

Ulrich Matter

Data Handling

To the students who have motivated and inspired this work.

Contents

List of Figures	vii
List of Tables	ix
Preface	xi
1 Introduction	1
2 Programming with Data	3
2.1 Why learn to program to handle data?	3
2.2 Why R?	4
2.2.1 The ‘data language’	4
2.2.2 High-level language, relatively easy to learn	4
2.2.3 Free, open source, large community	4
2.3 R/RStudio overview	5
2.3.1 The R-Console	6
2.3.2 R-Scripts	7
2.3.3 R Environment	8
2.3.4 File Browser	8
2.4 First steps with R	9
2.4.1 Values, Vectors, and Variables	9
2.4.2 Math operators	12
2.5 Basic programming concepts in R	14
2.5.1 Loops	15
2.5.2 For-loops	15
2.5.3 Booleans and logical statements	18
2.5.4 R functions	20
3 Data	23

3.1	Processing data: simple calculations in numeral systems	24
3.1.1	The binary system	25
3.1.2	The hexadecimal system	27
3.2	Character encoding	28
3.3	Computer code and text files	29
3.4	Data processing basics	30
3.4.1	Components of a standard computing environment	31
3.4.2	Illustration	32
4	Data Storage and Data Structures	37
4.1	Unstructured data in text files	37
4.2	Structured data formats	40
4.2.1	CSVs and fixed-width format	42
4.3	Units of information/data storage	44
4.4	Data structures and data types in R	45
4.4.1	Data types	46
4.4.2	Data structures	47
5	High-Dimensional Data	53
5.1	Deciphering XML	54
5.2	Deciphering JSON	59
5.3	Parsing XML and JSON in R	61
5.4	HTML	62
5.4.1	Write a simple webpage with HTML	63
5.4.2	Two ways to read a webpage into R	66
6	Data Sources, Data Gathering, Data Import	71
6.1	Sources/formats	71
6.2	Data gathering procedure	72
6.3	Loading/importing rectangular data	74
6.3.1	Loading built-in datasets	74
6.3.2	Importing rectangular data from text-files	76
6.4	Import and parse with <code>readr</code>	78
6.4.1	Basic usage of <code>readr</code> functions	78
6.4.2	Parsing and data types	80
6.5	Importing web data formats	81

6.5.1	XML in R	81
6.5.2	JSON in R	83
6.5.3	Tutorial (advanced): Importing data from a HTML table (on a website)	84
7	Data Preparation	89
7.1	Cleaning data with basic string operations	89
7.1.1	Find/replace character strings, recode factor levels	90
7.1.2	Removing individual characters from a string	92
7.1.3	Splitting strings	92
7.1.4	Parsing dates	94
7.2	Reshaping datasets	95
7.2.1	Tidying messy datasets.	95
7.2.2	Pivoting from 'wide to long' ("gathering") . .	96
7.2.3	Pivoting from 'long to wide' ("spreading") . .	98
7.3	Tutorial: Hotel Bookings Time Series	99
7.4	Set up and import	101
8	Data Analysis: First Steps	105
8.1	Merging datasets	105
8.2	Selecting subsets	108
8.2.1	Filtering datasets	108
8.2.2	Mutating datasets	109
8.2.3	Aggregation and summary statistics	110
8.3	Tutorial: Analyse messy Excel sheets	112
8.3.1	Preparatory steps	112
9	Basic Econometrics in R	117
9.1	Statistical modeling	117
9.1.1	Illustration with pseudo-data	118
9.2	Estimation and Application	122
9.2.1	Model specification	122
9.2.2	Raw data	123
9.3	Derivation and implementation of OLS estimator . .	124
9.3.1	Regression toolbox in R	127
10	Data Visualization	129

10.1	Data display	129
10.2	Data visualization with <code>ggplot2</code>	131
10.3	‘Grammar of Graphics’	132
10.3.1	<code>ggplot2</code> basics	132
10.3.2	Tutorial	132
10.3.3	Loading/preparing the data	133
10.3.4	Geometries (~ type of plot)	134
10.4	Dynamic documents	140
10.5	Tutorial: How to give the wrong impression with data visualization	141
10.5.1	Background and aim	141
10.5.2	Data import and preparation	142
10.5.3	Plot	144
10.5.4	Cosmetics: theme	146
Appendix		149
Appendix A		149
.1	Understanding statistics and probability with code	149
.1.1	Random draws and distributions	149
.1.2	Illustration of variability	149
.1.3	Skewness and Kurtosis	150
.1.4	The Law of Large Numbers	156
.1.5	The Central Limit Theorem	159

List of Figures

1	Creative Commons License	xi
2.1	Running R in the Mac/OSX terminal.	5
2.2	Running R in the original R GUI/IDE.	6
2.3	Running R in RStudio (IDE).	7
2.4	Running R in the Mac/OSX terminal.	7
2.5	The R Script window in RStudio.	8
2.6	The environment window in RStudio.	8
2.7	The file browser window in RStudio.	9
2.8	For-loop illustration.	15
2.9	While-loop illustration.	18
3.1	A simple abacus.	24
3.2	The ‘blackbox’ of data processing.	30
3.3	Basic components of a standard computing environment.	31
3.4	Components involved in visiting a website.	33
3.5	Data involved in visiting a website.	33
3.6	Accessing data in RAM, processing it, and storing the result in RAM.	35
3.7	Writing data stored in RAM to a Mass Storage device (hard drive).	35
4.1	Writing data stored in RAM to a Mass Storage device (hard drive). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).	44
4.2	Illustration of a numeric vector (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).	48
4.3	Illustration of a factor (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).	48

4.4	Illustration of a numeric matrix (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).	49
4.5	Illustration of a data frame (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ). . .	50
4.6	Illustration of a data frame (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ). . .	51
5.1	HTML (DOM) tree diagram (by Lubaochuan 2014, licensed under the Creative Commons Attribution-Share Alike 4.0 International license).	65
5.2	Source: https://en.wikipedia.org/wiki/Economy_of_Switzerland	66
9.1	(ref:causality)	118

List of Tables



Preface

This textbook introduces students to the fundamental practices of Data Science in the context of economic research. The course covers basic theoretical concepts and practical skills in gathering, preparing/cleaning, visualizing, storing, and analyzing digital data for research purposes.



FIGURE 1: Creative Commons License

The online version of this book is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License¹.

Ulrich Matter St. Gallen, Switzerland

¹<http://creativecommons.org/licenses/by-nc-sa/4.0/>



I

Introduction

Lower computing costs, a stark decrease in storage costs for digital data, as well as the diffusion of the Internet, have led to the development of new products (e.g., smartphones) and services (e.g., web search engines, cloud computing) over the last few decades. A side product of these developments is a strong increase in the availability of digital data describing all kinds of everyday human activities (Einav and Levin, 2014; Matter and Stutzer, 2015). As a consequence, new business models and economic structures are emerging with data as their core commodity (i.e., AI-related technological and economic change). For example, the current hype surrounding ‘Artificial Intelligence’ (AI) - largely fueled by the broad application of machine-learning techniques such as ‘deep learning’ (a form of artificial neural networks) - would not be conceivable without the increasing abundance of large amounts of digital data on all kind of socio-economic entities and activities. In short, without understanding and handling the underlying data streams properly, the AI-driven economy cannot function. The same rationale applies, of course, to other ways of making use of digital data. Be it traditional big data analytics or scientific research (e.g., applied econometrics).

The need for proper handling of large amounts of digital data has given rise to the interdisciplinary field of ‘Data Science’¹ and increasing demand for ‘Data Scientists’. While nothing within Data Science is particularly new on its own, the combination of skills and insights from different fields (particularly Computer Science and Statistics) has proven to be very productive in meeting new challenges posed by a data-driven economy. In that sense, Data Science is rather a craft than a scientific field. As such, it presupposes a more practical and

¹https://en.wikipedia.org/wiki/Data_science

broader understanding of the data than traditional Computer Science and Statistics from which Data Science borrows its methods. This book focuses on the skills necessary for *acquiring, cleaning, and manipulating* digital data for research/analytics purposes.

The book presupposes a basic knowledge of undergraduate economics and statistics and relates several case studies to practical questions in these fields. Finally, the aim is to give you firsthand practical insights into each part of the data science pipeline in the context of business and economics research.

2

Programming with Data

2.1 Why learn to program to handle data?

With lower computing costs, lower storage costs for digital data, and the diffusion of the Internet, we have recently witnessed a stark increase in the availability of digital data describing all kind of everyday human activities (Einav and Levin, 2014; Matter and Stutzer, 2015). As a consequence, new business models and economic structures are emerging with data as their core commodity (i.e., AI-related technological and economic change). A ‘data-driven’ economy heavily relies on processing/analyzing/handling large amounts of digital data.

The need for proper handling of large amounts of digital data has given rise to the interdisciplinary field of ‘Data Science’¹ as well as an increasing demand for ‘Data Scientists’. While nothing within Data Science is particularly new on its own, it is the combination of skills and insights from different fields (particularly Computer Science and Statistics) that has proven to be very productive in meeting new challenges posed by a data-driven economy. The various facets of this new craft are often illustrated in the ‘Data Science’ Venn-Diagram (see, for example, <http://berkeleysciencereview.com/how-to-become-a-data-scientist-before-you-graduate/>), reflecting the combination of knowledge and skills from Mathematics/Statistics, substantive expertise in the particular scientific field in which Data Science is applied, and ‘hacking skills’, that is, the skills necessary for *acquiring, cleaning, and analyzing* data *programmatically*.

Apart from the current ‘Data Science developments’ and the related career opportunities for young economists, learning to program comes

¹https://en.wikipedia.org/wiki/Data_science

with many benefits for academic work in graduate economics. Many of the courses in your curriculum will at least partially touch upon practical programming exercises or concepts related to programming and scientific computing in an environment like R.

2.2 Why R?

2.2.1 The ‘data language’

The programming language and open-source statistical computing environment R² has over the last decade become a core tool for data science in industry and academia. It was originally designed as a tool for statistical analysis. Many characteristics of the language make R particularly useful to work with data. With the rise of the ‘data economy’ and ‘data science’, R is increasingly used in various domains, going well beyond the traditional applications of academic research.

2.2.2 High-level language, relatively easy to learn

R is a relatively easy computer language to learn for people with no previous programming experience. The syntax is rather intuitive and error messages are not too cryptic to understand (this facilitates learning by doing). Moreover, with R’s recent stark rise in popularity, there are plenty of freely accessible resources online that help beginners to learn the language.

2.2.3 Free, open source, large community

Due to its vast base of contributors, R serves as a valuable tool for users in various fields related to data analysis and computation (economics/econometrics, biomedicine, business analytics, etc.). R users have direct access to thousands of freely available ‘R-packages’ (small software libraries written in R), covering diverse aspects of data analysis, statistics, data preparation, and data import.

Hence, a lot of people using R as a tool in their daily work do not actu-

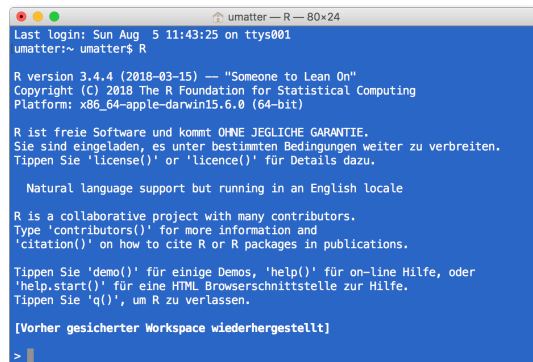
²www.r-project.org

ally ‘write programs’ (in the traditional sense of the word), but apply R packages. Applied econometrics with R is a good example of this. Almost any function a modern commercial computing environment with a focus on statistics and econometrics (such as STATA³) is offering, can also be found within the R environment. Furthermore, there are R packages covering all the areas of modern data analytics, including natural language processing, machine learning, big data analytics, etc. (see the CRAN Task Views⁴ for an overview). We thus do not actually have to write a program for many tasks we perform with R. Instead, we can build on already existing and reliable packages.

2.3 R/RStudio overview

R is the high-level (meaning ‘more user friendly’) programming language for statistical computing. Once we have installed R on our computer, we can run it...

- a. ...directly from the command line, by typing `R` and hit enter (here in the OSX terminal):



```
umatter — R — 80x24
Last login: Sun Aug 5 11:43:25 on ttys001
umatter:~ umatter$ R

R version 3.4.4 (2018-03-15) — "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R ist freie Software und kommt OHNE JEGliche GARANTIE.
Sie sind eingeladen, es unter bestimmten Bedingungen weiter zu verbreiten.
Tippen Sie 'license()' or 'licence()' für Details dazu.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Tippen Sie 'demo()' für einige Demos, 'help()' für on-line Hilfe, oder
'help.start()' für eine HTML Browserschnittstelle zur Hilfe.
Tippen Sie 'q()', um R zu verlassen.

[Vorher gesicherter Workspace wiederhergestellt]

>
```

FIGURE 2.1: Running R in the Mac/OSX terminal.

³<http://www.stata.com/>

⁴<https://cran.r-project.org/web/views/>

- b. ...with the simple Integrated Development Environment (IDE)⁵ delivered with the basic R installation

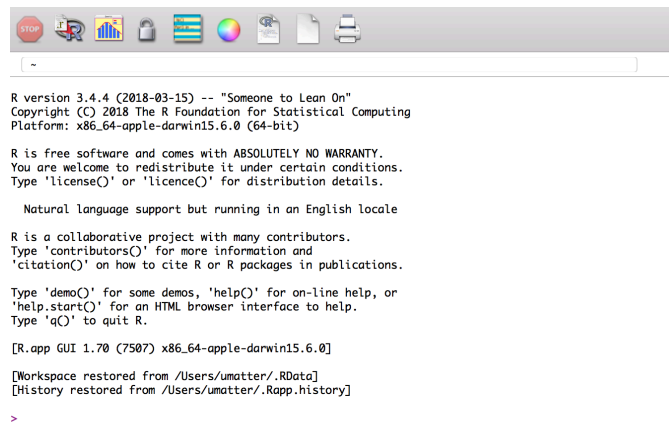


FIGURE 2.2: Running R in the original R GUI/IDE.

- c. ...or with the more elaborated and user-friendly IDE called *RStudio* (either locally or in the cloud, see, for example RStudio Cloud⁶):

The latter is what we will do throughout this course. RStudio is a very helpful tool for simple data analysis with R, writing R scripts (short R programs), or even for developing R packages (software written in R), as well as building interactive documents, presentations, etc. Moreover, it offers many options to change its own appearance (Pane Layout, Code Highlighting, etc.).

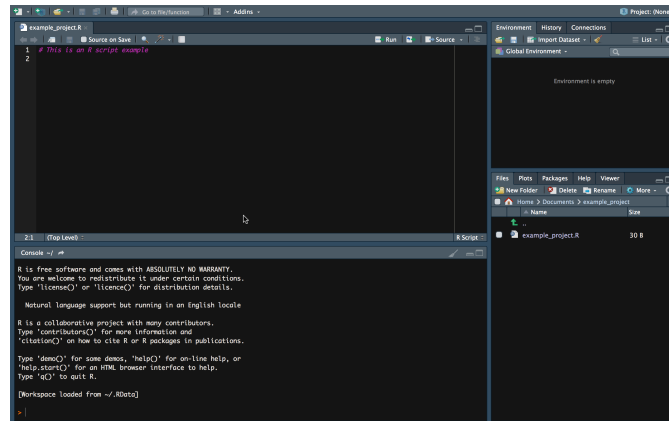
In the following, we have a look at each of the main panels that will be relevant in this course.

2.3.1 The R-Console

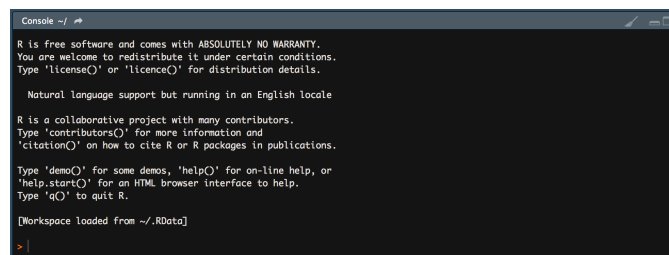
When working in an interactive session, we simply type R commands directly into the R console. Typically, the output of executing a com-

⁵https://en.wikipedia.org/wiki/Integrated_development_environment

⁶<https://rstudio.cloud/>

**FIGURE 2.3:** Running R in RStudio (IDE).

mand this way is also directly printed to the console. Hence, we type a command on one line, hit enter, and the output is presented on the next line.

**FIGURE 2.4:** Running R in the Mac/OSX terminal.

For example, we can tell R to print the phrase `Hello world` to the console, by typing to following command in the console and hit enter:

```
print("Hello world")
```

```
## [1] "Hello world"
```

2.3.2 R-Scripts

Apart from very short interactive sessions, it usually makes sense to write R code not directly in the command line but to an R-script in

the script panel. This way, we can easily execute several lines at once, comment the code (to explain what it does), save it on our hard disk, and further develop the code later on.

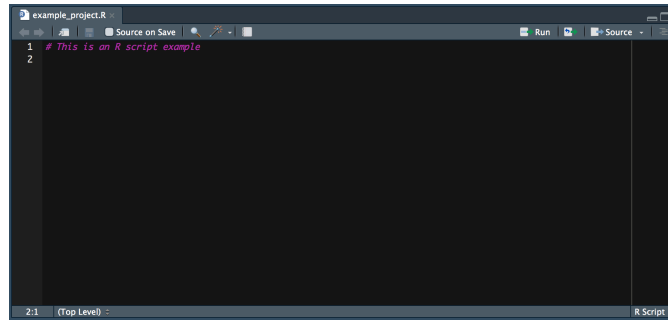


FIGURE 2.5: The R Script window in RStudio.

2.3.3 R Environment

The environment pane shows what variables, objects, and data are loaded in our current R session. Moreover, it offers functions to open documents and import data.

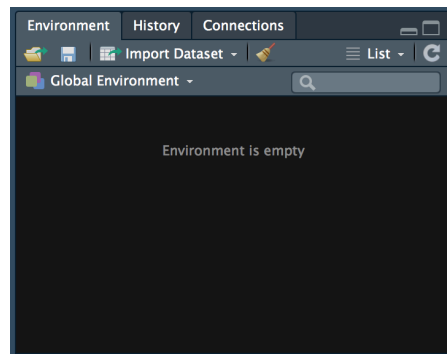


FIGURE 2.6: The environment window in RStudio.

2.3.4 File Browser

With the file browser window we can navigate through the folder structure and files on our computer's hard disk, modify files, and set the working directory of our current R session. Moreover, it has a pane to

show plots generated in R and a pane with help pages and R documentation.

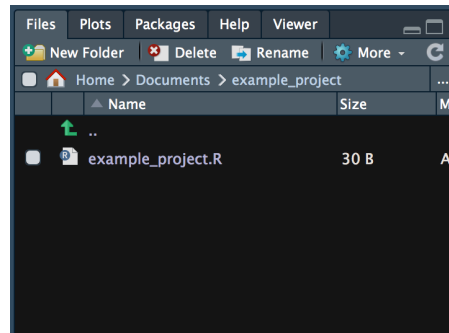


FIGURE 2.7: The file browser window in RStudio.

2.4 First steps with R

Before introducing some of the key functions and packages for data handling and data analysis with R, we should understand how such programs basically work and how we can write them in R. Once we understand the basics of the R language and how to write simple programs, understanding and applying already implemented programs is much easier.⁷

2.4.1 Values, Vectors, and Variables

The simplest objects to work with in R are vectors. In fact, even a simple numeric value such as 5.5 or a string of characters (text) like "Hello" is considered a vector (a scalar).⁸

A first good step to get familiar with coding in R is to assign names to the objects/values you are working with. For example, when summing up two numeric values, you might want to store the result in a separate

⁷In fact, since R is an open source environment, you can directly look at already implemented programs in order to learn how they work.

⁸You can try this out in the R console by typing `is.vector(5.5)` and `is.vector("Hello")`.

object and call this object `result`. This is done with the assignment operator `<-` (or `=`, which serves the same purpose).

```
# assign the variable name "result" to the sum of two numeric values
result <- 25.6 + 53.4
```

Whenever you want to re-use the just computed sum, you can directly call the object by its name (the variable `result`):

```
# check what "is stored in" result
result
```

```
## [1] 79
```

```
# further work with the value in result
result - 20
```

```
## [1] 59
```

With the combine-function (`c()`), you easily form vectors with several elements and name the elements in the vector. By doing so, you create a very simple dataset. For example, suppose you survey the age of a sample of persons. The age values (in years) gives you an integer vector. By naming each integer value (each vector element) after the corresponding person's name, you then have a simple dataset stored in a named R vector.

```
# a simple integer vector
a <- c(10, 22, 33, 22, 40)

# give names to vector elements
names(a) <- c("Andy", "Betty", "Claire", "Daniel", "Eva")
a
```

```
##   Andy  Betty Claire Daniel   Eva
##    10    22    33    22    40
```

To retrieve specific values from this vector, you can either select the corresponding vector element with the element's index (the first, second, third, etc. element) or via its name.

```
# indexing either via a number of vector element (start count with 1)  
# or by element name  
a[3]
```

```
## Claire  
##      33
```

```
a["Claire"]
```

```
## Claire  
##      33
```

When not sure what kind of object `a` is, the `str()` (structure) function, provides you with a short summary.

```
# inspect the object you are working with  
str(a) # returns the structure of the object ("what is in variable a?")
```

```
## Named num [1:5] 10 22 33 22 40  
## - attr(*, "names")= chr [1:5] "Andy" "Betty" "Claire" "Daniel" ...
```

If you want to learn more about what the `c()` or `str()` functions (or any other pre-defined R functions) do and how they should be used, type `help(FUNCTION-NAME)` or `?FUNCTION-NAME` in the console and hit enter. A help-page with detailed explanations of what the function is for and how it can be used will appear in one of the R-Studio panels.

```
help(str)  
?c
```

2.4.2 Math operators

Above, we have just in a side remark introduced the very intuitive syntax for two common math operators in R: + for the addition of numeric or integer values, and - for subtraction. R knows all basic math operators and has a variety of functions to handle more advanced mathematical problems. One basic practical application of R in academic life is to use it as a sophisticated (and programmable) calculator.

```
# basic arithmetic  
2+2
```

```
## [1] 4
```

```
sum_result <- 2+2  
sum_result
```

```
## [1] 4
```

```
sum_result -2
```

```
## [1] 2
```

```
4*5
```

```
## [1] 20
```

```
20/5
```

```
## [1] 4
```

```
# order of operations  
2+2*3
```

```
## [1] 8
```



```
(2+2)*3
```

```
## [1] 12
```

```
(5+5)/(2+3)
```

```
## [1] 2
```

```
# work with variables
```

```
a <- 20
```

```
b <- 10
```

```
a/b
```

```
## [1] 2
```

```
# arithmetics with vectors
```

```
a <- c(1,4,6)
```

```
a * 2
```

```
## [1] 2 8 12
```

```
b <- c(10,40,80)
```

```
a * b
```

```
## [1] 10 160 480
```

```
a + b
```

```
## [1] 11 44 86
```

```
# other common math operators and functions
```

```
4^2
```

```
## [1] 16
```

```
sqrt(4^2)
```

```
## [1] 4
```

```
log(2)
```

```
## [1] 0.6931
```

```
exp(10)
```

```
## [1] 22026
```

```
log(exp(10))
```

```
## [1] 10
```

To look up the most common math operators in R and get more details about how to use them type

```
?`+`
```

in the R console and hit enter.

2.5 Basic programming concepts in R

In very simple terms, programming/coding is all about using a computer language to instruct a computer what to do. What reads very complex at first sight, is actually rather simple in (at least for a large array of basic programming problems). At the core of almost any R program is the right application and combination of just a handful of basic programming concepts: loops, logical statements, control statements, and functions. Once you a) conceptually understand what these concepts are for, and b) have learned the syntax of how to use

these concepts when writing a program in R, addressing all kind of data handling problems efficiently with R, will simply become a matter of training/practice.

2.5.1 Loops

A loop is typically a sequence of statements executed a specific number of times. How often the code ‘inside’ the loop is executed depends on a clearly defined control statement. If we know in advance how often the code inside the loop has to be executed, we typically write a so-called ‘for-loop’. We typically write a so-called ‘while-loop’ if the number of iterations is not clearly known before executing the code. The following subsections illustrate both of these concepts in R.

2.5.2 For-loops

In simple terms, a for-loop tells the computer to execute a sequence of commands ‘for each case in a set of n cases’. The flowchart in Figure @ref(fig: for) illustrates the concept.

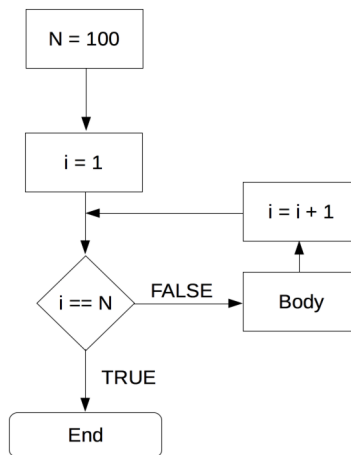


FIGURE 2.8: For-loop illustration.

For example, a for-loop could be used, to sum up each element in a numeric vector of fixed length (thus, the number of iterations is clearly defined). In plain English, the for-loop would state something like: “Start with 0 as the current total value, for each of the elements in the

vector, add the value of this element to the current total value.” Note how this logically implies that the loop will ‘stop’ once the value of the last element in the vector is added to the total. Let’s illustrate this in R. Take the numeric vector `c(1, 2, 3, 4, 5)`. A for loop to sum up all elements can be implemented as follows:

```
# vector to be summed up
numbers <- c(1, 2.1, 3.5, 4.8, 5)
# initiate total
total_sum <- 0
# number of iterations
n <- length(numbers)
# start loop
for (i in 1:n) {
  total_sum <- total_sum + numbers[i]
}

# check result
total_sum
```

```
## [1] 16.4
```

```
# compare with the result of sum() function
sum(numbers)
```

```
## [1] 16.4
```

2.5.2.1 Nested for-loops

In some situations, a simple for-loop might not be sufficient. Within one sequence of commands, there might be another sequence of commands that also has to be executed for a number of times each time the first sequence of commands is executed. In such a case, we speak of a ‘nested for-loop’. We can illustrate this easily by extending the example of the numeric vector above to a matrix for which we want to sum up the values in each column. Building on the loop implemented above, we would say ‘for each column *j* of a given numeric matrix, execute the for-loop defined above’.

```
# matrix to be summed up
numbers_matrix <- matrix(1:20, ncol = 4)
numbers_matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
```

```
# number of iterations for the outer loop
m <- ncol(numbers_matrix)
# number of iterations for the inner loop
n <- nrow(numbers_matrix)
# start outer loop (loop over columns of the matrix)
for (j in 1:m) {
  # start inner loop
  # initiate total
  total_sum <- 0
  for (i in 1:n) {
    total_sum <- total_sum + numbers_matrix[i, j]
  }
  print(total_sum)
}
```

```
## [1] 15
## [1] 40
## [1] 65
## [1] 90
```

2.5.2.2 While-loop

In a situation where a program has to repeatedly run a sequence of commands, but we don't know in advance how many iterations we need to reach the intended goal, a while-loop can help. In simple terms, a while loop keeps executing a sequence of commands as long

as a certain logical statement is true. The flow chart in Figure @ref(fig: while) illustrates this point.

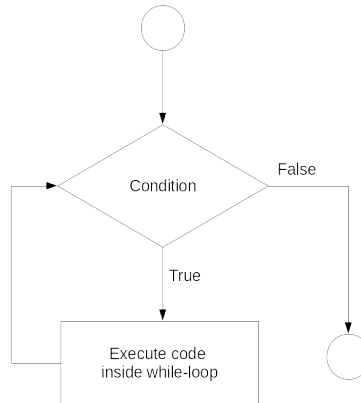


FIGURE 2.9: While-loop illustration.

