

Ulrich Matter

A Brief Introduction to Programming with R

To the students who have motivated and inspired this work.

Contents

List of Tables	v
List of Figures	vii
Preface	ix
1 Why learn to program (now)?	1
1.1 Why R?	2
1.1.1 The ‘data language’	2
1.1.2 High-level language, relatively easy to learn	2
1.1.3 Free, open source, large community	2
1.2 The tools: R/RStudio	3
1.2.1 The R-Console	5
1.2.2 R-Scripts	5
1.2.3 R Environment	6
1.2.4 File Browser	6
1.3 Exercises	7
1.3.1 Exercise A: Setting up a Working Environment	7
1.3.2 Exercise B: R Scripts	8
2 First steps with R	9
2.1 Variables and Vectors	9
2.2 Math Operators	10
2.3 Basic programming concepts in R	13
2.3.1 Loops	13
2.3.2 Booleans and Logical Statements	17
2.3.3 R Functions	18
2.4 Data Structures and Indices	20
2.4.1 Vectors and Lists	20
2.4.2 Lists	21

2.4.3	Matrices and Data Frames	23
2.4.4	Classes and Data Structures	25
2.5	Exercises	26
2.5.1	Exercise A: Write a Sum Function	26
2.5.2	Exercise B: Robustness and Warnings	26
2.5.3	Exercise C: Standard Deviation Function	27
2.5.4	Exercise D: Standard Error Function	27
2.5.5	Exercise E: T-test	28
3	Working with Data	29
3.1	Basics Loading/Importing Data	29
3.2	Loading built-in datasets	29
3.2.1	Importing data from files	31
3.3	Data Visualization with <i>R</i> (<i>ggplot2</i>)	33
3.3.1	‘Grammar of Graphics’	33
3.4	<i>ggplot2</i> basics	33
3.4.1	Tutorial	34
3.4.2	Data and aesthetics	35
3.4.3	Geometries (~the type of plot)	36
3.5	Basic Statistics and Econometrics with <i>R</i>	42
3.5.1	Regression analysis with <i>R</i>	43
3.5.2	Linear model	45
3.5.3	Other econometric models and useful regression packages	46
3.5.4	Regression tables	47

List of Tables

3.1	49
-----	-------	----



List of Figures

1	Creative Commons License	ix
1.1	Running R in the Mac/OSX terminal.	3
1.2	Running R in the original R GUI/IDE.	4
1.3	Running R in RStudio (IDE).	4
1.4	Running R in the Mac/OSX terminal.	5
1.5	The R Script window in RStudio.	6
1.6	The environment window in RStudio.	6
1.7	The file browser window in RStudio.	7
2.1	For-loop illustration.	14
2.2	While-loop illustration.	16



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

Why learn to program (now)?

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 with many benefits for academic work in graduate economics. Many of the courses in your curriculum will at least partially touch upon

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

practical programming exercises or concepts related to programming and scientific computing in an environment like R.

1.1 Why R?

1.1.1 The ‘data language’

The programming language and open-source statistical computing environment *R*² 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.

1.1.2 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.

1.1.3 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 actually ‘write programs’ (in the traditional sense of the word), but apply *R* packages. Applied econometrics with *R* is a good example of this.

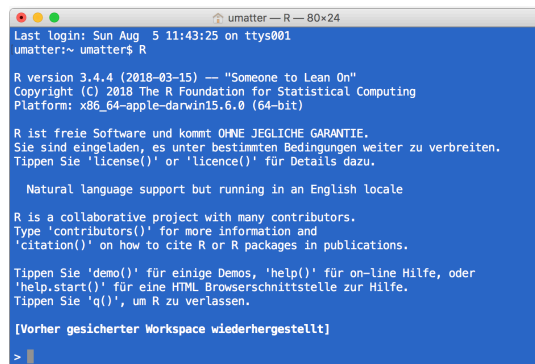
²www.r-project.org

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.

1.2 The tools: R/RStudio

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
Last login: Sun Aug  5 11:43:25 on ttys001
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 1.1: Running R in the Mac/OSX terminal.

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

³<http://www.stata.com/>

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

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

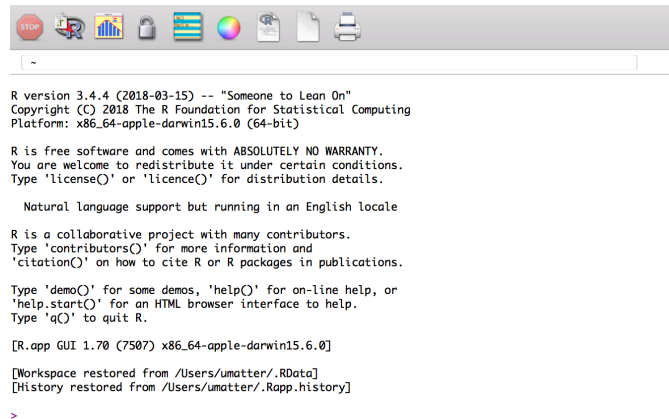


FIGURE 1.2: Running R in the original R GUI/IDE.

