

Daniel Dauber

R for Non-Programmers: A Guide for Social Scientists



Table of contents

Welcome	ix
About the author	1
Acknowledgements	3
1 Readme. before you get started	5
1.1 A starting point and reference book	5
1.2 Download the companion R package	6
1.3 A ‘tidyverse’ approach with some basic R	6
1.4 Understanding the formatting of this book	7
2 Why learn a programming language as a non-programmer?	9
2.1 Learning new tools to analyse your data is always essential	10
2.2 Programming languages enhance your conceptual thinking	10
2.3 Programming languages allow you to look at your data from a different angle	11
2.4 Learning any programming language will help you learn other programming languages.	11
3 Setting up <i>R</i> and RStudio	13
3.1 Installing R	13
3.2 Installing RStudio	16
3.3 When you first start RStudio	18
3.4 Updating R and RStudio: Living at the pulse of innovation	20
3.5 RStudio Cloud	21
4 The RStudio Interface	25
4.1 The Console window	25
4.2 The Source window	26
4.3 The Environment / History / Connections / Tutorial window .	27
4.4 The Files / Plots / Packages / Help / Viewer window	30
4.5 Customise your user interface	35
5 <i>R</i> Basics: The very fundamentals	37
5.1 Basic computations in <i>R</i>	37
5.2 Assigning values to objects: ‘<-’	40

5.3	Functions	45
5.4	R packages	48
5.4.1	Installing packages using <code>install.packages()</code>	48
5.4.2	Installing packages via RStudio's package pane	50
5.4.3	Install all necessary <i>R</i> packages for this book	52
5.4.4	Using <i>R</i> Packages	52
5.5	Coding etiquette	53
5.6	Exercises	56
6	Starting your <i>R</i> projects	57
6.1	Creating an <i>R</i> Project file	57
6.2	Organising your projects	61
6.3	Creating an <i>R</i> Script	62
6.4	Using <i>Quarto</i>	66
7	Data Wrangling	69
7.1	Import your data	70
7.1.1	Import data from the <i>Files</i> pane	70
7.1.2	Importing data from the <i>Environment</i> pane	73
7.1.3	Importing data using functions directly	73
7.2	Inspecting your data	75
7.3	Cleaning your column names: Call the <code>janitor</code>	78
7.4	Data types: What are they and how can you change them	80
7.5	Handling factors	84
7.5.1	Recoding factors	85
7.5.2	Reordering factor levels	88
7.5.3	Removing factor levels	92
7.6	Handling dates, times and durations	93
7.6.1	Converting dates and times for analysis	94
7.6.2	Computing durations with dates, times and date-time variables	98
7.7	Dealing with missing data	99
7.7.1	Mapping missing data	100
7.7.2	Identifying patterns of missing data	102
7.7.3	Replacing or removing missing data	106
7.7.4	Two more methods of replacing missing data you hardly ever need	112
7.7.5	Main takeaways regarding dealing with missing data	115
7.8	Latent constructs and their reliability	115
7.8.1	Computing latent variables	116
7.8.2	Checking internal consistency of items measuring a latent variable	118
7.8.3	Confirmatory factor analysis	122
7.8.4	Reversing items with opposite meaning	125
7.9	Once you finished with data wrangling	132

7.10 Exercises	133
8 Descriptive Statistics	135
8.1 Plotting in R with <code>ggplot2</code>	135
8.2 Frequencies and relative frequencies: Counting categorical variables	143
8.3 Central tendency measures: Mean, Median, Mode	145
8.3.1 Mean	146
8.3.2 Median	150
8.3.3 Mode	154
8.4 Indicators and visualisations to examine the spread of data	156
8.4.1 Boxplot: So much information in just one box	156
8.4.2 Histogram: Do not mistake it as a bar plot	158
8.4.3 Density plots: Your smooth histograms	163
8.4.4 Violin plot: Your smooth boxplot	166
8.4.5 QQ plot: A ‘cute’ plot to check for normality in your data	170
8.4.6 Standard deviation: Your average deviation from the mean	171
8.5 Packages to compute descriptive statistics	174
8.5.1 The <code>psych</code> package for descriptive statistics	175
8.5.2 The <code>skimr</code> package for descriptive statistics	176
9 Sources of bias: Sampling, outliers, normality and other ‘conundrums’	177
9.1 The origins of biases: A short introduction to sampling theory .	178
9.2 Linearity and additivity	186
9.3 Independence	188
9.4 Normality	188
9.5 Homogeneity of variance (homoscedasticity)	193
9.6 Outliers and how to deal with them	196
9.6.1 Detecting outliers using the standard deviation	196
9.6.2 Detecting outliers using the interquartile range (IQR) .	202
9.6.3 Removing or replacing outliers	204
9.6.4 Concluding remarks about outliers	207
10 Correlations	211
10.1 Plotting correlations	212
10.2 Computing correlations	215
10.3 Significance: A way to help you judge your findings	218
10.4 Limitations of correlations	223
10.4.1 Correlations are not causal relationships	223
10.4.2 Correlations can be spurious	223
10.4.3 Simpson’s Paradox: When correlations betray you . .	226
11 Comparing groups	231

11.1 Comparability: Apples vs Oranges	232
11.2 Comparing two groups	234
11.2.1 Two unpaired groups	235
11.2.2 Two paired groups	240
11.3 Comparing more than two groups	245
11.3.1 Multiple unpaired groups	245
11.3.2 Multiple paired groups	251
11.4 Comparing groups based on factors: Contingency tables	260
11.4.1 Unpaired groups of categorical variables	263
11.4.2 Paired groups of categorical variables	272
11.4.3 A final remark about comparing groups based on categorical data	278
11.5 Reviewing group comparisons	280
12 Power: You either have it or you don't	281
12.1 Ingredients to achieve the power you deserve	282
12.2 Computing power	283
12.3 Plotting power	286
12.4 Concluding remarks about power analysis	288
13 Regression: Creating models to predict future observations	291
13.1 Single linear regression	292
13.1.1 Fitting a regression model by hand, i.e. trial and error .	294
13.1.2 Fitting a regression model computationally	299
13.2 Multiple regression	303
13.2.1 Outliers in multiple regressions	308
13.2.2 Standardised beta (β) vs. unstandardised beta (B) . .	332
13.2.3 Multicollinearity: The dilemma of highly correlated independent variables	339
13.3 Hierarchical regression	342
13.3.1 Regressions with control variables	343
13.3.2 Moderated regression	348
13.3.3 Centering predictors: Making β s more interpretable .	352
13.4 Other regression models: Alternatives to OLS	356
14 Mixed-methods research: Analysing qualitative data in <i>R</i>	359
14.1 The tidy process of working with textual data	359
14.2 Stop words: Removing noise in the data	362
14.3 N-grams: Exploring correlations of words	374
14.4 Exploring more mixed-methods research approaches in <i>R</i> . .	384
15 Where to go from here: The next steps in your <i>R</i> journey	387
15.1 GitHub: A Gateway to even more ingenious <i>R</i> packages . .	387
15.2 Books to read and expand your knowledge	388
15.3 Engage in regular online readings about <i>R</i>	389
15.4 Join the X/Twitter community and hone your skills	390

<i>Contents</i>	vii
Epilogue	393
Appendix	395
Comparing two unpaired groups	395
Comparing two paired groups	395
Comparing multiple unpaired groups	396
References	397



Welcome

Welcome to *R for Non-Programmers: A Guide for Social Scientists (R4NP)*. This book provides a helpful resource for anyone looking to use *R* for their research projects, regardless of their level of programming or statistical analysis experience. Each chapter presents essential concepts in data analysis in a concise, thorough, and user-friendly manner. *R4NP* will guide you through your first steps on a possibly endless journey full of ‘awe and wonder’.

This book is designed to be accessible to beginners while also providing valuable insights for experienced analysts looking to transition to *R* from other computational software. You don’t need any prior knowledge or programming skills to get started. *R4NP* provides a comprehensive entry into R programming, regardless of your background.



About the author

I am a Reader¹ at the University of Warwick. My research interfaces with multiple disciplines, such as International Management, Organisational Behaviour, Cross-cultural Psychology and Intercultural Communication. Most of my research investigates real-life issues in organisational contexts using a diagnostic angle with an emphasis to help improve organisational members' experiences and organisational performance.

For more than 15 years, I have enjoyed teaching students how to turn data into meaningful recommendations for businesses, universities and the broader society by conveying relevant theory-driven knowledge as well as research methods skills. In addition, I am enthusiastic about new approaches to analysing data, whether it is quantitative or qualitative.

My most recent publications can be found on ResearchGate², and I regularly post tutorials about R, productivity and other hidden gems of academic life on my blog: The Social Science Sofa³. If you like to connect with me, you can find me on X/Twitter⁴.

¹A Reader is a senior academic position, a rank between Associate Professor and Full Professor.

²<https://www.researchgate.net/profile/Daniel-Dauber>

³<https://www.thesocialsciencesofa.com>

⁴https://twitter.com/daniel_dauber



Acknowledgements

Special thanks are due to my wife, who supported me in so many ways to get this book completed. Also, I would like to thank my son and daughter, who patiently watched me sitting at the computer typing this book and encouraged me with hugs, smiles and kisses. Finally, my mother-in-law, deserves special thanks because she ensured I would not starve while passionately completing this mammoth project.



1

Readme. before you get started

1.1 A starting point and reference book

My journey with *R* began rather suddenly one night. After many months of contemplating learning *R* and too many excuses not to get started, I put all logic aside and decided to use *R* on my most significant project to date. Admittedly, at the time, I had some knowledge of other programming languages and felt maybe overconfident learning a new tool. This is certainly not how I would recommend learning *R*. In hindsight, though, it enabled me to do things that pushed the project much further than I could have imagined. However, I invested a lot of additional time on top of my project responsibilities to learn this programming language. In many ways, *R* opened the door to research I would not have thought of a year earlier. Today, I completely changed my style of conducting research, collaborating with others, composing my blog posts and writing my research papers.

It is true what everyone told me back then: *R* programming has a steep learning curve and is challenging. To this date, I would agree with this sentiment if I ignored all the available resources. The one thing I wish I had to help me get started was a book dedicated to analysing data from a Social Scientist perspective and which guides me through the analytical steps I partially knew already. Instead, I spent hours searching on different blogs and YouTube to find solutions to problems in my code. However, at no time I was tempted to revert to my trusted statistics software of choice, i.e. SPSS. I certainly am an enthusiast of learning programming languages, and I do not expect that this is true for everyone. Nevertheless, the field of Social Sciences is advancing rapidly and learning at least one programming language can take you much further than you think.

Thus, the aim of this book is narrowly defined: A springboard into the world of *R* without having to become a full-fledged *R* programmer or possess abundant knowledge in other programming languages. This book will guide you through the most common challenges in empirical research in the Social Sciences. Each chapter is dedicated to a common task we have to achieve to answer our research questions. At the end of each chapter, exercises are provided to hone your skills and allow you to revisit key aspects.

This book likely caters to your needs irrespective of whether you are a seasoned analyst and want to learn a new tool or have barely any knowledge about data analysis. However, it would be wrong to assume that the book covers everything you possibly could know about *R* or the analytical techniques covered. There are dedicated resources available to you to dig deeper. Several of these resources are cited in this book or are covered in Chapter 15. The primary focal point of the book is on learning *R* in the context of Social Sciences research projects. As such, it serves as a starting point on what hopefully becomes an enriching, enjoyable, adventurous, and lasting journey.

1.2 Download the companion R package

The learning approach of this book is twofold: Convey knowledge and offer opportunities to practice. Therefore, more than 50% of the book are dedicated to examples, case studies, exercises and code you can directly use yourself. To facilitate this interactive part, this book is accompanied by a so-called *R package* (see also Section 5.4), which contains all datasets used in this book and enables you to copy and paste any code snippet¹ and work along with the book.

Once you have worked through Chapter 3 you can easily download the package `r4np` using the following code snippet in your console (see Section 4.1):

```
devtools::install_github("ddrauber/r4np")
```

1.3 A ‘tidyverse’ approach with some basic R

As you likely know, every language has its flavours in the form of dialects. This is no different to programming languages. The chosen ‘dialect’ of *R* in this book is the `tidyverse` approach. Not only is it a modern way of programming in *R*, but it is also a more accessible entry point. The code written with `tidyverse` reads almost like a regular sentence and, therefore, is much easier to read, understand, and remember. Unfortunately, if you want a comprehensive introduction to learning the underlying basic *R* terminology, you will have to consult other books. While it is worthwhile to learn different ways of conducting research in *R*, basic *R* syntax is much harder to learn, and I opted against covering it as an entry point for novice users. After working through this book,

¹A *code snippet* is a piece of programming code that can be used directly as is.

you will find exploring some of the *R* functions from other ‘dialects’ relatively easy, but you likely miss the ease of use from the `tidyverse` approach.

1.4 Understanding the formatting of this book

The formatting of this book carries special meaning. For example, you will find actual *R* code in boxes like these.

```
name <- "Daniel"
food <- "Apfelstrudel"

paste("My name is ", name, ", and I love ", food, ".", sep = "")
```

[1] "My name is Daniel, and I love Apfelstrudel."

You can easily copy this *code chunk* by using the button in the top-right corner. Of course, you are welcome to write the code from scratch, which I would recommend because it accelerates your learning.

Besides these blocks of code, you sometimes find that certain words are formatted in a particular way. For example, datasets, like `halloween`, included in the *R* package `r4np`, are highlighted. Every time you find a highlighted word, it refers to one of the following:

- A dataset,
- A variable,
- A data type,
- The output of code,
- The name of an *R* package,
- The name of a function or one of its components.

This formatting style is consistent with other books and resources on *R* and, therefore, easy to recognise when consulting other content, such as those covered in Chapter 15.



2

Why learn a programming language as a non-programmer?

‘R’, it is not just a letter you learn in primary school, but a powerful programming language. While it is used for a lot of quantitative data analysis, it has grown over the years to become a powerful tool that excels (#no-pun-intended) in handling data and performing customised computations with quantitative and qualitative data.

R is now one of my core tools to perform various types of work, for example,

- Statistical analysis,
 - Corpus analysis,
 - Development of online dashboards for interactive data visualisations and exploration,
 - Connection to social media APIs for data collection,
 - Creation of reporting systems to provide individualised feedback to research participants,
 - Writing research articles, books, and blog posts,
 - etc.
-

Learning R is like learning a foreign language. If you enjoy learning languages, then ‘R’ is just another one.

While *R* has become a comprehensive tool for data scientists, it has yet to find its way into the mainstream field of Social Sciences. Why? Well, learning programming languages is not necessarily something that feels comfortable to everyone. It is not like Microsoft Word, where you can open the software and explore it through trial and error. Learning a programming language is like learning a foreign language: You have to learn vocabulary, grammar and syntax. Similar to learning a new language, programming languages also have steep learning curves and require quite some commitment.

For this reason, most people do not even dare to learn it because it is time-consuming and often not considered a ‘*core method*’ in Social Sciences disciplines. Apart from that, tools like SPSS have very intuitive interfaces, which seem much easier to use (or not?). However, the feeling of having ‘*mastered*’ *R* (although one might never be able to claim this) can be extremely rewarding.

I guess this introduction was not necessarily helpful in convincing you to learn any programming language. However, despite those initial hurdles, there are a series of advantages to consider. Below I list some good reasons to learn a programming language as they pertain to my own experiences.

2.1 Learning new tools to analyse your data is always essential

Theories change over time, and new insights into certain social phenomena are published every day. Thus, your knowledge might get outdated quite quickly. This is not so much the case for research methods knowledge. Typically, analytical techniques remain over many years. We still use the mean, mode, quartiles, standard deviation, etc., to describe our quantitative data. However, there are always new computational methods that help us to crunch the numbers even more. *R* is a tool that allows you to venture into new analytical territory because it is open source. Thousands of developers provide cutting-edge research methods free of charge for you to try with your data. You can find them on platforms like GitHub¹. *R* is like a giant supermarket, where all products are available for free. However, to read the labels on the product packaging and understand what they are, you have to learn the language used in this supermarket.

2.2 Programming languages enhance your conceptual thinking

While I have no empirical evidence for this, I am very certain it is true. I would argue that my conceptual thinking is quite good, but I would not necessarily say that I was born with it. Programming languages are very logical. Any error in your code will make you fail to execute it properly. Sometimes you face challenges in creating the correct code to solve a problem. Through creative abstract thinking (I should copyright this term), you start to approach your

¹<https://github.com>

*2.3 Programming languages allow you to look at your data from a different angle*¹¹

problems differently, whether it is a coding problem or a problem in any other context. For example, I know many students enjoy the process of qualitative coding. However, they often struggle to detach their insights from the actual data and synthesise ideas on an abstract and more generic level. Qualitative researchers might refer to this as challenges in '*second-order deconstruction of meaning*'. This process of abstraction is a skill that needs to be honed, nurtured and practised. From my experience, programming languages are one way to achieve this, but they might not be recognised for this just yet. Programming languages, especially functions (see Section 5.3), require us to generalise from a particular case to a generic one. This mental mechanism is also helpful in other areas of research or work in general.

2.3 Programming languages allow you to look at your data from a different angle

There are commonly known and well-established techniques regarding how you should analyse your data rigorously. However, it can be quite some fun to try techniques outside your discipline. This does not only apply to programming languages, of course. Sometimes, learning about a new research method enables you to look at your current tools in very different ways too. One of the biggest challenges for any researcher is to reflect on one's own work. Learning new and maybe even '*strange*' tools can help with this. Admittedly, sometimes you might find out that some new tools are also a dead-end. Still, you likely have learned something valuable through the process of engaging with your data differently. So shake off the rust of your analytical routine and blow some fresh air into your research methods.

2.4 Learning any programming language will help you learn other programming languages.

Once you understand the logic of one language, you will find it relatively easy to understand new programming languages. Of course, if you wanted to, you could become the next '*Neo*' (from '*The Matrix*')² and change the reality of your research forever. On a more serious note, though, if you know any programming language already, learning *R* will be easier because you have accrued some basic understanding of these particular types of languages.

²https://www.imdb.com/title/tt0133093/?ref_=ext_shr_lnk

Having considered everything of the above, do you feel ready for your next foreign language?

3

Setting up R and RStudio

Every journey starts with gathering the right equipment. This intellectual journey is not much different. The first step that every *R* novice has to face is to set everything up to get started. There are essentially two strategies:

- Install *R*¹ and RStudio²
- or
- Run RStudio in a browser via RStudio Cloud³

While installing *R* and RStudio requires more time and effort, I strongly recommend it, especially if you want to work offline or make good use of your computer's CPU. However, if you are unsure whether you enjoy learning *R*, you might wish to look at RStudio Cloud first. Either way, you can follow the examples of this book no matter which choice you make.

3.1 Installing R

The core module of our programming is *R* itself, and since it is an open-source project, it is available for free on Windows, Mac and Linux computers. So, here is what you need to do to install it properly on your computer of choice:

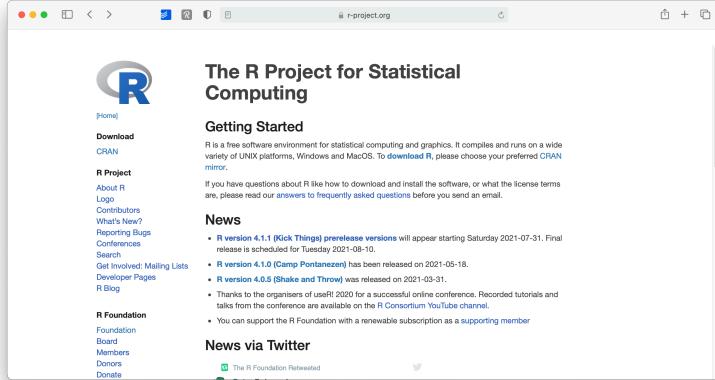
1. Go to www.r-project.org⁴

¹<https://www.r-project.org>

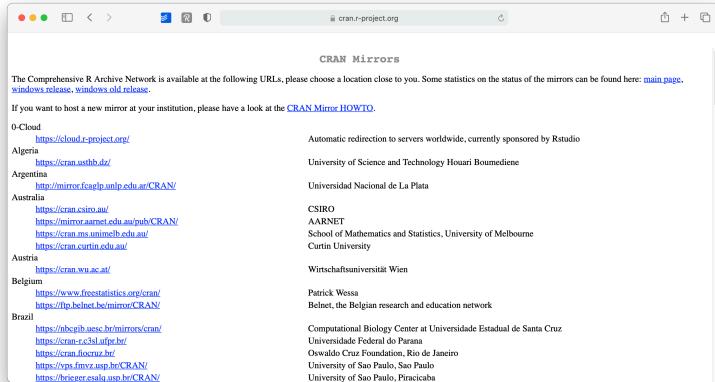
²<https://www.rstudio.com>

³<https://rstudio.cloud>

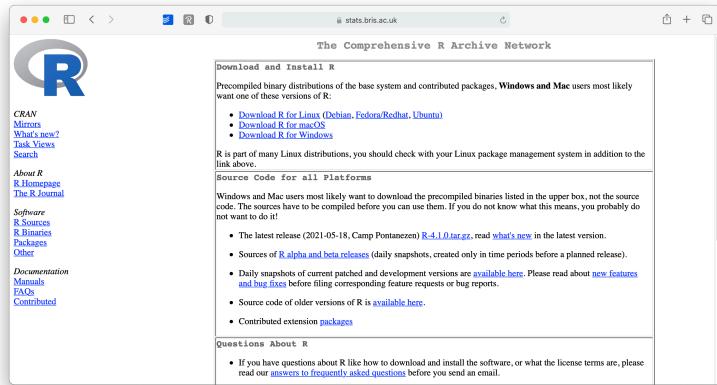
⁴[https://www.r-project.org%5D\(https://www.r-project.org\)](https://www.r-project.org%5D(https://www.r-project.org))



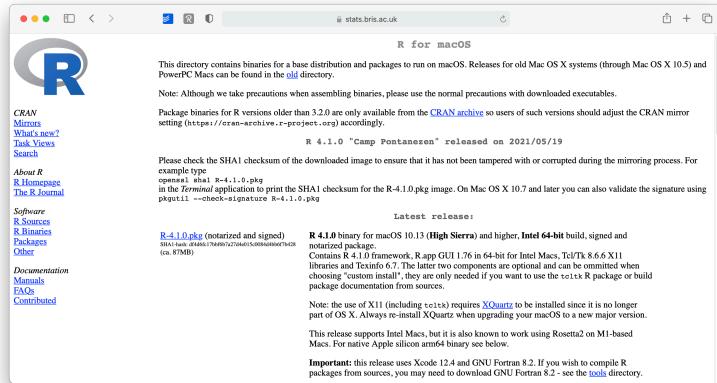
2. Click on CRAN where it says Download.
3. Choose a server in your country (all of them work, but downloads will perform quicker if you choose your country or one that is close to where you are).



4. Select the operating system for your computer, for example Download R for macOS.



5. Select the version you want to install (I recommend the latest version)



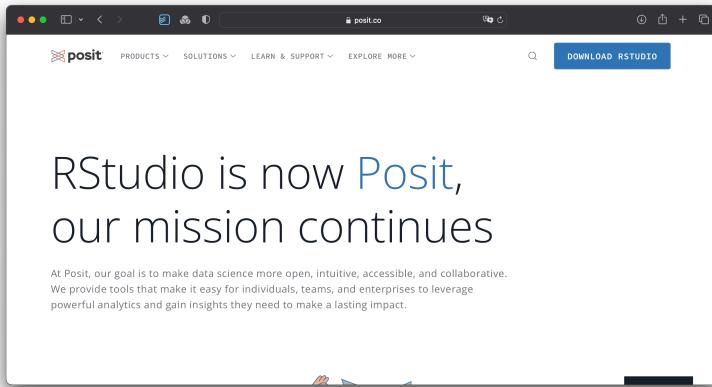
6. Open the downloaded file and follow the installation instructions. I recommend leaving the suggested settings as they are.

This was relatively easy. You now have *R* installed. Technically you can start using *R* for your research, but there is one more tool I strongly advise installing: RStudio.

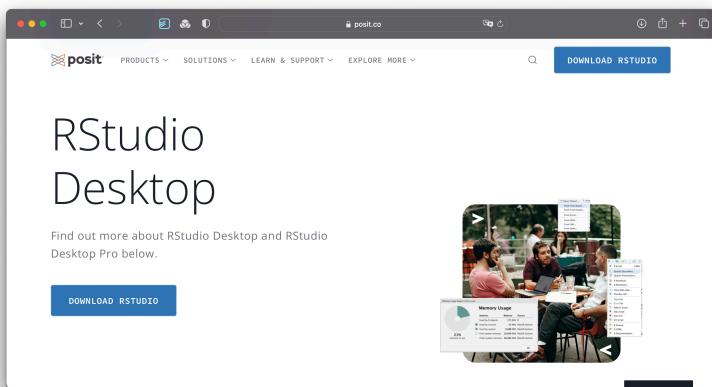
3.2 Installing RStudio

R by itself is just the ‘beating heart’ of *R* programming, but it has no particular user interface. If you want buttons to click and actually ‘see’ what you are doing, there is no better way than RStudio. RStudio is an *integrated development environment* (IDE) and will be our primary tool to interact with *R*. It is the only software you need to do all the fun parts and, of course, to follow along with the examples of this book. To install RStudio perform the following steps:

1. Go to <http://posit.co> and click on DOWNLOAD RSTUDIO.

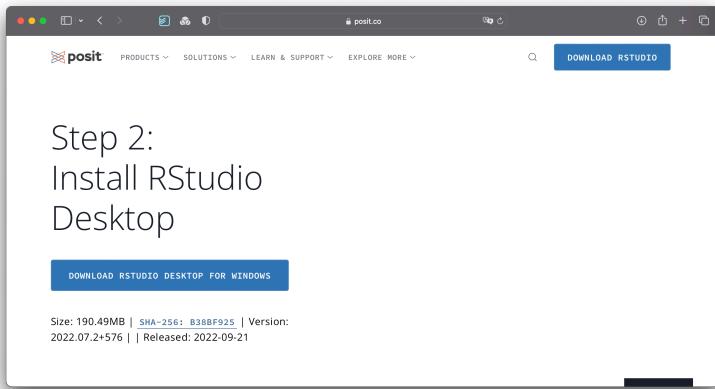


2. Click on DOWNLOAD RSTUDIO on this page.



3. This leads you to the page where you can install *R* as a first step

and RStudio as a second step. Since we installed *R* already, we can click on DOWNLOAD RSTUDIO DESKTOP FOR WINDOWS (or for Mac if you are not using a PC).