For example, a while-loop in plain English could state something like “start with 0 as the total, add 1.12 to the total until the total is larger than 20.” We can implement this in R as follows.

```

# initiate starting value
total <- 0
# start loop
while (total <= 20) {
  total <- total + 1.12
}

# check the result
total

```

```
## [1] 20.16
```

2.5.3 Booleans and logical statements

Note that in order to write a meaningful while-loop we have to make use of a logical statement such as “the value stored in the variable `total` is smaller or equal to 20” (`total <= 20`). A logical statement results

in a 'Boolean' data type. That is, a data type with the only two possible values TRUE or FALSE (1 or 0).

```
2+2 == 4
```

```
## [1] TRUE
```

```
3+3 == 7
```

```
## [1] FALSE
```

Logical statements play an important role in fundamental programming concepts. In particular, they are crucial to make conditional statements ('if-statements') that build the control structure of a program, controlling the 'direction' the program takes (given certain conditions).

```
condition <- TRUE
if (condition) {
  print("This is true!")
} else {
  print("This is false!")
}
```

```
## [1] "This is true!"
```

```
condition <- FALSE
if (condition) {
  print("This is true!")
} else {
  print("This is false!")
}
```

```
## [1] "This is false!"
```

2.5.4 R functions

R programs heavily rely on functions. Conceptually, ‘functions’ in R are very similar to what we know as ‘functions’ in math (i.e., $f : X \rightarrow Y$). A function can thus, e.g., take a variable X as input and provide value Y as output. The actual calculation of Y based on X can be something as simple as $2 \times X = Y$. But it could also be a very complex algorithm or an operation that does not directly have anything to do with numbers and arithmetic.⁹

In R—and many other programming languages—functions take ‘parameter values’ as input, process those values according to a predefined program, and ‘return’ the result. For example, a function could take a numeric vector as input and return the sum of all the individual numeric values in the input vector.

When we open RStudio, all basic functions are already loaded automatically. This means we can directly call them from the R-Console or by executing an R-Script. As R is made for data analysis and statistics, the basic functions loaded with R cover many aspects of tasks related to working with and analyzing data. Besides these basic functions, thousands of additional functions covering all kinds of topics related to data analysis can be loaded additionally by installing the respective R-packages (`install.packages("PACKAGE-NAME")`) and then loading the packages with `library(PACKAGE-NAME)`. In addition, it is straightforward to define our own functions.

2.5.4.1 Case study: Compute the mean

To illustrate the point of how functions work in R and how we can write our own functions in R, the following code-example illustrates how to implement a function that computes the mean/average value, given a numeric vector.

⁹Of course, on the very low level, everything that happens in a microprocessor can, in the end, be expressed in some formal way using math. However, the point here is that at the level we work with R, a function could simply process different text strings (i.e., stack them together). Thus for us as programmers, R functions do not necessarily have to do anything with arithmetic and numbers but could serve all kinds of purposes, including the parsing of HTML code, etc.

First, we initiate a simple numeric vector which we then use as an example to test the function. Whenever you implement a function, it is very useful to first define a simple example of an input for which you know what the output should be.

```
# a simple integer vector, for which we want to compute the Mean
a <- c(5.5, 7.5)
# desired functionality and output:
# my_mean(a)
# 6.5
```

In this example, we would thus expect the output to be 6.5. Later, we will compare the output of our function with this in order to check whether our function works as desired.

In addition to defining a simple example and the desired output, it makes sense to also think about *how* the function is expected to produce this output. When implementing functions related to statistics (such as the mean), it usually makes sense to have a look at the mathematical definition:

$$\bar{x} = \frac{1}{n} \left(\sum_{i=1}^n x_i \right) = \frac{x_1 + x_2 + \dots + x_n}{n}.$$

Now, we can start thinking about implementing the function based on built-in R functions. From looking at the mathematical definition of the mean (\bar{x}), we recognize that there are two main components to computing the mean:

- $\sum_{i=1}^n x_i$: the sum of all the elements in vector x
- n : the number of elements in vector x .

Once we know how to get these two components, computing the mean is straightforward. In R, there are two built-in functions that deliver exactly these two components:

- `sum()` returns the sum of all the values in its arguments (i.e., if x is a numeric vector, `sum(x)` returns the sum of all elements in x).
- `length()` returns the length of a given vector.

With the following short line of code, we thus get the mean of the elements in vector `a`.

```
sum(a)/length(a)
```

```
## [1] 6.5
```

All that is left to do is to pack all this into the function body of our newly defined `my_mean()` function:

```
# define our own function to compute the mean, given a numeric vector
my_mean <- function(x) {
  x_bar <- sum(x) / length(x)
  return(x_bar)
}
```

Now we can test it based on our example:

```
# test it
my_mean(a)
```

```
## [1] 6.5
```

Moreover, we can test it by comparing it with the built-in `mean()` function:

```
b <- c(4,5,2,5,5,7)
my_mean(b) # our own implementation
```

```
## [1] 4.667
```

```
mean(b) # the built_in function
```

```
## [1] 4.667
```

3

Data

To better understand the role of data in today's economy and society, we study the usage forms and purposes of data records in human history. In the second step, we look at how a computer processes digital data.

Throughout human history, the recording and storage of data have primarily been motivated by measuring, quantifying, and keeping records of both our social and natural environments. Early on, data recording was related to economic activity and scientific endeavors. The neolithic transition from hunter-gatherer societies to agriculture and settlements (the economic development sometimes referred to as the 'first industrial revolution') came along with a division of labor and more complex organizational structures of society. Because of the change in agricultural food production, more people could be fed. But it also implied that food production would need to follow a more careful planning (e.g., the right time to seed and harvest) and that the produced food (e.g., grains) would partly be stored and not consumed entirely on the spot. It is believed that partly due to these two practical problems, keeping track of time and keeping record of production quantities, neolithic societies started to use signs (numbers/letters) carved in stone or wood ([Hogben, 1983](#)). Keeping record of the time and later measuring and keeping record of production quantities in order to store and trade likely led to the first 'data sets'. At the same time the development of mathematics, particularly geometry, took shape.

3.1 Processing data: simple calculations in numeral systems

In order to keep track of calculations with large numbers, humans started to use mechanical aids such as pebbles or shells and later developed the counting frame ('abacus')^{1, 2}. We can understand the counting frame as a simple mechanical tool to process numbers (data). In order to use the abacus properly, one must agree on a standard regarding what each column of the frame represents, as well as how many beads each column can contain. In other words, one has to define what the *base* of the frame's numeral system is. For example, the Roman numeral system is essentially of base 10, with 'I' representing one, 'X' representing ten, 'C' representing one hundred, and 'M' representing one thousand. Figure 3.1 illustrates how a counting frame based on this numeral system works (examples are written out in Arabic numbers). The first column on the right represents units of $10^0 = 1$, the second $10^1 = 10$, and so forth.

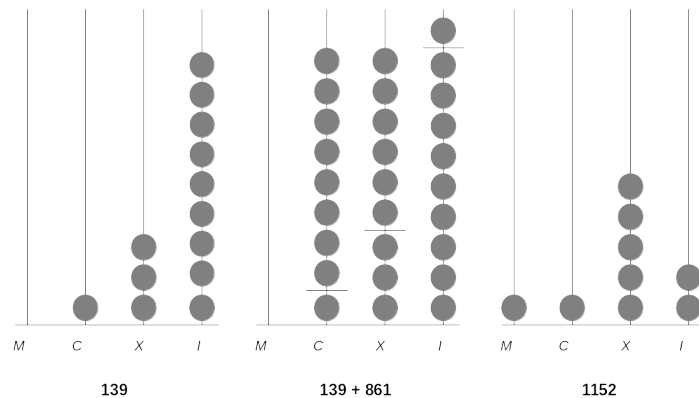


FIGURE 3.1: A simple abacus.

From inspecting Figure 3.1 we recognize that the columns of the aba-

¹<https://en.wikipedia.org/wiki/Abacus>

²See Hogben (1983), Chapter 1 for a detailed account of the abacus' origin and usage.

cus are the positions of the digits, which also denote the power of 10 with which the digit is multiplied: $139 = (1 \times 10^2) + (3 \times 10^1) + (9 \times 10^0)$. In addition, a base of 10 means that the system is based on 10 different signs (0, 1, ..., 9) with which we distinguish one-digit numbers. Further, we recognize that each column in the abacus has 9 beads (when we agree on the standard that a column without any bead represents the sign 0).

The numeral system with base 10 (the ‘decimal system’) is what we use to measure/count/quantify things in our everyday life. Moreover, it is what we normally work with in applied math and statistics. However, the simple but very useful concept of the counting frame also works well for numeral systems with other bases. Historically, numeral systems with other bases existed in various cultures. For example, ancient cultures in Mesopotamia used different forms of a sexagesimal system (base 60), consisting of 60 different signs to distinguish ‘one-digit’ numbers.

Note that this logic holds both ways: if a numeral system only consists of two different signs, it follows that the system has to be of base 2 (i.e., a ‘binary system’). As it turns out, this is exactly the kind of numeral system that becomes relevant once we use electronic tools, that is, digital computers, to calculate and process data.

3.1.1 The binary system

Anything related to electronically processing (and storing) *digital data* has to be built on a binary system. The reason is that a microprocessor (similar to a light switch) can only represent two signs (states): *on* and *off*. We usually refer to ‘off’ with the sign ‘0’ and to ‘on’ with the sign ‘1’. A numeral system consisting only of the signs 0 and 1 must, thus, be of base 2. It follows that an abacus of this binary system has columns $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, and so forth. Moreover, each column has only one bead (1) or none (0). Nevertheless, we can express all (natural) numbers from the decimal system.³ For example, the num-

³Representing fractions is much harder in a binary system than representing natural numbers. The reason is fractions such as $1/3 = 0.333\dots$ actually constitute an infinite sequence of 0s and 1s. The representation of such numbers in a computer (via so-called ‘floating point’ numbers) is thus in reality not 100% accurate.

ber 139 which we have expressed with the ‘decimal abacus’, would be expressed as follows with a base 2 system:

$$(1 \times 2^7) + (1 \times 2^3) + (1 \times 2^1) + (1 \times 2^0) = 139.$$

More precisely, when including all columns of our binary system abacus (that is, including also the columns set to 0), we would get the following:

$$(1 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 139.$$

Now, compare this with the abacus in the decimal system above. There, we set the third column to 1, the second column to 3, and the first column (from the right) to 9, resulting in 139. If we do the same in the binary system (where we can set each column either to 0 or 1), we get 10001011. That is, the number 139 in the decimal system corresponds to 10001011 in the binary system.

How can a computer know that? To correctly print the three symbols 139 to our computer screen when dealing with the binary expression 10001011, the computer needs to rely on a predefined mapping between binary coded values and the symbols for (Arabic) decimal numbers like 3. Since a computer can only understand binary expressions, we have to define a *standard* of how 0s and 1s correspond to symbols, colors, etc. that we see on the screen.

Of course, this does not change the fact that any digital data processing is, in the end, happening in the binary system. But in order to avoid having to work with a keyboard consisting only of an on/off (1/0) switch, low-level standards that define how symbols like 3, A, #, etc. corresponding expressions in 0s and 1s help us to interact with the computer. It makes it easier to enter data and commands into the computer as well as understand the output (i.e., the result of a calculation performed on the computer) on the screen. A standard defining how our number symbols in the decimal system correspond to binary numbers (again, reflecting the idea of an abacus) can be illustrated in the following table:

Number	128	64	32	16	8	4	2	1
0 =	0	0	0	0	0	0	0	0
1 =	0	0	0	0	0	0	0	1
2 =	0	0	0	0	0	0	1	0
3 =	0	0	0	0	0	0	1	1
...								
139 =	1	0	0	0	1	0	1	1

3.1.2 The hexadecimal system

From the above table, we also recognize that binary numbers can become quite long rather quickly (have many digits). In Computer Science it is, therefore, quite common to use another numeral system to refer to binary numbers: the *hexadecimal* system. In this system, we have 16 symbols available, consisting of 0-9 (used like in the decimal system) and A-F (for the numbers 10 to 15). Because we have 16 symbols available, each digit represents an increasing power of 16 (16^0 , 16^1 , etc.). The decimal number 139 is expressed in the hexadecimal system as follows.

$$(8 \times 16^1) + (11 \times 16^0) = 139.$$

More precisely, following the convention of the hexadecimal system used in Computer Science (where B stands for 11), it is:

$$(8 \times 16^1) + (B \times 16^0) = 8B = 139.$$

Hence, 10001011 in the binary system is 8B in the hexadecimal system and 139 in the decimal system. The primary use of hexadecimal notation when dealing with binary numbers is the more 'human-friendly' representation of binary-coded values. First, it is shorter than the raw binary representation (as well as the decimal representation). Second, with a little bit of practice it is much easier for humans to translate forth and back between hexadecimal and binary notation than it is between decimal and binary. This is because each hexadecimal digit can directly be converted into its four-digit binary equivalent (from looking at the table above): $8 = 1000$, $B = 11 = 1011$, thus 8B in hexadecimal notation corresponds to 10001011 (1000 1011) in binary coded values.

3.2 Character encoding

Computers are not only used to process numbers but in a wide array of applications, they are used to process text. Most fundamentally, when writing computer code, we type in commands in the form of text consisting of common alphabetical letters. How can a computer understand text if it only understands 0s and 1s? Well, again, we have to agree on a certain standard of how 0s and 1s correspond to characters/letters. While the conversions of integers between different numeral systems follow all essentially the same principle (~ the idea of a ‘digital’ abacus), the introduction of standards defining how 0s and 1s correspond to specific letters of different human languages is way more arbitrary.

Today, many standards define how computers translate 0s and 1s into more meaningful symbols like numbers, letters, and special characters in various natural (human) languages. These standards are called *character encodings*. They consist of what is called a ‘coded character set’, basically a mapping of unique numbers (in the end in binary coded values) to each character in the set. For example, the classical ASCII (American Standard Code for Information Interchange) assigns the following numbers to the respective characters:

Binary	Hexadecimal	Decimal	Character
0011 1111	3F	63	?
0100 0001	41	65	A
0110 0010	62	98	b

The convention that 0100 0001 corresponds to A is only true by definition of the ASCII character encoding.⁴ It does not follow any law of nature or fundamental logic. Somebody simply defined this standard at some point. To have such standards (and widely accept them) is paramount to have functioning software and meaningful data sets that can be used and shared across many users and computers. If we

⁴Each character digit is expressed in 1 byte (8 0/1 digits). The ASCII character set thus consists of $2^8 = 256$ different characters.

write a text and store it in a file based on the ASCII standard, and that text (which, under the hood, only consists of 0s and 1s) is read on another computer only capable of mapping the binary code to another character set, the output would likely be complete gibberish.

In practice, the software we use to write emails, browse the Web, or conduct statistical analyses all have some of these standards built into them. Usually, we do not have to worry about encoding issues when simply interacting with a computer through such programs. The practical relevance of such standards becomes much more relevant once we *store* or *read* previously stored data/computer code.

3.3 Computer code and text files

Understanding the fundamentals of what digital data is and how computers process them is the basis for approaching two core themes of this book: (I) How *data* can be *stored* digitally and be read by/imported to a computer (this will be the main subject of the next chapter), and (II) how we can give instructions to a computer by writing *computer code* (this will be the main topic of the first two exercises/workshops).

In both of these domains, we mainly work with one simple type of document: *text files*. Text files are a collection of characters (with a given character encoding). Following the logic of how binary code is translated forth and back into text, they are a straightforward representation of the underlying information (0s and 1s). They are used to store both structured data (e.g., tables), unstructured data (e.g., plain texts), or semi-structured data (such as websites). Similarly, they are used to write and store a collection of instructions to the computer (i.e., computer code) in any computer language.

From our everyday use of computers (notebooks, smartphones, etc.), we are accustomed to certain software packages to write text. Most prominently, we use programs like Microsoft Word or email clients (Outlook, Thunderbird, etc.). However, these programs are a poor choice for writing/editing plain text. Such programs tend to add all kinds of formatting to the text and use specific file formats to store

formatting information in addition to the raw text. In other words, when using this type of software, we actually do not only write plain text. However, any program to read data or execute computer code expects only plain text. Therefore, we must only use *text editors* to write and edit text files. For example, with Atom⁵ or RStudio⁶.

3.4 Data processing basics

By now, we already have a much better understanding of what the 0s and 1s stand for, how they might enter the computer, where they might be stored on the computer, and where we might see the output of processing data on a computer.



FIGURE 3.2: The ‘blackbox’ of data processing.

This section briefly summarizes the key components of processing data with a computer, following Murrell (2009) (chapter 9, pages 199-204).

⁵<https://atom.io/>

⁶<https://www.rstudio.com/products/RStudio/>

3.4.1 Components of a standard computing environment

Figure @ref(fig: components) illustrates the key components of a standard computing environment (PC, notebook, tablet, etc.) with which we can process data. A programming language (such as R) allows us to work with these hardware components (i.e., control them) in order to perform our tasks (e.g., cleaning/analyzing data).

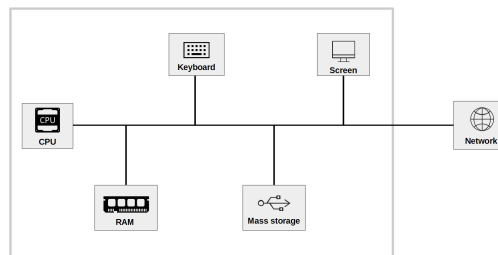


FIGURE 3.3: Basic components of a standard computing environment.

- The component actually *processing* data is the Central Processing Unit (CPU). When using R to process data, R commands are translated into complex combinations of a small set of basic operations which the CPU then executes.
- To work with data (e.g., in R), it first must be loaded into the *memory* of our computer. More specifically, into the Random Access Memory (RAM). Typically, data is only loaded in the RAM as long as we work with it.
- The *Keyboard* is the typical *input* hardware. It is the key tool we need to interact with the computer in this book (in contrast, in a book on graphic design this would rather be the mouse).
- *Mass Storage* refers to the type of computer memory we use to store data in the long run. Typically this is what we call the *hard drive* or *hard disk*. In these days, the relevant hard disk for storing and accessing data is actually often not the one physically built into our computer but a hard disk ‘in the cloud’ (built into a server to which we connect over the Internet).

- The *Screen* is our primary output hardware that is key to interact with the computer.
- Today almost all computers are connected to a local *network* which in turn is connected to the Internet. With the rise of ‘Big Data’ this component has become increasingly relevant to understand, as the key data sources might reside in this network.

Note how understanding the role of these components helps us to capture what happened in the initial example of this book (online survey of the class):

1. You have used your computers to access a website (over the Internet) and used your keyboards to enter data into this website (a Google sheet in that case).
2. My R program has accessed the data you have entered into the Google sheet (again over the Internet), downloaded the data, and loaded it into the RAM on my notebook.
3. There, the data has been processed in order to produce an output (in the form of statistics/plots), which in the end has been shown on the screen.

In the following, we look at a related example (taken from [Murrell \(2009\)](#), Chapter 9), illustrating which hardware components are involved when extracting data from a website.

3.4.2 Illustration

3.4.2.1 Downloading/accessing a website

First, we use our keyboard to enter the address of a website in the address bar of our browser (i.e., Firefox). The browser then accesses the network and loads the webpage to the RAM. Figure 3.4 illustrates this point.

In this simple task of accessing/visiting a website, several parts of data are involved, as shown in 3.5:

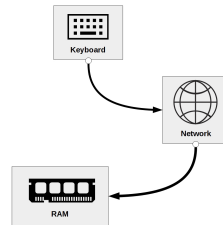


FIGURE 3.4: Components involved in visiting a website.

1. The string of characters that we type into the address bar of the browser
2. The source code of the website (written in Hypertext markup language, HTML), being sent through the network
3. The HTML document (the website) stored in the RAM

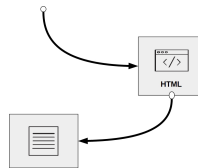


FIGURE 3.5: Data involved in visiting a website.

What we just did with our browser can also be done via R. First, we tell R to download the webpage and read its source code into RAM:

```
clockHTML <- readLines("https://www.census.gov/popclock/")
```

And we can have a look at the first lines of the web page's source code:

```
head(clockHTML)
```

```
## [1] "<!DOCTYPE html>"
## [2] "<html lang=\"en\">"
```

```
## [3] "<head>"
## [4] "\t<title>Population Clock</title>"
## [5] "      <meta charset=\"UTF-8\">"
## [6] "      <meta name=\"description\" content=\"Shows estimates of current USA Population overa
```

Note that what happened is essentially the same as displayed in the diagram above. The only difference is that here, we look at the raw code of the webpage, whereas in the example above, we looked at it through our browser. What the browser did, is to *render* the source code, meaning it translated it into the nicely designed webpage we see in the browser window.

3.4.2.2 Searching/filtering the webpage

Having the webpage loaded into RAM, we can now process this data in different ways via R. For example; we can search the source code for the line that contains the part of the website with the world population counter:

```
line_number <- grep('id="world-pop-container"', clockHTML)
```

That is, we ask R on which line in `clockHTML` (the source code stored in RAM) the text `id="world-pop-container"` is and store the answer (the line number) in RAM under the variable name `line_number`.

We can now check on which line the text was found.

```
line_number
```

```
## [1] 441
```

Again, we can illustrate with a simple diagram which computer components were involved in this task. First, we entered an R command with the keyboard; the CPU is processing it and as a result accesses `clockHTML` in RAM, executes the search, returns the line number, and stores it in a new variable called `line_number` in RAM. In our current R session, we thus now have two ‘objects’ stored in RAM: `clockHTML` and `line_number`, which we can further process with R.

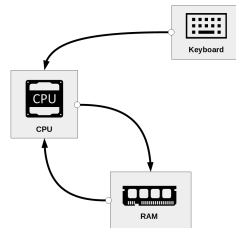


FIGURE 3.6: Accessing data in RAM, processing it, and storing the result in RAM.

Finally, we can tell R to store the source code of the downloaded web-page on our local hard drive.

```
writeLines(clockHTML, "clock.html")
```

That means we tell R to access the data stored in `clockHTML` in RAM and write this data to a text file called “clock.html” on our local hard drive (in the current working directory of our R session). The following diagram again illustrates the hardware components involved in these steps.

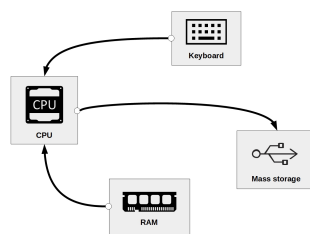


FIGURE 3.7: Writing data stored in RAM to a Mass Storage device (hard drive).



4

Data Storage and Data Structures

A core part of the practical skills covered in this book has to do with writing, executing, and storing *computer code* (i.e., instructions to a computer in a language that it understands) as well as storing and reading *data*. The way data is typically stored on a computer follows quite naturally from the outlined principles of how computers process data (both technically speaking and in terms of practical applications). Based on a given standard of how 0s and 1s are translated into the more meaningful symbols we see on our keyboard, we simply write data (or computer code) to a *text file* and save this file on the hard-disk drive of our computer. Again, what is stored on disk is in the end only consisting of 0s and 1s. But, given the standards outlined in the previous chapter, these 0s and 1s properly map to characters that we can understand when reading the file again from the disk and looking at its content on our computer screen.

4.1 Unstructured data in text files

In the simplest case, we want to store a text literally. For example, we store the following phrase

```
Hello, World!
```

in a text file named `helloworld.txt`. Technically, this means that we allocate a block of computer memory to `helloworld.txt` (a ‘file’ is, in the end, a block of computer memory). The ending `.txt` indicates to the operating system of our computer what kind of data is stored in this file. When clicking on the file’s symbol, the operating system will open the file (read it into RAM) with the default program assigned to open

.txt-files (for example, Atom¹ or RStudio²). That is, the *format* of a file says something about how to interpret the 0s and 1s. If the text editor displays `Hello World!` as the content of this file, the program correctly interprets the 0s and 1s as representing plain text and uses the correct character encoding to convert the raw 0s and 1s yet again to symbols that we can read more easily. Similarly, we can show the content of the file in the command-line terminal (here OSX or Linux):

```
cat helloworld.txt
```

```
## Hello World!
```

Or, from the R-console:

```
system("cat helloworld.txt")
```

However, we can also use a command-line program (here, `xxd`) to display the content of `helloworld.txt` without actually ‘translating’ it into ASCII characters.³ That is, we can directly look at how the content looks like as 0s and 1s:

```
xxd -b helloworld.txt
```

```
## 00000000: 01001000 01100101 01101100 01101100 01101111 00100000 Hello
## 00000006: 01010111 01101111 01110010 01101100 01100100 00100001 World!
```

Similarly, we can display the content in hexadecimal values:

```
xxd helloworld.txt
```

```
## 00000000: 4865 6c6c 6f20 576f 726c 6421 Hello World!
```

Next, consider the text file `hastamanana.txt`, which was written and stored with another character encoding. When looking at the content,

¹<https://atom.io/>

²<https://www.rstudio.com/>

³To be precise, this program shows both the raw binary content as well as its ASCII representation.

assuming the now common UTF-8 encoding, the content seems a little odd:

```
cat hastamanana.txt
```

```
## Hasta Ma?ana!
```

We see that it is a short phrase written in Spanish. Strangely it contains the character `?` in the middle of a word. The occurrence of special characters in unusual places of text files is an indication of using the wrong character encoding to display the text. Let's check what the file's encoding is.

```
file -b hastamanana.txt
```

```
## ISO-8859 text
```

This tells us that the file is encoded with ISO-8859 ('Latin1'), a character set for several Latin languages (including Spanish). Knowing this, we can check how the content looks like when we change the encoding to UTF-8 (which then properly displays the content on the screen)⁴:

```
iconv -f iso-8859-1 -t utf-8 hastamanana.txt | cat
```

```
## Hasta Mañana!
```

When working with data, we must therefore ensure that we apply the proper standards to translate the binary coded values into a character/text representation that is easier to understand and work with. In recent years, much more general (or even 'universal') standards have been developed and adopted worldwide, particularly UTF-8. Thus, if we deal with recently generated data sets, we usually can expect that

⁴Note that changing the encoding in this way only works well if we know what the original encoding of the file is (i.e., the encoding used when the file was initially created). While the `file` command above simply tells us what the original encoding was, it has to make an educated guess in most cases (as the original encoding is often not stored as metadata describing a file). See, for example, the Mozilla Charset Detectors⁵.

they are encoded in UTF-8, independent of the data's country of origin. However, when working on a research project with slightly older data sets from different sources, encoding issues still occur quite frequently. It is thus crucial to understand the origin of the problem early on when the data seem to display weird characters. Encoding issues are among the most basic problems that can occur when importing data.

So far, we have only looked at very simple examples of data stored in text files, essentially only containing short phrases of text. Nevertheless, the most common formats of storing and transferring data (CSV, XML, JSON, etc.) build on exactly the same principle: characters in a text-file. The difference between such formats and the simple examples above is that their content is structured in a very specific way. That is, on top of the lower-level conversion of 0s and 1s into characters/text, we add another standard, a data format, giving the data more *structure*, making it even simpler to interpret and work with the data on a computer. Once we understand how data is represented at a low level (the topic of the previous chapter), it is much easier to understand and distinguish different forms of storing structured data.

4.2 Structured data formats

As nicely pointed out by [Murrell \(2009\)](#), the most commonly used software today is designed in a very 'user-friendly' way, leading to situations in which we as users are 'told' by the computer what to do (rather than the other way around). A prominent symptom of this phenomenon is that specific software is automatically assigned to open files of specific formats, so we don't have to bother about what the file's format actually is but only have to click on the icon representing the file.

However, if we actually want to engage seriously with a world driven by data, we have to go beyond the habit of simply clicking on (data-)files and let the computer choose what to do with it. Since most common formats to store data are in essence text files (in research contexts usu-

ally in ASCII or UTF-8 encoding), a good way to start engaging with a data set is to look at the raw text file containing the data in order to get an idea of how the data is structured.

For example, let's have a look at the file `ch_gdp.csv`. When opening the file in a text editor we see the following:

```
year,gdp_chfb
1980,184
1985,244
1990,331
1995,374
2000,422
2005,464
```

At first sight, the file contains a collection of numbers and characters. Having a closer look, certain structural features become apparent. The content is distributed over several rows, with each row containing a comma character (,). Moreover, the first row seems systematically different from the following rows: it is the only row containing alphabetical characters, all the other rows contain numbers. Rather intuitively, we recognize that the specific way in which the data in this file is structured might, in fact, represent a table with two columns: one with the variable `year` describing the year of each observation (row), the other with the variable `gdp_chfb` with numerical values describing another characteristic of each observation/row (we can guess that this variable is somehow related to GDP and CHF/Swiss francs).

The question arises, how do we work with this data? Recall that it is simply a text file. All the information stored in it is displayed above. How could we explain to the computer that this is a table with two columns containing two variables describing six observations? We would have to come up with a sequence of instructions/rules (i.e., an algorithm) of how to distinguish columns and rows when all that a computer can do is sequentially read one character after the other (more precisely one 0/1 after the other, representing characters).