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

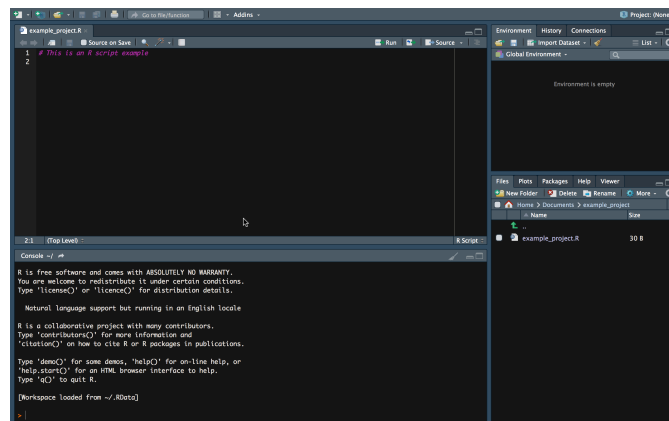


FIGURE 1.3: Running R in RStudio (IDE).

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),

⁶<https://rstudio.cloud/>

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.

1.2.1 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 command 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.

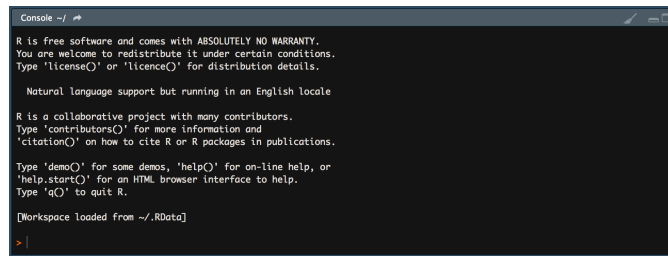


FIGURE 1.4: Running *R* in the Mac/OSX terminal.

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

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

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

1.2.2 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.

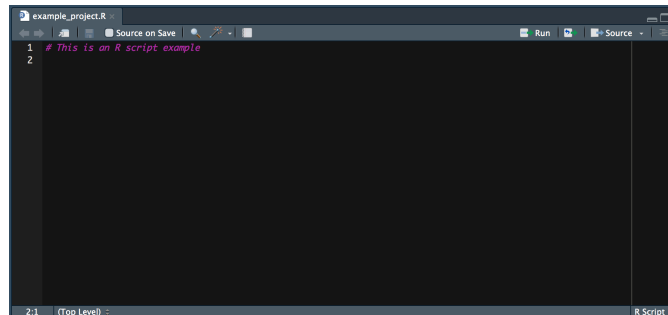


FIGURE 1.5: The R Script window in RStudio.

1.2.3 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.

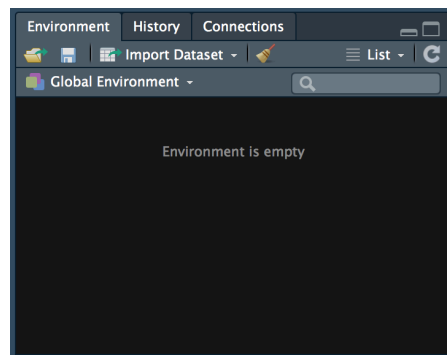


FIGURE 1.6: The environment window in RStudio.

1.2.4 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.

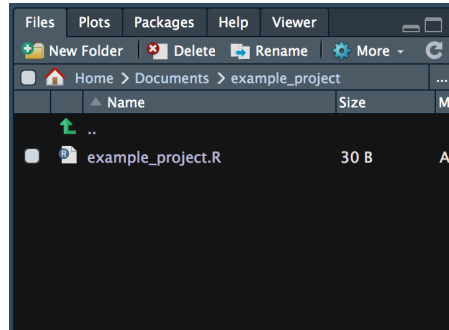


FIGURE 1.7: The file browser window in RStudio.

1.3 Exercises

1.3.1 Exercise A: Setting up a Working Environment

1. Open RStudio and get familiar with the file browser pane on the lower right. Navigate to a folder on your hard disk in which you want to work throughout this course (and store all the code you write in this course).
2. Use the 'New Folder'-button to create a new folder. Name this new folder `r_course`.
3. You should see the new folder listed in the file browser. Click on it to navigate to its contents (so far empty). Now, click on the 'More' button and select 'Set as Working Directory' in the drop-down menu.
4. Again, use the 'New Folder'-button in order to create two new folders called `data` and `code`.
5. Finally, click on the project button in the top-right corner of the RStudio window (Project: (None)) and select 'New Project' in the drop-down menu. In the pop-up window, select 'Existing directory', browse to and select your `r_course` folder, then click 'Create Project'.

Now you know how to set up a meaningful basic folder structure and


working environment for an R project. The next exercise teaches you how to write R scripts in this environment.

1.3.2 Exercise B: R Scripts

1. Switch to the R console and type the following line of code and hit enter (see example from above).

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

You should see the words "Hello world" printed on screen. This is the usual way of working with R in an interactive session. However, as pointed out above, in most circumstances it makes sense to write the R code to an R script (in order to store and document it) and then execute the code from there.

2. In the RStudio menu bar select File/New File/R Script to create a new file, shown/opened in the Script pane.
3. Type `print("Hello world")` to the first line of the script, and click on 'Run' ( Run) to execute the code in the console.
4. Save the file as `hello_world.R` (File/Save As...) in the sub folder `code` (created in the previous exercise).
5. Type the following command into the command line and hit enter:

```
source("code/hello_world.R")
```

```
## [1] "Hello World!"
```

Now you know how to execute R code directly in the console (interactive session), how to execute lines of code written to an R script, as well as how to execute the entire R script (stored on disk) from the command line.