4. If for some reason the version for your operating system is not showing up correctly, you can scroll down and find other version ready to be installed.

A screenshot of the posit.co website showing a table of RStudio desktop download links. The table has four columns: OS, Download, Size, and SHA-256. There are three rows: Windows 10/11, macOS 10.15+, and Ubuntu 18+/Debian 10+. Each row shows a download link, file size, and SHA-256 hash.

OS	Download	Size	SHA-256
Windows 10/11	RSTUDIO-2022.07.2-576.EXE	190.49MB	B38BF925
macOS 10.15+	RSTUDIO-2022.07.2-576.DMG	224.49MB	35028D02
Ubuntu 18+/Debian 10+	RSTUDIO-2022.07.2-576-AMD64.DEB	133.19MB	B7D6C366

5. Open the downloaded file and follow the installation instructions. Again, keep it to the default settings as much as possible.

Congratulations, you are all set up to learn *R*. From now on you only need to start RStudio and not *R*. Of course, if you are the curious type, nothing shall stop you to try *R* without RStudio.

3.3 When you first start RStudio

Before you start programming away, you might want to make some tweaks to your settings right away to have a better experience (in my humble opinion). To open the Rstudio settings you have to click on

- RStudio > Tools > Global Options or press `⌘ + ,`, if you are on a Mac.
- RStudio > Tools > Global Options or press `Ctrl + ,`, if you work on a Windows computer.

I recommend to at least make the following changes to set yourself up for success right from the beginning:

1. Already on the first tab, i.e. General > Basic, we should make one of the most significant changes. Deactivate every option that starts with `Restore`. This will ensure that every time you start RStudio, you begin with a clean slate. At first sight, it might sound counter-intuitive not to restart everything where you left off, but it is essential to make all your projects easily reproducible. Furthermore, if you work together with others, not restoring your personal settings also ensures that your programming works across different computers. Therefore, I recommend having the following unticked:

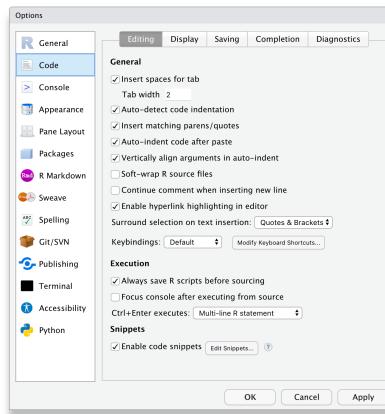
- `Restore most recently opened project at startup,`
- `Restore previously open source documents at startup,`
- `Restore .RData into workspace at startup`



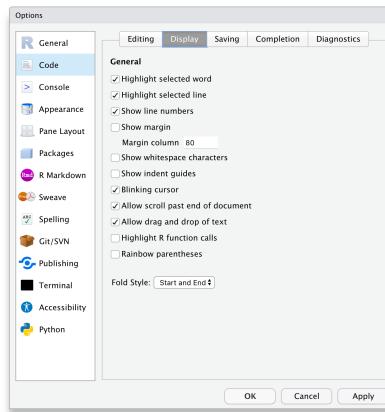
2. In the same tab under `Workspace`, select `Never` for the setting `Save workspace to .RData on exit`. One might think it is wise to keep intermediary results stored from one R session to another. However,

I often found myself fixing issues due to this lazy method, and my code became less reliable and, therefore, reproducible. With experience, you will find that this avoids many headaches.

3. In the `Code > Editing` tab, make sure to have at least the first five options ticked, especially the `Auto-indent code after paste`. This setting will save time when trying to format your coding appropriately, making it easier to read. Indentation is the primary way of making your code look more readable and less like a series of characters that appear almost random.

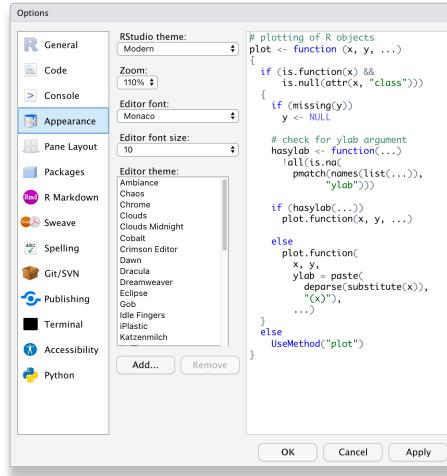


4. In the `Display` tab, you might want to have the first three options selected. In particular, `Highlight selected line` is helpful because, in more complicated code, it is helpful to see where your cursor is.



Of course, if you wish to customise your workspace further, you can do so. The visually most impactful way to alter the default appearance of RStudio

is to select **Appearance** and pick a completely different colour theme. Feel free to browse through various options and see what you prefer. There is no right or wrong here. Just make it your own.



3.4 Updating R and RStudio: Living at the pulse of innovation

While not strictly something that helps you become a better programmer, this advice might come in handy to avoid turning into a frustrated programmer. When you update your software, you need to update *R* and RStudio separately from each other. While both *R* and RStudio work closely with each other, they still constitute separate pieces of software. Thus, it is essential to keep in mind that updating RStudio will not automatically update *R*. This can become problematic if specific tools you installed via RStudio (like a fancy learning algorithm) might not be compatible with earlier versions of *R*. Also, additional *R* packages (see Section 5.4) developed by other developers are separate pieces which require updating too, independently from *R* and RStudio.

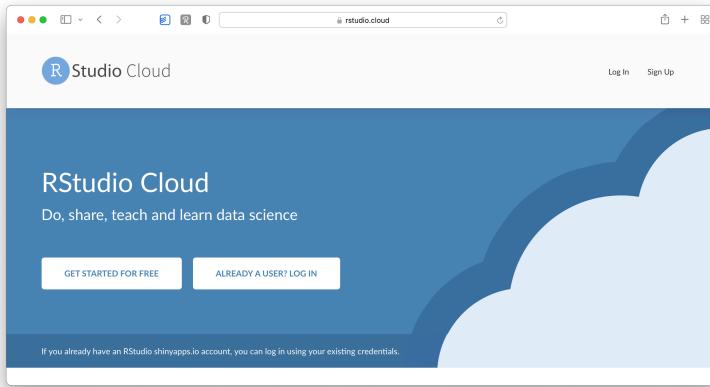
I know what you are thinking: This already sounds complicated and cumbersome. However, rest assured, we take a look at how you can easily update all your packages with RStudio. Thus, all you need to remember is: *R* needs to be updated separately from everything else.

3.5 RStudio Cloud

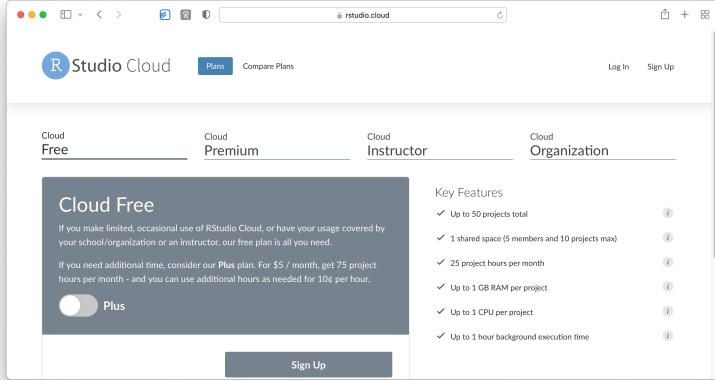
RStudio Cloud is an application that runs in your web browser. It allows you to write and access *R* code wherever you go. It even works on your tablet because it does not require installation. However, do expect it to run slower if your internet connection is not fast or stable. To work with RStudio Cloud, you also need an internet connection. However, there are many benefits to using RStudio in the cloud, such as running time-consuming scripts without using your own device. Also, it is much easier to collaborate with others, and since no installation is required, you can work on projects on any device as long as you are connected to the internet. However, RStudio Cloud's most significant advantage is that you can get started with programming within seconds compared to a desktop installation. Still, I prefer my locally-run offline version of RStudio, because I appreciate working offline as much as online. Nevertheless, I recommend setting up an account because you never know when you need it.

To get started with RStudio Cloud, we have to undertake a couple of steps:

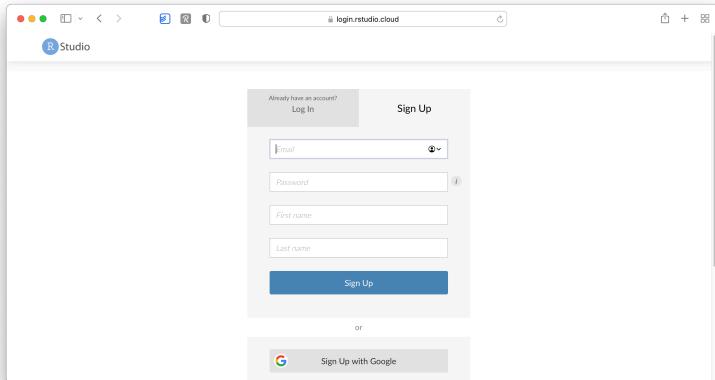
1. Open your web browser of choice and navigate to <https://rstudio.cloud>.



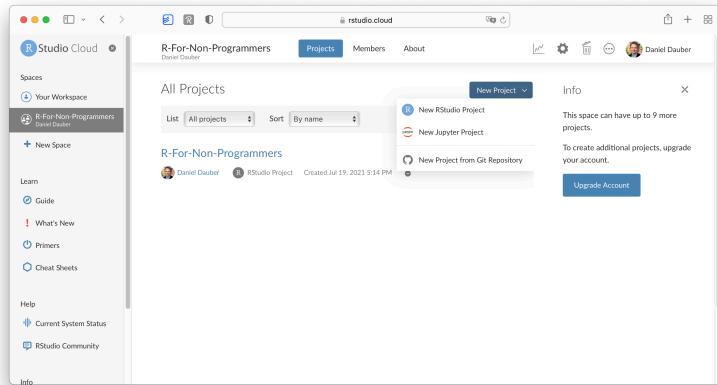
2. Click on **Sign Up** to create your account.
3. On the next page, make sure you have the free plan selected and click on **Sign up**.



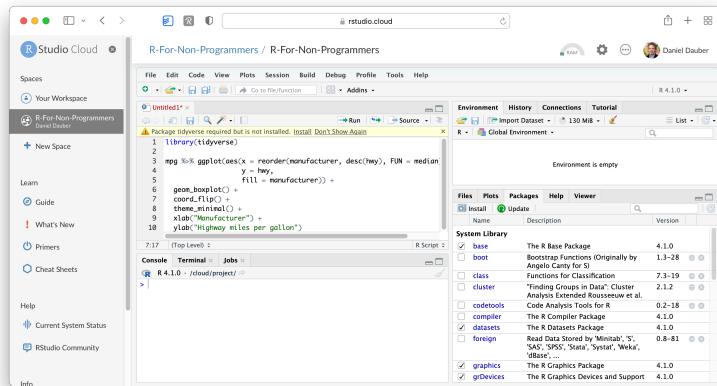
4. To finalise the registration process, you are required to provide your credentials.



5. Once you complete your registration, you are redirected to Your Workspace, the central hub for all your projects. As you can tell, I already added another workspace called R for Non-Programmers. However, it is fine to use the default one.

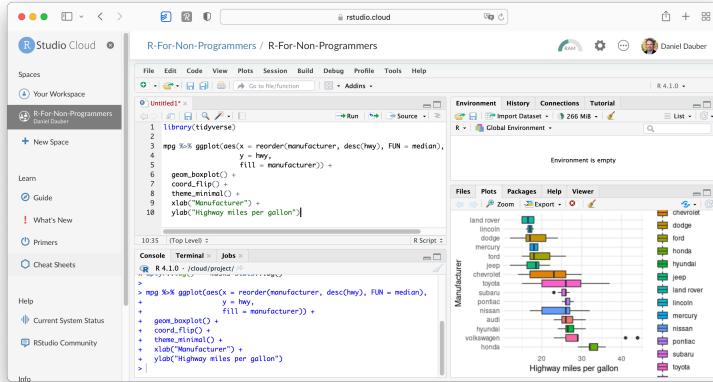


6. To start a new *R* project, you can click on **New Project > New RStudio Project**. This will open RStudio in the Cloud, which looks identical to the desktop version. You can immediately start writing your code. The example below computes a plot.⁵



7. Now we can execute the code as we would on the desktop version (see Chapter 5).

⁵RStudio and RStudio Cloud warn you if you need to install certain R packages to successfully run code. More about R packages and what they are can be found in Section 5.4.



No matter whether you choose to use a desktop version of RStudio or RStudio Cloud, you will be able to follow along in this book with no problem.

4

The RStudio Interface

The RStudio interface is composed of quadrants, each of which fulfils a unique purpose:

- The `Console` window,
- The `Source` window,
- The `Environment / History / Connections / Tutorial` window, and
- The `Files / Plots / Packages / Help / Viewer` window

Sometimes you might only see three windows and wonder where the `Source` window has gone in your version of RStudio. In order to use it you have to either open a file or create a new one. You can create a new file by selecting `File > New File > R Script` in the menu bar, or use the keyboard shortcut `Ctrl + Shift + N` on PC and `Cmd + Shift + N` on Mac.

I will briefly explain the purpose of each window/pane and how they are relevant to your work in *R*.

4.1 The Console window

The console is located in the bottom-left, and it is where you often will find the output of your coding and computations. It is also possible to write code directly into the console. Let's try the following example by calculating the sum of `10 + 5`. Click into the console with your mouse, type the calculation into your console and hit `Enter/Return ↵` on your keyboard. The result should be pretty obvious:

```
# We type the below into the console  
10 + 5
```

```
[1] 15
```

Here is a screenshot of how it should look like at your end in RStudio:



You just successfully performed your first successful computation. I know, this is not quite impressive just yet. *R* is undoubtedly more than just a giant calculator.

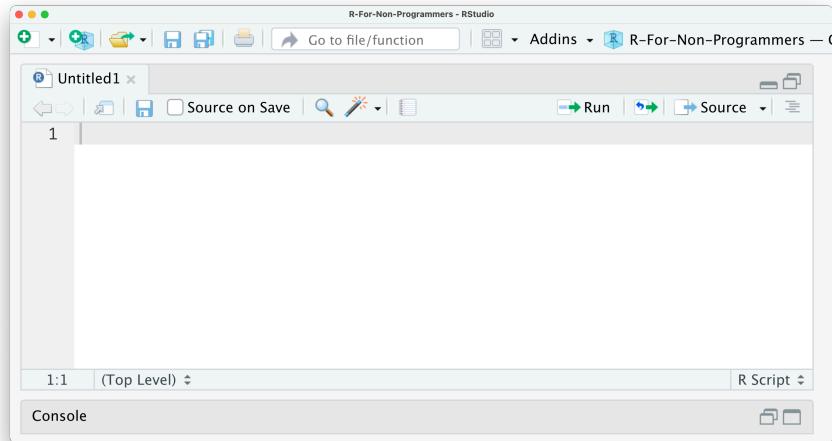
In the top right of the console, you find a symbol that looks like a broom. This one is quite an important one because it clears your console. Sometimes the console can become very cluttered and difficult to read. If you want to remove whatever you computed, you can click the broom icon and clear the console of all text. I use it so frequently that I strongly recommend learning the keyboard shortcut, which is **Ctrl + L** on PC and Mac.

4.2 The Source window

In the top left, you can find the source window. The term ‘source’ can be understood as any type of file, e.g. data, programming code, notes, etc. The source panel can fulfil many functions, such as:

- Inspect data in an Excel-like format (see also Section 7.1)
- Open programming code, e.g. an R Script (see Section 6.3)
- Open other text-based file formats, e.g.
 - Plain text (.txt),
 - Markdown (.md),
 - Websites (.html),

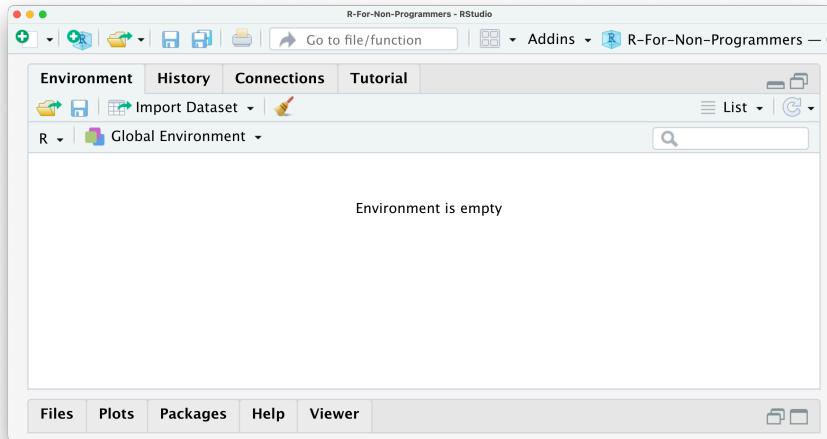
- LaTeX (.tex),
- BibTex (.bib),
- Edit scripts with code in it,
- Run the analysis you have written.



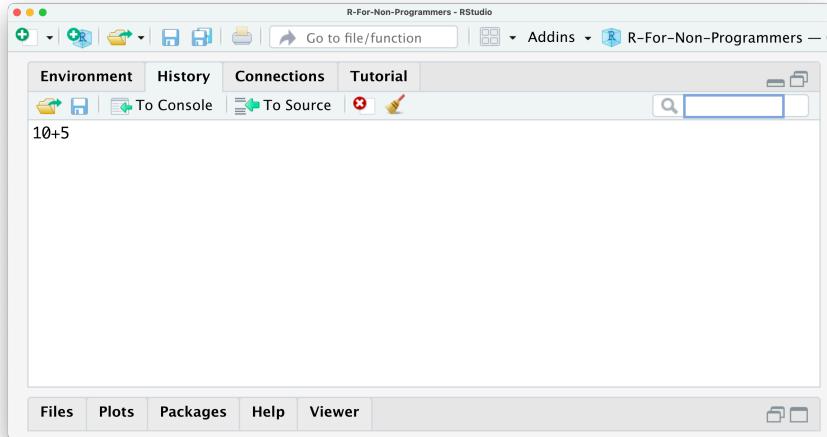
In other words, the source window will show you whatever file you are interested in, as long as RStudio can read it - and no, Microsoft Office Documents are not supported. Another limitation of the source window is that it can only show text-based files. So opening images, etc. would not work.

4.3 The Environment / History / Connections / Tutorial window

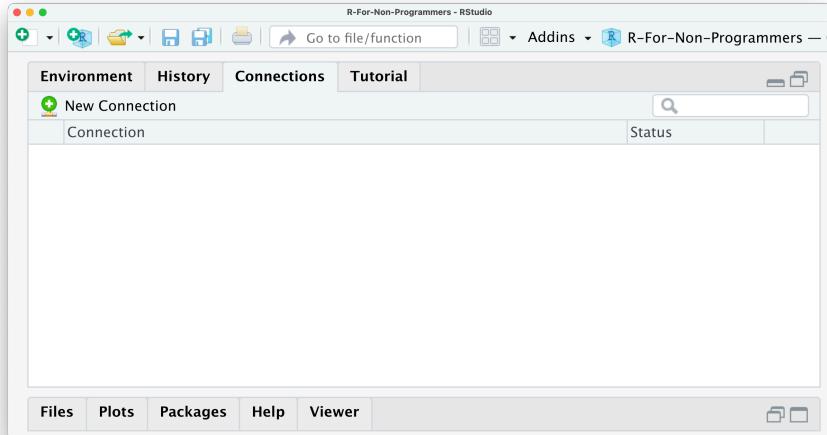
The window in the top right shows multiples panes. The first pane is called *Environment* and shows you objects which are available for computation. One of the first objects you will create is your dataset because, without data, we cannot perform any analysis. Another object could be a plot showing the number of male and female participants in your study. To find out how to create objects yourself, you can take a glimpse at it (see Section 7.2). Besides datasets and plots, you will also find other objects here, e.g. lists, vectors and functions you created yourself. Don't worry if none of these words makes sense at this point. We will cover each of them in the upcoming chapters. For now, remember this is a place where you can find different objects you created.



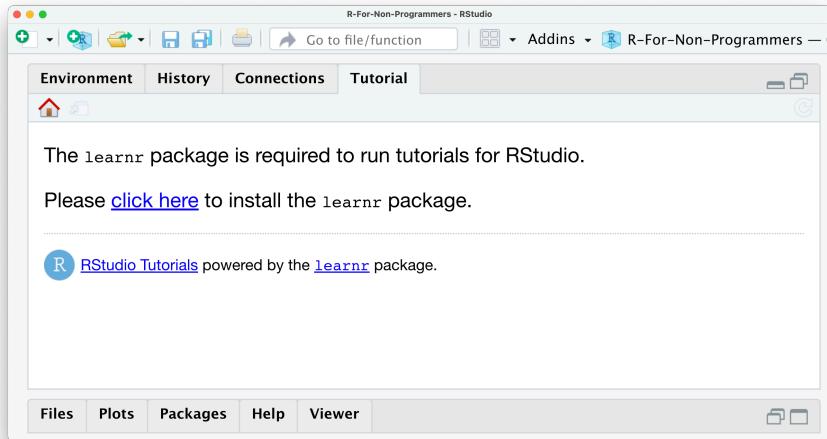
The *History* pane is very easy to understand. Whatever computation you run in the console will be stored. So you can go back and see what you coded and rerun that code. Remember the example from above where we computed the sum of $10+5$? This computation is stored in the history of RStudio, and you can rerun it by clicking on $10+5$ in the history pane and then click on **To Console**. This will insert $10+5$ back into the console, and we can hit **Return** \square to retrieve the result. You also have the option to copy the code into an existing or new *R* Script by clicking on **To Source**. By doing this, you can save this computation on your computer and reuse it later. Finally, if you would like to store your history, you can do so by clicking on the **floppy disk** symbol. There are two more buttons in this pane, one allows you to delete individual entries in the history (the white page with the red circle on it), and the last one, a **broom**, clears the entire history (irrevocably).



The pane *Connections* allows you to tab into external databases directly. This can come in handy when you work collaboratively on the same data or want to work with extensive datasets without having to download them. However, for an introduction to *R*, we will not use this feature of RStudio.

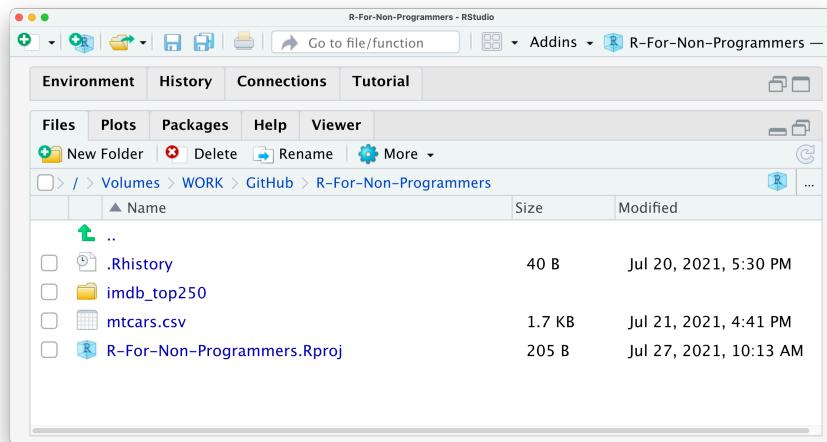


The last pane is called *Tutorial*. Here you can find additional materials to learn *R* and RStudio. If you search for more great content to learn *R*, this serves as a great starting point.



4.4 The Files / Plots / Packages / Help / Viewer window

The last window consists of five essential panes. The first one is the *Files* pane. As the name indicates, it lists all the files and folders in your root directory. A root directory is the default directory where RStudio saves your files, for example, your analysis. However, we will look at how you can properly set up your projects in RStudio in Chapter 6. Thus, the *Files* pane is an easy way to load data into RStudio and create folders to keep your research project well organised.