This algorithm could be something along the lines of:

1. Start with an empty table consisting of one cell (1 row/column).
2. While the end of the input file is not yet reached, do the following: Read characters from the input file, and add them one-by-one to the current cell. If you encounter the character ',', ignore it, create a new field, and jump to the new field. If you encounter the end of the line, create a new row and jump to the new row.

Consider the usefulness of this algorithm. It would certainly work quite well for the particular file we are looking at. But what if another file we want to read data from does not contain any ',' but only ';' ? We would have to tweak the algorithm accordingly. Hence, again, it is extremely useful if we can agree on certain standards of how data structured as a table/matrix is stored in a text file.

4.2.1 CSVs and fixed-width format

Incidentally, the example we are looking at here is in line with such a standard, i.e., Comma-Separated Values (CSV, therefore `.csv`). In technical terms, the simple algorithm outlined above is a CSV parser⁶. A CSV parser is a software component which takes a text file with CSV standard structure as input and builds a data structure in the form of a table (represented in RAM). If the input file does not follow the agreed-on CSV standard, the parser will likely fail to properly 'parse' (read/translate) the input.

CSV files are very often used to transfer and store data in a table-like format. They essentially are based on two rules defining the structure of the data: commas delimit values/fields in a row and the end of a line indicates the end of a row.⁷

The comma is clearly visible when looking at the raw text content of `ch_gdp.csv`. However, how does the computer know that a line is end-

⁶https://en.wikipedia.org/wiki/Parsing#Computer_languages

⁷In addition, if a field contains itself a comma (the comma is part of the data), the field needs to be surrounded by double-quotes ("). If a field contains double-quotes it also has to be surrounded by double-quotes, and the double quotes in the field must be preceded by an additional double-quote.

ing? By default most programs to work with text files do not show an explicit symbol for line endings but instead display the text on the next line. Under the hood, these line endings are, however, non-printable characters. We can see this when investigating `ch_gdp.csv` in the command-line terminal (via `xxd`):

```
xxd ch_gdp.csv
```

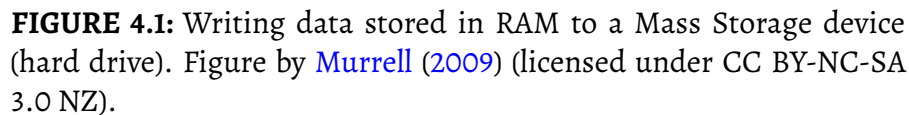
```
## 00000000: efbb bf79 6561 722c 6764 705f 6368 6662  ...year,gdp_chfb
## 00000010: 0d31 3938 302c 3138 340d 3139 3835 2c32  .1980,184.1985,2
## 00000020: 3434 0d31 3939 302c 3333 310d 3139 3935  44.1990,331.1995
## 00000030: 2c33 3734 0d32 3030 302c 3432 320d 3230  ,374.2000,422.20
## 00000040: 3035 2c34 3634                                05,464
```

When comparing the hexadecimal values with the characters they represent on the right side, we recognize that a full stop (.) is printed to the output right before every year. Moreover, when inspecting the hexadecimal code, we recognize that this . corresponds to `0d` in the hexadecimal code. `0d` is indeed the sequence of `0s` and `1` indicating the end of a line. Because this character does not actually correspond to a symbol printed on the screen, it is replaced by . in the printed output of the text.⁸

While CSV files have become a very common way to store ‘flat’/table-like data in plain text files, several similar formats can be encountered in practice. Most commonly they either use a different delimiter (for example, tabs/white space) to separate fields in a row, or fields are defined to consist of a fixed number of characters (so-called fixed-width formats). In addition, various more complex standards to store and transfer digital data are widely used to store data. Particularly in the context of web data (data stored and transferred online), files storing data in these formats (e.g., XML, JSON, YAML, etc.) are in the end, just plain text files. However, they contain a larger set of special characters/delimiters (or combinations of these) to indicate the structure of the data stored in them.

⁸The same applies to other sequences of `0s` and `1s` that do not correspond to a printable character.

The question of how to store digital data on computers also raises the question of storage capacity. Because every type of digital data can, in the end, only be stored as 0s and 1s, it makes perfect sense to define the smallest unit of information in computing as consisting of either a 0 or a 1. We call this basic unit a *bit* (from *binary digit*; abbrev. ‘b’). Recall that the decimal number 139 corresponds to the binary number 10001011. To store this number on a hard disk, we require a capacity of 8 bits or one *byte* (1 byte = 8 bits; abbrev. ‘B’). Historically, one byte encoded a single character of text (i.e., in the ASCII character encoding system). 4 bytes (or 32 bits) are called a *word*. The following figure illustrates this point.



- 1 kilobyte (KB) = $1000^1 \approx 2^{10}$ bytes
- 1 megabyte (MB) = $1000^2 \approx 2^{20}$ bytes
- 1 gigabyte (GB) = $1000^3 \approx 2^{30}$ bytes
- 1 terabyte (TB) = $1000^4 \approx 2^{40}$ bytes
- 1 petabyte (PB) = $1000^5 \approx 2^{50}$ bytes
- 1 exabyte (EB) = $1000^6 \approx 2^{60}$ bytes
- 1 zettabyte (ZB) = $1000^7 \approx 2^{70}$ bytes

1ZB = 100000000000000000000 bytes = 1 billion terabytes = 1 trillion gigabytes.

4.4 Data structures and data types in R

So far, we have focused on how data is stored on the hard disk. That is, we discovered how 'O's and 'I's correspond to characters (using specific standards: character encodings), how a sequence of characters in a text file is a useful way to store data on a hard disk, and how specific standards are used to structure the data in a meaningful way (using special characters). When thinking about how to store data on a hard disk, we are usually concerned with how much space (in bytes) the data set needs, how well it is transferable/understandable, and how well we can retrieve information from it. All of these aspects go into the decision of what structure/format to choose etc.

However, none of these considers how we actively work with the data. For example, when we want to sum up the `gdp_chfb` column in `ch_gdp.csv` (the table example above), do we actually work within the CSV data structure? The answer is usually no.

Recall from the basics of data processing that data stored on the hard disk is loaded into RAM to work with (analysis, manipulation, cleaning, etc.). The question arises as to how data is structured in RAM? That is, what data structures/formats are used to work with the data actively?

We distinguish two basic characteristics:

1. Data **types**: integers; real numbers ('numeric values', floating point numbers); text ('string', 'character values').
2. Basic **data structures** in RAM: - *Vectors* - *Factors* - *Arrays/matrices* - *Lists* - *Data frames* (very R-specific)

Depending on the data structure/format stored in on the hard disk, the data will be more or less usefully represented in one of the above structures in RAM. For example, in the R language it is quite common to represent data stored in CSVs on disk as *data frames* in RAM. Similarly, it is quite common to represent a more complex format on disk, such as JSON, as a nested list in RAM.

Importing data from the hard disk (or another mass storage device) into RAM in order to work with it in R essentially means reading the sequence of characters (in the end, of course, 0s and 1s) and mapping them, given the structure they are stored in, into one of these structures for representation in RAM.⁹.

4.4.1 Data types

From the previous chapters, we know that digital data (0s and 1s) can be interpreted in different ways depending on the *format* it is stored in. Similarly, data loaded into RAM can be interpreted differently by R depending on the data *type*. Some operators or functions in R only accept data of a specific type as arguments. For example, we can store the numeric values 1.5 and 3 in the variables `a` and `b`, respectively.

```
a <- 1.5  
b <- 3
```

R interprets this data as type `double` (class 'numeric'; a 'double precision floating point number'):

```
typeof(a)
```

```
## [1] "double"
```

```
class(a)
```

```
## [1] "numeric"
```

Given that these bytes of data are interpreted as numeric, we can use operators (here: math operators) that can work with such types:

```
a + b
```

```
## [1] 4.5
```

⁹In the chapter on data gathering and data import, we will cover this crucial step in detail.

If we define `a` and `b` as follows, R will interpret the values stored in `a` and `b` as text (character).

```
a <- "1.5"  
b <- "3"
```

```
typeof(a)
```

```
## [1] "character"
```

```
class(a)
```

```
## [1] "character"
```

Now the same line of code as above will result in an error:

```
a + b
```

```
## Error in a + b: non-numeric argument to binary operator
```

The reason is that an operator `+` expects numeric or integer values as arguments. When importing data sets with many different variables (columns), it is thus necessary to ensure that each column is interpreted in the intended way. That is, we have to make sure R is assigning the right type to each of the imported variables. Usually, this is done automatically. However, with large and complex data sets, the automatic recognition of data types when importing data can fail when the data is not perfectly cleaned/prepared (in practice, this is very often the case).

4.4.2 Data structures

For now, we have only looked at individual bytes of data. An entire data set can consist of gigabytes of data containing text and numeric values. How can such collections of data values be represented in R? Here, we look at the main data structures implemented in R. All of these structures will play a role in some of the hands-on exercises.

4.4.2.1 Vectors

Vectors are collections of values of the same type. They can contain either all numeric values or all character values. We will use the following symbol to refer to vectors.



1
2
3

FIGURE 4.2: Illustration of a numeric vector (symbolic). Figure by [Murrell \(2009\)](#) (licensed under CC BY-NC-SA 3.0 NZ).

For example, we can initiate a character vector containing the names of persons:

```
persons <- c("Andy", "Brian", "Claire")
persons
```

```
## [1] "Andy" "Brian" "Claire"
```

And we can initiate a numeric vector with the age of these persons:

```
ages <- c(24, 50, 30)
ages
```

```
## [1] 24 50 30
```

4.4.2.2 Factors

Factors are sets of categories. Thus, the values come from a fixed set of possible values.



F
M
F

FIGURE 4.3: Illustration of a factor (symbolic). Figure by [Murrell \(2009\)](#) (licensed under CC BY-NC-SA 3.0 NZ).

For example, we might want to initiate a factor that indicates the gender of a number of people.

```
gender <- factor(c("Male", "Male", "Female"))
gender
```

```
## [1] Male   Male   Female
## Levels: Female Male
```

4.4.2.3 Matrices/Arrays

Matrices are two-dimensional collections of values, arrays of higher-dimensional collections of values of the same type.

1	4	7
2	5	8
3	6	9

FIGURE 4.4: Illustration of a numeric matrix (symbolic). Figure by [Murrell \(2009\)](#) (licensed under CC BY-NC-SA 3.0 NZ).

For example, we can initiate a three-row/two-column numeric matrix as follows

```
my_matrix <- matrix(c(1,2,3,4,5,6), nrow = 3)
my_matrix
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

And a three-dimensional numeric array as follows.

```
my_array <- array(c(1,2,3,4,5,6,7,8), dim = c(2,2,2))
my_array
```

```
## , , 1
##
```

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

4.4.2.4 Data frames, tibbles, and data tables

Data frames are the typical representation of a (table-like) data set in R. Each column can contain a vector of a given data type (or a factor), but all columns need to be of identical length. Thus in the context of data analysis, we would say that each row of a data frame contains an observation, and each column contains a characteristic of this observation.

1	F	a
2	M	b
3	F	c

FIGURE 4.5: Illustration of a data frame (symbolic). Figure by [Murrell \(2009\)](#) (licensed under CC BY-NC-SA 3.0 NZ).

The historical implementation of data frames in R is not very appropriate to work with large datasets.¹⁰ These days there are new implementations of the data frame concept in R provided by different packages, which aim at making data processing based on ‘data frames’ faster. One is called `tibbles`, implemented and used in the `tidyverse` packages. The other is called `data table`, implemented in the `data. table`-package. In this book, we will encounter primarily classical `data. frame` and `tibble` (however, there will also be some hints to tutorials with `data. table`). In any case, once we understand data frames in general, working with `tibble` and `data. table` is quite easy because functions that accept

¹⁰In the early days of R, this was not an issue because datasets that are rather large by today’s standards (in the Gigabytes) could not have been handled properly by normal computers anyway (due to a lack of RAM).

classical data frames as arguments also accept those newer implementations.

Here is how we define a data frame in R, based on the examples of vectors and factors shown above.

```
df <- data.frame(person = persons, age = ages, gender = gender)
df
```

```
##   person age gender
## 1   Andy  24   Male
## 2  Brian  50   Male
## 3 Claire  30 Female
```

4.4.2.5 Lists

Unlike data frames, lists can contain different data types in each element. Moreover, they even can contain different data structures of *different dimensions* in each element. For example, a list could contain different other lists, data frames, and vectors with differing numbers of elements.



FIGURE 4.6: Illustration of a data frame (symbolic). Figure by [Murrell \(2009\)](#) (licensed under CC BY-NC-SA 3.0 NZ).

This flexibility can easily be demonstrated by combining some of the data structures created in the examples above:

```
my_list <- list(my_array, my_matrix, df)
my_list
```

```
## [[1]]
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
```

```
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
##
##
## [[2]]
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
##
## [[3]]
##   person age gender
## 1   Andy  24   Male
## 2  Brian  50   Male
## 3 Claire  30 Female
```


5

High-Dimensional Data

So far, we have only looked at data structured in a flat/table-like representation (e.g. CSV files). In applied econometrics/statistics, it is common to only work with data sets stored in such format. Data manipulation, filtering, aggregation, etc. presupposes data in a table-like format. Hence, storing data in this format makes perfect sense.

As we observed in the previous chapter, the CSV structure has some disadvantages when representing more complex data in a text file. This is in particular true if the data contains nested observations (i.e., hierarchical structures). While a representation in a CSV file is theoretically possible, it is often far from practical to use other formats for such data. On the one hand, it is likely less intuitive to read the data correctly. On the other hand, storing the data in a CSV file might introduce a lot of redundancy. That is, the identical values of some variables would have to be repeated in the same column. The following code block illustrates this point for a data set on two families (([Murrell, 2009](#)), p. 116).

father	mother	name	age	gender
		John	33	male
		Julia	32	female
John	Julia	Jack	6	male
John	Julia	Jill	4	female
John	Julia	John jnr	2	male
		David	45	male
		Debbie	42	female
David	Debbie	Donald	16	male
David	Debbie	Dianne	12	female

From simply looking at the data, we can make the best guess which observations belong together (are one family). However, the implied

hierarchy is not apparent at first sight. While it might not matter too much that several values have to be repeated several times in this format, given that this data set is so small, the repeated values can become a problem when the data set is much larger. For each time `John` is repeated in the `father` column, we use 4 bytes of memory. Suppose millions of people are in this data set, and we have to transfer this data set very often over a computer network. In that case, these repetitions can become quite costly (as we would need more storage capacity and network resources).

Issues with complex/hierarchical data (with several observation types), intuitive human readability (self-describing), and efficiency in storage, as well as transfer, are all of great importance on the Web. In the context of web technologies, several data formats have been put forward to address these issues. Here, we discuss the two most prominent of these formats: Extensible Markup Language (XML)¹ and JavaScript Object Notation (JSON)².

5.1 Deciphering XML

Before going into more technical details, let's try to figure out the basic logic behind the XML format by simply looking at some raw example data. For this, we turn to the Point Nemo case study in (Murrell, 2009). The following code block shows the upper part of the data set downloaded from NASA's LAS server (here in a CSV-type format).

```
VARIABLE : Monthly Surface Clear-
sky Temperature (ISCCP) (Celsius)
FILENAME : ISCCPMonthly_avg.nc
FILEPATH : /usr/local/fer_data/data/
BAD FLAG : -1.E+34
SUBSET   : 48 points (TIME)
LONGITUDE: 123.8W(-123.8)
LATITUDE : 48.8S
```

¹<https://en.wikipedia.org/wiki/XML>

²<https://en.wikipedia.org/wiki/JSON>

```

123.8W
16-JAN-1994 00    9.200012
16-FEB-1994 00    10.70001
16-MAR-1994 00    7.5
16-APR-1994 00    8.100006

```

Below, the same data is now displayed in XML format. Note that in both cases, the data is simply stored in a text file. However, it is stored in a format that imposes a different *structure* on the data.

```

<?xml version="1.0"?>
<temperatures>
  <variable>Monthly Surface Clear-sky Temperature (ISCCP) (Celsius)</variable>
  <filename>ISCCPMonthly_avg.nc</filename>
  <filepath>/usr/local/fer_data/data/</filepath>
  <badflag>-1.E+34</badflag>

  <subset>48 points (TIME)</subset>
  <longitude>123.8W(-123.8)</longitude>
  <latitude>48.8S</latitude>
  <case date="16-JAN-1994" temperature="9.200012" />
  <case date="16-FEB-1994" temperature="10.70001" />
  <case date="16-MAR-1994" temperature="7.5" />
  <case date="16-APR-1994" temperature="8.100006" />

  ...
</temperatures>

```

What features does the format have? What is its logic? Is there room for improvement? The XML data structure becomes more apparent by using indentation and code highlighting.

```

<?xml version="1.0"?>
  <temperatures>
    <variable>Monthly Surface Clear-sky Temperature (ISCCP) (Celsius)</variable>
    <filename>ISCCPMonthly_avg.nc</filename>
    <filepath>/usr/local/fer_data/data/</filepath>
    <badflag>-1.E+34</badflag>
  
```

```

<subset>48 points (TIME)</subset>
<longitude>123.8W(-123.8)</longitude>
<latitude>48.8S</latitude>
<case date="16-JAN-1994" temperature="9.200012" />
<case date="16-FEB-1994" temperature="10.70001" />
<case date="16-MAR-1994" temperature="7.5" />
<case date="16-APR-1994" temperature="8.100006" />

...
</temperatures>

```

First, note how special characters are used to define the document's structure. We notice that < and >, containing some text labels seem to play a key role in defining the structure. These building blocks are called 'XML tags'. We are free to choose what tags we want to use. In essence, we can define them ourselves to most properly describe the data. Moreover, the example data reveals the flexibility of XML to depict hierarchical structures.

The actual content we know from the CSV-type example above is nested between the 'temperatures'-tags, indicating what the data is about.

```

<temperatures>
...
</temperatures>

```

Comparing the actual content between these tags with the CSV-type format above, we further recognize that there are two principal ways to link variable names to values.

```

<variable>Monthly Surface Clear-sky Temperature (ISCCP) (Celsius)</variable>
<filename>ISCCPMonthly_avg.nc</filename>
<filepath>/usr/local/fer_data/data/</filepath>
<badflag>-1.E+34</badflag>
<subset>48 points (TIME)</subset>

```

```
<longitude>123.8W(-123.8)</longitude>
<latitude>48.8S</latitude>
<case date="16-JAN-1994" temperature="9.200012" />
<case date="16-FEB-1994" temperature="10.70001" />
<case date="16-MAR-1994" temperature="7.5" />
<case date="16-APR-1994" temperature="8.100006" />
```

One way is to define opening and closing XML-tags with the variable name and surround the value with them, such as in `<filename>ISCCPMonthly_avg.nc</filename>`. Another way would be to encapsulate the values within one tag by defining tag attributes such as in `<case date="16-JAN-1994" temperature="9.200012" />`. In many situations, both approaches can make sense. For example, the way the temperature measurements are encoded in the example data set is based on the tag-attributes approach:

```
<case date="16-JAN-1994" temperature="9.200012" />
<case date="16-FEB-1994" temperature="10.70001" />
<case date="16-MAR-1994" temperature="7.5" />
<case date="16-APR-1994" temperature="8.100006" />
```

We could rewrite this by only using XML tags and no attributes:

```
<cases>
  <case>
    <date>16-JAN-1994</date>
    <temperature>9.200012</temperature>
  </case>
  <case>
    <date>16-FEB-1994</date>
    <temperature>10.70001</temperature>
  </case>
  <case>
    <date>16-MAR-1994</date>
    <temperature>7.5</temperature>
  </case>
</cases>
```

```
<case/>
<case>
  <date>16-APR-1994<date/>
  <temperature>8.100006<temperature/>
<case/>
<cases/>
```

As long as we follow the basic XML syntax, both versions are valid, and XML parsers can read them equally well.

Note the key differences between storing data in XML format in contrast to a flat, table-like format such as CSV:

- Storing the actual data and metadata in the same file is straightforward (as the above example illustrates). Storing metadata in the first lines of a CSV file (such as in the example above) is theoretically possible. However, by doing so, we break the CSV syntax, and a CSV-parser would likely break down when reading such a file (recall the simple CSV parsing algorithm). More generally, we can represent much more *complex (multi-dimensional)* data in XML files than what is possible in CSVs. In fact, the nesting structure can be arbitrarily complex as long as the XML syntax is valid.
- The XML syntax is largely self-explanatory and thus both *machine-readable and human-readable*. That is, not only can parsers/computers more easily handle complex data structures, but human readers can intuitively understand what the data is all about by looking at the raw XML file.

A potential drawback of storing data in XML format is that variable names (in tags) are repeated. Since each tag consists of a couple of bytes, this can be highly inefficient compared to a table-like format where variable names are only defined once. Typically, this means that if the data at hand is only two-dimensional (observations/variables), a CSV format makes more sense.

5.2 Deciphering JSON

In many web applications, JSON serves the same purpose as XML³. An obvious difference between the two conventions is that JSON does not use tags but attribute-value pairs to annotate data. The following code example shows how the same data can be represented in XML or in JSON (example code taken from <https://en.wikipedia.org/wiki/JSON>):

XML:

```
<person>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <age>25</age>
  <address>
    <streetAddress>21 2nd Street</streetAddress>
    <city>New York</city>
    <state>NY</state>
    <postalCode>10021</postalCode>
  </address>
  <phoneNumber>
    <type>home</type>
    <number>212 555-1234</number>
  </phoneNumber>
  <phoneNumber>
    <type>fax</type>
    <number>646 555-4567</number>
  </phoneNumber>
  <gender>
    <type>male</type>
  </gender>
</person>
```

JSON:

³Not that programs running on the server side are frequently capable of returning the same data in either format

```
{ "firstName": "John",  
  "lastName": "Smith",  
  "age": 25,  
  "address": {  
    "streetAddress": "21 2nd Street",  
    "city": "New York",  
    "state": "NY",  
    "postalCode": "10021"  
  },  
  "phoneNumber": [  
    {  
      "type": "home",  
      "number": "212 555-1234"  
    },  
    {  
      "type": "fax",  
      "number": "646 555-4567"  
    }  
  ],  
  "gender": {  
    "type": "male"  
  }  
}
```

Note that despite the syntax differences, the similarities regarding the nesting structure are visible in both formats. For example, `postalCode` is embedded in `address`, `firstName` and `lastName` are at the same nesting level, etc.

Both XML and JSON are predominantly used to store rather complex, multi-dimensional data. Because they are both human- and machine-readable, these formats are often used to transfer data between applications and users on the Web. Therefore, we encounter these data formats particularly often when we collect data from online sources. Moreover, in the economics/social science context, large and complex data sets ('Big Data') often come along in these formats exactly because

of the increasing importance of the Internet as a data source for empirical research in these disciplines.

5.3 Parsing XML and JSON in R

As in the case of CSV-like formats, there are several parsers in R that we can use to read XML and JSON data. The following examples are based on the example code shown above (the two text-files `persons.json` and `persons.xml`)

```
# load packages
library(xml2)

# parse XML, represent XML document as R object
xml_doc <- read_xml("persons.xml")
xml_doc

## {xml_document}
## <person>
## [1] <firstName>John</firstName>
## [2] <lastName>Smith</lastName>
## [3] <age>25</age>
## [4] <address>\n  <streetAddress>21 2nd Street</stree ...
## [5] <phoneNumber>\n  <type>home</type>\n  <number>21 ...
## [6] <phoneNumber>\n  <type>fax</type>\n  <number>646 ...
## [7] <gender>\n  <type>male</type>\n</gender>
```

```
# load packages
library(jsonlite)

# parse the JSON-document shown in the example above
json_doc <- from_JSON("persons.json")
```

```
# check the structure
str(json_doc)

## List of 6
## $ firstName : chr "John"
## $ lastName  : chr "Smith"
## $ age       : int 25
## $ address   :List of 4
## ..$ streetAddress: chr "21 2nd Street"
## ..$ city        : chr "New York"
## ..$ state       : chr "NY"
## ..$ postalCode  : chr "10021"
## $ phoneNumber:'data.frame': 2 obs. of 2 variables:
## ..$ type : chr [1:2] "home" "fax"
## ..$ number: chr [1:2] "212 555-1234" "646 555-4567"
## $ gender   :List of 1
## ..$ type: chr "male"
```

5.4 HTML

Recall the data processing example in which we investigated how a webpage can be downloaded, processed, and stored via the R command line. The code constituting a webpage is written in HyperText Markup Language (HTML)⁴, designed to be read in a web browser. Thus, HTML is predominantly designed for visual display in browsers, not for storing data. HTML is used to annotate content and define the hierarchy of content in a document to tell the browser how to display (‘render’) this document on the computer screen. Interestingly for us, its structure is very close to that of XML. However, the tags that can be used are strictly pre-defined and are aimed at explaining the structure of a website.

While not intended as a format to store and transfer data, HTMLdocu-

⁴<https://en.wikipedia.org/wiki/HTML>

ments (webpages) have de facto become a very important data source for many data science applications both in industry and academia.⁵ Note how the two key concepts of computer code and digital data (both residing in a text file) are combined in an HTML document. From a web designer's perspective, HTML is a tool to design the layout of a webpage (and the resulting HTML document is rather seen as *code*). On the other hand, from a data scientist's perspective, HTML gives the data contained in a webpage (the actual content) a certain degree of structure which can be exploited to systematically extract the data from the webpage. In the context of HTML documents/webpages as data sources, we thus also speak of 'semi-structured data': a webpage can contain an HTML table (structured data) but likely also contains just raw text (unstructured data). In the following, we explore the basic syntax of HTML by building a simple webpage.

5.4.1 Write a simple webpage with HTML

We start by opening a new text-file in RStudio (File->New File->TextFile). On the first line, we tell the browser what kind of document this is with the `<!DOCTYPE>` declaration set to `HTML`. In addition, the content of the whole HTML document must be put within `<html>` and `</html>`, which represents the 'root' of the HTML document. In this, you already recognize the typical (XML-like) annotation style in HTML with so-called HTML tags, starting with `<` and ending with `>` or `/>` in the case of the closing tag, respectively. What is defined between two tags is either another HTML tag or the actual content. An HTML document usually consists of two main components: the head (everything between `<head>` and `</head>`) and the body. The head typically contains metadata describing the whole document like, the title of the document: `<title>hello, world</title>`. The body (everything between `<body>` and `</body>`) contains all kinds of specific content: text, images, tables, links, etc. In our very simple example, we add a few words of plain text. We can now save this text document as `mysite.html` and open it in a web browser.

⁵The systematic collection and extraction of data from such web sources (often referred to as Web Data Mining) goes well beyond the scope of this book.

```
<!DOCTYPE html>

<html>
  <head>
    <title>hello, world</title>
  </head>
  <body>
    <h2> hello, world </h2>
  </body>
</html>
```

From this example, we can learn a few important characteristics of HTML:

1. It becomes apparent how HTML is used to annotate/'mark up' data/text (with tags) to define the document's content, structure, and hierarchy. Thus if we want to know what the title of this document is, we have to look for the `<title>`-tag.
2. This systematic structuring adheres to the nesting principle: 'head' and 'body' are nested within the 'HTML' document, at the same hierarchy level. The 'title' is defined within the 'head', one level lower. In other words, the 'title' is a component of the 'head,' which is a component of the 'html' document. This logic of encapsulating one part in another holds true in all correctly defined HTML documents. This is the type of 'data structure' mentioned above, which can be used to extract specific parts of data from HTML documents in a systematic manner.
3. We recognize that HTML code essentially expresses what is what in a document (note the similarity of the concept to storing data in XML). HTML code does not contain explicit instructions like programming languages, telling the computer what to do. If an HTML document contains a link to another website, all that is needed, is to define (with HTML tag ``) that this is a link. We do not have to explic-

itly express something like “if the user clicks on a link, then execute this and that...”.

What to do with the HTML document is thus in the hands of the person who works with it. From the data scientist’s perspective, we need to know how to traverse and exploit the structure of an HTML document in order to systematically extract the specific content/data that we are interested in. We thus have to learn how to tell the computer (with R) to extract a specific part of an HTML document. And to do so, we have to acquire a basic understanding of the nesting structure implied by HTML (essentially the same logic as with XML).

One way to think about an HTML document is to imagine it as a tree-diagram, with `<html>...</html>` as the ‘root’, `<head>...</head>` and `<body>...</body>` as the ‘children’ of `<html>...</html>` (and ‘siblings’ of each other), `<title>...</title>` as the child of `<head>...</head>`, etc. Figure 5.1 illustrates this point.

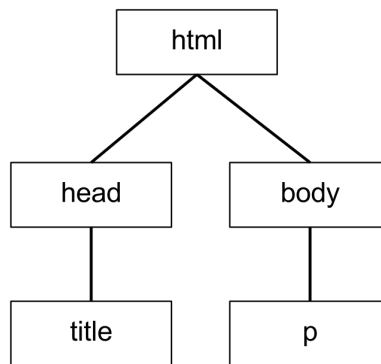


FIGURE 5.1: HTML (DOM) tree diagram (by Lubaochuan 2014, licensed under the Creative Commons Attribution-Share Alike 4.0 International license).

The illustration of the nested structure can help to understand how we can instruct the computer to find/extract a specific part of the data from such a document. In the following exercise, we revisit the example shown in the chapter on data processing to do exactly this.

5.4.2 Two ways to read a webpage into R

In this example, we look at Wikipedia's Economy of Switzerland page⁶, which contains the table depicted in Figure @ref(fig: swiss).

Year	GDP (billions of CHF)	US Dollar Exchange
1980	184	1.67 Francs
1985	244	2.43 Francs
1990	331	1.38 Francs
1995	374	1.18 Francs
2000	422	1.68 Francs
2005	464	1.24 Francs
2006	491	1.25 Francs
2007	521	1.20 Francs
2008	547	1.08 Francs
2009	535	1.09 Francs
2010	546	1.04 Francs
2011	659	0.89 Francs
2012	632	0.94 Francs
2013	635	0.93 Francs
2014	644	0.92 Francs
2015	646	0.96 Francs
2016	659	0.98 Francs
2017	668	1.01 Francs
2018	694	1.00 Francs

FIGURE 5.2: Source: https://en.wikipedia.org/wiki/Economy_of_Switzerland.

As in the example shown in the data processing chapter ('world population clock'), we first tell R to read the lines of text from the HTML document that constitutes the webpage.