2

First steps with R

Before introducing some of the key functions and packages for 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. In fact, since R is an open source environment, you can directly look at the source code of R packages in order to learn how they work.

2.1 Variables and Vectors

```
# 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

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

## Claire
##    33
```

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

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

```
# Inspect the object you are working with  
class(a) # returns the class(es) of the object
```

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

```
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" ...
```

2.2 Math Operators

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
```

```
# arithmetic 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
```

```
# special numbers  
# Euler's number  
exp(1)
```

```
## [1] 2.718
```

```
# Pi  
pi
```

```
## [1] 3.142
```

2.3 Basic programming concepts in R

2.3.1 Loops

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

2.3.1.1 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 following flow-chart illustrates the concept.

For example, a for-loop could be used to sum up each of the elements in a numeric vector of fix 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 element i of the n elements in the vector add the element’s value 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

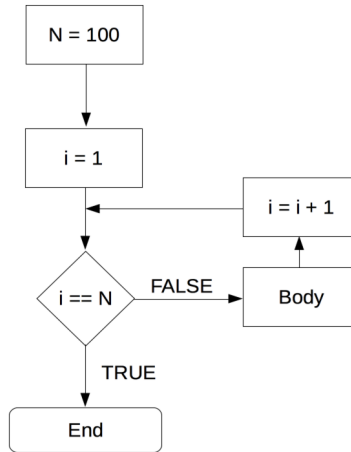


FIGURE 2.1: For-loop illustration.

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 result of sum() function
sum(numbers)

```



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

2.3.1.2 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 outer loop
m <- ncol(numbers_matrix)
# number of iterations for inner loop
n <- nrow(numbers_matrix)
# start outer loop (loop over columns of 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
```

2.3.1.3 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 in order 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 2.2 illustrates this point.

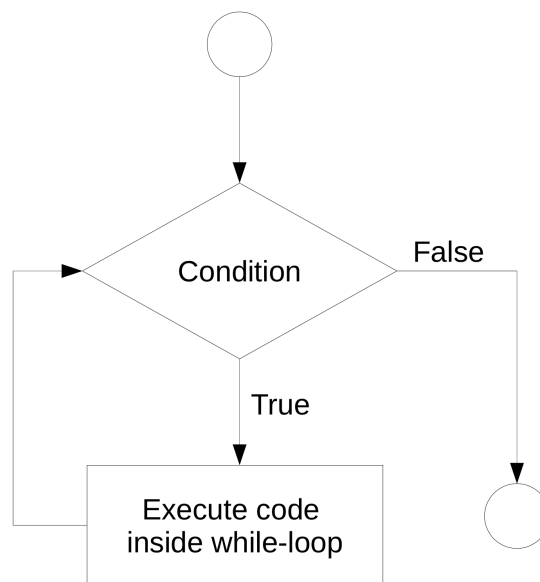


FIGURE 2.2: While-loop illustration.

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
```

2.3.2 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!"
```

2.3.3 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 have anything to do with numbers and arithmetic.¹ As such, functions facilitate the re-usage of blocks of code that take care of a certain task. Functions take ‘parameter values’ as input, process those values and ‘return’ the result.

¹Of course, on the very low level, everything that happens in a microprocessor can in the end be expressed in some formal way using math (and involves binary numbers). 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 kind of purposes, including the parsing of HTML code, etc.

When we open RStudio, all basic R 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 kind 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.

2.3.3.1 Compute the mean

For example, suppose we want to compute the mean for several variables. We do so by summing up the elements in a vector with `sum()` and dividing the returned sum by the number of elements in the vector, which we get with `length()`.

```
# own implementation: use R-function for summing up the elements in a vector  
# and getting the number of elements in a vector  
sum(a) / length(a)
```

```
## [1] 3.667
```

Instead of repeatedly writing the above line of code in a script, we can once define a function called `my_mean()` and then simply call this function to get the same result.

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

```
## [1] 3.667
```

```
# compare the result with the function delivered with the R installation
mean(a)

## [1] 3.667
```

2.4 Data Structures and Indices

R provides different object classes to work with data. In simple terms, they differ with regard to how data is structured when stored in the R environment. This, in turn, defines how we can access specific data values. For example, if we store a bunch of numbers in a numeric vector, we can access each of these numbers individually, by telling R to give us the first, second, n th element of this vector. If we store data in a two-dimensional object such as a matrix, we can access the data column-wise, row-wise, or cell-wise. Depending on the task, one or the other object class is preferable to store and work with data.

2.4.1 Vectors and Lists

2.4.1.1 Vectors

A vector containing integer (or numeric) values:

```
# integer
integer_vector <- 10:20
integer_vector[2]
```

```
## [1] 11
```

```
integer_vector[2:5]
```

```
## [1] 11 12 13 14
```

```
# numeric
numeric_vector <- c(1.4, 5.6, 7.2)
numeric_vector[1]
```

```
## [1] 1.4
```

```
numeric_vector[1:2]
```

```
## [1] 1.4 5.6
```

A string/character vector ('a vector containing text'):

```
# character vector
string_vector <- c("a", "b", "c")
string_vector[-3]
```

```
## [1] "a" "b"
```

2.4.2 Lists

A list can contain objects of differing data types and differing length in its elements, for example a numeric vector and a string_vector:

```
# initiate a list containing vectors
mylist <- list(numbers = numeric_vector, letters = string_vector)
mylist
```

```
## $numbers
## [1] 1.4 5.6 7.2
##
## $letters
## [1] "a" "b" "c"
```

We can access the elements of a list in various ways

```
# with the element's name  
mylist$numbers
```

```
## [1] 1.4 5.6 7.2
```

```
mylist["numbers"]
```

```
## $numbers  
## [1] 1.4 5.6 7.2
```

```
# via the index  
mylist[1]
```

```
## $numbers  
## [1] 1.4 5.6 7.2
```

With `[]` we can access directly the content of the element

```
mylist[[1]]
```

```
## [1] 1.4 5.6 7.2
```

Lists can also be nested (list of lists of lists....)

