2

Reading and Exploring Data in Tables

CONTENTS

2.1 Introduction

Data often are provided in a simple table-like format in a plain text file where rows correspond to observations and columns to variables. In this chapter, we examine several file formats for tables; address considerations in choosing the best structure for working with the data in *R*, which includes choosing data types, reshaping the table, and merging tables;

describe techniques for cleaning and validating data; and discuss approaches for exploring data. We use several data sets as examples throughout the chapter. These are described below. Each description includes a summary of what constitutes rows and columns in the data table.

Not all data sets have a simple table-like structure, and in Chapter 4 we tackle the challenge of working with data in more complex formats. Furthermore, in Chapter 14 we consider situations that require extensive programming to acquire and clean the data, in Chapter 9 we work with data in relational databases, and in Chapter 12 we examine data available through Web scraping, forms and APIs.

Example 2-1 Traffic on California Freeways

The freeway Performance Measurement System (PeMS) has a Web site [3] that allows registered users to get information about traffic on California's freeways. The Web site provides access to data from loop detectors, which are recording devices embedded in the road at various locations in California's freeway system. We have an extract from PeMS that contains recordings, for each of 3 lanes, of the flow (number of cars) and occupancy (the percentage of time cars were over the loop) in successive 5 minute summaries over several days at one location.

row Five minute interval in a day.

columns Time of day, and flow and occupancy at one location for each of 3 lanes (in the same direction).

■

Example 2-2 Drug Abuse Warning Network Survey

The US federal government conducts several large complex surveys to inform Congress and government agencies on important issues facing the US public and businesses. One such survey is the annual Drug Abuse Warning Network Survey (DAWN), which studies substance-related emergency room visits [8]. Each record in DAWN corresponds to one emergency-room visit for substance-related reasons. For each visit, information is recorded about the patient, including the type of visit (e.g., suicide attempt, adverse reaction, accidental ingestion) and the substances found in the patient.

row Emergency-room visit for a substance-related reason.

columns Patient information (e.g., age, sex, race) and substances present in patient.

■

Example 2-3 World Bank Reports

The World Bank provides financial and technical assistance to developing countries. In 2010, the World Bank launched an Open Data Website [9] that provides access to data from their reports on topics such as GDP, education, health, and the environment.

row A country reported in the World Bank.

columns Information for a particular year about the country, such as GDP, birth rate, educational attainment for females in various age groups, and employment to population ratios.

■

Example 2-4 Hillary Clinton's Emails

The US State Department has released thousands of Hillary Clinton's emails from the time

when she was Secretary of State. These emails are controversial because they include work-related emails that were sent or received by Clinton on her private email account. The emails were released as PDFs and have been made available by the Wall Street Journal at [5]. Ben Hammer has processed these PDFs into plain text files and they have been made available on Kaggle [4] for public analysis.

row An email sent to or received by Hillary Clinton's private email address.

columns Information about the email, including who sent the email, who received the email, the date the email was sent, and the redacted text of the email (extracted from a PDF file).

■

Example 2-5 Handwritten Digits

MNIST (Modified National Institute of Standards and Technology) is a collection of images of handwritten digits for developing algorithms to classify digits [**?**]. The image for each digit is a 28×28 grid of pixels. The value for a pixel is an integer between 0 and 255, inclusive, which indicates the lightness or darkness of the pixels with 0 being white or empty and 255 being black. These images have been 'hand'-classified so that we know the true number that was written.

row An image of a handwritten digit.

columns The values for the 784 pixels that compose the image.

■

2.2 Formats for Tabular Data

Our first step in accessing tabular data for exploration and analysis is to determine how the information is organized in the source file, such as whether or not the values are separated by commas. In this chapter, we consider 3 standard formats for the data: delimited, fixed width, and key-value. We describe each of these formats and provide examples from the data introduced in Section 2.1. We also examine binary data in Section 2.4.

Plain Text or Binary?

Statisticians often work with plain text source files. We can typically determine whether or not the file is plain text from its filename extension. Extensions such as *csv*, *txt*, *fwf*, and *dcf* correspond to plain text files. These are shorthand for comma-separated values, text, fixed-width format, and debian control format, respectively. However, filename extensions are only a convention, and they do not guarantee that the contents of the file is plain text nor need they determine the format of the data. We can confirm a file is plain text by viewing the contents in a plain text editor, such as Emacs, NotePad++, Sublime, TextWrangler, and Vim. Additionally, the file may be accompanied with documentation that indicates whether or not the contents are plain text.

Filename extensions, such as *xls*, *rda*, and *sas*, correspond to Excel, *R*, and SAS binary files. If we open one of these files with a plain text editor, the contents appear as gibberish. With these files, we typically use the associated software to read and analyze the data. Data in these formats can sometimes be exchanged between these software programs. For example, the *R* packages `haven` [14] and `R.matlab` [2] provide functions to read SPSS and

MATLAB data files into *R*, respectively. Data can also be exchanged via binary formats that do not directly map to plain text and that are not specially formatted for particular software tools. In order to read these files, we need precise knowledge of the file format. An example is provided in Section 2.4.1.

2.2.1 Delimited Data

In a typical arrangement of tabular data, one line in a source file corresponds to one row in a table, which becomes one record in an *R* data frame. For each line in the file, we need to be able to distinguish the values for the various variables. Delimited data use a delimiter, such as a comma, to separate these values. In addition to the comma, other typical delimiters include the tab, semicolon, and white space. With these input files, each row is simply divided into pieces at the locations of the delimiters. The file displayed below is an example of comma-delimited data. These data are for the simple `family` data frame introduced in Q.1-7 (page 21). Each row corresponds to a family member, and the values for the person's first name, sex, age, height, etc. are separated by commas, i.e.,

```
firstName,sex,age,height,weight,bmi,overWt
Tom,m,77,70,175,25.16239,TRUE
Maya,f,33,64,124,21.50106,FALSE
Joe,m,79,73,185,24.45884,FALSE
Robert,m,47,67,156,24.48414,FALSE
Sue,f,27,61,98,18.51492,FALSE
Liz,f,33,68,190,28.94981,TRUE
...
```

Notice how the lines in the file have different lengths, depending on the number of characters in the person's name, the number of digits in the weight, and whether the value for the over weight indicator is `TRUE` or `FALSE`. We can use the commas in a line to split the text into values for the variables. That is, the value before the 1st comma is assigned to the 1st variable (firstName), the value between the 1st and 2nd commas is the value for the 2nd variable (sex), and so on. We have color-coded the values for sex and weight to show that they do not line up from one record to the next, but the value for sex appears after the 1st comma and weight appears after the 4th comma in each row. If, for example, height is missing for an individual, then this can be conveyed by the absence of a value between the 3rd and 4th commas or by the presence of a special value such as `-9` for height. Of course, the values in a comma-delimited file cannot contain a comma, unless they appear within a quoted value.

2.2.1.1 Comma-Delimited Traffic Data

The file *flow-occ.txt* contains recordings from the loop detector system described in Q.2-1 (page 52). That is, each row in the file corresponds to a recording taken at 3 loop detectors. These measurements are the flow (number of cars) and the occupancy (the percentage of time cars were over the loop) in a 5-minute interval for each lane. The data contain readings for succesive 5-minute intervals over several days. Below is a snippet of the source file:

```
'Timestamp','Lane 1 Occ','Lane 1 Flow','Lane 2 Occ',\
 'Lane 2 Flow','Lane 3 Occ','Lane 3 Flow'
3/14/2003 00:00:00,.01,14,.0186,27,.0137,17
3/14/2003 00:05:00,.0133,18,.025,39,.0187,25
3/14/2003 00:10:00,.0088,12,.018,30,.0095,11
3/14/2003 00:15:00,.0115,16,.0203,33,.0217,19
```

```
3/14/2003 00:20:00,.0069,8,.0178,25,.0123,13
3/14/2003 00:25:00,.0077,11,.0151,24,.0092,13
```

The filename extension of *txt* indicates this is a plain text file. When we examine the contents of the file, we can see that these data are comma-delimited. Notice that the first row of the file is a 'header' row that contains variable names. The labels `'Lane 1 Occ'` and `'Lane 1 Flow'` refer to the occupancy and flow measurements for the leftmost lane on the freeway. Similarly, lane 2 is the center lane, and lane 3 is the farthest right lane. The header line is displayed here across 2 rows in order for it to fit within the page margins (The \ indicates that the line in the file is continued on the following line in the display.)

We have the information required to determine how to read the data into a data frame in *R*. Specifically, the values are comma separated, and the first line is a header. We use read_delim() in the `readr` package [13] to read the file with

```
library(readr)
traffic = read_delim("flow-occ.txt", delim = ",")
```

Notice that we provided 2 arguments to read_delim() – the file name and the delimiter, both as strings. The return value from read_delim() is a data frame (see Section 1.8). Let's examine the first few rows of this data frame to confirm that the data are read into *R* as expected

```
head(traffic)
```

```
Source: local data frame [6 x 7]

         'Timestamp' 'Lane 1 Occ' 'Lane 1 Flow' 'Lane 2 Occ'
               (chr)        (dbl)         (int)        (dbl)
1 3/14/2003 00:00:00       0.0100            14       0.0186
2 3/14/2003 00:05:00       0.0133            18       0.0250
3 3/14/2003 00:10:00       0.0088            12       0.0180
4 3/14/2003 00:15:00       0.0115            16       0.0203
5 3/14/2003 00:20:00       0.0069             8       0.0178
6 3/14/2003 00:25:00       0.0077            11       0.0151
Variables not shown: 'Lane 2 Flow' (int), 'Lane 3 Occ' (dbl),
 'Lane 3 Flow' (int)
```

The classes of the variables are automatically determined by read_delim(). We see that the occupancy variables are `numeric` (also known as double-precision, or dbl for short), and since flow is measured in counts of cars, the flow variables are integer vectors. The `Timestamp` variable is treated as a character vector. In Section 2.5.1, we consider whether or not these are appropriate data types for analysis and issues about the organization of the data, e.g., whether or not to keep 3 separate vectors for occupancy.

There are many functions available in *R* for reading data in *csv* files; these include read.csv(), read.delim() and read.table() that belong to the utilities provided in *R* (i.e., there is no need to load a separate package), and in the `readr` package we have read_csv() and read_delim(). We do not go into the details of the parameters for each of these functions, but note only that they provide very similar functionality. In this chapter, we primarily use the functions in `readr` because they have a consistent set of parameters.

2.2.2 Fixed Width Format

Another common format for tabular data is fixed width, where the value for a variable appears in the same position in each row in the source file. For example, below is a fixed-width version of the simple family from Chapter 1:

```
Tom     m7770175 25.16239TRUE
Maya    f3364124 21.50106FALSE
Joe     m7973185 24.45884FALSE
Robert  m4767156 24.48414FALSE
Sue     f2761  98 18.51492FALSE
Liz     f3368190 28.94981TRUE
...
```

Here, the 1st 8 characters in each line contain the person's name. When, the name is, say, 3 letters long, then the trailing 5 characters are blank. Similarly, the values for whether or not the person is over weight (`TRUE` or `FALSE`) appear in the 25th to 29th characters in each row. Notice how the values for, say, Tom's, age, height, weight, and bmi, appear as a string of digits with no separators, e.g., `777017525.16239`. In order to pull apart this string of digits into 77, 70, 175, and 25.16239 for age, height, weight, and bmi, respectively, we need to know that age appears in positions 10-11, height in 12-13, weight in 14-16, and BMI in 17-24. We have color-coded the values for sex and weight in each record, and we can see clearly that for each record these values are located in column 9 and columns 14-16, respectively. Note that even when a weight is only 2 digits, e.g., 98, the weight value has a leadning blank so that it takes up 3 positions.

2.2.2.1 Fixed Width Formatted Drug Abuse Warning Network Survey

The DAWN survey data are in the file *34565-0001-Data.txt*. We assume from the file's extension that it is plain text, but we need to examine the file contents or read the accompanying codebook to determine how the values are organized in the file. Below is a single record from this file, which corresponds to one emergency room visit related to substance abuse (see Q.2-2 (page 52)).

```
     1 2251082    .9426354082   3 4 1 2201141 2 865 105 1102\
005 1 2 1 2.00-7.00-7.0000-7.0000-7.00001255 105 1142032 4 1\
 1 2.50 5.00 5.0100-7.0000-7.0000  -7  -7  -7  -7-7-7-7-7.00\
-7.00-7.0000-7.0000-7.0000  -7  -7  -7  -7-7-7-7-7.00-7.00-7\
.0000-7.0000-7.0000  -7  -7  -7  -7-7-7-7-7.00-7.00-7.0000-7\
.0000-7.0000  -7  -7  -7  -7-7-7-7-7.00-7.00-7.0000-7.0000-7\
.0000  -7  -7  -7  -7-7-7-7-7.00-7.00-7.0000-7.0000-7.0000  \
-7  -7  -7  -7-7-7-7-7.00-7.00-7.0000-7.0000-7.0000  -7  -7 \
 -7  -7-7-7-7-7.00-7.00-7.0000-7.0000-7.0000  -7  -7  -7  -7\
-7-7-7-7.00-7.00-7.0000-7.0000-7.0000  -7  -7  -7  -7-7-7-7-\
7.00-7.00-7.0000-7.0000-7.0000  -7  -7  -7  -7-7-7-7-7.00-7.\
00-7.0000-7.0000-7.0000  -7  -7  -7  -7-7-7-7-7.00-7.00-7.00\
00-7.0000-7.0000  -7  -7  -7  -7-7-7-7-7.00-7.00-7.0000-7.00\
00-7.0000  -7  -7  -7  -7-7-7-7-7.00-7.00-7.0000-7.0000-7.00\
00  -7  -7  -7  -7-7-7-7-7.00-7.00-7.0000-7.0000-7.0000  -7 \
 -7  -7  -7-7-7-7-7.00-7.00-7.0000-7.0000-7.0000  -7  -7  -7\
  -7-7-7-7-7.00-7.00-7.0000-7.0000-7.0000  -7  -7  -7  -7-7-\
7-7-7.00-7.00-7.0000-7.0000-7.0000  -7  -7  -7  -7-7-7-7-7.0\
0-7.00-7.0000-7.0000-7.0000  -7  -7  -7  -7-7-7-7-7.00-7.00-\
```

```
7.0000-7.0000-7.0000  -7  -7  -7  -7-7-7-7-7.00-7.00-7.0000-\
7.0000-7.00008 611001
```

We can see that this record has a fixed-width format. Since there are no delimiters between values, we must specify the positions for the variable values when reading the data into *R*. These records are very long, with this one record taking 21 lines of text to be displayed here. The data file comes with a 2,356 page codebook that describes the format, which is essential for determining the locations of the variables.

Figure 2.1 shows screenshots for the codebook entries for the SEX and CASETYPE variables. We see from the codebook that the sex of the patient is provided in columns 36-37. We also see that 1 stands for male, 2 for female, and −8 for 'not documented'. Similarly, the type of visit appears at location 1214 and has 8 possible values.

SEX — **GENDER**

Location: 36-37 (width: 2; decimal: 0)
Variable Type: numeric
Range of Missing Values (M): -8

Value	*Label*	*Unweighted Frequency*	*%*	*Valid %*
1	MALE:(1)	119111	52.0 %	52.0%
2	FEMALE:(2)	110030	48.0 %	48.0%
-8 (M)	NOT DOCUMENTED:(-8)	70	0.0 %	-

Based upon 229141 valid cases out of 229211 total cases.

CASETYPE — **TYPE OF VISIT**

Location: 1214-1214 (width: 1; decimal: 0)
Variable Type: numeric

Value	*Label*	*Unweighted Frequency*	*%*	*Valid %*
1	SUICICDE ATTEMPT:(1)	9033	3.9 %	3.9%
2	SEEKING DETOX:(2)	14841	6.5 %	6.5%
3	ALCOHOL ONLY (AGE < 21):(3)	7421	3.2 %	3.2%
4	ADVERSE REACTION:(4)	88096	38.4 %	38.4%
5	OVERMEDICATION:(5)	18146	7.9 %	7.9%
6	MALICIOUS POISONING:(6)	793	0.3 %	0.3%
7	ACCIDENTAL INGESTION:(7)	3253	1.4 %	1.4%
8	OTHER:(8)	87628	38.2 %	38.2%

Based upon 229211 valid cases out of 229211 total cases.

Figure 2.1: Screenshot of Codebook Entries for the DAWN Survey . *The codebook entry for a variable provides the variable name, definition, location in the row, possible values and their labels, etc. For the sex variable, these are* SEX*;* GENDER*; 36-37;* 1*,* 2*, and* −8*; and male, female, and 'not documented', respectively. The counts for each value are provided to help us check whether or not we have correctly read the data.*

We have skimmed the codebook and selected several variables to extract for further investigation. To do this, we place the starting position of each variable in a vector, which we call `start`; the ending positions appear in `end`, and variable names of our choosing are in `varNames`, e.g.,

```
start = c(1, 9, 11, 14, 15, 30, 34, 36, 38, 45, 46, 66, 68, 70,
          75, 80, 87, 94, 1214, 1215, 1217, 1218, 1219, 1220, 1221)
end = c(6, 10, 13, 14, 29, 33, 35, 37, 39, 45, 47, 67, 69, 74, 79,
        86, 93, 100, 1214, 1216, 1217, 1218, 1219, 1220, 1221)
varNames = c("id", "strata", "psu", "replicate", "wt",
             "psuframe", "age", "sex", "race", "daypart",
             "numsubs", "toxtest", "s1", "s2", "s3", "s4",
```

```
              "s5", "s6", "type", "disp", "alc", "nonalc",
              "pharma", "nonmedpharma", "allabuse")
```

We use the fwf_positions() function in `readr` to collect this information into one object that we pass to the read_fwf() function, i.e.,

```
library(readr)
pos = fwf_positions(start, end, col_names = varNames)

dawn = read_fwf("ICPSR_34565/DS0001/34565-0001-Data.txt",
                col_positions = pos)
```

We explore and check values for these variables later in Section 2.3.2.

2.2.3 Key-Value Pairs

We mention one other plain text format: the key-value pair. The idea is that the value for a variable in a record is provided along with its variable name, which is called the 'key'. A typical key-value pair appears as `age:77`, i.e., the delimiter between the key, `age` and its value `77` is often a colon. Another common delimiter is the equal sign. A typical record has each variable appearing on its own line in the source file with records separated by blank lines. For example, below are the key-value pairs for the first two family members.

```
firstName:Tom
sex:m
age:77
height:70
weight:175
bmi:25.16
overWt:TRUE

firstName:Maya
sex:f
age:33
height:64
weight:125
bmi:21.50
overWt:FALSE
```

This particular arrangement of key-value pairs is know as *dcf*, short for Debian control format. It is commonly used to hold properties lists on UNIX-type operating systems. Another arrangement of key-value pairs has all of the key-value pairs for one record appear on one line with delimiters separating these pairs.