Since the Console cannot reproduce data visualisations, RStudio offers a way to do this very easily. It is through the *Plots* pane. This pane is exclusively designed to show you any plots you have created using *R*. Here is a simple example that you can try. Type into your console `boxplot(mtcars$hp)`.

```
# Here we create a nice boxplot using a dataset called 'mtcars'  
boxplot(mtcars$hp)
```

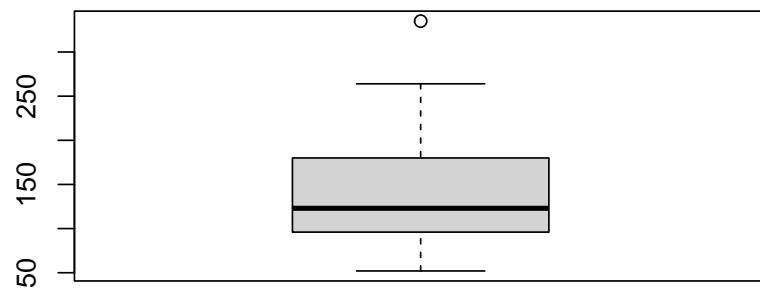
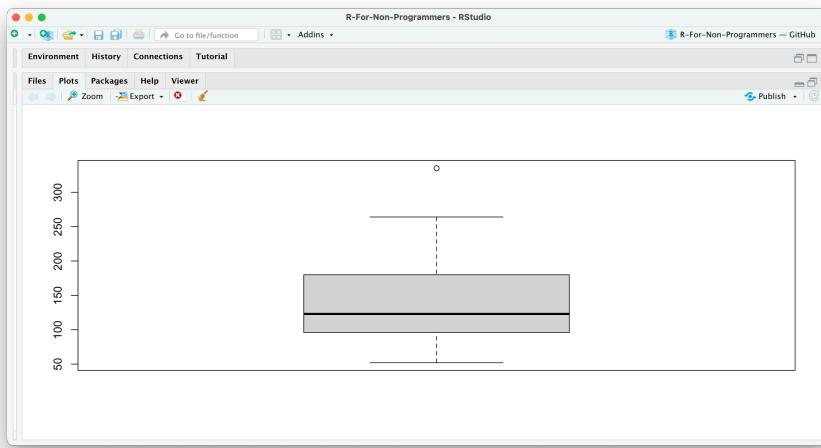


Figure 4.1

Although this is a short piece of coding, it performs quite a lot of steps:

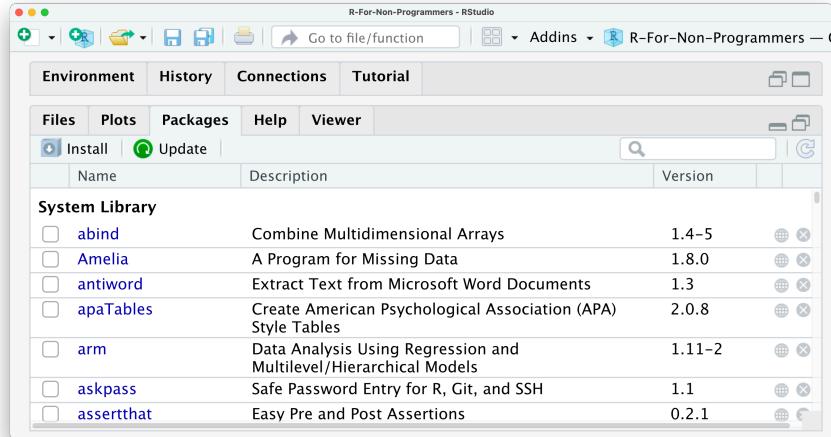
- it uses a function called `boxplot()` to draw a boxplot of
- a variable called `hp` (for horsepower), which is located in
- a dataset named `mtcars`,
- and it renders the graph in your *Plots* pane

This is how the plot should look like in your RStudio *Plots* pane.

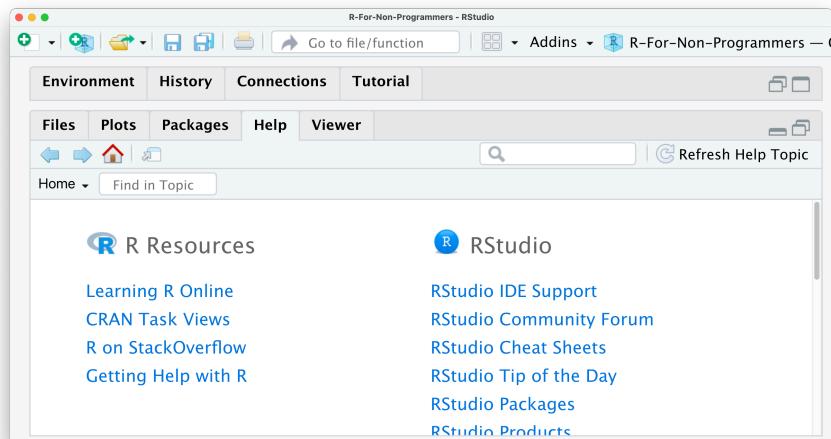


If you wish to delete the plot, you can click on the `red circle with a white x symbol`. This will delete the currently visible plot. If you wish to remove all plots from this pane, you can use the `broom`. There is also an option to export your plot and move back and forth between different plots.

The next pane is called *Packages*. Packages are additional tools you can import and use when performing your analysis. A frequent analogy people use to explain packages is your phone and the apps you install. Each package you download is equivalent to an app on your phone. It can enhance different aspects of working in *R*, such as creating animated plots, using unique machine learning algorithms, or simply making your life easier by doing multiple computations with just one single line of code. You will learn more about *R packages* in Section 5.4.



If you are in dire need of help, RStudio provides you with a *Help* pane. You can search for specific topics, for example how certain computations work. The *Help* pane also has documentation on different datasets that are included in *R*, RStudio or *R packages* you have installed. If you want a more comprehensive overview of how you can find help, have a look at CRAN's 'Getting Help with R'¹ webpage.

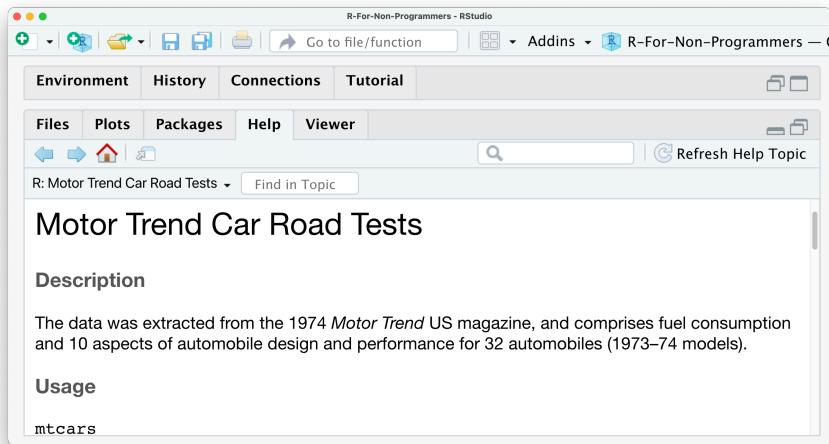


So, for example, if you want to know what the `mtcars` dataset is, you can either use the search window in the *Help* pane or, much easier, use a `?` in the console to search for it:

¹<https://www.r-project.org/help.html>

```
# Type a '?' followed by the name of a dataset/function/etc.
# to look up helpful information about it.
?mtcars
```

This will open the *Help* pane and give you more information about this dataset:



There are many different ways of how you can find help with your coding beyond RStudio and this book. My top three platforms to find solutions to my programming problems are:

- Google²
- stackoverflow.com³
- X/Twitter⁴ (with #RStats⁵)

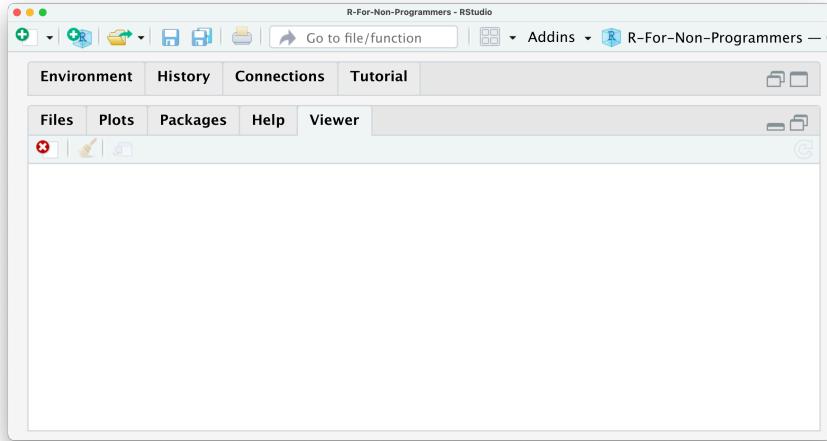
Lastly, we have the *Viewer* pane. Not every data visualisation we create in *R* is a static image. You can create dynamic data visualisations or even websites with *R*. This type of content is displayed in the *Viewer* pane rather than in the *Plots* pane. Often these visualisations are based on HTML and other web-based programming languages. As such, it is easy to open them in your browser as well. However, in this book, we mainly focus on two-dimensional static plots, which are the ones you likely need most of the time, either for your assignments, thesis, or publication.

²<https://www.google.com>

³<https://stackoverflow.com>

⁴<https://twitter.com/home>

⁵<https://twitter.com/hashtag/rstats>



4.5 Customise your user interface

As a last remark in this chapter, I would like to make you aware that you can modify each window. There are three basic adjustments you can make:

- Hide panes by clicking on the window symbol in the top right corner of each window,
- Resize panes by dragging the border of a window horizontally or vertically, or
- Add and remove panes by going to `RStudio > Preferences > Pane Layout`, or use the keyboard shortcut `⌘ + ,` if you are on a Mac. Unfortunately, there is no default shortcut for PC users.

If you want a fully customised experience you can also alter the colour scheme of RStudio itself (`RStudio > Preferences > Appearance`) and if the themes offered are not enough for you, you can create a custom theme here⁶.

⁶<https://tmtheme-editor.herokuapp.com/#!/editor/theme/Monokai>



5

R Basics: The very fundamentals

After a likely tedious installation of *R* and RStudio, as well as a somewhat detailed introduction to the RStudio interface, you are finally ready to ‘do’ things. By ‘*doing*’, I mean ‘*coding*’. The term ‘*coding*’ in itself can instil fear in some of you, but you only need one skill to do it: Writing. As mentioned earlier, learning coding or programming means learning a new language. However, once you have the basic grammar down, you already can communicate quite a bit. In this section, we will explore the fundamentals of *R*. These build the foundation for everything that follows. After that, we dive right into some analysis.

5.1 Basic computations in *R*

The most basic computation you can do in *R* is arithmetic operations. In other words, addition, subtraction, multiplication, division, exponentiation and extraction of roots. In short, *R* can be used like your pocket calculator, or more likely the one you have on your phone. For example, in Section 4.1 we already performed an addition. Thus, it might not come as a surprise how their equivalents work in *R*. Let’s take a look at the following examples:

```
# Addition  
10 + 5
```

```
[1] 15
```

```
# Subtraction  
10 - 5
```

```
[1] 5
```

```
# Multiplication  
10 * 5
```

```
[1] 50
```

```
# Division  
10 / 5
```

```
[1] 2
```

```
# Exponentiation  
10 ^ 2
```

```
[1] 100
```

```
# Square root  
sqrt(10)
```

```
[1] 3.162278
```

They all look fairly straightforward except for the extraction of roots. As you probably know, extracting the root would typically mean we use the symbol $\sqrt{}$ on our calculator. To compute the square root in *R*, we have to use a function instead to perform the computation. So we first put the name of the function `sqrt` and then the value `10` within parenthesis `()`. This results in the following code: `sqrt(10)`. If we were to write this down in our report, we would write $\sqrt{10}$.

Functions are an essential part of *R* and programming in general. You will learn more about them in Section 5.3. Besides arithmetic operations, there are also logical queries you can perform. Logical queries always return either the value `TRUE` or `FALSE`. Here are some examples which make this clearer:

```
#1 Is it TRUE or FALSE?  
1 == 1
```

```
[1] TRUE
```

```
#2 Is 45 bigger than 55?  
45 > 55
```

```
[1] FALSE
```

```
#3 Is 1982 bigger or equal to 1982?  
1982 >= 1982
```

```
[1] TRUE
```

```
#4 Are these two words NOT the same?
"Friends" != "friends"

[1] TRUE

#5 Are these sentences the same?
"I love statistics" == "I love statistícs"

[1] FALSE
```

Reflecting on these examples, you might notice three important aspects of logical queries:

1. We have to use `==` instead of `=`,
2. We can compare non-numerical values, i.e. text, which is also known as `character` values, with each other,
3. The devil is in the details (consider #5).

One of the most common mistakes of *R* novices is the confusion around the `==` and `=` notation. While `==` represents `equal to`, `=` is used to assign a value to an object (for more details on assignments see Section 5.2). However, in practice, most *R* programmers tend to avoid `=` since it can easily lead to confusion with `==`. As such, you can strike `=` out of your *R* vocabulary for now.

There are many different logical operations you can perform. Table 5.1 lists the most frequently used logical operators for your reference. These will become important, for example when we filter our data for analysis, e.g. include only female or male participants.

Table 5.1: Logical operators in R

Operator	Description
<code>==</code>	is equal to
<code>!=</code>	is not equal to
<code>>=</code>	is bigger or equal to
<code>></code>	is bigger than
<code><=</code>	is smaller or equal to
<code><</code>	is smaller than
<code>a b</code>	a or b
<code>a & b</code>	a and b
<code>!a</code>	is not a

5.2 Assigning values to objects: ‘<-’

Another common task you will perform is assigning values to an object. An object can be many different things:

- a dataset,
- the results of a computation,
- a plot,
- a series of numbers,
- a list of names,
- a function,
- etc.

In short, an object is an umbrella term for many different things which form part of your data analysis. For example, objects are handy when storing results that you want to process further in later analytical steps. We have to use the assign operator `<-` to assign a value to an object. Let's have a look at an example.

```
# I have a friend called "Fiona"
friends <- "Fiona"
```

In this example, I created an object called `friends` and added "Fiona" to it. Since "Fiona" represents a `string`, we need to use `""`. So, if you wanted to read this line of code, you would say, '`friends` gets the value "Fiona". Alternatively, you could also say "Fiona" is assigned to `friends`'.

If you look into your environment pane, you will find the object `friends`. You can see it carries the value "Fiona". We can also print values of an object in the console by simply typing the name of the object `friends` and hit `Return`.

```
# Who are my friends?
friends
```

```
[1] "Fiona"
```

Sadly, it seems I only have one friend. Luckily we can add some more, not the least to make me feel less lonely. To create objects with multiple values, we can use the function `c()`, which stands for 'concatenate'. The Cambridge Dictionary (2021) defines this word as follows:

‘concatenate’,
to put things together as a connected series.

Let’s concatenate some more friends into our `friends` object.

```
# Adding some more friends to my life
friends <- c("Fiona",
           "Lukas",
           "Ida",
           "Georg",
           "Daniel",
           "Pavel",
           "Tigger")

# Here are all my friends
friends
```

[1] "Fiona" "Lukas" "Ida" "Georg" "Daniel" "Pavel" "Tigger"

To concatenate values into a single object, we need to use a comma , to separate each value. Otherwise, *R* will report an error back.

```
friends <- c("Fiona" "Ida")
```

```
Error in parse(text = input): <text>:1:22: unexpected string constant
1: friends <- c("Fiona" "Ida"
               ^
```

R’s error messages tend to be very useful and give meaningful clues to what went wrong. In this case, we can see that something ‘unexpected’ happened, and it shows where our mistake is. You can also concatenate numbers

Btw, if you add () around your code, you can automatically print the content of the object to the console. Thus,

- `(milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020))` is the same as
- `milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020)` followed by `milestones_of_my_life`.

You can copy and paste the following example to illustrate what I explained.

```
# Important years in my life
milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020)
milestones_of_my_life
```

```
[1] 1982 2006 2011 2018 2020
```

```
# The same as above - no second line of code needed
(milestones_of_my_life <- c(1982, 2006, 2011, 2018, 2020))
```

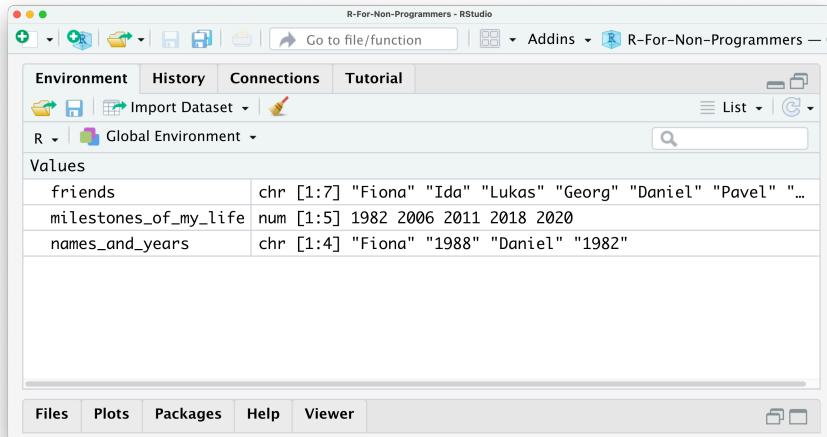
```
[1] 1982 2006 2011 2018 2020
```

Finally, we can also concatenate numbers and character values into one object:

```
(names_and_years <- c("Fiona", 1988, "Daniel", 1982))
```

```
[1] "Fiona"  "1988"   "Daniel" "1982"
```

This last example is not necessarily something I would recommend to do, because it likely leads to undesirable outcomes. For example, if you look into your environment pane you currently have three objects: `friends`, `milestones_of_my_life`, and `names_and_years`.



The `friends` object shows that all the values inside the object are classified as `chr`, which denotes `character`. For this object, this is correct because it only includes the names of my friends. On the other hand, the object `milestones_of_my_life` only includes `numeric` values, and therefore it says `num` in the environment pane. However, for the object `names_and_years` we know we want to have `numeric` and `character` values included. Still, *R* recognises them

as `character` values because values inside objects are meant to be of the same type. Remember that in *R*, numbers can always be interpreted as `character` and `numeric`, but text only can be considered as `character`.

Consequently, mixing different types of data into one object is likely a bad idea. This is especially true if you want to use the `numeric` values for computation. We will return to data types in Section 7.4 where we learn how to change them, but for now we should ensure that our objects are all of the same data type.

However, there is an exception to this rule. ‘Of course’, you might say. There is one object that can have values of different types: a `list`. As the name indicates, a `list` object holds several items. These items are usually other objects. In the spirit of ‘Inception¹’, you can have lists inside lists, which contain more lists or other objects.

Let’s create a `list` called `x_files` using the `list` function and place all our objects inside.

```
# This creates our list of objects
x_files <- list(friends,
                 milestones_of_my_life,
                 names_and_years)

# Let's have a look what is hidden inside the x_files
x_files
```

```
[[1]]
[1] "Fiona"   "Lukas"   "Ida"      "Georg"   "Daniel"  "Pavel"   "Tigger"

[[2]]
[1] 1982 2006 2011 2018 2020

[[3]]
[1] "Fiona"   "1988"   "Daniel"   "1982"
```

You will notice in this example that I do not use "" for each value in the list. This is because `friends` is not a `character` value, but an object. When we refer to objects, we do not need quotation marks.

We will encounter `list` objects quite frequently when we perform our analysis. Some functions return the results in the format of lists. This can be very helpful because otherwise our environment pane will be littered with objects and we would not necessarily know how they relate to each other, or worse, to which analysis they belong. Looking at the list item in the environment pane

¹https://www.imdb.com/title/tt1375666/?ref_=ext_shr_lnk

(Figure 5.1), you can see that the object `x_files` is classified as a `List of 3`, and if you click on the blue icon, you can inspect the different objects inside.

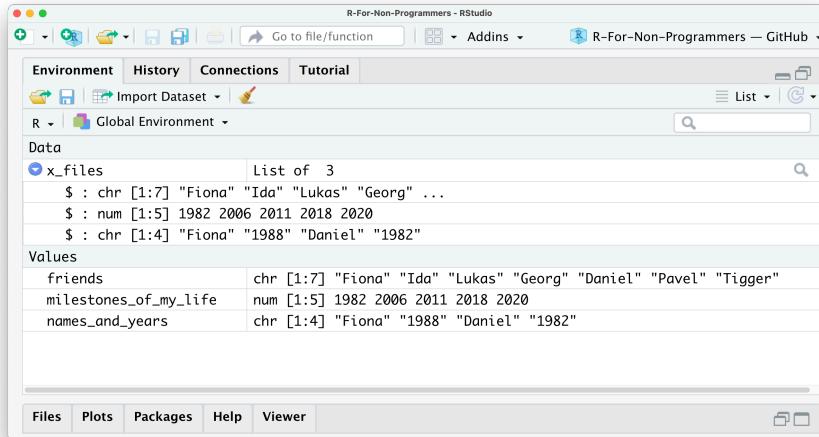


Figure 5.1: The environment pane showing our objects and our list `x_files`

In Section 5.1 , I mentioned that we should avoid using the `=` operator and explained that it is used to assign values to objects. You can, if you want, use `=` instead of `<-`. They fulfil the same purpose. However, as mentioned before, it is not wise to do so. Here is an example that shows that, in principle, it is possible.

```
# DO
(avengers1 <- c("Iron Man",
                 "Captain America",
                 "Black Widow",
                 "Vision"))

[1] "Iron Man"          "Captain America" "Black Widow"      "Vision"

# DON'T
(avengers2 = c("Iron Man",
                 "Captain America",
                 "Black Widow",
                 "Vision"))

[1] "Iron Man"          "Captain America" "Black Widow"      "Vision"
```

On a final note, naming your objects is limited. You cannot chose any name.

First, every name needs to start with a letter. Second, you can only use letters, numbers _ and . as valid components of the names for your objects (see also Chapter 4.2 in Wickham and Grolemund 2016). I recommend to establish a naming convention that you adhere to. Personally I prefer to only user lower-case letters and _ to separate/connect words. Ideally, you want to keep names informative, succinct and precise. Here are some examples of what some might consider good and bad choices for names.

```
# Good choices
income_per annum
open_to_exp          # for 'openness to new experiences'
soc_int              # for 'social integration'

# Bad choices
IncomePerAnnum
measurement_of_boredom_of_watching_youtube
Sleep.per_monthsIn.hours
```

Ultimately, you need to be able to effectively work with your data and output. Ideally, this should be true for others as well who want or need to work with your analysis and data as well, e.g. your co-investigator or supervisor. The same is applies to column names in datasets (see Section 7.3). Some more information about coding style and coding etiquette can be found in Section 5.5.

5.3 Functions

I used the term ‘*function*’ multiple times, but I never thoroughly explained what they are and why we need them. In simple terms, functions are objects. They contain lines of code that someone has written for us or we have written ourselves. One could say they are code snippets ready to use. Someone else might see them as shortcuts for our programming. Functions increase the speed with which we perform our analysis and write our computations and make our code more readable and reliable. Consider computing the `mean` of values stored in the object `pocket_money`.

```
# First we create an object that stores our desired values
pocket_money <- c(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89)

#1 Manually compute the mean
sum <- 0 + 1 + 1 + 2 + 3 + 5 + 8 + 13 + 21 + 34 + 55 + 89
sum / 12 # There are 12 items in the object
```

```
[1] 19.33333
```

```
#2 Use a function to compute the mean
mean(pocket_money)
```

```
[1] 19.33333
```

```
#3 Let's make sure #1 and #2 are actually the same
sum / 12 == mean(pocket_money)
```

```
[1] FALSE
```

If we manually compute the mean, we first calculate the sum of all values in the object `pocket_money`². Then we divide it by the number of values in the object, which is 12. This is the traditional way of computing the mean as we know it from primary school. However, by simply using the function `mean()`, we not only write considerably less code, but it is also much easier to understand because the word ‘*mean*’ does precisely what we would expect. Which approach do you find easier: Using the function `mean()` or compute it by hand?

To further illustrate how functions look like, let’s create one ourselves and call it `my_mean`.

```
my_mean <- function(numbers){
  # Compute the sum of all values in 'numbers'
  sum <- sum(numbers)

  # Divide the sum by the number of items in 'numbers'
  result <- sum/length(numbers)

  # Return the result in the console
  return(result)
}

my_mean(pocket_money)
```

```
[1] 19.33333
```

Don’t worry if half of this code does not make sense to you. Writing functions is an advanced *R* skill. However, it is good to know how functions look on the ‘inside’. You certainly can see the similarities between the code we have written before, but instead of using actual numbers, we work with placeholders like

²If you find the order of numbers suspicious, it is because it represents the famous Fibonacci sequence³.

numbers. This way, we can use a function for different data and do not have to rewrite it every time. Writing and using functions relates to the skill of abstraction mentioned in Section 2.2.

