

Statistics with R: a hands-on approach

Myriam Luce

October 11, 2018

Contents

1	Data review	4
1.1	Possibly interesting extra tidbits	4
1.2	Variables	4
1.3	Gotchas	5
1.3.1	Tidy data	5
1.3.2	Correlation is not causation, or of the importance of DoE	6
2	Unavoidable R before we begin	7
2.1	Packages	7
2.2	Types	8
2.3	Accessing collection elements	11
2.4	Misc	12
2.4.1	c is for concatenate	12
2.4.2	# is for comments	12
2.4.3	Use argument names	12
2.4.4	Dates	13
2.4.5	Gotcha: graph image format	16
2.5	Saving, a.k.a scripts	16
2.6	Data: import, export, tidy	17
2.6.1	Import	17
2.6.2	Export	23
2.6.3	Tidy	23
3	Data presentation	28
3.1	Frequency table (1D) or contingency table (2D)	28
3.2	Pie chart	28
3.3	Bar chart	31
3.4	Histogram	42
3.5	Scatter plot	44
3.6	Box and whiskers graph	48
3.7	Putting it all together: summary figure	51

4	Data description	52
4.1	Center tendency measurements	52
4.1.1	Mean	52
4.1.2	Median	60
4.1.3	Mode	62
4.2	Dispersion measurements	62
4.2.1	Range	62
4.2.2	Variance	62
4.2.3	Standard deviation	62
4.2.4	Coefficient of variation	62
4.2.5	Quartiles and percentiles	62
4.3	Shape measurements	62
4.3.1	Skewness	62
4.3.2	Kurtosis	62
4.3.3	L-moments	62
5	Probabilities	63
5.1	Factorial	63
5.2	Combinations	63
5.3	Permutations	63
5.4	Probability Mass/Density Function	63
6	Statistics	64
6.1	Discrete variables	64
6.1.1	Binomial distribution	64
6.1.2	Multinomial distribution	64
6.1.3	Poisson distribution	64
6.1.4	Inverse binomial distribution	64
6.1.5	Hypergeometric distribution	64
6.2	Continuous variables	64
6.2.1	Normal distribution	64
6.2.2	Exponential distribution	64
6.2.3	Gamma distribution	64
6.2.4	c2 distribution	64
6.2.5	Fisher-Snedecor distribution	64
6.2.6	Student's law	64
7	Inferential statistics	65
7.1	Student's test	65
7.2	Student's paired test	65
7.3	Bartlett's test	65
7.4	Single-factor ANOVA	65
7.5	c2 test	65
7.6	Wilcoxon-Mann-Whitney test	65
7.7	Kolmogorov-Smirnov test	65
7.8	Kruskal-Wallis test	65

7.9	Pearson's test	65
7.10	Spearman's test	65
7.11	Kendall's test	65
7.12	Simple linear regression	65
7.13	Multiple linear regression	65
8	Programming	66
8.1	Sequence, iteration, branching	66
8.1.1	Iteration	66
8.1.2	Branching	66
8.2	Functions	66
8.3	Misc	66
8.3.1	Real saga	66
8.3.2	Rounding	66
9	More eccentric graphs	67
9.1	Rate plot	67
9.2	Violin	69
9.3	Map	69
10	Cheat sheet	70
10.1	Plumbing	70
10.2	Data import and export	70
	Glossary	73

Chapter 1

Data review

1.1 Possibly interesting extra tidbits

In the making of this tutorial, I used several tools that you might like to access as well. Being the tedium-averse programmer I am, I use a reference manager program, in my case Zotero. You can find the full bibliography for this project, including a few entries that did not make it into the references section here because I did not cite them, at the Zotero project page.

I also want to point out awesome-public-datasets, where I foraged for the examples in this tutorial. It has several interesting datasets in the public health domain.

Finally, this tutorial is not necessarily intended to be done in order. In particular, section 2.6 should be read as needed, rather than all the beginning. Not only the section on tidying up data might be boring as a first subject on R, but it's also unlikely to make much sense without some hands-on experience using the software. However, it seemed to make sense to keep all things R together, to make it easier to find again later, when the reader encounters the unavoidable data QA issue.

1.2 Variables

When “doing science”, you will be taking measurements, usually in the hopes of understanding a phenomena often in the shape of a relationship between things you are measuring. When working with this data, the nature of the things you measure (*variables*) will influence the presentation and analysis that are appropriate.

A **qualitative variable** refers to categories, or a variable recorded with words, as opposed to a **quantitative variable**, which is measured with numbers. Examples of qualitative variables would include US state, restaurant chains and college major. Quantitative variables could be time to execute a task, waist circumference, disease rate or spending amounts.

R refers to **qualitative variables** as *factors*. **Qualitative variables** can further be divided into non ordinal and **ordinal variables**, depending on whether there is a natural order among the categories. For example, dog breed (chihuahua, husky, labrador) is a non ordinal variable, whereas level of satisfaction (dissatisfied, neutral, satisfied) is ordinal.

Quantitative variables are either **discrete variables**, where measurements are done in integers, or **continuous variables**, where they come in real numbers (you could get an infinity of decimals with a theoretical instrument of infinite precision). **Discrete variables** could be number of children, cancer deaths, or wedding age. **Continuous variables** include temperature, blood sugar level, and weight.

Furthermore, when studying variables in relationship with one another, changes in a **dependent variable** are driven or explained by an **independent variable**. Typically, this means the “x” axis of a graph will be the **independent variable**, while the “y” axis will be the **dependent variable**.

1.3 Gotchas

1.3.1 Tidy data

When working with data in R, analysis will be easier if your data is *tidy*, that is, each column in your data set contains one and only one variable. Or, more completely:

1. Each variable is in its own column
2. Each observation is in its own row
3. Each value is in its own cell

(Garrett Grolemund gives an excellent introduction to the subject [6].)

For example, in a cancer dataset that we will use later, the original data is presented as in table 1.1. Here, we have four variables: cancer type, sex, number of cases, and number of deaths. While the first column is one and only one variable, the other columns mix sex with number of cases or sex with number of deaths. If you would like to analyze deaths by sex or cases per cancer type, some data manipulation will be necessary to combine the relevant columns.

Cancer Type	Cases		Deaths	
	Male	Female	Male	Female
...

Table 1.1: Cancer data set format.

If you keep your data tidy, R can usually do the combining for you, if you know how to ask nicely. As such, it is recommended that the first thing you do after successfully importing data into R is to verify it is tidy. Tools to divide or merge columns will be discussed in section 2.6.

1.3.2 Correlation is not causation, or of the importance of DoE

DoE refers to design of experiments. The common trope that “correlation is not causation” refers to the fact that because two variables vary together does not necessarily mean that one causes the other to change; for instance, they might both be responding to a common cause, when it’s not just plain old coincidence. My personal favorite exemplification of “correlation is not causation” is that the divorce rate in Maine correlates with the per capita consumption of margarine [21].

To distinguish between correlation and causation with certainty, a controlled experiment must be run. Depending on the phenomena studied, this might mean using a control group, a placebo, or directly controlling environmental conditions. For example, to determine the effect of low oxygen concentration in water on cod growth, several tanks can be set up where individual cods are randomly distributed and where oxygen levels are controlled. If you were to simply measure oxygen levels in water and cod growth at different locations and subsequently find a correlation between the two, you couldn’t tell for certain whether the difference is due to oxygen levels, or another factor like water temperature, or even the fact that cod compete with one another and that the runts, who would grow slower anyway, end up pushed into less desirable low-oxygen environments. Of course, sometimes running a controlled experiment is not feasible for practical reasons (one can’t control amount of natural sunlight, for example) or ethical reasons (having an untreated control group of people with a serious condition, when a potentially life-saving treatment might exist, is questionable).

When dealing with humans, to determine whether medication has a positive effect on an health issue, the health issue can be measured for a group who took the medication (treatment group), a group who took a placebo, and a group who took no medication at all (control group). In humans, particular effort must be placed on controlling or measuring the placebo effect, for the test subjects as well as the professionals. A recent spectacular example is the recommendation to abandon arthroscopic surgeries for knee pain because it did not show better results than physical therapy in randomized trials, despite it being the most common orthopaedic procedure in several countries [18].

Where experimental design is concerned, key factors are randomization, blocking, and replication. If terms like Completely Randomized Design, Latin Squares or Factorial Design are not familiar, I would recommend investing some time into learning the basics of experimental design before embarking in an experiment, in the interest of avoiding some common and easily remedied mistakes ([10] appears to be a well-rounded textbook).

Chapter 2

Unavoidable R before we begin

Despite the title, this chapter is not meant to be read from start to finish before you get to the other chapters. First, a long text about programming quirks can get pretty sleep-inducing. Second, the concerns addressed will probably not resonate much with you unless you’ve personally encountered this problem before, so the information retention is likely to be low.

That being said, the three first sections (2.1, 2.2, 2.3) should probably be read right now if you are not familiar with R at all. The other ones are referenced later in the text when they are relevant and can be read at that time.

2.1 Packages

R [16] should be relatively straightforward to install: download and execute, follow the wizard instructions.

Where things get a bit more complicated is when it comes to packages. While the basic R program has a lot of functions built-in, there will come a time when you will need something that is not offered out of the box. Thankfully, R has a very dynamic community with a ton of packages. For instance, a very popular package to produce figures is `ggplot2`. Let’s install it to see how packages are managed in R.

First off, to install packages in R, you will need to launch it as an administrator. If you don’t, you will get the rather unhelpful message shown in figure 2.1. To launch with administrator rights, right-click on your R launcher and find the option “Run as administrator”. How to do so from the Windows 10 start menu is shown in figure 2.2. (As a note, you should launch as administrator *only* when installing packages, as opposed to modifying your shortcut to always launch as administrator.)

Now that R is launched as administrator, you can install `ggplot2` using either the convenient Packages menu or the command line if you’re that hard-

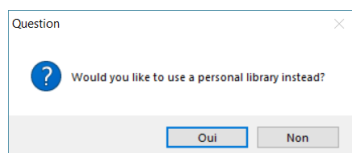


Figure 2.1: Error message displayed by R if trying to install packages without administrator rights.

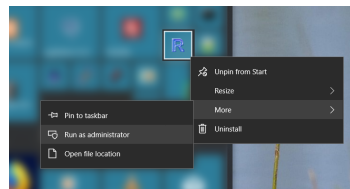


Figure 2.2: How to launch R with administrative rights.

core. Personally I use the menu; after choosing a mirror (different mirrors offer different packages; Canada NS has a wide selection and is vaguely geographically close), you can select your desired package and hit “install”, as shown in figure 2.3. Then you only need to wait until R is done doing its thing. If all the lines say “successfully unpacked”, all good; otherwise, an error has occurred and you will have to decipher the message to figure out how to remedy the situation. (If you run into any trouble, I would first recommend updating to the latest release of R.)

Note that installing a package is not enough to use it; you must also load it. Again this can be done either from the menu or with the command line, as shown in figure 2.4. This operation must be repeated *every time* you restart R.

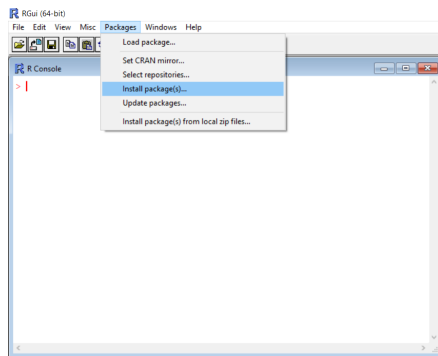
Now let’s say that `ggplot2` becomes your favoritest package in the whole world and you end up using it every day. After a week or so you will probably be very annoyed having to reload the package every time you open R. You can add packages to be loaded automatically in either of two files: `Rprofile.site` or `.Rprofile`. `Rprofile.site` is located in your R installation folder in the `etc` folder and is always executed. `.Rprofile` is located in the user home folder and is applied after the changes made by `Rprofile.site`. As the appropriate location to put `.Rprofile` seems to change from R version to version, I will showcase the `Rprofile.site` here. At the bottom of the file, add the following:

```
.First <- function(){
  library('ggplot2')
  # other libraries go here
}
```

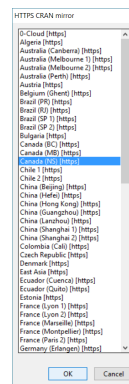
2.2 Types

While relatively accessible as far as programming languages go, R is still a programming language. As such, it has a few concerns that, while painful for the non-programmers among us, are useful tools to diagnose problems, typically when importing data that has not been properly QA’d or when a function refuses to compute because the input is in a form it refuses.

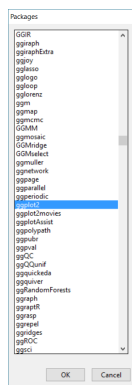
Let’s talk about *types*. Types refer to the nature of a variable in a computer



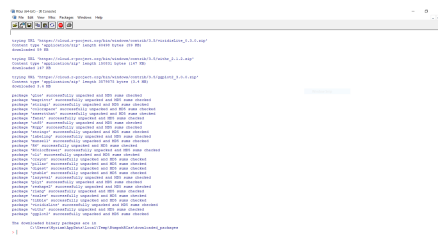
(a) Packages menu in R Gui.



(b) List of mirrors.



(c) List of packages.



(d) All successfully unpacked: successful installation.

Figure 2.3: Package installation process.

program; is it text, a number, etc.? If it's a number, can it be any number, or integers only? This is important because computing the means of grades in a class makes sense, while computing the means of the name of the students doesn't. Further, since R is a statistical program, it also includes types not typically seen in programming languages, like factors and ordinal variables. The following table gives an overview of variable types in R.

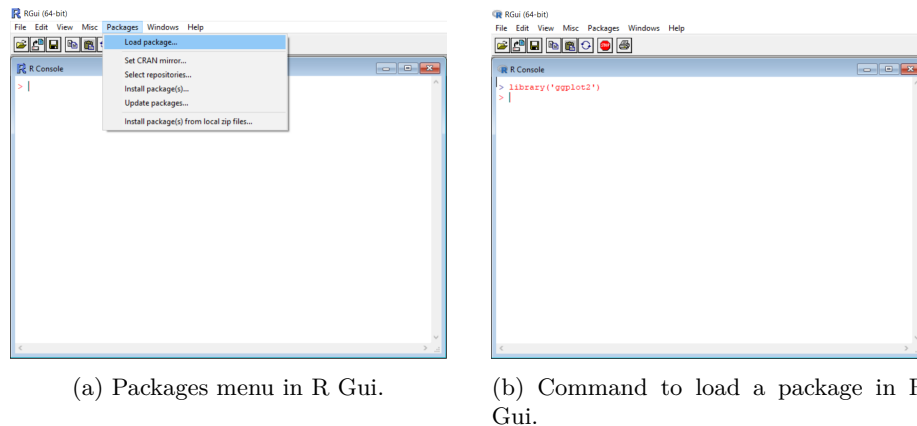


Figure 2.4: Loading packages in R.

Logical:	TRUE or FALSE
Numeric:	real, by the math definition (ex. 12.3). Double is a numeric with better precision.
Integer:	integer, by the math definition (ex. 12).
Character:	text of any length; defined by typing “text” or ‘text’
Factor:	a type that represents a qualitative variable
Ordered:	a type that represents an ordinal variable
List:	a 1D collection of “things” (may be strings, numbers, or a mix of them)
Vector:	a 1D collection of things of <i>one type</i>
Matrix:	a 2D collection of things of <i>one type</i>
Array:	a nD collection of things of <i>one type</i>
Data Frame:	a (mostly) 2D collection of things, where each column can be of a different type

For future reference, Quick R gives an excellent introduction on the subject [15]. You can convert a variable to anything reasonable (R will turn “2” into an integer, but not “abc”) using the host of `as.xyz` functions.

The data frame is of particular interest, since it allows the use of a specific syntax we will use later. A data frame is closest to a “table” you would have in your spreadsheet software: it holds values for several variables, where each column is a variable, and the headers hold sensible names.

To see which type a variable has, `class` and `str` (structure) are most informative. `class` will return the type of the variable (for example, “data frame”), while `str` will make a summary of the variable and its components, if any (for example, the various columns of a data frame).

2.3 Accessing collection elements

Some of the types presented in the previous section group several values. At some point, you'll want to access one or many of the elements, but not all. Let's say you have a data frame, for example ebola deaths by country [12]:

```
> ebola
```

	Country	Deaths
1	Guinea	2543
2	Liberia	4809
3	Sierra Leone	3956
4	Mali	6
5	Nigeria	8
6	United States of America	1

Notice how the line with “Country” and “Deaths” is not numbered in the output? It means R is aware it's a header and not data. *Data frame columns* (not matrices) can be accessed by their name using the `$` operator, like so:

```
> ebola$Country
[1] Guinea          Liberia          Sierra
   → Leone
[4] Mali            Nigeria          United
   → States of America
6 Levels: Guinea Liberia Mali Nigeria ... United States of
   → America
```

If you want to access lines, elements or columns, you can use the `[row, column]` operator, like so:

```
> ebola[1,]
  Country Deaths
1  Guinea  2543
> ebola[1,2]
[1] 2543
> ebola[,2]
Deaths
[1] 2543
[2] 4809
[3] 3956
[4] 6
[5] 8
[6] 1
```

While the `$` operator is exclusive to data frames, the `[]` is used for all collections. Vectors, lists, matrices and arrays can be accessed with the `[index]` operator for 1D structure, `[row, column]` operator for 2D structures, and `[i, j, k...]` for nD structures.

In the case you want to access several items at once, you can use a colon inside the brackets, i.e. `[begin:end]` like so:

```
> ebola[1:3,]
      Country Deaths
1      Guinea    2543
2     Liberia    4809
3 Sierra Leone    3956
```

You can also use the bracket operator to *filter* your data, that is, to select only certain values. For example, you could select the list of deaths which are greater than 10, or the countries where there has been more than 10 deaths.

```
> ebola$Deaths[ebola$Deaths > 10] #deaths where deaths>10
[1] 2543 4809 3956
> #ebola[rows where deaths>10, all columns]
> ebola[ebola$Deaths > 10,]
      Country Deaths
1      Guinea    2543
2     Liberia    4809
3 Sierra Leone    3956
```

2.4 Misc

2.4.1 `c` is for concatenate

Sometimes, a function in R will use an argument that is actually a list of things; for example, the limits of the x axis are two values: the minimum and maximum values to display on the graph. Referring back to the types we just saw, R requires a **vector** of values. Since it's such a common usage, R offers a quick way to create a vector with the function `c()`.

```
> ... xlim = c(0, 100), color = c(255, 0, 0) ...
```

2.4.2 `#` is for comments

If you use `#` in R, it will consider anything to its right to be a *comment*, that is, not code, and it will be ignored. You can start a line with it or use it in the middle of a line. It can be useful to leave notes to yourself in long-ish scripts.