We can read the family data in this *dcf* format into *R* with the read.dcf() function as follows:

```
family.dcf = read.dcf("family.dcf")
```

When we examine the first few values of `family.dcf` and check the class, we find that all of the values are character strings and the structure is a matrix, not a data frame, i.e.,

```
head(family.dcf)
```

```
     firstName sex age  height weight bmi      overWt
[1,] "Tom"     "m" "77" "70"   "175"  "25.16" "TRUE"
[2,] "Maya"    "f" "33" "64"   "125"  "21.50" "FALSE"
[3,] "Joe"     "m" "79" "73"   "185"  "24.46" "FALSE"
[4,] "Robert"  "m" "47" "67"   "156"  "24.48" "FALSE"
[5,] "Sue"     "f" "27" "61"   "98"   "18.51" "FALSE"
[6,] "Liz"     "f" "33" "68"   "190"  "28.95" "TRUE"
```

```
class(family.dcf)
```

```
[1] "matrix"
```

The `matrix` class is discussed in more detail in Section 2.4. We leave to the exercises the task of converting this matrix into a data frame for analysis.

Reading Tabular Data

Data are often naturally arranged in a table-like format in plain text files. To read the raw data into *R*, we need to know how the records and their values are stored in the source file. We can address the following questions to ascertain this information.

- Is the file plain text? We can often determine this by viewing the file contents in a plain text editor. More simply, we can examine the filename extension (csv, txt, fwf, and dcf are examples of plain text filename extensions) or check any available documentation.
- Can the data be arranged in a table format, i.e., with rows for observations/records and columns for variables?
- How are the values for the variables distinguished –
  - Are there delimiters between values (e.g., comma-separated values)?
  - Does a value for a variable appear in the same position in each line in the source file (fixed width format)?
  - Are the values supplied in key-value pairs (with the key as the variable name)?
- Is there any encoding for special characters?

2.3 Validating and Cleaning the Data

After we read data into *R* we want to check the results to determine whether: a) we have correctly read the data; b) the data require further processing for analysis; c) there are problems with the data; and d) the data have unexpected characteristics that influence our approach to their analysis. Some typical mistakes that we can make when reading the data into *R* include the incorrect specification of the file format, e.g., misspecification of the delimiter, the presence or absence of a header line, and incorrect data types. The most common data type misspecification occurs when strings are treated as a `factor` rather than a `character` class, or vice versa. How do we decide between `factor` and `character`? If there are only a fixed set of pre-determined values the variable can take and these correspond

to categories in a variable, then we typically use the `factor` data type. On the other hand, when the strings can take on potentially any possible value, such as with a street address or person's name, then we typically use the `character` class. We provide an example of data-type conversion for the traffic data in Section 2.3.1.

What kind of further processing might we need to do? One example is the conversion of strings into a factor. Another example is the specification of missing values. We may want to re-read the data and specify how missing values are represented in the source file via a parameter in the function we use to read the data. For example, the *na* parameter in read_delim() allows us to provide a character vector of missing values which are converted into `NA` in the resulting data frame. Alternatively, we can replace specific values in a variable in the data frame, such as `-8` for gender, with `NA`. We provide an example of this with the DAWN survey data in Section 2.3.2. We may also need to process the data to create new variables. In Section 1.7 we calculated desired BMI from `height` and `desiredWt` for the `family`. This new variable can be added to the data frame to keep all measurements on the family together in one structure.

What kinds of problems might we uncover when we validate our data? In addition to the discoveries that we have described already that result from not properly reading the data, we may find problems with the quality of the data. When this happens, we need to determine if it's feasible to clean the data, and if so, how to fix the problems. We provide an example with the Clinton email data in Section 2.3.3.

We leave the topic of exploring the data for insights about the distribution of values to Section 2.7.

2.3.1 Updating Variable Names and Formatting Time for Traffic

In Section 2.2.1.1, we successfully read the traffic data into *R*. Although this is a simple data set, there are a couple of issues we can address to make the data easier to analyze and work with. First, the names of the variables in the data frame are long and contain blanks and quotation marks, i.e.,

```
names(traffic)
```

```
[1] "'Timestamp'"   "'Lane 1 Occ'"  "'Lane 1 Flow'" "'Lane 2 Occ'"
[5] "'Lane 2 Flow'" "'Lane 3 Occ'"  "'Lane 3 Flow'"
```

These variable names are cumbersome to use in expressions, e.g., to compute summary statistics for the flow in lane 1 we call summary() with

```
summary(traffic$"'Lane 1 Flow'")
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    0.0    17.0    67.0    65.8   101.0   203.0
```

We can rename these variables with shorter simpler names, e.g.,

```
names(traffic) = c("time", "occ1","flow1","occ2", "flow2",
                   "occ3", "flow3")
```

Notice that when we place `names(traffic)` on the lefthand side of an assignment statement, then the variable names are reassigned.

Alternatively, we can specify these shorter names when we read in the data. In this case, we want to skip the first line of the file (it contains the variable names) and specify the names ourselves. We can do this with

```
traffic = read_delim("flow-occ.txt", delim = ",", skip = 1,
                     col_names = c("time", "occ1","flow1","occ2",
                                   "flow2", "occ3", "flow3"))
```

Now it's much easier to write expressions with these variables, e.g.,

```
summary(traffic$flow1)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    0.0    17.0    67.0    65.8   101.0   203.0
```

Another issue with these data is that the times are stored in a character vector. Strings are not very useful in data analysis. For example, we can't easily use them to examine the times of day when traffic flow is at its peak. However, we can convert these strings to a time data type with

```
traffic$time = as.POSIXct(traffic$time,
                          format = "%m/%d/%Y %H:%M:%S")
```

Here, the format parameter specifies how the times are reported, e.g., `"3/14/2003 00¬ :00:00"` corresponds to a `month/day/year hour:minutes:second` format and the shorthand for this format is `"%m/%d/%Y %H:%M:%S"`. The `POSIXct` data type for dates is a standard format developed by the IEEE (Institute of Electrical and Electronics Engineers) Computer Society and is recognized and properly handled by many *R* functions. For example, we can plot flow against time with the plot() function, and plot() automatically labels the x-axis tick marks so they display the days of the week (see Figure 2.2).

```
plot(flow1 ~ time, data = traffic, type = "l",
     ylab = "Number of cars per 5 minute interval", xlab = "Time")
```

The plot in Figure 2.2 aids us with data validation. We can see that the peaks of traffic correspond to morning and evening rush hours and that traffic on the weekend has a lower volume and is less sharply peaked, which indicates that we have properly read and converted the time values. We continue our exploration of these data in Section 2.5.1 and Section 2.7.1.

2.3.2 Data Types for DAWN Survey

The codebook for the DAWN survey provides extensive documentation of the data, and we can use the information there to validate our work. We begin by examining the structure of our data frame to find that it has the correct dimensions, 229211 by 25, and that the variables are all `numeric`/`integer`. We do this with

```
str(dawn)
```

```
Classes 'tbl_df', 'tbl' and 'data.frame':
  229211 obs. of  25 variables:
 $ id          : int  1 2 3 4 5 6 7 8 9 10 ...
 $ strata      : int  25 29 7 8 22 10 3 51 3 46 ...
 $ psu         : int  108 129 25 29 94 40 6 232 5 210 ...
 $ replicate   : int  2 2 1 2 2 1 1 2 2 2 ...
 $ wt          : num  0.943 5.992 4.723 4.08 5.178 ...
 $ psuframe    : int  3 9 6 6 10 112 7 25 7 2 ...
 $ age         : int  4 11 11 2 6 11 5 8 1 7 ...
 $ sex         : int  1 1 2 1 1 1 2 1 1 1 ...
 $ race        : int  2 3 2 3 3 -8 3 1 3 1 ...
...
```

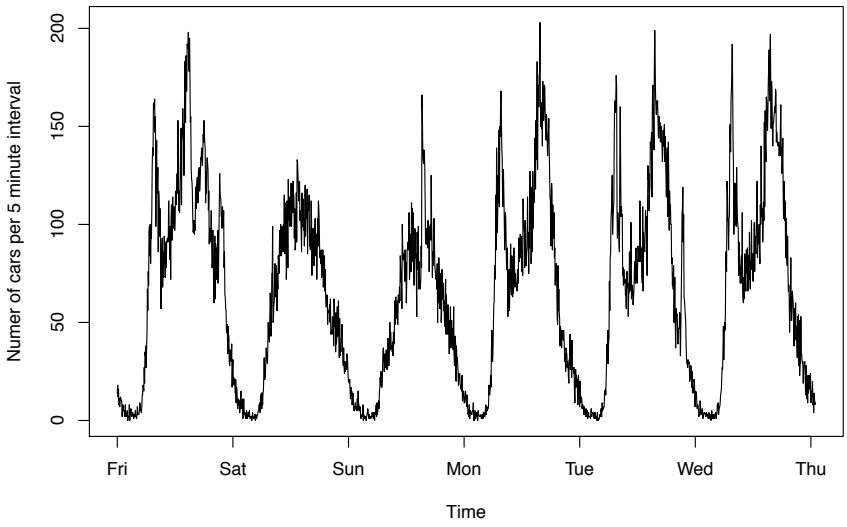


Figure 2.2: Traffic Flow in Leftmost Lane. *This line plot shows how the traffic flow varies throughout the day and across days. The sharp peaks correspond to heavy traffic, for Monday through Friday we see roughly two of these each day for morning and evening rush hours. The traffic on the weekend is generally lighter and less spiked.*

The str() function provides basic information about the variables, but to learn about the distribution of the values for a variable, we need another approach.

We can compare tallies of each variable's values to the codebook to see whether or not we have correctly identified positions of the variables in the records. With fixed-width formats, it is easy to make a mistake in specifying the start or end position of a variable. If a codebook is not available, then we want to examine these tallies to check that the values are reasonable. We check `sex`, `toxtest`, and `type` with

```
table(dawn$sex)
```

```
-8      1      2
70 119111 110030
```

```
table(dawn$toxtest)
```

```
    -9      1      2
  7946  23329 197936
```

```
table(dawn$type)
```

```
    1     2     3     4     5     6     7     8
 9033 14841  7421 88096 18146   793  3253 87628
```

These counts match those in the codebook (see Figure 2.1).

We also see that `sex` and `toxtest` have some negative values. Survey data often use different (and atypical) values to denote various reasons for missing information. The DAWN codebook tells us that -7 means not applicable, -8 is for undocumented, and -9 codes for

missing. We must decide whether we want to treat all 3 values as NA or keep the values as is and decide how the missing values are handled for each computation. In this instance, there are only a few missing values and there does not appear to be more than 1 type of missing value for any variable so we assign all negative values to NA. To do this, we re-read the raw data with read_fwf() and use the functions *na* argument to specify the values to convert to NA. We do this with

```
missing = c("-7", "-8", "-9")
dawn = read_fwf("ICPSR_34565/DS0001/34565-0001-Data.txt",
                col_positions = pos, na = missing)
```

We check **sex** and **toxtest** to confirm that −7 and −9 have been converted into NA. We do this with simple summary statistics, e.g.,

```
summary(dawn$sex)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   1.00    1.00    1.00    1.48    2.00    2.00      70
```

```
summary(dawn$toxtest)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
      1       2       2       2       2       2    7946
```

Notice that these statistics are not particularly meaningful. What does it mean that the average value of **sex** is 1.48? The problem is that **sex** is a categorical variable so averages are not useful summaries (the same holds for **toxtest** and **type**). A more meaningful statistic is the proportion of males and females. If the variable is a factor, then the summary() function provides that sort of tally.

Survey data often contain many categorical variables, which is the case with the DAWN data. This typically occurs because the questions on a survey ask the respondent to choose an answer from a list of possibilities, e.g. race, education level. Also, for privacy reasons, exact values are often not ascertained, e.g., income level. Unlike the functions to read data in base *R*, the functions in the readr package do not allow the caller to specify factor as a category. This means that we must convert these variables to factors after we have read them into a data frame. To do this, we use the factor() function with

```
dawn$sex = factor(dawn$sex, labels = c("male", "female"))
dawn$toxtest = factor(dawn$toxtest, labels = c("yes", "no"))

typeLabels = c("suicide", "detox", "alcohol", "adv react",
               "overmed", "poisoning", "acc ingest","other")
dawn$type = factor(dawn$type, labels = typeLabels)

ageLabels = c("5 & under", "6 - 11", "12 - 17", "18 - 20",
              "21 - 24", "25 - 29", "30 - 34","35 - 44",
              "45 - 54", "55 - 64", "65 & over")
dawn$age = factor(dawn$age, labels = ageLabels, ordered = TRUE)
```

Note that we have converted **age** to an ordered factor because the categories correspond to age intervals that have an ordering, e.g., a person in the '5 & under' category is younger than someone in the '6 - 11' category, and so on. The variables **sex**, **toxtest**, and **type** are not ordered.

Again, we confirm that the tallies for the various categories for these variables match the codebook in Figure 2.1.

```
summary(dawn$sex)
```

```
  male female   NA's
119111 110030     70
```

```
summary(dawn$type)
```

```
   suicide      detox    alcohol  adv react    overmed
      9033      14841       7421      88096      18146
 poisoning acc ingest      other
       793       3253      87628
```

These counts match those in the codebook, except that the labels for the levels are used rather than numeric value and the counts for negative values are shown as counts for NAs.

2.3.3 Validating Text – Hillary Clinton's Email

The file made available on Kaggle (see Q.2-4 (page 52)) is called *Emails.csv*. To confirm that the values are indeed comma-separated and whether or not there is a header, we open the file in TextEdit (see Figure 2.3). There we see that the file contains a header line and the values in the first row are comma-separated. Given this information, we read the raw data into *R* with

```
emails = read_delim("clinton/Emails.csv", delim = ",",
                    col_names = TRUE)
```

We shorten the name of the variable `SenderPersonId` to `FrId` (short for 'From Id') and examine the first few records of the data frame to compare the values of `ExtractedFrom` and `MetadataFrom` as we would expect them to be the same or similar. There is limited documentation about these variables on Kaggle and the associated github site so we want to explore them in greater depth to understand their distinctions.

```
names(emails)[6] = "FrId"
head(emails[ , c("ExtractedFrom", "MetadataFrom", "FrId")], 10)
```

```
Source: local data frame [10 x 3]

                              ExtractedFrom      MetadataFrom  FrId
                                      (chr)             (chr) (int)
1 Sullivan, Jacob J <Sullivan11@state.gov> Sullivan, Jacob J    87
2                                       NA                NA    NA
3      Mills, Cheryl D <MillsCD@state.gov>   Mills, Cheryl D    32
4      Mills, Cheryl D <MillsCD@state.gov>   Mills, Cheryl D    32
5                                       NA                 H    80
6                                       NA                 H    80
7      Mills, Cheryl D <MillsCD@state.gov>   Mills, Cheryl D    32
8                                       NA                 H    80
9  Sullivan, Jacob J <Sullivanli@stategov> Sullivan, Jacob J    87
10                                      NA                NA    NA
```

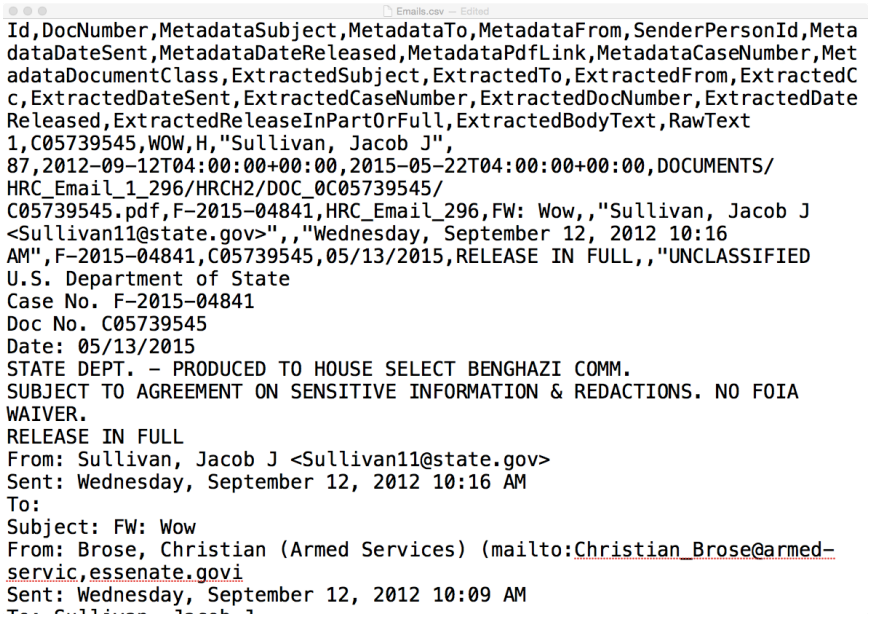


Figure 2.3: Screenshot of a TextEdit Display of *Emails.csv*. *From this TextEdit display we see that the first line (which is wrapped across 5 lines in the display) contains a header and the variable names are comma-separated. The remainder of the display shows the contents of the second record in the file, the bulk of which is the value for the RawText field.*

We make several observations: it appears that `H` in `MetadataFrom` stands for Hillary Clinton; the `FrId` appears to be a unique numeric identifier for the sender; the extracted version of the 'from' field contains both names and email addresses and the metadata version contains only names; Jacob J Sullivan has 2 different email addresses and the second one appears to be incorrect as the domain is `stategov` rather than `state.gov`. This suggests that we may want to investigate how clean are the data.

We have found out a lot about the data from looking at the first 10 records. However, this is a very small fraction of the data and we want to examine more to learn about the variables. One way to do this is to examine the distribution of the frequency of senders, i.e., the counts of emails received from different addresses. We use `MetadataFrom` to compute these counts, as the meta data are provided by the Freedom of Information Act (FOIA) release and may be more accurate than the values extracted from the PDFs.

```
fromCounts = table(emails$MetadataFrom)
```

We make a histogram of these counts with

```
hist(fromCounts, xlab = "Emails Sent", main = "")
```

The histogram appears in Figure 2.4. We can see that the number of emails from different addresses is highly skewed with a handful of addresses sending hundreds (even thousands) of emails.