All functions in *R* share the same structure. They have a `name` followed by `()`. Within these parentheses, we put `arguments`, which have specific `values`. For example, a generic structure of a function would look something like this:

```
name_of_function(argument_1 = value_1,
                  argument_2 = value_2,
                  argument_3 = value_3)
```

How many arguments there are and what kind of values you can provide is very much dependent on the function you use. Thus, not every function takes every value. In the case of `mean()`, the function takes an object which holds a sequence of `numeric` values. It would make very little sense to compute the mean of our `friends` object, because it only contains names. *R* would return an error message:

```
mean(friends)
```

```
Warning in mean.default(friends): argument is not numeric or logical: returning NA
```

```
[1] NA
```

`NA` refers to a value that is ‘*not available*’. In this case, *R* tries to compute the mean, but the result is not available, because the values are not `numeric` but a `character`. In your dataset, you might find values that are `NA`, which means there is data missing. If a function attempts a computation that includes even just a single value that is `NA`, *R* will return `NA`. However, there is a way to fix this. You will learn more about how to deal with `NA` values in Section 7.7.

Sometimes you will also get a message from *R* that states `NaN`. `NaN` stands for ‘*not a number*’ and is returned when something is not possible to compute, for example:

```
# Example 1
0 / 0
```

```
[1] NaN
```

```
# Example 2
sqrt(-9)
```

```
Warning in sqrt(-9): NaNs produced
```

```
[1] NaN
```

5.4 R packages

R has many built-in functions that we can use right away. However, some of the most interesting ones are developed by different programmers, data scientists and enthusiasts. To add more functions to your repertoire, you can install *R* packages. *R* packages are a collection of functions that you can download and use for your own analysis. Throughout this book, you will learn about and use many different *R* packages to accomplish various tasks. To give you another analogy,

- *R* is like a global supermarket where everyone can offer their products,
- RStudio is like my shopping cart where I can put the products I like, and
- *R* packages are the products I can pick from the shelves.

Luckily, *R* packages are free to use, so I do not have to bring my credit card. For me, these additional functions, developed by some of the most outstanding scientists, is one of many reasons that keeps me addicted to performing my research in *R*.

R packages do not only include functions but often include datasets and documentation of what each function does. This way, you can easily try every function right away, even without your own dataset and read through what each function in the package does. Figure 5.2 shows the documentation of an *R* package called `ggplot2`.

However, how do you find those *R* packages? They are right at your fingertips. You have two options:

1. Use the function `install.packages()`, or
2. Use the packages pane in RStudio.

5.4.1 Installing packages using `install.packages()`

The simplest and fastest way to install a package is calling the function `install.packages()`. You can either use it to install a single package or install a series of packages all at once using our trusty `c()` function. All you need to know is the name of the package. This approach works for all packages that are on CRAN (remember CRAN from Section 3.1?).

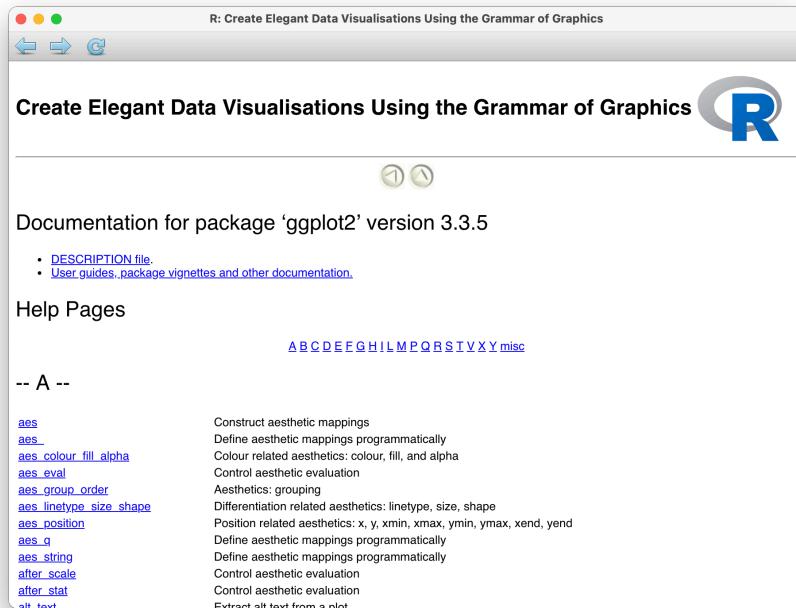


Figure 5.2: The R package documentation for ‘ggplot2’

```
# Install a single package
install.packages("tidyverse")

# Install multiple packages at once
install.packages(c("tidyverse", "naniar", "psych"))
```

If a package is not available from CRAN, chances are you can find them on GitHub⁴. GitHub is probably the world’s largest global platform for programmers from all walks of life, and many of them develop fantastic R packages that make programming and data analysis not just easier but a lot more fun. As you continue to work in R, you should seriously consider creating your own account to keep backups of your projects (see also Section 15.1).

An essential companion for this book is `r4np`, which contains all datasets for this book and some useful functions to get you up and running in no time. Since it is currently only available on Github, you can use the following code

⁴<https://github.com>

snippet to install it. However, you have to install the package `devtools` first to use the function `install_github()`.

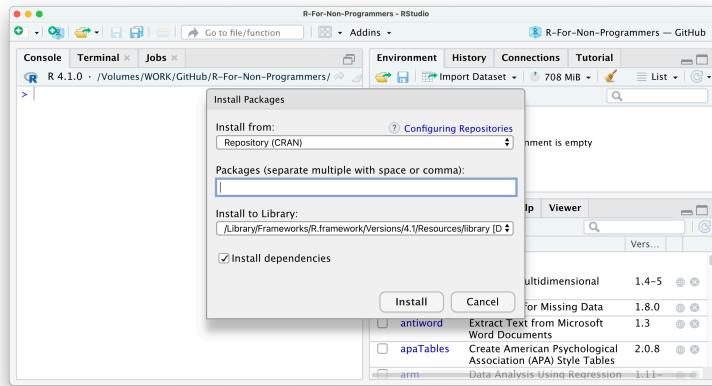
```
# Install the 'devtools' package first
install.packages("devtools")

# Then install the 'r4np' package from GitHub
devtools::install_github("ddrauber/r4np")
```

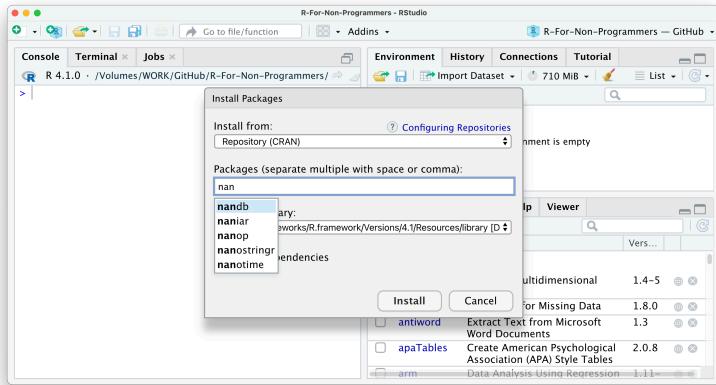
5.4.2 Installing packages via RStudio's package pane

RStudio offers a very convenient way of installing packages. In the packages pane, you cannot only see your installed packages, but you have two more buttons: **Install** and **Update**. The names are very self-explanatory. To install an *R* package you can follow the following steps:

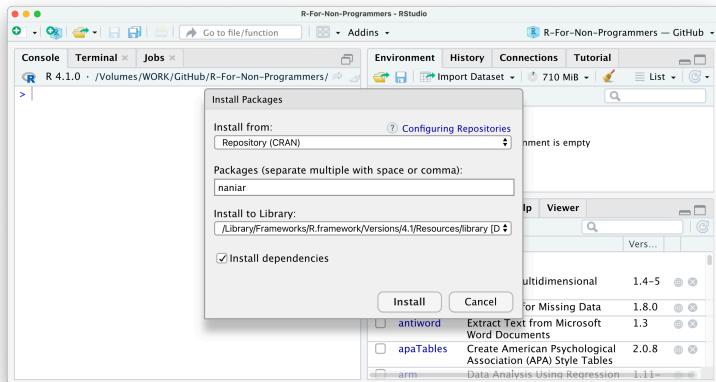
1. Click on **Install**.
2. In most cases, you want to make sure you have **Repository (CRAN)** selected.



3. Type in the name of the package you wish to install. RStudio offers an auto-complete feature to make it even easier to find the package you want.



4. I recommend NOT to change the option which says `Install to library`. The default library settings will suffice.
5. Finally, I recommend to select `Install dependencies`, because some packages need other packages to function properly. This way, you do not have to do this manually.



The only real downside of using the packages pane is that you cannot install packages hosted on GitHub only. However, you can download them from there and install them directly from your computer using this option. This is particularly useful if you do not have an internet connection but you already downloaded the required packages onto a hard drive. However, in 99.9% of cases it is much easier to use the function `devtools::install_github()`.

5.4.3 Install all necessary *R* packages for this book

Lastly, if you would like to install the necessary packages to follow the examples in this book, the `r4np` package comes with a handy function to install them all at once. Of course, you need to have `r4np` installed first, as shown above.

```
# Install all R packages used in this book
r4np::install_r4np()
```

5.4.4 Using *R* Packages

Now that you have a nice collection of *R* packages, the next step would be to use them. While you only have to install *R* packages once, you have to ‘activate’ them every time you start a new session in RStudio. This process is also called ‘loading an *R* package’. Once an *R* package is loaded, you can use all its functions. To load an *R* package, we have to use the function `library()`.

```
library(tidyverse)
```

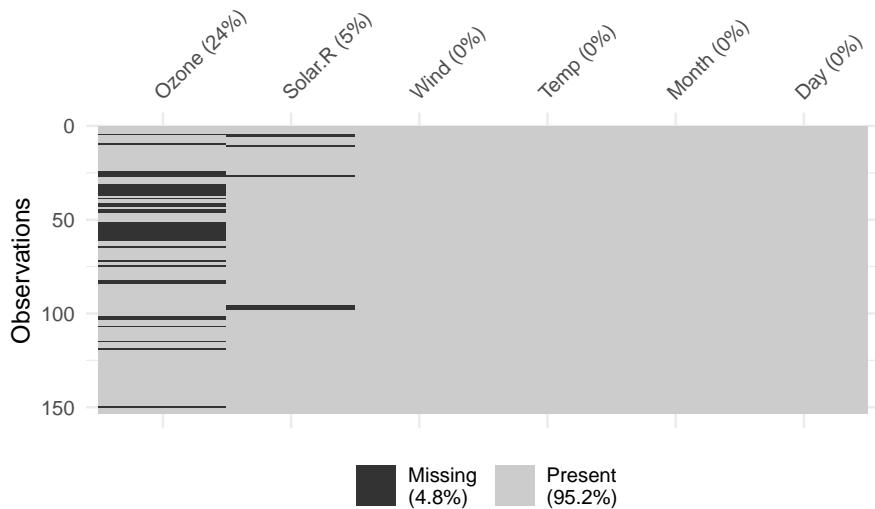
The `tidyverse` package is a special kind of package. It contains multiple packages and loads them all at once. Almost all included packages you will use at some point when working through this book.

I know what you are thinking. Can you use `c()` to load all your packages at once? Unfortunately not. However, there is a way to do this, but it goes beyond the scope of this book to fully explain this. If you are curious, you can take a peek at the code here on stackoverflow.com⁵.

Besides, it is not always advisable to load all functions of an entire package. One reason could be that two packages contain a function with the same name but with a different purpose. Two functions with the same name create a conflict between these two packages, and one of the functions would not be usable. Another reason could be that you only need to use the function once, and loading the whole package to use only one specific function seems excessive. Instead, you can explicitly call functions from packages without loading the package. For example, we might want to use the `vis_miss()` function from the `naniar` package to show where data is missing in our dataset `airquality`. Using functions without `library()` is also much quicker than loading the package and then calling the function if you don’t use it repeatedly. Make sure you have `naniar` installed (see above). We will work with this package when we explore missing data in Section 7.7. To use a function from an *R* package without loading it, we have to use `::` between the package’s name and the function we want to use. Copy the code below and try it yourself.

⁵<https://stackoverflow.com/questions/8175912/load-multiple-packages-at-once>

```
# Here I use the dataset 'airquality', which comes with R
naniar::vis_miss(airquality)
```



5.5 Coding etiquette

Now you know everything to get started, but before we jump into our first project, I would like to briefly touch upon coding etiquette. This is not something that improves your analytical or coding skills directly, but is essential in building good habits and making your life and those of others a little easier. Consider writing code like growing plants in your garden. You want to nurture the good plants, remove the weed and add labels that tell you which plant it is that you are growing. At the end of the day, you want your garden to be well-maintained. Treat your programming code the same way.

A script (see Section 6.3) of programming code should always have at least the following qualities:

- Only contains code that is necessary,
- Is easy to read and understand,
- Is self-contained.

With simple code this is easily achieved. However, what about more complex and longer code representing a whole set of analytical steps?

```
# Very messy code

library(tidyverse)
library(jtools)
model1 <- lm(covid_cases_per_1m ~ idv, data = df)
summ(model1, scale = TRUE, transform.response = TRUE, vifs = TRUE)
df |> ggplot(aes(x = covid_cases_per_1m, y = idv, col = europe, label = country))+
  theme_minimal() + geom_label(nudge_y = 2) + geom_point()
model2 <- lm(cases_per_1m ~ idv + uai + idv*europe + uai*europe, data = df)
summ(model2, scale = TRUE, transform.response = TRUE, vifs = TRUE)
anova(model1, model2)
```

How about the following in comparison?

```
# Nicely structured code

# Load required R packages
library(tidyverse)
library(jtools)

# ---- Modelling COVID-19 cases ----

## Specify and run a regression
model1 <- lm(covid_cases_per_1m ~ idv, data = df)

## Retrieve the summary statistics of model1
summ(model1,
      scale = TRUE,
      transform.response = TRUE,
      vifs = TRUE)

# Does is matter whether a country lies in Europe?

## Visualise rel. of covid cases, idv and being a European country
df |>
  ggplot(aes(x = covid_cases_per_1m,
             y = idv,
             col = europe,
             label = country)) +
  theme_minimal() +
  geom_label(nudge_y = 2) +
  geom_point()
```

```
## Specify and run a revised regression
model2 <- lm(cases_per_1m ~ idv + uai + idv*europe + uai*europe,
              data = df)

## Retrieve the summary statistics of model2
summ(model2,
      scale = TRUE,
      transform.response = TRUE,
      vifs = TRUE)

## Test whether model2 is an improvement over model1
anova(model1, model2)
```

I hope we can agree that the second example is much easier to read and understand even though you probably do not understand most of it yet. For once, I separated the different analytical steps from each other like paragraphs in a report. Apart from that, I added comments with # to provide more context to my code for someone else who wants to understand my analysis. Admittedly, this example is a little excessive. Usually, you might have fewer comments. Commenting is an integral part of programming because it allows you to remember what you did. Ideally, you want to strike a good balance between commenting on and writing your code. How many comments you need will likely change throughout your R programming journey. Think of comments as headers for your programming script that give it structure.

We can use # not only to write comments but also to tell R not to run particular code. This is very helpful if you want to keep some code but do not want to use it yet. There is also a handy keyboard shortcut you can use to ‘deactivate’ multiple lines of code at once. Select whatever you want to ‘comment out’ in your script and press **Ctrl+Shift+C** (PC) or **Cmd+Shift+C** (Mac).

```
# mean(pocket_money) # R will NOT run this code
mean(pocket_money) # R will run this code
```

RStudio helps a lot with keeping your coding tidy and properly formatted. However, there are some additional aspects worth considering. If you want to find out more about coding style, I highly recommend to read through ‘*The tidyverse style guide*⁶’ (Wickham 2021).

⁶<https://style.tidyverse.org>

5.6 Exercises

It is time to practice the newly acquired skills. The `r4np` package comes with interactive tutorials. In order to start them, you have two options. If you have installed the `r4np` package already and used the function `install_r4np()` you can use Option 1:

```
# Option 1:  
learnr::run_tutorial("ex_r_basics", package = "r4np")
```

If you have not installed the `r4np` package and/or not run the function `install_r4np()`, you will have to do this first using Option 2:

```
# Option 2:  
## Install 'r4np' package  
devtools::install_github("ddrauber/r4np")  
  
## Install all relevant packages for this book  
r4np::install_r4np()  
  
## Start the tutorial for this chapter  
learnr::run_tutorial("ex_r_basics", package = "r4np")
```

6

Starting your R projects

Every new project likely fills you with enthusiasm and excitement. And it should. You are about to find answers to your research questions, and you hopefully come out more knowledgeable because of it. However, you likely find certain aspects of data analysis less enjoyable. I can think of two:

- Keeping track of all the files my project generates
- Data wrangling

While we cover data wrangling in great detail in the next Chapter (Chapter 7), I would like to share some insights from my work that helped me stay organised and, consequently, less frustrated. The following applies to small and large research projects, which makes it very convenient no matter the situation and the scale of the project. Of course, feel free to tweak my approach to whatever suits you. However, consistency is king.

6.1 Creating an *R* Project file

When working on a project, you likely create many different files for various purposes, especially *R* Scripts (see Section 6.1). If you are not careful, this file is stored in your system's default location, which might not be where you want them to be. RStudio allows you to manage your entire project intuitively and conveniently through *R* Project files. Using *R* Project files comes with a couple of perks, for example:

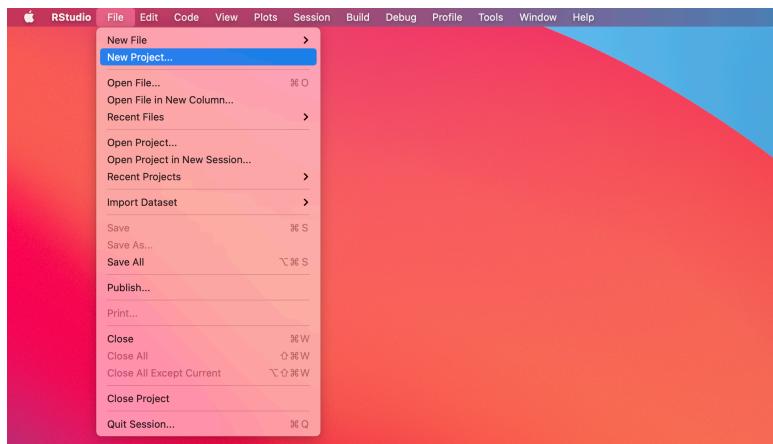
- All the files that you generate are in the same place. Your data, your coding, your exported plots, your reports, etc., all are in one place together without you having to manage the files manually. This is due to the fact that RStudio sets the root directory to whichever folder your project is saved to.
- If you want to share your project, you can share the entire folder, and others can quickly reproduce your research or help fix problems. This is because all file paths are relative and not absolute.
- You can, more easily, use GitHub for backups and so-called ‘version control’,

which allows you to track changes you have made to your code over time (see also Section 15.1).

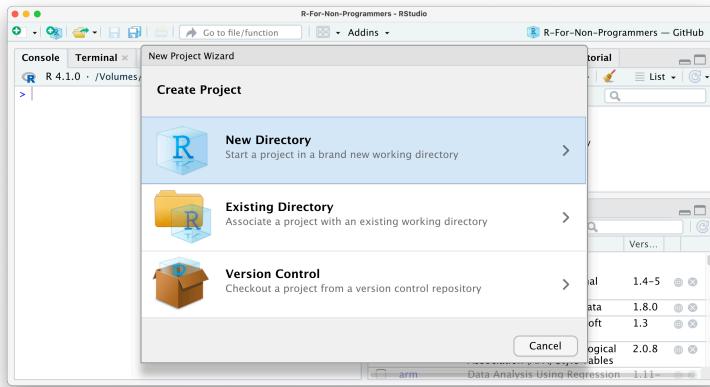
For now, the most important reason to use *R* Project files is the convenience of the organisation of files and the ability to share it easily with co-investigators, your supervisor, or your students.

To create an *R* Project, you need to perform the following steps:

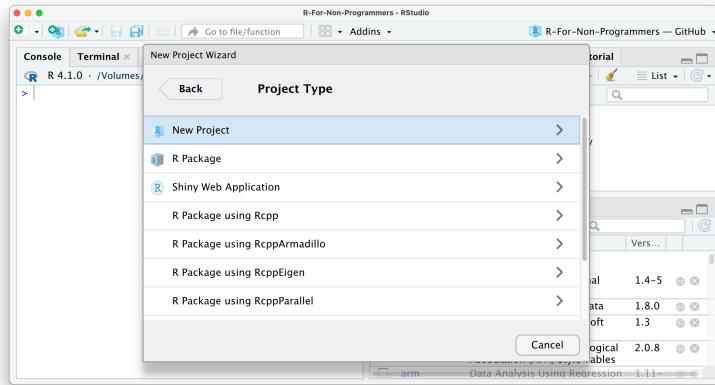
1. Select **File > New Project...** from the menu bar.



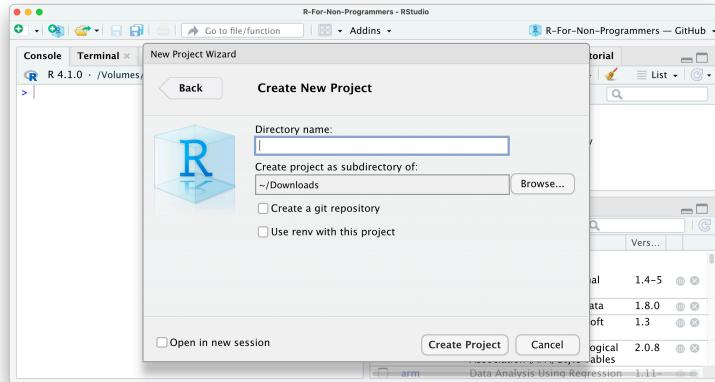
2. Select **New Directory** from the popup window.



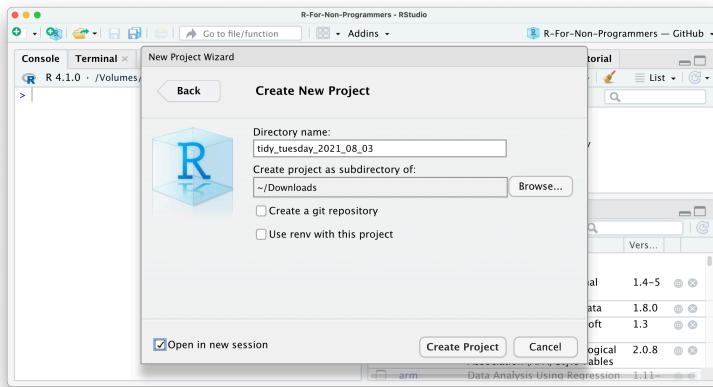
3. Next, select **New Project**.



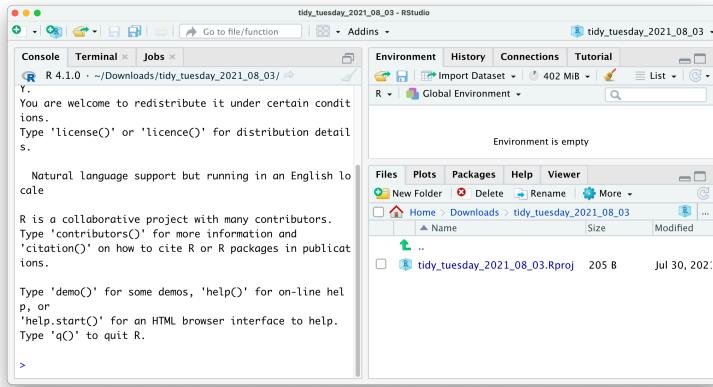
4. Pick a meaningful name for your project folder, i.e. the `Directory Name`. Ensure this project folder is created in the right place. You can change the `subdirectory` by clicking on `Browse...`. Ideally the subdirectory is a place where you usually store your research projects.



5. You have the option to `Create a git repository`. This is only relevant if you already have a GitHub account and wish to use version control. For now, you can happily ignore it if you do not use GitHub.
6. Lastly, tick `Open in new session`. This will open your *R* Project in a new RStudio window.



- Once you are happy with your choices, you can click **Create Project**. This will open a new *R* Session, and you can start working on your project.



If you look carefully, you can see that your RStudio is now ‘branded’ with your project name. At the top of the window, you see the project name, the files pane shows the root directory where all your files will be, and even the console shows on top the file path of your project. You could set all this up manually, but I would not recommend it, not the least because it is easy and swift to work with *R* Projects.

6.2 Organising your projects

This section is not directly related to RStudio, *R* or data analysis in general. Instead, I want to convey to you that a good folder structure can go a long way. It is an excellent habit to start thinking about folder structures before you start working on your project. Placing your files into dedicated folders, rather than keeping them loosely in one container, will speed up your work and save you from the frustration of not finding the files you need. I have a template that I use regularly. You can either create it from scratch in RStudio or open your file browser and create the folders there. RStudio does not mind which way you do it. If you want to spend less time setting this up, you might want to use the function `create_project_folder()` from the `r4np` package. It creates all the folders as shown in Figure 6.1.

```
# Install 'r4np' from GitHub
devtools::install_github("ddrauber/r4np")

# Create the template structure
r4np::create_project_folder()
```

To create a folder, click on `New Folder` in the *Files* pane. I usually have at least the following folders for every project I am involved in:

- A folder for my raw data. I store ‘untouched’ datasets in it. With ‘untouched’, I mean they have not been processed in any way and are usually files I downloaded from my data collection tool, e.g. an online questionnaire platform.
- A folder with ‘tidy’ data. This is usually data I exported from *R* after cleaning it, i.e. after some basic data wrangling and cleaning (see Chapter 7).
- A folder for my *R* scripts
- A folder for my plots
- A folder for reports.