```
mynestedlist <- list(a = mylist, b = 1:5)  
mynestedlist
```

```
## $a  
## $a$numbers  
## [1] 1.4 5.6 7.2  
##  
## $a$letters  
## [1] "a" "b" "c"  
##  
##  
## $b
```



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

2.4.3 Matrices and Data Frames

When working on a data analysis task in R we typically deal with data in a two-dimensional ('table-like') format with observations in rows and variables describing these observations in columns. If all variables in a data set are of the same type we can store it in a `matrix`. In case the variables of a data set are of differing types, we typically store it in a `data.frame`.

2.4.3.1 Matrices and Arrays

Matrices (or more generally, 'arrays') are essentially vectors with an additional dimensionality definition.

```
# initiate an integer vector
integer_vector <- 1:8
# initiate a matrix based on this vector
mymatrix <- matrix(integer_vector, nrow = 4)
```

The indexing of a matrix is very similar to vectors. However, within the `[]`-expression, we indicate the dimension we want to access with a comma.

```
# return the second row
mymatrix[2,]
```

```
## [1] 2 6
```

```
# return the first two columns
mymatrix[,1:2]
```

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

```
# return the cell in the second row of the first column  
mymatrix[2,1]
```

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

2.4.3.2 Data-Frames

Data frames are the most common objects to store/work with data for analytics purposes in R. Most functions to import/load data into the R environment return the data in the form of a `data.frame`. Technically they are similar to a list with elements of identical length.

```
# data frames ("lists as columns")  
mydf <- data.frame(Name = c("Alice", "Betty", "Claire"), Age = c(20, 30, 45))  
mydf
```

```
##      Name Age  
## 1  Alice  20  
## 2  Betty  30  
## 3 Claire  45
```

We can access elements of the `data.frame` in various ways

```
# select the age column  
mydf$Age
```

```
## [1] 20 30 45
```

```
mydf[, "Age"]
```

```
## [1] 20 30 45
```

```
mydf[, 2]
```

```
## [1] 20 30 45
```

```
# select the second row  
mydf[2,]
```

```
##      Name Age  
## 2 Betty  30
```

2.4.4 Classes and Data Structures

When importing data to R, using a new R-package, or extracting data from web sources, it is imperative to understand how the data is structured in the R-object (variable) one is working with. A lot of errors occur due to giving R functions input in the wrong format/structure. Having a close look at how the data is stored in a variable that we want to work with further can help avoid such errors.² The function `class()` returns the class(es) of an R object:

```
# have a look at the class of the object  
class(mydf)
```

```
## [1] "data.frame"
```

```
class(mymatrix)
```

```
## [1] "matrix" "array"
```

To get a clear idea of the structure of data stored in a specific variable, we can call the function `str()` (for structure) with the object of interest as the only function argument. This is particularly helpful when dealing with a complex/nested data structure.

²Note that basic R functions as well as functions distributed via R packages are usually very well documented (look up documentation with `help(FUNCTION-NAME)` or `?FUNCTION-NAME`. An important part of such documentations is what class the input arguments of a function must have and what the class of the returned object is. See, for example the documentation of `mean()` by typing `?mean` into the R-console and hit enter.

```
# have a closer look at the data structure  
str(mydf)
```

```
## 'data.frame':  3 obs. of  2 variables:  
## $ Name: chr  "Alice" "Betty" "Claire"  
## $ Age : num  20 30 45
```

2.5 Exercises

2.5.1 Exercise A: Write a Sum Function

In the code example above (introducing the for-loop) you see how we can use for loops to sum up numbers stored in a numeric vector. Use this code example and the following empty function construct to implement a function that takes a numeric vector as input and returns the sum of the vector's elements.

```
my_sum <-  
  function(x){  
  
  }
```

Test your function by comparing its output with the output of `sum()` (the pre-implemented sum function in R) when calling it with the same input.

2.5.2 Exercise B: Robustness and Warnings

Once you have implemented and successfully tested `my_sum()`, see what happens if you call it with the following vector as input

```
numbers2 <- c("1", "2", "3")
```

Why do we get an error? Investigate by comparing the vector `numbers`

from the code example above with the new vector `numbers2` (the functions `str()` and `class()` might help you).

Extend your `my_sum()` function with a control statement that checks whether the input is of the right class for the function to work (Hint: `class(x)!="numeric"`). The goal is to make the function more robust to wrong input. That is, if the input is wrong, issue a warning but don't break down with an error. Use the following code for the warning message:

```
warning("Wrong input! This function only accepts numeric or integer values!")
```

Test your enhanced sum function with the two input vectors `numbers` and `numbers2`.

2.5.3 Exercise C: Standard Deviation Function

In the example above we've implemented our own R function to compute the mean, given a numeric vector with the help of already implemented R functions (`length()` and `sum()`; alternatively use your own `my_sum()`). Now implement an R function called `my_sd` that computes the standard deviation $SD = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2}$. (Hint: see the math operators section above for some of the ingredients).

Test your implementation by comparing its results with the already implemented `sd()` function.

2.5.4 Exercise D: Standard Error Function

Building on the function written in Exercise C, implement a function to compute (the estimate of) the sample mean standard error as defined here: $SE_{\bar{x}} = \frac{SD}{\sqrt{n}}$. Hint: the idea is to use your own implementation (in Exercise C) to compute SD as a first step of your standard error function.

Exercises C and D should illustrate how you can approach more complex problems by breaking them up into smaller problems and addressing each of these step-by-step. That is, when looking at the problem of

computing $SE_{\bar{x}} = \frac{SD}{\sqrt{n}}$, you see that computing SD is a part of the overall problem, which can be addressed separately.

2.5.5 Exercise E: T-test

Implement a function for the one-sample t-test³: $t = \frac{\bar{x} - \mu_0}{\frac{SE}{\sqrt{n}}}$.

³https://en.wikipedia.org/wiki/Student%27s_t-test

3

Working with Data

3.1 Basics Loading/Importing Data

There are many ways to import data into R from various sources. Here, we look at how to import data from the most common sources when you use R for basic data analysis or econometric exercises.

3.2 Loading built-in datasets

The basic R installation provides some example data sets to try out R's statistics functions. In the introduction to visualization techniques with R as well as in the statistics examples further below, we will rely on some of these datasets for sake of 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 online tutorials also use these datasets (see, for example the ggplot2 documentation¹). And, they are very useful when searching help on Stackoverflow² in the context of data analysis/manipulation with R.³

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

¹<https://ggplot2.tidyverse.org/>

²<https://stackoverflow.com/questions/tagged/r>

³Properly posting a question on Stackoverflow requires the provision of a code example based on data that everybody can easily load and access.

```
data(swiss)
```

In this case, we load a dataset called `swiss`. After loading it, the data is 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 simply type `data()` into the con-

sole and hit enter. To get more information about a given dataset use the help function (for example, `?swiss`)

3.2.1 Importing data from files

In most cases of applying R for econometrics and 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 (e.g., for your Master's Thesis) on your own, you might rely on a spreadsheet tool like Microsoft Excel. Data from all of these formats can quite easily be imported into R (in some cases, additional packages have to be loaded, though).

3.2.1.1 Comma Separated Values (CSV)

A most common format to store and transfer datasets is CSV. 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:

```
","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")
```

3.2.1.2 Spreadsheets/Excel

In order 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`.

```
# load the package
library(readxl)

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

3.2.1.3 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")
```

3.3 Data Visualization with R (*ggplot2*)

3.3.1 ‘Grammar of Graphics’

A few years back, Leland Wilkinson (statistician and computer scientist) wrote an influential book called ‘The Grammar of Graphics’. In the 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. A key point of this grammar is that we can think of graphics as consisting of different design-layers and that we can build and describe them layer by layer (see here⁴ for an illustration of this idea).

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

3.4 *ggplot2* basics

Using *ggplot2* to generate a basic plot in R is quite simple. Basically, it involves three key points:

1. The data must be stored in a `data.frame`(preferably in ‘long’ 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’ (what variables are mapped to the x-/y-axes):

⁴<http://bloggotype.blogspot.ch/2016/08/holiday-notes2-grammar-of-graphics.html>

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

3.4.1 Tutorial

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

3.4.1.1 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
```

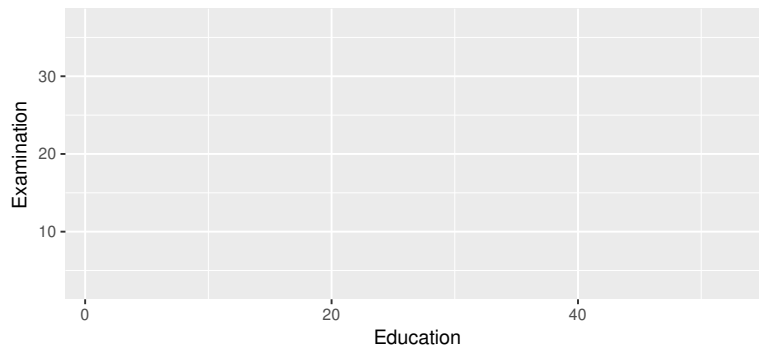
As we do not only want to use this continuous measure in the data visualization, we generate an additional factor variable called `Religion` which has either the value `'Protestant'` or `'Catholic'` depending on whether more than 50 percent of the inhabitants of the province are Catholics.

```
# 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)
```