Let's examine a few of the frequent senders; we can do this by sorting the counts in the `fromCounts` table and examining, say, the top 5:

```
sort(fromCounts, decreasing = TRUE)[1:5]
```

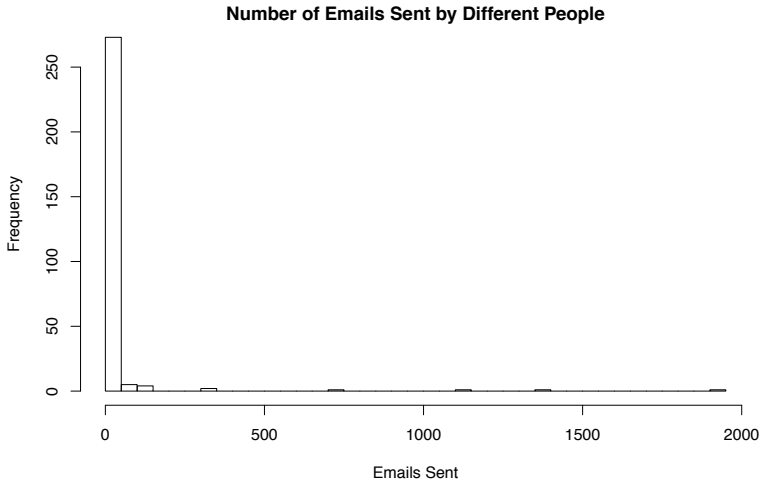


Figure 2.4: Distribution of the Number of Emails Sent to Clinton. *This histogram shows that most people who send emails to Clinton send very few, i.e., well under 25, and a few people send hundreds, even thousands of emails. To get a better sense of the distribution, we may want to plot the histogram on log scale, or to zoom in on those sending fewer than 100 emails.*

```
                H      Abedin, Huma   Mills, Cheryl D
             1906              1380              1146
Sullivan, Jacob J         sbwhoeop
              750               316
```

In hindsight, these are not surprising because the emails include those sent and received by Clinton, and because Mills, Abedin, and Sullivan are 3 of Clinton's long-time staff and advisors. We earlier uncovered a problem with multiple values for Sullivan in `ExtractedFrom` so it makes sense to check whether there are any issues of multiplicity of identifiers in `MetadataFrom`. To do this, we use order() to find the cell in the table that corresponds to this count for Sullivan.

```
order(fromCounts, decreasing = TRUE)[4]
```

```
[1] 257
```

This implies Sullivan is the 257th record in `fromCounts`. Since the table `fromCounts` is in alphabetical order according to `MetadataFrom`, let's examine the neighboring cells to `Sullivan, Jacob J`, e.g.,

```
fromCount[253:260]
```

```
      Strider, Burns Sullivan JJ@state.gov        Sullivan, Jack
                   2                     2                     2
     Sullivan, Jacob     Sullivan, Jacob J        Sullivan, Jake
                   2                   750                    37
    Sullivan, Jake J  SullivanJJ@state.gov
                  14                    24
```

It appears that 7 of these entries correspond to the same person, but their names differ slightly. That is, the values in `MetadataFrom` include: a nickname such as Jake, rather than Jacob; the presence of a middle initial J or not; and an email address rather than a person's name. The most common form of Sullivan's `MetatdataFrom` value occurs 750 times and the other forms have counts that vary between 2 and 37. This may seem small, but when added together they amount to 81 emails, which is about 10% of the total sent by Sullivan.

We also want to corroborate the sender ID with these values, i.e., do each of these variants of Sullivan's name have the same `FrId`? We saw from the first few rows of the data frame that Sullian is associated with the id `87`. Let's count how many records have an `FrId` of `87`. We do this with

```
sum(emails$FrId == 87, na.rm = TRUE)
```

```
[1] 871
```

This is 40 more than the counts we have been examining. On closer inspection of the `fromCounts` table, we find that there are 3 more cells for Sullivan that begin with `'jake'` and these account for 40 emails. This indicates that if we want to track who is who, then we better use `FrId`. Furthermore, if we want to associate a name with the `FrId` number, then we should use a second file provided by Kaggle called *Persons.csv*. This file links the identifiers with names. We take this task up in Section 2.6.1.

These explorations highlight the problems related to working with data derived from text. Text data are often very messy and it can be a long process to clean the data to get more accurate values. In this cleaning process, it's advisable to use code to change values in variables, rather than editing the source file 'by hand'. This way there is a record of how the data were modified, and if the data are later updated and reissued, then the code can be re-run to clean the new version. We do not pursue this topic further here, but note that to edit text data, we typically employ string manipulation techniques and regular expressions. These are the topic of Chapter 8.

Data Validation Checklist

In our initial examination of the data, we want to check the following characteristics of the variables to confirm that they are reasonable and as expected.

- Range of values
- Rough distribution of values
- Prevalence of missing values
- Unusual or anomalous values
- Consistency across subgroups
- Consistency between variables

To validate the data, we typically compute the following numerical summaries and visualizations:

- Examine the first and last few records
- Compute numerical summaries (min, max, quartiles, etc.) for quantitative data and tables of counts of values for qualitative data.

- Visualize the data with
  - univariate plots of individual variables, e.g., histograms, density curves, bar plots, dot charts.
  - comparative plots of individual variables across subgroups, e.g., side-by-side box plots, bar plots, and dot charts, and super-posed density plots.
  - bivariate plots of relationships, e.g., scatter plots, mosaic plots.

  These plots are reviewed in Chapter 3.

2.4 Selecting a Structure: Data Frame, Matrix and Array

The data frame is a natural structure for tabular data, where our variables can have different data types and the rows all have the same number of variables in each of them. However, it isn't the sole way for us to represent data in *R*. We introduce some alternatives in Chapter 4, including examples of list structure for data that are provided in *JSON* format. In addition in Chapter 9, we examine data arranged in multiple tables in relational databases, and in Chapter 12, we introduce the *XML* format which can represent complex data structures. In this section, we introduce one more example of tabular data that are naturally stored in a collection of matrices, rather than a data frame. With a matrix, the columns must all have the same data type. For this reason, the matrix does not have the same flexibility as the data frame, but nonetheless, it can be a very useful structure. We present an example, where the columns all contain the same type of quantity: pixels in an image that take on values ranging between 0 and 255. We also introduce functionality for working with matrices; in particular, we demonstrate how to use the apply() function, which is especially suited for working with matrices and arrays.

2.4.1 Collections of Matrices – Handwritten Digits Data

In Q.2-5 (page 53), we described data containing thousands of images of handwritten digits. Each image consists of a 28×28 pixel grid, where each pixel has an integer value between 0 and 255. All together, there are 28×28 or 784 pixel values for each image. As described in that example, we can think of a tabular format for these data where a row represents an image and each row contains 784 values, corresponding to a sequence of the 784 pixels in the image. That is, the 784 values are arranged so that the same pixel position in each image falls in the same column in the table. However, there is an alternative arrangement that is a more natural organization of the values as a matrix for each image. In this case, the cells in the matrix map directly to the layout of the pixels in the image, i.e., each image is represented as a 28×28 matrix of values. For example, the value in the 1st row and 28th column of the matrix corresponds to the top right pixel in the image. Furthermore, we can collect these matrices into a set of matrices, i.e., a 3-dimensional array. In this array, the 1st dimension corresponds to a row of pixels in an image, the 2nd to a column of pixels in that image, and the 3rd dimension of the array corresponds to the different images. Figure 2.5 provides a conceptual diagram of a similar, but smaller structure.

The files that contains these data are not plain text. The values are in a binary format, where each pixel is represented by 1 byte, i.e., a collection of 8 bits (or 0-1 values). For example, the 8-bit binary representation of the number 121 is `01111001` because these 8

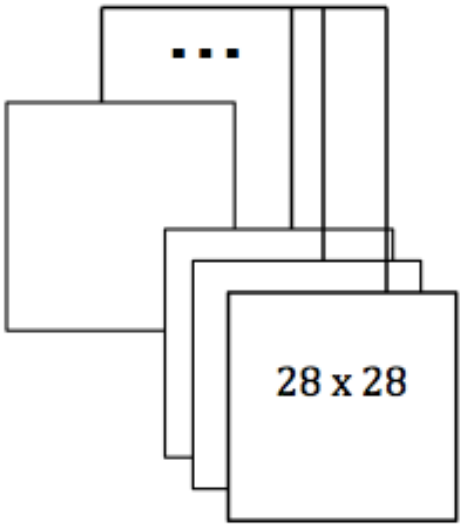


Figure 2.5: Conceptual Diagam of a 3-Dimensional Array. *This diagram shows a conceptual representation of a 3-dimensional array for handwritten digits. It is a collection of 1000 matrices that have 28 rows and 28 columns. Each matrix represents the pixel values for one handwritten image.*

bits provide the coefficients for the powers of 2 that correspond to that number, i.e.,

$$0 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$$

Notice the exponent corresponds to the position in the binary number and the coefficient is the value at that position. We typically work with numbers in base 10, but the same idea holds. That is, 121 is

$$1 * 10^2 + 2 * 10^1 + 1 * 10^0$$

In base 10, the coefficients can range from 0 to 9.

There is a lot of flexibility in how binary data can be stored, which means that we have to provide more information in order to read the values in the file. We must specify that the data are 1-byte unsigned integers, meaning that each number consists of 8-bits like the example of 121 as `01111001`. In addition, we must specify how many bytes to read from the file.The online documentation indicates that each file contains 1000 images so we want to read $1000 * 28 * 28$, or 784000 bytes from a file. We also must manage the opening and closing of the file for reading. That is, we need to establish a connection to a file so that we can read the data, and once we have completed reading the data, we close the connection.

We begin by reading the data for the digit 4. The images for each digit are stored in separate files, e.g., *data4* contains all 1000 images for 4. We chose this digit because it is asymmetric so that we can easily tell from a plot if we have correctly read the data or mistakenly reversed the order of the pixels. We open the connection to the file, *data4* with

```
digitdata = file("data4", "rb")
```

Then, we read the 1000 sets of 784 pixels with

```
pixels = readBin(digitdata, integer(), size = 1, signed = FALSE,
                 n = 1000 * 28 * 28)
class(pixels)
```

```
[1] "integer"
```

```
length(pixels)
```

```
[1] 784000
```

We see that all $1000 \times 28 \times 28$ pixel values have been read into R as an `integer` vector. We can check that the values are all between 0 and 255 with

```
summary(pixels)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      0       0       0      31       0     255
```

It appears that we have properly read the data so we close the connection with

```
close(digitdata)
```

We can arrange the vector into a 3-dimensional array with

```
imageArray = array(pixels, dim = c(28,28,1000))
dim(imageArray)
[1]   28   28 1000
```

How can we confirm that we have properly arranged the data? For example, does each matrix correspond to an image of the number 4? Given that our data are somewhat unusual in format, we can't answer these questions with simple summary statistics. Instead, to validate our data, we can make a plot that corresponds to the image, i.e., a 28×28 grid of shaded squares where the shading scales with the pixel value.

Let's start by checking the first image, i.e., `imageArray[ , , 1]`. Note that we have indexed into the array using the position of the 3rd dimension and all rows and all columns (see Section 1.9 for more details about how to make subsets). We can create this plot with

```
grays = rev(gray.colors(10))
image(z = imageArray[ , , 1], col = grays,
      xaxt = "n", yaxt = "n")
```

We see in Figure 2.6 that it appears that the image is of the digit 4, except that the y coordinates are arranged incorrectly. We can fix this by plotting `imageArray[ , 28:1, 1]`, i.e., we reorder the y values in each column.

2.4.1.1 Applying Functions to Matrices and Arrays

We have checked that the first matrix in our array of matrices looks like the digit 4, but what about the rest of them? As mentioned earlier, we cannot simply study a statistical summary of these pixel values in an attempt to validate our data. Instead, we can create a unique summary, such as a plot of the average of each pixel across the 1000 images. To calculate this average, we use the apply() function. The apply() function is designed for matrices and arrays. With it, we can apply the provided function across different dimension(s). In this case, we want to take the mean for each row and column entry across the 1000 'pages' of images so we specify the *MARGIN* argument as `1:2`, i.e.,

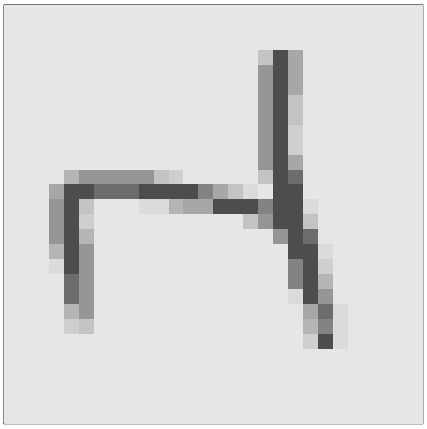


Figure 2.6: The 28×28 Pixel Image of a Handwritten Digit. *This plot is created from the pixel values for an image of a hand written numeral 4. The arrangement of the pixel values in the* 28×28 *matrix is incorrect. We can see that the y-values run in the wrong direction.*

```
avg4 = apply(imageArray, MARGIN = 1:2, FUN = mean)
```

We confirm that the result is a single 28×28 matrix, i.e.,

```
class(avg4)
```

```
[1] "matrix"
```

```
dim(avg4)
```

```
[1] 28 28
```

A plot of `avg4` appears on the left side of Figure 2.7. Although faint, we see that the 'average' image looks like the digit 4. Given the large number of 0 values in the pixels (we saw that over 3/4 of the pixel values are 0), we may want to summarize the pixel values with a statistic other than the mean. As an example, we calculate the upper quartile with

```
uq4 = apply(imageArray, 1:2, quantile, probs = 0.75)
```

The image of the upper quartile of the values for each pixel appears on the righthand side of Figure 2.7. We can see the digit 4 quite clearly.

2.5 Reshaping Data Tables

We may find that the organization of the data is not conducive to a particular analysis, and we want to reshape the data. For example, we may want to stack columns of data together to produce a longer data frame with more rows and fewer columns. Reciprocally, we may determine that analysis is better suited to a wider organization of the data, and we break

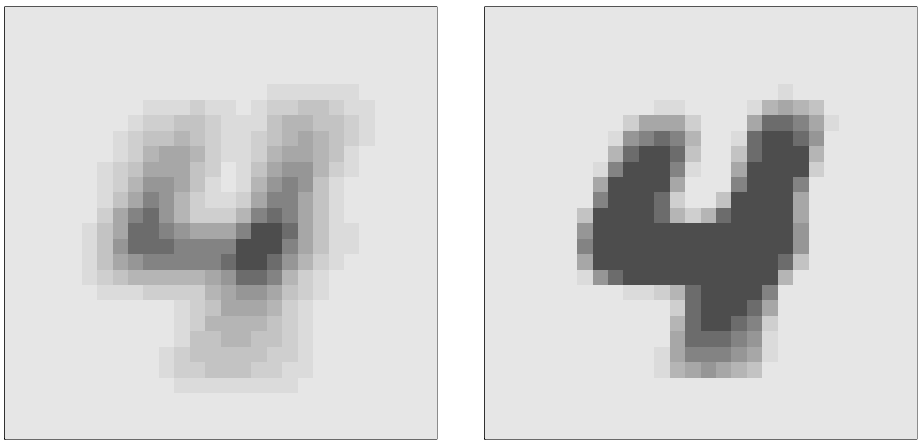


Figure 2.7: Image Plot of 1000 Digits. *The lefthand plot shows the average pixel value for each pixel across 1000 images. The plot on the right shows the upper quartile of the 1000 values for each pixel in the 28-by-28 pixel image.*

columns up into multiple columns with fewer rows. We demonstrate this reshaping with two examples. In Section 2.5.1, we create a longer, narrower data frame from the PeMS traffic data (see Q.2-1 (page 52)), and in Section 2.5.2, we create a wider, shorter data frame from the World Bank country data (see Q.2-3 (page 52)).

2.5.1 Stacking Traffic Flow

The PeMS data provide flow and occupancy over a loop detector in 6 distinct variables for 3 lanes of traffic. These data consist of 3 types of measurements: flow, occupancy, and the lane in which the measurements were made. However, with the current organization, the information about the freeway lane is found in the variable name, not in its own column in the data frame. This organization can be useful, if, for example, we want to compare the flow of traffic in the various lanes. We can make a scatter plot where a point corresponds to the flow for a particular 5-minute interval for the right and left lanes. We make the scatter plot and add a reference line with

```
plot(flow1 ~ flow3, data = traffic, pch = 19, cex = 0.3,
     ylim = c(0, 200), ylab = "", xlab = "",
     main = "Flow in Left Lane vs Right Lane")
abline(a = 0, b = 1, lty = 2, lwd = 3, col = "pink")
```

This plot and a similar plot for the flows in the center and right lanes appear in Figure 2.8. When we compare the locations of the points to the line, we see that the center lane consistently has more vehicles than the right. On the other hand, the left lane has fewer cars than the right when traffic is light, but when traffic is heavy (more than about 50 cars per 5 minutes), then the left lane serves many more vehicles.