2.4.3 Use argument names

When you use a function, for example `barplot`, there are a certain number of unavoidable parameters, followed by several optional parameters. The optional parameters typically have a default value, so if you don't specify them, the function works as expected, using the default values. For example, let's consult the help page for `barplot` by typing `?barplot` at the R prompt.

```

...
barplot(height, ...)

## Default S3 method:
barplot(height, width = 1, space = NULL,
        names.arg = NULL, legend.text = NULL, beside = FALSE,
        horiz = FALSE, density = NULL, angle = 45,
        col = NULL, border = par("fg"),
        main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
        xlim = NULL, ylim = NULL, xpd = TRUE, log = "",
        axes = TRUE, axisnames = TRUE,
        cex.axis = par("cex.axis"), cex.names = par("cex.axis"),
        inside = TRUE, plot = TRUE, axis.lty = 0, offset = 0,
        add = FALSE, args.legend = NULL, ...)
...

```

In this example, the parameter `height`, which is not followed by an “=” is a necessary parameter; you can’t compute a bar plot without giving it some values to put in the graph! All the other parameters are optional, and the help page lists their default value.

If you decide you want your bars to be beside one another rather than stacked, you will have to set the parameter `beside`, which is the fifth optional parameter. To avoid really strange and unfortunate guesswork on R’s part when it tries to figure out which parameter you set among the gazillion optional parameters, *always use the optional parameter names*, for example:

```

> heights = c(15, 5, 1, 12, 28)
> # barplot(heights, TRUE)
> # BAD!
> # Don't make R guess!
> # It's bad at it!
> barplot(heights, beside = TRUE)           # Good

```

2.4.4 Dates

As I may have mentioned, dates in programming are their very own Pandora box. R internally saves its dates as the number of days since January 1st, 1970.

In real life, you will likely encounter dates in a file in one of various text formats, for example something like “mm/dd/yyyy” (and since not all languages list their dates in the same order, you should pay close attention when dealing with international data), or as three different columns for the day, the month and the year. If you want R to do nice things for you like calculating the number of days between two dates, you need to transform these eminently unstandard formats into R’s own format. So let’s see how this is done for an arbitrary text format and for the three numeric columns cases.

Text to Date

For this, let's use the traffic data from the city of Chicago [1]. After downloading the csv, in your spreadsheet software, assign appropriate types to the date, vehicle volume, latitude and longitude columns, then tweak header names, save as csv and load into R. You should obtain the following:

```
> traffic = read.csv("C:/.../r-tutorial/traffic.csv")
> str(traffic)
'data.frame': 1279 obs. of 9 variables:
 $ ID      : Factor w/ 1279 levels "1","1,000","1,001",...: 453...
 $ Address : Factor w/ 1203 levels "1 West","10 East",...: 212 ...
 $ Street  : Factor w/ 251 levels "100th St","101st St",...: 40 ..
 $ Date    : Factor w/ 95 levels "1/24/2007","1/30/2007",...:54...
 $ Volume  : int 14600 16500 18200 21600 18300 8600 10000 ...
 $ Direction: Factor w/ 1223 levels "East Bound: 1000 / West
   ↳ Bound: 3400",...: 397 442 484 20 495 257 939 936 981 909 ...
 $ Latitude : num 41.8 41.8 41.8 41.8 41.8 ...
 $ Longitude: num -87.7 -87.7 -87.6 -87.6 -87.6 ...
 $ Location : Factor w/ 1276 levels "(41.651861, -87.54501)",...
```

So your date has been read as a factor; dates are a kind of category, I suppose, but R doesn't make computations on factors. You can't subtract blue eyes from green eyes, but you definitely can subtract 2010/01/08 from 2010/02/07 to compute how many days have passed between them.

Thankfully, R offers the function `as.Date` to convert text to `Date`. It uses a series of codes that you can see in the details of the help page for `strptime`. For instance, `%m`, `%d` and `%Y` stand for decimal month, decimal day and decimal year including century, respectively. It is used in the following manner:

```
> traffic$Date = as.Date(traffic$Date, format="%m/%d/%Y")
> str(traffic)
'data.frame': 1279 obs. of 9 variables:
 $ ID      : Factor w/ 1279 levels "1","1,000","1,001",...
 $ Address : Factor w/ 1203 levels "1 West","10 East",...: 212 ...
 $ Street  : Factor w/ 251 levels "100th St","101st St",...
 $ Date    : Date, format: "2006-03-09" "2006-02-28" ...
 $ Volume  : int 14600 16500 18200 21600 18300 8600 10000 ...
 $ Direction: Factor w/ 1223 levels "East Bound: 1000 / West
   ↳ Bound: 3400",...: 397 442 484 20 495 257 939 936 981 909 ...
 $ Latitude : num 41.8 41.8 41.8 41.8 41.8 ...
 $ Longitude: num -87.7 -87.7 -87.6 -87.6 -87.6 ...
 $ Location : Factor w/ 1276 levels "(41.651861, -87.54501)",...
```

Three numbers to Date

While R does not offer a specific function that takes three numbers and returns a date, it is easy enough to build a date string from three numbers, and then

feed that to `as.Date`. For example, if the dates in the traffic data were in three columns, you could proceed this way:

```
> str(traffic2)
'data.frame': 1279 obs. of 12 variables:
#...
$ Month : int 3 2 2 2 2 3 3 3 3 3 ...
$ Day : int 9 28 28 28 28 9 7 7 7 7 ...
$ Year : int 2006 2006 2006 2006 2006 2006 2006 2006 2006 ...
#...
> dateStrings = paste(traffic2$Year, traffic2$Month,
  ↪ traffic2$Day, sep="/")
> dateStrings[1:10]
[1] "2006/3/9" "2006/2/28" "2006/2/28" "2006/2/28"
[5] "2006/2/28" "2006/3/9" "2006/3/7" "2006/3/7"
[9] "2006/3/7" "2006/3/7"
> traffic2$Date = as.Date(dateStrings, format="%Y/%m/%d")
> str(traffic2)
'data.frame': 1279 obs. of 12 variables:
#...
$ Date : Date, format: "2006-03-09" ...
$ Month : int 3 2 2 2 2 3 3 3 3 3 ...
$ Day : int 9 28 28 28 28 9 7 7 7 7 ...
$ Year : int 2006 2006 2006 2006 2006 2006 2006 2006 2006 2006
  ↪ 2006 ...
#...
```

Date to text

Converting a date to text is straightforward enough and allows you to specify the format you want. By default, R prints dates following the ISO convention of YYYY-MM-DD [2]. If you really want your heretic US format, you could use `format` (which, by the way, also allows you to format numbers by defining leading zeros or number of decimals).

```
> today = Sys.Date()
> format(today, format="%m/%d/%y")
[1] "09/27/18"
```

Date computations

You can make calculations on dates. The following examples should tell you the gist of it.

```
> lastyear = Sys.Date()-365
> lastyear
[1] "2017-09-27"
```



```
> today-lastyear
Time difference of 365 days
> nextmonth=today+30
> nextmonth
[1] "2018-10-27"
```

2.4.5 Gotcha: graph image format

Assuming you will want to take your graphs out of R at some point, you can save them quite simply by right-clicking anywhere in the plot window and choosing **Save as...** I would recommend *metafile* if you intend to paste it into the Microsoft Office suite, *postscript* any other time you can, and *bitmap* if all else fails.

The metafile plays quite well with the Microsoft Office suite, avoiding the blurriness of bitmap in that context. LibreOffice and LaTeX will accommodate vector graphics, i.e. postscript, which means *no* blurring no matter how you scale the image. Bitmap will produce a good old flat image that every piece of software out there will know how to display, but it might appear blurry if you change its size.

2.5 Saving, a.k.a scripts

As long as you're doing simple things fitting on two or three lines, you probably won't feel the need to "save your file". However, as you start doing more elaborate data treatment or need to document a process used, you will want to save your progress.

One way to save is to use R's built-in **workspace**. A **workspace** is a **.RData** file that contains all the variables (used here in the computer science sense: a value that you attached a name to) you have defined since you started R. For example, if you typed the following:

```
> 1+2
[1] 3
> a = "I am text"
> x = 5+3
> y = x-8
> x
[1] 8
```

In this case, **a**, **x** and **y** would be saved in your workspace. Next time you started R, you could load the **workspace** and R would know that **x** is worth 8.

Another useful feature of R is the history, that is, the 250 (by default) last commands you typed in the window. You can access them by pressing the up arrow, which can be pretty handy when you want to tweak a command to fix a typo. You can save it in a **.Rhistory** file that you can also load the next time you start R.

Finally, if there is a small routine that you need to save, you can save it in a simple text file that you can load and execute. In the **File** menu, choose **New script** and type some text in the window, for instance:

```
x = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
y = 2*x
x
y
plot(x, y)
```

As you type the commands and press enter, nothing happens, as opposed to using the main window. To run these commands, you need to select what you want to run (for example, everything: select with ctrl+a), then press F5. If you do, you should see the outputs appearing in the main window. Once you're reassured that your basic algorithm is running, you might want to tweak the plot function to add axes and labels before re-running the script. Once you're done, you can save this simple text file and have an easily viewable record of what you did.

2.6 Data: import, export, tidy

2.6.1 Import

Delimited data: `read.csv` is your friend

A typical workflow to get data from wherever into R would be as follows:

1. Copy-paste the data into your favorite spreadsheet software (Microsoft Excel, LibreOffice Calc, Google Sheets, etc.).
2. If necessary, transpose your data so that variables are in columns (rather than rows).
3. Tweak column names so they have no spaces and no special characters (é, \$, etc).
4. Assign a reasonable format (text, number, thousand separators, etc.) to all columns.
5. With your operating systems using an English locale, save as csv.
6. Use `read.csv` in R with the *full path* using *forward slashes*, and the appropriate options.

As a case study, let's import the data for infant mortality [9]. Data is already in columns and country names contain no special characters. So let's just change the first column header to "Country". Next, let's set the columns B and up (excluding the header for practical reasons seen later) to format "number". You might note that the decimal separator used in this file is a comma rather

than a dot. However, setting the number format should be enough for your Spreadsheet software to convert them properly. Where to set number format will vary depending on your spreadsheet program; how to apply number formatting in LibreOffice is shown in figure 2.5. In LibreOffice in particular, make sure your number format locale is English. Save the modified file in csv format.

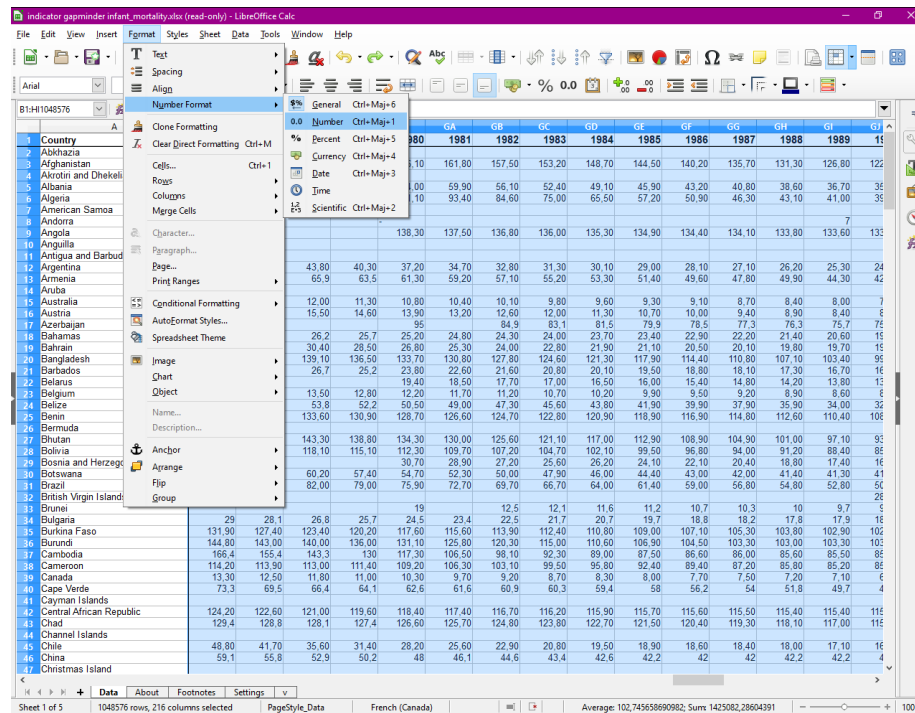


Figure 2.5: Applying number format to selected columns in LibreOffice.

Once in R, import the data using `read.csv`. Once that is done, however, you should always doubt that everything went well. Just to prove my point, let's examine the imported data a little more closely (see section 2.2 about data frames and section 2.3 about the `$` operator):

```
> infant = read.csv('C:/.../r-tutorial/infant.csv', header=TRUE)
> class(infant)
[1] "data.frame"
> str(infant)
'data.frame': 260 obs. of 217 variables:
 $ Country: Factor w/ 260 levels "Abkhazia", "Afghanistan", ...: 1
  ...
 $ X1800 : int NA NA NA NA NA NA NA NA NA NA ...
 $ X1801 : int NA NA NA NA NA NA NA NA NA NA ...
# ...
```

```
$ X1861 : int  NA NA NA NA NA NA NA NA NA NA NA ...
$ X1862 : Factor w/ 11 levels  "", ".", "110", "131", ...: 1 1 1 1 ...
$ X1863 : Factor w/ 13 levels  "", ".", "106", "113", ...: 1 1 1 1 ...
# ...
```

Wait, what? The country is of type factor, we all agree on that, but infant mortality rate for 1862 is a factor? Let's pick 1862 and see if we can't just eyeball the problem:

```
> infant$X1862
[1] 250.00
[16] 150.00
[31]
[46] 131.00
[61]
[76] 163.00
[91]
[106]
[121]
[136]
[151] 193.00 81.00
[166] 110.00
[181]
[196]
[211] 175.00 139.00
[226] .
[241]
[256]
Levels: . 110.00 131.00 139.00 150.00 163.00 175.00 193.00
→ 250.00 81.00
```

I don't know if you can see it, or if you think it's a speck of dust on your screen, but there's a lonely dot there somewhere between lines 226 and 241. I am going to assume it means it's a missing data point, since nothing else makes sense.

(In other languages, like French for instance, the decimal “point” is a comma, and therefore the “comma-separated” part of comma-separated value leads to some issues, not to mention the English thousand separator. This is why you set your data format in your spreadsheet: once the cell are properly formatted, your spreadsheet will export them sensibly into the csv.)

Let's try again, this time telling `read.csv` that dots are missing data:

```
> infant = read.csv("C:/.../r-tutorial/infant.csv", header=TRUE,
→ na.strings=".")
> str(infant)
'data.frame': 260 obs. of 217 variables:
$ Country: Factor w/ 260 levels "Abkhazia", "Afghanistan", ...: 1 2
→ 3 5 6 7 8 9 10 11 ...
```

```
$ X1800 : num NA NA NA NA NA NA NA NA NA NA ...
# ...
$ X1897 : num NA NA NA NA NA NA NA NA NA NA ...
[list output truncated]
```

This looks better, however the last line displayed is for 1897. If you're a trusting person (and you should never trust a computer), you might think everything is okay now. However, I cheated and skipped ahead and tried to use this data before and encountered another QA gem. So let's make sure to display *all* the variables with a trick we will see in detail later (section 8.1.1; for now let's just say it applies `class` to every part of `infant`).

```
> sapply(infant, class)
Country      X1800      X1801      X1802      X1803      X1804
  → X1805      X1806      X1807      X1808      X1809
"factor" "numeric" "numeric" "numeric" "numeric" "numeric"
  → "numeric" "numeric" "numeric" "numeric" "numeric"
# ...
X1953      X1954      X1955      X1956      X1957      X1958      X1959
  → X1960      X1961      X1962      X1963
"numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
  → "numeric" "factor" "numeric" "numeric" "numeric"
# ...
# See that pesky 1960?
> infant$X1960
[1]      245.00      115.40 148.20      -      208.00
  → 59.87      20.30 37.30
# ...
[256] 88.00 123.20 92.60
146 Levels: - 100.60 101.60 102.00 102.10 102.20 105.00 106.70
  → 107.40 107.50 110.60 112.00 115.40 115.50 ... 94.00
```

So, apparently “-” also means missing data? Gods forbid the authors hit a snag while importing data and their software didn't warn them something foul was afoot (like R just did to us, thank you R) and they just pasted it in the global csv without noticing. (This is why science should be in databases. Real databases. They don't let you put a dash in a number field, they just don't.)