```
swiss_econ <- readLines("https://en.wikipedia.org/wiki/Economy_of_Switzerland")

## Warning in readLines("https://en.wikipedia.org/
## wiki/Economy_of_Switzerland"): incomplete final
## line found on 'https://en.wikipedia.org/wiki/
## Economy_of_Switzerland'
```

⁶https://en.wikipedia.org/wiki/Economy_of_Switzerland

And we can check if everything worked out well by having a look at the first lines of the web page's HTML code:

```
head(swiss_econ)
```

```
## [1] "<!DOCTYPE html>"
## [2] "<html class=\"client-nojs\" lang=\"en\" dir=\"ltr\">"
## [3] "<head>"
## [4] "<meta charset=\"UTF-8\"/>"
## [5] "<title>Economy of Switzerland - Wikipedia</title>"
## [6] "<script>document.documentElement.className=\"client-
js\";RLCONF={\"wgBreakFrames\":false,\"wgSeparatorTransformTable\":[\"\", \"\"],\"wgDigitTransl
9e23-4e2f-96db-a907014176c8\", \"wgCSPNonce\":false,\"wgCanonicalNamespace\":\"\", \"wgCanonical
language sources (de)\", \"Articles with German-
language sources (de)\", \"Webarchive template wayback links\", \"Articles with French-
language sources (fr)\", \"Articles with short description\", \"Short description is different from
```

The next thing we do is to look at how we can filter the webpage for certain information. For example, we search in the source code for the line that contains the part of the webpage with the table showing data on the Swiss GDP:

```
line_number <- grep('US Dollar Exchange', swiss_econ)
```

Recall: we ask R on which line in `swiss_econ` (the source code stored in RAM) the text `US Dollar Exchange` is and store the answer (the line number) in RAM under the variable name `line_number`.

Then, we check on which line the text was found.

```
line_number
```

```
## [1] 230
```

Knowing that the R object `swiss_econ` is a character vector (with each element containing one line of HTML code as a character string), we can look at this particular code:

```
swiss_econ[line_number]
```

```
## [1] "<th>US Dollar Exchange"
```

Note that this rather primitive approach is ok to extract a chunk of code from an HTML document, but it is far from practical when we want to extract specific parts of data (the actual content, not including HTML code). So far, we have completely ignored that the HTML tags give some structure to the data in this document. That is, we simply have read the entire document line by line, not making a difference between code and data. The approach to filter the document could have equally well been taken for a plain text file.

If we want to exploit the structure given by HTML, we need to *parse* the HTML when reading the webpage into R. We can do this with the help of functions provided in the `rvest` package:

```
# install package if not yet installed
# install.packages("rvest")

# load the package
library(rvest)
```

After loading the package, we read the webpage into R, but this time using a function that parses the HTML code:

```
# parse the webpage, show the content
swiss_econ_parsed <- read_html("https://en.wikipedia.org/wiki/Economy_of_Switzerland")
swiss_econ_parsed
```

```
## {html_document}
## <html class="client-nojs" lang="en" dir="ltr">
## [1] <head>\n<meta http-equiv="Content-Type" content= ...
## [2] <body class="mediawiki ltr sitedir-ltr mw-hide-e ...
```

Now we can easily separate the data/text from the HTML code. For ex-

ample, we can extract the HTML table containing the data we are interested in as data frames.

```
tab_node <- html_node(swiss_econ_parsed, xpath = "//*[@id='mw-content-text']/div/table[2]")
tab <- html_table(tab_node)
tab
```

```
## # A tibble: 19 x 3
##   Year `GDP (billions of CHF)` `US Dollar Exchange`
##   <int>          <int> <chr>
## 1  1980          184 1.67 Francs
## 2  1985          244 2.43 Francs
## 3  1990          331 1.38 Francs
## 4  1995          374 1.18 Francs
## 5  2000          422 1.68 Francs
## 6  2005          464 1.24 Francs
## 7  2006          491 1.25 Francs
## 8  2007          521 1.20 Francs
## 9  2008          547 1.08 Francs
## 10 2009          535 1.09 Francs
## 11 2010          546 1.04 Francs
## 12 2011          659 0.89 Francs
## 13 2012          632 0.94 Francs
## 14 2013          635 0.93 Francs
## 15 2014          644 0.92 Francs
## 16 2015          646 0.96 Francs
## 17 2016          659 0.98 Francs
## 18 2017          668 1.01 Francs
## 19 2018          694 1.00 Francs
```



6

Data Sources, Data Gathering, Data Import

In this chapter, we put the key concepts learned so far (text files for data storage, parsers, encoding, data structures) together and apply them to master the first key bottleneck in the data pipeline: how to *import* raw data from various sources and *export/store* them for further processing in the pipeline.

6.1 Sources/formats

In the previous chapters, we learned how data is stored in text files and how different data structures/formats/syntaxes help to organize the data in these files. Along the way, we have encountered key data formats that are used in various settings to store and transfer data:

- CSV (typical for rectangular/table-like data)
- Variants of CSV (tab-delimited, fix length, etc.)
- XML and JSON (useful for complex/high-dimensional data sets)
- HTML (a markup language to define the structure and layout of web-pages)
- Unstructured text

Depending on the *data source*, data might come in one or the other form. With the increasing importance of the Internet as a data source for economic research, properly handling XML, JSON, and HTML is becoming more important. However, in applied economic research, various other formats can be encountered:

- Excel spreadsheets (.xls)
- Formats specific to statistical software packages (SPSS: .sav, STATA: .dat, etc.)

- Built-in R datasets
- Binary formats

While we will cover/revisit how to import all of these formats here, it is important to keep in mind that the learned fundamental concepts are as important (or even more important) than knowing which function to call in R for each of these cases. New formats might evolve and become more relevant in the future for which no R function yet exists. However, the underlying logic of how formats to structure data work will hardly change.

6.2 Data gathering procedure

Before we set out to gather/import data from diverse sources, we should start organizing the procedure in an R script. This script will be the beginning of our pipeline!

First, open a new R script in RStudio and save it as `import_data.R` in your `code` folder. Take your time to meaningfully describe what the script is all about in the first few lines of the file:

```
#####
# Data Handling Course: Example Script for Data Gathering and Import
#
# Imports data from ...
# Input: links to data sources (data comes in ... format)
# Output: cleaned data as CSV
#
# U. Matter, St.Gallen, 2019
#####
```

RStudio recognizes different sections in a script, whereby section headers are indicated by `-----`. This helps to organize the script into different tasks further. Usually, it makes sense to start with a ‘meta’ section in which all necessary packages are loaded and fix variables initiated.

```
#####
# Data Handling Course: Example Script for Data Gathering and Import
#
# Imports data from ...
# Input: links to data sources (data comes in ... format)
# Output: cleaned data as CSV
#
# U. Matter, St.Gallen, 2019
#####

# SET UP -----
# load packages
library(tidyr)

# set fix variables
INPUT_PATH <- "/rawdata"
OUTPUT_FILE <- "/final_data/datafile.csv"
```

Finally, we add sections with the actual code (in the case of a data import script, maybe one section per data source).

```
#####
# Project XY: Data Gathering and Import
#
# This script is the first part of the data pipeline of project XY.
# It imports data from ...
# Input: links to data sources (data comes in ... format)
# Output: cleaned data as CSV
#
# U. Matter, St.Gallen, 2019
#####

# SET UP -----
# load packages
```

```
library(tidyr)

# set fix variables
INPUT_PATH <- "/rawdata"
OUTPUT_FILE <- "/final_data/datafile.csv"

# IMPORT RAW DATA FROM CSVs -----
```

6.3 Loading/importing rectangular data¹

6.3.1 Loading built-in datasets

We start with the simplest case of loading/importing data. The basic R installation provides some example datasets to try out R's statistics functions. In the introduction to visualization techniques with R and the statistics examples in the chapters to come, we will rely on some of these datasets for simplicity. Note that the usage of these simple datasets shipped with basic R are very helpful when practicing/learning R on your own. Many R packages use these datasets over and over again in their documentation and examples. Moreover, extensive documentations and tutorials online also use these datasets (see for example the *ggplot2* documentation²). And, they are very useful when searching help on Stackoverflow³ in the context of data analysis/manipulation with R, as you should provide a code example based on some data that everybody can easily load and access.

In order to load such datasets, simply use the `data()`-function:

```
data(swiss)
```

In this case, we load a dataset called `swiss`. After loading it, the data is

¹This section is based on [Matter \(2018a\)](#).

²<https://ggplot2.tidyverse.org/>

³<https://stackoverflow.com/questions/tagged/r>

stored in a variable of the same name as the dataset (here 'swiss'). We can inspect it and have a look at the first few rows:

```
# inspect the structure
str(swiss)
```

```
## 'data.frame':    47 obs. of  6 variables:
## $ Fertility      : num  80.2 83.1 92.5 85.8 76.9 76.1 83.8 92.4 82.4 82.9 ...
## $ Agriculture    : num  17 45.1 39.7 36.5 43.5 35.3 70.2 67.8 53.3 45.2 ...
## $ Examination    : int   15 6 5 12 17 9 16 14 12 16 ...
## $ Education      : int   12 9 5 7 15 7 7 8 7 13 ...
## $ Catholic       : num   9.96 84.84 93.4 33.77 5.16 ...
## $ Infant.Mortality: num  22.2 22.2 20.2 20.3 20.6 26.6 23.6 24.9 21 24.4 ...
```

```
# look at the first few rows
head(swiss)
```

```
##           Fertility Agriculture Examination
## Courtelary      80.2         17.0         15
## Delemont        83.1         45.1          6
## Franches-Mnt    92.5         39.7          5
## Moutier         85.8         36.5         12
## Neuveville      76.9         43.5         17
## Porrentruy      76.1         35.3          9
##           Education Catholic Infant.Mortality
## Courtelary      12      9.96             22.2
## Delemont         9     84.84             22.2
## Franches-Mnt     5     93.40             20.2
## Moutier          7     33.77             20.3
## Neuveville      15      5.16             20.6
## Porrentruy       7     90.57             26.6
```

To get a list of all the built-in datasets, type `data()` into the console and hit enter. To get more information about a given dataset, use the help function (e.g., `?swiss`)

6.3.2 Importing rectangular data from text-files

In most cases of applying R for data analysis, students and researchers rely on importing data from files stored on the hard disk. Typically, such datasets are stored in a text file format such as ‘Comma Separated Values’ (CSV). In economics, one also frequently encounters data stored in specific formats of commercial statistics/data analysis packages such as SPSS or STATA. Moreover, when collecting data on your own, you might rely on a spreadsheet tool like Microsoft Excel. Data from all these formats can easily be imported into R (in some cases, additional packages have to be loaded, though). Thereby, what happens ‘under the hood’ is essentially the same for all of these formats. Somebody has implemented the respective *parser* as an R function that accepts a character string with the path or URL to the data source as input.

6.3.2.1 Comma Separated Values (CSV)

Recall how in this format, data values of one observation are stored in one row of a text file, while commas separate the variables/columns. For example, the following code block shows how the first two rows of the `swiss`-dataset would look like when stored in a CSV:

```
"District","Fertility","Agriculture","Examination","Education","Catholic","Infant.Mortality"  
"Courtelary",80.2,17,15,12,9.96,22.2
```

The function `read.csv()` imports such files from disk into R (in the form of a `data frame`). In this example, the `swiss`-dataset is stored locally on our disk in the folder `data`:

```
swiss_imported <- read.csv("data/swiss.csv")
```

Alternatively, we could use the newer `read_csv()` function, which would return a `tibble`.

6.3.2.2 Spreadsheets/Excel

To read excel spreadsheets, we need to install an additional R package called `readxl`.


```
# install the package
install.packages("readxl")
```

Then we load this additional package (‘library’) and use the package’s `read_excel()`-function to import data from an excel-sheet. In the example below, the same data as above is stored in an excel-sheet called `swiss.xlsx`, again in a folder called `data`.

```
## New names:
## * `` -> `...1`
```

```
# load the package
library(readxl)

# import data from a spreadsheet
swiss_imported <- read_excel("data/swiss.xlsx")
```

6.3.2.3 Data from other data analysis software

The R packages `foreign` and `haven` contain functions to import data from formats used in other statistics/data analysis software, such as SPSS and STATA.

In the following example we use `haven`’s `read_spss()` function to import a version of the `swiss`-dataset stored in SPSS’ `.sav`-format (again stored in the folder called `data`).

```
# install the package (if not yet installed):
# install.packages("haven")

# load the package
library(haven)

# read the data
swiss_imported <- read_spss("data/swiss.sav")
```

6.4 Import and parse with `readr`⁴

The `readr` package is automatically installed and loaded with the installation/loading of `tidyverse`. It provides a set of functions to read different types of rectangular data formats and is usually more robust and faster than similar functions in the basic R distribution. Each of these functions expects either a character string with a path pointing to a file or a character string directly containing the data.

6.4.1 Basic usage of `readr` functions

For example, we can parse the first lines of the `swiss` dataset directly like this.

```
library(readr)
```

```
##
## Attaching package: 'readr'

## The following object is masked from 'package:rvest':
##
##      guess_encoding
```

```
read_csv('"District","Fertility","Agriculture","Examination","Education","Catholic","Infant.Mo
"Courtelary",80.2,17,15,12,9.96,22.2')
```

```
## Rows: 1 Columns: 7

## -- Column specification -----
## Delimiter: ","
## chr (1): District
## dbl (6): Fertility, Agriculture, Examination, Educa...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

⁴This is a summary of Chapter 8 in [Wickham and Grolemund \(2017\)](#).

```
## # A tibble: 1 x 7
##   District Fertility Agricu~1 Exami~2 Educa~3 Catho~4
##   <chr>      <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 Courtelary 80.2      17      15      12      9.96
## # ... with 1 more variable: Infant.Mortality <dbl>,
## # and abbreviated variable names 1: Agriculture,
## # 2: Examination, 3: Education, 4: Catholic
## # i Use `colnames()` to see all variable names
```

or read the entire `swiss` dataset by pointing to the file

```
swiss <- read_csv("data/swiss.csv")
```

```
## Rows: 47 Columns: 7
## -- Column specification -----
## Delimiter: ","
## chr (1): District
## dbl (6): Fertility, Agriculture, Examination, Educa...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

In either case, the result is a tibble:

```
swiss
```

```
## # A tibble: 47 x 7
##   District Ferti~1 Agric~2 Exami~3 Educa~4 Catho~5
##   <chr>      <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 Courtelary 80.2      17      15      12      9.96
## 2 Delemont   83.1     45.1      6       9     84.8
## 3 Franches-Mnt 92.5     39.7      5       5     93.4
## 4 Moutier    85.8     36.5     12       7     33.8
## 5 Neuveville 76.9     43.5     17      15     5.16
## 6 Porrentruy 76.1     35.3      9       7     90.6
## 7 Broye      83.8     70.2     16       7     92.8
## 8 Glane      92.4     67.8     14       8     97.2
## 9 Gruyere    82.4     53.3     12       7     97.7
```

```
## 10 Sarine          82.9    45.2      16      13    91.4
## # ... with 37 more rows, 1 more variable:
## #   Infant.Mortality <dbl>, and abbreviated variable
## #   names 1: Fertility, 2: Agriculture,
## #   3: Examination, 4: Education, 5: Catholic
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

The other `readr` functions have practically the same syntax and behavior. They are used for fixed-width files or CSV-type files with other delimiters than commas.

6.4.2 Parsing and data types

From inspecting the `swiss` tibble pointed out above, we recognize that `read_csv` not only correctly recognizes observations and columns (parses the CSV correctly) but also automatically guesses the data type of the values in each column. The first column is of type double, the second one of type integer, etc. That is, `read_csv` also parses each column-vector of the data set with the aim of recognizing which data type it is. For example, the data value "12:00" could be interpreted simply as a character string. Alternatively, it could also be interpreted as a `time` format.

If "12:00" is an element of the vector `c("12:00", "midnight", "noon")` it must be interpreted as a character string. If however it is an element of the vector `c("12:00", "14:30", "20:01")` we probably want R to import this as a `time` format. Now, how can `readr` handle the two cases? In simple terms, the package first guesses for each column vector which type is most appropriate. Then, it uses a couple of lower-level parsing functions (one written for each possible data type) in order to parse each column according to the respective guessed type. We can demonstrate this for the two example vectors above.

```
read_csv('A,B
12:00, 12:00
14:30, midnight
20:01, noon')
```

```
## Rows: 3 Columns: 2
## -- Column specification -----
## Delimiter: ","
## chr (1): B
## time (1): A
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

## # A tibble: 3 x 2
##   A      B
##   <time> <chr>
## 1 12:00  12:00
## 2 14:30  midnight
## 3 20:01  noon
```

Under the hood `read_csv()` used the `guess_parser()`-function to determine which type the two vectors likely contain:

```
guess_parser(c("12:00", "midnight", "noon"))
```

```
## [1] "character"
```

```
guess_parser(c("12:00", "14:30", "20:01"))
```

```
## [1] "time"
```

6.5 Importing web data formats

6.5.1 XML in R⁵

There are several XML-parsers already implemented in R packages specifically written for working with XML data. Thus, we do not have to understand the XML syntax in every detail to work with this data format in R. The already familiar package `xml2` (automatically loaded

⁵This section is based on [Matter \(2018b\)](#).

when loading `rvest`) provides the `read_xml()` function which we can use to read the exemplary XML-document.

```
# load packages
library(xml2)

# parse XML, represent XML document as R object
xml_doc <- read_xml("data/customers.xml")
xml_doc
```

```
## {xml_document}
## <customers>
## [1] <person>\n  <name>John Doe</name>\n  <orders>\n ...
## [2] <person>\n  <name>Peter Pan</name>\n  <orders>\n ...
```

The same package also has various functions to access, extract, and manipulate data from a parsed XML document. In the following code example, we have a look at the most useful functions for our purposes (see the package's vignette⁶ for more details).

```
# navigate through the XML document (recall the tree-like nested structure similar to HTML)
# navigate downwards
# 'customers' is the root node, persons are 'their children'
persons <- xml_children(xml_doc)
# navigate sideways
xml_siblings(persons)
```

```
## {xml_nodeset (2)}
## [1] <person>\n  <name>Peter Pan</name>\n  <orders>\n ...
## [2] <person>\n  <name>John Doe</name>\n  <orders>\n ...
```

```
# navigate upwards
xml_parents(persons)
```

```
## {xml_nodeset (1)}
```

⁶<https://cran.r-project.org/web/packages/xml2/vignettes/modification.html>

```
## [1] <customers>\n <person>\n <name>John Doe</nam ...
```

```
# find data via XPath
customer_names <- xml_find_all(xml_doc, xpath = ".//name")
# extract the data as text
xml_text(customer_names)
```

```
## [1] "John Doe" "Peter Pan"
```

6.5.2 JSON in R⁷

Again, we can rely on an R package (`jsonlite`) providing high-level functions to read, manipulate, and extract data when working with JSON documents in R. An important difference between working with XML- and HTML-documents is that XPath is not compatible with JSON. However, as `jsonlite` represents parsed JSON as R objects of class `list` and/or `data-frame`, we can work with the parsed document as with any other R-object of the same class. The following example illustrates this point.

```
# load packages
library(jsonlite)

# parse the JSON document shown in the example above
json_doc <- fromJSON("data/person.json")

# look at the structure of the document
str(json_doc)

# navigate the nested lists, extract data
# extract the address part
json_doc$address
# extract the gender (type)
json_doc$gender$type
```

```
## List of 6
```

⁷This section is based on [Matter \(2018b\)](#).

```
## $ firstName : chr "John"
## $ lastName  : chr "Smith"
## $ age       : int 25
## $ address   :List of 4
## ..$ streetAddress: chr "21 2nd Street"
## ..$ city       : chr "New York"
## ..$ state      : chr "NY"
## ..$ postalCode  : chr "10021"
## $ phoneNumber:'data.frame': 2 obs. of 2 variables:
## ..$ type : chr [1:2] "home" "fax"
## ..$ number: chr [1:2] "212 555-1234" "646 555-4567"
## $ gender   :List of 1
## ..$ type: chr "male"

## $streetAddress
## [1] "21 2nd Street"
##
## $city
## [1] "New York"
##
## $state
## [1] "NY"
##
## $postalCode
## [1] "10021"
##
## [1] "male"
```

6.5.3 Tutorial (advanced): Importing data from a HTML table (on a website)

In the chapter on high-dimensional data, we discussed the *Hypertext Markup Language (HTML)* as code to define the structure/content of a website and HTML-documents as semi-structured data sources. The following tutorial revisits the basic steps in importing data from an HTML table into R.

The aim of the tutorial is to generate a CSV file containing data on

‘divided government’⁸ in US politics. We use the following Wikipedia page as a data source: https://en.wikipedia.org/wiki/Divided_government_in_the_United_States. The page contains a table indicating the president’s party, and the majority party in the US House and the US Senate per Congress (2-year periods). The first few rows of the cleaned data are supposed to look like this:

##	year	president	senate	house
## 1:	1861	Lincoln	R	R
## 2:	1862	Lincoln	R	R
## 3:	1863	Lincoln	R	R
## 4:	1864	Lincoln	R	R
## 5:	1865	A. Johnson	R	R
## 6:	1866	A. Johnson	R	R

In a first step, we initiate fix variables for paths and load additional R packages needed to handle data stored in HTML documents.

```
# SETUP -----

# load packages
library(rvest)
library(data.table)

# fix vars
SOURCE_PATH <- "https://en.wikipedia.org/wiki/Divided_government_in_the_United_States"
OUTPUT_PATH <- "data/divided_gov.csv"
```

Now we write the part of the script that fetches the data from the Web. This part consists of three steps. First we fetch the entire website (HTML document) from Wikipedia with (`read_html()`). Second, we extract the part of the website containing the table with the data we want (via `html_node()`). Finally, we parse the HTML table and store its content in a data frame called `tab`. The last line of the code chunk below removes the last row of the data frame (you can see on the website that this row is not needed)

⁸https://en.wikipedia.org/wiki/Divided_government

```
# FETCH/FORMAT DATA -----

# fetch from web
doc <- read_html(SOURCE_PATH)
tab <- html_table(doc, fill=TRUE)[[2]]
tab <- tab[-nrow(tab), ] # remove last row (not containing data)
```

Now we clean the data to get a data set more suitable for data analysis. Note that the original table contains information per congress (2-year periods). However, as the sample above shows, we aim for a panel at the year level. The following code iterates through the rows of the data frame and generates for each row per congress several two rows (one for each year in the congress).⁹

```
# generate year-level data. frame

# prepare loop
all_years <- list() # the container
n <- length(tab$Year) # number of cases to iterate through
length(all_years) <- n
# generate year-level observations. row by row.
for (i in 1:n){
  # select row
  row <- tab[i,]
  y <- row$Year
  #
  begin <- as.numeric(unlist(strsplit(x = y, split = "[\\-\\-]", perl = TRUE))[1])
  end <- as.numeric(unlist(strsplit(x = y, split = "[\\-\\-]"))[2])
  tabnew <- data.frame(year=begin:(end-1), president=row$President, senate=row$Senate, hou
  all_years[[i]] <- tabnew # store in container
}
```

⁹See `?strsplit`, `?unlist`, and this introduction to regular expressions¹⁰ for the background of how this is done in the code example here.

```
# stack all rows together
allyears <- bind_rows(all_years)
```

In a last step, we inspect the collected data and write it to a CSV file.

```
# WRITE TO DISK -----

# inspect
head(allyears)
```

```
##   year  president senate house
## 1 1861    Lincoln      R      R
## 2 1862    Lincoln      R      R
## 3 1863    Lincoln      R      R
## 4 1864    Lincoln      R      R
## 5 1865 A. Johnson      R      R
## 6 1866 A. Johnson      R      R
```

```
# write to CSV
write_csv(allyears, file=OUTPUT_PATH)
```



7

Data Preparation

Importing a dataset properly is just the first of several milestones until an analysis-ready dataset is generated. In some cases, cleaning the raw data is a necessary step to facilitate/enable proper parsing of the data set to import it. However, most of the cleaning/preparation (‘wrangling’) with the data follows after properly parsing structured data. Many aspects of data wrangling are specific to certain datasets and an entire curriculum could be filled with different approaches and tools to address specific problems. Moreover, proficiency in data wrangling is generally a matter of experience in working with data, gained over many years. Here, we focus on two quite general and broadly applicable techniques that are central to cleaning and preparing a dataset for analysis: Simple string operations (find/replace parts of text strings) and reshaping rectangular data (wide to long/long to wide). The former is focused on individual variables at a time, while the latter typically happens at the level of the entire dataset.

7.1 Cleaning data with basic string operations

Recall that most of the data we read into R for analytic purposes is a collection of raw text (structured with special characters). When parsing the data to read it into R with high-level functions such as the ones provided in the `readr`-package, both the structure and the data types are considered. The resulting `data.frame/tibble` might thus contain variables (different columns) of type `character`, `factor`, or `integer`, etc. At this stage, it often happens that the raw data is not clean enough for the parser to recognize the data types in each column correctly, and it resorts to just parsing it as `character`. Indeed, if we have to deal with

a very messy dataset, it can make a lot of sense to constrain the parser such that it reads each column as `character`.

We first load this package as we rely on functions provided in the `tidyverse`.

```
library(tidyverse)
```

Let's create a sample dataset to illustrate some of the typical issues regarding unclean data that we might encounter in empirical economic research (and many similar domains of data analysis).¹

```
messy_df <- data.frame(last_name = c("Wayne", "Trump", "Karl Marx"),
                       first_name = c("John", "Melania", ""),
                       gender = c("male", "female", "Man"),
                       date = c("2018-11-15", "2018.11.01", "2018/11/02"),
                       income = c("150,000", "250000", "10000"),
                       stringsAsFactors = FALSE)
```

Assuming we have managed to read this dataset from a local file (with all columns as type `character`), the next step is to clean each of the columns such that the dataset is ready for analysis. Thereby we want to make sure that each variable (column) is set to a meaningful data type, once it is cleaned. The *cleaning* of the parsed data is often easier to do when the data is of type `character`. Once it is cleaned, however, we can set it to a type that is more useful for the analysis part. For example, a column containing numeric values in the final dataset should be stored as `numeric` or `integer`, so we can perform math operations on it later on (compute sums, means, etc.).

7.1.1 Find/replace character strings, recode factor levels

Our dataset contains a typical categorical variable: `gender`. In R, storing such variables as type `factor` is good practice. Without really looking at the data values, we might thus be inclined to do the following:

¹The option `stringsAsFactors = FALSE` ensures that all of the columns in this data frame are of type `character`.

```
messy_df$gender <- as.factor(messy_df$gender)
messy_df$gender
```

```
## [1] male    female Man
## Levels: female male Man
```

The column is now of type `factor`. And we see that R defined the factor variable such that an observation can be one of three categories ('levels'): `female`, `male`, or `Man`. In terms of content, that probably does not make too much sense. If we were to analyze the data later and compute the sample's share of males, we would only count one instead of two. Hence, we better *recode* the gender variable of male subjects as `male` and not `Man`. How can this be done programmatically?