For other analyses and plots, this arrangement of flow and occupancy in 6 variables can be inconvenient. For example, if we want to compute summary statistics for flow regardless of lane, then we need to concatenate the 3 vectors in the data frame together, e.g., `summary(c(traffic$flow1, traffic$flow2, traffic$flow3))`. Instead, if the 3 columns

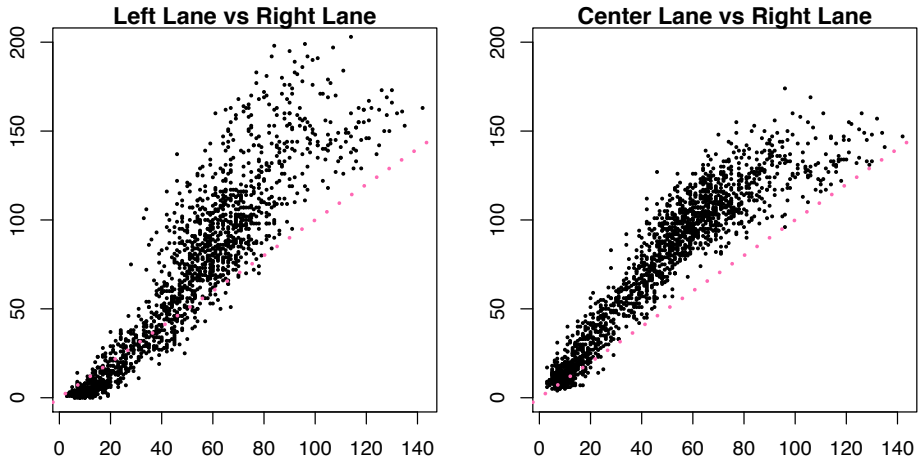


Figure 2.8: Comparison of Traffic Flow in Different Freeway Lanes. *The scatter plot on the left compares flow in the left and right lanes. We see that when throughput is high, the left and center lanes serve more vehicles than the right. Also, it appears that when flow in the right lane exceeds 60, the flow in the other two lanes is more variable.*

are combined into 1, i.e., stacked one on top of the other into `flow`, then we simply call `summary(flow)`. Additionally, if we have a variable that tracks the lane in which the measurements were taken, then we can make lane comparisons with formulas and grouping arguments to the plotting functions. We reformat the data and place the 3 vectors that contain flow measurements one above the other with

```
flow = stack(traffic[ , c("flow1", "flow2", "flow3")])$values
```

The return value from the call to stack() is a two-column data frame with vectors `values` and `ind`; the `values` vector contains the concatenated flow measurements and `ind` contains the name of the original variable (`flow1`, `flow2`, and `flow3`) to indicate the original variable the value cames from. We keep only `values` because we plan to create `lane`, an equivalent but more informative version of `ind`. Similarly, we create `occ` with

```
occ = stack(traffic[ , c("occ1", "occ2", "occ3")])$values
```

Then, we create the lane 'key' as a `factor`; we do this with

```
lane = factor(rep(c("left", "middle", "right"),
                  each = nrow(traffic)))
```

We can combine these variables into a single data frame, and include `time` as well with

```
trafficLong = data.frame(lane, flow, occ,
                         time = rep(traffic$time, 3))
```

We confirm that the data are arranged correctly by examining the first and last few rows in `trafficLong` with head() and tail(); we find,

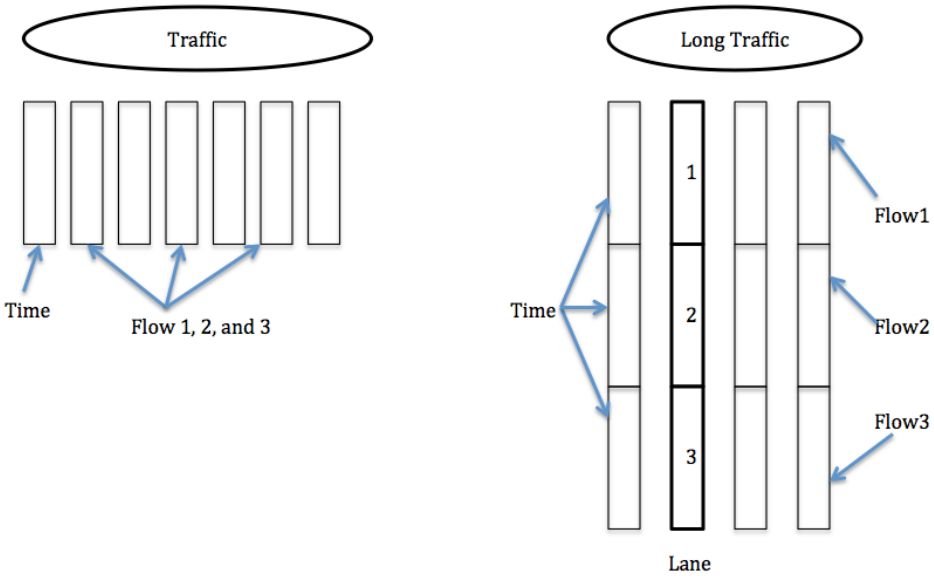


Figure 2.9: Diagram of Stacking Tables. *The `traffic` data frame (left) contains 3 variables with flow information, one variable for each lane. The data frame on the right is a 'stacked' version, where these 3 variables are combined into one. This data frame also has a stacked version of occupancy. We need to create a new variable (`lane`) so that we can keep track of the lane for flow and occupancy. Similarly, we need to stack 3 copies of `time`.*

```
  lane flow    occ                time
1 left   14 0.0100 2003-03-14 00:00:00
2 left   18 0.0133 2003-03-14 00:05:00
3 left   12 0.0088 2003-03-14 00:10:00
4 left   16 0.0115 2003-03-14 00:15:00
5 left    8 0.0069 2003-03-14 00:20:00
6 left   11 0.0077 2003-03-14 00:25:00
...
      lane flow    occ                time
5215 right   18 0.0199 2003-03-20 00:30:00
5216 right    9 0.0059 2003-03-20 00:35:00
5217 right   18 0.0234 2003-03-20 00:40:00
5218 right   13 0.0206 2003-03-20 00:45:00
5219 right    8 0.0063 2003-03-20 00:50:00
5220 right   12 0.0105 2003-03-20 00:55:00
```

It appears that our long data frame has been correctly constructed.

An example of the benefit of the long version of the traffic data frame is in the creation of the line plot in Figure 2.10. We create this from `trafficLong` with the xyplot() function in the `lattice` package, e.g.,

```
library(lattice)
xyplot(flow ~ time, data = trafficLong, groups = lane,
       type = "l", xlab = "", ylab = "Flow",
```

```
        auto.key = list(corner = c(0.2, 1), points = FALSE,
                        lines = TRUE))
```

Notice how the formula `flow ~ time` expresses the desired relationship; that is, the formula indicates that we want to visualize how flow depends on time. We also specify `groups = lane` to indicate that we want to compare this relationship across lanes, and as a result, we obtain 3 (color-coded) line plots in the plotting region. This plot reveals structure in the data that is not evident in Figure 2.8 because here we have the opportunity to observe the relationship between flows in all 3 lanes in time. Specifically, we see the ordering of flow between the 3 lanes change with time. In heavy traffic, the left lane has the greatest flow and the right lane is lowest, but immediately after a peak in flow, the left lane's flow dips sooner and faster to levels below the middle lane.

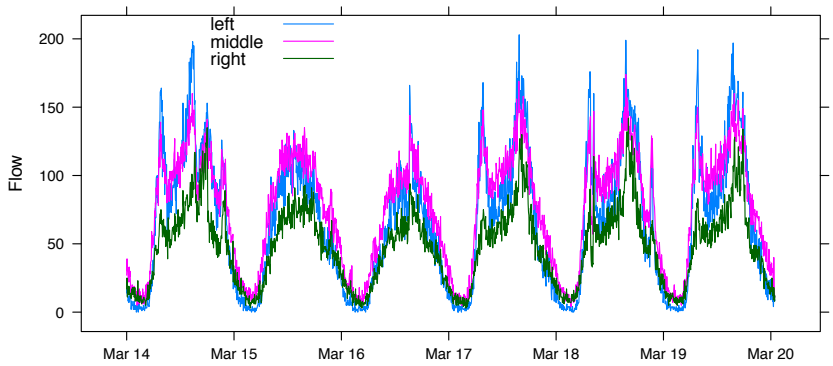


Figure 2.10: Comparison of Traffic Flow Across Lanes Over Time. *These three line plots show the patterns in throughput for 3 lanes of traffic. The 3 lanes have the same general shape, but the time of the peaks differ slightly and the swings from high to low are greater for some lanes than others.*

An alternative approach to stacking the columns uses the gather() function in the `tidyr` package [12]. The gather() function performs the same task as stack() but can be more convenient with its *key* and *value* arguments that allow us to name the vectors in the resulting data frame. We do not take advantage of this feature here because we prefer to create our own factor with labels: left, center, and right. In any event, we can use gather() as follows:

```
library(tidyr)
flow = gather(traffic[, c("flow1", "flow2", "flow3")])$value
occ = gather(traffic[, c("occ1", "occ2", "occ3")])$value
```

We have introduced gather() because other functions in `tidyr` can be quite convenient; in particular the spread() function helps us create a wider, shorter data frame in Section 2.5.2.

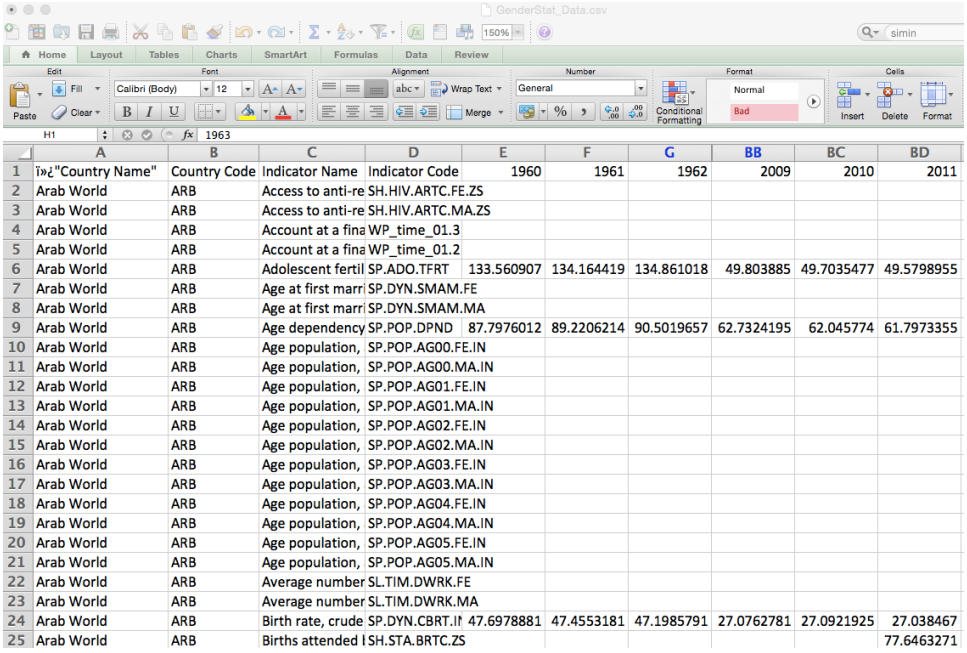
2.5.2 Rearranging World Bank Country Statistics

The data in *GenderStat_Data.csv* come from the World Bank's Open Data Website described in Q.2-3 (page 52). At that site we can choose from a collection of prepared data

files or use an interface to select particular variables and countries of interest. We downloaded the collection of gender related variables in *csv* format. These data include annual measurements from 1960 through 2015.

To get a sense of the format of the data, we can view it in a plain text editor, or we can open it in Excel. With Excel, the software arranges the values into a spreadsheet, much as read_delim() arranges values into a data frame. The spreadsheet view of the data can be easier for us to examine the organization of the data than a plain text editor or data frame in *R* because the columns are aligned. Also, the file is too large for the data viewer in RStudio.

A screenshot of a portion of the Excel spreadsheet appears in Figure 2.11. We immediately notice several characteristics that are important to working with and analyzing these data: a) some of the entries for countries are actually conglomerates, e.g., Arab World; b) the column labeled `'Indicator Name'` contains what appear to be variable names so one country occupies many rows (one row for each variable); c) the columns labeled 1960, 1961, etc. contain the annual values of the variable listed under Indicator Name; d) there appear to be many empty cells. For this example, we simplify the data and concern our selves with one year only. We pick 2010 because it is relatively current and, by observation, appears to have fewer empty cells.



	A	B	C	D	E	F	G	BB	BC	BD
1	ï»¿"Country Name"	Country Code	Indicator Name	Indicator Code	1960	1961	1962	2009	2010	2011
2	Arab World	ARB	Access to anti-re	SH.HIV.ARTC.FE.ZS						
3	Arab World	ARB	Access to anti-re	SH.HIV.ARTC.MA.ZS						
4	Arab World	ARB	Account at a fina	WP_time_01.3						
5	Arab World	ARB	Account at a fina	WP_time_01.2						
6	Arab World	ARB	Adolescent fertil	SP.ADO.TFRT	133.560907	134.164419	134.861018	49.803885	49.7035477	49.5798955
7	Arab World	ARB	Age at first marr	SP.DYN.SMAM.FE						
8	Arab World	ARB	Age at first marr	SP.DYN.SMAM.MA						
9	Arab World	ARB	Age dependency	SP.POP.DPND	87.7976012	89.2206214	90.5019657	62.7324195	62.045774	61.7973355
10	Arab World	ARB	Age population,	SP.POP.AG00.FE.IN						
11	Arab World	ARB	Age population,	SP.POP.AG00.MA.IN						
12	Arab World	ARB	Age population,	SP.POP.AG01.FE.IN						
13	Arab World	ARB	Age population,	SP.POP.AG01.MA.IN						
14	Arab World	ARB	Age population,	SP.POP.AG02.FE.IN						
15	Arab World	ARB	Age population,	SP.POP.AG02.MA.IN						
16	Arab World	ARB	Age population,	SP.POP.AG03.FE.IN						
17	Arab World	ARB	Age population,	SP.POP.AG03.MA.IN						
18	Arab World	ARB	Age population,	SP.POP.AG04.FE.IN						
19	Arab World	ARB	Age population,	SP.POP.AG04.MA.IN						
20	Arab World	ARB	Age population,	SP.POP.AG05.FE.IN						
21	Arab World	ARB	Age population,	SP.POP.AG05.MA.IN						
22	Arab World	ARB	Average number	SL.TIM.DWRK.FE						
23	Arab World	ARB	Average number	SL.TIM.DWRK.MA						
24	Arab World	ARB	Birth rate, crude	SP.DYN.CBRT.I	47.6978881	47.4553181	47.1985791	27.0762781	27.0921925	27.038467
25	Arab World	ARB	Births attended I	SH.STA.BRTC.ZS						77.6463271

Figure 2.11: Excel Spreadsheet of World Bank Data. *This screen shot displays a portion of a World Bank csv file that has been opened in Excel.*

Even with this simplification, the data have an unusual organization, i.e., the variable names are in one column rather than as names across the top of several columns. To get a better idea as to how we might reshape our data for analysis, we consider a simplified version of this format. In the data frame below, we have 3 countries (A, B, and C) and 2 variables (V1 and V2). Each country has 2 records in the data frame, one for each variable. The column, `vals`, contains the measurement of a variable for a country, and we can tell

which variable the measurement is associated with by examining the contents of `varName`, e.g., country `C` has a value of 29 for variable `V2`.

```
  ctry varName vals
1    A      V1   11
2    A      V2   20
3    B      V1   12
4    B      V2   24
5    C      V1   14
6    C      V2   29
```

We want to turn this data frame into a wider, shorter data frame where there is one row for each country, and a column for each variable (we want variables `ctry` and `V1` and `V2`). The spread() function in the `tidyr` package can do exactly this with

```
spread(x, key = varName, value = vals)
```

```
  ctry V1 V2
1    A 11 20
2    B 12 24
3    C 14 29
```

Now we have a 3×3, rather than 6×3, data frame.

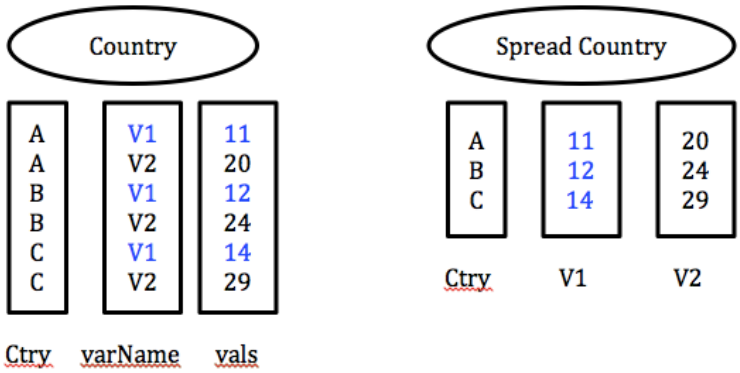


Figure 2.12: Conceptual Diagam of a 3-Dimensional Array. *This diagram shows a conceptual representation of a 3-dimensional array for handwritten digits. It is a collection of 1000 matrices that have 28 rows and 28 columns. Each matrix represents the pixel values for one handwritten image.*

We can apply this same technique to our World Bank data. We begin by reading the data into *R* with read_delim(), i.e.,

```
wb = read_delim("worldBank/GenderStat_Data.csv", delim = ",")
```

Then we drop the first 16534 rows because they contain aggregate data for regions rather than countries. We do this with subsetting by exclusion, e.g.,

```
wb = wb[-(1:16534), ]
```

We want to reshape our data as we did with the toy example. To do this, we need to identify the columns corresponding to the country name, variable name, and variable values. From the spreadsheet, we see that these are "Country Code", "Indicator Name", and "2¬ 000". We call spread() with the reduced wb data frame as follows:

```
wbWide = spread(data = wb[, c("Country Code", "Indicator Name",
                              "2000")],
                key = "Indicator Name", value = "2000")
```

An alternative approach uses the select() function in tidyr, rather than the square-bracket subsetting of wb, e.g.,

```
spread(data = select(wb, c(2, 3, 45)),
       key = "Indicator Name", value = "2000")
```

Or, we can also use the one_of() helper function in dplyr to specify the column names rather than positions, e.g.,

```
spread(data = select(wb, one_of("Country Code",
                                "Indicator Name", "2000")),
       key = "Indicator Name", value = "2000")
```

Some users find the select() and one_of() functions in dplyr easier to use as they do not involve the [operator. However, we will see in Section 4.2.1 that [is more versatile.

Let's compare the shapes of wb and wbWide with

```
nrow(wb)
```

```
[1] 107213
```

```
dim(wbWide)
```

```
[1] 214 502
```

The variable names in wbWide are cumbersome to work with as they are long and include blanks and special characters, e.g., the first 6 variable names are:

```
head(names(wbWide))
```

```
[1] "Country Code"
[2] "Access to anti-retroviral drugs, female (%)"
[3] "Access to anti-retroviral drugs, male (%)"
[4] "Account at a financial institution, female (% age 15+) [ts]"
[5] "Account at a financial institution, male (% age 15+) [ts]"
[6] "Adolescent fertility rate (births per 1,000 women ages 15-19)"
```

Let's rename the variables to shorter names that are easier to type.

For simplicity, we also limit the variables to explore. We are interested in gross domestic product (in column 200), health expenditure per capita (214), life expectancy (259), literacy rate of adult female (260), infant mortality rate (289), total population (340), prevalence of female obesity (347), and public spending on education (374). And, of course, country code (1). We create a vector of short names for these variables, and we create a subset that contains only these few variables. We do this with

```
varNames = c("ctry", "gdp", "healthPC", "lifeExp", "litRateF",
             "infantMort", "pop", "obesityFemale", "edSpending")

wbSub = wbWide[ , c(1, 200, 214, 259, 260, 289, 340, 347, 374)]
names(wbSub) = varNames
```

We view the first few records in this data frame to confirm that our subsetting worked as expected, e.g.,

```
head(wbSub)
```

```
Source: local data frame [6 x 9]

   ctry       gdp healthPC lifeExp litRateF infantMort      pop
  (chr)     (dbl)    (dbl)   (dbl)    (dbl)       (dbl)    (dbl)
1   ABW 1.873e+09       NA   73.72    97.07          NA    90858
2   ADO 1.402e+09  1954.98      NA       NA         3.9    65399
3   AFG        NA       NA   55.13       NA        95.4 19701940
4   AGO 9.130e+09    91.14   45.20       NA       128.3 15058638
5   ALB 3.632e+09   248.44   74.27       NA        23.2  3089027
6   ARE 1.043e+11  1882.34   74.45       NA         9.6  3050128
Variables not shown: obesityFemale (dbl), edSpending (dbl)
Source: local data frame [6 x 8]
```

It appears that we have properly read and reshaped the data and that many countries have a missing literacy rate.