Now, *finally*, we get:

```
> infant = read.csv('C:/.../r-tutorial/infant.csv', header=TRUE,
  → na.strings=c('-', '.'))
> sapply(infant, class)
Country      X1800      X1801      X1802      X1803      X1804
  → X1805      X1806      X1807      X1808      X1809
"factor" "numeric" "numeric" "numeric" "numeric" "numeric"
  → "numeric" "numeric" "numeric" "numeric" "numeric"
# ...
```

Fixed-width data: `read.fwf`

Infant mortality rate,1950,1951,1952,1953,1954,1955,1956,1957,1958,1959,1960,1961,1962	Week	Nino1-2	Nino3	Nino3-4	Nino4
Akhzhia					
Afghanistan, "245,00", "240,50", "236,30", "232,30", "228,50", "224,60", "220,70", "	03JAN1990	23.4-0.4	25.3-0.3	26.4-0.4	26.6-0.3
	10JAN1990	23.4-0.4	25.3-0.3	26.4-0.4	26.6-0.3
Aktotiri and Dhekelia, "110,80", "110,80", "110,80", "106,50", "101,80", "97,90", "94,10", "90,20",	17JAN1990	24.2-0.3	25.3-0.3	26.5-0.1	26.6-0.3
Albania, "120,10", "115,40", "110,80", "106,50", "101,80", "97,90", "94,10", "90,20",	24JAN1990	24.4-0.5	25.8-0.4	26.5-0.1	26.6-0.3
Algeria, "150,40", "145,90", "145,50", "149,00", "148,80", "148,40", "148,20", "148,10", "	31JAN1990	25.9-0.3	26.7-0.2	27.0-0.2	27.0-0.2
	07FEB1990	25.8-0.2	26.1-0.1	26.8-0.1	26.4-0.3
	14FEB1990	25.8-0.2	26.1-0.1	26.8-0.1	26.4-0.3
Andorra, "180,00", "180,00", "180,00", "180,00", "180,00", "180,00", "180,00", "180,00",	21FEB1990	26.1-0.1	26.7-0.2	27.1-0.3	26.9-0.8
	28FEB1990	26.1-0.2	26.7-0.1	27.2-0.3	26.9-0.8
Angola, "208,00", "180,00", "180,00", "138,30", "137,50", "136,80", "136,00", "135,30",	06MAY1990	26.7-0.2	27.0-0.2	27.3-0.1	27.3-0.1
	13MAY1990	26.7-0.2	27.0-0.2	27.3-0.1	27.3-0.1
Antigua					
Antigua and Barbuda, "24,00", "24,00", "24,00", "24,00", "24,00", "24,00", "24,00", "24,00",	14MAY1990	26.1-0.2	26.9-0.2	27.0-0.1	26.8-0.4
	21MAY1990	26.1-0.2	27.0-0.2	27.4-0.3	26.8-0.7
Argentina, 68, 67, 65, 63, 60, 62, 57, 68, 61, 59, "59,87", "59,73", "59,58", "59,39", "59,25", "59,11",	28MAY1990	25.6-0.3	27.0-0.2	27.3-0.1	26.8-0.4
	04JUN1990	25.6-0.3	27.0-0.3	27.9-0.4	26.8-0.4
Armenia, "70,80", "68,30", "65,93", "63,59", "61,30", "59,20", "57,10", "	11JUN1990	25.1-0.6	27.6-0.2	27.9-0.2	26.8-0.3
	18JUN1990	25.1-0.6	27.6-0.2	27.9-0.2	26.8-0.3
Australia, 61, 62, 50, "48,10", "46,60", "45,20", "43,60", "41,70", "39,60", "37,30", "35,00", "32,10",	25JUN1990	25.1-0.0	27.7-0.4	28.2-0.4	29.2-0.6
	02JUL1990	25.1-0.0	27.7-0.4	28.2-0.4	29.2-0.6
Austria, 66, 61, 52, 50, "48,10", "46,60", "45,20", "43,60", "41,70", "39,60", "37,30", "35,00", "32,10",	09JUL1990	24.2-0.2	27.8-0.3	28.1-0.3	26.9-0.1
	16JUL1990	24.2-0.2	27.8-0.3	28.1-0.3	26.9-0.1
Azerbaijan, "95,84", "95,84", "83,11", "81,21", "79,79", "78,54", "77,3", "77,3",	14MAY1990	24.3-0.1	27.4-0.3	28.0-0.2	26.8-0.1
	21MAY1990	24.3-0.1	27.4-0.3	28.0-0.2	26.8-0.1
Bahamas, "51,25", "25,51", "26,26", "26,26", "26,26", "26,26", "26,26", "26,26",	28MAY1990	23.4-0.1	27.1-0.3	27.9-0.2	26.9-0.1

As a case study, let's use historic water levels for a river in Canada [14]. Data looks like this:

21

```

17 PL: La donnée correspond %*à la premi\ère lecture de
    niveau d'eau de la journée.
18 R:  Le débit est corrigé pour tenir compte de l'effet
    de refoulement.
19 S:  La donnée est saisie manuellement.
20 Z:  La donnée provient d'une redistribution temporelle
    des données enregistrées en raison d'une dé
    fectuosité de l'appareil de mesure.

```

```

21
22 Station          Date          Débit (m3/s)
    Remarque
23 070401          1973/08/17        86.60          J
24 070401          1973/08/18        79.90          MC
25 070401          1973/08/19        73.30          MC
26 070401          1973/08/20        68.80          MC
27 070401          1973/08/21        65.10          MC
28 070401          1973/08/22        63.70          MC
29 070401          1973/08/23        63.70          MC
30 070401          1973/08/24        62.00          MC
31 ...
32 070401          1994/09/25        47.55          MC
33 070401          1994/09/26        46.74          MC
34 070401          1994/09/27        45.95          MC
35 070401          1994/09/28        46.26          MC
36 070401          1994/09/29        54.98          MC
37 070401          1994/09/30        80.23          MC

```

Save it as a text file on your computer as is. To read fixed-width data, you need to explicitly tell R the widths of each column. The popular Windows text editor Notepad++ shows column index, which allows to calculate them quickly: 6, 20, 15 and 12 (column starts with whitespace and ends with data). Furthermore, the “table” part of the file starts on line 23.

(Since there are accents in the file, if you were interested in the comments, you might want to take an extracurricular dive into encodings. Since this is an English document, I will not add another painful tangent, but as a quick note, if you ever encounter trouble importing European documents, try CP-1252 (Windows default) or ISO-8859-1 (Latin extended, covers French, German and Spanish, for instance).)

(Another note, there is a function to read Fortran files, which I never needed to use but might be useful to you. My condolences on dealing with Fortran.)

`read.fwf` is used this way:

```

> debit = read.fwf("C:/.../r-tutorial/debit.txt", widths=c(6, 20,
  ↪ 15, 12), header=FALSE, skip=22, strip.white=TRUE,
  ↪ col.names=c('Station', 'Date', 'Debit', 'Remarque'))
> str(debit)
'data.frame':  7715 obs. of  4 variables:

```

```

$ Station : int 70401 70401 70401 70401 70401 70401 70401 ...
$ Date    : Factor w/ 7715 levels "1973/08/17","1973/08/18",...: 1
↪  2 3 4 5 6 7 8 9 10 ...
$ Debit   : num 86.6 79.9 73.3 68.8 65.1 63.7 63.7 62 59.2 ...
$ Remarque: Factor w/ 5 levels "E","J","MC","R",...: 2 3 3 3 3 ...
> debit[1:5,]
  Station      Date Debit Remarque
1  70401 1973/08/17  86.6         J
2  70401 1973/08/18  79.9        MC
3  70401 1973/08/19  73.3        MC
4  70401 1973/08/20  68.8        MC
5  70401 1973/08/21  65.1        MC

```

As opposed to `read.csv`, I used the argument `header=FALSE` with `read.fwf`. This is due to `read.fwf` being pickier about the header format: it wants the header to be *delimited* with a character that is not present in the rest of the file (to practice, type in `?read.fwf` in R and *attentively* read the help about the `header` argument). Since this was not the case, I manually set the column names with `col.names.strip.white = TRUE` automatically strips the white-space within the columns, so your date is '1973/08/17' and not ' 1973/08/17'.

The structure of the data frame informs us that the date as been read as a factor. Since dates are their own Pandora boxes in computer science, we will not deal with them here, but you can look at section 2.4.4 if you're a masochist.

2.6.2 Export

Exporting data is useful to save it for later use or send to a spreadsheet software. Despite some internationalization issues, I would recommend using `csv` for the output file, since it is easy to import into spreadsheet software. In that simple case, the `write.csv` function works quite well:

```

> demo
[,1] [,2] [,3] [,4] [,5]
[1,] 82.94 115.94 89.48 101.06 91.23
[2,] 111.22 117.65 94.64 115.79 103.91
[3,] 82.10 95.96 101.11 82.44 98.84
> write.csv(demo, file='path.../demo.csv', row.names=FALSE)

```

With data frames, the column headers will make sense and, should your object have row names, you can remove the `row.names` argument.

2.6.3 Tidy

This section is quite heavy on R programmy-like stuff, so you may want to skip it until you are more familiar with R or actually need to disentangle a data set, whichever comes first.

Since their help pages are, in my opinion, easier to understand, I use the functions in the `tidyr` package, so you might want to install and load it to follow along.

n -> 1 rows: spread

When the data is placed in a table where the name of a variable is used as a value in cells, you need to make them into columns. In other words, your table has a column of *keys* followed by a column of *values*. You can do so with the `spread` function, the result of which is represented in figure 2.7.

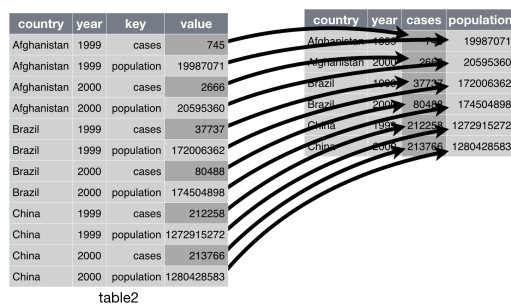


Figure 2.7: Putting variables in their own columns with `spread` (taken from [6]).

1 -> n rows: gather

When the data is placed in a table where the column headers are values of a variable, you need to put them into one column, as shown in figure 2.8. You can do so with the `gather` function.

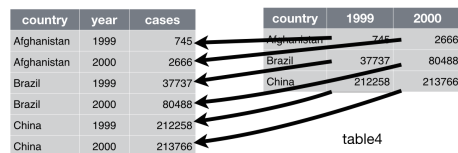


Figure 2.8: Putting a variable into only one column with `gather` (taken from [6]).

1 -> n columns: `separate`

If one of the columns contain more than one variable, for example it is a rate written as cases/population, since number of cases and population are two separate variables, you will need to split this column into several others. You can use the `separate` function for this, as shown in figure 2.9.

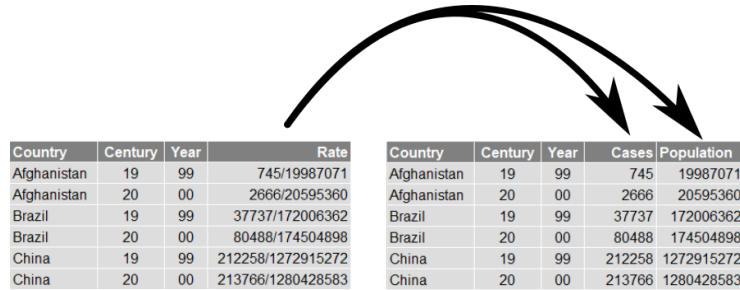


Figure 2.9: Separate one column into two with `separate` (inspired by [6]).

n -> 1 columns: `unite`

If one variable is spread across multiple columns, for example a date split in year, month and day, while the date is a single variable, you will need to combine them into a single column. This can be done with the `unite` function, as pictured in figure 2.10.

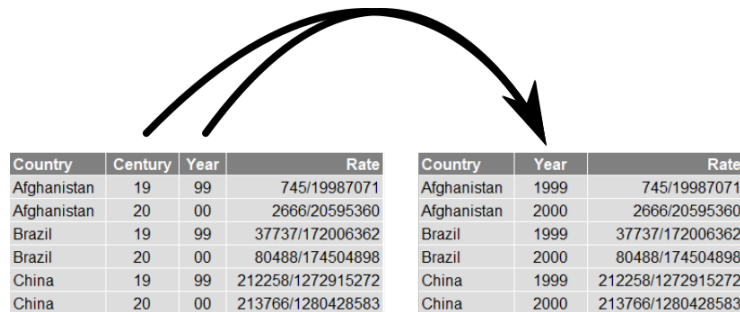


Figure 2.10: Merge two columns into one with `unite` (inspired by from [6]).

Example

The cancer data mentioned as an example in section 1.3.1 will be used as a case study for these functions. Data can most easily be copy-pasted into a Spreadsheet software from Table 1 of the peer-reviewed article version of the report online [17]. As a first step, remove sum lines and columns, remove the

thousand separator and tweak header names (also, the hyphen is not a real hyphen, so retype it).

Let's identify the variables: cancer type, sex of patient, number of cases, and number of deaths. Here, we need to make sex its own column; and since there are two columns with female data, we will proceed in several steps: first, make all four columns one "variable" with `gather`. Arguments `key` and `value` are simply the name to give to the new columns that `gather` will create. The last argument is the columns to "gather", while the columns not listed will be duplicated as necessary.

```
> cancer = read.csv('C:/.../r-tutorial/cancer.csv')
> str(cancer)
'data.frame':  47 obs. of  5 variables:
 $ cancer      : Factor w/ 47 levels "Acute lymphocytic
   ↪ leukemia",...
 $ cases_male   : int  12490 7980 14250 2440 13480 16520 ...
 $ cases_female : int  4620 5600 3340 820 3810 9720 5040 ...
 $ deaths_male  : int  1750 1770 2480 1280 12850 6510 810 ...
 $ deaths_female: int   760 880 750 360 3000 4290 640 23240 ...
> cancer = gather(cancer, key="tmpvar", value="n", 2:5)
> str(cancer)
'data.frame':  188 obs. of  3 variables:
 $ cancer: Factor w/ 47 levels "Acute lymphocytic leukemia",...
 $ tmpvar: chr  "cases_male" "cases_male" "cases_male" ...
 $ n      : int  12490 7980 14250 2440 13480 16520 5430 49690 ...
> cancer
      cancer      tmpvar      n
1  Tongue  cases_male 12490
2   Mouth  cases_male  7980
#...
48 Tongue  cases_female 4620
49  Mouth  cases_female 5600
#...
95 Tongue  deaths_male 1750
96  Mouth  deaths_male 1770
#...
142 Tongue deaths_female   760
143  Mouth deaths_female   880
#...
```

Now, we need to separate the two words in the "tmpvar" column into two columns with `separate`. Arguments should be self-explanatory.

```
> cancer = separate(cancer, col="tmpvar", into=c("category",
   ↪ "sex"), sep="_")
> cancer
cancer category sex      n
```

```

1   Tongue    cases    male  12490
2   Mouth     cases    male   7980
#...
48  Tongue    cases    female  4620
49  Mouth     cases    female  5600
#...
95  Tongue    deaths    male   1750
96  Mouth     deaths    male   1770
#...
142 Tongue    deaths    female   760
143 Mouth     deaths    female   880
#...

```

Finally, we need to bring “cases” and “deaths” back as their own column with `spread`. The argument `key` is the name of the column containing the variable names that you want to make into their own column. The argument `value` is the name of the column whose values you want to show under the new columns to be created. Other columns will be arranged accordingly.

```

> cancer = spread(cancer, key="category", value="n")
> str(cancer)
'data.frame':   94 obs. of  4 variables:
 $ cancer: Factor w/ 47 levels "Acute lymphocytic leukemia",...
 $ sex    : chr  "female" "male" "female" "male" ...
 $ cases  : int   2670 3290 9140 10380 5620 2960 1510 1940 ...
 $ deaths : int    640 830 4490 6180 680 480 660 930 7340 9490 ...
> cancer
      cancer    sex  cases deaths
#...
37  Mouth female   5600     880
38  Mouth  male   7980    1770
#...
81  Tongue female   4620     760
82  Tongue  male  12490    1750
#...

```

Finally, let’s correctly specify that “sex” is a **qualitative variable** rather than just text.

```

> cancer$sex = as.factor(cancer$sex)
> str(cancer)
'data.frame':   94 obs. of  4 variables:
 $ cancer: Factor w/ 47 levels "Acute lymphocytic leukemia",...
 $ sex    : Factor w/ 2 levels "female","male": 1 2 1 2 1 2 1 2 ...
 $ cases  : int   2670 3290 9140 10380 5620 2960 1510 1940 10160 ...
 $ deaths : int    640 830 4490 6180 680 480 660 930 7340 9490 ...

```

There we go! Tidy data set!

Chapter 3

Data presentation

3.1 Frequency table (1D) or contingency table (2D)

For qualitative, discrete and continuous variables

If you feel the need to make a table with your data, use a spreadsheet software. ;) R is superior in statistics and (arguably) in figures, but spreadsheets definitely have their uses when it comes to tables.

3.2 Pie chart

For qualitative and discrete variables, max 2 values

A pie chart is a graph that can be used to visually represent proportions of a **qualitative variable** or **discrete variable**. Note that they have their critics, who recommend never using them, as our brain is bad at comparing the size of slices [13].

As an example data set, let's use ebola deaths by country [12]. An excerpt giving the source data is shown in figure 3.1. Enter the data in your favorite spreadsheet software and save it as a csv. You should get the following:

```
Country,Deaths
Guinea,2543
Liberia,4809
Sierra Leone,3956
Mali,6
Nigeria,8
United States of America,1
```

Go ahead and load your small csv into R with `read.csv`. You can then use the function `pie` to produce a pie chart. However, as shown below, a naive approach might disappoint.

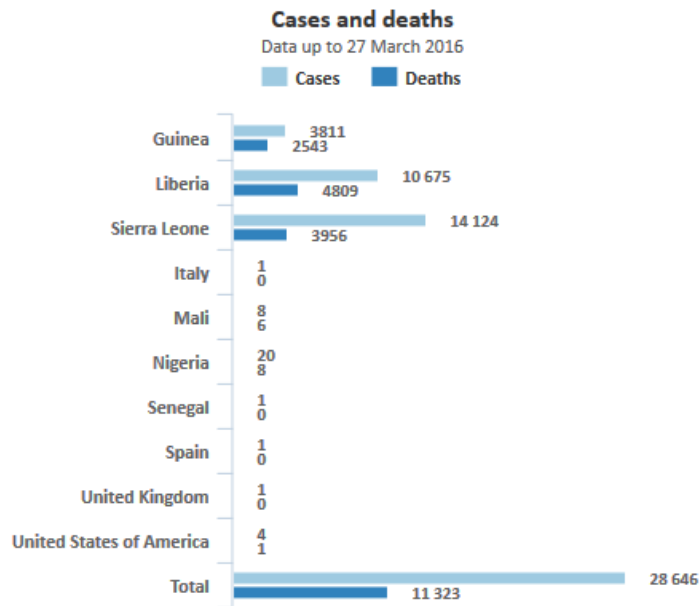


Figure 3.1: Excerpt from [12].

```
> ebola = read.csv('C:/.../r-tutorial/ebola.csv', header=TRUE)
> ebola
      Country Deaths
1      Guinea  2543
2     Liberia  4809
3  Sierra Leone 3956
4         Mali    6
5      Nigeria    8
6 United States of America 1
> pie(ebola)
Error in pie(ebola) : 'x' values must be positive.
```

You might be scratching your head and wondering which part of 2543 or 6 is not positive, and you'd be justified to do so. Here, one must dive into computer programming concerns to understand what is going on. The “not positive” message hints at a problem with the format or the type of the input data (see section 2.2). Let's demonstrate:

```
> values = c(2543, 4809, 3956, 6, 8, 1)
> labels = c('Guinea', 'Liberia', 'Sierra Leone', 'Mali',
  → 'Nigeria', 'United States of America')
> pie(values, labels=labels)      # works! produces figure
  → below
```

```

> class(values)
[1] "numeric"
> class(labels)
[1] "character"
> class(ebola)
[1] "data.frame"
> class(ebola$Country)
[1] "factor"
> class(ebola$Deaths)
[1] "integer"
> pie(ebola$Deaths, labels=ebola$Country)      # works too now!