One approach is to select all entries in `messy_df$gender` that are equal to `"Man"` and replace these entries with `"male"`.

```
messy_df$gender[messy_df$gender == "Man"] <- "male"
messy_df$gender
```

```
## [1] male    female male
## Levels: female male Man
```

Note, however, that this approach is not perfect because R still considers `Man` as a valid possible category in this column. This can have consequences for certain analyses we might want to run on this dataset later on.² Alternatively, we can use a function `fct_recode()` (provided in `tidyverse`), specifically for such operations with factors.

```
messy_df$gender <- fct_recode(messy_df$gender, "male" = "Man")
messy_df$gender
```

```
## [1] male    female male
## Levels: female male
```

²If we perform the same operation on this variable *before* coercing it to a factor, this problem does not occur.

The latter can be very useful when several factor levels need to be re-coded at once. Note that in both cases, the underlying logic is that we search for strings that are identical to "Man" and replace those values with "male". Now, the gender variable is ready for analysis.

7.1.2 Removing individual characters from a string

The `income` column contains numbers, so let's try to set this column to type `integer`.

```
as.integer(messy_df$income)
```

```
## Warning: NAs introduced by coercion
```

```
## [1]      NA 250000 10000
```

R is warning us that something did not go well when executing this code. We see that the first value of the original column has been replaced with `NA` ('Not Available'/'Not Applicable'/'No Answer'). The reason is that the original value contained a comma (,), a special character. The function `as.integer()` does not know how to translate such a symbol to a number. Hence, the original data, value cannot be translated into a number (integer). In order to resolve this issue, we have to remove the comma (,) from this string. Or, more precisely, we will locate this specific character *within* the string and replace it with an empty string (""). To do so, we'll use the function `str_replace()` (for 'string replace').

```
messy_df$income <- str_replace(messy_df$income, pattern = ",", replacement = "")
```

Now we can successfully set the column as type `integer`.

```
messy_df$income <- as.integer(messy_df$income)
```

7.1.3 Splitting strings

From looking at the `last_name` and `first_name` columns of our messy dataset, it becomes clear that the last row is not accurately coded. `Karl`

should show up in the `first_name` column. In order to correct this, we have to extract a part of one string and store this sub-string in another variable. There are several ways to do this. Here, it probably makes sense to split the original string into two parts, as the white space between `Karl` and `Marx` indicates the separation of first and last names. For this, we can use the function `str_split()`.

First, we split the strings at every occurrence of white space (" "). Setting the option `simplify=TRUE`, we get a matrix containing the individual sub-strings after the splitting.

```
splitnames <- str_split(messy_df$last_name, pattern = " ", simplify = TRUE)
splitnames

##      [,1]      [,2]
## [1,] "Wayne" ""
## [2,] "Trump" ""
## [3,] "Karl"  "Marx"
```

As the first two observations did not contain any white space, there was nothing to split there, and the function simply returned empty strings "". In a second step, we replace empty observations in the `first_name` column with the corresponding values in `splitnames`.

```
problem_cases <- messy_df$first_name == ""
messy_df$first_name[problem_cases] <- splitnames[problem_cases, 1]
```

Finally, we must correct the `last_name` column by replacing the respective values.

```
messy_df$last_name[problem_cases] <- splitnames[problem_cases, 2]
messy_df
```

```
##   last_name first_name gender      date income
## 1   Wayne      John   male 2018-11-15 150000
## 2   Trump    Melania female 2018.11.01 250000
## 3   Marx      Karl    male 2018/11/02  10000
```

7.1.4 Parsing dates

Finally, we take a look at the `date`-column of our dataset. For many data preparation steps as well as visualization and analysis, it is advantageous to have times and dates properly parsed as type `Date`. In practice, dates and times are often particularly messy because no unique standard has been used to define the format in the data collection phase. This also seems to be the case in our dataset. In order to work with dates, we load the `lubridate` package.

```
library(lubridate)
```

This package provides several functions to parse and manipulate date and time data. From the 'date' column, we see that the format is year, month, and day. Thus, we can use the `ymd()`-function provided in the `lubridate`-package to parse the column as `Date` type.

```
messy_df$date <- ymd(messy_df$date)
```

Note how this function automatically recognizes how different special characters have been used in different observations to separate years from months/days.

Now, our dataset is cleaned up and ready to go.

```
messy_df
```

```
##   last_name first_name gender      date income
## 1   Wayne      John   male 2018-11-15 150000
## 2   Trump     Melania female 2018-11-01 250000
## 3   Marx      Karl    male 2018-11-02  10000
```

```
str(messy_df)
```

```
## 'data.frame':   3 obs. of  5 variables:
##  $ last_name : chr  "Wayne" "Trump" "Marx"
##  $ first_name: chr  "John" "Melania" "Karl"
```

```
## $ gender      : Factor w/ 2 levels "female","male": 2 1 2
## $ date        : Date, format: "2018-11-15" ...
## $ income      : int  150000 250000 10000
```

7.2 Reshaping datasets

Besides cleaning and standardizing individual data columns, preparing a dataset for analysis often involves bringing the entire dataset in the right ‘shape.’ Typically, we mean this in a table-like (two-dimensional) format such as `data.frames` and `tibbles`, data with repeated observations for the same unit can be displayed/stored in either *long* or *wide* format. It is often seen as good practice to prepare data for analysis in *long* (‘tidy’) format. This way we ensure that we follow the (‘tidy’) paradigm of using the rows for individual observations and the columns to describe these observations.³ Tidying/reshaping a dataset in this way thus involves transforming columns into rows (i.e., *melting* the dataset). In the following, we first have a close look at what this means conceptually and then apply this technique in two examples.

7.2.1 Tidying messy datasets.

Consider the following stylized example ([Wickham, 2014](#)).

person	treatmenta	treatmentb
John Smith	NA	2
Jane Doe	16	11
Mary Johnson	3	1

The table shows observations of three individuals participating in an experiment. In this experiment, the subjects might have been exposed to treatment a and/or treatment b. Their reaction to either treatment is measured in numeric values (the results of the experiment). From

³Depending on the dataset, however, an argument can be made that storing the data in wide format might be more efficient (using up less memory) than long format.


```

income.2018 = c("150000", "250000", "10000"),
income.2017 = c("140000", "230000", "15000"),
stringsAsFactors = FALSE)

wide_df

```

```

##   last_name first_name gender income.2018 income.2017
## 1   Wayne      John   male      150000      140000
## 2   Trump      Melania female      250000      230000
## 3   Marx       Karl    male       10000       15000

```

The last two columns contain information on the same variable (income), but for different years. We thus want to pivot these two columns into a new `year` and `income` column, ensuring that columns correspond to variables and rows correspond to observations. For this, we call the `pivot_longer()`-function as follows:

```

long_df <- pivot_longer(wide_df, c(income.2018, income.2017), names_to = "year", values_to = "income")
long_df

```

```

## # A tibble: 6 x 5
##   last_name first_name gender year      income
##   <chr>      <chr>      <chr> <chr>      <chr>
## 1 Wayne      John      male  income.2018 150000
## 2 Wayne      John      male  income.2017 140000
## 3 Trump      Melania   female income.2018 250000
## 4 Trump      Melania   female income.2017 230000
## 5 Marx       Karl      male   income.2018 10000
## 6 Marx       Karl      male   income.2017 15000

```

We can further clean the `year` column to only contain the respective numeric values.

```

long_df$year <- str_replace(long_df$year, "income.", "")
long_df

```

```

## # A tibble: 6 x 5
##   last_name first_name gender year  income

```

```
##   <chr>      <chr>      <chr> <chr> <chr>
## 1 Wayne     John       male  2018  150000
## 2 Wayne     John       male  2017  140000
## 3 Trump     Melania    female 2018  250000
## 4 Trump     Melania    female 2017  230000
## 5 Marx      Karl        male   2018  10000
## 6 Marx      Karl        male   2017  15000
```

7.2.3 Pivoting from ‘long to wide’ (“spreading”)

As we want to adhere to the ‘tidy’ paradigm of keeping our data in long format, transforming ‘long to wide’ is less common. However, it might be necessary if the dataset at hand is particularly messy. The following example illustrates such a situation.

```
weird_df <- data.frame(last_name = c("Wayne", "Trump", "Marx",
                                     "Wayne", "Trump", "Marx",
                                     "Wayne", "Trump", "Marx"),
                      first_name = c("John", "Melania", "Karl",
                                     "John", "Melania", "Karl",
                                     "John", "Melania", "Karl"),
                      gender = c("male", "female", "male",
                                 "male", "female", "male",
                                 "male", "female", "male"),
                      value = c("150000", "250000", "10000",
                                "2000000", "5000000", "NA",
                                "50", "25", "NA"),
                      variable = c("income", "income", "income",
                                   "assets", "assets", "assets",
                                   "age", "age", "age"),
                      stringsAsFactors = FALSE)

weird_df
```

```
##   last_name first_name gender  value variable
## 1   Wayne      John    male 150000  income
## 2   Trump     Melania female 250000  income
## 3   Marx      Karl     male  10000  income
```

```
## 4 Wayne John male 2000000 assets
## 5 Trump Melania female 5000000 assets
## 6 Marx Karl male NA assets
## 7 Wayne John male 50 age
## 8 Trump Melania female 25 age
## 9 Marx Karl male NA age
```

While the data is somehow in a long format, the rule that each column should correspond to a variable (and vice versa) is ignored. Data on income, assets, and the age of the individuals in the dataset, are all put in the same column. We can call the function `pivot_wider()` with the two parameters `names` and `value` to correct this.

```
tidy_df <- pivot_wider(weird_df, names_from = "variable", values_from = "value")
tidy_df
```

```
## # A tibble: 3 x 6
##   last_name first_name gender income assets age
##   <chr>      <chr>      <chr> <chr> <chr> <chr>
## 1 Wayne    John      male  150000 2000000 50
## 2 Trump    Melania   female 250000 5000000 25
## 3 Marx     Karl      male   10000 NA     NA
```

7.3 Tutorial: Hotel Bookings Time Series

This tutorial guides you step-by-step through the cleaning script (with a few adaptations) of tidyuesday's Hotel Bookings repo⁴, dealing with the preparation and analysis of two datasets with *hotel demand data*. Along the way, you also get in touch with the janitor package⁵. For details about the two datasets, see the paper⁶ by Antonio et al. (2019), and

⁴<https://github.com/rfordatascience/tidyuesday/tree/master/data/2020/2020-02-11>

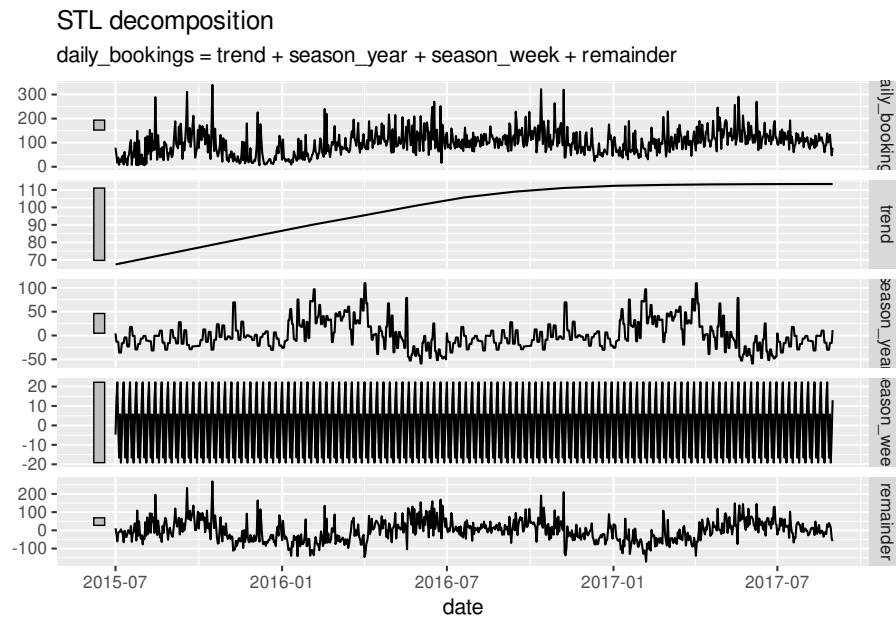
⁵<https://github.com/sfirke/janitor>

⁶<https://www.sciencedirect.com/science/article/pii/S2352340918315191#f0010>

for the original research contribution related to these datasets see the paper⁷ by Antonio et al. (2017).

Antonio et al. (2019) summarizes the content of the datasets as follows: “One of the hotels (H1) is a resort hotel, and the other is a city hotel (H2). Both datasets share the same structure, with 31 variables describing the 40,060 observations of H1 and 79,330 observations of H2. Each observation represents a hotel booking. Both datasets comprehend bookings due to arrive between the 1st of July 2015 and the 31st of August 2017, including bookings that effectively arrived and bookings that were canceled. Since this is real data, all data elements pertaining to hotel or customer identification were deleted. Due to the scarcity of real business data for scientific and educational purposes, these datasets can have an important role for research and education in revenue management, machine learning, or data mining, as well as in other fields.”

The aim of the tutorial is to get the data in the form needed for the following plot.



⁷<https://ieeexplore.ieee.org/document/8260781>

The first few rows and columns of the final dataset should combine the two source datasets and look as follows:

```
head(hotel_df)
```

```
## # A tibble: 6 x 32
##   hotel is_ca~1 lead_~2 arriv~3 arriv~4 arriv~5 arriv~6
##   <chr>   <dbl>   <dbl>   <dbl> <chr>       <dbl>   <dbl>
## 1 Reso~     0     342     2015 July         27     1
## 2 Reso~     0     737     2015 July         27     1
## 3 Reso~     0       7     2015 July         27     1
## 4 Reso~     0      13     2015 July         27     1
## 5 Reso~     0      14     2015 July         27     1
## 6 Reso~     0      14     2015 July         27     1
## # ... with 25 more variables:
## #   stays_in_weekend_nights <dbl>,
## #   stays_in_week_nights <dbl>, adults <dbl>,
## #   children <dbl>, babies <dbl>, meal <chr>,
## #   country <chr>, market_segment <chr>,
## #   distribution_channel <chr>,
## #   is_repeated_guest <dbl>, ...
## # i Use `colnames()` to see all variable names
```

7.4 Set up and import

All the tools we need for this tutorial are provided in `tidyverse` and `janitor`, and the data is directly available from the `tidytuesday` GitHub repository⁸. The original data is provided in CSV format.

```
# SET UP -----
```

```
# load packages
```

⁸<https://github.com/rfordatascience/tidytuesday/tree/master/data/2020/2020-02-11>

```

library(tidyverse)
library(janitor) # install.packages("janitor") (if not yet installed)

# fix variables
url_h1 <- "https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2020/2020-01-01/2020-01-01-hotel-resort.csv"
url_h2 <- "https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2020/2020-01-01/2020-01-01-hotel-city.csv"

## DATA IMPORT -----

h1 <- read_csv(url_h1)
h2 <- read_csv(url_h2)

```

In the next step, we clean the column names and add columns to clarify which of the two hotels the corresponding observations belong to (see dataset description above). Finally, we stack the observations (rows) together in one tibble/data.frame.

```

## CLEAN DATA -----

# use the janitor-package clean_names function. see ?clean_names for details
h1 <- clean_names(h1)
h2 <- clean_names(h2)

# add a column to clarify the origin of observation
h1 <- mutate(h1, hotel="Resort Hotel")
h2 <- mutate(h2, hotel="City Hotel")

# stack observations
hotel_df <- bind_rows(h1,h2)

# inspect the first observations
head(hotel_df)

```

```

## # A tibble: 6 x 32
##   is_canceled lead_time arriv~1 arriv~2 arriv~3 arriv~4
##         <dbl>     <dbl>   <dbl> <chr>     <dbl>   <dbl>

```

```
## 1      0      342    2015 July      27      1
## 2      0      737    2015 July      27      1
## 3      0        7    2015 July      27      1
## 4      0       13    2015 July      27      1
## 5      0       14    2015 July      27      1
## 6      0       14    2015 July      27      1
## # ... with 26 more variables:
## #   stays_in_weekend_nights <dbl>,
## #   stays_in_week_nights <dbl>, adults <dbl>,
## #   children <dbl>, babies <dbl>, meal <chr>,
## #   country <chr>, market_segment <chr>,
## #   distribution_channel <chr>,
## #   is_repeated_guest <dbl>, ...
## # i Use `colnames()` to see all variable names
```



8

Data Analysis: First Steps

In the first part of this chapter, we look at some key functions for applied data analysis in R. At this point, we have already implemented collecting/importing and cleaning the raw data. The analysis part can be thought of as a collection of tasks with the aim of making sense of the data. In practice, this can be explorative (discovering interesting patterns in the data) or inductive (testing of a specific hypothesis). Moreover, it typically involves functions for actual statistical analysis and various functions to select, combine, filter, and aggregate data. Similar to the topic of data cleaning/preparation, covering all aspects of applied data analysis with R goes well beyond the scope of one chapter. The aim is thus to give a practical overview of some of the basic concepts and their corresponding R functions (here from `tidyverse`).

8.1 Merging datasets

The following two data sets contain data on persons' characteristics and their consumption spending. Both are cleaned datasets. But, for analysis purposes, we have to combine the two datasets. There are several ways to do this in R, but most commonly (for `data.frames` as well as `tibbles`), we can use the `merge()`-function.

```
# load packages
library(tidyverse)

# initiate data frame on persons' spending
df_c <- data.frame(id = c(1:3, 1:3),
                   money_spent = c(1000, 2000, 6000, 1500, 3000, 5500),
```

```

currency = c("CHF", "CHF", "USD", "EUR", "CHF", "USD"),
year=c(2017,2017,2017,2018,2018,2018))

df_c

```

```

##   id money_spent currency year
## 1  1         1000      CHF 2017
## 2  2         2000      CHF 2017
## 3  3         6000      USD 2017
## 4  1         1500      EUR 2018
## 5  2         3000      CHF 2018
## 6  3         5500      USD 2018

```

```

# initiate data frame on persons' characteristics
df_p <- data.frame(id = 1:4,
                    first_name = c("Anna", "Betty", "Claire", "Diane"),
                    profession = c("Economist", "Data Scientist",
                                   "Data Scientist", "Economist"))

df_p

```

```

##   id first_name    profession
## 1  1      Anna    Economist
## 2  2    Betty Data Scientist
## 3  3   Claire Data Scientist
## 4  4    Diane    Economist

```

Our aim is to compute the average spending by profession. Therefore, we want to link `money_spent` with `profession`. Both datasets contain a unique identifier `id`, which we can use to link the observations via `merge()`.

```

df_merged <- merge(df_p, df_c, by="id")
df_merged

```

```

##   id first_name    profession money_spent currency
## 1  1      Anna    Economist         1000      CHF
## 2  1      Anna    Economist         1500      EUR

```

```
## 3 2      Betty Data Scientist      2000      CHF
## 4 2      Betty Data Scientist      3000      CHF
## 5 3      Claire Data Scientist      6000      USD
## 6 3      Claire Data Scientist      5500      USD
##   year
## 1 2017
## 2 2018
## 3 2017
## 4 2018
## 5 2017
## 6 2018
```

Note how only the exact matches are merged. The observation of "Diane" is not part of the merged data frame because there is no corresponding row in `df_c` with her spending information. If for some reason, we would like to have all persons in the merged dataset, we can specify the `merge()`-call accordingly:

```
df_merged2 <- merge(df_p, df_c, by="id", all = TRUE)
df_merged2
```

```
##   id first_name      profession money_spent currency
## 1  1      Anna      Economist      1000      CHF
## 2  1      Anna      Economist      1500      EUR
## 3  2      Betty Data Scientist      2000      CHF
## 4  2      Betty Data Scientist      3000      CHF
## 5  3      Claire Data Scientist      6000      USD
## 6  3      Claire Data Scientist      5500      USD
## 7  4      Diane      Economist         NA      <NA>
##   year
## 1 2017
## 2 2018
## 3 2017
## 4 2018
## 5 2017
## 6 2018
## 7   NA
```

8.2 Selecting subsets

For our analysis's next steps, we do not need to have all columns. Via the `select()`-function provided in `tidyverse` we can easily select the columns of interest

```
df_selection <- select(df_merged, id, year, money_spent, currency)
df_selection
```

```
##   id year money_spent currency
## 1  1 2017      1000      CHF
## 2  1 2018      1500      EUR
## 3  2 2017      2000      CHF
## 4  2 2018      3000      CHF
## 5  3 2017      6000      USD
## 6  3 2018      5500      USD
```

8.2.1 Filtering datasets

In the next step, we want to select only observations with specific characteristics. Say we want to select only observations from 2018. Again there are several ways to do this in R, but the most comfortable way is to use the `filter()` function provided in `tidyverse`.

```
filter(df_selection, year == 2018)
```

```
##   id year money_spent currency
## 1  1 2018      1500      EUR
## 2  2 2018      3000      CHF
## 3  3 2018      5500      USD
```

We can use several filtering conditions simultaneously:

```
filter(df_selection, year == 2018, money_spent < 5000, currency=="EUR")
```

```
##   id year money_spent currency
```



```
## 1 1 2018      1500      EUR
```

8.2.2 Mutating datasets

Before we compute aggregate statistics based on our selected dataset, we have to deal with the fact that the `money_spent`-variable is not tidy. It describes each observation's characteristic, but it is measured in different units (here, different currencies) across some of these observations. If the aim was to have a perfectly tidy dataset, we could address the issue with `spread()`. However, in this context, it could be more helpful to add an additional variable/column with a normalized amount of money spent. That is, we want to have every value converted to one currency (given a certain exchange rate). In order to do so, we use the `mutate()` function (again provided in `tidyverse`).

First, we look up the USD/CHF and EUR/CHF exchange rates and add those as a variable (CHF/CHF exchange rates are equal to 1, of course).

```
exchange_rates <- data.frame(exchange_rate= c(0.9, 1, 1.2),
                             currency=c("USD", "CHF", "EUR"), stringsAsFactors = FALSE)
df_selection <- merge(df_selection, exchange_rates, by="currency")
```

Now we can define an additional variable with the money spent in CHF via `mutate()`:

```
df_mutated <- mutate(df_selection, money_spent_chf = money_spent * exchange_rate)
df_mutated
```

```
##   currency id year money_spent exchange_rate
## 1      CHF 1 2017      1000         1.0
## 2      CHF 2 2017      2000         1.0
## 3      CHF 2 2018      3000         1.0
## 4      EUR 1 2018      1500         1.2
## 5      USD 3 2017      6000         0.9
## 6      USD 3 2018      5500         0.9
##   money_spent_chf
## 1             1000
## 2             2000
```

```
## 3      3000
## 4      1800
## 5      5400
## 6      4950
```

8.2.3 Aggregation and summary statistics

Now we can start analyzing the dataset. Typically, the first step of analyzing a dataset is to get an overview by computing some summary statistics. This helps to better understand the dataset at hand. Key summary statistics of the variables of interest are the mean, standard deviation, median, and a number of observations. Together, they give a first idea of how the variables of interest are distributed.

As you know from previous chapters, R has several built-in functions that help us do this. In practice, these basic functions are often combined with functions implemented particularly for this step of the analysis, such as `summarise()` provided in `tidyverse`.

As the first output in our report, we want to show the key characteristics of the spending data in one table.

```
summarise(df_mutated,
  mean = mean(money_spent_chf),
  standard_deviation = sd(money_spent_chf),
  median = median(money_spent_chf),
  N = n())
```

```
##   mean standard_deviation median N
## 1 3025                1789   2500 6
```

Moreover, we can compute the same statistics grouped by certain observation characteristics. For example, we can compute the same summary statistics per year of observation.

```
by_year <- group_by(df_mutated, year)
summarise(by_year,
  mean = mean(money_spent_chf),
```

```

standard_deviation = sd(money_spent_chf),
median = median(money_spent_chf),
N = n())

```

```

## # A tibble: 2 x 5
##   year mean standard_deviation median      N
##   <dbl> <dbl>                <dbl> <dbl> <int>
## 1  2017  2800                2307.  2000     3
## 2  2018  3250                1590.  3000     3

```

Alternatively, to the more user-friendly (but less flexible) `summarise` function, we can use lower-level functions to compute aggregate statistics provided in the basic R distribution. A good example of such a function is `sapply()`. In simple terms, `sapply()` takes a list as input and applies a function to the content of each element in this list (here: compute a statistic for each column). To illustrate this point, we load the already familiar `swiss` dataset.

```

# load data
data("swiss")

```

Now we want to compute the mean for each variable in this dataset. Technically speaking, a data frame is a list, where each list element is a column of the same length. Thus, we can use `sapply()` to ‘apply’ the function `mean()` to each of the columns in `swiss`.

```
sapply(swiss, mean)
```

```

##      Fertility      Agriculture      Examination
##      70.14         50.66         16.49
##      Education      Catholic Infant.Mortality
##      10.98         41.14         19.94

```

By default, `sapply()` returns a vector or a matrix.¹ We can get a simi-

¹The related function `lapply()`, returns a list (see `lapply(swiss, mean)`).

lar result by using `summarise()`. However, we would have to explicitly mention which variables we want as input.

```
summarise(swiss,
  Fertility = mean(Fertility),
  Agriculture = mean(Agriculture)) # etc.

##   Fertility Agriculture
## 1      70.14      50.66
```

8.3 Tutorial: Analyse messy Excel sheets

The following tutorial is a (substantially) shortened and simplified version of Ista Zahn and Daina Bouquin's "Cleaning up messy data tutorial" (Harvard Datafest 2017)². The tutorial aims to clean up an Excel sheet provided by the UK Office of National Statistics that provides data on the most popular baby names in England and Wales in 2015. The dataset is stored in `data/2015boysnamesfinal.xlsx`

8.3.1 Preparatory steps

```
## SET UP -----
# load packages
library(tidyverse)
library(readxl)

# fix variables
INPUT_PATH <- "data/2015boysnamesfinal.xlsx"
```

Before diving into the data import and cleaning, it is helpful to first open the file in Excel. We notice a couple of things there: first, there are several sheets in this Excel file. For this exercise, we only rely on the sheet called "Table 1". Second, in this sheet, we notice intuitively

²<https://rawgit.com/izahn/R-data-cleaning/master/dataCleaning.html>

some potential problems with importing this dataset due to the way the spreadsheet is organized. The actual data entries only start on row 7. These two issues can be considered when importing the data with `read_excel()`.

```
## LOAD/INSPECT DATA -----
```

```
# import the excel sheet
```

```
boys <- read_excel(INPUT_PATH, col_names = TRUE,
                    sheet = "Table 1", # the name of the sheet to be loaded into R
                    skip = 6 # skip the first 6 rows of the original sheet,
                    )
```

```
## New names:
```

```
## * `Rank` -> `Rank...1`
## * `Name` -> `Name...2`
## * `Count` -> `Count...3`
## * `since 2014` -> `since 2014...4`
## * `since 2005` -> `since 2005...5`
## * `` -> `...6`
## * `Rank` -> `Rank...7`
## * `Name` -> `Name...8`
## * `Count` -> `Count...9`
## * `since 2014` -> `since 2014...10`
## * `since 2005` -> `since 2005...11`
```

```
# inspect
```

```
boys
```

```
## # A tibble: 61 x 11
```

```
##   Rank...1 Name...2 Count...3 since 20~1 since~2 ...6
##   <chr>      <chr>      <dbl> <chr>      <chr>      <lgl>
## 1 <NA>      <NA>          NA <NA>      <NA>      NA
## 2 1         OLIVER        6941      +4        NA
## 3 2         JACK         5371      -1        NA
## 4 3         HARRY         5308      +6        NA
## 5 4         GEORGE        4869 +3      +13        NA
```

```
## 6 5      JACOB      4850 -1      +16      NA
## 7 6      CHARLIE    4831 -1      +6        NA
## 8 7      NOAH      4148 +4      +44        NA
## 9 8      WILLIAM    4083 +2              NA
## 10 9     THOMAS     4075 -3      -6        NA
## # ... with 51 more rows, 5 more variables:
## #   Rank...7 <chr>, Name...8 <chr>, Count...9 <dbl>,
## #   `since 2014...10` <chr>, `since 2005...11` <chr>,
## #   and abbreviated variable names
## #   1: `since 2014...4`, 2: `since 2005...5`
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

Note that by default, `read_excel()` “repairs” the column names of imported datasets to ensure all columns have unique names. We do not need to worry about the automatically assigned column names. However, some of the columns are not needed for analytics purposes. In addition, we note that some rows are empty (contain NA values). In the next step we *select* only those columns needed and *filter* incomplete observations out.