We examine the summary statistics for all variables with

```
summary(wbSub)
```

```
     ctry                gdp               healthPC
 Length:214         Min.   :1.37e+07   Min.   :   6
 Class :character   1st Qu.:1.73e+09   1st Qu.:  93
 Mode  :character   Median :8.04e+09   Median : 307
                    Mean   :1.66e+11   Mean   : 665
                    3rd Qu.:5.22e+10   3rd Qu.: 754
                    Max.   :1.03e+13   Max.   :4818
                    NA's   :16         NA's   :28
   lifeExp        litRateF      infantMort
 Min.   :38.7   Min.   :12.8   Min.   :  3.1
 1st Qu.:60.0   1st Qu.:55.4   1st Qu.: 10.9
 Median :70.3   Median :81.6   Median : 26.7
 Mean   :67.0   Mean   :72.9   Mean   : 39.7
 3rd Qu.:74.5   3rd Qu.:92.3   3rd Qu.: 64.5
 Max.   :81.1   Max.   :99.8   Max.   :143.3
 NA's   :13     NA's   :171    NA's   :22
     pop           obesityFemale   edSpending
 Min.   :9.42e+03   Min.   : NA   Min.   : 0.80
 1st Qu.:6.20e+05   1st Qu.: NA   1st Qu.: 3.00
 Median :5.10e+06   Median : NA   Median : 4.17
 Mean   :2.85e+07   Mean   :NaN   Mean   : 4.38
 3rd Qu.:1.65e+07   3rd Qu.: NA   3rd Qu.: 5.44
 Max.   :1.26e+09   Max.   : NA   Max.   :11.49
                    NA's   :214   NA's   :93
```

From this numerical output, we make several observations about the data: all values for `obesityFemale` are missing, over 3/4 of `litRateF` is missing, and about half of `edSpending` is missing; the range of `gdp` and `pop` cover 6 orders of magnitude, and the range in `healthPC` spans 3 orders of magnitude; country code is character and since it is unique across rows we do not convert it to a `factor`.

We can get a better sense of the distributions of the variables that have only a few missing values by examining density plots. For example, let's explore life expectancy. From the summary statistics, we see that life expectancy ranges from 38 to 81 with half of the countries having a life expectancy over 70. We make a density plot of life expectancy with

```
library(ggplot2)
ggplot(data = wbSub, aes(x = lifeExp)) + geom_density() +
  geom_rug(col="darkred", alpha = .4) +
  labs(x = "Life Expectancy") + xlim(c(35, 85))
```

Figure 2.13 reveals a skew-left distribution with the great majority of countries having a life expectancy of 70 to 80 years. Yet in some countries, average life expectancy falls well below 65 years. The rug plot along the x-axis allows us to see the concentration of values for the countries. We continue our exploration of these data in Section 2.7.2.

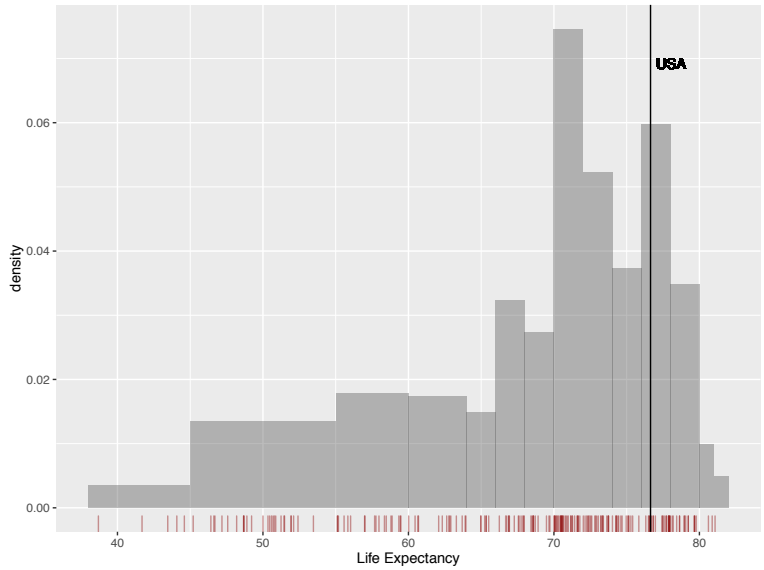


Figure 2.13: Average Life Expectancy for 214 Countries. *This histogram of average life expectancy shows a highly skewed distribution with many countries having a life expectancy below 65. The average life expectancy in the US is marked by a vertical line at 76. The individual country values are displayed in the rug plot below the histogram. These data are for the year 2000 as reported by the World Bank.*

2.6 Merging Data Tables

At times the variables that we want to analyze are not contained in a single data table and we need to merge two (or more) tables together to obtain a structure suitable for analysis. The variables may be divided across tables so some information is contained in one table and other information appears in a second table. In order to use information in both tables, we need to combine them into one table. However, the observations may not line up across tables, meaning that the order of the observations may be different from one table to the other. In order for us to correctly consolidate the variables into one table, we need a way to match observations between the tables. Typically, we have a uniquely identifying 'key' for each observation, and we use this key to match the observations across tables. (See Figure 2.14.)

A further complication sometimes arises where the two tables do not contain the exact same observations. That is, there is overlap in the records, but each table may contain observations (or keys) that do not appear in the other. When this happens, we need to determine whether we want only those observations that appear in both tables, or we want all those that appear in one table even if that means there will be missing values for the records where we do not find a match. As long as we have a key to match the records we can merge the data in any of these ways (see Figure 2.14.

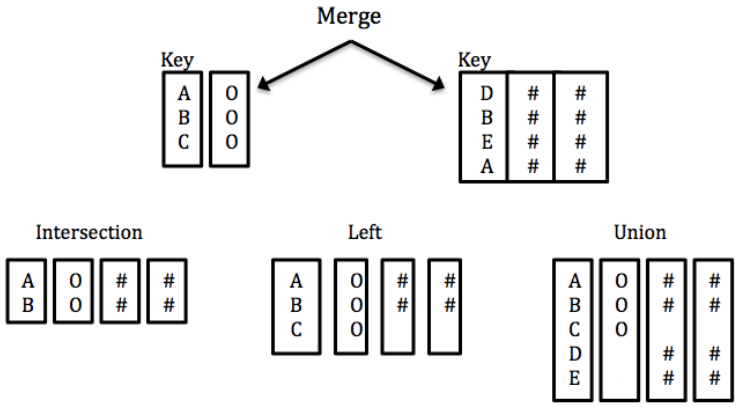


Figure 2.14: Diagram of Types of Merges. *The two data frame in the top row can be merged in several ways. The key for merging is the first column in each data frame. The merged data frame can contain only records where a match is found in both (bottom left), records only from the left data frame (bottom middle) or records from either data frame (bottom right). Missing values (blanks in the diagram) occur when a key appears in one file but not the other.*

We mention one last type of merge. We can have a case where the key is unique in one file but not in the other. For example, with the Clinton email data, we find in the next section that there is a 2-column table that contains an identification number and name for all individuals who sent or received an email. There is one record in this file for each identification numer (or key). In the original table that we examined, we have an identification number for the sender of the email. This file has one record per email, and many people send more than one email so it's possible for there to be many records with the same identification number. Indeed we saw that Hillary Clinton sent thousands of emails,

and her identification number (80) appears thousands of times in this file. Although the key is not unique in the email table, we can merge these 2 tables together in order to add the sender's name to email. This is one of the data merges that we carry out in the next section.

2.6.1 Merging Names and Emails

We discovered in Section 2.3.3 that the variable `MetadataFrom` was not a good source for the name of the email sender. When we wanted to count how many emails each person sent, we did not get accurate counts because some individuals had multiple representations of their name. Instead, it is more accurate to use the identification number that is also supplied in that table. That is, we can tally the emails and examine the top 10 senders with

```
sort(table(emails$FrId), decreasing = TRUE)[1:10]
```

```
  80   81   32   87  194  116  170  124   10  180
1993 1437 1318  871  374  341  159  155  131  118
```

In order to figure out who is person #194, we need to do some sort of look up. We can look at the values for `MetadataFrom`, e.g.,

```
head(emails$MetadataFrom[ emails$FrId == 194 ])
```

```
[1] NA                     NA                     NA
[4] "Blumenthal, Sidney" NA                     "Blumenthal, Sidney"
```

Some of the entries are `NA` while others list Sidney Blumenthal as the sender. We see again why `MetadataFrom` is not very clean.

The file we downloaded from Kaggle was a zipped file that contained several *csv* files. In addition to *Emails.csv* that we read into *R* as the `emails` data frame, the zipped file also contained *Persons.csv*, which has a unique name for each identifier, and *EmailReceivers.csv*, which we describe shortly. When we view *Persons.csv* in a plain text editor (or Excel or RStudio viewer) we see that it contains 2 variables, `Id` and `Name`. When we scroll though this file, we see that there is one name for each identification number. We can match the identification number used in an email record to the identification number in this file and pick up the person's name to add to the record. Then we can tally the emails by this clean version of the sender's name.

To do this, we read in *Persons.csv* with

```
persons = read_csv("Persons.csv", col_names = TRUE)
str(persons)
```

```
Classes 'tbl_df', 'tbl' and 'data.frame': 513 obs. of 2 variables:
 $  Id : int  1 2 3 4 5 6 7 8 9 10 ...
 $ Name: chr  "111th Congress" "AGNA USEMB Kabul Afghanistan" ...
```

We confirm that the unique identifier is called `Id` and the individual's name is held in `Name`. Next, we use `merge()` to combine the `emails` and `persons` data frame with

```
emails = merge(emails, persons, by.x = "FrId", by.y = "Id",
               all.x = TRUE)
```

In addition to providing merge() with the 2 data frames, the arguments we have specified give the names of the variables in each data frame that contain the key to merge on and the type of merge to perform. The variable `FrId` in `emails` and `Id` in `persons` hold the unique identifiers. Since `emails` is the first data frame provided in the function call, it is referred to as x in *by.x*, and *by.y* refers to the second data frame, i.e., `persons`. The expression `all.x = TRUE` indicates that we want to keep all of the records in `emails` so the value in `Name` in `persons` is added to all records where we find a match in the keys. When a record has a value for `FrId` that doesn't match any value in `Id`, then `NA` is assigned to `Name` for that record.

After merging the data frames, when we tally the emails, we can use `Name`. We find that the top email senders are

```
sort(table(emails$Name), decreasing = TRUE)[1:6]
```

```
     Hillary Clinton          Huma Abedin         Cheryl Mills
                1993                 1437                 1318
       Jake Sullivan    Sidney Blumenthal        Lauren Jiloty
                 871                  374                  341
```

Now, all of the emails corresponding to Jacob J Sullivan are counted together and appear under the label 'Jake Sullivan' and similarly for Sidney Blumenthal, etc.

What happens when we try to merge these two data frames and specify `all.y = TRUE` rather than `all.x = TRUE`? Or, what if we include both arguments, i.e., `all.x = TRUE, all.y = TRUE`? Try it and examine the dimension of the resulting data frames to convince yourself of how the merge is working. Be careful not to assign the merged file to `email` because we can get unwanted records. For example, we can have a record where all of the variables except `Name` are `NA`. Who are these people? The `persons` data frame contains identifiers and names of people who sent or received emails so anyone who received an email from Clinton but did not send an email to her will have a record added to the resulting data frame. All of the variables in this record (except `Name`) are missing.

Let's take our investigation one step further and consider the senders and recipients of email. We are interested in who sends and who receives emails and how often any two people exchange email. In other words, we're interested in the connections between people. The file *EmailReceivers.csv* contains the recipients of each email. Let's read in this file and examine it

```
recipients = read_csv("EmailReceivers.csv", col_names = TRUE)
```

We can examine a few records in our file and find emails with multiple recipients. For example,

```
recipients[16:25, ]
```

```
      Id EmailId PersonId
   (int)   (int)    (int)
1     16      15       80
2     17      16       80
3     18      17       80
4     19      17       32
5     20      17      229
6     21      17      170
7     22      17       87
8     23      18       80
9     24      18       32
10    25      18      230
```

The variable `EmailId` holds the identifier for the email record in `emails`. Notice that there are 5 entries in the `recipients` data frame that have a value of 17 for `EmailID`. For these records, we have 5 different values for `PersonId`, including #80, #32, and #87 whom we already know from our previous investigation are Clinton, Mills, and Sullivan, respectively. We can examine the `MetadataTo` and find that it shows only `H` for Clinton, but the `ExtractedTo` variable has the string:

```
H; Mills, Cheryl D; Sullivan, Jacob J; Nuiand, Victoria J;
 Reines, Philippe
```

These 5 recipients all appear in the string.

What kind data structure do we need in order to track who sent and received email to whom? What if we take the `recipients` data frame and augment it with the sender information? We can also add the name of the recipient, just as we added the name of the sender to `email`. We want to merge `recipients` and `emails`, which we can think of as senders. Let's start by creating a `senders` data frame that contains only the information we need in the merge. This is, the email identifier and the sender's identifier and name, we can do this with

```
senders = emails[ , c("Id", "FrId", "Name")]
```

Now `senders` is a 3-column data frame. Recall that `recipients` has 3 variables, `Id`, `EmailId`, and `PersonId`. To avoid confusion, let's rename `PersonId` to `ToId` and drop `Id`. We can do this with

```
names(recipients)[3] = "ToId"
recipients = recipients[ , -1]
```

We can add the person's name to this file with a merge, similar to the merge we already performed between `emails` and `persons`, i.e.,

```
recipients = merge(recipients, persons,
                   by.x = "ToId", by.y = "Id", all.x = TRUE)
```

Again, we have 2 data frames at different levels of granularity. The `recipients` is at the level of email recipient and `senders` (and `emails`) is at the level of email message. Since an email message can have multiple recipients and only one sender, we have more rows in `recipients` than in `senders`. Again, we want to merge the data in such a way as to keep all records in `recipients` and add to each record the sender ID and name. We can do this with

```
ToFrs = merge(recipients, senders,
              by.x = "EmailId", by.y = "Id", all.x = TRUE,
              suffixes = c("To", "Fr"))
```

We added one more argument to the call to merge(), i.e., `suffixes = c("To", "Fr")`. This argument is used to differentiate between variables with the same name in the 2 files. We have `Name` in both files, and after the merge, we have `Name.To` for the recipient's name and `Name.Fr` for the sender's name.

Now that our data are merged into 1 data frame, we can tally the communications between the people, i.e.,

```
emailCts = with(ToFrs, table(NameFr, NameTo))
```

It's difficult to study these data because `emailCts` has more than 100 rows and columns. Researchers developed a 2-dimensional line plot called BioFabric [1] where individuals are represented by horizontal lines and vertical lines represent exchanges between these individuals (an *R* implementation is available in the `RBioFabric` package [6]). Figure 2.15 is an example of this type of plot. What we can see here is that Clinton exchanges email with just about everyone (which makes sense since it is her email that we are studying), and her staff seem to be on email exchanges with smaller subgroups of people. Later, in [?] we consider these and other advanced visualizations. We include it here to hint at the possibilities in designing specialized visualizations for particular data structures.

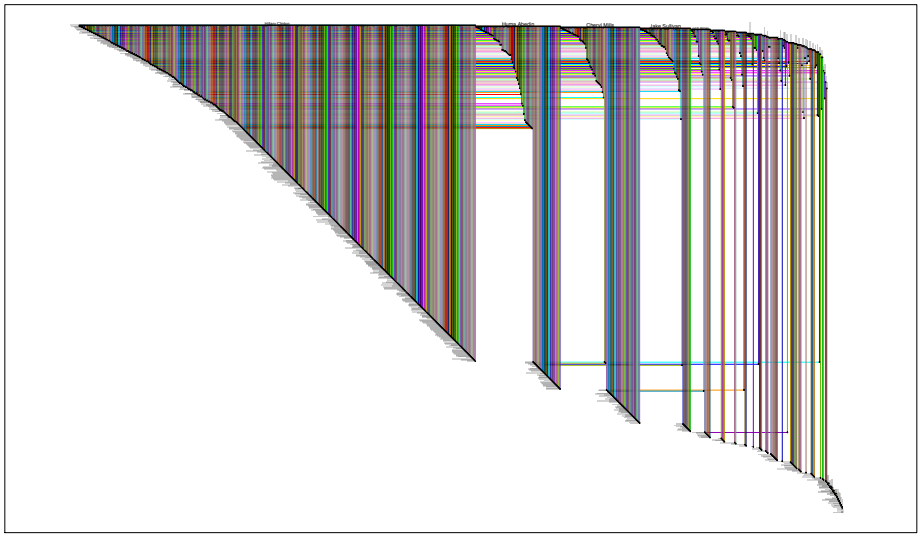


Figure 2.15: Clinton Emails Sender and Recipients. *This plot was designed to represent complex graphs. A node in the graph appears as a horizontal line segment, and edges between nodes are represented as vertical line segments that terminate at the two rows associated with the endpoint nodes.*

The `dplyr` package offers the functionality of merge() that employs the terminology common to relational databases. For example, the full_join() function finds the union of the two data frames, i.e., it works like merge() when both *all.x* and *all.y* are `TRUE`, and inner_join() is equivalent to them both being `FALSE`. Similarly, when only one of these arguments is `TRUE`, then we are performing a left_join() (`all.x = TRUE`). The `dplyr` package also offers other ways to join 2 data frames together. We examine how to merge tables in relational databases in Chapter 9.

2.7 Exploratory Data Analysis

We have provided examples of exploratory data analysis (EDA) throughout this chapter as we have read and cleaned various data sets. This section contains a more comprehensive treatment of EDA. EDA is like detective work. We aim to have an open mind about what me

might uncover and a willingness to find something surprising that points us in a direction for analysis that we had not previously considered. We examine the structure in the data in an iterative fashion; as we uncover aspects of our data, this can lead us to re-examine our understanding of the data and continue in our exploration. This approach typically relies on plotting our data in multiple ways, e.g., changing the scale to zoom in or out, transforming a variable to straighten or flatten, and consider a different type of plot to see the data from another view point. With EDA we try to remain flexible in the techniques that we use to examine the data. According to Tukey, the founder of EDA, "exploratory data analysis is actively incisive rather than passively descriptive, with real emphasis on the discovery of the unexpected." He further explains that "'Exploratory data analysis' is an attitude, a state of flexibility, a willingness to look for those things that we believe are not there, as well as those we believe to be there." Importantly, we also consider the manner in which the data were collected and how this can impact the conclusions drawn from our explorations. In the next sections, we explore more deeply the traffic data, country statistics, and emergency room survey results as examples of how we might carry out EDA.

2.7.1 Exploring Traffic on California Freeways

Recall from Q.2-1 (page 52) that we have an extract of traffic data from PeMS (the California freeway Performance Measurement System). These data consist of 5-minute summaries of the number of cars (flow) passing over a loop detector and the proportion of time a car was above the loop (occupancy). These 5-minute summaries are measured over 6 days for one set of loop detectors on 3-lanes of a freeway. In Section 2.2.1.1 we read the data into a data frame with 7 variables: flow and occupancy for the 3 lanes of traffic, plus time. We also changed the type of `time` from `character` to `POSIXct` so that we can use it in plotting. In Section 2.5.1 we re-shaped this data frame by stacking the measurements of flow for the 3 lanes into one variable (`flow`) and the 3 sets of occupancy values into one variable (`occ`). We also created a new variable (`lane`) so that we can distinguish which measurements came from which lane. With this format, we separated the information about the lane in which the traffic is measured from the variable name and created a variable that we can use more easily in analysis (see Figure 2.9). The `trafficLong` data frame contains the results from this data manipulation. It consists of 4 variables, `time`, `lane`, `flow`, and `occ`. As we carried out these actions on the data frame, we made several plots. In this section, we continue to examine the traffic data, but our focus is on exploration rather than data validation.