```

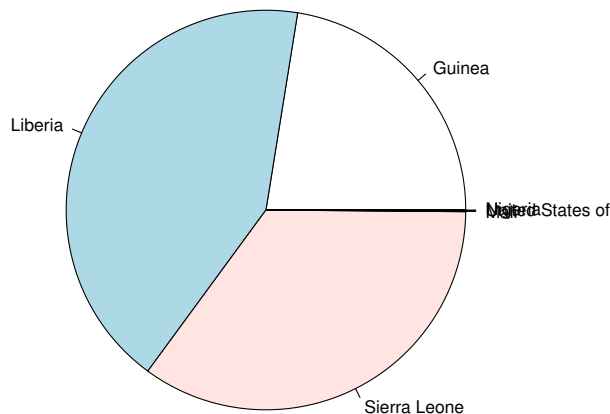


Figure 3.2: Ebola deaths in 2015-2016 by country.

Technically, `read.csv` returns a `data.frame`, while `pie` only accepts numbers. Accessing the columns of the data frame (see section 2.3) to feed `pie` the right types of arguments allows it to produce the expected figure.

Now that we have our basic pie chart, you might be thinking, “That squiggle on the right with the tiny pie slices is quite unseemly”. In addition, you might want to tweak other aspects of the graph, like adding a title or choosing colors. We will discuss common graph properties in a following section, to keep it all

in the same place. As a note, all options are always listed in the function's help page.

Let's just deal with the pie-chart specific problem of small slices here (I reiterate, you should run away, run away into the arms of a bar chart.), and add a percent annotation, as that is a common occurrence. R does not offer an option to deal with small slices out of the box (probably because it tells you in its own manual to use bar charts instead), so let's just manually tweak the labels:

```
> labels = as.character(ebola$Country)
> labels[4]='Others'
> labels[5:6]=' '
> labels
[1] "Guinea"          "Liberia"          "Sierra Leone" "Others"
↪ ""
[6] ""
> percents = ebola$Deaths/sum(ebola$Deaths)*100
> percents
[1] 22.458712355 42.471076570 34.937737349 0.052989490
↪ 0.070652654
[6] 0.008831582
> percents[4] = sum(percents[4:6])
> percents
[1] 22.458712355 42.471076570 34.937737349 0.132473726
↪ 0.070652654
[6] 0.008831582
> percents = round(percents, 2)
> percents
[1] 22.46 42.47 34.94 0.13 0.07 0.01
> labels[1:4] = paste(labels[1:4], percents[1:4], '%')
> labels
[1] "Guinea 22.46 %"      "Liberia 42.47 %"      "Sierra Leone
↪ 34.94 %"
[4] "Others 0.13 %"
> pie(ebola$Deaths, labels)
```

Hacky, but it works, and no more time should be dedicated to pie charts, so let's move on.

3.3 Bar chart

For qualitative and discrete variables

A bar chart, sometimes called a line graph, is used to represent a **qualitative variable** or a **discrete variable**, and the bars *do not touch*. As an example, data on infant mortality by country can be found at Gapminder [9]. The import process is detailed in section 2.6.1.

A barplot is relatively straightforward to produce with R, but we will see all “common” (imho) plot options here, so tie your winter hat down with wire, you’ll be sitting here a while. Let’s start by simply plotting infant mortality rate by country. To keep the plot readable, let’s choose a subset of G8 countries: Canada, France, Germany, Italy, Japan, Russia, United Kingdom and United States of America. Let’s also start by studying the mortality rate in 2000. First, we will select each of the countries by its row number, then we will stitch the G8 back together with a function called `rbind`, which binds data frames together by row, as long as all data frames have the same columns.

```
> canada = infant[38,]
> france = infant[77,]
> germany = infant[83,]
> italy = infant[109,]
> japan = infant[111,]
> russia = infant[186,]
> uk = infant[240,]
> usa = infant[241,]
> g8 = rbind(canada, france, germany, italy, japan, russia, uk,
  ↪ usa)
```

Producing a barplot now is easy:

```
> barplot(g8$X2000, names.arg=g8$Country)
```

Several things are wrong with this graph. Glaringly, a bar should not extend beyond its axis. Axes are set as plot options with `xlim` and `ylim`. Also, should you want a box around the graph, `bty` takes care of that. Usually. Bar plots are special and you need to call an extra function after your plot appears. See all graph options with `?par`, which we will use a lot more as we customize our graphs.

```
> barplot(g8$X2000, names.arg=g8$Country, ylim=c(0,20),
  ↪ bty='o')
> # why oh why won't bty work like everywhere else!
> box()
```

You probably also want all country names to show up. Easiest way to do that is to tilt the axis label text. Here we will learn about `par`, used *before* your graph function to specify general plotting settings. For this next iteration, let’s do a few things at once. First, let’s make all labels perpendicular to their axis with `par` and `las`. Let’s also demonstrate color manipulation by making each country’s bar the dominant color on their flag (I may have made some arbitrary choices) with `col`.

```
> colors = c('red', 'blue', 'black', 'green', 'white', 'snow',
  ↪ 'purple', 'purple4')
> par(las=2) # axis labels: perpendicular
```

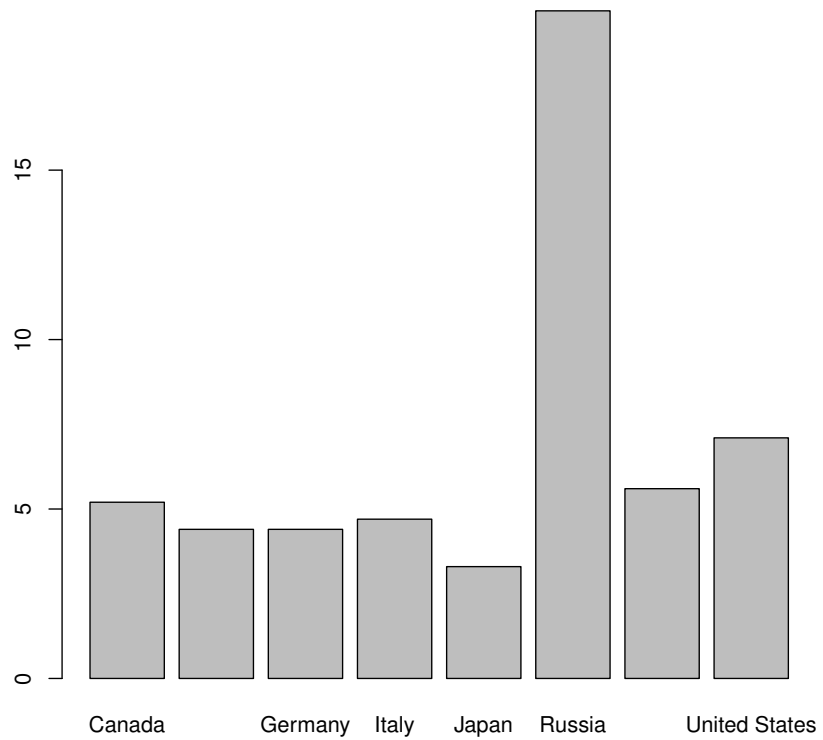


Figure 3.3: Simplest bar plot: infant mortality rate per country.

```
> barplot(g8$X2000, names.arg=g8$Country, ylim=c(0,20),  
  ↪ col=colors)  
> box()
```

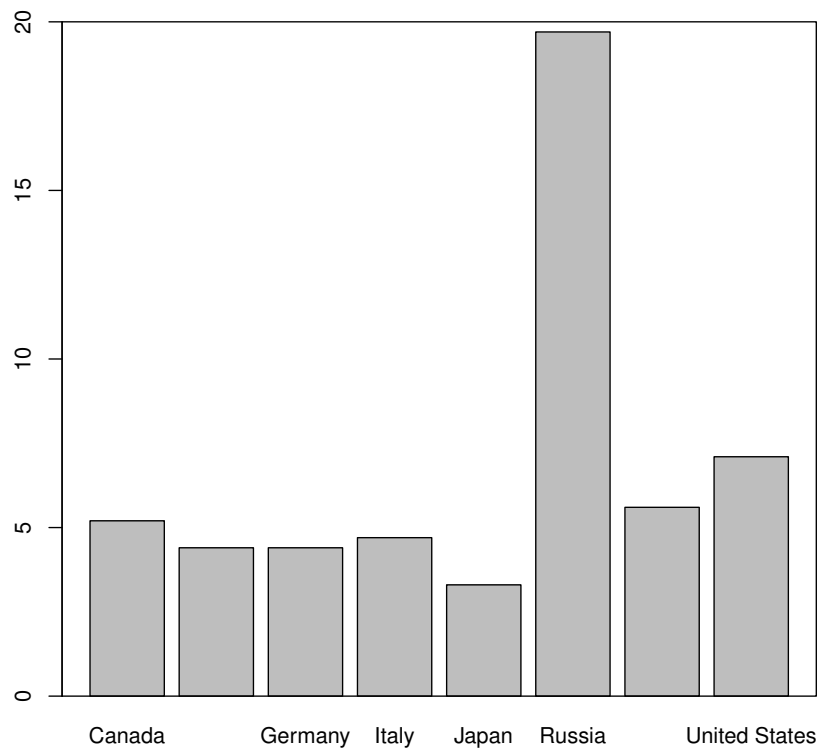


Figure 3.4: Simple bar plot: infant mortality rate per country, axis set.

Colors in R

Colors in R can be specified by their names, if they are among R's list of predefined colors, which you can see by calling `colors()`.

A more visually helpful version can be found at Color Chart [4] which, incidentally, has other fascinating references about the use of color in science (good vs. bad color ramps, color blindness, etc).

Additionally, colors can be specified in other formats like `#RRGGBB`. These values can be found with graphics software or off a color generator on the internet.

Finally, if color space is a concern, additional functions exist: `rgb`, `hsv`, `hcl`, `gray` and `rainbow`.

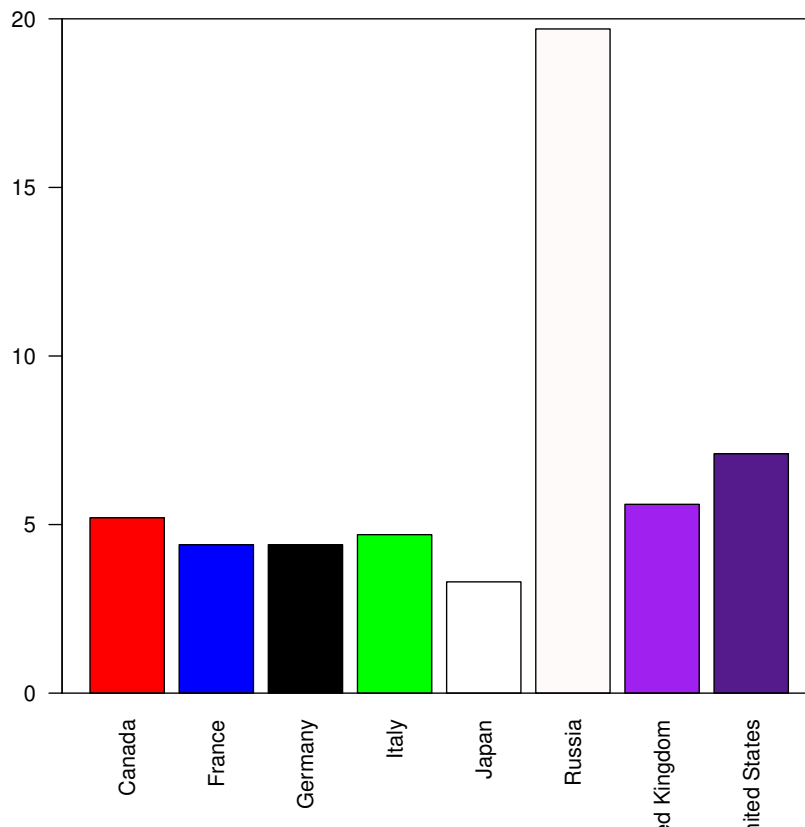


Figure 3.5: Psychedelic bar plot: infant mortality rate per country, axis set, labels perpendicular, colors.

With the country names printed at the vertical, they are running out of space at the bottom of the graph. More margin is needed there. Figures have two types of margins in R: outer and inner. The inner margin is used to draw the figure title and the axis ticks and labels and can be set in inches with `mai=c(bottom, left, top, right)` or in lines with `mar=c(bottom, left, top, right)`. The outer margin is outside the figure; it makes more sense when several plots are displayed together, as we will do a few exercises down the line. The outer margin as well can be set in inches with `omi=c(bottom, left, top, right)` or in lines with `oma=c(bottom, left, top, right)`. As for the appropriate margin necessary to display the full country name, that's a matter of trial and error. Starting with the current parameters' value of inner margin, I found that a value of 8 worked well.

```

par()$mar
[1] 5.1 4.1 4.1 2.1
> # mar = c(bottom, left, top, right)
> par(mar=c(8, 4.1, 4.1, 2.1))
> barplot(g8$X2000, names.arg=g8$Country, ylim=c(0,20),
  → col=colors)
> box()

```

Now, bar charts often use stacked bars. For example, let's use cancer rates [19]. This data includes number of cases and number of deaths by sex and cancer type. If the thing you would most like to compare is the number of cancer by type, you would stack the sexes into one bar. Let's use the tidy version of the data set produced in section 2.6.3.

```

> str(cancer)
'data.frame':  94 obs. of  4 variables:
 $ cancer: Factor w/ 47 levels "Acute lymphocytic leukemia",...
 $ sex   : Factor w/ 2 levels "female","male": 1 2 1 2 1 2 1 2 ...
 $ cases : int  2670 3290 9140 10380 5620 2960 1510 1940 10160 ...
 $ deaths: int   640 830 4490 6180 680 480 660 930 7340 9490 ...

```

The help page of `barplot` tells you that the first argument can be either a vector or matrix. If `height` is a vector, for example the single column `cancer$deaths`, then the barplot shows what one would expect. If `height` is a matrix, then each *column* must be either the values to stack (`beside=FALSE`) or the values for a group of bars (`beside=TRUE`). To visualize, let's plot data for five short-named cancers in the list. And since this is a perfect occasion, let's learn about a few more figure options: the `legend` argument and how to plot several figures into one plot window.

```

> subset=rbind(cancer[11:12,], cancer[17:18,], cancer[35:36,],
  → cancer[63:64,], cancer[79:80,])
> subset = subset[, 1:3]
> subset
      cancer  sex  cases deaths
11  Breast female 266120  40920
12  Breast  male   2550    480
17   Colon female 47530  23240
18   Colon  male 49690  27390
35 Melanoma female 36120   3330
36 Melanoma  male 55150   5990
63  Pharynx female  3340    750
64  Pharynx  male 14250   2480
79  Thyroid female 40900   1100
80  Thyroid  male 13090    960
> fordemo = spread(subset[, 1:3], cancer, cases) # let's just use
  → cases for now

```

```

> fordemo
sex Breast Colon Melanoma Pharynx Thyroid
1 female 266120 47530      36120      3340      40900
2  male   2550 49690      55150      14250      13090
> # each column of the matrix is one cancer to stack or group
> str(fordemo)
'data.frame':  2 obs. of  6 variables:
 $ sex      : Factor w/ 2 levels "female","male": 1 2
 $ Breast   : int  266120 2550
 $ Colon    : int  47530 49690
 $ Melanoma : int  36120 55150
 $ Pharynx  : int  3340 14250
 $ Thyroid  : int  40900 13090
> # but you want an actual matrix, not a data frame
> # with row and column names
> # and cells containing numbers only
> fordemo = as.matrix(fordemo[,2:6])
> fordemo
Breast Colon Melanoma Pharynx Thyroid
[1,] 266120 47530      36120      3340      40900
[2,]  2550 49690      55150      14250      13090
> rownames(fordemo) = c("female", "male")
> fordemo
      Breast Colon Melanoma Pharynx Thyroid
female 266120 47530      36120      3340      40900
male   2550 49690      55150      14250      13090

```

At this point, your data is ready in the shape that `barplot` wants it: it's a matrix with each column representing a stack or group. It's also in the shape *you* want it because the row and column names are readable. Now, let's use `mfrow` to plot the results of `beside=TRUE` and `beside=FALSE` side by side, as seen in figure 3.6.

```

> par(mfrow=c(1,2)) # 1 row, 2 columns, fill by row
> # par(mfcol=c(1,2)) # 1 row, 2 columns, fill by column
> # a plot window opens, ready with 1x2 plot regions
> barplot(fordemo, beside=FALSE, legend=TRUE, main="Parameter
→ beside=FALSE")
> # first plot appears on the left
> barplot(fordemo, beside=TRUE, legend=TRUE, main="Parameter
→ beside=TRUE")
> # second plot appears on the right

```

Note that setting the `legend` argument automatically produces a human-readable result from the row names of the “fordemo” matrix.