```
# FILTER/CLEAN -----

# select columns
boys <- select(boys, Rank...1, Name...2, Count...3, Rank...7, Name...8, Count...9)
# filter rows
boys <- filter(boys, !is.na(Rank...1))
```

Finally, we re-arrange the data by stacking them in a three-column format.

```
# stack columns
boys_long <- bind_rows(boys[,1:3], boys[,4:6])
```

```
## New names:
## New names:
## * `Rank...1` -> `Rank`
## * `Name...2` -> `Name`
```

```
## * `Count...3` -> `Count`
```

```
# inspect result
```

```
boys_long
```

```
## # A tibble: 114 x 3
```

```
##   Rank  Name    Count
```

```
##   <chr> <chr>   <dbl>
```

```
## 1 1    OLIVER  6941
```

```
## 2 2    JACK   5371
```

```
## 3 3    HARRY  5308
```

```
## 4 4    GEORGE 4869
```

```
## 5 5    JACOB  4850
```

```
## 6 6    CHARLIE 4831
```

```
## 7 7    NOAH   4148
```

```
## 8 8    WILLIAM 4083
```

```
## 9 9    THOMAS 4075
```

```
## 10 10  OSCAR   4066
```

```
## # ... with 104 more rows
```

```
## # i Use `print(n = ...)` to see more rows
```



9

Basic Econometrics in R

When we look at practical data analytics in an economics context, it becomes quickly apparent that the vast majority of applied econometric research builds in one way or the other on linear regression, specifically on *ordinary least squares* (OLS). OLS is the bread-and-butter model in microeconometrics for several reasons: it is rather easy to understand, its results are straightforward to interpret, it comes with several very useful statistical properties, and it can easily be applied to a large variety of economic research questions. In this section, we get to know the very basic idea behind OLS by looking at it through the lens of formal math notation as well as through the lens of R code.

9.1 Statistical modeling

Before we turn to a specific data example, let us introduce a general notation for this econometric problem. Generally we denote the *dependent variable* as y_i and the *explanatory variable* as x_i . All the rest that is not more specifically explained by explanatory variables (u_i) we call the '*residuals*' or the 'error term.' Hence we can rewrite the problem with a more general notation as

$$y_i = \alpha + \beta x_i + u_i.$$

The overarching goal of modern econometrics for such problems is to assess whether x has a *causal* effect on y . A key aspect of such an interpretation is that u_i is unrelated to x_i . Other unobservable factors might also play a role for y , but they must not affect both y and x . The great computer scientist Judea Pearl¹ has introduced an intuitive ap-

¹https://en.wikipedia.org/wiki/Judea_Pearl

proach to illustrating such causality problems. In the left panel of the figure below, we illustrate the unfortunate case where we have a so-called 'endogeneity' problem, a situation in which other factors affect both x and y .

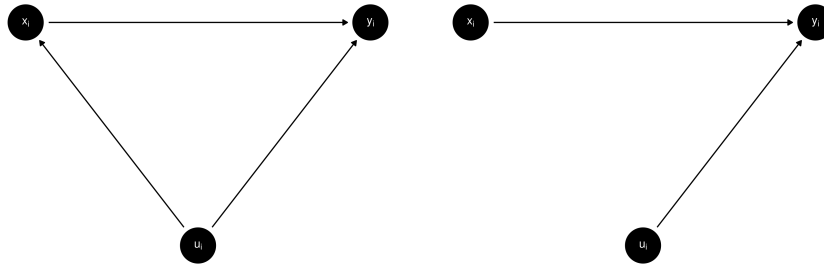


FIGURE 9.1: (ref:causality)

(ref: causality) Causal diagrams. On the left with an endogeneity issue (other factors affect both the dependent and the explanatory variable), and on the right without this endogeneity issue.

9.1.1 Illustration with pseudo-data

Let us now look at how we can estimate β under the assumption that u does not affect x . To make this more general and easily accessible, let us, for the moment, ignore the dataset above and generate simulated data for which this assumption certainly holds. The basic R installation provides a number of functions to easily generate vectors of pseudo-random numbers² The great advantage of using simulated data and code to understand statistical models is that we have full control over what the *actual* relationship between variables is. We can then easily assess how well an estimation procedure for a parameter is by comparing the parameter estimate with its true value (something we never observe when working with real-life data).

First, we define the key parameters for the simulation. We choose the actual values of α and β , and set the number of observations N .

²Computers cannot actually 'generate' true random numbers. However, there are functions that produce series of numbers which have the properties of randomly drawn numbers from a given distribution.

```
alpha <- 30  
beta <- 0.9  
N <- 1000
```

Now, we initiate a vector x of length N drawn from the uniform distribution (between 0 and 0.5). This will be our explanatory variable.

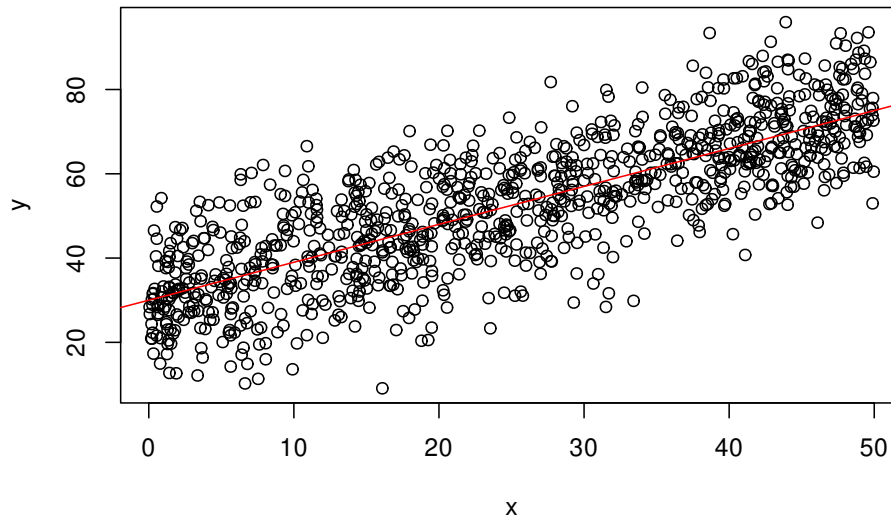
```
x <- runif(N, 0, 50)
```

Next, we draw a vector of random errors (residuals) u (from a normal distribution with mean=0 and SD=0.05) and compute the corresponding values of the dependent variable y . Since we impose that the actual relationship between the variables is exactly defined in our simple linear model, this computation is straightforward ($y_i = \alpha + \beta x_i + u_i$).

```
# draw the random errors (all the other factors also affecting y)  
epsilon <- rnorm(N, sd=10)  
# compute the dependent variable values  
y <- alpha + beta*x + epsilon
```

We can illustrate how y changes when x increases by plotting the raw data as a scatter plot as well as the true relationship (function) as a line.

```
plot(x,y)  
abline(a = alpha, b=beta, col="red")
```

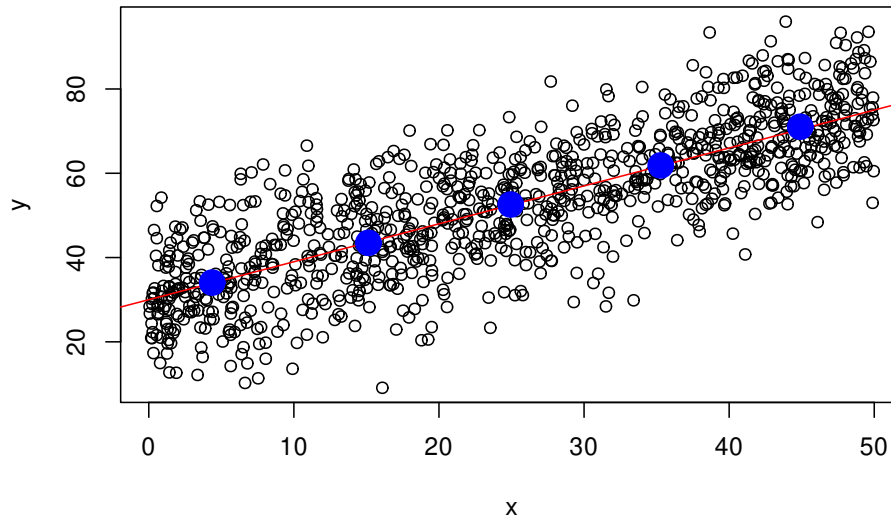


We see that while individual observations can be way off the true values (the red line), ‘on average’ the observations seem to follow the true relationship between x and y . We can look at this closely by computing the average observed y -values for different intervals along the x -axis.

```
# compute average y per x intervals
lower <- c(0,10,20,30,40)
upper <- c(lower[-1], 50)
n_intervals <- length(lower)
y_bars <- list()
length(y_bars) <- n_intervals
x_bars <- list()
length(x_bars) <- n_intervals
for (i in 1:n_intervals){
  y_bars[i] <- mean(y[lower[i] <= x & x < upper[i]])
  x_bars[i] <- mean(x[lower[i] <= x & x < upper[i]])
}
y_bars <- unlist(y_bars)
x_bars <- unlist(x_bars)

# add to plot
plot(x,y)
```

```
abline(a = alpha, b=beta, col="red")
points(x_bars, y_bars, col="blue", lwd=10)
```



The average values are much closer to the real values. That is, we can ‘average out’ the u to get a good estimate for the effect of x on y (to get an estimate of β). With this understanding, we can now formalize how to compute β (or, to be more precise, an estimate of it: $\hat{\beta}$). For simplicity, we take $\alpha = 30$ as given.

In a first step we take the averages of both sides of our initial regression equation:

$$\frac{1}{N} \sum y_i = \frac{1}{N} \sum (30 + \beta x_i + u_i),$$

rearranging and using the common ‘bar’-notation for means, we get

$$\bar{y} = 30 + \beta \bar{x} + \bar{u},$$

and solving for β and some rearranging then yields

$$\beta = \frac{\bar{y} - 30 - \bar{u}}{\bar{x}}.$$

While the elements in \bar{u} are unobservable, we can use the rest to compute an estimate of β :

$$\hat{\beta} = \frac{\bar{y} - 30}{\bar{x}}.$$

```
(mean(y) - 30) / mean(x)
```

```
## [1] 0.9076
```

9.2 Estimation and Application

Now that we have a basic understanding of the simple linear model, let us use this model to investigate an empirical research question based on real data. By doing so, we will take an alternative perspective to compute an estimation of the parameters of interest. The example builds on the ‘swiss’ dataset encountered in previous chapters. First, get again familiar with the dataset:

```
# load the data
data(swiss)
# look at the description
?swiss
```

Recall that observations in this dataset are at the province level. We use the simple linear model to better understand whether more years of schooling is improving educational outcomes. Thereby, we approximate educational attainment with the variable `Education` (the percentage of Swiss military draftees in a province that had education beyond primary school) and educational outcomes with the variable `Examination` (the percentage of draftees in a province that have received the highest mark on the standardized test as part of the army examination). We thus want to exploit that schooling systematically varies between provinces but that all military draftees need to take the same standardized examination during the military drafting process.

9.2.1 Model specification

Formally, we can express the relationship between these variables as

$$Examination_i = \alpha + \beta Education_i,$$

where the parameters α and β determine the percentage of draftees in a province that has received the highest mark, and the subscript i indicates that we model this relationship for each province i out of the N provinces (note that this also presupposes that α and β are the same in each province). The intuitive hypothesis is that in this equation, β is positive, indicating that a higher share of draftees with more years of schooling results in a higher share of draftees who reach the highest examination mark.

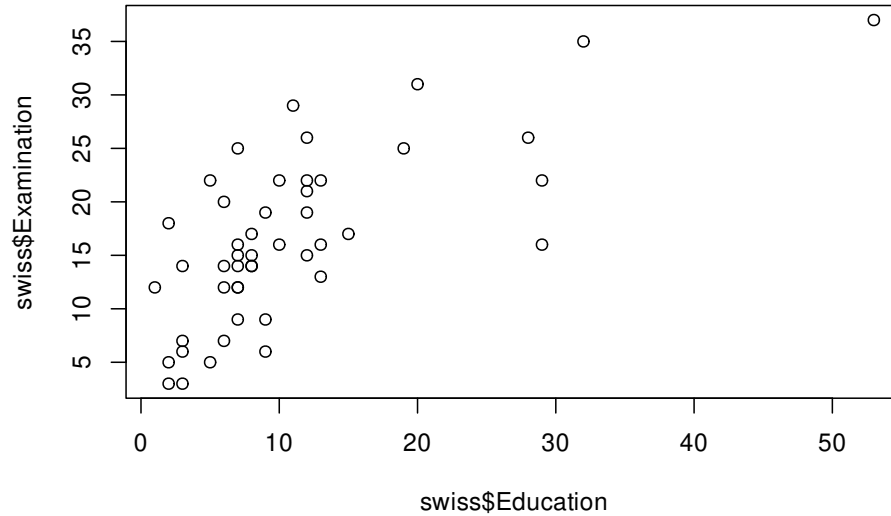
Yet, our model arguably has a couple of weaknesses. Most importantly, from an economic point of view, it seems rather unlikely that *Education* is the only variable that matters for examination success. For example, it might matter a lot how students allocate their time when not in school, which might vary by province. In some provinces, children and young adults might have engaged in work activities that foster their reading and math skills outside school. In contrast, in other provinces, they might have engaged in hard manual labor in the agricultural sector. To formally acknowledge that other factors might also play a role, we extend our model with the term u_i . For the moment, we thus subsume all other potentially relevant factors in that term:

$$Examination_i = \alpha + \beta Education_i + u_i.$$

9.2.2 Raw data

First, look at how these two variables are jointly distributed with a scatter plot.

```
plot(swiss$Education, swiss$Examination)
```



There seems to be a positive relationship between the years of schooling and examination success. Considering the raw data, we can formulate the problem of estimating the parameters of interest (α and β) as the question: What is the best way to draw a straight trend line through the cloud of dots? To further specify the problem, we need to specify a criterion to judge whether such a line ‘fits’ the data well. A rather intuitive measure is the sum of the squared differences between the line and the actual data points on the y -axis. That is, we compare what y -value we would get for a given x -value according to our trend-line (i.e., the function defining this line) with the actual y -value in the data (for simplicity, we use the general formulation with y as the dependent and x as the explanatory variable).

9.3 Derivation and implementation of OLS estimator

From the model equation, we easily see that these ‘differences’ between the predicted and the actual values of y are the remaining unexplained component u :

$$y_i - \hat{\alpha} - \hat{\beta}x_i = u_i.$$

Hence, we want to minimize the *sum of squared residuals (SSR)*: $\sum u_i^2 =$

$\sum (y_i - \hat{\alpha} - \hat{\beta}x_i)^2$. Using calculus, we define the two first-order conditions:

$$\frac{\partial SSR}{\partial \hat{\alpha}} = \sum -2(y_i - \hat{\alpha} - \hat{\beta}x_i) = 0$$

$$\frac{\partial SSR}{\partial \hat{\beta}} = \sum -2x_i(y_i - \hat{\alpha} - \hat{\beta}x_i) = 0$$

The first condition is relatively easily solved by getting rid of the -2 and considering that $\sum y_i = N\bar{y}$:

$$\hat{\alpha} = \bar{y} - \hat{\beta}\bar{x}.$$

By plugging the solution for $\hat{\alpha}$ into the first order condition regarding $\hat{\beta}$ and again considering that $\sum y_i = N\bar{y}$, we get the solution for the slope coefficient estimator:

$$\frac{\sum x_i y_i - N\bar{y}\bar{x}}{\sum x_i^2 - N\bar{x}^2}.$$

To compute the actual estimates, we first compute $\hat{\beta}$ and then $\hat{\alpha}$. With all that, we can implement our OLS estimator for the simple linear regression model in R and apply it to the estimation problem at hand:

```
# implement the simple OLS estimator
# verify implementation with simulated data from above
# my_ols(y,x)
# should be very close to alpha=30 and beta=0.9
my_ols <-
  function(y,x) {
    N <- length(y)
    betahat <- (sum(y*x) - N*mean(x)*mean(y)) / (sum(x^2) - N*mean(x)^2)
    alphahat <- mean(y) - betahat*mean(x)

    return(list(alpha=alphahat,beta=betahat))
  }

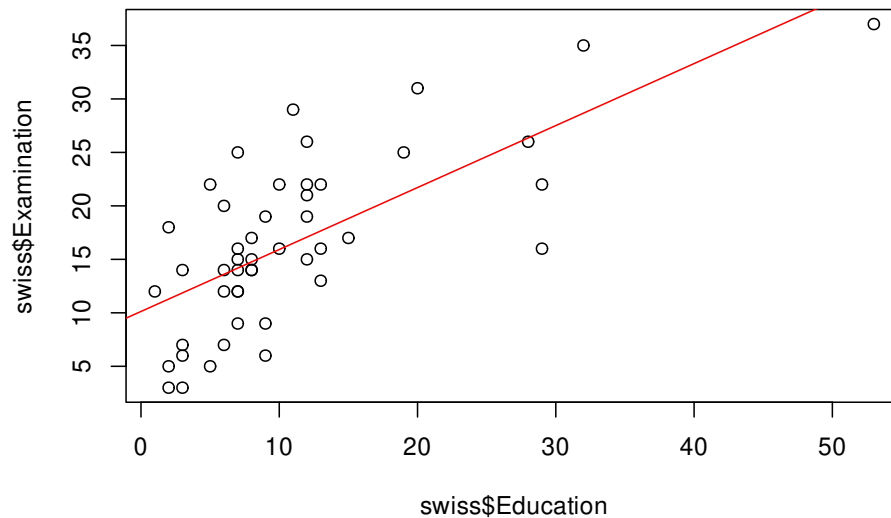
# estimate the effect of Education on Examination
```

```
estimates <- my_ols(swiss$Examination, swiss$Education)
estimates
```

```
## $alpha
## [1] 10.13
##
## $beta
## [1] 0.5795
```

Finally, we can visually inspect the estimated parameters (i.e., does the line fit well with the data?):

```
plot(swiss$Education, swiss$Examination)
abline(estimates$alpha, estimates$beta, col="red")
```



The fit looks rather reasonable. There seems to be indeed a positive relationship between Education and Examination. In the next step, we would likely want to know whether we can say something meaningful about how precisely this relationship is measured statistically (i.e., how 'significant' $\hat{\beta}$ is). We will leave this issue for another class.

9.3.1 Regression toolbox in R

When working on data analytics problems in R, there is usually no need to implement one's own estimator functions. For most data analytics problems, the basic R installation (or an additional R package) already provides easy-to-use functions with many more features than our very simple example above. For example, the work-horse function for linear regressions in R is `lm()`, which accepts regression equation expressions as formulas such as `Examination~Education` and a data-frame with the corresponding dataset as arguments. As a simple example, we use this function to compute the same regression estimates as before:

```
estimates2 <- lm(Examination~Education, data=swiss)
estimates2

##
## Call:
## lm(formula = Examination ~ Education, data = swiss)
##
## Coefficients:
## (Intercept)      Education
##      10.127         0.579
```

With one additional line of code, we can compute all the common statistics about the regression estimation:

```
summary(estimates2)

##
## Call:
## lm(formula = Examination ~ Education, data = swiss)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -10.932  -4.763  -0.184   3.891  12.498
##
## Coefficients:
```

```
##           Estimate Std. Error t value Pr(>|t|)
## (Intercept) 10.1275      1.2859    7.88 5.2e-10 ***
## Education   0.5795      0.0885    6.55 4.8e-08 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.77 on 45 degrees of freedom
## Multiple R-squared:  0.488, Adjusted R-squared:  0.476
## F-statistic: 42.9 on 1 and 45 DF, p-value: 4.81e-08
```

The t-tests displayed in this summary for the intercept (α) and the slope-coefficient concerning Education (β) assess how probable it is to observe such parameter estimates if the true values of these parameters are 0 (this is one way of thinking about statistical ‘significance’). Specifically for the research question in this example: How often would we observe a value of $\hat{\beta} = 0.579..$ or larger if we were to draw a random sample from the same population repeatedly and if more schooling in the population does not increase the average high-success rate at the standardized examination ($\beta = 0$)? The P-values (last column) suggests that this would hardly ever be the case. If we are confident that the factors not considered in this simple model do not affect Education, we could conclude that Education has a significant and positive effect on Examination.

10

Data Visualization

10.1 Data display

In the last part of a data pipeline, we typically deal with data visualization and statistical results for presentation/communication. Typical output formats are reports, a thesis (BA, MA, Dissertation chapter), interactive dashboards, and websites. R (and particularly RStudio) provides a very flexible framework to manage the steps involved in visualization/presentation for all of these output formats. The first (low-level) step in preparing data/results for publication is the formatting of data values for publication. Typically, this involves some string operations to make numbers and text look nicer before we show them in a table or graph.

Consider, for example, the following summary statistics.

```
# load packages and data
library(tidyverse)
data("swiss")
# compute summary statistics
swiss_summary <-
  summarise(swiss,
    avg_education = mean(Education, na.rm = TRUE),
    avg_fertility = mean(Fertility, na.rm = TRUE),
    N = n()
  )
swiss_summary
```

```
##   avg_education avg_fertility N
## 1          10.98         70.14 47
```

We likely do not want to present these numbers with that many decimal places. The function `round()` can take care of this.

```
swiss_summary_rounded <- round(swiss_summary, 2)
swiss_summary_rounded
```

```
##   avg_education avg_fertility  N
## 1           10.98           70.14 47
```

More specific formatting of numeric values is easier when coercing the numbers to character strings (text).¹ For example, depending on the audience (country/region) to which we want to communicate our results, different standards of how to format numbers are expected. In the English-speaking world it is quite common to use `.` as decimal mark, in the German-speaking world it is rather common to use `,`. The `format()`-function provides an easy way to format numbers in this way (once they are coerced to character).

```
swiss_summary_formatted <- format(swiss_summary_rounded, decimal.mark=",")
swiss_summary_formatted
```

```
##   avg_education avg_fertility  N
## 1           10,98           70,14 47
```

R also provides various helpful functions to better format/display text strings. See, for example:

- Uppercase/lowercase: `toupper()/tolower()`.
- Remove white spaces: `trimws()`,

```
string <- "AbCD "
toupper(string)
```

```
## [1] "ABCD "
```

¹Note that this step only makes sense if we are sure that the numeric values won't be further analyzed or used in a plot (except for labels).

```
tolower(string)
```

```
## [1] "abcd "
```

```
trimws(tolower(string))
```

```
## [1] "abcd"
```

10.2 Data visualization with *ggplot2*

A key technique to convincingly communicate statistical results and insights from data is visualization. How can we visualize raw data and insights gained from statistical models with R? It turns out that R is a really useful tool for data visualization, thanks to its very powerful graphics engine (i.e., the underlying low-level R functions that handle things like colors, shapes, etc.). Building on this graphics engine, there are particularly three R packages with a variety of high-level functions to plot data in R:

- The original *graphics* package ([R Core Team \(2018\)](#); shipped with the base R installation).
- The *lattice* package ([Sarkar, 2008](#)), an implementation of the original Bell Labs ‘Trellis’ system.
- The *ggplot2* package ([Wickham, 2016](#)), an implementation of Leland Wilkinson’s ‘Grammar of Graphics’.

While these packages provide well-documented high-level R functions to plot data, their syntax differs in some important ways. For R beginners, it thus makes sense to first learn how to generate plots in R with *one* of these packages. Here, we focus on *ggplot2* because it is part of the *tidyverse*.

10.3 ‘Grammar of Graphics’

A few years back, statistician and computer scientist Leland Wilkinson wrote an influential book called ‘The Grammar of Graphics’. In this book, Wilkinson develops a formal description (‘grammar’) of graphics used in statistics, illustrating how different types of plots (bar plot, histogram, etc.) are special cases of an underlying framework. In short, his idea was that we can think of graphics as consisting of different design layers and thus can build and describe graphics/plots *layer by layer* (see here² for an illustration of this idea).

This framework got implemented in R with the prominent `ggplot2`-package, building on the already powerful R graphics engine. The result is a user-friendly environment to visualize data with enormous potential to plot almost any graphic illustrating data.

10.3.1 `ggplot2` basics

Using `ggplot2` to generate a basic plot in R is quite simple. It involves three key points:

1. The data must be stored in a `data.frame/tibble` (in tidy format).
2. The starting point of a plot is always the function `ggplot()`.
3. The first line of plot code declares the data and the ‘aesthetics’ (e.g., which variables are mapped to the x-/y-axes):

```
ggplot(data = my_dataframe, aes(x= xvar, y= yvar))
```

10.3.2 Tutorial

In the following, we learn the basic functionality of `ggplot` by applying it to the `swiss` dataset.

²<http://bloggotype.blogspot.ch/2016/08/holiday-notes2-grammar-of-graphics.html>

10.3.3 Loading/preparing the data

First, we load and inspect the data. Among other variables, it contains information about the share of inhabitants of a given Swiss province who indicate to be of Catholic faith (and not Protestant).

```
# load the R package
library(ggplot2)
# load the data
data(swiss)
# get details about the data set
# ?swiss
# inspect the data
head(swiss)
```

```
##           Fertility Agriculture Examination
## Courtelary      80.2         17.0          15
## Delemont        83.1         45.1           6
## Franches-Mnt    92.5         39.7           5
## Moutier         85.8         36.5          12
## Neuveville      76.9         43.5          17
## Porrentruy      76.1         35.3           9
##           Education Catholic Infant.Mortality
## Courtelary      12      9.96          22.2
## Delemont         9     84.84          22.2
## Franches-Mnt     5     93.40          20.2
## Moutier          7     33.77          20.3
## Neuveville      15      5.16          20.6
## Porrentruy       7     90.57          26.6
```

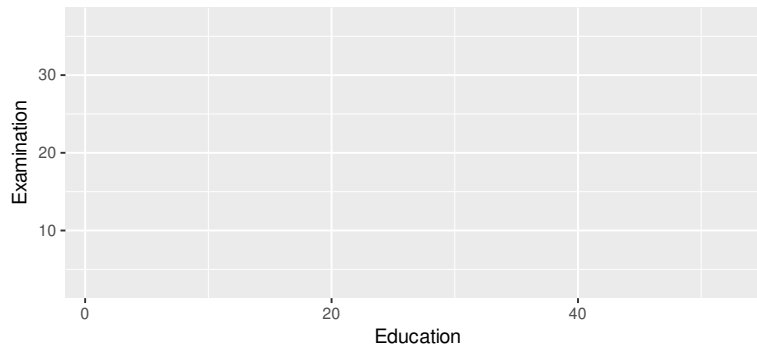
As we do not only want to use this continuous measure in the data visualization, we generate an additional factor variable called `Religion` which has either the value 'Protestant' or 'Catholic' depending on whether more than 50 percent of the inhabitants of the province are Catholics.

```
# code province as 'Catholic' if more than 50% are catholic
swiss$Religion <- 'Protestant'
swiss$Religion[50 < swiss$Catholic] <- 'Catholic'
swiss$Religion <- as.factor(swiss$Religion)
```

10.3.3.1 Data and aesthetics

We initiate the most basic plot with `ggplot()` by defining which data to use and, in the plot aesthetics, which variable to use on the x and y axes. Here, we are interested in whether the level of education beyond primary school in a given district is related to how well draftees from the same district do in a standardized army examination (% of draftees that get the highest mark in the examination).

```
ggplot(data = swiss, aes(x = Education, y = Examination))
```

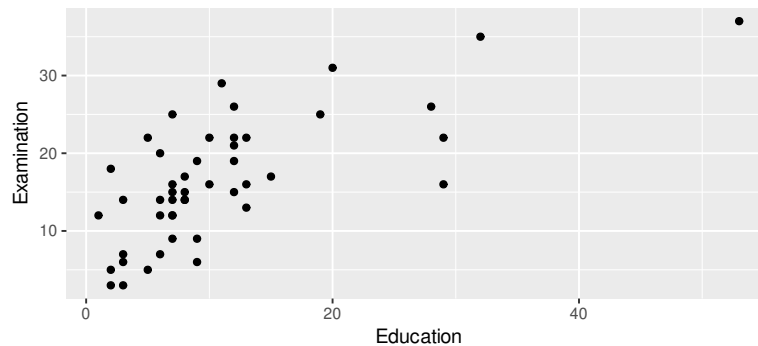


As we have not yet defined according to what rules the data shall be visualized, all we get is an empty ‘canvas’ and the axes (with the respective label and ticks indicating the range of the values).

10.3.4 Geometries (~ type of plot)

To actually plot the data, we have to define the ‘geometries’, defined according to which function the data should be mapped/visualized. In other words, geometries define which ‘type of plot’ we use to visualize the data (histogram, lines, points, etc.). In the example code below, we use `geom_point()` to get a simple point plot.