Thus, in RStudio, it would look something like this:

You probably noticed that my folders have numbers in front of them. I do this to ensure that all folders are in the order I want them to be, which is usually not the alphabetical order my computer suggests. I use two digits because I may have more than nine folders for a project, and folder ten would otherwise be listed as the third folder in this list. With this filing strategy in place, it will be easy to find whatever I need. Even others can easily understand what I stored where. It is simply ‘tidy’, similar to how we want our data to be.

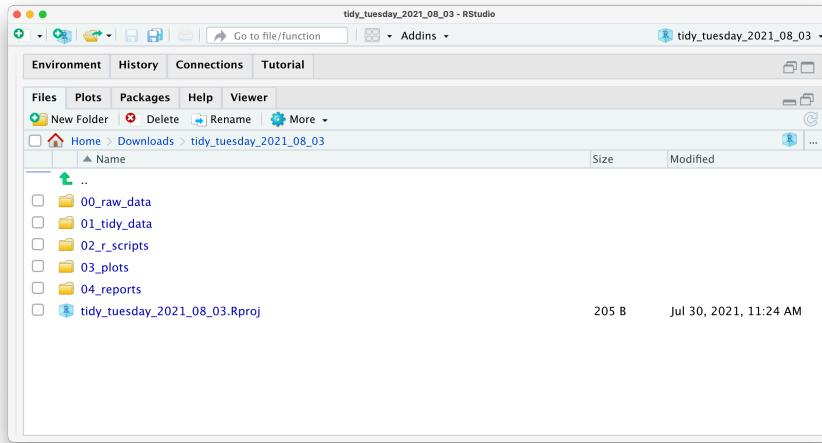


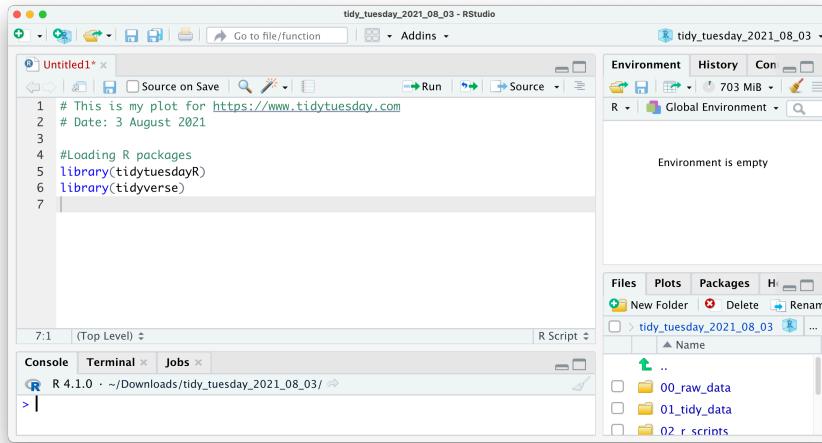
Figure 6.1: An example of a scalable folder structure for your project

6.3 Creating an *R* Script

Code quickly becomes long and complex. Thus, it is not very convenient to write it in the console. So, instead, we can write code into an *R* Script. An *R* Script is a document that RStudio recognises as *R* programming code. Files that are not *R* Scripts, like .txt, .rtf or .md, can also be opened in RStudio, but any code written in it will not be automatically recognised.

When opening an *R* script or creating a new one, it will display in the *Source* window (see Section 4.2) Some refer to this window as the ‘script editor’. An *R* script starts as an empty file. Good coding etiquette (see Section 5.5) demands that we use the first line to indicate what this file does by using a comment #. Here is an example for a ‘TidyTuesday’¹ *R* Project.

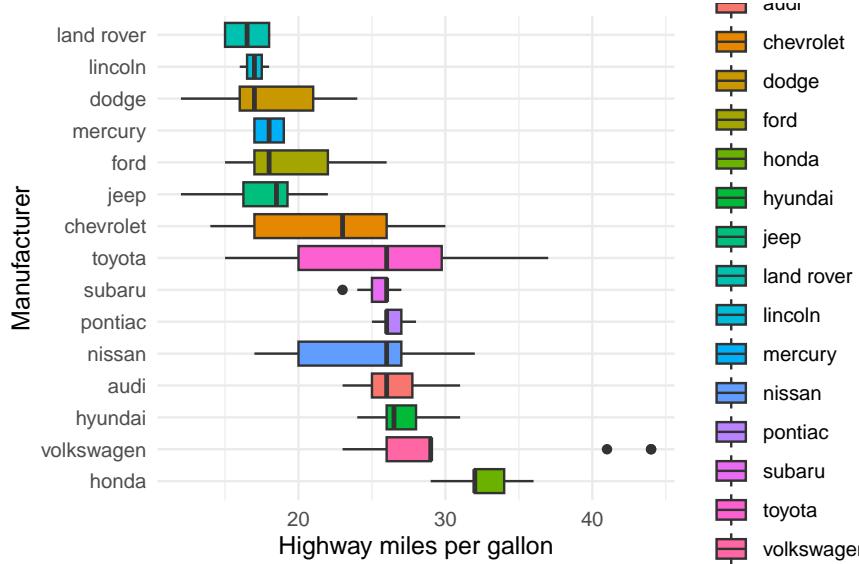
¹<https://www.tidyTuesday.com>



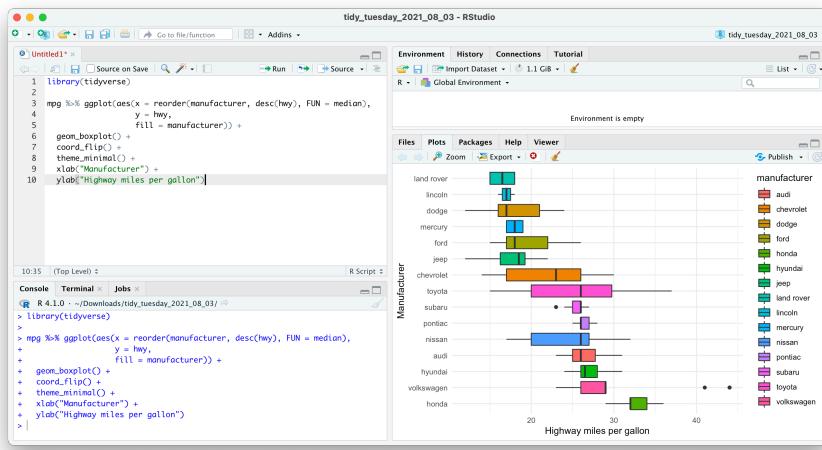
All examples in this book can easily be copied and pasted into your own *R* script. However, for some code you will have to install the *R* package *r4np* (see above). Let's try it with the following code. The plot this code creates reveals which car manufacturer produces the most efficient cars. Copy and paste this code into your *R* script.

```
library(tidyverse)

mpg |> ggplot(aes(x = reorder(manufacturer, desc(hwy), FUN = median),
                     y = hwy,
                     fill = manufacturer)) +
  geom_boxplot() +
  coord_flip() +
  theme_minimal() +
  xlab("Manufacturer") +
  ylab("Highway miles per gallon")
```



You are probably wondering where your plot has gone. Copying the code will not automatically run it in your *R* script. However, this is necessary to create the plot. If you tried pressing **Return ↵**, you would only add a new line. Instead, you need to select the code you want to run and press **Ctrl+Return ↵** (PC) or **Cmd+Return ↵** (Mac). You can also use the **Run** command at the top of your source window, but it is much more efficient to press the keyboard shortcut. Besides, you will remember this shortcut quickly, because we need to use it very frequently. If all worked out, you should see the following:



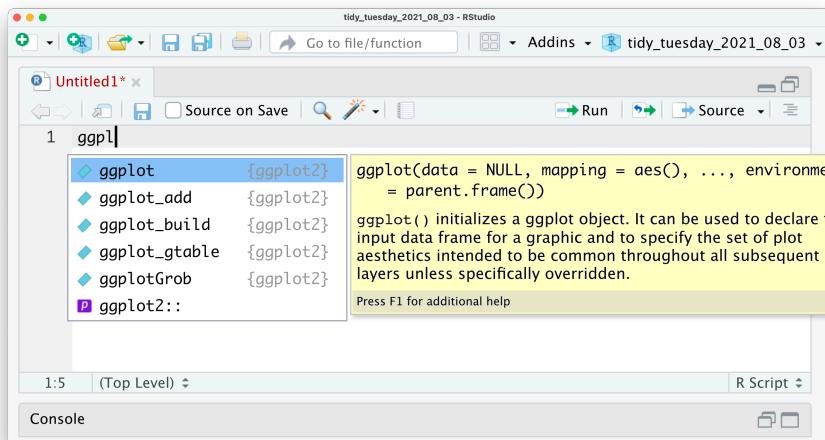
As you can see, cars from Honda appear to drive furthest with the same amount

of fuel (a gallon) compared to other vehicles. Thus, if you are looking for very economical cars, you now know where to find them.

The *R* script editor has some conveniences for writing your code that are worth pointing out. You probably noticed that some of the code we have pasted is blue, and some other code is in green. These colours help to make your code more readable because they carry a specific meaning. In the default settings, green stands for any values in "", which usually stands for characters. Automatic colouring of our programming code is also called '*syntax highlighting*'.

Moreover, code in *R* scripts will be automatically indented to facilitate reading. If, for whatever reason, the indentation does not happen, or you accidentally undo it, you can reindent a line with **Ctrl+I** (PC) or **Cmd+I** (Mac).

Lastly, the console and the *R* script editor both feature code completion. This means that when you start typing the name of a function, *R* will provide suggestions. These are extremely helpful and make programming a lot faster. Once you found the function you were looking for, you press **Return ↵** to insert it. Here is an example of what happens when you have the package `tidyverse` loaded and type `ggpl`. Only functions that are loaded via packages or any object in your environment pane benefit from code completion.



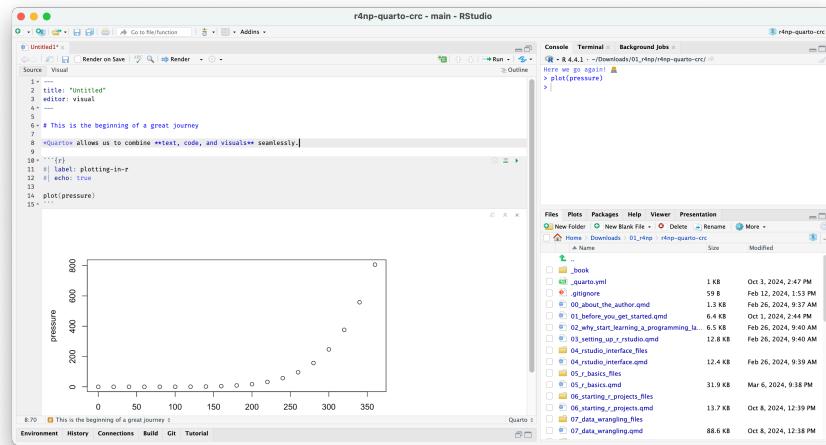
Not only does RStudio show you all the available options, but it also tells you which package this function is from. In this case, all listed functions are from the `ggplot2` package. Furthermore, when you select one of the options but have not pressed **Return ↵** yet, you also get to see a yellow box, which provides you with a quick reference of all the arguments that this function accepts. So you do not have to memorise all the functions and their arguments.

6.4 Using Quarto

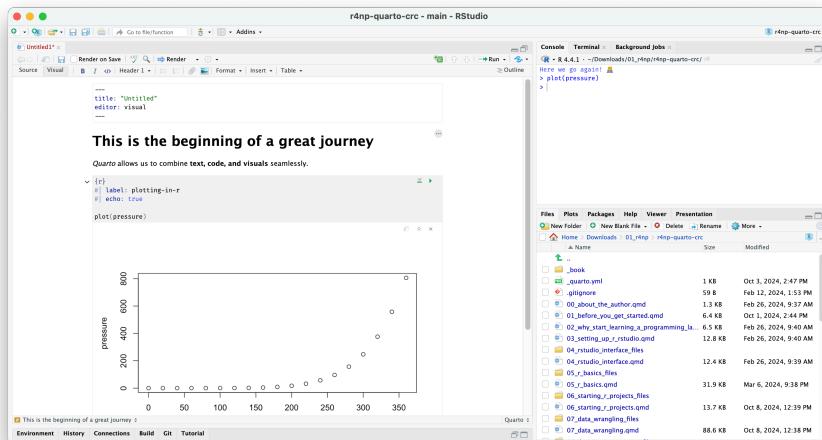
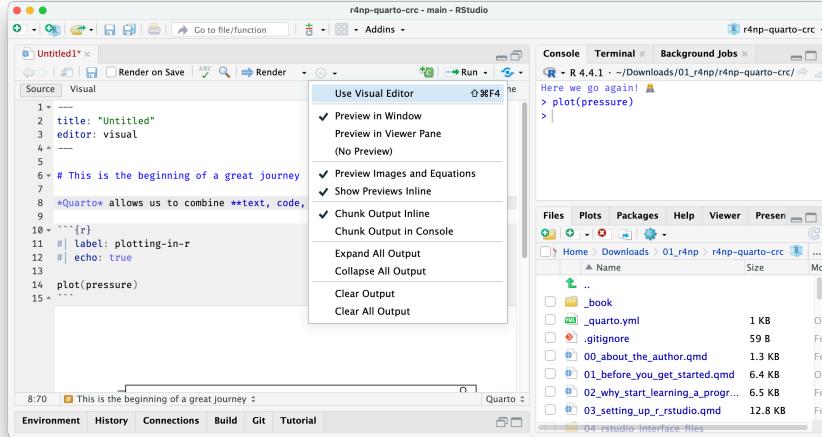
There is too much to say about *Quarto*, which is why I only will highlight that it exists and point out the one feature that might convince you to choose this format over plain *R* scripts: They look like a Word document (almost).

The predecessor of *Quarto* was called *R Markdown*, which is a combination of *R* and '*Markdown*'. '*Markdown*' is a way of writing and formatting text documents without needing software like MS Word. Instead, you write everything in plain text. Such plain text can be converted into many different document types such as HTML websites, PDF or Word documents. If you would like to see how it works, I recommend looking at the *Quarto* Website. To create a *Quarto* file, click on **File > New File > Quarto Document....**

A *Quarto* file works oppositely to an *R* script. By default, an *R* script considers everything as code and only through commenting `#` we can include text to describe what the code does. This is what you have seen in all the coding examples so far. On the other hand, a *Quarto* file considers everything as text, and we have to specify what is code. We can do so by inserting '*code chunks*'. Therefore, there is less of a need to use comments `#` in *Quarto* files because you can write about it like you would in MS Word. Another convenience of *Quarto* files is that results from your analysis are immediately shown underneath the code chunk and not in the console.



If you switch your view to the **Visual Editor** it almost looks like you are writing a report in MS Word. In order to switch to the **Visual Editor** you need to either click on **visual** in the top left or click on the gear icon and select **Use Visual Editor**.



So, when should you use an *R* script, and when should you use *Quarto*. The rule-of-thumb is that if you intend to write a report, assignment, thesis or another form of publication, it might be better to work in a *Quarto* file. If this does not apply, you might want to use an *R* Script. As mentioned above, *Quarto* files emphasise writing about your analysis, while *R* scripts primarily focus on code. In my projects, I often have a mixture of both. I use *R* Scripts to carry out data wrangling and my primary analysis and then use *Quarto* files to present the findings, e.g. creating plots, tables, and other components of a report. By the way, this book is written in *Quarto*.

No matter your choice, it will neither benefit nor disadvantage you in your *R*

journey or when working through this book. The choice is all yours. You will likely come to appreciate both formats for what they offer.

7

Data Wrangling

Data wrangling — also called **data cleaning**, **data remediation**, or **data munging**—refers to a variety of processes designed to transform raw data into more readily used formats. The exact methods differ from project to project depending on the data you’re leveraging and the goal you’re trying to achieve. (Stobierski 2021)

You collected your data over months (and sometimes years), and all you want to know is whether your data makes sense and reveals something nobody would have ever expected. However, before we can truly go ahead with our analysis, it is essential to understand whether our data is ‘tidy’. What I mean is that our data is at the quality we would expect it to be and can be reliably used for data analysis. After all, the results of our research are only as good as our data and the methods we deploy to achieve the desired insights.

Very often, the data we receive is everything else but clean, and we need to check whether our data is fit for analysis and ensure it is in a format that is easy to handle. For small datasets, this is usually a brief exercise. However, I found myself cleaning data for a month because the dataset was spread out into multiple spreadsheets with different numbers of columns and odd column names that were not consistent. Thus, data wrangling is an essential first step in any data analysis. It is a step that cannot be skipped and has to be performed on every new dataset. If your background is in qualitative research, you likely conducted interviews which you had to transcribe. The process of creating transcriptions is comparable to data wrangling in quantitative datasets - and equally tedious.

Luckily, *R* provides many useful functions to make our lives easier. You will be in for a treat if you are like me and used to do this in Excel. It is a lot simpler using *R* to achieve a clean dataset.

Here is an overview of the different steps we usually work through before starting with our primary analysis. This list is certainly not exhaustive:

- Import data,
 - Check data types,
 - Recode and arrange factors, i.e. categorical data, and
 - Run missing data diagnostics.
-

7.1 Import your data

The `r4np` package hosts several different datasets to work with, but at some point, you might want to apply your *R* knowledge to your own data. Therefore, an essential first step is to import your data into RStudio. There are three different methods, all of which are very handy:

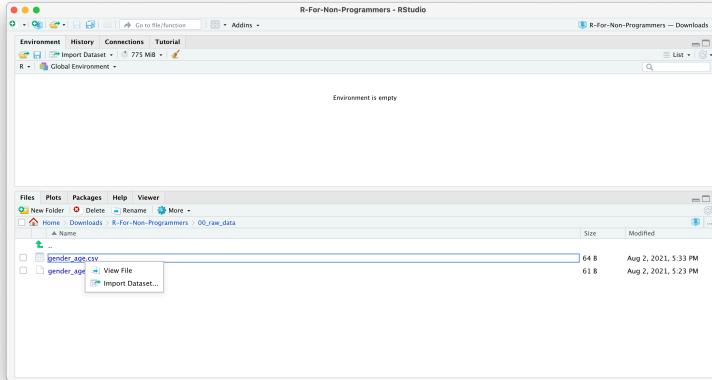
1. Click on your data file in the *Files* pane and choose **Import Dataset**.
2. Use the **Import Dataset** button in the *Environment* pane.
3. Import your data calling one of the `readr` functions in the *Console* or *R* script.

We will use the `readr` package for all three options. Using this package we can import a range of different file formats, including `.csv`, `.tsv`, `.txt`. If you want to import data from an `.xlsx` file, you need to use another package called `readxl`. Similarly, if you used SPSS, Stata or SAS files before and want to use them in *R* you can use the `haven` package. The following sections will primarily focus on using `readr` via RStudio or directly in your *Console* or *R* script.

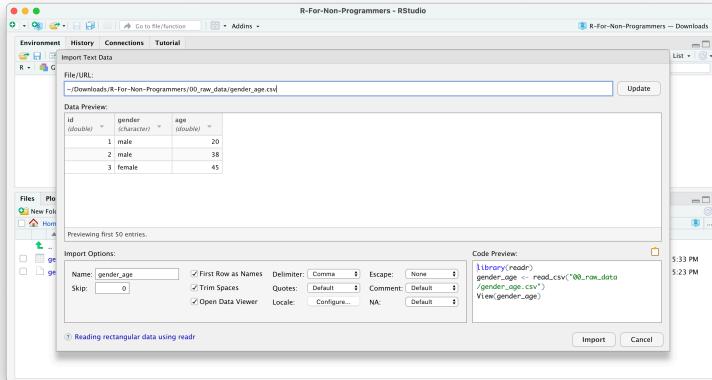
7.1.1 Import data from the *Files* pane

This approach is by far the easiest. Let's assume you have a dataset called `gender_age.csv` in your `00_raw_data` folder. If you wish to import it, you can do the following:

1. Click on the name of the file.
2. Select **Import Dataset**.



3. A new window will open, and you can choose different options. You also see a little preview of how the data looks like. This is great if you are not sure whether you did it correctly.

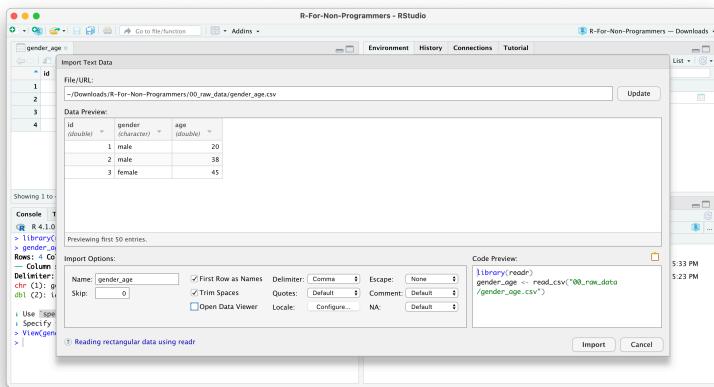


4. You can change how data should be imported, but the default should be fine in most cases. Here is a quick breakdown of the most important options:

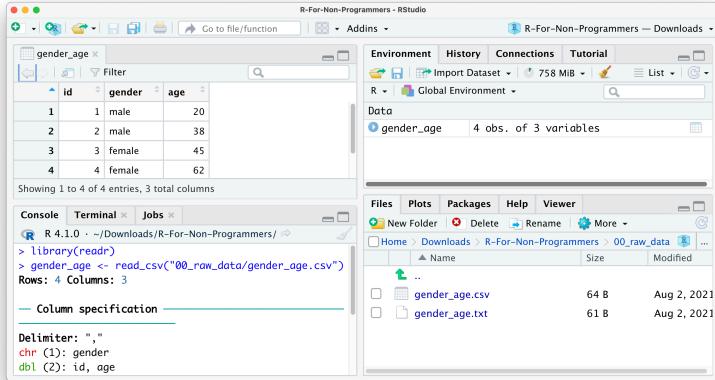
- **Name** allows you to change the object name, i.e. the name of the object this data will be assigned to. I often use `df_raw` (`df` stand for `data frame`, which is how *R* calls such rectangular datasets).
- **Skip** is helpful if your data file starts with several empty rows at the top. You can remove them here.
- **First Row as Names** is ticked by default. In most Social Science

projects, we tend to have the name of the variables as the first row in our dataset.

- **Trim Spaces** removes any unnecessary whitespace in your dataset. Leave it ticked.
- **Open Data Viewer** allows you to look at your imported dataset. I use it rarely, but it can be helpful at times.
- **Delimiter** defines how your columns are separated from each other in your file. If it is a .csv file, it would imply it is a ‘comma-separated value’, i.e.,. This setting can be changed for different files, depending on how your data is delimited. You can even use the option **Other...** to specify a custom separation option.
- **NA** specifies how missing values in your data are acknowledged. By default, empty cells in your data will be recognised as missing data.



5. Once you are happy with your choices, you can click on **Import**.
6. You will find your dataset in the *Environment* pane.



In the *Console*, you can see that *R* also provides the *Column specification*, which we need later when inspecting ‘data types’. *readr* automatically imports all text-based columns as *chr*, i.e. character values. However, this might not always be the correct data type. We will cover more of this aspect of data wrangling in Section 7.4.

7.1.2 Importing data from the *Environment* pane

The process of importing datasets from the *Environment* pane follows largely the one from the *Files* pane. Click on **Import Dataset** > **From Text** (*readr*).... The only main difference lies in having to find the file using the **Browse...** button. The rest of the steps are the same as above.

Be aware that you will have to use the *Environment* pane for importing data from specific file types, e.g. *.txt*. The *File* pane would only open the file but not import the data for further processing.

7.1.3 Importing data using functions directly

If you organised your files well, it could be effortless and quick to use all the functions from *readr* directly. Here are two examples of how you can use *readr* to import your data. Make sure you have the package loaded.

```
# Import data from '.csv'
read_csv("00_raw_data/gender_age.csv")

# Import data from any text file by defining the separator
read_delim("00_raw_data/gender_age.txt", delim = "|")
```

You might be wondering whether you can use *read_delim()* to import *.csv*

files too. The answer is ‘Yes, you can!’. In contrast to `read_delim()`, `read_csv()` sets the delimiter to `,` by default. This is mainly for convenience because `.csv` files are one of the most popular file formats used to store data.

You might also be wondering what a ‘*delimiter*’ is. When you record data in a plain-text file, it is easy to see where a new observation starts and ends because it is defined by a row in your file. However, we also need to tell our software where a new column starts, i.e. where a cell begins and ends. Consider the following example. We have a file that holds our data which looks like this:

```
id age gender
124 male
256 female
333 male
```

The first row we probably can still decipher as `id`, `age`, `gender`. However, the next row makes it difficult to understand which value represents the `id` of a participant and which value reflects the `age` of that participant. Like us, computer software would find it hard too to decide on this ambiguous content. Thus, we need to use delimiters to make it very clear which value belongs to which column. For example, in a `.csv` file, the data would be separated by a `,`

```
id,age,gender
1,24,male
2,56,female
3,33,male
```

Considering our example from above, we could also use `|` as a delimiter, but this is very unusual.

```
id|age|gender
1|24|male
2|56|female
3|33|male
```

An important advantage of this approach is the ability to load large data files much more quickly. The previous options offer a preview of your data, but loading this preview can take a long time for large datasets. Therefore, I highly recommend using the `readr` functions directly in the console or your R script to avoid frustration when importing large datasets.