The main variables of interest are `flow` and `occ` because we are interested in `time` and `lane` only to help explain the relationship between flow and occupancy. As a first step, we examine the distribution of flow and of occupancy. Occupancy can be viewed as a measure of congestion. Since it is a numeric variable we can summarize it with

```
summary(trafficLong$occ)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  0.0183  0.0466  0.0579  0.0680  0.5100
```

The mean is about 25% larger than the median indicating that the distribution is skewed right. To get a better sense of the shape of the distribution of occupancy values, we examine a density curve. The density curve is a smooth version of a histogram (see the plot on the left of Figure 2.16). We create this curve with

```
ggplot(data = trafficLong) +
  geom_density(mapping = aes(x = occ)) +
  labs(x = "Occupancy")
```

By examining a density curve, we can glean additional features about a distribution that are not from in the summary statistics. From a density curve we can see which values are more common (high density) or rare. Most noticeably, the density curve for occupancy shows a bimodal distribution with two sharp peaks at about 0.02 and 0.06. We also see that the distribution has a long right tail when occupancy reaches 0.20 – 0.40. To get a better view of the main portion of the distribution, we zoom in to the region from 0 to 0.15 and shrink the bandwidth, which controls the amount of smoothness of the curve, so it follows the data more closely. The resulting curve is displayed on the right in Figure 2.16. There the 2nd mode appears to contain 3 small peaks, but this may be simply an artifact of a small bandwidth, which tends to yield curves that follow the data values too closely. However, since there are 3 bumps, it raises the question of how occupancy varies with lane.

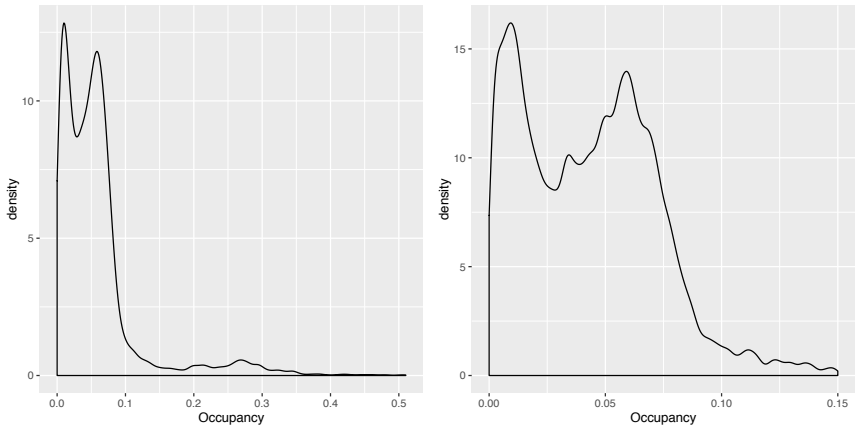


Figure 2.16: Distribution of Occupancy. *The density curve on the left shows a bimodal distribution for occupancy. The first mode is more peaked and the distribution has a long right tail. The plot on the right shows the distribution of occupancy over the range from 0 to 0.15. Here, the peakedness of the 1st mode and the larger spread in the 2nd mode are more apparent.*

In order to tease out the contributions from the 3 lanes to the distribution of occupancy, we create 3 density curves, one for each lane, and overlay them on the same plotting region, e.g.,

```
ggplot(data = trafficLong) +
  geom_density(mapping = aes(x = occ, fill = lane, color = lane),
               alpha = 0.3) +
  labs(x = "Occupancy") + xlim(c(0, 0.15))
```

This plot appears in Figure 2.17. We can see that each lane has a bimodal distribution, but the locations and peakedness of the modes differ. The middle lane tends to have a higher occupancy than the other lanes; we see this in the smaller first peak and in the location of the second peak at a higher occupancy than the other lanes. Additionally, the left lane appears to be more skewed than the other lanes with its large right tail. Further, the two modes in the right lane are closer together and the distribution is less spread.

Let's now examine the distribution of flow. Since the distribution of occupancy varied with the freeway lane, we examine flow as a function of lane too. The lefthand plot in Figure 2.18 displays side-by-side box plots of flow for the 3 lanes of traffic. These box and

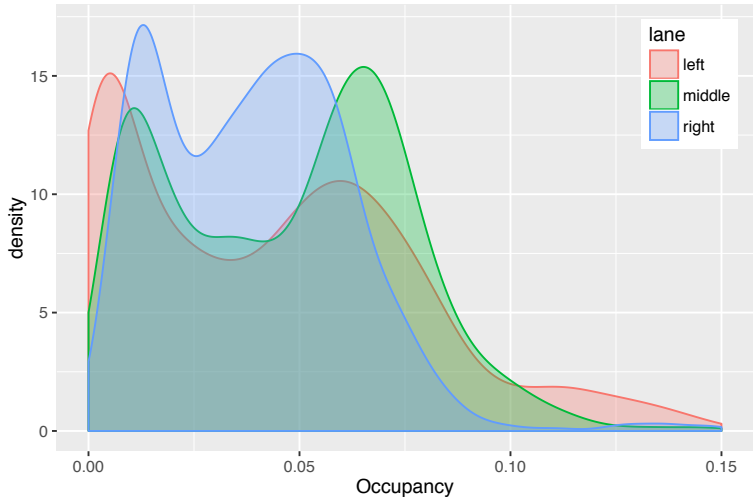


Figure 2.17: Distribution of Occupancy for 3 Lanes of Traffic. *This plot shows the distribution of occupancy separately for the 3 lanes of traffic. The distribution varies with lane where each lane exhibits 2 levels of occupancy with a sharper initial peak and a broader 2nd peak. Additionally, all 3 density curves have long right tailss.*

whisker plots provide a visual display of the quantiles of the data. The ends of the box mark the upper and lower quartiles, the line through the interior locates the median, and whiskers are drawn from the box to the largest data value within 1.5 of the inner quartile range (IQR) beyond the upper (or lower) quartile. Values beyond this are denoted individually. These plots reveal more information about a distribution than the summary statistics we computed earlier, but they are not typically as informative as a density curve or histogram because they cannot, for example, show modes and gaps. We do see that the flow in the right lane tends to be smaller than the flow in the other lanes as at least 3/4 of the values fall below the medians of the left and middle lanes. It also appears that the left lane has a long right tail and appears more skew than the other lanes. The violin plot on the righthand side of Figure 2.18 is similar to the boxplot except that a density curve of the data (and its mirror image) is plotted along each vertical line. The violin plot reveals the bimodal nature of the distribution of flow. We make these two sets of plots with

```
boxFlow = ggplot(data = trafficLong) +
  geom_boxplot(aes(x = lane, y = flow)) +
  labs(x = "Lane", y = "Flow")
vioFlow = ggplot(data = trafficLong) +
  geom_violin(aes(x = lane, y = flow), bw = 5) +
  ylim(c(0, 200)) + labs(x = "Lane", y = "Flow")
grid.arrange(boxFlow, vioFlow, ncol = 2)
```

These density curves, box plots, and violin plots do not address questions about how flow in the lanes vary together, how flow varies over time, and how flow (which is considered a measure of throughput) and occupancy (regarded as a measure of congestion) vary together. In the first case, we can, for example, make a scatter plot of flow in the left and right lanes. We made this plot earlier (see Figure 2.8). In that plot, a point corresponds to a particular 5-minute time interval so there are 1740 points plotted. We saw that flow in the left and

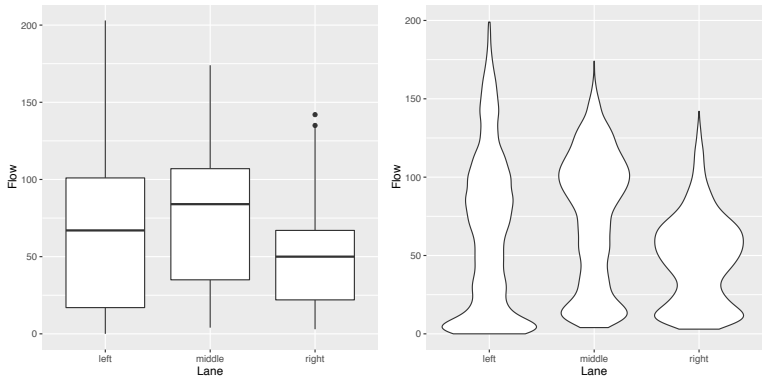


Figure 2.18: Distribution of Flow for 3 Lanes of Traffic. *The box plots on the left and the violin plots on the right compare flow for 3 lanes of traffic. We can see the bimodal nature of the distributions in the violin plot.*

right lanes are correlated in the sense that when flow is low in one lane then it tends to be low in the others. However, we also can see that flow in these two lanes is roughly linear with slope near 1 for flow under 50, and after that, the flow in the left lane increases more quickly than the right, and the variability also increases. This change is connected with the locations of the second modes in these distributions. Also the longer right tail of flow in the left lane is evident in the increase in variability at higher flow.

To understand how flow changes in time, we want to plot flow against time. We typically place time on the x-axis and make a line plot, e.g., we connect the flow measurement at one time to the measurement 5 minutes later and so on. Given the differences we have already observed between the behavior of flow across lanes, we super-pose the line plots of flow against time for each lane. We made this plot in Figure 2.10. As mentioned earlier, we see the ordering of flow between the 3 lanes change with throughput. In heavy traffic, the left lane has the greatest flow and the right lane is lowest, but immediately after a peak in flow, the left lane's flow dips sooner and faster to levels below the middle lane. Again, we can connect our observations to the location of the modes of the distributions of flow for the 3 lanes.

In order to see how throughput depends on congestion, we can plot one against the other for each of the 3 lanes. We do this with

```
ggplot(data = trafficLong) +
  geom_point(mapping = aes(x = occ, y = flow), alpha = 0.2) +
  facet_grid(lane ~ .) +
  labs(x = "Occupancy", y = "Flow")
```

Here we juxtapose the scatter plots to avoid over plotting and to make it easier for us to compare the relationship between flow and occupancy. We see the same pattern of points in each plot, but there are also important differences. The pattern shows that when there are few cars on the road, flow is small and so is congestion. The linear collection of points indicates that adding more cars increases congestion and flow also increases. However, at some point this linear relationship breaks down and as congestion increases traffic slows and flow decreases (i.e., we have a traffic jam). Two key differences between these 3 scatter plots are the slope of the line and the place where the breakdown occurs. The relationship between flow and occupancy breaks down soonest in the right lane (in terms of level of

congestion) then in the middle and last in the left lane. The pre-breakdown slope is smaller in the right than the middle and left lanes, which appear to have about the same slope. If we work with the simple assumption that all vehicles have the same length, then this slope is proportional to velocity.

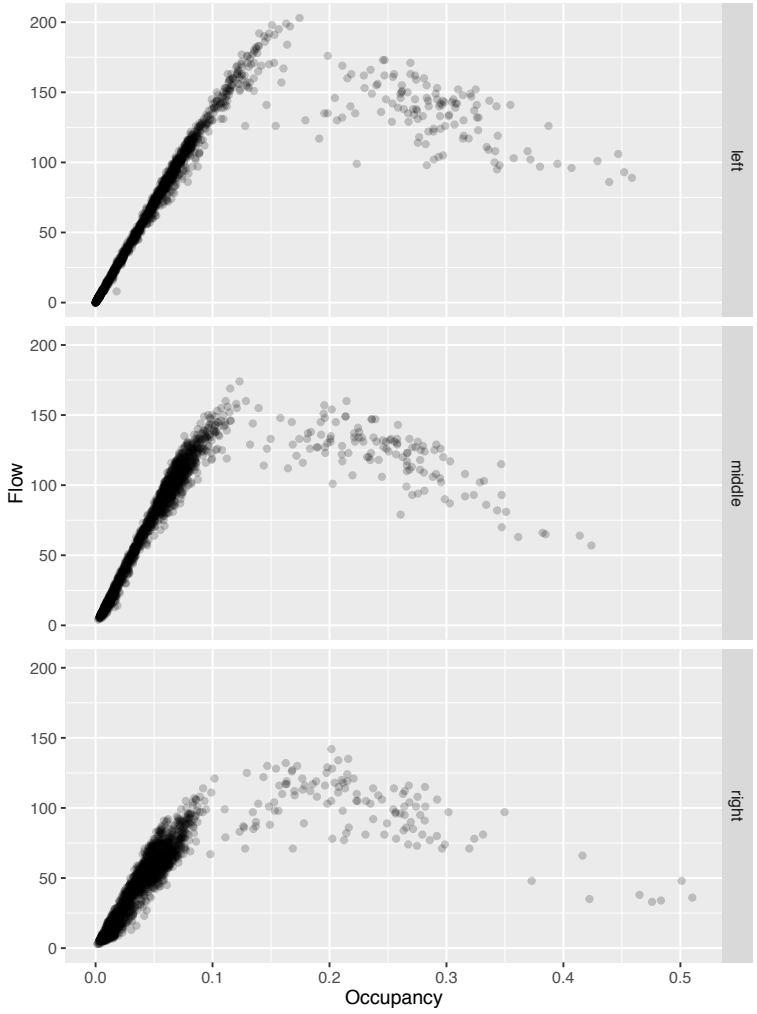


Figure 2.19: Flow vs. Occupancy for 3 Lanes of Traffic. *These 3 juxtaposed scatter plots display flow and occupancy measurements in 5-minute intervals for 3 lanes of traffic. The linear relationship between flow and occupancy breaks down at different occupancies for the 3 lanes.*

These exploratory plots have revealed many insights into the data. For example we have seen that the lanes appear to behave similarly in general but somewhat differently in specifics, and that the linear relationship between flow and occupancy breaks down. These insights help us in our more formal analysis. For example, we probably want to model the relationship between flow and occupancy with a smooth curve or a piecewise linear curve, and we want to take the lane into account in estimates for the breakdown point.

Finally, with this traffic example, we have collected all measurements at one location in

a particular week. If we want to generalize the relationships we have discovered between flow and occupancy or flow and day of the week, then we need to consider how representative is this location of the California freeway system and how similar are the other weeks of the year to this particular week in March.

2.7.2 Exploring Country Statistics

The World Bank data provide summary statistics for nearly all countries in the world. With these data we can examine relationships between these various statistics. As an example, we explore the relationship between health care expenditures per capita and life expectancy at birth. Both variables are quantitative. We earlier examined the distribution of life expectancy (see Figure 2.13), where we found a skew left distribution. That is, most countries have an average life expectancy of 70 to 80 years, but a substantial number of countries have low life expectancies that range from about 40 to 65 years.

A histogram of health care expenditures per capita for these 200+ countries appears in Figure 2.20. The dark red line segments along the bottom of the plot show the individual values for the countries. The color is somewhat transparent so that when segments overlap they show a darker color and denser regions display thicker clumps. These red 'threads' create a rug plot. Again, we have a skewed distribution, but this time it's right skewed. The vast majority of countries spend less than $500 per person. This distribution has some unusually large values; it appears that one country spends about $5000, another about $4000, and a handful of countries spend between $2000 and $3000 per capita. We make this histogram with

```
ggplot(data = wbSub, aes(x = healthPC)) +
    geom_histogram(aes(y = ..density..), alpha = 0.4) +
    geom_rug(col="darkred", alpha = .4) +
    labs(x = "Health Expenditure per Capita")
```

We are interested in relationships between variables and these two histograms do not reveal this connection. We imagine that health care expenditure (per capita) is positively correlated with a longer life expectancy, but we can't glean this information from the two univariate distributions. We make a scatter plot where each country corresponds to one point so we can see how life expectancy and health care expenditures vary together. We create the scatter plot in Figure 2.21 with

```
ggplot(data = wbSub) +
  geom_point(aes(x = healthPC, y = lifeExp)) +
  labs(y = "Life Expectancy",
       x = "Health Care Expenditure per Capita")
```

In this figure, we see a highly curved relationship with the countries that spend less than $500 per capita having a great range in life expectancy. Also, the curve flattens out over a wide range of high-expenditure countries.

If we can transform the variables and straighten the curvature, then this helps us better ascertain the relationship between the 2 variables. In this case, we take the logarithm of health care spending to bring in the large values and make the distribution more symmetric.

```
ggplot(data = wbSub) +
  geom_point(aes(x = healthPC, y = lifeExp)) +
  scale_x_log10() +
  labs(y = "Life Expectancy",
       x = "Health Care Expenditure per Capita")
```

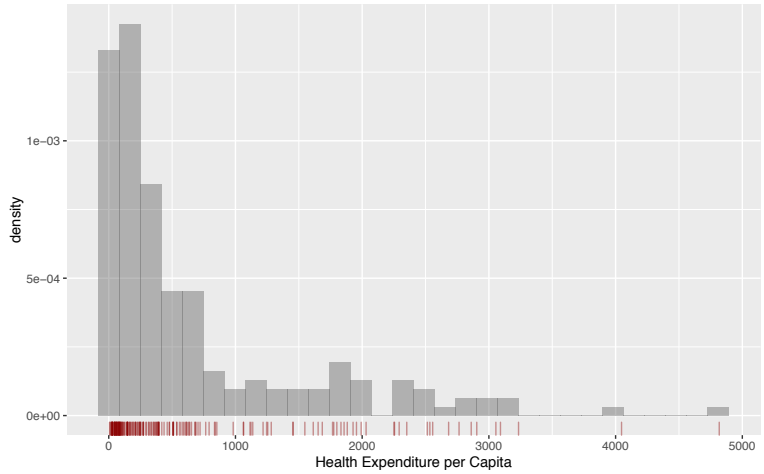


Figure 2.20: Histogram of Health Expenditure per Capita. *This histogram and accompanying rug plot show the distribution of average health expenditure per capita in 2000 for countries, as reported by the World Bank. The distribution is highly skewed to the right with a few countries spending over $2000 per capita, and there appears to be two outliers at $4000 and $5000.*

Now we see in Figure 2.22 a linear association between life expectancy and the log of health care expenditures. The correlation coefficient is a measure of linear association and we find a strong linear association of about 0.78 with

```
with(wbSub, cor(log(healthPC), lifeExp, use = "complete.obs"))
```

```
[1] 0.7836
```

Although this relationship is roughly linear, the variability in life expectancy for low-expenditure countries is greater than for high-expenditure countries. Note, we can compute the correlation even when the variables do not display a linear association. For example, the correlation between the life expectancy and the untransformed health expenditure is 0.62, despite our original scatter plot showing a clear lack of a linear association. It's a good idea to not rely on correlation alone to assess linear association.