3.4.2 Data and aesthetics

We initiate the most basic plot with `ggplot()` by defining which data we want to use and in the plot aesthetics which variable we want 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 with 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))
```

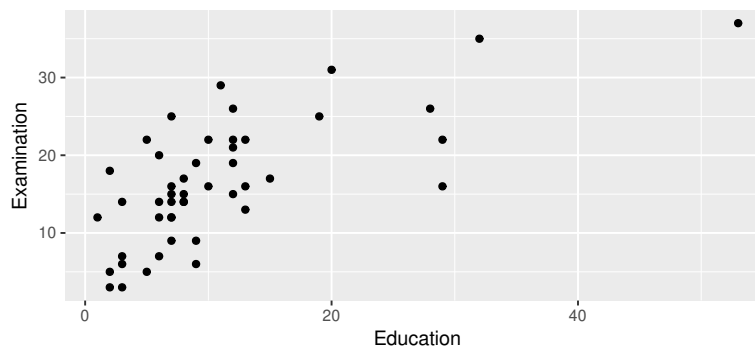


As we have not yet defined according to what rules the data shall be visualized, all we get is an empty 'canvas' and the axes (with the respective label and ticks indicating the range of the values).

3.4.3 Geometries (~the type of plot)

To actually plot the data we have to define the ‘geometries’, defining 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()
```

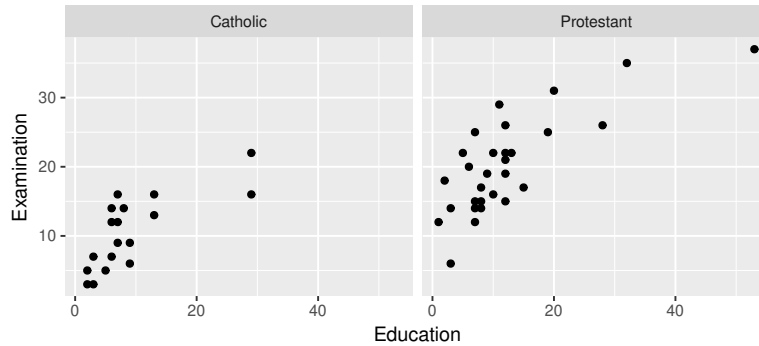


The result indicates that there is a positive correlation between the level of education and how well draftees do in the examination. We want to better understand this correlation. Particularly, what other factors could drive this picture.

3.4.3.1 Facets

According to a popular thesis, the protestant reformation and the spread of the protestant movement in Europe was 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. In order 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)
```



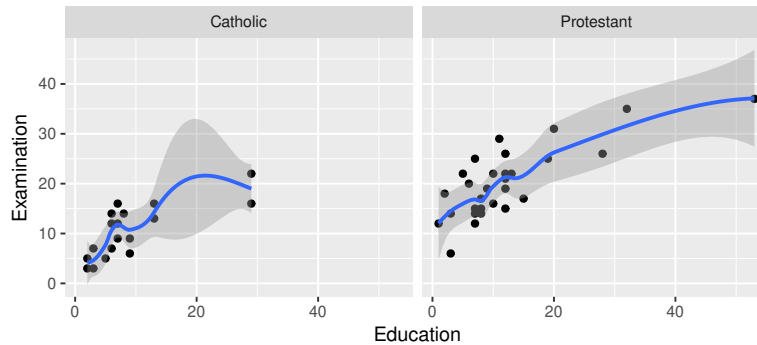
Draftees from protestant districts tend to do generally better (which might be an indication of better primary schools, or a generally stronger focus on scholastic achievements of Protestant children). However, the relationship between education (beyond primary schools) and examination success seems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

3.4.3.2 Additional layers and statistics

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

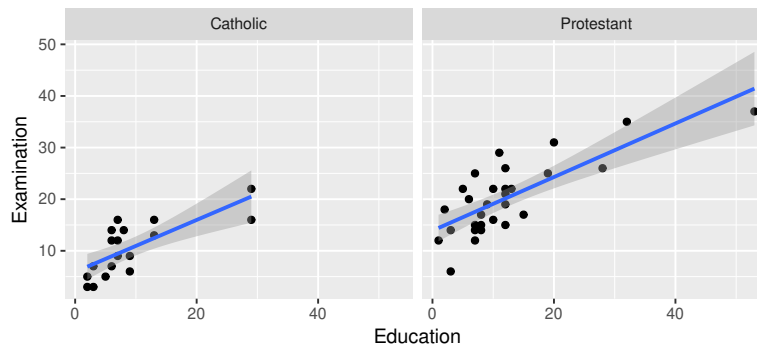
```
ggplot(data = swiss, aes(x = Education, y = Examination)) +
  geom_point() +
  geom_smooth(method = 'loess') +
  facet_wrap(~Religion)
```

```
## `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'



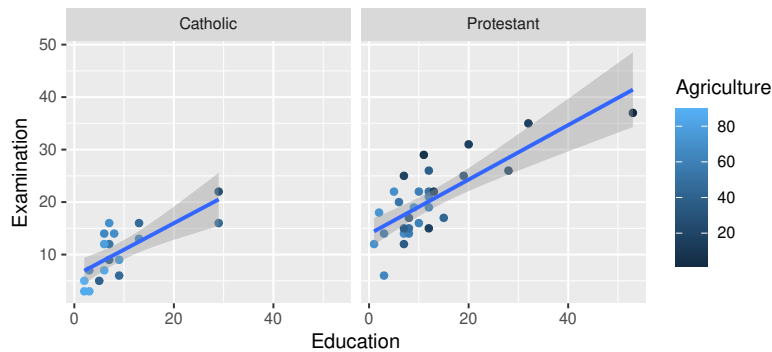
3.4.3.3 Additional aesthetics

Knowing a little bit about Swiss history and geography, we realize that particularly rural cantons in mountain regions remained Catholic during the reformation. In addition, cantonal school systems historically took into account 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 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 occupation in agriculture tend to have rather higher levels of education as well as higher achievements in the examination.