There is a lot more to `readr` than could be covered in this book. If you want to know more about this *R* package, I highly recommend looking at the `readr` webpage¹.

¹<https://readr.tidyverse.org>

7.2 Inspecting your data

For the rest of this chapter, we will use the World Value Survey data (Haerpfer et al. 2022) (`wvs`) from the `r4np` package. However, we do not know much about this dataset, and therefore we cannot ask any research questions worth investigating. Therefore, we need to look at what it contains. The first method of inspecting a dataset is to type the name of the object, i.e. `wvs`.

```
# Ensure you loaded the 'r4np' package first
library(r4np)

# Show the data in the console
wvs
```

	<code>'Participant ID'</code>	<code>'Country name'</code>	<code>Gender</code>	<code>Age</code>	<code>relationship_status</code>
	<code><dbl></code>	<code><chr></code>	<code><dbl></code>	<code><dbl></code>	<code><chr></code>
1	68070001	Bolivia, Plurinational Sta~	0	60	married
2	68070002	Bolivia, Plurinational Sta~	0	40	married
3	68070003	Bolivia, Plurinational Sta~	0	25	single
4	68070004	Bolivia, Plurinational Sta~	1	71	widowed
5	68070005	Bolivia, Plurinational Sta~	1	38	married
6	68070006	Bolivia, Plurinational Sta~	1	20	separated
7	68070007	Bolivia, Plurinational Sta~	0	39	single
8	68070008	Bolivia, Plurinational Sta~	1	19	single
9	68070009	Bolivia, Plurinational Sta~	0	20	single
10	68070010	Bolivia, Plurinational Sta~	1	37	married
# i 8,554 more rows					
# i 2 more variables: <code>Freedom.of.Choice</code> <code><dbl></code> , <code>'Satisfaction-with-life'</code> <code><dbl></code>					

The result is a series of rows and columns. The first information we receive is: `A tibble: 69,578 x 7`. This indicates that our dataset has 69,578 observations (i.e. rows) and 9 columns (i.e. variables). This rectangular format is the one we encounter most frequently in Social Sciences (and probably beyond). If you ever worked in Microsoft Excel, this format will look familiar. However, rectangular data is not necessarily `tidy` data. Wickham (2014) (p. 4) defines `tidy` data as follows:

-
1. Each variable forms a column.

2. Each observation forms a row.
 3. Each type of observational unit forms a table.
-

Even though it might be nice to look at a tibble in the console, it is not particularly useful. Depending on your monitor size, you might only see a small number of columns, and therefore we do not get to see a complete list of all variables. In short, we hardly ever will find much use in inspecting data this way. Luckily other functions can help us.

If you want to see each variable covered in the dataset and their data types, you can use the function `glimpse()` from the `dplyr` package which is loaded as part of the `tidyverse` package.

```
glimpse(wvs)
```

```
Rows: 8,564
Columns: 7
$ `Participant ID`      <dbl> 68070001, 68070002, 68070003, 68070004, 68070~
$ `Country name`        <chr> "Bolivia, Plurinational State of", "Bolivia, ~
$ Gender                 <dbl> 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, ~
$ Age                   <dbl> 60, 40, 25, 71, 38, 20, 39, 19, 20, 37, 42, 7~
$ relationship_status    <chr> "married", "married", "single", "widowed", "m~
$ Freedom.of.Choice     <dbl> 7, 8, 10, 5, 5, 5, 10, 6, 9, 5, 8, 8, 7, 5~
$ `Satisfaction-with-life` <dbl> 5, 7, 8, 5, 7, 3, 3, 8, 7, 8, 9, 6, 7, 7, 4, ~
```

The output of `glimpse` shows us the name of each column/variable after the `$`, for example, ``Participant ID``. The `$` is used to lookup certain variables in our dataset. For example, if we want to inspect the column `relationship_status` only, we could write the following:

```
wvs$relationship_status
```

```
[1] "married"                  "married"
[3] "single"                   "widowed"
[5] "married"                  "separated"
[7] "single"                   "single"
....
```

When using `glimpse()` we find the recognised data type for each column, i.e. each variable, in `<...>`, for example `<chr>`. We will return to data types in Section 7.4. Lastly, we get examples of the values included in each variable. This output is much more helpful.

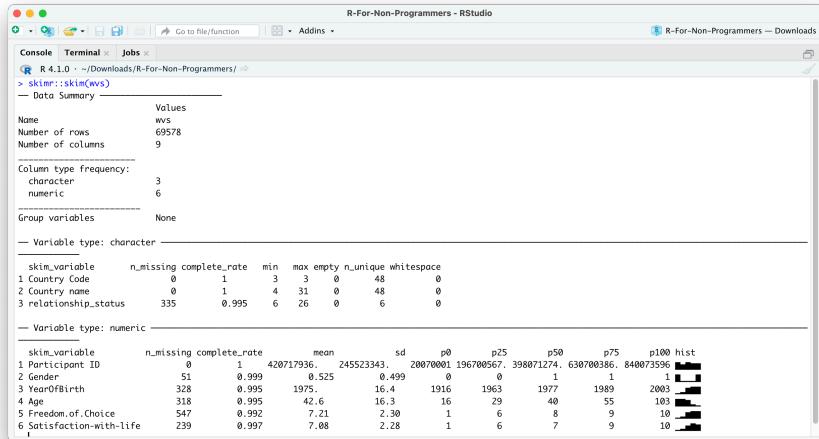
I use `glimpse()` very frequently for different purposes, for example:

- to understand what variables are included in a dataset,
- to check the correctness of data types,
- to inspect variable names for typos or unconventional names,
- to look up variable names.

There is one more way to inspect your data and receive more information about it by using a specialised *R* package. The `skimr` package is excellent in ‘skimming’ your dataset. It provides not only information about variable names and data types but also provides some descriptive statistics. If you installed the `r4np` package and called the function `install_r4np()`, you will have `skimr` installed already.

```
skimr::skim(wvs)
```

The output in the *Console* should look like this:



As you can tell, there is a lot more information in this output. Many descriptive statistics that could be useful are already displayed. `skim()` provides a summary of the dataset and then automatically sorts the variables by data type. Depending on the data type, you also receive different descriptive statistics. As a bonus, the function also provides a histogram for numeric variables. However, there is one main problem: Some of the numeric variables are not numeric: `Participant ID` and `Gender`. Thus, we will have to correct the data types in a moment.

Inspecting your data in this way can be helpful to get a better understanding of

what your data includes and spot problems with it. In addition, if you receive data from someone else, these methods are an excellent way to familiarise yourself with the dataset relatively quickly. Since I prepared this particular dataset for this book, I also made sure to provide documentation for it. You can access it by using `?wvs` in the *Console*. This will open the documentation in the *Help* pane. Such documentation is available for every dataset we use in this book.

7.3 Cleaning your column names: Call the `janitor`

If you have eagle eyes, you might have noticed that most of the variable names in `wvs` are not consistent or easy to read/use.

```
# Whitespace and inconsistent capitalisation
Participant ID
Country name
Gender
Age

# Difficult to read
Freedom.of.Choice
Satisfaction-with-life
```

From Section 5.5, you will remember that being consistent in writing your code and naming your objects is essential. The same applies, of course, to variable names. *R* will not break using the existing names, but it will save you a lot of frustration if we take a minute to clean the names and make them more consistent. Since we will use column names in almost every line of code we produce, it is best to standardise them and make them easier to read and write.

You are probably thinking: “This is easy. I just open the dataset in Excel and change all the column names.” Indeed, it would be a viable and easy option, but it is not very efficient, especially with larger datasets with many more variables. Instead, we can make use of the `janitor` package. By definition, `janitor` is a package that helps to clean up whatever needs cleaning. In our case, we want to tidy our column names. We can use the function `clean_names()` to achieve this. We store the result in a new object called `wvs` to keep those changes. The object will also show up in our *Environment* pane.

```
wvs_clean <- janitor::clean_names(wvs)

glimpse(wvs_clean)
```

Rows: 8,564
 Columns: 7
\$ participant_id <dbl> 68070001, 68070002, 68070003, 68070004, 6807000~
\$ country_name <chr> "Bolivia, Plurinational State of", "Bolivia, Pl~
\$ gender <dbl> 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, ~
\$ age <dbl> 60, 40, 25, 71, 38, 20, 39, 19, 20, 37, 42, 73, ~
\$ relationship_status <chr> "married", "married", "single", "widowed", "mar~
\$ freedom_of_choice <dbl> 7, 8, 10, 5, 5, 5, 10, 6, 9, 5, 8, 8, 7, 5, ~
\$ satisfaction_with_life <dbl> 5, 7, 8, 5, 7, 3, 3, 8, 7, 8, 9, 6, 7, 7, 4, 8, ~

Now that `janitor` has done its magic, we suddenly have easy to read variable names that are consistent with the '*Tidyverse style guide*' (Wickham 2021).

If, for whatever reason, the variable names are still not looking the way you want, you can use the function `rename()` from the `dplyr` package to manually assign new variable names.

```
wvs_clean <-
  wvs_clean |>
  rename(satisfaction = satisfaction_with_life,
         country = country_name)

glimpse(wvs_clean)
```

Rows: 8,564
 Columns: 7
\$ participant_id <dbl> 68070001, 68070002, 68070003, 68070004, 68070005, ~
\$ country <chr> "Bolivia, Plurinational State of", "Bolivia, Pluri~
\$ gender <dbl> 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, ~
\$ age <dbl> 60, 40, 25, 71, 38, 20, 39, 19, 20, 37, 42, 73, 32~
\$ relationship_status <chr> "married", "married", "single", "widowed", "marrie~
\$ freedom_of_choice <dbl> 7, 8, 10, 5, 5, 5, 10, 6, 9, 5, 8, 8, 7, 5, 10, ~
\$ satisfaction <dbl> 5, 7, 8, 5, 7, 3, 3, 8, 7, 8, 9, 6, 7, 7, 4, 8, 5, ~

You are probably wondering what `|>` stands for. This symbol is called a '*piping operator*', and it allows us to chain multiple functions together by considering the output of the previous function. So, do not confuse `<-` with `|>`. Each operator serves a different purpose. The `|>` has become synonymous with the `tidyverse` approach to *R* programming and is the chosen approach for this book. Many functions from the `tidyverse` are designed to be chained together.

If we wanted to spell out what we just did, we could say:

1. `wvs <-:` We assigned whatever happened to the right of the assignment operator to the object `wvs`.
2. `wvs |>:` We defined the dataset we want to use with the functions defined after the `|>`.
3. `rename(satisfaction = satisfaction_with_life):` We define a new name `satisfaction` for the column `satisfaction_with_life`. Notice that the order is `new_name = old_name`. Here we also use `=`. A rare occasion where it makes sense to do so.

Just for clarification, the following two lines of code accomplish the same task. The only difference is that with `|>` we could chain another function right after it. So, you could say, it is a matter of taste which approach you prefer. However, in later chapters, it will become apparent why using `|>` is very advantageous.

```
# Renaming a column using '|>'  
wvs_clean |> rename(satisfaction_new = satisfaction)  
  
# Renaming a column without '|>'  
rename(wvs_clean, satisfaction_new = satisfaction)
```

Since you will be using the pipe operator very frequently, it is a good idea to remember the keyboard shortcut for it: **Ctrl+Shift+M** for PC and **Cmd+Shift+M** for Mac.

Besides `|>` operator you might also find that some sources use a different pipe operator `%>%`. The original `tidyverse` approach used `%>%` before R introduced their own pipe operator. In short: You can use both of them if you like, but in 90% of cases, you probably will be fine using the one that comes shipped with RStudio by default, i.e. `|>`. More and more R resources adopt the `|>` operator and there are only few situations where `%>%` is preferred. For the content of this book and the techniques demonstrated here, you can use both operators interchangeably and the code will work just fine.

7.4 Data types: What are they and how can you change them

When we inspected our data, I mentioned that some variables do not have the correct data type. You might be familiar with different data types by classifying them as:

- *Nominal data*, which is categorical data of no particular order,

- *Ordinal data*, which is categorical data with a defined order, and
- *Quantitative data*, which is data that usually is represented by numeric values.

In *R* we have a slightly different distinction:

- `character` / `<chr>`: Textual data, for example the text of a tweet.
- `factor` / `<fct>`: Categorical data with a finite number of categories with no particular order.
- `ordered` / `<ord>`: Categorical data with a finite number of categories with a particular order.
- `double` / `<dbl>`: Numerical data with decimal places.
- `integer` / `<int>`: Numerical data with whole numbers only (i.e. no decimals).
- `logical` / `<lgl>`: Logical data, which only consists of values `TRUE` and `FALSE`.
- `date` / `date`: Data which consists of dates, e.g. `2021-08-05`.
- `date-time` / `dttm`: Data which consists of dates and times, e.g. `2021-08-05 16:29:25 BST`.

For a complete list of data types, I recommend looking at ‘Column Data Types’² (Müller and Wickham 2021).

R has a fine-grained categorisation of data types. The most important distinction, though, lies between `<chr>`, `<fct>/<ord>` and `<dbl>` for most datasets in the Social Sciences. Still, it is good to know what the abbreviations in your `tibble` mean and how they might affect your analysis.

Now that we have a solid understanding of different data types, we can look at our dataset and see whether `readr` classified our variables correctly.

```
glimpse(wvs_clean)
```

```
Rows: 8,564
Columns: 7
$ participant_id    <dbl> 68070001, 68070002, 68070003, 68070004, 68070005, ~
$ country          <chr> "Bolivia, Plurinational State of", "Bolivia, Pluri~
$ gender            <dbl> 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, ~
$ age               <dbl> 60, 40, 25, 71, 38, 20, 39, 19, 20, 37, 42, 73, 32~
$ relationship_status <chr> "married", "married", "single", "widowed", "marrie~
$ freedom_of_choice   <dbl> 7, 8, 10, 5, 5, 5, 10, 6, 9, 5, 8, 8, 7, 5, 10, ~
$ satisfaction       <dbl> 5, 7, 8, 5, 7, 3, 3, 8, 7, 8, 9, 6, 7, 7, 4, 8, 5, ~
```

²<https://tibble.tidyverse.org/articles/types.html>

`readr` did a great job in identifying all the numeric variables. However, by default, `readr` imports all variables that include text as `<chr>`. It appears as if this is not entirely correct for our dataset. The variables `country`, `gender` and `relationship_status` specify a finite number of categories. Therefore they should be classified as a `factor`. The variable `participant_id` is represented by numbers, but its meaning is also rather categorical. We would not use the ID numbers of participants to perform additions or multiplications. This would make no sense. Therefore, it might be wise to turn them into a `factor`, even though we likely will not use it in our analysis and would make no difference. However, I am a stickler for those kinds of things.

To perform the conversion, we need to use two new functions from `dplyr`:

- `mutate()`: Changes, i.e. ‘mutates’, a variable.
- `as_factor()`: Converts data from one type into a `factor`.

If we want to convert all variables in one go, we can put them into the same function, separated by a `,`.

```
wvs_clean <-  
  wvs_clean |>  
  mutate(country = as_factor(country),  
         gender = as_factor(gender),  
         relationship_status = as_factor(relationship_status),  
         participant_id = as_factor(participant_id)  
  )  
  
glimpse(wvs_clean)
```

Rows: 8,564
 Columns: 7
\$ participant_id <fct> 68070001, 68070002, 68070003, 68070004, 68070005, ~
\$ country <fct> "Bolivia, Plurinational State of", "Bolivia, Pluri~
\$ gender <fct> 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, ~
\$ age <dbl> 60, 40, 25, 71, 38, 20, 39, 19, 20, 37, 42, 73, 32~
\$ relationship_status <fct> married, married, single, widowed, married, separa~
\$ freedom_of_choice <dbl> 7, 8, 10, 5, 5, 5, 10, 6, 9, 5, 8, 8, 7, 5, 10, ~
\$ satisfaction <dbl> 5, 7, 8, 5, 7, 3, 3, 8, 7, 8, 9, 6, 7, 7, 4, 8, 5, ~

The output in the console shows that we successfully performed the transformation and our data types are as we intended them to be. Mission accomplished.

If you need to convert all `<chr>` columns you can use `mutate_if(is.character, as_factor)` instead. This function will look at each column and if it is a `character` type variable, it will convert it into a `factor`. However, use this function

only if you are certain that all `character` columns need converting. If this is the case, it can be a huge time-saver. Here is the corresponding code snippet:

```
wvs_clean |>
  mutate_if(is.character, as_factor)
```

Lastly, there might be occasions where you want to convert data into other formats than from `<chr>` to `<fct>`. Similar to `as_factor()` there are more functions available to transform your data into the correct format of your choice:

- `as.integer()`: Converts values into integers, i.e. without decimal places. Be aware that this function does not round values as you might expect. Instead, it removes any decimals as if they never existed.
- `as.numeric()`: Converts values into numbers with decimals if they are present.
- `as.character()`: Converts values to string values, even if they are numbers. If you use `readr` to import your data, you will not often use this function because `readr` imports unknown data types as `<chr>` by default.

All these function can be used in the same as we did before. Here are three examples to illustrated their use.

```
# Conversion to character values
converted_data <-
  wvs_clean |>
  mutate(participant_id = as.character(participant_id),
         age = as.character(age))

glimpse(converted_data)
```

```
Rows: 8,564
Columns: 7
$ participant_id    <chr> "68070001", "68070002", "68070003", "68070004", "6~
$ country          <fct> "Bolivia, Plurinational State of", "Bolivia, Pluri~
$ gender            <fct> 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, ~
$ age               <chr> "60", "40", "25", "71", "38", "20", "39", "19", "2~
$ relationship_status <fct> married, married, single, widowed, married, separa~
$ freedom_of_choice   <dbl> 7, 8, 10, 5, 5, 5, 10, 6, 9, 5, 8, 8, 7, 5, 10, ~
$ satisfaction       <dbl> 5, 7, 8, 5, 7, 3, 3, 8, 7, 8, 9, 6, 7, 7, 4, 8, 5, ~
```

```
# Conversion to numeric values
converted_data |>
  mutate(age = as.numeric(age))
```

```
# A tibble: 8,564 x 7
  participant_id country   gender age relationship_status freedom_of_choice
  <chr>          <fct>     <fct>  <dbl> <fct>                  <dbl>
1 68070001      Bolivia, P~ 0    60 married                7
2 68070002      Bolivia, P~ 0    40 married                8
3 68070003      Bolivia, P~ 0    25 single                 10
4 68070004      Bolivia, P~ 1    71 widowed                5
5 68070005      Bolivia, P~ 1    38 married                5
6 68070006      Bolivia, P~ 1    20 separated               5
7 68070007      Bolivia, P~ 0    39 single                 5
8 68070008      Bolivia, P~ 1    19 single                 10
9 68070009      Bolivia, P~ 0    20 single                 6
10 68070010     Bolivia, P~ 1   37 married                9
# i 8,554 more rows
# i 1 more variable: satisfaction <dbl>

# Convert numeric values to integers
numeric_values <- tibble(values = c(1.3, 2.6, 3.8))

numeric_values |>
  mutate(integers = as.integer(values))

# A tibble: 3 x 2
  values integers
  <dbl>     <int>
1 1.3       1
2 2.6       2
3 3.8       3
```

7.5 Handling factors

Factors are an essential way to classify observations in our data in different ways. In terms of data wrangling, there are usually at least three steps we take to prepare them for analysis:

- Recoding factors,
- Reordering factor levels, and
- Removing factor levels

7.5.1 Recoding factors

Another common problem we have to tackle when working with data is their representation in the dataset. For example, gender could be measured as `male` and `female`³ or as `0` and `1`. *R* does not mind which way you represent your data, but some other software does. Therefore, when we import data from somewhere else, the values of a variable might not look the way we want. The practicality of having your data represented accurately as what they are, becomes apparent when you intend to create tables and plots.

For example, we might be interested in knowing how many participants in `wvs_clean` were `male` and `female`. The function `count()` from `dplyr` does precisely that. We can sort the results, i.e. `n`, in descending order by including `sort = TRUE`. By default, `count()` arranges our variable `gender` in alphabetical order.

```
wvs_clean |> count(gender, sort = TRUE)
```

```
# A tibble: 2 x 2
  gender     n
  <fct>   <int>
1 1         4348
2 0         4216
```

Now we know how many people were `male` and `female` and how many did not disclose their `gender`. Or do we? The issue here is that you would have to know what `0` and `1` stand for. Surely you would have a coding manual that gives you the answer, but it seems a bit of a complication. For `gender`, this might still be easy to remember, but can you recall the ID numbers for 48 countries?

It certainly would be easier to replace the `0`s and `1`s with their corresponding labels. This can be achieved with a simple function called `fct_recode()` from `forcats`. However, since we ‘mutate’ a variable into something else, we also have to use the `mutate()` function.

```
wvs_clean <-
wvs_clean |>
  mutate(gender = fct_recode(gender,
                            "male" = "0",
                            "female" = "1"))
```

If you have been following along very carefully, you might spot one oddity in this code: “`0`” and “`1`”. You likely recall that in Chapter 5, I mentioned that

³A sophisticated research project would likely not rely on a dichotomous approach to gender, but appreciate the diversity in this regard.

we use "" for character values but not for numbers. So what happens if we run the code and remove "".

```
wvs_clean |>
  mutate(gender = fct_recode(gender,
                             "male" = 0,
                             "female" = 1))

Error in `mutate()`:
i In argument: `gender = fct_recode(gender, male = 0, female = 1)` .
Caused by error in `fct_recode()`:
! Each element of `...` must be a named string.
i Problems with 2 arguments: male and female
```

The error message is easy to understand: `fct_recode()` only expects strings as input and not numbers. R recognises 0 and 1 as numbers, but `fct_recode()` only converts a factor value into another factor value and since we ‘mutated’ gender from a numeric variable to a factor variable, all values of gender are no longer numbers. To refer to a factor level (i.e. one of the categories in our factor), we have to use "". In other words, data types matter and are often a source of problems with your code. Thus, always pay close attention to them. For now, it is important to remember that factor levels are always string values, i.e. text.

Now that we have change the factor levels to our liking, we can rerun our analysis and generate a frequency table for `gender`. This time we receive a much more readable output.

```
wvs_clean |> count(gender)

# A tibble: 2 × 2
  gender     n
  <fct>   <int>
1 male     4216
2 female   4348
```

Another benefit of going through the trouble of recoding your factors is the readability of your plots. For example, we could quickly generate a bar plot based on the above table and have appropriate labels instead of 0 and 1.

```
wvs_clean |>
  count(gender) |>
  ggplot(aes(gender, n)) +
  geom_col()
```

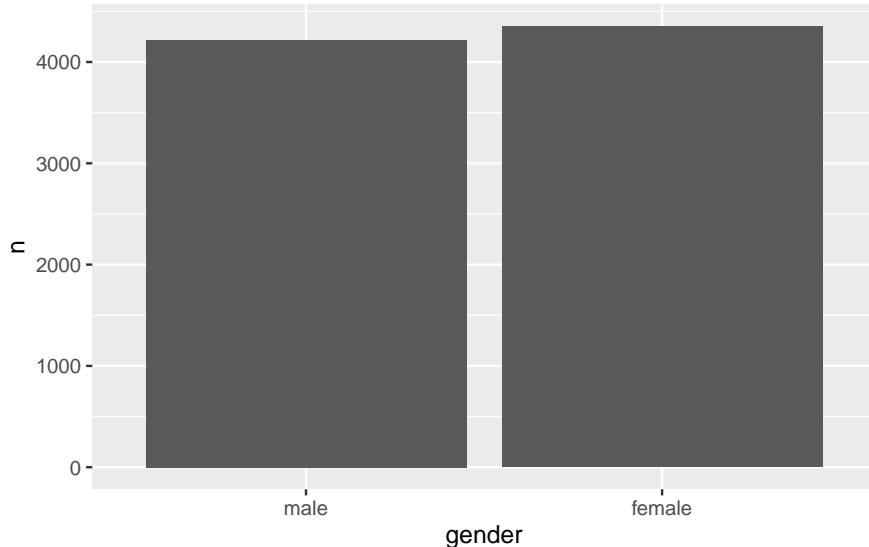


Figure 7.1: A bar plot of the factor levels `male` and `female`

Plots are an excellent way to explore your data and understand relationships between variables. We will learn more about this when we start to perform analytical steps on our data (see Chapter 8 and beyond).

Another use case for recoding factors could be for purely cosmetic reasons. For example, when looking through our dataset, we might notice that some country names are very long and do not look great in data visualisations or tables. Thus, we could consider shortening them which is a slightly more advanced process.