```
ggplot(data = swiss, aes(x = Education, y = Examination)) +  
  geom_point()
```

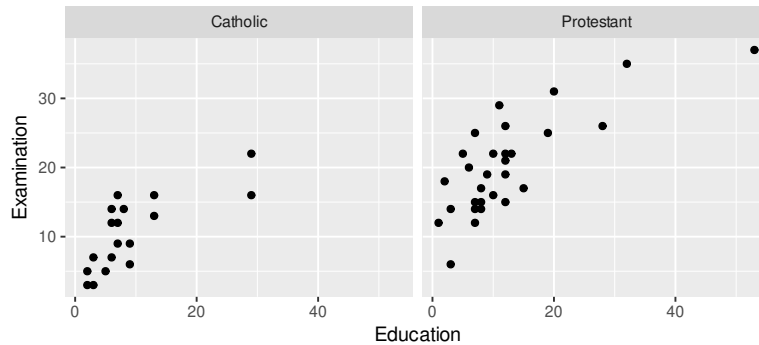


The result indicates a positive correlation between the level of education and how well draftees do in the examination. We want to understand this correlation better. Particularly what other factors could drive this picture?

10.3.4.1 Facets

According to a popular thesis, the protestant reformation and the spread of the protestant movement in Europe were driving the development of compulsory schooling. It would thus be reasonable to hypothesize that the picture we see is partly driven by differences in schooling between Catholic and Protestant districts. To make such differences visible in the data, we use 'facets' to show the same plot again, but this time separating observations from Catholic and Protestant districts:

```
ggplot(data = swiss, aes(x = Education, y = Examination)) +  
  geom_point() +  
  facet_wrap(~Religion)
```



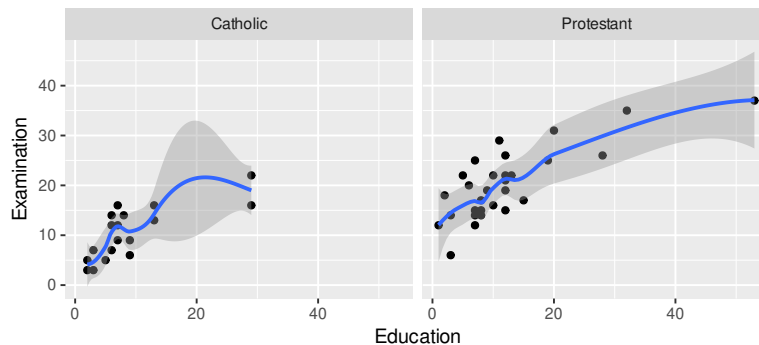
Draftees from protestant districts tend to do generally better (which might indicate better primary schools or a generally stronger focus on the educational achievements of Protestant children). However, the relationship between education (beyond primary schools) and examination success seems to hold for either type of district.

10.3.4.2 Additional layers and statistics

Let's visualize this relationship more clearly by drawing trend lines through the scattered diagrams. Once with the non-parametric 'loess'-approach and once forcing a linear model on the relationship between the two variables.

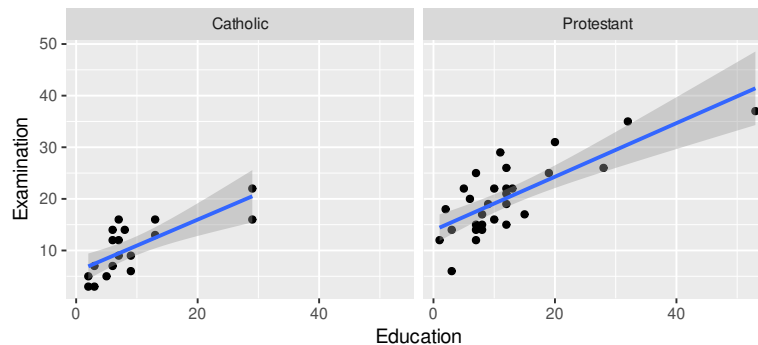
```
ggplot(data = swiss, aes(x = Education, y = Examination)) +
  geom_point() +
  geom_smooth(method = 'loess') +
  facet_wrap(~Religion)
```

`geom_smooth()` using formula 'y ~ x'



```
ggplot(data = swiss, aes(x = Education, y = Examination)) +
  geom_point() +
  geom_smooth(method = 'lm') +
  facet_wrap(~Religion)
```

```
## `geom_smooth()` using formula 'y ~ x'
```

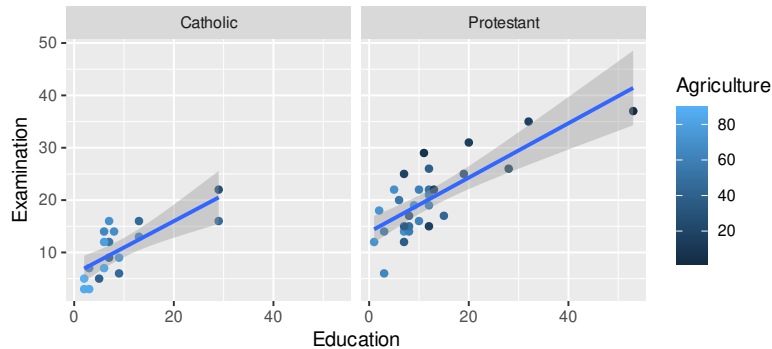


10.3.4.3 Additional aesthetics

Knowing a little about Swiss history and geography, we realize that rural cantons in mountain regions remained Catholic during the Reformation. In addition, cantonal school systems historically considered that children have to help their parents on the farms during the summers. Thus in some rural cantons, schools were closed from spring until autumn. Hence, we might want to indicate in the plot which point refers to a predominantly agricultural district. We use the aesthetics of the point geometry to color the points according to the 'Agriculture' variable (the % of males involved in agriculture as an occupation).

```
ggplot(data = swiss, aes(x = Education, y = Examination)) +
  geom_point(aes(color = Agriculture)) +
  geom_smooth(method = 'lm') +
  facet_wrap(~Religion)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



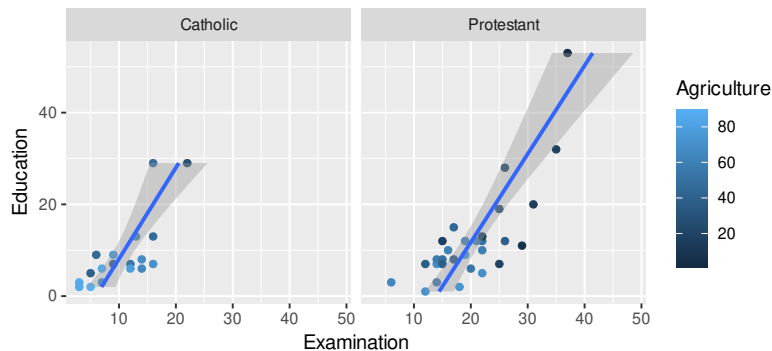
The resulting picture is in line with what we have expected. Overall, the districts with a lower share of agricultural occupation tend to have higher levels of education and achievements in the examination.

10.3.4.4 Coordinates/Themes: Fine-tuning the plot

Finally, there are countless options to refine the plot further. For example, we can easily change the orientation/coordinates of the plot:

```
ggplot(data = swiss, aes(x = Education, y = Examination)) +
  geom_point(aes(color = Agriculture)) +
  geom_smooth(method = 'lm') +
  facet_wrap(~Religion) +
  coord_flip()
```

`geom_smooth()` using formula 'y ~ x'

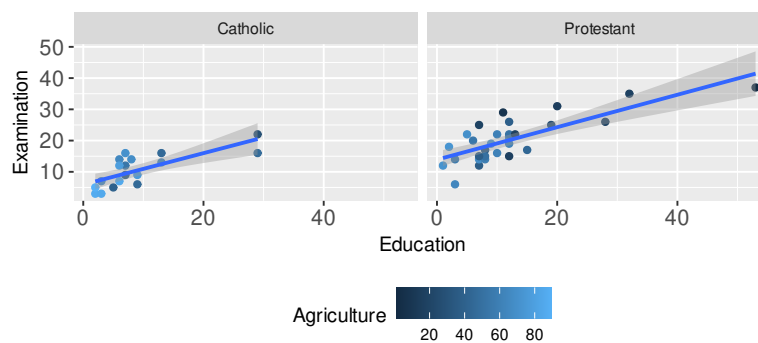


In addition, the `theme()`-function allows changing almost every aspect

of the plot (margins, font face, font size, etc.). For example, we might prefer to have the plot legend at the bottom and have larger axis labels.

```
ggplot(data = swiss, aes(x = Education, y = Examination)) +
  geom_point(aes(color = Agriculture)) +
  geom_smooth(method = 'lm') +
  facet_wrap(~Religion) +
  theme(legend.position = "bottom", axis.text=element_text(size=12) )
```

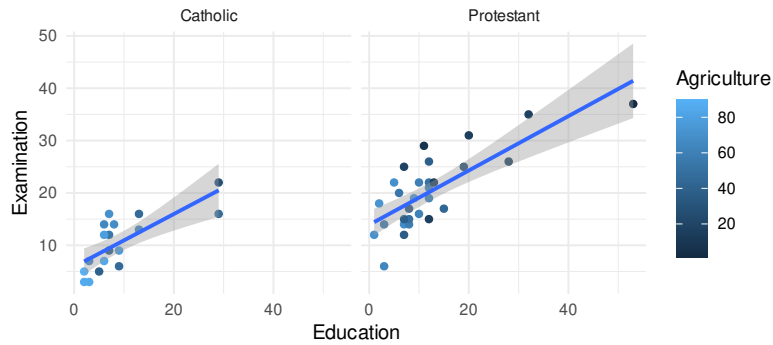
'geom_smooth()' using formula 'y ~ x'



Moreover, several theme templates offer ready-made designs for plots:

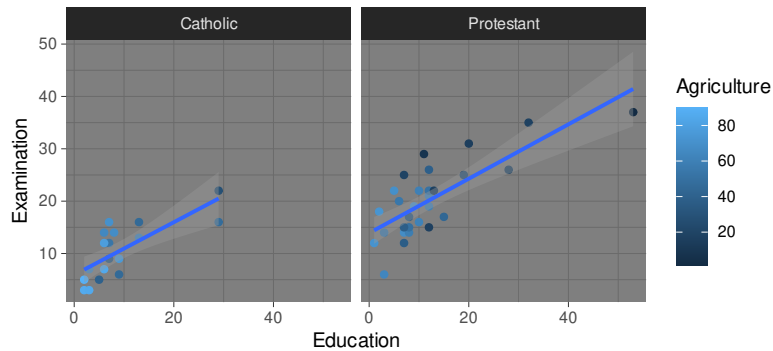
```
ggplot(data = swiss, aes(x = Education, y = Examination)) +
  geom_point(aes(color = Agriculture)) +
  geom_smooth(method = 'lm') +
  facet_wrap(~Religion) +
  theme_minimal()
```

'geom_smooth()' using formula 'y ~ x'



```
ggplot(data = swiss, aes(x = Education, y = Examination)) +
  geom_point(aes(color = Agriculture)) +
  geom_smooth(method = 'lm') +
  facet_wrap(~Religion) +
  theme_dark()
```

`geom_smooth()` using formula 'y ~ x'



10.4 Dynamic documents

Dynamic documents are a way to directly/dynamically integrate the results of an analysis in R (numbers, tables, plots) in written text (a report, thesis, slide set, website, etc.). That is, we can write a report in the so-called 'R-Markdown' format and place it directly in the same document 'chunks' of R code, which we want to be executed each time

we ‘knit’ the report. Knitting the document means that the following steps are executed under the hood:

1. The code in the R chunks is executed, and the results are cached and formatted for print.
2. The formatted R output is embedded in a so-called ‘Markdown’-file (.md).
3. The markdown file is rendered as either a PDF, HTML, or Word file (with additional formatting options, depending on the output format).

The entire procedure of importing, cleaning, analyzing, and visualizing data can thus be combined in one document, based on which we can generate a meaningful output to communicate our results.

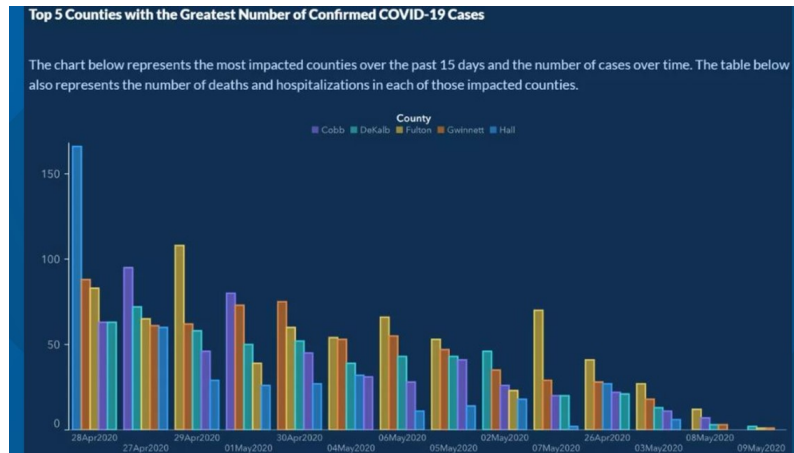
10.5 Tutorial: How to give the wrong impression with data visualization

10.5.1 Background and aim

Referring to the supposed trade-off between public health and economic performance in the context of the SARS-CoV-2 pandemic, Georgia Governor Brian Kemp has “reopened the state on April 24 [2020], citing a downward trend of COVID-19 cases being reported by the Georgia Department of Health [...] Georgians have been relying on this data (number of cases, hospitalizations and deaths in their areas) to determine whether or not it’s safe for them to go out.” (FIRSTCOAST NEWS³) It later turned out that there was no downward trend at all but that the data on Covid cases was intentionally visualized in such a way to give the impression of a downward trend. In this tutorial, we

³<https://www.firstcoastnews.com/article/news/local/georgia/georgia-data-numbers-misrepresented/77-08c31538-3f26-4348-9f30-d81b11dd4d24>

aim to replicate the doctored downward trend visualization, using the original data and `ggplot2`.⁴



10.5.2 Data import and preparation

The original raw data is provided in a zip-file (compressed file). The code chunk below shows some tricks of how to download and import data from an online zip-file as part of a simple data analytics script. First, we generate a temporary file with `tempfile()`. We then download the compressed zip file containing the Covid-data to the temporary file via `download.file(DATA_URL, destfile = tmp)`. Finally, we use `unzip(tmp, files = "epicurve_rpt_date.csv")` in order to decompress the one CSV-file in the zip-file we want to work with. After that, we can simply import the CSV-file the usual way (with the `read_csv()` function).

```
# SET UP -----

# load packages
library(tidyverse)
library(ggplot2)
```

⁴Note that the original data is not 100% identical with the data used in the original plot. The reason is that Covid-numbers are updated/corrected over time, and the original plot uses very recent data.

```

# fix vars
DATA_URL <- "https://ga-covid19.ondemand.sas.com/docs/ga_covid_data.zip"
COUNTIES <- c("Cobb", "DeKalb", "Fulton", "Gwinnett", "Hall")
MIN_DATE <- as.Date("2020-04-26")
MAX_DATE <- as.Date("2020-05-09")

# FETCH/IMPORT DATA -----

# create a temporary file to store the downloaded zipfile
tmp <- tempfile()
download.file(DATA_URL, destfile = tmp)
# unzip the file we need
path <- unzip(tmp, files = "epicurve_rpt_date.csv" )
# read the data
cases <- read_csv(path)
# inspect the data
head(cases)

## # A tibble: 6 x 20
##   measure      county report_date cases deaths cases_~1
##   <chr>         <chr>   <date>     <dbl> <dbl>    <dbl>
## 1 state_total Georgia 2020-02-01      0      0        0
## 2 state_total Georgia 2020-02-02      0      0        0
## 3 state_total Georgia 2020-02-03      0      0        0
## 4 state_total Georgia 2020-02-04      0      0        0
## 5 state_total Georgia 2020-02-05      0      0        0
## 6 state_total Georgia 2020-02-06      0      0        0
## # ... with 14 more variables: death_cum <dbl>,
## #   moving_avg_cases <dbl>, moving_avg_deaths <dbl>,
## #   antigen_cases <dbl>, probable_deaths <dbl>,
## #   antigen_case_hospitalization <dbl>,
## #   confirmed_case_hospitalization <dbl>,
## #   antigen_cases_cum <dbl>,
## #   probable_deaths_cum <dbl>, total_cases <dbl>, ...
## # i Use `colnames()` to see all variable names

```

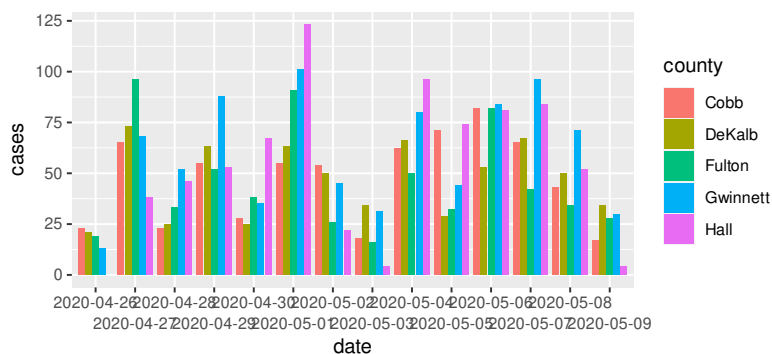
Once the data is imported, we select/filter the part of the dataset used in the original bar plot. Note the tidyverse-functions introduced in previous lectures to prepare an analytic dataset out of the raw data efficiently. At one point, we use the `order()`-function to order the observations according to the report date of covid cases. This is done when displaying the frequency of cases over time. This way, in the following plot, the x-axis serves as a time axis, displaying the date of the corresponding reported case numbers from the beginning of the observation period on the left to the end of the observation period on the right).

```
# only observations in the five major counties and during the relevant days
cases <- filter(cases, county %in% COUNTIES, MIN_DATE <= report_date, report_date <= MAX_DATE)
# only the relevant variables
cases <- select(cases, county, report_date, cases)
# order according to date
cases <- cases[order(cases$report_date),]
# we add an additional column in which we treat dates as categories
cases <- mutate(cases, date=factor(report_date))
```

10.5.3 Plot

First, we show what the actual (honest) plot would essentially look like.

```
ggplot(cases, aes(y=cases, x=date, fill=county)) +
  geom_bar(position = position_dodge2(), stat = "identity") +
  scale_x_discrete(guide = guide_axis(n.dodge = 2)) # this avoids overlapping of x-axis labels
```

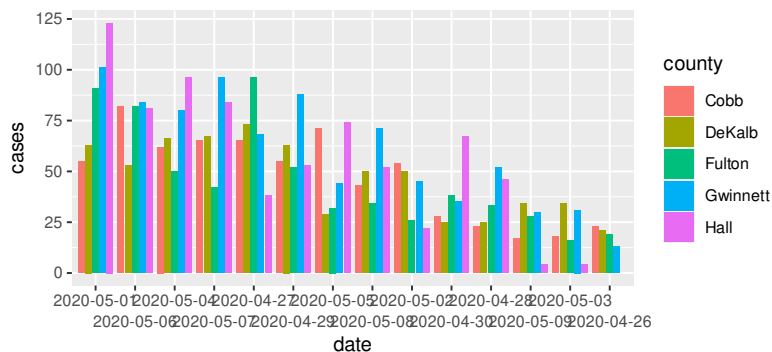


There is no obvious downward trend visible. Now, let us look at what steps are involved in manipulating the visualization in order to give the wrong impression of a downward trend. Importantly, all the manipulation is purely done via the data plotting. We do not touch/manipulate the underlying data (such as removing observations or falsifying numbers).

Two things become apparent when comparing the original plot with what we have so far. First, the order of days on the X-axis is not chronological but seems to be based on the total number of cases per day. Second, the order of the bars of a given day does not follow the alphabetical order of the county names but the number of cases in each of the counties. Let's address the two aspects one at the time. To change the order on the x-axis, we have to re-order the factor levels in `date`.

```
cases2 <- mutate(cases, date= fct_reorder(date, cases, sum,.desc = TRUE)) # re-order the dates

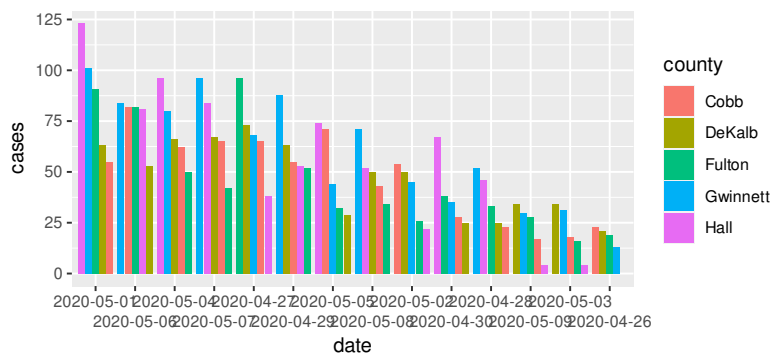
ggplot(cases2, aes(y=cases, x=date, fill=county)) +
  geom_bar(position = position_dodge2(), stat = "identity") +
  scale_x_discrete(guide = guide_axis(n.dodge = 2)) # this avoids overlapping of x-axis labels
```



Note that the number of cases is not exactly the same as in the original. Quite the reason for this is that the numbers have been updated during May. Given that the overall pattern is very similar, there is no reason to believe that the actual numbers underlying the original figure had been manipulated too. Now, let us address the second aspect (ordering of bars per date). For this, we use the `group` aesthetic, indicating to `ggplot` that we want the number of cases to be

used to order the bars within each point in time (date). Setting `position_dodge2(reverse=TRUE)` simply means we want the bars per date to be ordered in decreasing order (per default, it would be increasing; try out the difference by changing it to `position_dodge2(reverse=FALSE)` or simply `position_dodge2()`).

```
ggplot(cases2, aes(y=cases, x=date, fill=county, group=cases)) +
  geom_bar(aes(group=cases), position = position_dodge2(reverse=TRUE), stat = "identity") +
  scale_x_discrete(guide = guide_axis(n.dodge = 2)) # this avoids overlapping of x-axis labels
```



This already looks much more like the original plot.

10.5.4 Cosmetics: theme

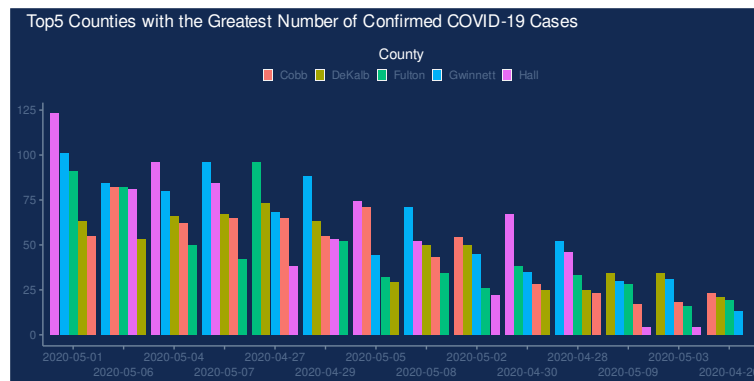
Finally, we tweaked the plot's theme to make it look more similar to the original. The following code is a first shot at addressing the most obvious aspects to make the plot more similar to the original. More steps might be needed to make it essentially identical (consider, for example, the color scheme of the bars). This part of the tutorial is a nice illustration of how versatile the `ggplot2-theme()`-function is to tweak every cosmetic detail of a plot.

```
ggplot(cases2, aes(y=cases, x=date, fill=county, group=cases)) +
  geom_bar(aes(group=cases), position = position_dodge2(reverse=TRUE), stat = "identity") +
  ggtitle("Top5 Counties with the Greatest Number of Confirmed COVID-19 Cases") +
  scale_x_discrete(guide = guide_axis(n.dodge = 2)) + # this avoids overlapping of x-axis labels
  guides(fill=guide_legend(title = "County",
```

```

        title.position = "top",
        direction = "horizontal",
        title.hjust = 0.5)) +
theme(legend.position = "top",
      plot.title = element_text(colour = "white", hjust = -0.1),
      legend.key.size = unit(0.5,"line"),
      panel.grid.major = element_blank(),
      panel.grid.minor = element_blank(),
      panel.background = element_rect(fill="#132a52"),
      legend.background = element_rect(fill="#132a52"),
      plot.background = element_rect(fill="#132a52"),
      legend.text = element_text(colour = "#536b8d"),
      legend.title = element_text(colour = "white"),
      axis.text = element_text(colour = "#536b8d"),
      axis.ticks = element_line(colour = "#71869e"),
      axis.line = element_line(colour="#71869e"),
      axis.title = element_blank())

```





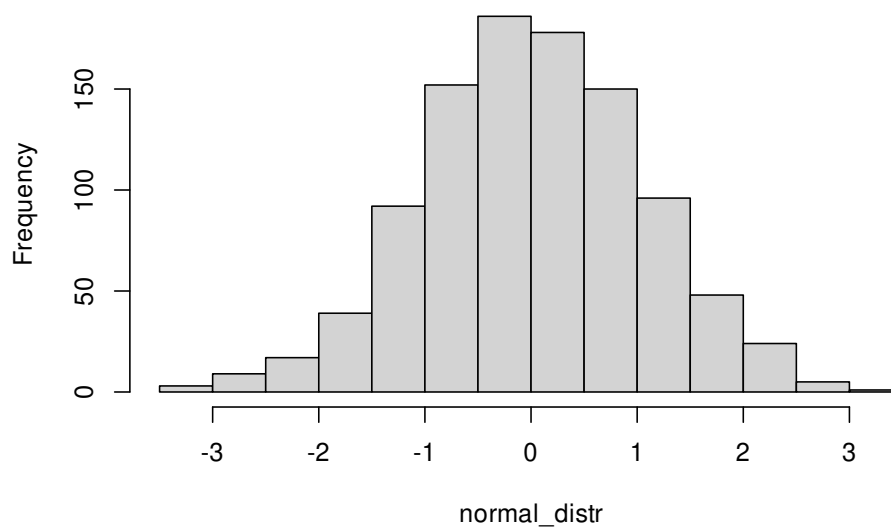
Appendix A

.1 Understanding statistics and probability with code

.1.1 Random draws and distributions

```
normal_distr <- rnorm(1000)  
hist(normal_distr)
```

Histogram of normal_distr

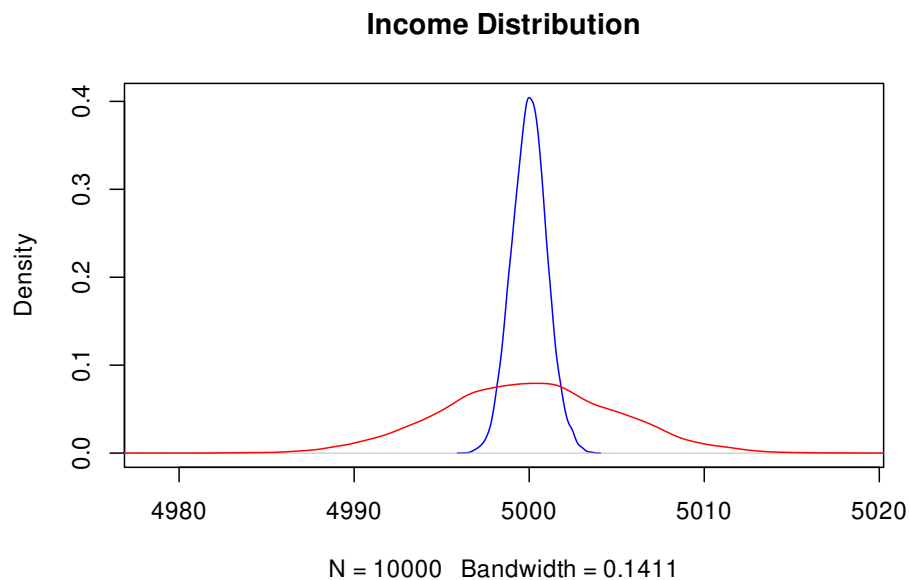


.1.2 Illustration of variability

```
# draw a random sample from a normal distribution with a large standard deviation  
largevar <- rnorm(10000, mean = 5000, sd = 5)  
# draw a random sample from a normal distribution with a small standard deviation
```

```
littlevvar <- rnorm(10000, mean = 5000, sd = 1)

# visualize the distributions of both samples with a density plot
plot(density(littlevvar), col = "blue",
      xlim=c(min(largevar), max(largevar)), main="Income Distribution")
lines(density(largevar), col = "red")
```



Note: the red curve illustrates the distribution of the sample with a large standard deviation (a lot of variability) whereas the blue curve illustrates the one with a rather small standard deviation.