3.4.3.4 Coordinates

Finally, there are countless options to further refine the plot. 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'
```

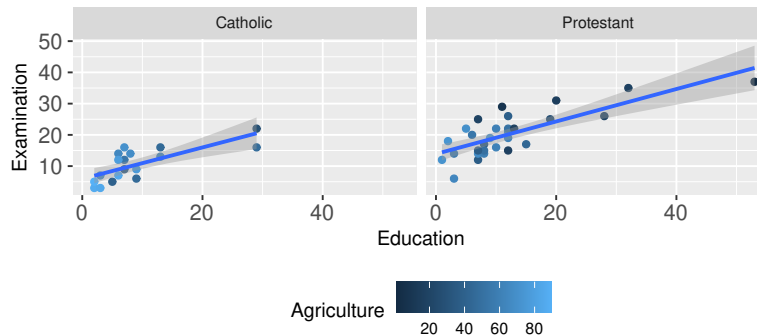


3.4.3.5 Themes: Fine-tuning the plot

In addition, the `theme()`-function allows to change 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) )
```

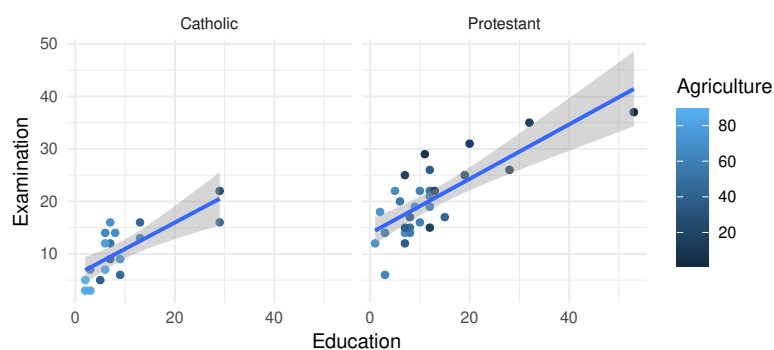
`geom_smooth()` using formula 'y ~ x'



Moreover, several theme-templates offer ready-made designs for plots:

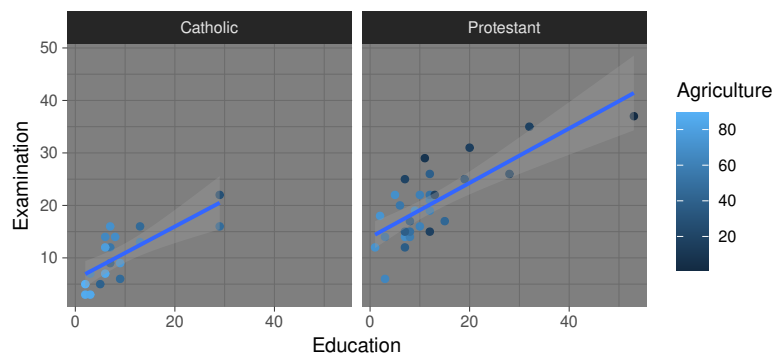
```
ggplot(data = swiss, aes(x = Education, y = Examination)) +
  geom_point(aes(color = Agriculture)) +
  geom_smooth(method = 'lm') +
  facet_wrap(~Religion) +
  theme_minimal()
```

```
## `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'
```



3.5 Basic Statistics and Econometrics with R

In the previous part we have learned how to implement basic statistics functions in R, as well as notices that for most of these statistics R already has a build-in function.

For example, mean and median:

```
# initiate sample
a <- c(10,22,33, 22, 40)
names(a) <- c("Andy", "Betty", "Claire", "Daniel", "Eva")

# compute the mean
mean(a)
```

```
## [1] 25.4
```

```
# compute the median
median(a)
```

```
## [1] 22
```

...as well as different measures of variability,

```
range(a)
```

```
## [1] 10 40
```

```
var(a)
```

```
## [1] 132.8
```

```
sd(a)
```

```
## [1] 11.52
```

and test statistics such as the t-test.

```
# define size of sample
n <- 100
# draw the random sample from a normal distribution with mean 10 and sd 2
sample <- rnorm(n, mean = 10, sd = 2)

# Test H0: mean of population = 10
t.test(sample, mu = 10)

##
## One Sample t-test
##
## data: sample
## t = 1.5, df = 99, p-value = 0.1
## alternative hypothesis: true mean is not equal to 10
## 95 percent confidence interval:
## 9.895 10.749
## sample estimates:
## mean of x
## 10.32
```

3.5.1 Regression analysis with R

Finally, R and additional R packages offer functions for almost all models and tests in modern applied econometrics. While the syntax and return values of these functions are not 100% consistent (due to the fact that many different authors contributed some of these functions), applying them is usually quite similar. Functions for econometric models typically have the two main parameters `data` (the dataset to use in the estimation, in `data.frame`-format) and `formula` (a symbolic description of the model to be fitted, following a defined convention). Typically, the following three steps are involved when running regressions in R (assuming the data is already loaded and prepared):