First, we need to find out which country names are particularly long. There are 48 countries in this dataset, so it could take some time to look through them all. Instead, we could use the function `filter()` from `dplyr` to pick only countries with a long name. However, this poses another problem: How can we tell the filter function to pick only country names with a certain length? Ideally, we would want a function that does the counting for us. As you probably anticipated, there is a package called `stringr`, which also belongs to the `tidyverse`, and has a function that counts the number of characters that represent a value in our dataset: `str_length()`. This function takes any `character` variable and returns the length of it. This also works with `factors` because this function can ‘coerce’ it into a `character`, i.e. it just ignores that it is a `factor` and looks at it as if it was a regular `character` variable. Good news for us, because now we can put the puzzle pieces together.

```
wvs_clean |>
  filter(str_length(country) >= 15) |>
  count(country)
```

```
# A tibble: 3 x 2
  country                n
  <fct>                  <int>
1 Bolivia, Plurinational State of 2067
2 Iran, Islamic Republic of     1499
3 Korea, Republic of          1245
```

I use the value 15 arbitrarily after some trial and error. You can change the value and see which other countries would show up with a lower threshold. However, this number seems to do the trick and returns three countries that seem to have longer names. All we have to do is replace these categories with new ones the same way we recoded gender.

```
wvs_clean <- wvs_clean |> mutate(country = fct_recode(country,
  "Bolivia" = "Bolivia, Plurinational State of",
  "Iran" = "Iran, Islamic Republic of",
  "Korea" = "Korea, Republic of"))
```

Now we have country names that nicely fit into tables and plots we want to create later and do not have to worry about it later.

7.5.2 Reordering factor levels

Besides changing factor levels, it is also common that we want to reconsider the arrangement of these levels. To inspect the order of factor levels, we can use the function `fct_unique()` from the `forcats` package and provide a factor variable, e.g. `gender`, in the `wvs_clean` dataset.

```
fct_unique(wvs_clean$gender)
```

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

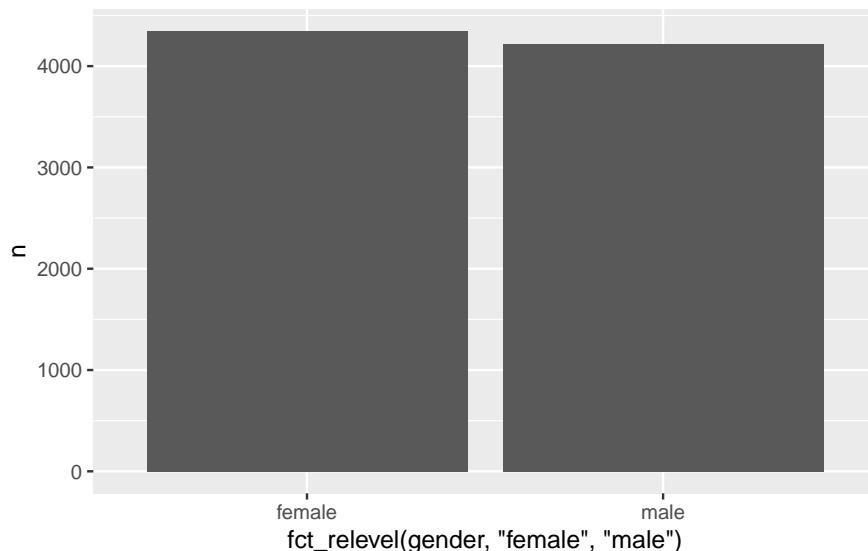
A reason for changing the order of factor levels could be to arrange how factor levels are displayed in plots. For example, Figure 7.1 shows the bar of `male` participants first, but instead, we might want to show `female` participants first. The `forcats` package offers several options to rearrange factor levels based on different conditions. Depending on the circumstances, one or the

other function might be more efficient. Four of these functions⁴ I tend to use fairly frequently:

- `fct_relevel()`: To reorder factor levels by hand.
- `fct_reorder()`: To reorder factor levels based on another variable of interest.
- `fct_rev()`: To reverse the order of factor levels.
- `fct_infreq()`: To reorder factor levels by frequency of appearance in our dataset.

Each function requires slightly different arguments. If we wanted to change the order of our factor levels, we could use all four of them to achieve the same result (in our case):

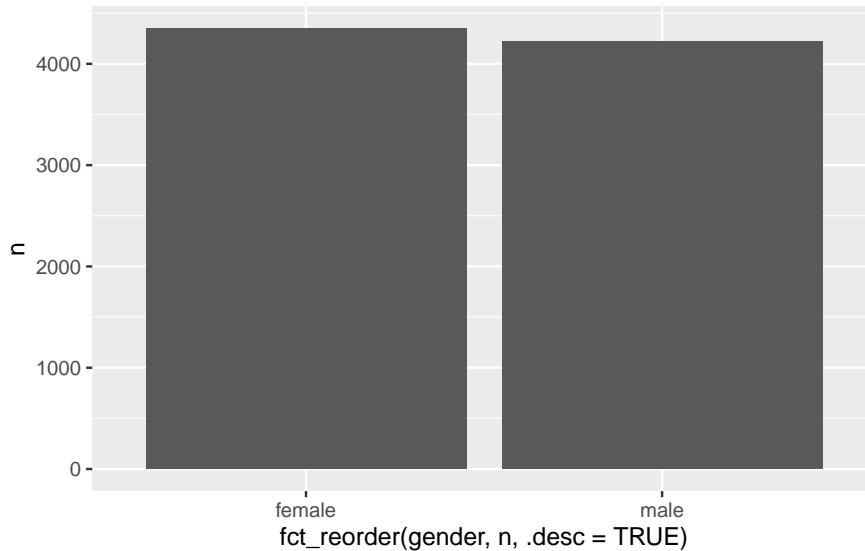
```
# Reorder by hand
wvs_clean |>
  count(gender) |>
  ggplot(aes(x = fct_relevel(gender, "female", "male"),
             y = n)) +
  geom_col()
```



```
# Reorder by another variable, e.g. the frequency 'n'
wvs_clean |>
  count(gender) |>
```

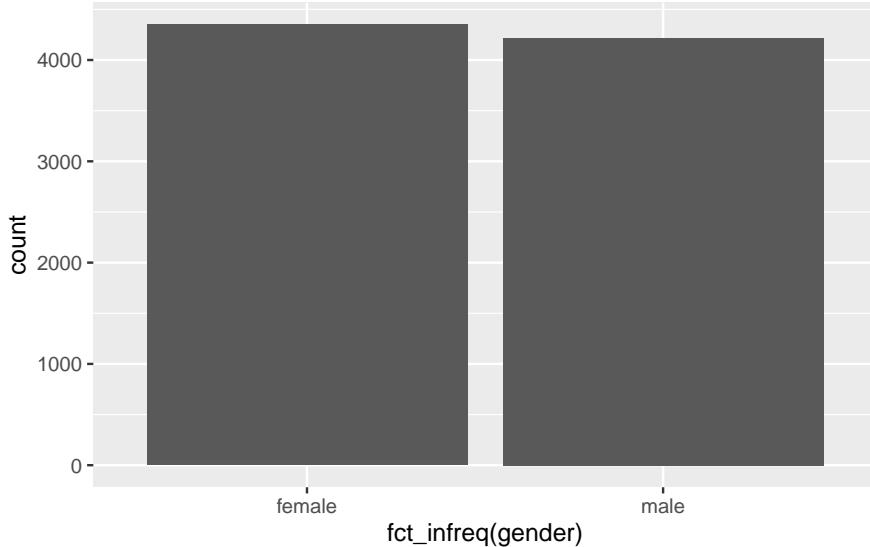
⁴<https://forcats.tidyverse.org/reference/index.html>

```
ggplot(aes(x = fct_reorder(gender, n, .desc = TRUE),  
           y = n)) +  
  geom_col()
```

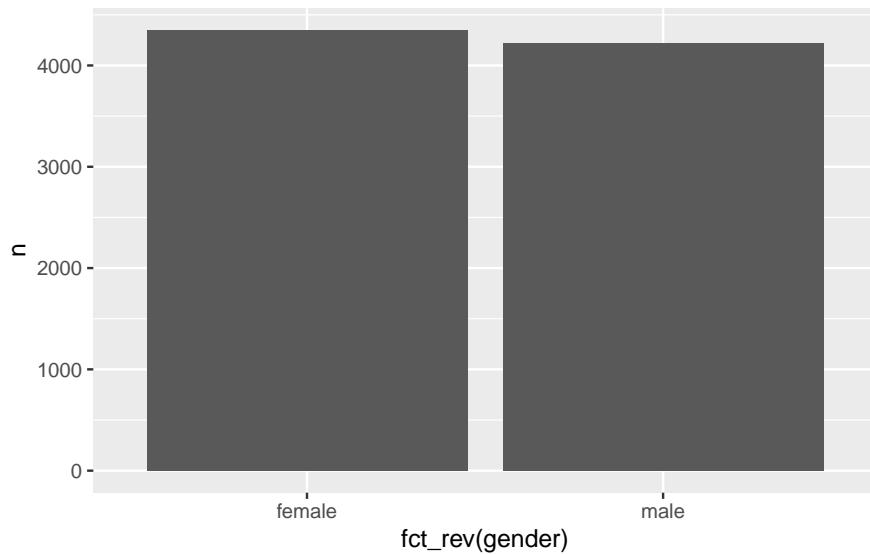


The function `fct_infreq()` is particularly convenient when looking at frequencies, because it will automatically order factor levels according to their size. Thus, we can skip the part where we `count()` factor levels and use `geom_bar()` instead.

```
# Reorder based on frequency of appearance  
wvs_clean |>  
  ggplot(aes(x = fct_infreq(gender))) +  
  geom_bar()
```



```
# Reverse the order of factor levels
wvs_clean |>
  count(gender) |>
  ggplot(aes(x = fct_rev(gender),
             y = n)) +
  geom_col()
```



There are many ways to rearrange factors, and it very much depends on the data you have and what you wish to achieve. For most of my plots, I tend to revert to a basic R function, i.e. `reorder()`, which is much shorter to type and does essentially the same as `fct_reorder()`. There is no right or wrong in this regard, just personal taste.

7.5.3 Removing factor levels

When we work with factors, it can sometimes happen that we have factor levels that are not linked to any of our observations in the dataset, but when we try to perform certain analytical steps they might get in the way (for example in Section 13.3.1). Therefore it is important to perform some housekeeping and remove unneeded factor levels.

```
# A dataset which contains a factor with 4 levels
data <- tibble(gender = factor(c("female",
                                "male",
                                "non-binary"),

                                # Setting the factor levels
                                levels = c("female",
                                           "male",
                                           "non-binary",
                                           "unknown"))
                )
            )

# A tibble: 3 × 1
  gender
  <fct>
  1 female
  2 male
  3 non-binary
```

The `data` dataframe contains the variable `gender` which has four levels, but only three levels are represented in the data, i.e. the factor level `unknown` matches none of our observations in the dataset. However, it is still there:

```
# All four factors are still here
levels(data$gender)
```

[1] "female" "male" "non-binary" "unknown"

If we want to remove the unused factor level we have two options: We can either use `droplevels()` or `fct_drop()` from the `forcats` package. A nice little perk of the `droplevels()` function is that it can remove unused factor levels from all factors in the dataset. Here are the different options in action:

```
# Option 1: Using 'droplevels()'
data_3_lvls <-
  data |>
  mutate(gender = droplevels(gender))

levels(data_3_lvls$gender)

[1] "female"      "male"        "non-binary"

# Option 2: Using 'fct_drop()'
data_3_lvls <-
  data |>
  mutate(gender = fct_drop(gender))

levels(data_3_lvls$gender)

[1] "female"      "male"        "non-binary"

# Removing unused factor levels of all factors
data_3_levels <-
  data |> droplevels()

levels(data_3_lvls$gender)

[1] "female"      "male"        "non-binary"
```

In my experience, it is always good to remove unused categories (i.e. levels) because sometimes they can lead to unexpected outcomes. Thus, it is a matter of good practice than an absolute necessity. Besides, it is fairly easy to tidy your factors with the above functions.

7.6 Handling dates, times and durations

There are many reasons why we encounter dates, times and durations in our research. The most apparent reason for handling dates and times is longitudinal studies that focus on change over time. Therefore, at least one of our

variables would have to carry a timestamp. When it comes to data cleaning and wrangling, there are usually two tasks we need to perform:

- Convert data into a suitable format for further analysis, and
- Perform computations with dates and times, e.g. computing the duration between two time points.

The following sections will provide some guidance on solving commonly occurring issues and how to solve them.

7.6.1 Converting dates and times for analysis

No matter which data collection tool you use, you likely encounter differences in how various software report dates and times back to you. Not only is there a great variety in formats in which dates and times are reported, but there are also other issues that one has to take care of, e.g. time zone differences. Here are some examples of how dates could be reported in your data:

```
26 November 2022
26 Nov 2022
November, 26 2022
Nov, 26 2022
26/11/2022
11/26/2022
2022-11-26
20221126
```

It would be nice if there was a standardised method of recording dates and times and, in fact, there is one: ISO 8601⁵. But unfortunately, not every software, let alone scholar, necessarily adheres to this standard. Luckily there is a dedicated *R* package that solely deals with these issues called `lubridate`, which is also part of the `tidyverse`. It helps reformat data into a coherent format and takes care of other essential aspects, such as ‘*time zones, leap days, daylight savings times, and other time related quirks*’ (Grolemund and Wickham 2011). Here, we will primarily focus on the most common issues, but I highly recommend looking at the ‘`lubridate`’ website⁶ to see what else the package has to offer.

Let’s consider the dates (i.e. the data type `date`) shown above and try to convert them all into the same format. To achieve this we can use three different functions, i.e. `dmy()`, `mdy()` and `ymd()`. It is probably very obvious what these letters stand for:

- d for day,

⁵<https://www.iso.org/iso-8601-date-and-time-format.html>

⁶<https://lubridate.tidyverse.org/>

- m for month, and
- y for year.

All we have to do is identify the order in which the dates are presented in your dataset, and the `lubridate` package will do its magic to standardise the format.

```
library(lubridate)

# All of the below will yield the same output
dmy("26 November 2022")
```

```
[1] "2022-11-26"
```

```
dmy("26 Nov 2022")
```

```
[1] "2022-11-26"
```

```
mdy("November, 26 2022")
```

```
[1] "2022-11-26"
```

```
mdy("Nov, 26 2022")
```

```
[1] "2022-11-26"
```

```
dmy("26/11/2022")
```

```
[1] "2022-11-26"
```

```
mdy("11/26/2022")
```

```
[1] "2022-11-26"
```

```
ymd("2022-11-26")
```

```
[1] "2022-11-26"
```

```
ymd("20221126")
```

```
[1] "2022-11-26"
```

Besides dates, we sometimes also have to deal with times. For example, when did someone start a survey and finish it. If you frequently work with software like Qualtrics, SurveyMonkey, etc., you are often provided with such information. From a diagnostic perspective, understanding how long it took someone to complete a questionnaire can provide valuable insights into survey optimisation and design or show whether certain participants properly engaged with your data collection tool. In *R*, we can use either `time` or `dttm` (date-time) as a data type to capture hours, minutes and seconds. The `lubridate` package handles `time` similarly to `date` with easily recognisable abbreviations:

- `h` for hour,
- `m` for minutes, and
- `s` for seconds.

Here is a simple example that demonstrates how conveniently `lubridate` manages to handle even the most outrages ways of specifying times:

```
dinner_time <- tibble(time = c("17:34:21",
                                "17:::34:21",
                                "17 34 21",
                                "17 : 34 : 21"))
                               )

dinner_time |> mutate(time_new = hms(time))

# A tibble: 4 × 2
  time      time_new
  <chr>    <Period>
1 17:34:21 17H 34M 21S
2 17:::34:21 17H 34M 21S
3 17 34 21 17H 34M 21S
4 17 : 34 : 21 17H 34M 21S
```

Lastly, `dttm` data types combine both: `date` and `time`. To illustrate how we can work with this format we can look at some actual data from a questionnaire. For example, the ``quest_time`` dataset has information about questionnaire completion dates and times..

```
quest_time

# A tibble: 84 × 3
  id start          end
  <dbl> <chr>        <chr>
1     1 12/05/2016 20:46 12/05/2016 20:49
2     2 19/05/2016 22:08 19/05/2016 22:25
```

```

3     3 20/05/2016 07:05 20/05/2016 07:14
4     4 13/05/2016 08:59 13/05/2016 09:05
5     5 13/05/2016 11:25 13/05/2016 11:29
6     6 13/05/2016 11:31 13/05/2016 11:37
7     7 20/05/2016 11:29 20/05/2016 11:45
8     8 20/05/2016 14:07 20/05/2016 14:21
9     9 20/05/2016 14:55 20/05/2016 15:01
10    10 20/05/2016 16:12 20/05/2016 16:25
# i 74 more rows

```

While it all looks neat and tidy, keeping dates and times as a `chr` is not ideal, especially if we want to use it to conduct further analysis. Therefore, it would be important to transform these variables into the appropriate `dttm` format. We can use the function `dmy_hm()` similar to previous functions we used.

```

quest_time_clean <-
  quest_time |>
  mutate(start = dmy_hm(start),
         end = dmy_hm(end)
  )

quest_time_clean

# A tibble: 84 x 3
  id start           end
  <dbl> <dttm>        <dttm>
1     1 2016-05-12 20:46:00 2016-05-12 20:49:00
2     2 2016-05-19 22:08:00 2016-05-19 22:25:00
3     3 2016-05-20 07:05:00 2016-05-20 07:14:00
4     4 2016-05-13 08:59:00 2016-05-13 09:05:00
5     5 2016-05-13 11:25:00 2016-05-13 11:29:00
6     6 2016-05-13 11:31:00 2016-05-13 11:37:00
7     7 2016-05-20 11:29:00 2016-05-20 11:45:00
8     8 2016-05-20 14:07:00 2016-05-20 14:21:00
9     9 2016-05-20 14:55:00 2016-05-20 15:01:00
10    10 2016-05-20 16:12:00 2016-05-20 16:25:00
# i 74 more rows

```

While the difference between this format and the previous one is relatively marginal and unimpressive, the main benefit of this conversation becomes apparent when we perform operations on these variables, e.g. computing the time it took participants to complete the questionnaire.

7.6.2 Computing durations with dates, times and date-time variables

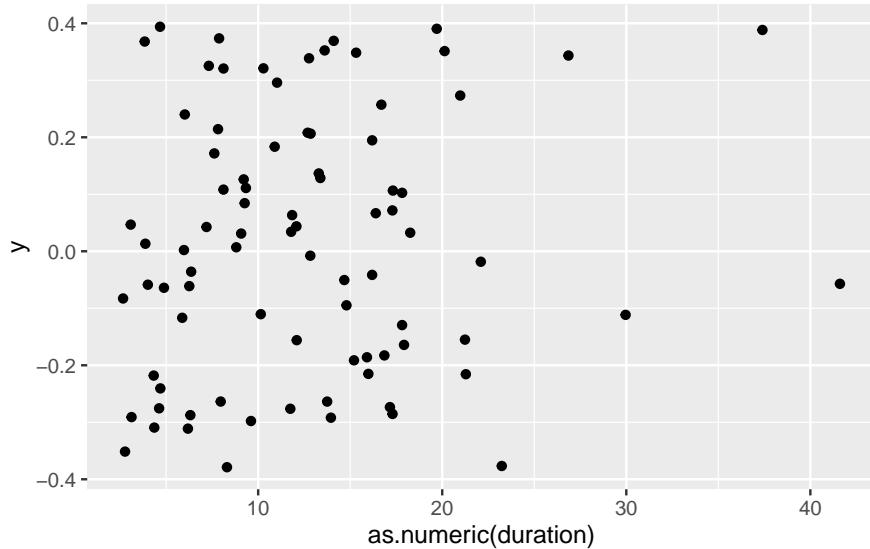
Once we have ‘corrected’ our variables to show as dates, times and date-times, we can perform further analysis and ask important questions about durations, periods and intervals of observations.

Let’s consider the `quest_time` dataset which we just prepared. As mentioned earlier, it is often essential to understand how long it took participants to complete a given questionnaire. Therefore we would want to compute the duration based on the `start` and `end` time.

```
df_duration <-  
  quest_time_clean |>  
  mutate(duration = end - start)  
  
df_duration  
  
# A tibble: 84 x 4  
  id    start          end        duration  
  <dbl> <dttm>       <dttm>     <drtn>  
1 1    2016-05-12 20:46:00 2016-05-12 20:49:00 3 mins  
2 2    2016-05-19 22:08:00 2016-05-19 22:25:00 17 mins  
3 3    2016-05-20 07:05:00 2016-05-20 07:14:00 9 mins  
4 4    2016-05-13 08:59:00 2016-05-13 09:05:00 6 mins  
5 5    2016-05-13 11:25:00 2016-05-13 11:29:00 4 mins  
6 6    2016-05-13 11:31:00 2016-05-13 11:37:00 6 mins  
7 7    2016-05-20 11:29:00 2016-05-20 11:45:00 16 mins  
8 8    2016-05-20 14:07:00 2016-05-20 14:21:00 14 mins  
9 9    2016-05-20 14:55:00 2016-05-20 15:01:00 6 mins  
10 10  2016-05-20 16:12:00 2016-05-20 16:25:00 13 mins  
# i 74 more rows
```

From these results, it seems that some participants have either super-humanly fast reading skills or did not engage with the questionnaire. To get a better overview, we can plot `duration` to see the distribution better.

```
df_duration |>  
  ggplot(aes(x = as.numeric(duration),  
             y = 0)) +  
  geom_jitter()
```



For now, you should not worry about what this code means but rather consider the interpretation of the plot, i.e. that most participants (represented by a black dot) needed less than 20 minutes to complete the questionnaire. Depending on the length of your data collection tool, this could either be good or bad news. Usually, the shorter the questionnaire can be, the higher the response rate, unless you have incentives for participants to complete a longer survey.

In Chapter 8, we will cover data visualisations like this boxplot in greater detail. If you cannot wait any longer and want to create your own visualisations, you are welcome to skip ahead. However, several data visualisations are awaiting you in the next chapter as well.

7.7 Dealing with missing data

There is hardly any Social Sciences project where researchers do not have to deal with missing data. Participants are sometimes unwilling to complete a questionnaire or miss the second round of data collection entirely, e.g. in longitudinal studies. It is not the purpose of this chapter to delve into all aspects of analysing missing data but provide a solid starting point. There are mainly three steps involved in dealing with missing data:

1. Mapping missing data,

2. Identifying patterns of missing data, and
3. Replacing or removing missing data

7.7.1 Mapping missing data

Every study that intends to be rigorous will have to identify how much data is missing because it affects the interpretability of our available data. In *R*, this can be achieved in multiple ways, but using a specialised package like *naniar* does help us to do this very quickly and systematically. First, we have to load the *naniar* package, and then we use the function `vis_miss()` to visualise how much and where exactly data is missing.

```
library(naniar)

vis_miss(wvs_clean)
```

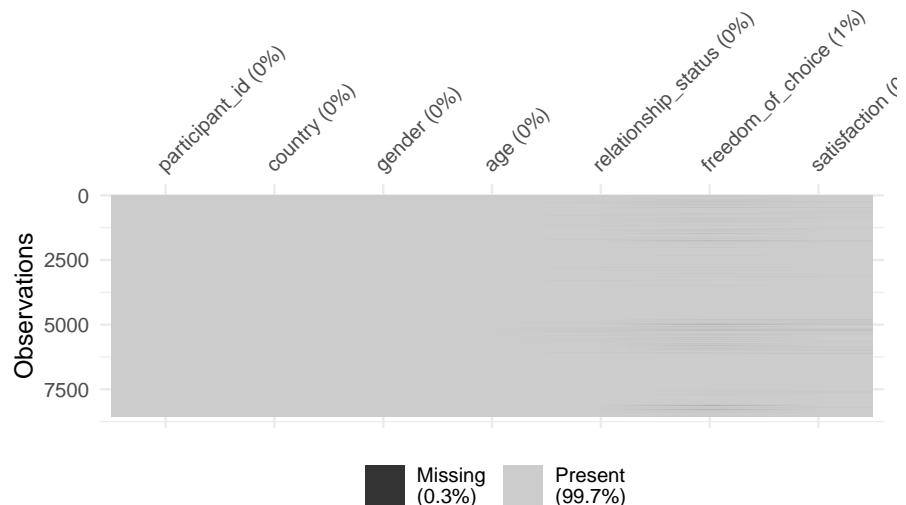


Figure 7.2: Mapping missing data with *naniar*

naniar plays along nicely with the *tidyverse* approach of programming. As such, it would also be possible to write `wvs |> vis_miss()`.

As we can see, 99.7% of our dataset is complete, and we are only missing 0.3%. The dark lines (actually blocks) refer to missing data points. On the x-axis, we can see all our variables, and on the y-axis, we see our observations. This is the same layout as our rectangular dataset: Rows are observations,

and columns are variables. Overall, this dataset appears relatively complete (luckily). In addition, we can see the percentage of missing data per variable. `freedom_of_choice` is the variable with the most missing data, i.e. 0.79%. Still, the amount of missing data is not very large.