The scatter plots we have examined show all countries, which have a tremendous range in prosperity. When we zoom in to examine a more homogeneous subset of countries, we can get a closer (and possibly better) view of the relationship between life expectancy and health care expenditure. For example, let's examine the countries with the highest gross domestic product (GDP). In this case, there are few enough points that we can label them with country codes (see Figure 2.23). We create this scatter plot with

```
topGDP = order(wbSub$gdp, decreasing = TRUE)[1:50]

ggplot(data = wbSub[topGDP, ]) +
  geom_point(aes(x = healthPC, y = lifeExp)) +
  geom_text(aes(x = healthPC, y = lifeExp, label = ctry),
            size = 3) + scale_x_log10() +
  labs(y = "Life Expectancy",
       x = "Health Care Expenditure per Capita")
```

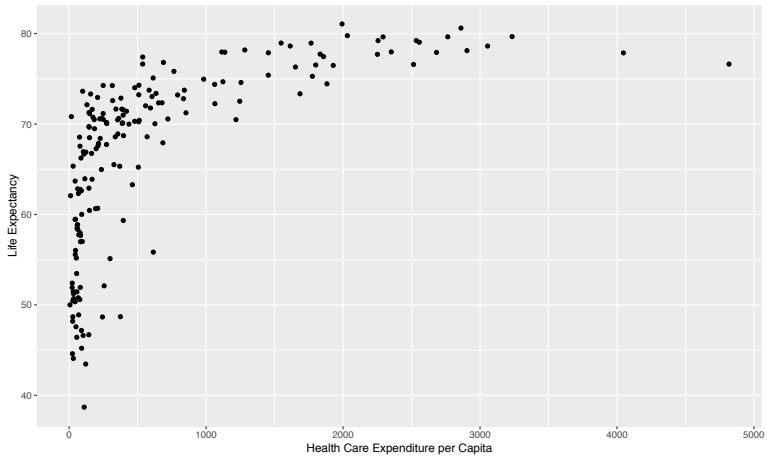


Figure 2.21: Life Expectancy vs. Health Care Expenditures per Capita. *This scatter plot displays the average life expectancy and health care expenditure per capita in 2000 for countries reported by the World Bank. The pattern of points is highly curved and expenditures range over several magnitudes.*

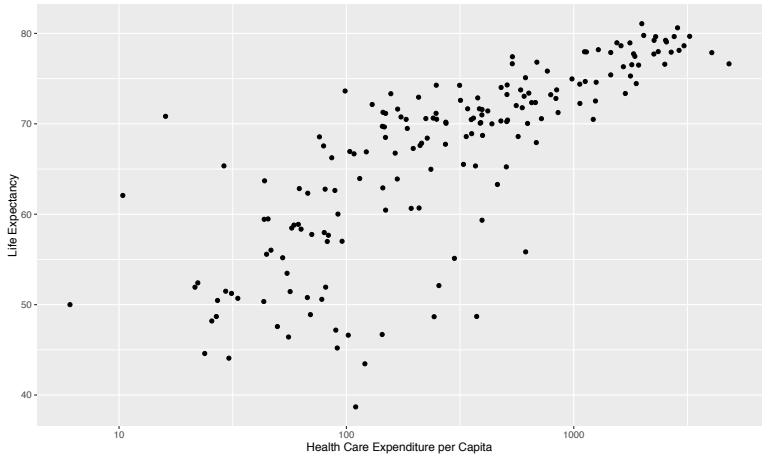


Figure 2.22: Life Expectancy vs. Health Care Expenditures per Capita (Log Scale). *In contrast to Figure 2.21, this scatter plot shows health care expenditure per capita on a log scale. The relationship appears roughly linear with a large variability in life expectancy for countries spending under $1000.*

The correlation between life expectancy and health expenditures for this subset of data remains high,

```
with(wbSub[topGDP, ], cor(log(healthPC), lifeExp,
                          use = "complete.obs"))
```

```
[1] 0.8029
```

In some situations, the correlation drops because the subset has a smaller range of values;

this is not the case here because the strong curvature and high variability are gone when we exclude the low-GDP countries.

This scatter plot has several interesting features. South Africa (ZAF) has a much lower life expectancy compared to all other top countries, especially those with similar expenditures per capita. The US spends 10 times more than any other country on health care but is outstripped in terms of life expectancy by 20 countries or more.

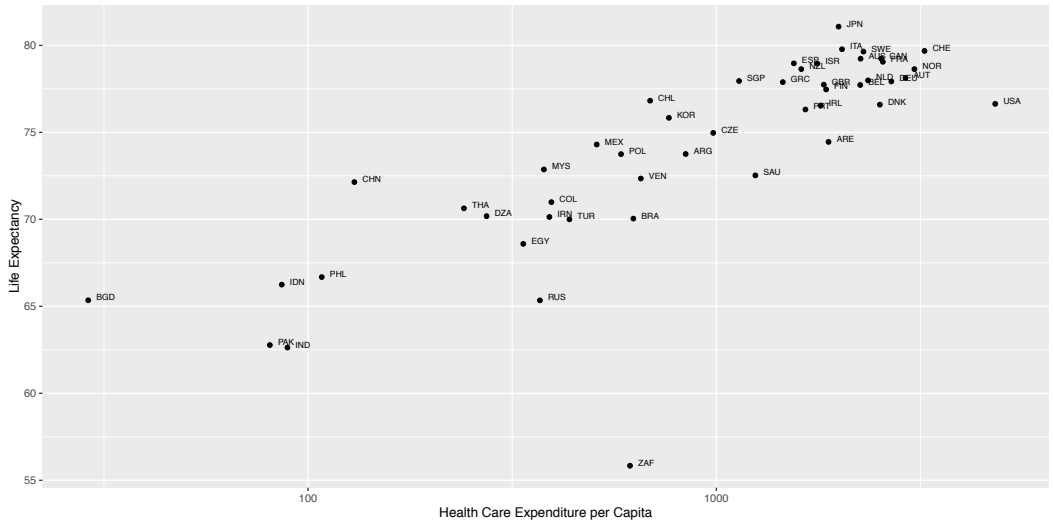


Figure 2.23: Life Expectancy vs. Health Care Expenditures per Person (Top Countries). *This scatter plot zooms in on those countries with GDP in the top 50 of all countries in the World Bank (see Figure 2.22 for plot with all 214 countries). Points are identified with their country ID. South Africa (ZAF) has unusually low life expectancy given its expenditures.*

Despite these interesting findings, we must be careful when interpreting them for reasons that are related to the kind of data. The relationships observed are based on rates and averages, e.g., the infant mortality rate in the country and the average years of education. Measurements such as these tend to overstate the strength of an association. For example, if we have data on individual's income and education, the variability between these tends to increase and as a result the correlation tends to be lower than for the group averages. Also these data are observational, not experimental so although we expect that spending more on health care leads to an increase in life expectancy, we can imagine situations where this wouldn't occur. For example, in the US many people go without adequate health care so increasing expenditures may not lead to increased life expectancy if funding for the underserved does not change. Lastly, if we model this and other relationships, we must bear in mind that the variability in average life expectancy is not constant across expenditures.

2.7.3 Exploring Emergency Room Visits due to Drug Abuse

The DAWN survey data (Q.2-2 (page 52)) studies carefully selected emergency room visits. A probability scheme was used to choose the visits for inclusion in the survey. In particular, all hospitals with 24-hour emergency departments in the US were divided into regional groups and within each group a sample of hospitals was chosen at random. The chance a hospital was selected to participate depended on the size of the hospital and the region. Then, for each hospital selected, a random sample of emergency department medical records

was taken from those with a drug-related problem. More information may be found at `www.samhsa.gov/data/`. Although a probability sample, every possible subset of ER visits is not equally likely because of the complex sampling scheme. This means that we need to incorporate each record's probability of selection into the analysis. As an example, if the proportion of ER visits in Los Angeles were sampled at twice the rate of those in other regions, then LA is over-represented in the sample and we need to correspondingly down weight their responses. Before setting up the survey design for our analysis, we show a simple example of why it's important to consider how the subjects are selected for the survey.

As mentioned in Section 2.3.2, most, if not all, of the variables in this survey are categorical, some of which are ordered. This means that we summarize and plot the data differently from the previous examples. Let's begin by comparing the reason for the ER visit for men and women. The variable `allabuse` is binary (dichotomous) and simply records whether or not the purpose of the visit was entirely drug-abuse related. Since this information is qualitative, we convert `allabuse` into a factor with

```
dawn$allabuse = factor(dawn$allabuse, labels = c("no", "yes"))
```

We make a 2-way table of `allabuse` and `sex` with

```
totRaw = table(dawn$sex)
100 * table(dawn$sex, dawn$allabuse) / as.numeric(totRow)
```

```
            no    yes
  male    31.98  68.02
  female  53.99  46.01
```

Note that we normalized the proportions in the 2-way table by row (which we calculated and saved in `totRow`) so the row proportions add to 1. This way, we control for an unequal number of males and females in the sample. These data show that roughly 68% of the male and 46% of the female visits to the ER for drug-abuse related reasons were solely for drug abuse. However, when we take the probability of selection into account, these numbers change to 57% and 40%, respectively. That is, both percentages decrease and they are closer together.

The DAWN data includes information about the survey design and weights to adjust for the unequal representation of ER visits. We do not go into the details of the sampling scheme, but note that we can use the `survey` package to describe the design and incorporate the design into tallies and plots. We set up the design with

```
library(survey)
dawndsgn = svydesign(id = ~ strata + psu, weights = ~ wt,
                     data = dawn)
```

Then, we can compute the tables with svytable(), which uses the design to adjust the tallies, e.g.,

```
totSex = svytable(~ sex, dawndsgn, Ntotal = 100)
totSexAbuse = svytable(~ sex + allabuse, dawndsgn, Ntotal = 100)
percentWinSex= 100 * totSexAbuse / as.numeric(totSex)
percentWinSex
```

```
        allabuse
sex          no   yes
  male    42.40 57.60
  female 59.58 40.42
```

A statistical graph of a table can be much easier for us to make comparisons. We don't get bogged down in the particular digits and instead get a general understanding of the differences between groups. For example, in the bar plot of Figure 2.24, the heights of the bars match the percentages in the table above. These are color-coded to make it easier to distinguish between males and females, and the placement of the two bars for 'not all abuse' are close to each other to further facilitate this comparison. We make this plot with

```
barplot(percentWinSex, beside = TRUE,
        col = c("lightblue", "mistyrose"),
        ylab = "Percentage",
        xlab = "ER Visit Solely for Substance Abuse")
legend("topleft", fill = c("lightblue", "mistyrose"),
       legend = rownames(percentWinSex), bty = "n")
```

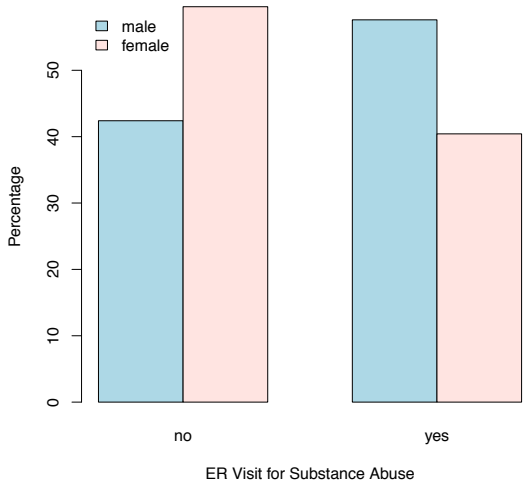


Figure 2.24: Comparison of Reason for ER Visit. *This bar plot compares males and females according to whether the ER visit was entirely abuse related (yes) or due to a combination of reasons including drug abuse (no).*

The mosaic plot offers another kind of visual comparison. We create the mosaic plot on the left in Figure 2.25 with

```
plot(svytable(~ sex + allabuse, dawndsgn), main = "")
```

This plot makes clear the comparison between males and females. To create it, a rectangular region is divided vertically according to the proportion of males and females in the sample. Then, each new rectangle is divided horizontally according to the proportion of yeses and noes within that subgroup. As a comparison, the mosaic plot on the right in Figure 2.25 ignores the survey design. We make this with

```
with(dawn, mosaicplot(table(sex, allabuse), main = "", ylab = ""))
```

These plots make it apparent that we need to include the design of the survey in our analysis or we risk creating an incorrect picture of the results.

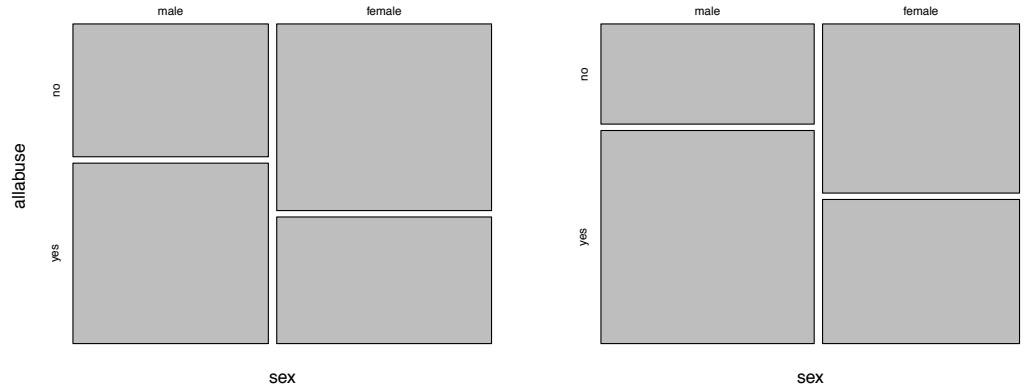


Figure 2.25: ER Visit Due Solely to Substance Abuse by Sex. *The mosaic plots on the left compares males and females according to whether the ER visit was entirely abuse related (yes) or due to a combination of reasons including drug abuse (no). The plot on the right was made without taking into account the sampling design. The unweighted mosaic plot overstates the yes-category for both sexes.*

Another variable in the data provides the reason for the ER visit. This can be one of the following: suicide attempt, seeking detox, alcohol only (for those under 21), adverse reaction, overmedication, malicious poisoning, accidental ingestion, and other. Again, to examine the relationship between sex and the reason for the visit, we can make a bar plot or mosaic plot. Instead we use a dot chart, which is a simplification of the bar plot. Since there is no meaning attributed to the width of the bars in a bar plot, we can shrink the bar and simply mark its length with a dot along a line (see Figure 2.26). We organize this dot chart to make the comparisons between males and females for each category of reason by computing the percentage for a reason within males (and similarly for females) and plotting the male and female figures side by side for each reason. We do this with

```
totSexType = svytable(~ sex + type, dawndsgn, Ntotal = 100)
percentTypeWinSex= 100 * totSexType / as.numeric(totSex)

dotchart(percentTypeWinSex,
         xlab = "Percent of ER Visit Type within Sex")
```

We see in Figure 2.26 that the distribution of reasons for males and females are very similar with 2 striking exceptions – adverse reactions and the 'other' category. A greater proportion of females visited the ER due to adverse reactions, and a greater proportion of the males visited the ER for a reason not captured by the 7 possibilities given. This large fraction of males in the other category (40%) indicates a problem with the coding of this variable that the survey researchers may want to further investigate.

We can also create line plots for these data (see Figure 2.27). Like the dot chart, the line plot represents a proportion with a point. However, these points are plotted one above or below the other for males and females for each reason, and the points for each sex are connected by line segments. We make this line chart with

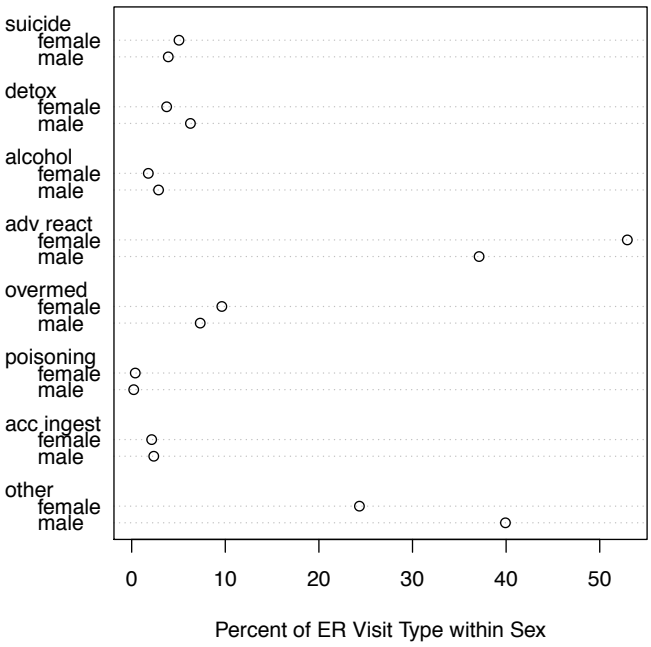


Figure 2.26: Reason for ER Visit by Sex. *This dot chart compares males and females according to the reason for visiting the ER (all visits are drug-abuse related). The percentages for males and females are quite close for all reasons except adverse reaction, where a greater proportion of females visit the ER for this reason, and the 'other' category where a greater proportion of males visit for this reason.*

```
plot(percentTypeWinSex["male", ] ~ I(1:8), type = "l",
     ylim = c(0, 55),
     ylab = "Percentage of ER Visit Type w/in Sex",
     xlab = "Type of Abuse", axes = FALSE)
points(I(1:8), percentTypeWinSex["female", ], type = "l", lty = 2)
box()
axis(side = 2)
axis(side = 1, at = x, labels = colnames(cage))
legend("topleft", legend = rownames(cage),
       border = NULL, bty = "n", lty = c(1, 2))
```

This layout helps us see the male values together and in comparison to the female values. The differences between males and females for these two categories is quite striking, especially because they make up a large fraction of the responses.

We are also interested in exploring the 3-way relationship between sex, age, and the reason for the ER visit. Particularly, we might want to know whether the reason for the ER visit changes with age and whether we observe a difference between males and females for these age groups. To do this, we can create a mosaic plot with

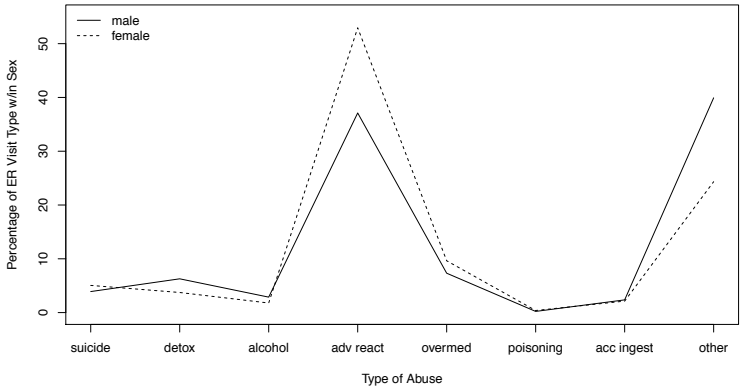


Figure 2.27: Interaction Plot for the Reason for ER Visit by Sex. *This interaction plot displays the same information that is in the dot chart in Figure 2.26. Here, the reasons are arranged along the x-axis rather than the y-axis and the values for each sex are connected by line segments.*

```
plot(svytable(~ sex + age + type, dawndsgn), main = "",
     ylab = "", xlab = "", las = 2)
```

In this case, the rectangular area is first divided vertically according to sex, then horizontally according to age within sex, and lastly, vertically within each sex-age region according to the type of visit. We see in Figure 2.28 how the proportion of ER visits due to an adverse reaction changes by age with it being the predominant reason for the young and old. Also striking is the large proportion of suicide attempts for teenage girls. There are many other interesting observations we can make from this mosaic plot. We also note that if we change the order of the variables in the formula `~ sex + age + type`, then the order of division of the regions changes and a very different looking plot appears. The order determines the sequence to use in slicing up the rectangles. Each rectangle is subdivided based on the proportion within the corresponding group. This order can be important for facilitating particular comparisons.