1. In `formula`, we define the model we want to fit as an object of

class ‘formula’. For example, we want to regress `Examination` on `Education` with the `swiss`-dataset used above.

```
modell1 <- Examination~Education
```

2. Fit the model with the respective estimator. Here we use `lm()` the R workhorse for linear models to fit the model with OLS:

```
fit1 <- lm(formula = modell1, data = swiss)
```

3. Get summary statistics with the generic `summary()`. In the case of an object of class `lm` (the output of `lm()`), this amounts to the typical coefficient t-statistics and p-values shown in regression tables:

```
summary(fit1)
```

```
##
## Call:
## lm(formula = modell1, 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
```

3.5.2 Linear model

Putting the pieces together, we can write a simple script to analyse the `swiss`-dataset (in spirit of the data visualization above):

```
# load data
data(swiss)

# linear regression with one variable
# estimate coefficients
model_fit <- lm(Examination~Education, data = swiss)
# t-tests of coefficients (and additional statistics)
summary(model_fit)
```

```
##
## 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
```

Specifying the linear model differently can be done by simply changing

the formula of the model. For example, we might want to control for the share of agricultural occupation.

```
# multiple linear regression
# estimate coefficients
model_fit2 <- lm(Examination~Education + Catholic + Agriculture, data = swiss)
# t-tests of coefficients (and additional statistics)
summary(model_fit2)
```

```
##
## Call:
## lm(formula = Examination ~ Education + Catholic + Agriculture,
##     data = swiss)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -9.315  -2.831  -0.349   3.349   7.417
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  18.5370     2.6371    7.03  1.2e-08 ***
## Education     0.4242     0.0868    4.89  1.5e-05 ***
## Catholic     -0.0798     0.0168   -4.75  2.3e-05 ***
## Agriculture  -0.0676     0.0396   -1.71   0.095 .
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.3 on 43 degrees of freedom
## Multiple R-squared:  0.728, Adjusted R-squared:  0.709
## F-statistic: 38.3 on 3 and 43 DF, p-value: 3.21e-12
```

3.5.3 Other econometric models and useful regression packages

Various additional packages provide all kind of functions to estimate diverse econometric models. Many generalized linear models can be fitted with the `glm()`-function (formula and data used as in `lm()`):

- Probit model: `glm(..., family = binomial(link = "probit"))`.
- Logit model: `glm(..., family = "binomial")`.
- Poisson regression (count data): `glm(..., family="poisson")`.

Other frequently used packages for econometric regression analysis:

- Panel data econometrics: package `plm`, `fixest`, `lfe`.
- Time-series econometrics: package `tseries`.
- Basic IV regression: `ivreg()` in package `AER`.
- Generalized methods of moments: package `gmm`.
- Generalized additive models: package `gam`.
- Lasso regression: package `glmnet`.
- Survival analysis: package `survival`.

See also the CRAN Econometrics Task View⁵.

3.5.4 Regression tables

Economists typically present regression results in detailed regression tables, with regression coefficients in rows and model specifications in columns. The package `stargazer` provides some functions to generate such tables very easily for outputs of the most common regression functions (`lm()`, `glm()`, etc.):

```
# load packages
library(stargazer)
```

```
# print regression results as text
stargazer(model_fit, model_fit2, type = "text")
```

```
##
## =====
##                               Dependent variable:
##                               -----
##                               Examination
##                               (1)                (2)
```

⁵<https://cran.r-project.org/web/views/Econometrics.html>

```
## -----
-
## Education          0.579***          0.424***
##                   (0.089)          (0.087)
##
## Catholic          -0.080***
##                   (0.017)
##
## Agriculture       -0.068*
##                   (0.040)
##
## Constant          10.130***          18.540***
##                   (1.286)          (2.637)
##
## -----
-
## Observations          47          47
## R2                    0.488          0.728
## Adjusted R2           0.476          0.709
## Residual Std. Error   5.773 (df = 45)    4.304 (df = 43)
## F Statistic           42.850*** (df = 1; 45) 38.350*** (df = 3; 43)
## =====
## Note:                  *p<0.1; **p<0.05; ***p<0.01
```

Alternatively the regression table can be directly generated as LaTeX table or as HTML table, which can be parsed and rendered for an html document or pdf document, when generated as part of an Rmd-file (such as this lecture notes).

```
# conditional on the output format of the whole document,
# generate and render a HTML or a LaTeX table.
if (knitr::is_latex_output()) {
  stargazer(model_fit, model_fit2, type = "latex",
            header = FALSE, table.placement = "H")
} else {
  stargazer(model_fit, model_fit2, type = "html")
}
```

TABLE 3.1

	<i>Dependent variable:</i>	
	Examination	
	(1)	(2)
Education	0.579*** (0.089)	0.424*** (0.087)
Catholic		−0.080*** (0.017)
Agriculture		−0.068* (0.040)
Constant	10.130*** (1.286)	18.540*** (2.637)
Observations	47	47
R ²	0.488	0.728
Adjusted R ²	0.476	0.709
Residual Std. Error	5.773 (df = 45)	4.304 (df = 43)
F Statistic	42.850*** (df = 1; 45)	38.350*** (df = 3; 43)
<i>Note:</i> *p<0.1; **p<0.05; ***p<0.01		



Bibliography

Einav, L. and Levin, J. (2014). Economics in the age of big data. *Science*, 346(6210):1243089–1–1243089–6.

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/>)).