Now, if you want to stack sexes and group cases and deaths per cancer, this is where `barplot` would fail you and you would have to resort to a package, for

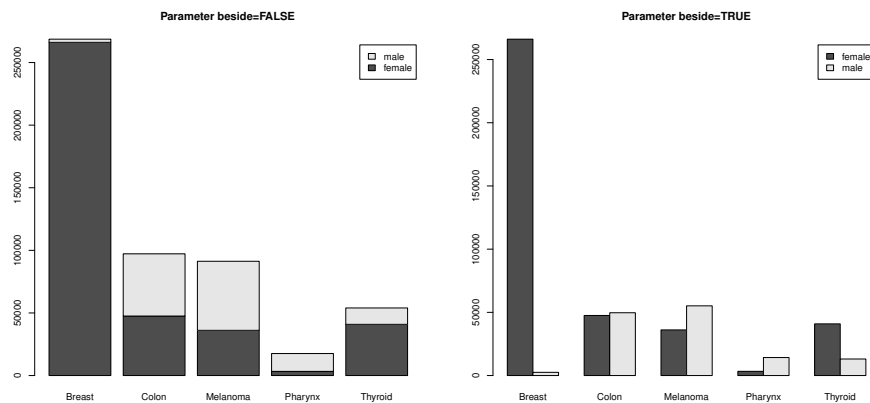


Figure 3.6: Comparing the effect of the `beside` parameter for a matrix fed to `barplot`, using `mfrow`.

example `ggplot2`. According to discussions in the R mailing lists, the basic bar plot function does not support this option because the developers felt it was too much information to fit on one figure, so keep that in mind and ask yourself whether it's a good idea. However, as shown by the pie chart section, I'm not here to judge, so here's a way to do that.

`ggplot2` is so ubiquitously used to produce peer-review quality graphics in R, while being so different from the basic graphics we have used so far that it warrants a bit of an introduction.

`ggplot2` works by sequentially adding *layers* from a *data frame*. So first, you say, "I want to make a new `ggplot`". An empty graphics window opens. Then you say, "I want to add points to this blank canvas". Then you can say, "I want to add a line in addition to the points already there", and then "I also want a main title", and so on. So you will add new function calls separated by a plus sign.

With our cancer study case, our situation is that we want to stack sexes, as well as group cases and deaths (which we will call effect) per cancer. You need each of these in their own column, and you need a data frame since that's what `ggplot` uses. Our data must be transformed this way:

```
> forgg = gather(subset, "effect", "n", 3:4)
> forgg
cancer  sex effect      n
1  Breast female cases 266120
2  Breast  male cases   2550
3  Colon  female cases  47530
4  Colon  male cases  49690
5 Melanoma female cases  36120
6 Melanoma  male cases  55150
```

7	Pharynx	female	cases	3340
8	Pharynx	male	cases	14250
9	Thyroid	female	cases	40900
10	Thyroid	male	cases	13090
11	Breast	female	deaths	40920
12	Breast	male	deaths	480
13	Colon	female	deaths	23240
14	Colon	male	deaths	27390
15	Melanoma	female	deaths	3330
16	Melanoma	male	deaths	5990
17	Pharynx	female	deaths	750
18	Pharynx	male	deaths	2480
19	Thyroid	female	deaths	1100
20	Thyroid	male	deaths	960

As for the plotting part of the work, `ggplot` loves to use scientific notation, so we'll turn that off. `ggplot` comes with several “themes”, which control a series of layout options like background color, presence of grids, etc. We'll use the classic theme since it looks most like the basic graphics package we have been using.

```
options(scipen=999)
theme_set(theme_classic())
```

`ggplot`'s first argument is the source data frame. Then, you will need to specify what it calls an *aesthetic*: what do you want to show in x and y? We want our variable “n” to be shown as y, and we want “effect” and “cancer” on x.

This leads me to talk of *facets* and *formulas*. Facets are a `ggplot2` thing, while formulas are used throughout R. So basically, we want to divide our graph into several smaller graphs, one for each type of cancer, and show them side by side. So each “subgraph” is a facet. Therefore, we will use “cancer” in the facet function, and it leaves “effect” as the x in the `ggplot`'s aesthetic.

As for formulas, they define a more “talkative” way for you to specify which relationship you want R to study for you. The most basic form is $y \sim x$, which reads “y as a function of x”. More elaborate forms include $y + z \sim x$, which is “y and z as a function of x” (note, the plus sign means “and”, not a literal addition), and $y \sim x \mid a$, which is “y as a function of x, per value of a”.

With this preliminary knowledge, we are ready to look at the code, which produces figure 3.7.

```
> ggplot(forgg, aes(effect, n)) +
  # use data frame forgg, x=effect, y=n
  geom_col(aes(fill=sex)) +
  # draw columns, fill them according to sex
  facet_wrap(~cancer, nrow=1) +
  # subdivide as a function of cancer, on 1 row
```



```
coord_cartesian(ylim = c(0, 300000)) +
  # adjust y axis limits
scale_fill_manual(values = c("female"="#FF5577",
  ↪ "male"="#0000FF"))
  #set specific colors
```

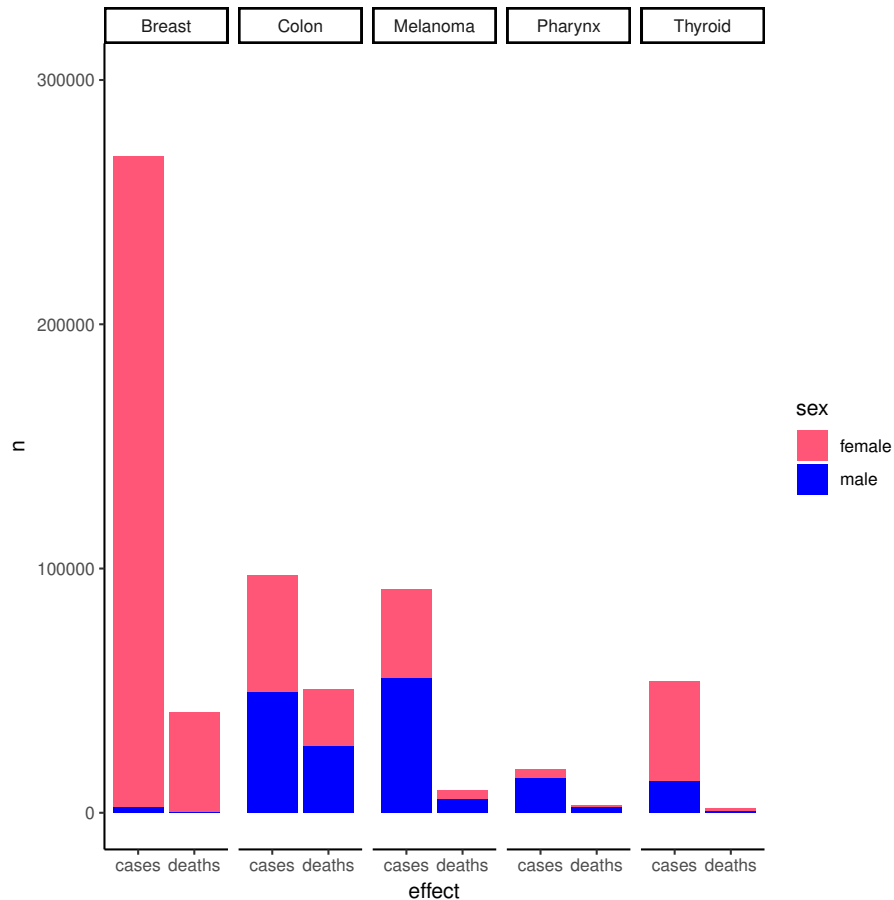


Figure 3.7: Number of cases and deaths by sex for five cancers, with the help of ggplot.

Your options of types of plots include: `geom_point` for points, `geom_col` for bars showing heights, `geom_bar` for bars showing counts.

You can control color and size of points, as well as fill of bars, by setting the appropriate options in the aesthetic, for example:

```
geom_point(aes(size=population, col=state))
geom_col(aes(fill=sex))
```

As you may have noted when we made our figure, the legend appears automatically, with the colors and sizes (if relevant) used.

You can use `facet_wrap` to make your "subgraphs" follow one after the other, or `facet_grid` to have them form a grid, for instance, see the code below, which produces figure 3.8. As for layout options regarding colors, titles, annotations, etc, they are so numerous that I can't detail them here. Digging into the help pages and asking the internet how to do things is the way to go.

```
> ggplot(forgg, aes(effect, n)) +
  geom_col(aes(fill=sex)) +
  facet_grid(cancer~sex, ) +
  scale_fill_manual(values = c("female"="#FF5577",
    ↪ "male"="#0000FF"))
```

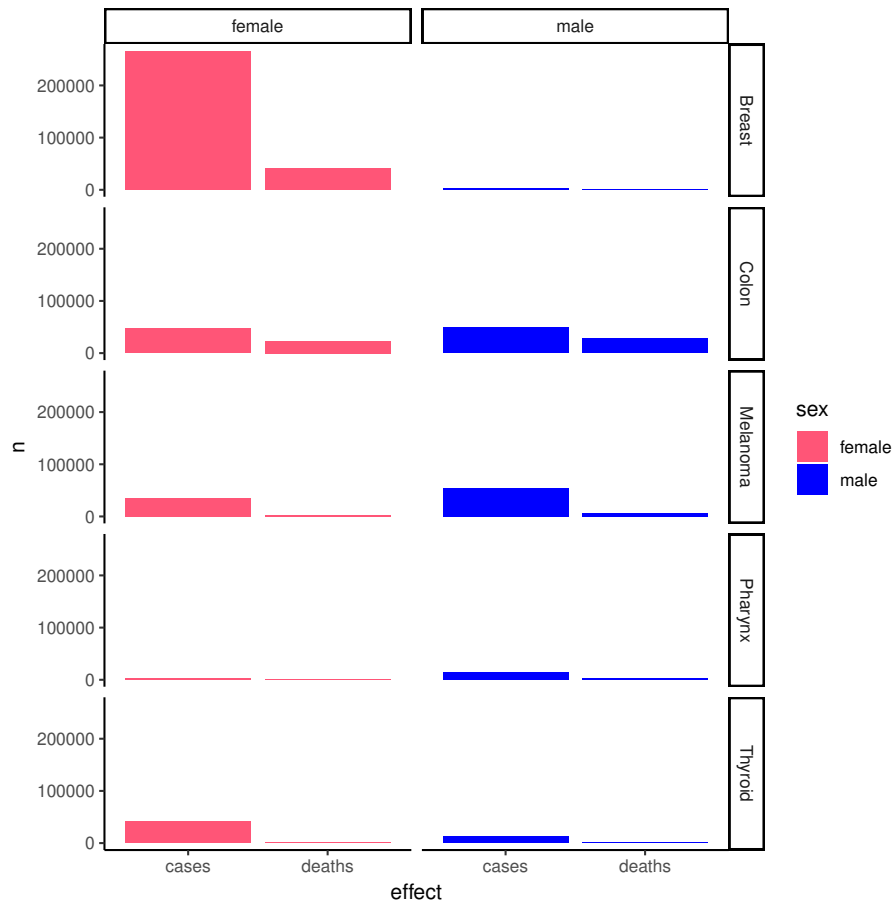


Figure 3.8: Grid presentation of number of cases and deaths per cancer per sex.

3.4 Histogram

For continuous variables

A histogram is a representation of a **continuous variable**, and *the bars touch*. It is used to visually present the shape of the distribution by compiling counts for “bins”, ranges of values. Bins are typically, but not necessarily, of equal width.

For our example, let’s use traffic data in the city of Chicago, with the import process described in section 2.4.4. A simple call to `hist`, will produce the figure 3.9.

```
> str(traffic)
'data.frame':  1279 obs. of  9 variables:
 $ ID      : Factor w/ 1279 levels "1","1,000","1,001",...
 $ Address : Factor w/ 1203 levels "1 West","10 East",...: 212 ...
 $ Street  : Factor w/ 251 levels "100th St","101st St",...
 $ Date    : Date, format: "2006-03-09" "2006-02-28" ...
 $ Volume  : int  14600 16500 18200 21600 18300 8600 10000 ...
 $ Direction: Factor w/ 1223 levels "East Bound: 1000 ..." ...
 $ Latitude : num  41.8 41.8 41.8 41.8 41.8 ...
 $ Longitude: num  -87.7 -87.7 -87.6 -87.6 -87.6 ...
 $ Location : Factor w/ 1276 levels "(41.651861, -87.54501)",...
> hist(traffic$Volume, xlab="Car volume", ylab="Count", main="",
  ↪ ylim=c(0, 600))
> box()
```

You will note that we fed `hist` the list of car volumes, and the function automatically computed the bins and counted how many observations were in each bin before plotting the resulting histogram.

There are several ways to compute bins for histograms, but the most used is the Sturges method, used by default in R, which outputs more bins the greater the range of the data (more or less) [8].

As you may have noted, there are several outliers at the right of the graph, with such small numbers that their bar is barely visible. I don’t necessarily recommend this course of action, since you change the representativity of your figure, but you could decide to group together the small bins at the end so they are visible. Let’s say you wanted to group together the bins at the right with five or less observations. Here, let’s point out a neat feature of R. In order to produce a plot, `hist` must compute bins, counts, etc., and it actually gives you access to them if you save the output of `hist`. You can then make use those to produce your personalized histogram, producing figure 3.10.

```
> h = hist(traffic$Volume, xlab="Car volume", ylab="Count",
  ↪ main="", ylim=c(0, 600))
> h
$`breaks`
[1]      0  10000  20000  30000  40000  50000  60000  70000
```

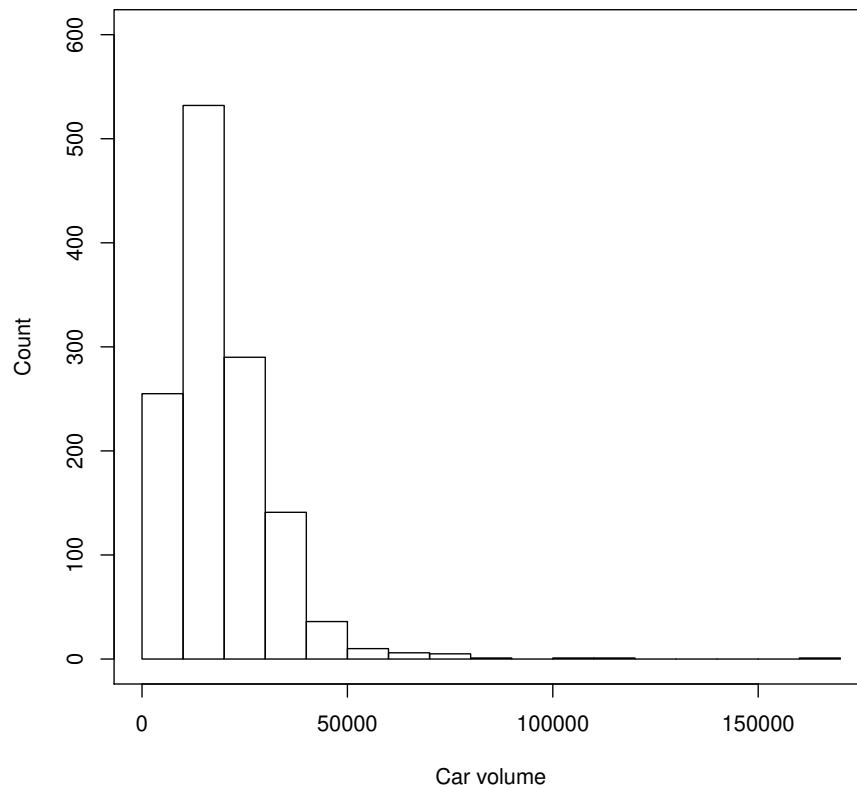


Figure 3.9: Histogram of car volumes in the city of Chicago.

```
[9] 80000 90000 100000 110000 120000 130000 140000 150000
[17] 160000 170000
```

\$counts

```
[1] 255 532 290 141 36 10 6 5 1 0 1 1 0 0
[15] 0 0 1
```

\$density

```
[1] 0.00001993745113 0.00004159499609 0.00002267396403
[4] 0.00001102423769 0.00000281469898 0.00000078186083
[7] 0.00000046911650 0.00000039093041 0.00000007818608
[10] 0.00000000000000 0.00000007818608 0.00000007818608
[13] 0.00000000000000 0.00000000000000 0.00000000000000
```

```

[16] 0.000000000000000 0.00000007818608

$mids
[1] 5000 15000 25000 35000 45000 55000 65000 75000
[9] 85000 95000 105000 115000 125000 135000 145000 155000
[17] 165000

$xname
[1] "traffic$Volume"

$equidist
[1] TRUE

attr("class")
[1] "histogram"
> h$breaks
[1] 0 10000 20000 30000 40000 50000 60000 70000
[9] 80000 90000 100000 110000 120000 130000 140000 150000
[17] 160000 170000
> h$counts
[1] 255 532 290 141 36 10 6 5 1 0 1 1 0 0
> mybreaks = c(h$breaks[1:7], 170000)
> mybreaks
[1] 0 10000 20000 30000 40000 50000 60000 170000
> h2 = hist(traffic$Volume, breaks=mybreaks, xlab="Car volume",
→ main="")
> h2
$`breaks`
[1] 0 10000 20000 30000 40000 50000 60000 170000

$counts
[1] 255 532 290 141 36 10 15

```

This figure has *density* as its y axis, meaning that the area (width*height) of all the bars sum to 1. This is the default behavior when using bins of different sizes. Note that your last bin, [60000, 170000[, contains 15 observations, more than the previous bin, while appearing lower in the figure; this is due to the density calculations. The *area* of the bar is representative of its data, not the *height*.

