DOHMH-New-York-City-Restaurant-\texttt{Inspection-Results/xx67-kt59¿

References

Anderson, Sean. 2013. "An Introduction to Reshape2." http://seananderson.ca/2013/10/19/reshape.html.

Arino, Eduardo. 2014a. "JJ Allaire - Interview by DataScience.LA at UseR 2014." https://www.youtube.com/watch?v=HKSlWA2dY9I.

———. 2014b. "Joe Cheng - Interview by DataScience.LA at UseR 2014." https://www.youtube.com/watch?v=uJm-its3ZWM.

Baumer, B, M Cetinkaya-Rundel, A Bray, L Loi, and NJ Horton. n.d.

Bryan, Jenny, and Joanna Zhao. 2015. "Googlesheets Basic Usage." http://htmlpreview.github.io/?https://raw.githubusercontent.com/jennybc/googlesheets/master/vignettes/basic-usage.html.

Chambers, JM, and T Hastie. 1992. *Statistical Models in S*. Wadsworth & Brooks/Cole.

Chambers, John. 2008. *Software for Data Analysis: Programming with R*.

Chang, Winston. 2012. *R Graphics Cookbook*. O'Reilly.

Cleveland, William S. 1993. *Visualizing Data*.

Cobb, Edgar F. 1970. "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM* 5 (13): 377–87.

Csardi, Gabor, and Tamas Nepusz. 2006. "The Igraph Software Package for Complex Network Research." *InterJournal* Complex Systems: 1695. http://igraph.org.

DataCamp. 2015. "This R Import Tutorial Is Everything You Need." http://blog.datacamp.com/r-data-import-tutorial/.

Dowle, Matt. 2014. "Data.table Package for R."

https://cran.r-project.org/web/packages/data.table/index.html.

Gandrud, Christopher. 2013. *Reproducible Research with R and RStudio*. Chapman & Hall/CRC the R Series. Chapman & Hall/CRC Press. http://www.crcpress.com/product/isbn/9781466572843.

Gelman, Andrew, and Jennifer Hill. 2006. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge University Press.

Grolemund, Garrett. 2015. "Data Manipulation in R with Dplyr." https://www.datacamp.com/courses/dplyr-data-manipulation-r-tutorial.

Igraph. 2013. "Igraph Package for R." https://cran.r-project.org/web/packages/igraph/index.html.

James, G., D. Witten, T. Hastie, and R. Tibshirani. 2013. *An Introduction to Statistical Learning: With Applications in R*. Springer Texts in Statistics. Springer. /bib/james/james2013introduction/ISLR+First+Printing.pdf,http://www-bcf.usc.edu/~gareth/ISL/,http://books.google.com.tr/books?id=qcI/_AAAAQBAJ.

Kaplan, Daniel T. 2013a. *Start R in Calculus*. Project MOSAIC Books.

———. 2013b. *Statistical Modeling: A Fresh Approach*. 2nd edition. Project MOSAIC.

Kaplan, Daniel, Randall Pruim, and Nicholas Horton. 2012. "Randomization-Based Inference Using the Mosaic Package." https://cran.r-project.org/web/packages/mosaic/vignettes/Resampling.pdf.

Lantz, B. 2013. *Machine Learning with R*. Community Experience Distilled. Packt Publishing. http://books.google.com.tr/books?id=ZQu8AQAAQBAJ,/bib/lantz/lantz2013machine/2148OS_Machine+Learning+with+R_eBook.pdf.

Marin, Mike. 2013. "Installing R and RStudio." https://www.youtube.com/watch?v=cX532N_XLIs.

Matloff, Norman. 2011. *The Art of R Programming: A Tour of Statistical Software Design*. No Start Press.

Peng, Roger D, Jeff Leek, and Brian Caffo. 2015. "Coursera Course: Reproducible Research." Accessed August 9. https://www.coursera.org/course/repdata.

Pruim, Randall, Daniel Kaplan, and Nicholas Horton. 2011. "Mosaic Package for R." https://cran.r-project.org/web/packages/mosaic.

Pruim, RJ, NJ Horton, and DT Kaplan. 2015. *Start Teaching with R*. Project MOSAIC Books.

Robson, E, M Campbell-Kelly, M Croarken, and RG Flood (eds.). 2003. *The History of Mathematical Tables from Sumer to Spreadsheets*. Oxford University Press.

RStudio. 2015. "Shiny." Accessed August 9. http://shiny.rstudio.com/.

Stevens, M. Henry H. 2009. *A Primer of Ecology with R*. Use R. Springer.

Tufte, Edward R. 2001. *The Visual Display of Quantitative Information*. 2nd edition.

Verzani, John. 2014. *Using R for Introductory Statistics*. 2nd edition. Chapman & Hall/CRC.

Wickham, Hadley. 2010a. "A Layered Grammar of Graphics." *Journal of Computational and Graphical Statistics* 19 (1): 3–28. doi:10.1198/jcgs.2009.07098.

———. 2010b. "Ggplot2: Elegant Graphics for Data Analysis." Springer.

———. 2013a. "Statistical Graphics." *Encyclopedia of Environmetrics*. http://vita.had.co.nz/papers/stat-graphics.html.

———. 2013b. "Tidy Data." *Journal of Statistical Software* 59 (10).

———. 2014a. "Reshape2 Package for R." https://cran.r-project.org/web/packages/reshape2/index.html.

———. 2014b. "Two Table Verbs." https://cran.r-project.org/web/packages/dplyr/vignettes/two-table.html.

———. 2014c. "Why Dplyr?" http://www.r-bloggers.com/hadley-wickham-presents-dplyr-at-user-2014/.

———. 2015a. *Advanced R*. CRC Press.

———. 2015b. "Introduction to Dplyr." https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html.

Wickham, Hadley, and Romain Francois. 2014. "Dplyr Package for R." https://cran.r-project.org/web/packages/dplyr.

Wikipedia. 2015. "Reproducible Research." Ac-

cessed August 9. https://en.wikipedia.org/wiki/Reproducibility#Reproducible_research.

Wilkinson, Leland. 2005. *The Grammar of Graphics*. 2nd edition. Springer.

Xie, Yihui. 2013. *Dynamic Documents with R and Knitr*. Chapman & Hall/CRC. https://github.com/yihui/knitr-book/.

Yu-Wei, Chiu. 2015. *Machine Learning with R Cookbook*. Packt Publishing.

D
Index