2.7.4 EDA Summary

In this section, we summarize the basic concepts of EDA and considerations in creating visualizations that we demonstrated in the previous examples. More details on how to construct informative statistical graphs appears in Chapter 3.

How were the data collected?
It's important to keep in mind where the data come from because this impacts whether and how we can generalize the findings of our exploration and analysis. One question to consider is: do the subjects under study form a census or a sample? And, if the data are a sample, how were the subjects selected for the sample? We also consider whether these data represent a snapshot in time, and if repeated measurements were made over time. We saw with the traffic data that we have only measurements at one location for 6 days so if we want to generalize from these data to traffic on California freeways, we must consider how representative is this location and time period. The World Bank data consists of averages and rates for all

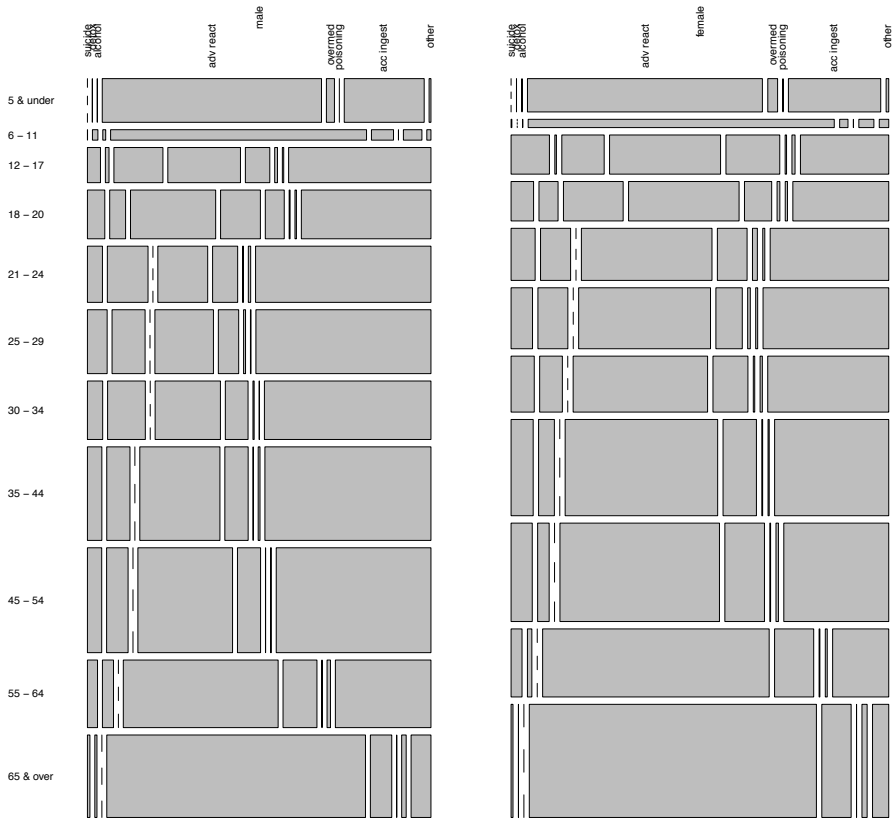


Figure 2.28: Type of Substance Abuse by Sex and Age. *This mosaic plot compares age and sex for different types of substance abuse. Males are on the left and females on the right. We see the type of abuse changes for different age groups.*

countries in a specific time period. Here, we must be careful not to overstate the strength of the relationships found when considering implications for individuals, assumptions of causality, and heterogeneity in variances. For the DAWN survey, a probability method was used to select the subjects (ER visits) for the survey and we must take care to include the design in our analysis so that relationships discovered are representative of the population.

What type of data are we exploring?

An important first step in making a statistical graph is to recognize the data type of the variable(s). We need to ascertain whether a measurement is quantitative or categorical because the approaches to viewing and summarizing quantitative information differs from qualitative. In the DAWN survey, we worked solely with qualitative variables, such as the sex of the patient and the reason for the ER visit. Additionally, age is qualitative in this study because it is reported in categories from '5 and under' to '65 and over'. These categories are naturally ordered, unlike with the categories for sex and reason for the ER visit so it may be tempting to treat this categorized age variable as quantitative. However, this can be problematic because the distance from one category to the next is not the same number of years and a person in, say, category 6 is not twice as old as someone in category 3. Alternatively, the sex variable in the World Bank survey is quantitative because it is the

proportion of males in the country, which ranges between 0 and 1. The measurements in the World Bank are generally quantitative as they are averages, e.g., health care expenditures per person, and rates, e.g., infant mortality per 1000 live births. Finally, the measurements of flow and occupancy over the loop detectors in the traffic data are quantitative measures, as is the time when a measurement was taken. (Here time is an 'interval' and not a 'ratio' variable because it makes sense to look at differences in time but not multiples.) Lastly, the lane in which the measurements were taken is qualitative.

What do we look for in a distribution?
To examine the distribution of values for a quantitative variable, we create a histogram or density curve. In histograms, the area of the bar represents the proportion of values that fall in that range. For this reason, the bins of a histogram need not be all the same width. (For density curves it's the area under the curve within the range of x values that represents the proportion). With a histogram or density curve, we get a sense of the following characteristics of the distribution: symmetry and skewness, number and location of modes (high frequency regions), length of tails, gaps where there are no observations, and unusually large or anomalous values. For qualitative data, the bar plot serves a similar role to the histogram in that we can visualize the 'popularity' or frequency of different categories. In a bar plot, the frequency of a category is represented by the height of the bar and the width carries no information about the distribution. For this reason, a dot chart is an alternative graphical display for qualitative data.

Summary statistics such as the mean and standard deviation (SD) or the median and interquartile range (IQR) give us an idea of the center of a quantitative variable's distribution and the spread of typical values. However, the density curve and histogram give us a more complete sense of the data distribution. The mean and SD are often useful summary statistics for symmetric distributions and the median and IQR are typically used with skewed distributions. Box and whisker plots provide a visual display of these quantile-based summaries. Box and whisker plots typically reveal more information about a distribution than the quartiles, but they are not as informative as a density curve or histogram because they cannot, for example, show modes and gaps.

Summary statistics for qualitative data amount to tables of categories with their frequencies or proportions, i.e., the same information conveyed in a bar plot and dot chart. The advantage of the plots is in the ease of making relative comparisons. For this reason, e.g., bars in a bar plot are often ordered according to their height, unless the ordering of the categories has some meaning, e.g., with education level.

Method of Comparison
We often want to make comparisons with our data. We may want to compare the distribution of a variable to an historical value or a benchmark of some kind; we can do this by adding a reference point or line to a plot (e.g., Figure 2.13). Alternatively, we may want to compare the relationship between two quantitative variables to a line with, say, slope 1, which we can do by simply adding the reference line to a scatter plot (e.g., Figure 2.8). Or, we may want to split our data into subgroups, according to the value of another variable and compare the resulting distributions through side-by-side box plots, violin plots (e.g., Figure 2.18), or super-posed density curves (e.g., Figure 2.17), side-by-side bar plots (e.g., Figure 2.24), or mosaic plots (e.g., Figure 2.25). The method of comparison can help provide context to a problem or insight into a distribution or relationship.

Looking for Relationships between Variables
Similar to the decision to make a histogram or a bar plot, we must consider data types when choosing a plot to visually explore the relationship between two variables. When both variables are quantitative, we can examine their relationship through a scatter plot. We

may want to consider transformations of the variables to straighten the relationship, or we may want to add a locally smoothed curve to the plot that shows the average value of the y variable for neighboring x values. Also, if one of the variables is time, then we typically place this variable on the x-axis and make a line plot that connects the y-values. With two qualitative variables, we make grouped bar plots, dot charts, and line plots, as well as mosaic plots. And in the case where we have one quantitative and one qualitative variable, we make side-by-side box plots and overlaid density curves.

Of course, these are general guidelines for the selection of a plot type, not strict rules. There can be situations where a particular graph is appropriate even though it doesn't fit into these guidelines. For example, suppose that we want to look at the age distribution in the DAWN survey. Although age is a categorical variable in this survey, a histogram is a natural way to summarize an age distribution. In this case, if we align the bins of the histogram with the categories used for age, then we can make a histogram from these categorical data. As another example, flow and time are numeric quantities, but it is reasonable to compare the distribution of flow across days of the week. To do this, we can make box plots of flow for each day of the week, essentially converting time into a categorical variable.

2.8 Summary

Tables with rows that correspond to observations and columns to various measurements on each observation are a convenient and common representation for data. Moreover, tabular data are often stored in plain text files where each observation appears on its own line in the file and a delimiter separates the values for the observations or the values appear in the same fixed location for all lines in the file.

The data frame is a natural structure for tabular data as the vectors in the data frame correspond to variables and these can be of different data types.

It is a simple process to read tables into a data frame. However, things can go wrong and data validation and cleaning are important steps when reading data into *R*. Simple data summaries and plots are often helpful in this validation step and can serve as the beginning of exploratory data analysis. EDA is a key part of the data analysis process. In addition to validating data, it serves to examine the properties of the variables and relationship between variables.

Facility with computations help us in this process. For example understanding data types and impact of EDA lead to data type conversions and decisions as to the best data structure to use, e.g., data frame, matrix, array, etc. Understanding the concept of a table helps us in determining the best shape for our data table and gives us skills for reshaping tables. Understanding formats for plain text files helps us choose the appropriate function for reading the data and providing the appropriate arguments so the data can be discerned.

We have not provided examples of the many versions of functions that can be used to read data into *R*. Instead, our focus is on understanding the computational and statistical aspects in determining how to read, clean, and organize the data. For more details on reading data see

EDA is like detective work where we actively seek clues about our data that can help direct us in later stages of analysis. It is important to not lose sight of where the data come from in our EDA and more generally. EDA is an important aspect of data analysis that tends to have limited development in statistics textbooks. Moreover, we learn the philosophy and goals by example. For more information on EDA see Tukey's original introduction to the subject [10]. See also [11] and [7] for more examples of EDA.

2.9 Functions for Reading and Exploring Data

This chapter introduced many of the functions available in *R* for working with data that are arranged in table-like formats. These are summarized below. A summary of the plotting functions appear at the end of Chapter 3.

read_delim() (in `readr`) Read a text file with values in a tabular arrangement and return a data frame with one row for each row in the table. The delimiter is specified with *delim*; common delimiters are white space (specified by `""`), the tab (`"\t"`), and the comma (`","`). To argument *col_names* is a logical to specify that the first line of the file contains column names (`TRUE` is the default); to skip the first n rows of the file (not include them in the data frame), set *skip*; specify the class of each vector in the data frame with *col_types* (`factor` is not accepted as a data type); and provide the values to be converted into `NA`s in a character vector to *na*.

read.table() Similar to read_delim(). The delimiter is specified with *sep* (default: `""`). To specify that the first line of the file contains column names, set *header* to `TRUE`; to skip the first n rows of the file (not include them in the data frame), set *skip*; specify the class of each vector in the data frame with *colClasses* or avoid the character vectors from automatic conversion into factors by setting *stringsAsFactors* to `FALSE`; and provide the values to be converted into `NA`s in *na.strings*.

readBin() Read a binary file from the connection specified in *con* and return a vector of type specified by *what*. Specify the maximum number of records to be read in with *n* and the the number of bytes per element with *size*. To specify that the data being read in is an unsigned integer, set *signed* to `FALSE`.

read_fwf() (in `readr`) Read in fixed width formatted data and return a data frame with one row for each row in the table. The columns positions are specified with *col_positions*, which takes in output from the functions fwf_empty(), fwf_widths(), or fwf_positions(). For fwf_positions(), provide the starting position, ending position, and optional column names with *start*, *end* and *col_names*, respectively. Other arguments match those of read_delim()

read.fwf() Similar to read_fwf(). The *widths* of the columns are specified as an integer vector. To specify that the first line of the file contains column names, set *header* to `TRUE`. Columns can be skipped by supplying a negative integer for a column width. Other arguments are similar to those of read.table().

read.dcf() Read in a Debian Control File (DCF) format and return a character matrix with one row for each record and one column for each field. One of the defining features of a DCF format is that each entry in the dataset is a `key: value` pair. The *fields* argument is used to specify fields to be read in. If there are multiple occurrences of a field, the default behavior is to read its last occurrence; to gather all occurrences, specify *all* to be `TRUE`.

with() Evaluate *R* expression on the supplied data object, usually a list or data frame. This is a convenience function to refer to elements in the data object without needing to refer to the data object itself in the expression, e.g., `with(dataframe, cor(x, y)` is equivalent to `cor(dataframe$x, dataframe$y)`.

2.10 Guided Practice

Saratoga
The file *Saratoga.txt* contains data that are a random sample of 1,728 homes taken from public records from the Saratoga County in New York. The data was collected by Candice Corvetti (Williams '07) and made available at http://community.amstat.org/stats101/home.

1. Examine the file *Saratoga.txt* to determine its format. Read the file into a data frame in *R* and name it `saratoga`. Examine the resulting data frame to determine its structure and whether the data have been read in properly.
2. Create a new factor variable called `fp` that indicates whether or not a house has a fireplace.

Weather
The file *weather.txt* contains daily temperature recordings for San Francisco in 2015. In the following exercises, we read the dataset into *R* and check that the data are properly read. This includes some data clean-up and some sanity checks.

1. Examine the file to determine the format and then read it into a data frame.
2. What are the names of the variables? Rename them to short informative names. (We suggest `month`, `day`, `low`, `high`, `nlow`, `nhigh`, `rlow`, `year.rl`, `rhigh`, `year.rh`, `precip`, `rprecip`, and `year.rp`.)
3. Explore the variables. What is the class of each variable? Do the values of each variable seem reasonable?
4. Why is `precip` not `numeric`? Make a table of the values in precipitation. The `T` stands for 'Trace'. Set it to 0 and convert to the `precip` variable to `numeric`.
5. Convert the month and day variables into a `POSIXct` formatted variable. Include the year in the format. Begin by creating a string of the year, month, and day, where each element is separated by a dash. Then use this string to create a `POSIXct` formatted date variable. The help file of strptime() is useful here.
6. On the same plot, make line plots of daily high and low temperatures and record high and low temperatures. Make sure to appropriately set the limits of the y-axis!

Olympics 2012
The file *olympics2012.txt* contains the country medal tallies and number of men and women competing for countries that participated in the Summer 2012 Olympics.

1. Examine the file to determine its format. Read the file into *R* as a character matrix. Make sure the data are read in properly.
2. Transform the matrix to a data frame. Convert the variables to `numeric` that should be.
3. Examine the relationship between the number of athletes competing and the number of medals. To do this, create a new variable which is the sum of the male and female athletes. Make a scatter plot of the number of medals won against the number of athletes. Describe the relationship.

4. Consider transformations to straighten the relationship between the number of athletes competing and the number of medals. Two common transformations for count data are the log and square root transformations. The variation in the counts tends to increase with the square root of the count. Compare the log and square root transformation of medal counts. Which does a better job of straightening the relationship?

5. We want to include data about the size and wealth of the countries in our analysis. Use the World Bank data and create a data frame with 2010 data on the countries. Follow the code from the chapter to read and reshape the World Bank data. Feel free to keep additional variables.

6. Merge these two data frames. Since we are only interested in countries that competed in Summer 2012 Olympics, we do not want countries in the World Bank that did not compete.

7. Examine the relationship between two variables in the World Bank data. Color the countries, i.e., the points in the scatter plot, by the number of medals won by the country, e.g., no medals, a few, etc. To use these colors, create a factor variable from the total number of medals using the cut() function. Consider transformations and normalizations, e.g., does it make sense to plot GDP or GDP per person?

Binary data
The file *data3* in the *digitsBinary* directory contains 1,000 binary images of handwritten digits for the number three.

1. Using the code in Section 2.4 as a guide, read *data3* into *R*. Reshape the data into an array of 1,000 28×28 matrices.

2. Explore the data by making an image plot like the one shown in Figure 2.7. This time try different values for the quantile. How does the plot change?

2.11 Exercises

Bibliography

[1] BioFabric. `http://www.biofabric.org/`, 2014.

[2] Henrik Bengtsson, Andy Jacobson, and Jason Reidy. `R.matlab`: Read and Write MAT Files and Call MATLAB from Within R. `http://cran.r-project.org/web/packages/R.matlab`, 2016. *R* package version 3.6.0.

[3] Caltrans. Caltrans Performance Measurement System. `http://pems.dot.ca.gov/`, 2014.

[4] Ben Hammer. Hillary Clinton's Emails. `https://www.kaggle.com/kaggle/hillary-clinton-emails`, 2016.

[5] Tim Hanrahan, Martin Burch, and Katie Marriner. *Search Hillary Clinton's Emails.* The Wall Street Journal, Mar 1, 2016.

[6] William Longabaugh. `RBioFabric`: BioFabric network visualization tool. `https://github.com/wjrl/RBioFabric`, 2013. *R* package version 0.3.

[7] Wendy L. Martinez, Angel R. Martinez, and Jeff Solka. *Exploratory Data Analysis with MATLAB, second edition*. Chapman and Hall/CRC, London, 2010.

[8] SAMHSA: Substance Abuse and Mental Health Services Administration. Drug Abuse Warning Network. `http://www.samhsa.gov/data/emergency-department-data-dawn`, 2010.

[9] The World Bank. World Bank Open Data. `data.worldbank.org/`, 2016.

[10] John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, Boston, 1977.

[11] Paul F. Velleman and David C. Hoaglin. *Applications, Basics and Computing of Exploratory Data Analysis*. Duxbury, Pacific Grove, CA, 1981.

[12] Hadley Wickham. `tidyr`: Easily Tidy Data with 'spread()' and 'gather()' Functions. `http://cran.r-project.org/web/packages/tidyr`, 2016. *R* package version 0.6.0.

[13] Hadley Wickham, Jim Hester, Romain Francois, Jukka Jaylnki, and Mikkel Jorgensen. `readr`: Read Tabular Data. `http://cran.r-project.org/web/packages/readr`, 2016. *R* package version 1.0.0.

[14] Hadley Wickham and Evan Miller. `haven`: Import 'SPSS', 'Stata' and 'SAS' Files. `http://cran.r-project.org/web/packages/haven`, 2016. *R* package version 0.2.1.