3.5 Scatter plot

For continuous variables

This figure is used to illustrate the relationship between two **quantitative variables**. Putting aside correlation-is-not-causation for the time being, let's

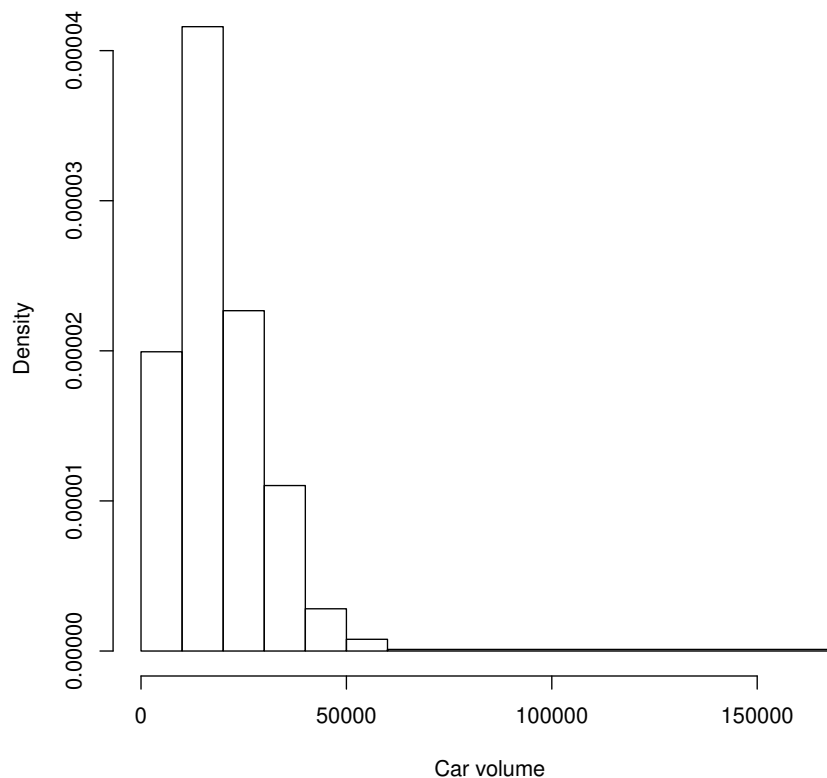


Figure 3.10: *Density* histogram of car volumes in the city of Chicago.

study the proportion of diabetes and of people self-reporting no exercise in the Community Health Status Indicators study [3].

The “CHSI Dataset” file will inform you that the file “Risk factors and access to care” holds the variables we’re interested in. Moreover, -9999, -2222, -2222.2, -2, -1111.1, -1111 and -1 are all various forms of missing data. This data is pretty clean, so loading it into R is straightforward.

```
> chsi = read.csv("C:/.../r-tutorial/chsi.csv",
+ na.strings=c(-9999, -2222, -2222.2, -2, -1111.1, -1111, -1))
> str(chsi)
'data.frame':  3141 obs. of  31 variables:
 $ State_FIPS_Code      : int   1 1 1 1 1 1 1 1 1 1 ...
 $ County_FIPS_Code     : int   1 3 5 7 9 11 13 15 17 19 ...
```

```

$ CHSI_County_Name      : Factor w/ 1847 levels
  ↪ "Abbeville","Acadia",...: 82 89 100 150 165 226 236 247 ...
$ CHSI_State_Name       : Factor w/ 51 levels
  ↪ "Alabama","Alaska",...: 1 1 1 1 1 1 1 1 1 ...
$ CHSI_State_Abbbr      : Factor w/ 51 levels
  ↪ "AK","AL","AR",...: 2 2 2 2 2 2 2 2 2 ...
$ Strata_ID_Number      : int   29 16 51 42 28 75 76 6 50 ...
$ No_Exercise           : num   27.8 27.2 NA NA 33.5 NA ...
$ CI_Min_No_Exercise    : num   20.7 23.2 NA NA 26.3 NA ...
$ CI_Max_No_Exercise    : num   34.9 31.2 NA NA 40.6 NA ..
$ Few_Fruit_Veg        : num   78.6 76.2 NA 86.6 74.6 NA ...
$ CI_Min_Fruit_Veg     : num   69.4 71.2 NA 77.8 66.1 NA ...
$ CI_Max_Fruit_Veg     : num   87.8 81.3 NA 95.4 83 NA ...
$ Obesity               : num   24.5 23.6 25.6 NA 24.2 NA ...
$ CI_Min_Obesity       : num   17.3 19.5 16.2 NA 17.2 NA ...
$ CI_Max_Obesity       : num   31.7 27.6 35 NA 31.2 NA ...
$ High_Blood_Pres      : num   29.1 30.5 NA NA NA NA NA ...
$ CI_Min_High_Blood_Pres : num   19.2 24.5 NA NA NA NA NA ...
$ CI_Max_High_Blood_Pres : num   39 36.6 NA NA NA NA NA ...
$ Smoker               : num   26.6 24.6 17.7 NA 23.6 NA ...
$ CI_Min_Smoker        : num   19.1 20.3 10.2 NA 16.7 NA ...
$ CI_Max_Smoker        : num   34 28.8 25.1 NA 30.4 NA ...
$ Diabetes             : num   14.2 7.2 6.6 13.1 8.4 NA ...
$ CI_Min_Diabetes      : num   9.1 5.2 2 4.7 4.4 NA 4.4 ...
$ CI_Max_Diabetes      : num   19.3 9.3 11.3 21.5 12.4 ...
$ Uninsured            : int   5690 19798 5126 3315 8131 ...
$ Elderly_Medicare     : int   4762 22635 3288 2390 5019 ...
$ Disabled_Medicare    : int   1209 3839 1092 974 1300 ...
$ Prim_Care_Phys_Rate  : num   45.3 67 45.8 41.8 16.2 ...
$ Dentist_Rate         : num   22.6 30.8 24.6 18.6 10.8 ...
$ Community_Health_Center_Ind: int 1 1 1 1 2 1 1 1 1 2 ...
$ HPSA_Ind             : int   2 2 2 1 1 1 2 2 2 1 ...

```

This file holds data for all the counties in the US. Using every county as a data point, plotting diabetes as a function of lack of exercise is easy. The code below produces figure 3.11.

```

plot(chsi$Diabetes ~ chsi$No_Exercise, pch=20,
     ylab="Proportion of diabetes",
     xlab="Proportion of inactive adults",
     main="Diabetes and inactivity for every county in the
     ↪ US")

```

You may notice that we used the “formula” way of specifying what we want to plot, like we did in section 3.3. Note that `plot(x, y)` also works and would produce the same thing.

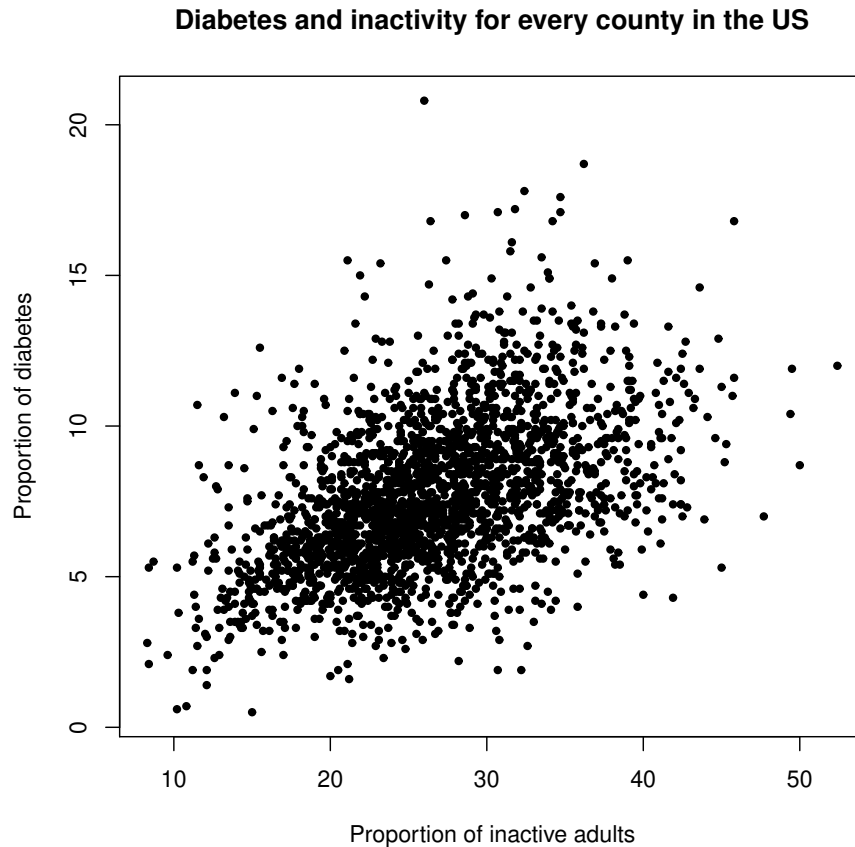


Figure 3.11: Scatter plot of diabetes and inactivity for US counties.

As you can see, the higher the proportion of physically inactive people in a county, the higher its rate of diabetes. While correlation does not necessarily mean that one causes the other, there is *definitely* a relationship between the two. Without discounting the wealth of research that says how bad inactivity is bad for your blood sugar levels, one cannot conclude on the basis of a graph like this that inactivity itself causes diabetes. For instance, diabetes *and* inactivity could both be caused by a third factor, say, living in a typical US town, where food is overabundant, road infrastructure is designed with cars in mind, and urbanism plans do not favor active transport.

Beyond that, one should also pay attention to the data itself. Here, we consider every county equally. That is, a county of 102 inhabitants in the sticks of Montana is one dot on the graph, the same way as a county of 40 000 Californians. This is a case of *misrepresentation* of data; you give one equal dot to each

county, regardless of the number of its residents, or the ratio of people who gave data to the CHIS study. To better represent this, you could easily use `ggplot2` to make the size or the transparency of each dot a function of the population of each county.

See also section 9.1 for a way to plot a more solid dataset, the NHIS raw data, which contains data on individuals, chosen to be representative of the US population as a whole. Since this data presents diabetes as a “yes/no” variable, a different, less common technique is used: the conditional-density plot.

3.6 Box and whiskers graph

For a quantitative variable

This graph, while a visual representation of data, offers a nice bridge to the next section, since it does a lot of *description* of the data.

Let’s say you wanted to see if health insurance in the US varied a lot by state. You of course could make a map of this data (I promise you can, see section 9.3), but this would not give you a good picture of the *variability* within the country. Enter the box-and-whiskers graph.

Let’s use the 2014 US Census Bureau data about health insurance per country graciously publicized by Wikipedia [7]. This clean data is easily copy-pasted into a Spreadsheet, saved as csv and loaded in R.

```
> hic = read.csv("C:/.../r-tutorial/hic.csv")
> str(hic)
'data.frame':   51 obs. of  17 variables:
 $ State: Factor w/ 51 levels "Alabama","Alaska",...: 1 2 3 4 5 ...
 $ X1999: num   12 18.3 19.4 13.9 19 14.1 7.3 9.7 14 17.4 ...
 # ...
 $ X2014: num   12.1 17.2 13.6 11.8 12.4 10.3 6.9 7.8 5.3 16.6 ...
```

Then, making a box-and-whiskers graph is easy:

```
> boxplot(hic$X2014, ylim=c(0, 20),
+ ylab="Percent of uninsured persons",
+ main="Uninsured rate for US states")
```

The lower and upper “whiskers” represent the minimum and maximum values found in your dataset, so in our case, between roughly 3 and 20% in 2014. The “box” represents 25, 50 and 75% of data points, so the darker middle line is the median, at around 10%. Since the box is rather squished vertically between long whiskers, you can conclude that most states are near the median, with fewer outliers stretching further out.

To showcase a bit more of the possibilities of the boxplot, let’s look at all the years together. To do this, we’ll need to tidy up the data with a good use of `gather`, and then remove the “X” in front of the year number.

```
> library(tidyr)
> hic = gather(hic, key="year", value="rate", 2:17)
```

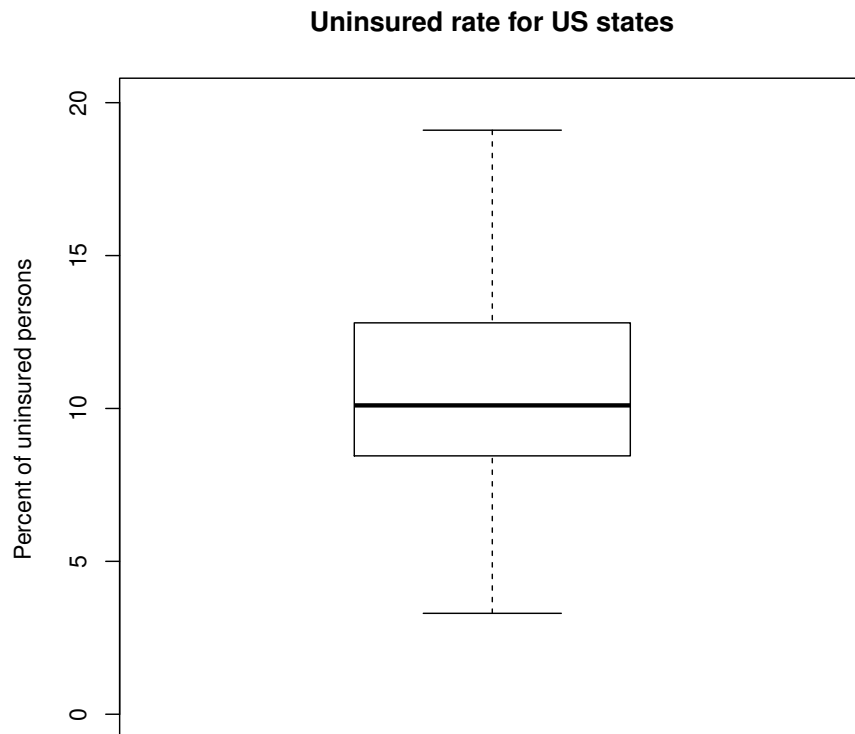


Figure 3.12: Boxplot of the uninsured rate among US states.

```
> hic$year = as.integer(substring(hic$year, 2, 5))
> str(hic)
'data.frame': 816 obs. of 3 variables:
 $ State: Factor w/ 51 levels "Alabama","Alaska",...: 1 2 3 4 5 ...
 $ year : int  1999 1999 1999 1999 1999 1999 1999 1999 1999 1999 ...
 $ rate : num  12 18.3 19.4 13.9 19 14.1 7.3 9.7 14 17.4 ...
> boxplot(hic$rate ~ hic$year, ylim=c(0, 30),
+ ylab="Percent of uninsured persons",
+ main="Uninsured rate for US states")
```

As you can see in figure 3.13, there is sometimes a single dot above or below a whisker, indicating that this lonely data point is sufficiently far away from the others to be plotted on its own. This behavior can be controlled with the `range` option. By default, points that are $>1.5 \times$ “height of the box” are plotted

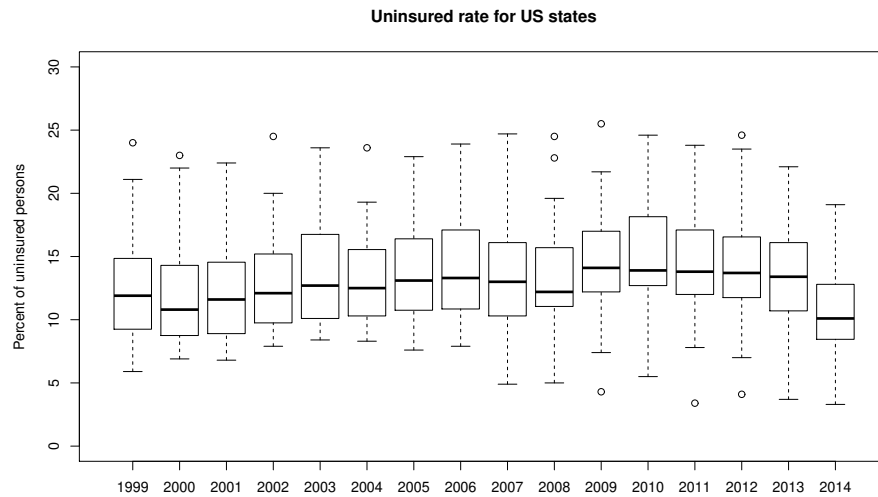


Figure 3.13: Boxplot of the uninsured rate among US states between 1999 and 2014.

alone. These lonesome outliers seem to become less common in the last few years of the data set, and the maxima and minima are both becoming lower. Overall range also seems to shrink, which means states are more similar since the adoption of a federal statute on health insurance.

To bridge even further ahead to the section about actual statistics, let's underline that this is an *impression*. How big must the change in range be for us to conclude that the states are really becoming more similar? That is the kind of question for inferential statistics.

3.7 Putting it all together: summary figure



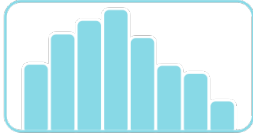



	Categorical Data	Continuous Data
Proportions	 Table or pie chart	
Frequencies	 Bar chart	 Histogram
Values	 Dot plot (rarely used)	 Scatter plot
Summary	 Box and whiskers plot	

Figure 3.14: When to plot what how and why.

Chapter 4

Data description

Data is typically described regarding three characteristics: where is the “middle” of the distribution (center tendency), how “spread out” is it (dispersion), and what “shape” is it.

4.1 Center tendency measurements

4.1.1 Mean

Tons of pleasant math characteristics

Everyone knows what a mean, also called average, is, if only for its widespread use. It is the sum of all observations, divided by the number of observations. Formally, this is:

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

...where \bar{x} is the mean, x_i is each observation, and n is the number of observations.

While the basic principle is very simple, there are a few subtleties that are worth pointing out about the mean. Since salaries are a classic example here, let’s use those of the employees of the Athletics department of Michigan University [20]. And since I can’t imagine why he’s not in the list, let’s add the head coach, Jim Harbaugh, at 7,004,000\$ [22]. Copy-paste the table into a spreadsheet, remove empty lines, add Mr. Harbaugh, do the usual number format tweaks, then load into R. Calculating the average is a simple case of calling up the function `mean`.

```
> umsalaries =  
  ↪ read.csv("C:/.../r-tutorial/um_athletics_salaries.csv")  
> str(umsalaries)  
'data.frame': 369 obs. of 4 variables:
```

```

$ Name      : Factor w/ 369 levels "Ablauf, David S",...: 137 ...
$ Title     : Factor w/ 132 levels "Academic Advisor/Couns ..."
$ Department: Factor w/ 1 level "Athletics": 1 1 1 1 1 1 ...
$ Salary    : int  7004000 1000000 1000000 1000000 842550 ...
> mean(umsalaries$Salary)
[1] 118022.5

```

Note that if some employees had a missing salary, you can determine what to do with missing data with the argument `na.rm`.