.1.3 Skewness and Kurtosis

```
# Install the R-package called "moments" with the following command (if not installed yet):
# install.packages("moments")

# load the package
library(moments)
```

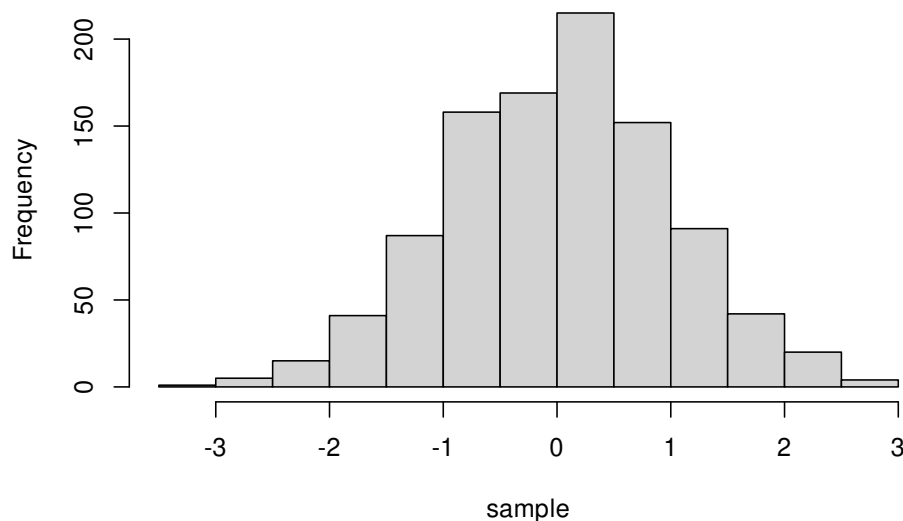
.1.3.1 Skewness

Skewness refers to how symmetric the frequency distribution of a variable is. For example, a distribution can be ‘positively skewed’ meaning it has a long tail on the right and a lot of ‘mass’ (observations) on the left. We can see that when visualizing the distribution in a histogram or a density plot. In R this looks as follow (consider the comments in the code explaining what each line does):

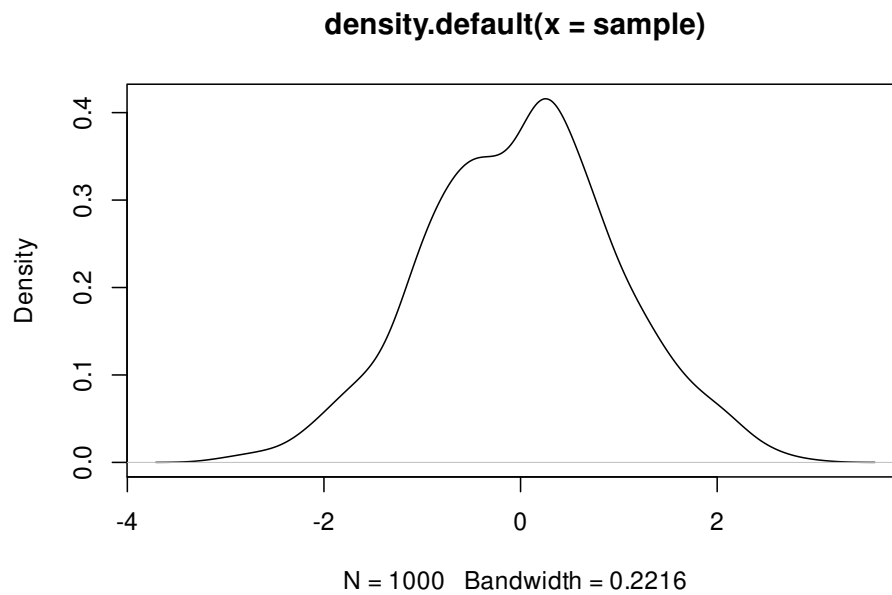
```
# draw a random sample of simulated data from a normal distribution
# the sample is of size 1000 (hence, n = 1000)
sample <- rnorm(n = 1000)

# plot a histogram and a density plot of that sample
# note that the distribution is neither strongly positively nor negatively skewed
# (this is to be expected, as we have drawn a sample from a normal distribution)
hist(sample)
```

Histogram of sample



```
plot(density(sample))
```



```
# now compute the skewness
```

```
skewness(sample)
```

```
## [1] -0.01866
```

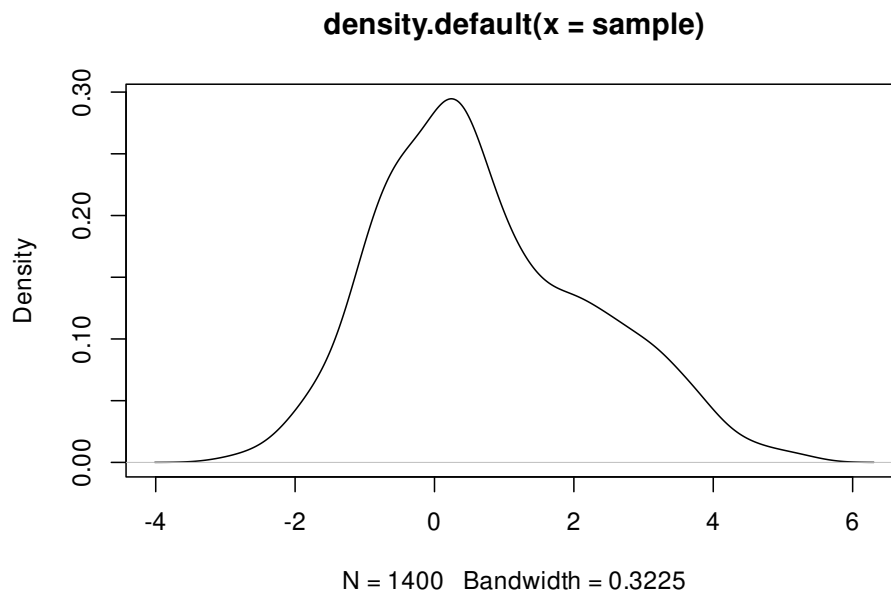
```
# Now we intentionally change our sample to be strongly positively skewed
```

```
# We do that by adding some outliers (observations with very high values) to the sample
```

```
sample <- c(sample, (rnorm(200) + 2), (rnorm(200) + 3))
```

```
# Have a look at the distribution and re-calculate the skewness
```

```
plot(density(sample))
```



```
skewness(sample)
```

```
## [1] 0.4765
```

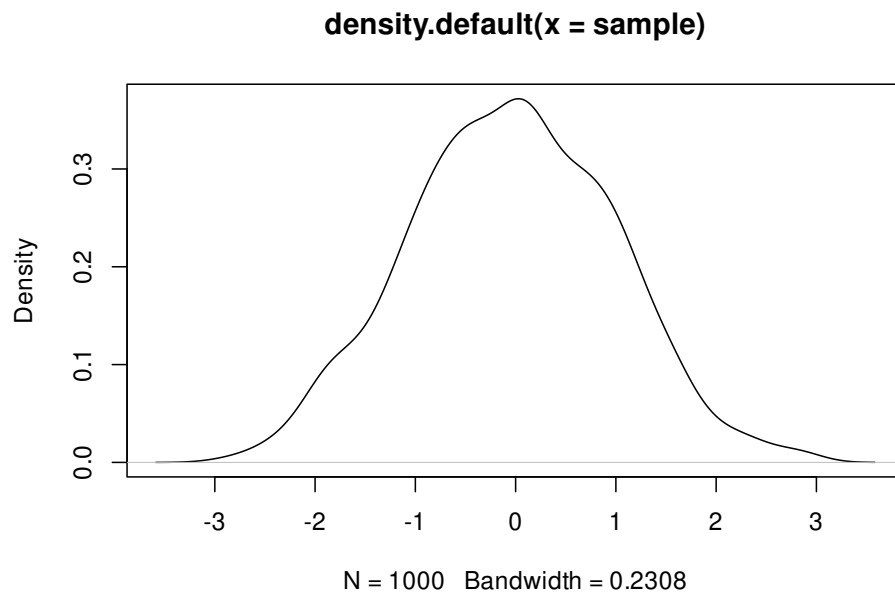
```
#
```

.1.3.2 Kurtosis

Kurtosis refers to how much ‘mass’ a distribution has in its ‘tails’. It thus tells us something about whether a distribution tends to have a lot of outliers. Again, plotting the data can help us understand this concept of kurtosis. Lets have a look at this in R (consider the comments in the code explaining what each line does):

```
# draw a random sample of simulated data from a normal distribution
# the sample is of size 1000 (hence, n = 1000)
sample <- rnorm(n = 1000)

# plot the density & compute the kurtosis
plot(density(sample))
```



```
kurtosis(sample)
```

```
## [1] 2.763
```

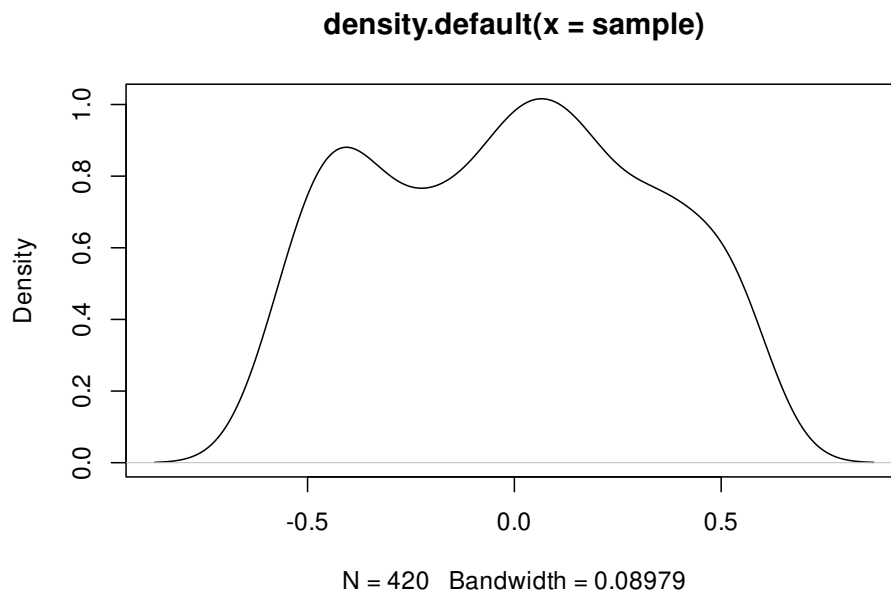
```
# now lets remove observations from the extremes in this distribution
```

```
# we thus intentionally alter the distribution to have less mass in its tails
```

```
sample <- sample[ sample > -0.6 & sample < 0.6]
```

```
# plot the distribution again and see how the tails of it (and thus the kurtosis) has changed
```

```
plot(density(sample))
```



```
# re-calculate the kurtosis  
kurtosis(sample)
```

```
## [1] 1.888
```

```
# as expected, the kurtosis has now a lower value
```

.1.3.3 Implement the formulas for skewness and kurtosis in R

Skewness

```
# own implementation  
sum((sample-mean(sample))^3) / ((length(sample)-1) * sd(sample)^3)
```

```
## [1] 0.01567
```

```
# implementation in moments package  
skewness(sample)
```

```
## [1] 0.01569
```

Kurtosis

```
# own implementation
sum((sample-mean(sample))^4) / ((length(sample)-1) * sd(sample)^4)
```

```
## [1] 1.883
```

```
# implementation in moments package
kurtosis(sample)
```

```
## [1] 1.888
```

.1.4 The Law of Large Numbers

The Law of Large Numbers (LLN) is an important statistical property which essentially describes how the behavior of sample averages is related to sample size. Particularly, it states that the sample mean can come as close as we like to the mean of the population from which it is drawn by simply increasing the sample size. That is, the larger our randomly selected sample from a population, the closer is that sample's mean to the mean of the population.

Think of playing dice. Each time we roll a fair die, the result is either 1, 2, 3, 4, 5, or 6, whereby each possible outcome can occur with the same probability ($1/6$). In other words we randomly draw die-values. Thus we can expect that the average of the resulting die values is $(1 + 2 + 3 + 4 + 5 + 6)/6 = 3.5$.

We can investigate this empirically: We roll a fair die once and record the result. We roll it again, and again we record the result. We keep rolling the die and recording results until we get 100 recorded results. Intuitively, we would expect to observe each possible die-value about equally often (given that the die is fair) because each time we roll the die, each possible value (1,2,...,6) is equally likely to be the result. And we would thus expect the average of the resulting die values to be around 3.5. However, just by chance it can obviously be the case that one value (say 5) occurs slightly more often than another value (say 1), leading to a sample mean slightly larger than 3.5. In this context, the

LLN states that by increasing the number of times we are rolling the die, we will get closer and closer to 3.5. Now, let's implement this experiment in R:

```
# first we define the potential values a die can take
dvalues <- 1:6 # the : operator generates a regular sequence of numbers (from:to)
dvalues

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

```
# define the size of the sample n (how often do we roll the die...)
# for a start, we only roll the die ten times
n <- 10
# draw the random sample: 'roll the die n times and record each result'
results <- sample( x = dvalues, size = n, replace = TRUE)
# compute the mean
mean(results)
```

```
## [1] 4.2
```

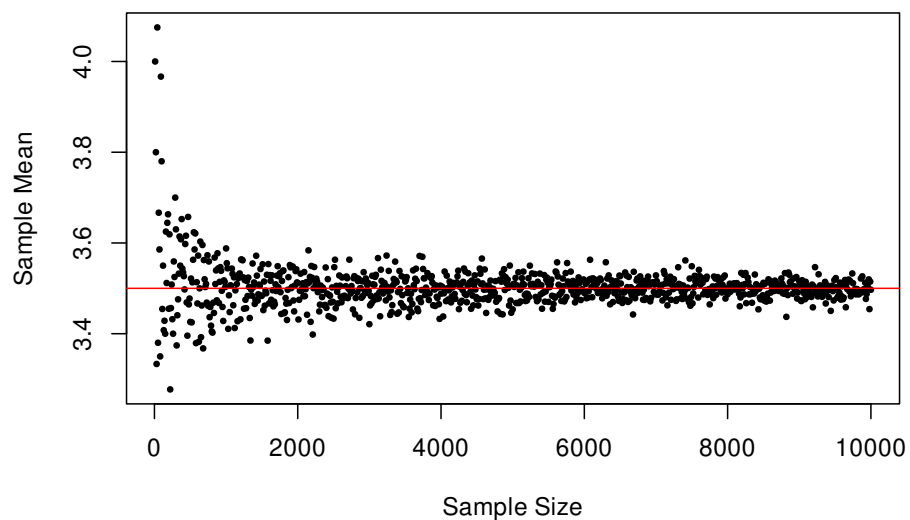
As you can see we are relatively close to 3.5, but not quite there. So let's roll the die more often and calculate the mean of the resulting values again:

```
n <- 100
# draw the random sample: 'roll the die n times and record each result'
results <- sample( x = dvalues, size = n, replace = TRUE)
# compute the mean
mean(results)
```

```
## [1] 3.69
```

We are already close to 3.5! Now let's scale up these comparisons and show how the sample means are getting even closer to 3.5 when increasing the number of times we roll the die up to 10'000.

```
# Essentially, what we are doing here is repeating the experiment above many times,  
# each time increasing n.  
# define the set of sample sizes  
ns <- seq(from = 10, to = 10000, by = 10)  
# initiate an empty list to record the results  
means <- list()  
length(means) <- length(ns)  
# iterate through each sample size: 'repeat the die experiment for each sample size'  
for (i in 1:length(ns)) {  
  
    means[[i]] <- mean(sample( x = dvalues,  
                              size = ns[i],  
                              replace = TRUE))  
}  
  
# visualize the result: plot sample means against sample size  
plot(ns, unlist(means),  
      ylab = "Sample Mean",  
      xlab = "Sample Size",  
      pch = 16,  
      cex = .6)  
abline(h = 3.5, col = "red")
```



We observe that with smaller sample sizes the sample means are broadly spread around the population mean of 3.5. However, the more we go to the right extreme of the x-axis (and thus the larger the sample size), the narrower the sample means are spread around the population mean.

1.5 The Central Limit Theorem

The Central Limit Theorem (CLT) is an almost miraculous statistical property enabling us to test the statistical significance of a statistic such as the mean. In essence, the CLT states that as long as we have a large enough sample, the t-statistic (applied, e.g., to test whether the mean is equal to a particular value) is approximately standard normal distributed. This holds independently of how the underlying data is distributed.

Consider the dice-play-example above. We might want to statistically test whether we are indeed playing with a fair die. In order to test that we would roll the die 100 times and record each resulting value. We would then compute the sample mean and standard deviation in order to assess how likely it was to observe the mean we observe if the population mean actually is 3.5 (thus our H_0 would be $\text{pop_mean} = 3.5$, or in plain English ‘the die is fair’). However, the distribution of the resulting die values are not standard normal distributed. So how can we interpret the sample standard deviation and the sample mean in the context of our hypothesis?

The simplest way to interpret these measures is by means of a *t-statistic*. A t-statistic for the sample mean under our working hypothesis that $\text{pop_mean} = 3.5$ is constructed as $t(3.5) = (\text{sample_mean} - 3.5) / (\text{sample_sd} / \sqrt{n})$. Let’s illustrate this in R:

```
# First we roll the die like above
n <- 100
# draw the random sample: 'roll the die n times and record each result'
results <- sample( x = dvalues, size = n, replace = TRUE)
# compute the mean
sample_mean <- mean(results)
```

```
# compute the sample SD
sample_sd <- sd(results)
# estimated standard error of the mean
mean_se <- sample_sd/sqrt(n)

# compute the t-statistic:
t <- (sample_mean - 3.5) / mean_se
t
```

```
## [1] 0.1671
```

At this point you might wonder what the use of t is if the underlying data is not drawn from a normal distribution. In other words: what is the use of t if we cannot interpret it as a value that tells us how likely it is to observe this sample mean, given our null hypothesis? Well, actually we can. And here is where the magic of the CLT comes in: It turns out that there is a mathematical proof (i.e. the CLT) which states that the t -statistic itself can arbitrarily well be approximated by the standard normal distribution. This is true independent of the distribution of the underlying data in our sample! That is, if we have a large enough sample, we can simply compute the t -statistic and look up how likely it is to observe a value at least as large as t , given the null hypothesis is true (\rightarrow the p -value):

```
# calculate the p-value associated with the t-value calculated above
2*pnorm(-abs(t))
```

```
## [1] 0.8673
```

In that case we could not reject the null hypothesis of a fair die. However, as pointed out above, the t -statistic is only asymptotically (meaning with very large samples) normally distributed. We might not want to trust this hypothesis test too much because we were using a sample of only 100 observations.

Let's turn back to R in order to illustrate the CLT at work. Similar to the illustration of the LLN above, we will repeatedly compute the t -statistic

of our dice play experiment and for each trial increase the number of observations.

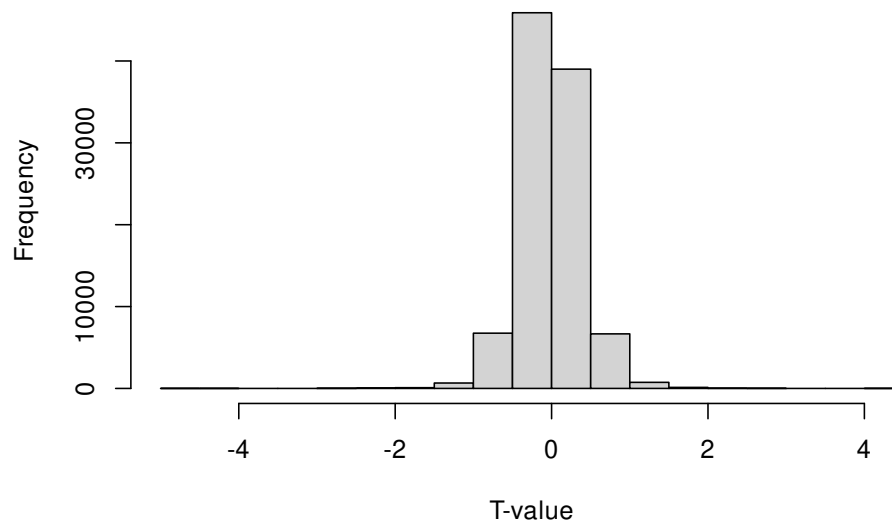
```
# define the set of sample sizes
ns <- c(10, 40, 100)
# initiate an empty list to record the results
ts <- list()
length(ts) <- length(ns)
# iterate through each sample size: 'repeat the die experiment for each sample size'
for (i in 1:length(ns)) {

  samples.i <- sapply(1:100000, function(j) sample( x = dvalues,
                                                    size = ns[i],
                                                    replace = TRUE))

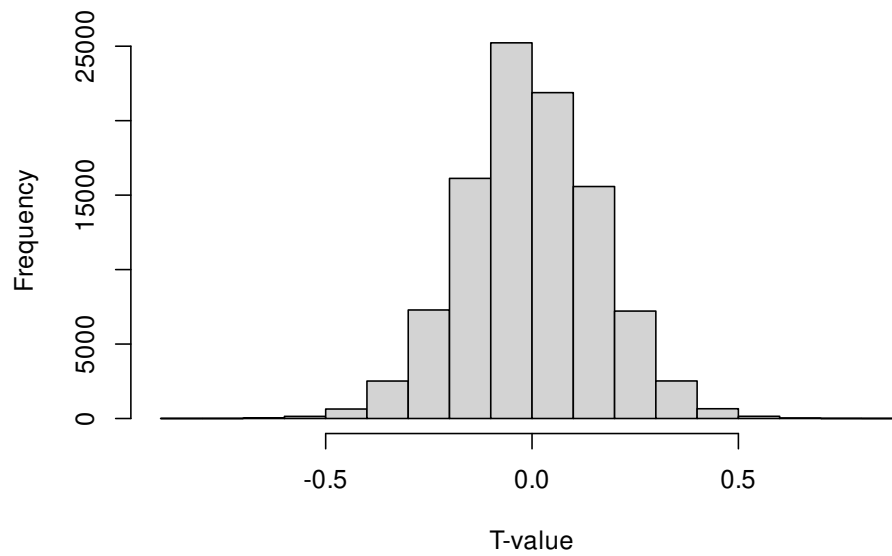
  ts[[i]] <- apply(samples.i, function(x) (mean(x) - 3.5) / sd(x), MARGIN = 2)
}

# visualize the result: plot the density for each sample size

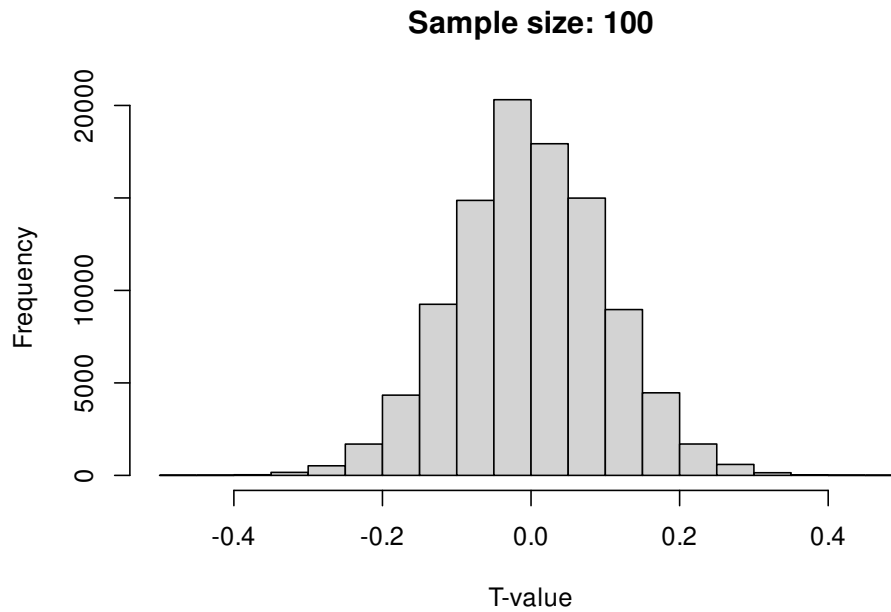
# plot the density for each set of t values
hist(ts[[1]], main = "Sample size: 10", xlab = "T-value")
```

Sample size: 10

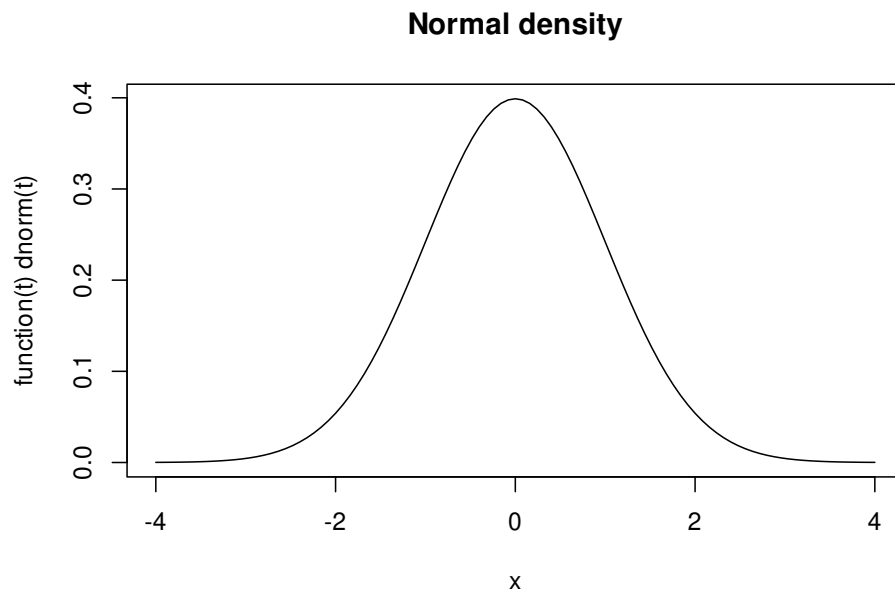
```
hist(ts[[2]], main = "Sample size: 40", xlab = "T-value")
```

Sample size: 40

```
hist(ts[[3]], main = "Sample size: 100", xlab = "T-value")
```



```
# finally have a look at the actual standard normal distribution as a reference point  
plot(function(t)dnorm(t), -4, 4, main = "Normal density")
```



Note how the histogram is getting closer to a normal distribution with increasing sample size.

Bibliography

- Antonio, N., de Almeida, A., and Nunes, L. (2017). Predicting hotel bookings cancellation with a machine learning classification model. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1049–1054.
- Antonio, N., de Almeida, A., and Nunes, L. (2019). Hotel booking demand datasets. *Data in Brief*, 22:41 – 49.
- Einav, L. and Levin, J. (2014). Economics in the age of big data. *Science*, 346(6210):1243089–1–1243089–6.
- Hogben, L. (1983). *Mathematics for the Million*. W.W Norton & Company, New York.
- Matter, U. (2018a). A brief introduction to programming with r.
- Matter, U. (2018b). Introduction to web data mining for social scientists.
- Matter, U. and Stutzer, A. (2015). pvsR: An Open Source Interface to Big Data on the American Political Sphere. *PLOS ONE*, 10(7):1–21. ([R package](<https://cran.r-project.org/web/packages/pvsR/index.html>), [Code](<https://github.com/umatter/pvsR>), [Blogpost](<http://www.bitss.org/2016/08/07/open-source-interfaces-with-the-programmable-web-facilitate-replications-of-big-data-analyses-in-social-science-research/>)).
- Murrell, P. (2009). *Introduction to Data Technologies*. CRC Press, London, UK.
- R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

- Sarkar, D. (2008). *Lattice: Multivariate Data Visualization with R*. Springer, New York. ISBN 978-0-387-75968-5.
- Wickham, H. (2014). Tidy data. *Journal of Statistical Software*, 59(10):1–23.
- Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.
- Wickham, H. and Golemund, G. (2017). O'Reilly, Sebastopol, CA.