Now, as you saw, calculating the mean is easy enough with R. However, to understand what the average entails, let's take a closer look at the data. The following code lead to the histogram in figure 4.1.

```

> options(scipen=999)
> h = hist(umsalaries$Salary,
          xlab="Salaries", xlim=c(0, 8000000),
          ylim=c(0, 400),
          main="Salaries of the Athletics department of Michigan
              ↪ University")
> abline(v=118022.5, col="red")
> box()
> h
$`breaks`
[1] 0 500000 1000000 1500000 2000000 2500000 3000000
[8] 3500000 4000000 4500000 5000000 5500000 6000000 6500000
[15] 7000000 7500000

$counts
[1] 363 5 0 0 0 0 0 0 0 0 0 0 0 0 1

#...

```

If you remember, our mean came out as 118,022.50\$. If you take a second look at the histogram now, that is the red vertical line, near the left end of the distribution. If the goal of an average is to tell you where the “middle” of a distribution is, you have to wonder where the “middle” of such an asymmetric histogram actually is and whether it means anything at all. Since the mathematic formula of the mean spreads the sum of all values equally over all of them, you can think of it as the *center of mass* of the distribution. Since a lot of low values reside to the left of the red line, and a few high values far to the right, the red line is the “point of equilibrium” of this mass.

While I will not contest that the mean is a useful metric to give you an idea of the “middle” of a list of values, I wish to make the point that the usefulness of the average is limited when the distribution is asymmetrical.

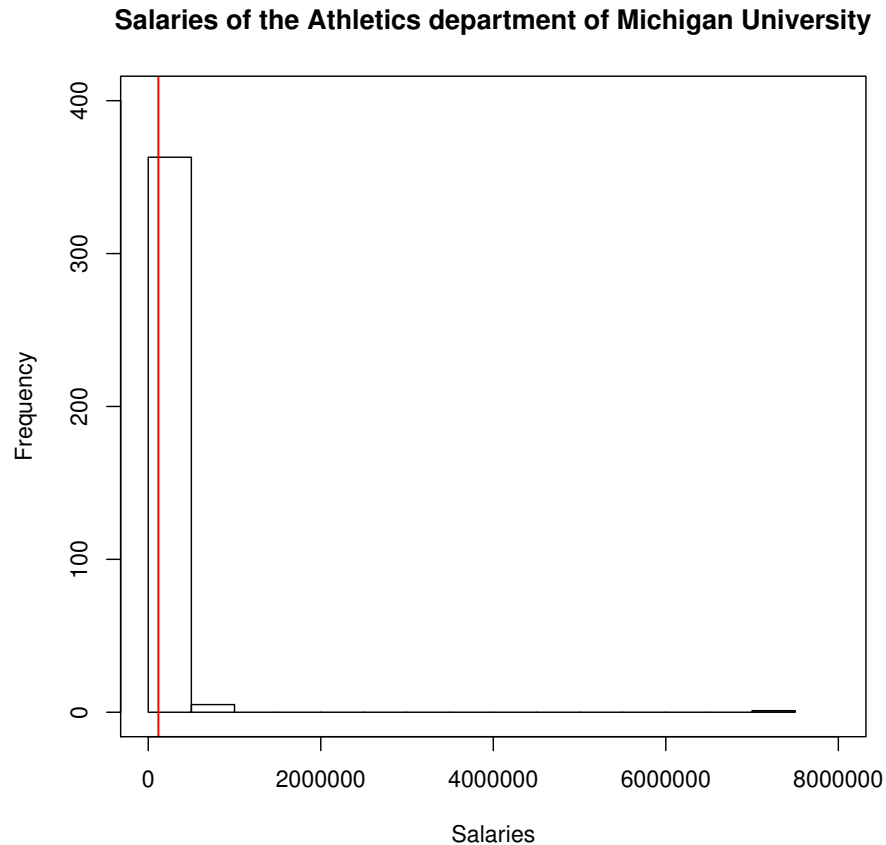


Figure 4.1: Michigan University salaries.

Standard error

Now, let's look at the mean from a different perspective. Let's say you are interested by the nose-to-tail length of one specific species of fish in one particular lake. Let's further assume that it's a small species of fodder fish where all sizes can be captured using one type of trap cage.

(If you were interested in large fishes like cod, for instance, the size of the net you choose would *de facto* eliminate smaller individuals who can slip through. Did I mention experimental design is super important?)

Let's say it's a super endangered species called *Raropiscus virtualis* with a grand total of 100 individuals left in the world, whose sizes in millimeters are shown below.

```
> fishies
```

```

[1] 6.653727 8.091868 8.522104 9.038613 9.177935 10.537482
[7] 10.788898 11.169217 11.250873 11.272964 11.524758 11.832570
[13] 11.837743 11.934911 12.027187 12.283124 12.388730 12.554695
[19] 12.586544 12.627581 12.675485 12.941469 13.031906 13.063111
[25] 13.067909 13.123860 13.236292 13.396793 13.405203 13.466648
[31] 13.540592 13.542691 13.602745 13.895082 13.898869 13.958740
[37] 14.124031 14.153446 14.263436 14.284507 14.301187 14.415875
[43] 14.601420 14.618754 14.798551 14.822237 14.910379 14.939965
[49] 14.941777 14.987264 15.016763 15.053211 15.170931 15.450270
[55] 15.492681 15.522237 15.601398 15.609749 15.704461 15.711137
[61] 16.210493 16.251649 16.472377 16.498656 16.663045 16.794268
[67] 16.852466 16.924979 17.013302 17.338347 17.395273 17.443367
[73] 17.444365 17.506961 17.526659 17.639866 17.814593 18.007851
[79] 18.068755 18.108487 18.346849 18.446874 18.508666 18.551282
[85] 18.683468 18.753444 18.763917 18.804852 18.969312 19.331139
[91] 19.391159 19.408782 19.645286 19.998824 20.197807 20.576269
[97] 20.784767 20.922783 21.389299 22.962449
> mean(fishies)
[1] 15.26858

```

So that value of 15.26858 mm is the *real* average size of *R. virtualis* individuals. However, if you go out in the field, put your trap cages down, and wait, you might catch ten of the individuals, and if your experimental design is smart, they will be picked at random, i.e., there is *no bias* in your sampling. It would be perfectly reasonable for you to then compute the mean of this sample and say, this is a good *estimate* of the average size of *all R. virtualis*. Let's see what that might mean:

```

> sample = sample(fishies, 10)
> sample
[1] 14.41587 15.60140 13.03191 16.21049 14.15345 16.66304
[7] 13.06311 13.12386 10.53748 22.96245
> mean(sample)
[1] 14.97631

> #... But if you come back the next day and do the same
> # thing again, you might get:
> sample = sample(fishies, 10)
> sample
[1] 12.28312 13.60275 11.16922 14.61875 13.06311 18.75344
[7] 17.44337 18.10849 13.06791 19.64529
> mean(sample)
[1] 15.17554

> # And one day, you might even pick the ten smallest
> # fishes in the lake, and then you'd get:

```



```

> sample = fishies[1:10]
> sample
[1] 6.653727 8.091868 8.522104 9.038613 9.177935 10.537482
[7] 10.788898 11.169217 11.250873 11.272964
> mean(sample)
[1] 9.650368

```

The point here is that if you *sample* individuals from a population, the mean of the sample is not equal to the mean of the population. It is dependent on the effects of random sampling. And you better hope that sampling is unbiased, or your sample mean will be even less representative of the population mean.

But the first two samples look pretty good, right? So, judging from those samples, you have a good idea of the average size of fishes in the population. But if you pick individuals at random, once in a blue moon, you *will* get a ridiculous sample like the third one. Keep repeating an experiment, and the unlikely will occur.

(Just ask your gamer friends, they'll have something to say about that time when 23 tries of something with a theoretical rate of 54% yielded 0 successes. They'll be understandably frustrated but they just might be the one among 12 million players it happened to. Not speaking from personal experience, ahem.)

If you keep repeating the sampling, something a little mind-bending happens: the sample means has a distribution of its own. So let's use R to virtually set your cages in the lake and randomly pick 10 individuals out of the population and compute the mean of that sample, and repeat the whole operation 200 times. The new bit of code, the `for` loop, is not the point here, and it will be explained at length in section 8.1.1. The interesting part is figure 4.2.

```

means = numeric(200) #create empty vector with 200 zeros in it

for(i in 1:200) {
  sample = sample(fishies, 10)
  means[i] = mean(sample) # save the sample mean
}
hist(means,
      xlab="Sample mean", xlim=c(12, 18),
      main="")

```

As you can see, sample averages of around 15, which is the population average, happen most often; the *average of sample averages* is the population average. Sample average follows a classic bell-shaped distribution.

Obviously, the bigger your sample, the better your estimate. If you put your cages down and wait until you've caught 100 different *R. virtualis*, obviously, you will have the perfect mean. At the opposite side of the spectrum, if you capture only one individual and take that as a population mean, your error is likely to be much greater.

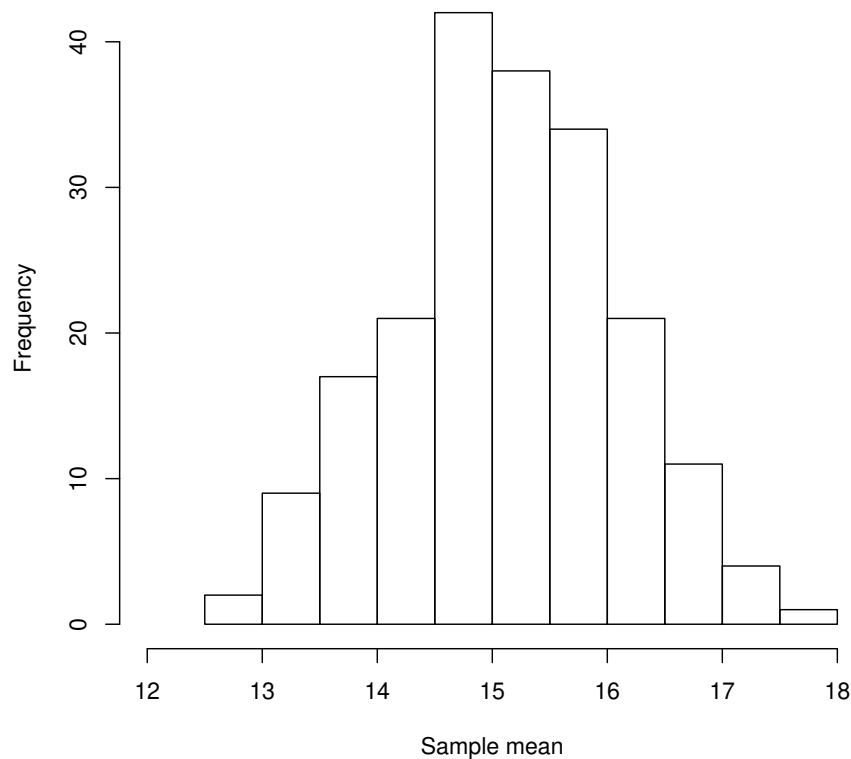


Figure 4.2: Means of 200 samples of 10 fishes from the population of 100 fishes.

Again using R programming that is not the point here, let's virtually sample our imaginary population repeatedly. Let's test all sample sizes from 1 to 100 individuals, and repeat each sample size 1000 times. Further, let's use `absolute(sample mean - population mean)` as a measure of error, that is, the distance between the sample and population means. As you might expect, your average error goes down quickly as you pick a bigger sample size, as seen in figure 4.3.

```
real_mean = mean(fishies)

error = numeric(100)
tmpe = numeric(1000)

for(n in 1:100) {    # for each sample size
```

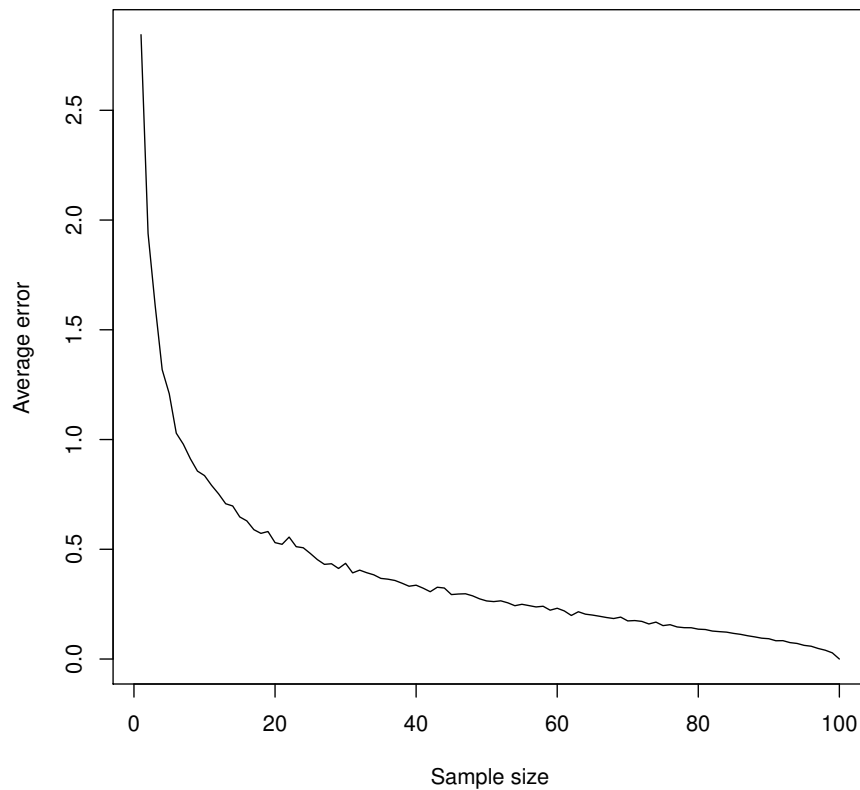


Figure 4.3: Average error for 1000 samplings for every possible sample size.

```

for(i in 1:1000) { # for 1000 repetitions
  sample = sample(fishies, n) # sample n fishes out of the 100
  tmpe[i] = abs(mean(sample)-real_mean) # compute error for
  ↪ this time
}
error[n] = mean(tmpe) # average all errors for this sample size
}
plot(x=1:100, y=error, type="l",
     xlab="Sample size",
     ylab="Average error")

```

So what we have just done is measure the *precision* of the sample mean as a function of sample size. Evidently, that's a very useful metric. However, in real life, you won't be going back to your lake and repeating every sample size

1000 times. (Unless you’re a monk, figuratively, or literally, like Mendel, who grew 10,000 pea plants in 8 years, hand-pollinating every pair [5].) What you will do is take *one* sample, and get *one* value of mean, and the error could be any point for that sample size in figure 4.4. Since you don’t know the “real mean” of the population, you can’t compute your actual error, but if you could get the value of the red line, that is, the average error for that sample size, you’d be in business.

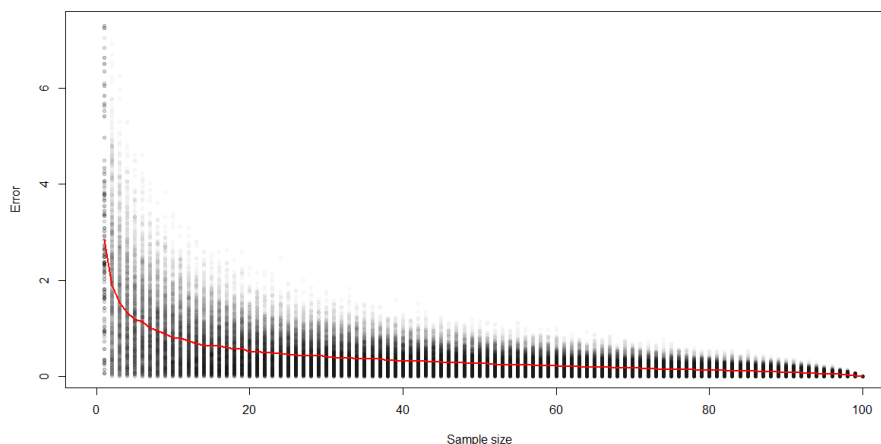


Figure 4.4: Sample errors for 1000 repetitions of each sample size.

Thankfully for you, ordinary-person-who-only-uses-not-makes-statistics (I’m in there too, don’t worry), some smart mathematicians came up with a solution! Let’s all hug a mathematician today, they’re a treasure.

This general measure of the precision of a sample mean is called the *standard error*, (not to be confused with the *standard deviation*, which we will see in section 4.2). The formula for the standard error is as follows:

$$s_{\bar{x}} = \frac{s}{n}$$

... where $s_{\bar{x}}$ is the standard error, s is the standard deviation and n is the sample size. Glossing over the definition of standard deviation for now, this formula allows you to compute the precision of the sample mean with a single sample. Of course, if you repeat the sampling process, you will get different values of standard error, since this value is computed with whatever you get out of your random sampling. However, contrary to the “distance” error shown on figure 4.4, the standard error is pretty consistent for reasonable sample sizes (i.e. not tiny samples). As a demonstration, figure 4.5 shows the standard errors computed from 1000 samplings of each sample size.

There is no function to compute the standard error in R, but it’s a simple matter of dividing the standard deviation by the sample size, like so, for instance:

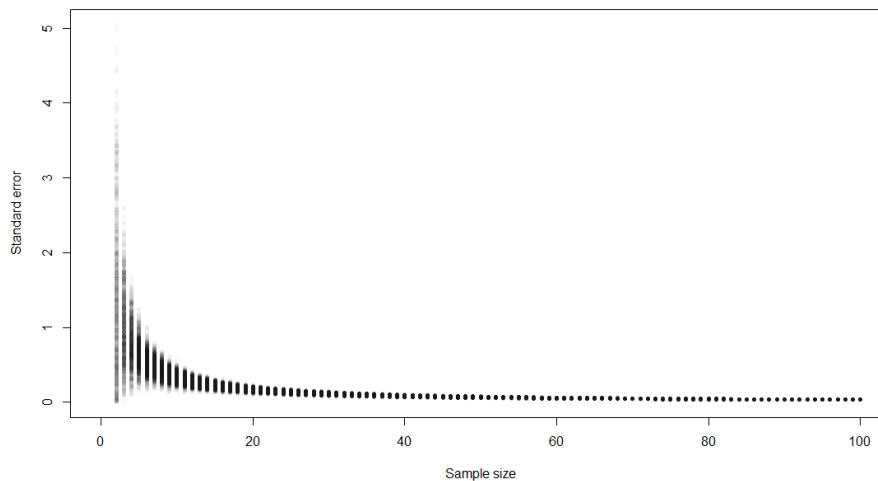


Figure 4.5: Standard errors for 1000 repetitions of each sample size.

```
> sample
[1] 17.230860 11.661620 16.713139 13.267614 17.825505 17.048323
[7] 15.843759 12.580526 15.479187 12.532484 14.751772 12.722589
#...
[97] 18.553211 18.877419 16.514153 14.546831
> n
[1] 100
> se = sd(sample)/n
> se
[1] 0.0340778
```

4.1.2 Median

Arguably the best central tendency measurement

Rewinding to the discussion of the mean's sensitivity to extreme values leads in perfectly to the second measure of central tendency, the median. The median is literally the middle value of a distribution; if you sort all the values in order, the one in the middle is the median. (Technically, if you have an odd number of observations, the median is the middle value; if you have an even number of observations, the median is the average of the two middle values.)

In R, calculating the median is a simple case of calling the appropriate function, `median`.

Compared to the mean, the median is less sensitive to extreme values, since it uses the *rank* rather than the *value* of the list of numbers. To contrast the median and the mean, let's go back to the salaries of employees of the Athletics department at Michigan University. If we zoom in to the salaries that are most

common, that is, under 600,000\$, we can compare the value of the mean (in red) and the median (in blue). The code below yields figure 4.6.

```
> median(umsalaries$Salary)
[1] 65000
> real_salaries = umsalaries$Salary[umsalaries$Salary < 600000]
> # ^ that's filtering, see 2.3
> hist(real_salaries, xlab="Salary",
+ xlim=c(0, 600000), ylim=c(0, 200), main="")
> abline(v=118022.5, col="red")
> abline(v=65000, col="blue")
```

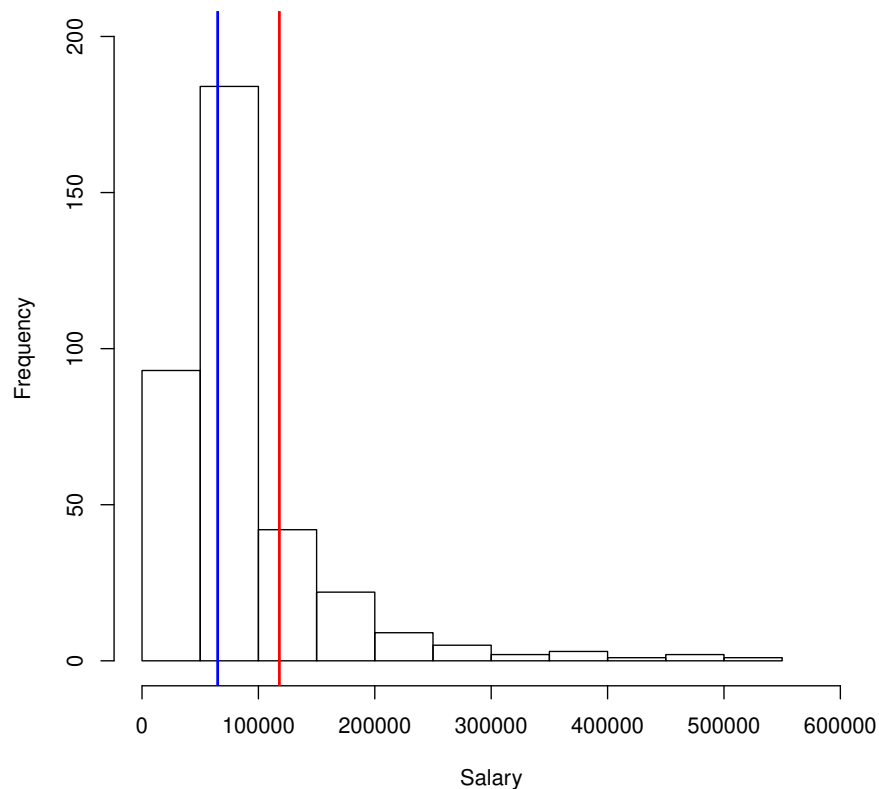


Figure 4.6: Common salaries at the Athletics department of Michigan University.

The blue line looks closer to the “middle” than the red line, doesn’t it? The

red line has been pulled towards the extreme high value of the head coach at 7 million\$, while the blue line hasn't. If you are looking to *describe* a distribution, I argue that the median will give you a better idea of the central tendency than the mean for this reason. However, the mean has mathematical properties that give it more applications in the realm of inferential statistics, as we will see in chapter 7.

4.1.3 Mode

Useful if you want to guess

4.2 Dispersion measurements

4.2.1 Range

4.2.2 Variance

Tons of pleasant math characteristics

4.2.3 Standard deviation

4.2.4 Coefficient of variation

4.2.5 Quartiles and percentiles

4.3 Shape measurements

4.3.1 Skewness

Symmetry

4.3.2 Kurtosis

Flatness

4.3.3 L-moments

Chapter 5

Probabilities

5.1 Factorial

5.2 Combinations

5.3 Permutations

5.4 Probability Mass/Density Function

Chapter 6

Statistics

6.1 Discrete variables

6.1.1 Binomial distribution

6.1.2 Multinomial distribution

6.1.3 Poisson distribution

6.1.4 Inverse binomial distribution

6.1.5 Hypergeometric distribution

6.2 Continuous variables

6.2.1 Normal distribution

6.2.2 Exponential distribution

6.2.3 Gamma distribution

6.2.4 χ^2 distribution

6.2.5 Fisher-Snedecor distribution

6.2.6 Student's law

Chapter 7

Inferential statistics

- 7.1 Student's test
- 7.2 Student's paired test
- 7.3 Bartlett's test
- 7.4 Single-factor ANOVA
- 7.5 χ^2 test
- 7.6 Wilcoxon-Mann-Whitney test
- 7.7 Kolmogorov-Smirnov test
- 7.8 Kruskal-Wallis test
- 7.9 Pearson's test
- 7.10 Spearman's test
- 7.11 Kendall's test
- 7.12 Simple linear regression
- 7.13 Multiple linear regression

Chapter 8

Programming

8.1 Sequence, iteration, branching

8.1.1 Iteration

8.1.2 Branching

8.2 Functions

8.3 Misc

8.3.1 Real saga

8.3.2 Rounding

```
round(.5 + -2:4) # IEEE / IEC rounding:  
↪ -2 0 0 2 2 4 4
```

Chapter 9

More eccentric graphs

9.1 Rate plot

For a categorical variable as a function of a continuous variable As an incendiary correlation-is-not-causation example, let's use the National Health Interview Survey data to study the relationship between type II diabetes and body mass index [11]. The Sample Adult file has the information we'll use. The Variable summary file will inform you that the column labels that interest you are DIBEV1 (diagnosis of diabetes), DIBTYPE (diabetes type) and BMI (body mass index). The variable layout will inform you of the meaning of the values in the file.

```
For diabetes diagnosis:
1  Yes
2  No
3  Borderline or prediabetes
7  Refused
8  Not ascertained
9  Don't know
Type of diabetes
1  Type 1
2  Type 2
3  Other
7  Refused
8  Not ascertained
9  Don't know
```

And BMI is the number you'd expect multiplied by 100.

For once, simply loading the csv into R gives sensible results. I will spare you the sanity check on import integrity, given the number of variables in the file. Let's just remove all the columns we're not interested with at the moment, divide BMI, and correctly create factors from the integer codes for the other two. Then let's take a quick look at sample data:

```

> nhis=read.csv("C:/.../r-tutorial/samadult.csv")
> nhis$BMI = nhis$BMI/100
> nhis = as.data.frame(cbind(nhis$BMI, nhis$DIBTYPE,
  ↪ nhis$DIBEV1))
> colnames(nhis)=c("BMI", "Type", "Diagnosis")
> nhis$Type = factor(nhis$Type,
+ levels=c(1, 2, 3, 7, 8, 9),
+ labels=c("Type I", "Type II", "Other", "Refused", "Not
  ↪ Ascertained", "Don't know"))
> nhis$Diagnosis = factor(nhis$Diagnosis,
+ levels=c(1, 2, 3, 7, 8, 9),
+ labels=c("Yes", "No", "Borderline or prediabetes",
  ↪ "Refused", "Not Ascertained", "Don't know"))
> str(nhis)
'data.frame': 26742 obs. of 3 variables:
 $ BMI : num 29.3 23.1 35.4 43.1 32.3 ...
 $ Type : Factor w/ 6 levels "Type I","Type II",...: NA
  ↪ NA ...
 $ Diagnosis: Factor w/ 6 levels "Yes","No", ...
> nhis[1:10,]
  BMI Type Diagnosis
1 29.30 <NA> No
2 23.09 <NA> No
3 35.44 <NA> No
4 43.13 <NA> No
5 32.27 <NA> No
6 23.25 <NA> No
7 24.67 <NA> No
8 41.97 Type II Yes
9 33.51 Type I Yes
10 26.62 <NA> No

```

Now, to simplify our analysis and our figure, let's just concern ourselves with the diagnosis values of yes or no and diabetes type II. To do this, I will use the `filter` function. Since R has two `filter` functions, one in the `stats` and the other in the `dplyr` package, I will manually specify which one I want. `filter` works by giving it some data, then specifying a condition, and it returns the rows where the condition is verified.

In our case, we want all the cases where the diagnosis is “No” (in that case, diabetes type is NA). We also want rows where diagnosis is “Yes”, but only if their value of type is II. Verbally, it is:

```

Diagnosis = No
OR
(Diagnosis = Yes AND Type = II)

```

Note that the parenthesis are important, even though it would not appear

to make a difference with our data. Instead, to demonstrate, let's consider this table:

Name	Sex	Eyes
Sarah	Female	Green
James	Male	Blue
Nicholas	Male	Brown

Now, let's compare the following parentheses placement:

```
Female OR (Male AND blue)
=> (Sarah) OR (James) => Sarah, James
(Female OR Male) AND blue
=> (Sarah, James, Nicholas) AND (blue) => James
```

This kind of statement is called boolean algebra, and it definitely needs some getting used to. Moreover, in R, when using “=” to mean a comparison rather than the assignment of a value to a variable, you must actually type “==”. | means “or” and & means “and”. A more in-depth look at booleans and conditions is given in section 8.1.2. This leads to the following code:

```
> library(dplyr)
> nhis = filter(nhis, Diagnosis=="No" | (Diagnosis=="Yes"
  ↳ & Type=="Type II"))
> nhis[1:10, ]
  BMI      Type Diagnosis
1  29.30  <NA>         No
2  23.09  <NA>         No
3  35.44  <NA>         No
4  43.13  <NA>         No
5  32.27  <NA>         No
6  23.25  <NA>         No
7  24.67  <NA>         No
8  41.97 Type II       Yes
9  26.62  <NA>         No
10 22.61  <NA>         No
```

Now, our goal is to plot rate of diabetes in relation to BMI. To do so, we will have to compute this rate; the very word *rate* implies several individuals used to calculate a proportion, so we will have to group the adults present in the file. To do this, we will make use of `hist` again.

9.2 Violin

A better box and whiskers

9.3 Map

For any variable as a function of a geographic variable

Chapter 10

Cheat sheet

10.1 Plumbing

<code>?</code>	<code>?exact_function_name</code>
<code>??</code>	<code>??keyword</code>
<code>class</code>	<code>class(R_variable)</code>
<code>str</code>	<code>str(R_variable)</code>
<code>colnames</code>	<code>colnames(R_variable)</code>
<code>as.integer</code>	<code>as.integer(R_variable)</code>
<code>rbind</code>	<code>rbind(var, var...)</code>
<code>cbind</code>	<code>cbind(var, var...)</code>

10.2 Data import and export

<code>read.csv</code>	<code>read.csv('delimited_data.csv', header=TRUE, sep=",", dec=".")</code>
<code>read.fwf</code>	<code>read.fwf('fixed_width_data.txt', widths=c(10, 5, 4), header=FALSE, skip=2, strip.white=TRUE, comment.char="#")</code>
<code>write.csv</code>	<code>write.csv(R_variable, file='desired_file_name.csv', row.names=FALSE, append=FALSE)</code>

Bibliography

- [1] *Average Daily Traffic Counts | City of Chicago | Data Portal*. URL: <https://data.cityofchicago.org/Transportation/Average-Daily-Traffic-Counts/pfsx-4n4m> (visited on 09/26/2018).
- [2] *Calendar date*. en. Page Version ID: 856764544. Aug. 2018. URL: https://en.wikipedia.org/w/index.php?title=Calendar_date&oldid=856764544 (visited on 09/27/2018).
- [3] *Community Health Status Indicators (CHSI) to Combat Obesity, Heart Disease and Cancer - Data.gov*. URL: <https://catalog.data.gov/dataset/community-health-status-indicators-chsi-to-combat-obesity-heart-disease-and-cancer> (visited on 09/30/2018).
- [4] Earl F. Glynn. *Chart of R Colors*. Apr. 2005. URL: <http://research.stowers.org/mcm/efg/R/Color/Chart/> (visited on 09/10/2018).
- [5] *Gregor Mendel (1822-1884) :: DNA Learning Center*. URL: <https://www.dnalc.org/view/16151-Biography-1-Gregor-Mendel-1822-1884-.html> (visited on 10/10/2018).
- [6] Garrett Grolemund. *Data Tidying*. 2015. URL: <http://garrettgman.github.io/tidying/> (visited on 09/16/2018).
- [7] *Health insurance coverage in the United States*. en. Page Version ID: 861600386. Sept. 2018. URL: https://en.wikipedia.org/w/index.php?title=Health_insurance_coverage_in_the_United_States&oldid=861600386 (visited on 10/05/2018).
- [8] *Histogram*. en. Page Version ID: 858967679. Sept. 2018. URL: <https://en.wikipedia.org/w/index.php?title=Histogram&oldid=858967679> (visited on 09/26/2018).
- [9] Klara Johansson, Mattias Lindgren, and Ola Rosling. *Infant mortality rate (per 1,000 live birth)*. Oct. 2015. URL: https://docs.google.com/spreadsheets/d/10HMMuHbSFKDolNHXsmgHYlkjSKfAZyyY1P-ddMu_Fz0/pub# (visited on 09/10/2018).
- [10] Douglas C. Montgomery. *Design and Analysis of Experiments*. English. 9th ed. Wiley, Apr. 2017. ISBN: 978-1-119-32093-7.
- [11] *NHIS - 2017 Data Release*. en-us. Aug. 2018. URL: https://www.cdc.gov/nchs/nhis/nhis_2017_data_release.htm (visited on 09/27/2018).

- [12] World Health Organization. *Ebola Situation Reports | Ebola*. URL: <http://apps.who.int/ebola/ebola-situation-reports> (visited on 09/10/2018).
- [13] *Pie chart*. en. Page Version ID: 856409948. Aug. 2018. URL: https://en.wikipedia.org/w/index.php?title=Pie_chart&oldid=856409948 (visited on 09/09/2018).
- [14] Centre d'expertise hydrique du Québec. *Données journalières de débit de la rivière Portneuf en amont des chutes Philias*. June 2012. URL: https://www.cehq.gouv.qc.ca/depot/historique_donnees/fichier/070401_Q.txt.
- [15] *Quick-R: Data Types*. URL: <https://www.statmethods.net/input/datatypes.html> (visited on 09/10/2018).
- [16] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2016. URL: <https://www.R-project.org/>.
- [17] Rebecca L. Siegel, Kimberly D. Miller, and Ahmedin Jemal. "Cancer statistics, 2018". en. In: *CA: A Cancer Journal for Clinicians* 68.1 (Jan. 2018), pp. 7–30. ISSN: 1542-4863. DOI: 10.3322/caac.21442. URL: <https://onlinelibrary.wiley.com/doi/abs/10.3322/caac.21442> (visited on 09/16/2018).
- [18] Reed A. C. Siemieniuk et al. "Arthroscopic surgery for degenerative knee arthritis and meniscal tears: a clinical practice guideline". en. In: *BMJ* 357 (May 2017), j1982. ISSN: 1756-1833. DOI: 10.1136/bmj.j1982. URL: <https://www.bmj.com/content/357/bmj.j1982> (visited on 09/18/2018).
- [19] American Cancer Society. *Cancer Facts & Figures 2018*. en. Tech. rep. Atlanta, 2018. URL: <https://www.cancer.org/research/cancer-facts-statistics/all-cancer-facts-figures/cancer-facts-figures-2018.html> (visited on 09/13/2018).
- [20] *UM Salary.info - University of Michigan Salary Information*. URL: <http://www.umsalary.info/deptsearch.php?Dept=athletics&Year=0&Campus=0> (visited on 10/10/2018).
- [21] Tyler Vigen. *Spurious Correlations*. English. Hachette Books, May 2015. ISBN: 978-0-316-33943-8.
- [22] Staff Writers. *The 25 Highest-Paid College Coaches of 2018*. en-US. Mar. 2018. URL: <https://thebestschools.org/magazine/highest-paid-college-coaches/> (visited on 10/10/2018).

Glossary

continuous variable A variable that refers to continuous data, i.e. that can take on an infinite number of values (ex. height in mm), as opposed to categorical data (ex. color of eyes). 5, 42

dependent variable The "explained" variable in a relationship, the one we try to understand as a consequence of another factor. For example, when studying the effect of smoking on lung cancer, lung cancer is the dependent variable.. 5

discrete variable A variable that refers to categorical data (ex. color of eyes), as opposed to continuous data (ex. height in mm). 5, 28, 31

independent variable The "explaining" variable in a relationship, the one that drives a phenomena. For example, when trying to understand the causes of diabetes, body mass index would be an independent variable.. 5

ordinal variable A qualitative variable where the values can be ordered (ex. small, medium, large). 5, 10

qualitative variable A variable that is recorded with words rather than numbers (ex. color of eyes, state of mind). 4, 5, 10, 27, 28, 31

quantitative variable A variable that is measured with numbers (ex. number of cases, height in mm). 4, 5, 44, 48