When working with larger datasets, it might also be helpful to rank variables by their degree of missing data to see where the most significant problems lie.

```
gg_miss_var(wvs_clean)
```

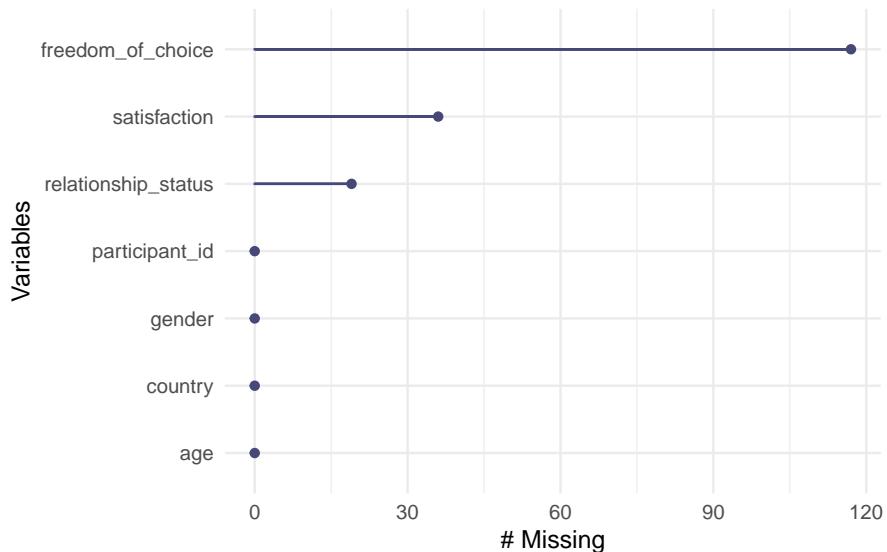


Figure 7.3: Missing data per variable

As we already know from the previous data visualisation, `freedom_of_choice` has the most missing data points, while `participant_id`, and `country_name` have no missing values. If you prefer to see the actual numbers instead, we can use a series of functions that start with `miss_` (for a complete list of all functions, see the reference page of naniar⁷). For example, to retrieve the numeric values which are reflected in the plot above, we can write the following:

```
# Summarise the missingness in each variable
miss_var_summary(wvs_clean)
```

```
# A tibble: 7 x 3
```

⁷<https://naniar.njtierney.com/reference/index.html>

variable	n_miss	pct_miss
<chr>	<int>	<num>
1 freedom_of_choice	117	1.37
2 satisfaction	36	0.420
3 relationship_status	19	0.222
4 participant_id	0	0
5 country	0	0
6 gender	0	0
7 age	0	0

I tend to prefer data visualisations over numerical results for mapping missing data, especially in larger datasets with many variables. This also has the benefit that patterns of missing data can be more easily identified as well.

7.7.2 Identifying patterns of missing data

If you find that your data ‘suffers’ from missing data, it is essential to answer another question: Is data missing systematically? This is quite an important diagnostic step since systematically missing data would imply that if we remove these observations from our dataset, we likely produce wrong results. The following section elaborate on this aspect of missing data. In general, we can distinguish missing data based on how it is missing, i.e.

- missing completely at random (MCAR),
- missing at random (MAR), and
- missing not at random (MNAR). (Rubin 1976)

7.7.2.1 Missing completely at random (MCAR)

Missing completely at random (MCAR) means that neither observed nor missing data can systematically explain why data is missing. It is a pure coincidence how data is missing, and there is no underlying pattern.

The `naniar` package comes with the very popular Little’s MCAR test (Little 1988), which provides insights into whether our data is missing completely at random. Thus, we can call the function `mcar_test()` and inspect the result.

```
wvs_clean |>
  # Remove variables which do not reflect a response
  select(-participant_id) |>
  mcar_test()

# A tibble: 1 x 4
  statistic   df p.value missing.patterns
  <dbl> <dbl>    <dbl>          <int>
1     113.    26 7.66e-13            7
```

When you run such a test, you have to ensure that variables that are not part of the data collection process are removed. In our case, the `participant_id` is generated by the researcher and does not represent an actual response by the participants. As such, we need to remove it using `select()` before we can run the test. A - inverts the meaning of `select()`. While `select(participant_id)` would do what it says, i.e. include it as the only variable in the test, `select(-participant_id)` results in selecting everything but this variable in our test. You will find it is sometimes easier to remove a variable with `select()` rather than listing all the variables you want to keep.

Since the `p.value` of the test is so small that it got rounded down to 0, i.e. $p < 0.0001$, we have to assume that our data is not missing completely at random. If we found that $p > 0.05$, we would have confirmation that data are missing completely at random. We will cover p-values and significance tests in more detail in Section 10.3.

7.7.2.2 Missing at random (MAR)

Missing at random (MAR) refers to a situation where the observed data can explain missing data, but not any unobserved data. Dong and Peng (2013) (p. 2) provide a good example when this is the case:

Let's suppose that students who scored low on the pre-test are more likely to drop out of the course, hence, their scores on the post-test are missing. If we assume that the probability of missing the post-test depends only on scores on the pre-test, then the missing mechanism on the post-test is MAR. In other words, for students who have the same pre-test score, the probability of (them) missing the post-test is random.

Thus, the main difference between MCAR and MAR data lies in the fact that we can observe some patterns of missing data if data is MAR. These patterns are only based on data we have, i.e. observed data. We also assume that no unobserved variables can explain these or other patterns.

Accordingly, we first look into variables with no missing data and see whether they can explain our missing data in other variables. For example, we could investigate whether missing data in `freedom_of_choice` is attributed to specific countries.

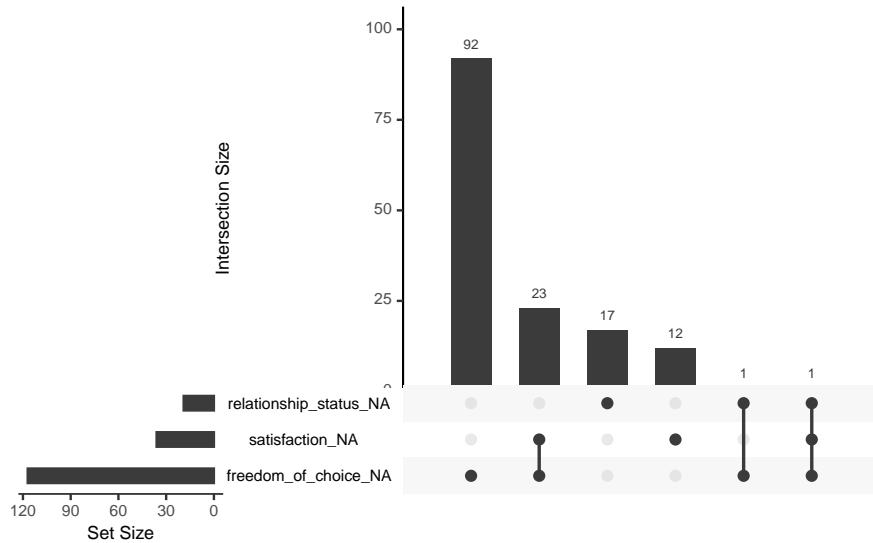
```
wvs_clean |>  
  group_by(country) |>  
  filter(is.na(freedom_of_choice)) |>  
  count(sort = TRUE)
```

```
# A tibble: 4 x 2  
# Groups:   country [4]  
  country     n  
  <fct>    <int>  
1 Japan      47  
2 Bolivia    40  
3 Egypt      23  
4 Iran       7
```

It seems four countries have exceptionally high numbers of missing data for `freedom_of_choice`: Japan, Brazil, New Zealand, Russia and Bolivia. Why this is the case lies beyond this dataset and is something only the researchers themselves could explain. Collecting data in different countries is particularly challenging, and one is quickly faced with different unfavourable conditions. Furthermore, the missing data is not completely random because we have some first evidence that the location of data collection might have affected its completeness.

Another way of understanding patterns of missing data can be achieved by looking at relationships between missing values, for example, the co-occurrence of missing values across different variables. This can be achieved by using *upset plots*. An *upset plot* consists of three parts: Set size, intersection size and a Venn diagram which defines the intersections.

```
gg_miss_upset(wvs_clean)
```



The most frequent combination of missing data in our dataset occurs when only `freedom_of_choice` is missing (the first column), but nothing else. Similar results can be found for `relationship_status` and `age`. The first combination of missing data is defined by two variables: `satisfaction` and `freedom_of_choice`. In total, 107 participants had `satisfaction` and `freedom_of_choice` missing but nothing else.

The ‘set size’ shown in the upset plot refers to the number of missing values for each variable in the diagram. This corresponds to what we have found when looking at Figure 7.3.

Our analysis also suggests that values are not completely randomly missing but that we have data to help explain why they are missing.

7.7.2.3 Missing not at random (MNAR)

Lastly, missing not at random (MNAR) implies that data is missing systematically and that other variables or reasons exist that explain why data is missing. Still, they are not fully known to us. In questionnaire-based research, an easily overlooked reason that can explain missing data is the ‘page-drop-off’ phenomenon. In such cases, participants stop completing a questionnaire once they advance to another page. Figure 7.4 shows this very clearly for a large scale project where an online questionnaire was used. After almost every page break in the questionnaire, some participants decided to discontinue. Finding these types of patterns is difficult when only working with numeric values. Thus, it is always advisable to visualise your data as well. Such missing data is linked to the design of the data collection tool.

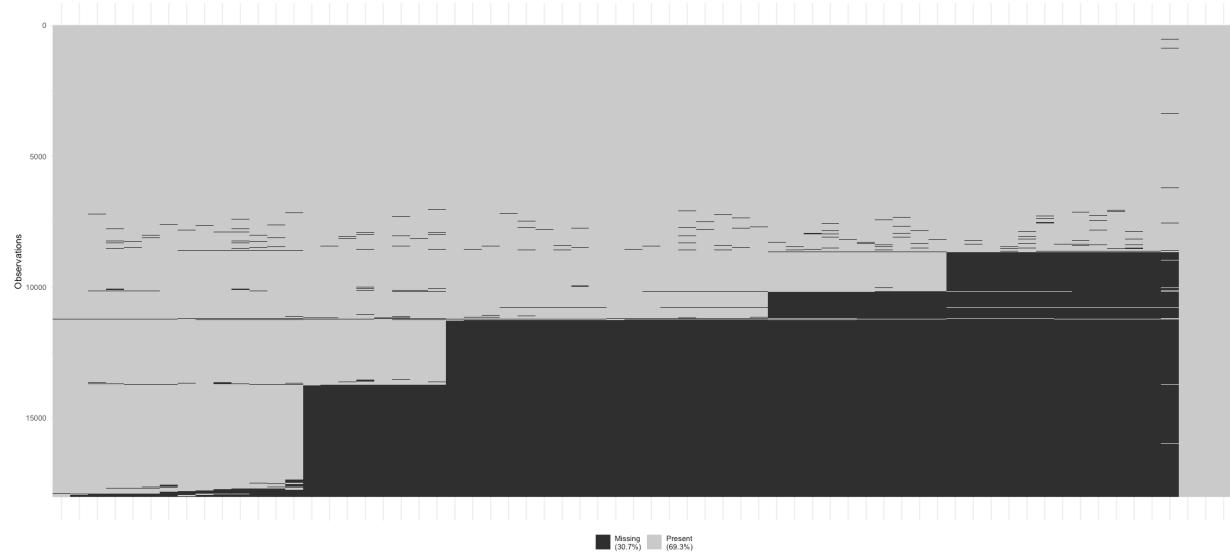


Figure 7.4: MNAR pattern in a dataset due to ‘page-drop-offs’

Defining whether a dataset is MNAR or not is mainly achieved by ruling out MCAR and MAR assumptions. It is not possible to test whether missing data is MNAR, unless we have more information about the underlying population available (Ginkel et al. 2020).

We have sufficient evidence that our data is MAR as was shown above, because we managed to identify some relationships between unobserved and observed data. In practice, it is very rare to find datasets that are truly MCAR (Buuren 2018). Therefore we might consider ‘imputation’ as a possible strategy to solve our missing data problem. More about imputation in the next Chapter.

If you are looking for more inspiration of how you could visualise and identify patterns of missingness in your data, you might find the ‘Gallery⁸’ of the `naniar` website particularly useful.

7.7.3 Replacing or removing missing data

Once you determined which pattern of missing data applies to your dataset, it is time to evaluate how we want to deal with those missing values. Generally, you can either keep the missing values as they are, replace them or remove them entirely.

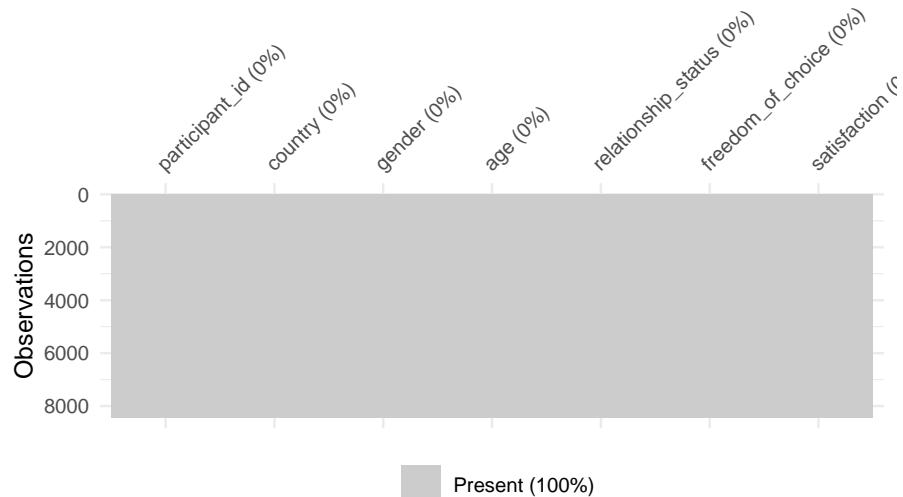
Jakobsen et al. (2017) provide a rule of thumb of 5% where researchers can consider missing data as negligible, i.e. we can ignore the missing values be-

⁸<https://naniar.njtierney.com/articles/naniar-visualisation.html>

cause they won't affect our analysis in a significant way. They also argue that if the proportion of missing data exceeds 40%, we should also only work with our observed data. However, with such a large amount of missing data, it is questionable whether we can rely on our analysis as much as we want to. If the missing data lies somewhere in-between this range, we need to consider the missing data pattern at hand.

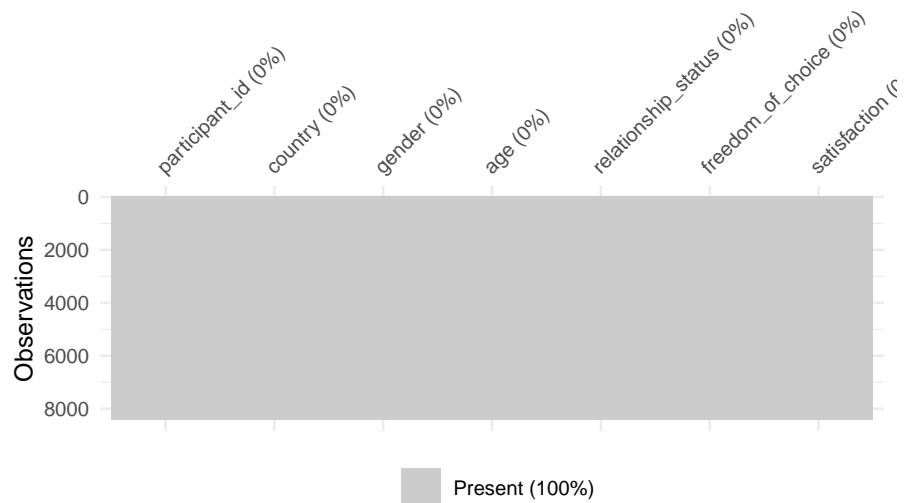
If data is MCAR, we could remove missing data. This process is called '*listwise deletion*', i.e. you remove all data from a participant with missing data. As just mentioned, removing missing values is only suitable if you have relatively few missing values in your dataset (Schafer 1999), as is the case with the `wvs` dataset. There are additional problems with deleting observations listwise, many of which are summarised by Buuren (2018) in his introduction⁹. Usually, we try to avoid removing data as much as possible. If you wanted to perform listwise deletion, it can be done with a single function call: `na.omit()` from the built-in `stats` package. Alternatively, you could also use `drop_na()` from the `tidyverse` package, which is automatically loaded when used `library(tidyverse)`. Here is an example of how we can apply these two functions to achieve the same effect.

```
# No more missing data in this plot
wvs_clean |> na.omit() |> vis_miss()
```



⁹<https://stefvanbuuren.name/fimd/sec-simplesolutions.html>

```
# Here is the alternative method with drop_na()
wvs_clean |> drop_na() |> vis_miss()
```



```
# How many observations are left after we removed all missing data?
wvs_clean |> na.omit() |> count()
```

```
# A tibble: 1 × 1
  n
  <int>
1 8418
```

If you wanted to remove the missing data without the plot, you could use `wvs_no_na <- na.omit(wvs)` or `wvs_no_na <- drop_na()`. However, I always recommend ‘saving’ the result in a new object to ensure we keep the original data available. This can be helpful when trying to compare how this decision affects my analysis, i.e. I can run the analysis with and without missing data removed. For the `wvs` dataset, using `na.omit()` does not seem to be the best option.

Based on our analysis of the `wvs` dataset (by no means complete!), we could assume that data is MAR. In such cases, it is possible to ‘impute’ the missing values, i.e. replacing the missing data with computed scores. This is possible because we can model the missing data based on variables we have. We cannot model missing data based on variables we have not measured in our study for obvious reasons.

You might be thinking: “Why would it make data better if we ‘make up’ numbers? Is this not cheating?” Imputation of missing values is a science in itself. There is plenty to read about the process of handling missing data, which would reach far beyond the scope of this book. However, the seminal work of Buuren (2018), Dong and Peng (2013) and Jakobsen et al. (2017) are excellent starting points. In short: Simply removing missing data can lead to biases in your data, e.g. if we removed missing data in our dataset, we would mainly exclude participants from Japan, Brazil, New Zealand, Russia and Bolivia (as we found out earlier). While imputation is not perfect, it can produce more reliable results than not using imputation at all, assuming our data meets the requirements for such imputation.

Even though our dataset has only a minimal amount of missing data (relative to the entire size), we can still use imputation. There are many different ways to approach this task, one of which is ‘multiple imputation’. As highlighted by Ginkel et al. (2020), multiple imputation has not yet reached the same popularity as listwise deletion, despite its benefits. The main reason for this lies in the complexity of using this technique. There are many more approaches to imputation, and going through them all in detail would be impossible and distract from the book’s main purpose. Still, I would like to share some interesting packages with you that use different imputation methods:

- mice¹⁰ (Uses multivariate imputation via chained equations)
- Amelia¹¹ (Uses a bootstrapped-based algorithm)
- missForest¹² (Uses a random forest algorithm)
- Hmisc¹³ (Uses additive regression, bootstrapping, and predictive mean matching)
- mi¹⁴ (Uses posterior predictive distribution, predictive mean matching, mean-imputation, median-imputation, or conditional mean-imputation)

Besides multiple imputation, there is also the option for single imputation. The `simputation` package offers a great variety of imputation functions, one of which also fits our data quite well `impute_knn()`. This function makes use of a clustering technique called ‘*K-nearest neighbour*’. In this case, the function will look for observations closest to the one with missing data and take the value of that observation. In other words, it looks for participants who answered the questionnaire in a very similar way. The great convenience of this approach is that it can handle all kinds of data types simultaneously, which is not true for all imputation techniques.

¹⁰<https://cran.r-project.org/web/packages/Amelia/index.html>

¹¹<https://cran.r-project.org/web/packages/Amelia/index.html>

¹²<https://cran.r-project.org/web/packages/missForest/index.html>

¹³<https://cran.r-project.org/web/packages/Hmisc/index.html>

¹⁴<https://cran.r-project.org/web/packages/mi/index.html>

If we apply this function to our dataset, we have to write the following:

```
wvs_nona <-
  wvs_clean |>
  select(-participant_id) |>

  # Transform our tibble into a data frame
  as.data.frame() |>
  simputation::impute_knn(. ~ .)

# Be aware that our new dataframe has different datatypes
glimpse(wvs_nona)

Rows: 8,564
Columns: 6
$ country      <fct> Bolivia, Bolivia, Bolivia, Bolivia, Bolivia, Bolivia...
$ gender       <fct> male, male, male, female, female, female, male, fe...
$ age          <dbl> 60, 40, 25, 71, 38, 20, 39, 19, 20, 37, 42, 73, 32...
$ relationship_status <fct> married, married, single, widowed, married, sepa...
$ freedom_of_choice <chr> "7", "8", "10", "5", "5", "5", "10", "6", "9"...
$ satisfaction   <chr> "5", "7", "8", "5", "7", "3", "3", "8", "7", "8", ...

# Let's fix this
wvs_nona <-
  wvs_nona |>
  mutate(age = as.numeric(age),
         freedom_of_choice = as.numeric(freedom_of_choice),
         satisfaction = as.numeric(satisfaction))
```

The function `impute_knn(. ~ .)` might look like a combination of text with an emoji (. ~ .). This imputation function requires us to specify a model to impute the data. Since we want to impute all missing values in the dataset and use all variables available, we put `.` on both sides of the equation, separated by a `~`. The `.` reflects that we do not specify a specific variable but instead tell the function to use all variables that are not mentioned. In our case, we did not mention any variables at all, and therefore it chooses all of them. For example, if we wanted to impute only `freedom_of_choice`, we would have to put `impute_knn(freedom_of_choice ~ .)`. We will elaborate more on how to specify models when we cover regression models (see Chapter 13).

As you will have noticed, we also had to convert our `tibble` into a `data frame` using `as.data.frame()`. As mentioned in Section 5.3, some functions require a specific data type or format. The `simputation` package works with `dplyr`, but it prefers `data frames`. Based on my experiences, the wrong data type or format is

one of the most frequent causes of errors that novice *R* programmers report. So, keep an eye on it and read the documentation carefully. Also, once you inspect the new data frame, you will notice that some of our `double` variables have now turned into `character` values. Therefore, I strongly recommend checking your newly created data frames to avoid any surprises further down the line of your analysis. *R* requires us to be very precise in our work, which I think is rather an advantage than an annoyance¹⁵.

Be aware that imputation of any kind can take a long time. For example, my MacBook Pro took about 4.22 seconds to complete `impute_knn()` with the `wvs` dataset. If we used multiple imputation, this would have taken considerably longer, i.e. several minutes and more.

We now should have a dataset that is free of any missing values. To ensure this is the case we can create the missing data matrix that we made at the beginning of this chapter (see Figure 7.2).

```
vis_miss(wvs_nona)
```

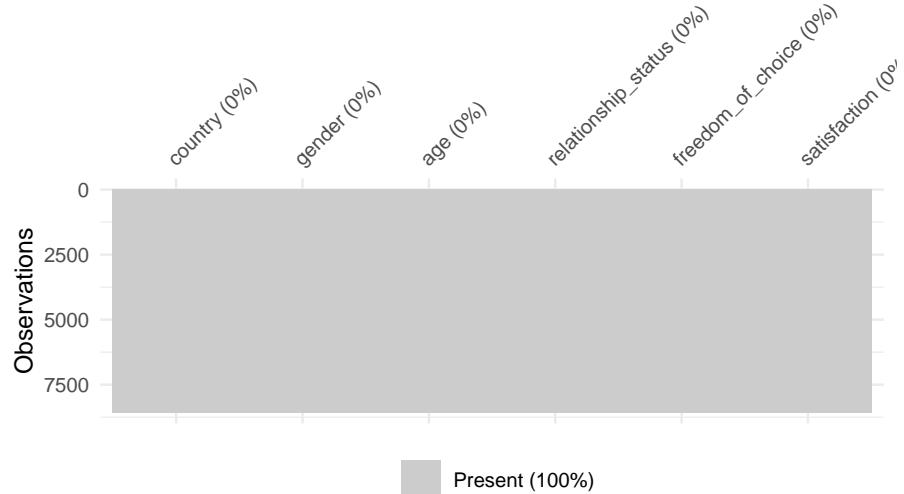


Figure 7.5

As the plot demonstrates, we managed to obtain a dataset which is now free of missing values and we can perform our next steps of data wrangling or analysis.

¹⁵Still, I understand that it often feels annoying.

7.7.4 Two more methods of replacing missing data you hardly ever need

There are two more options for replacing missing data¹⁶, but these I would hardly ever recommend. Still, this section would not be complete without mentioning them, and they have their use-cases.

The first method involves the function `replace_na()`. It allows you to replace any missing data with a value of your choice. For example, let's consider the following dataset which contains feedback from students regarding one of their lectures.

```
# The dataset which contains the feedback
student_feedback <- tibble(name = c("Thomas", NA, "Helen", NA),
                           feedback = as.numeric(c(6, 5, NA, 4)),
                           overall = as_factor(c("excellent", "very good", NA, NA)))

```

`student_feedback`

```
# A tibble: 4 × 3
  name    feedback overall
  <chr>     <dbl> <fct>
1 Thomas      6 excellent
2 <NA>        5 very good
3 Helen       NA <NA>
4 <NA>        4 <NA>
```

We have different types of data to deal with in `student_feedback`, and there are several missing data points, i.e. `NA` in each column. We can use the `replace_na()` function to decide how to replace the missing data. For `feedback`, we might want to insert a number, for example, the average of all available ratings $((6 + 5 + 4) / 3 = 5)$. However, we might wish to add some string values for `name` and `overall`. We have to keep in mind that `replace_na()` requires a `chr` variable as input, which is not the case for `overall`, because it is a `factor`. Consequently, we need to go back to Section 7.4 and change the data type before replacing `NAs`. Here is how we can achieve all this in a few lines of code:

```
student_feedback |>
  mutate(name = replace_na(name, "Unknown"),
         feedback = replace_na(feedback, 5),
         overall = replace_na(as.character(overall), "unrated"),
```

¹⁶Technically, there are many more options, but most of them work similar to the demonstrated techniques and are equally less needed in most scenarios.

```

overall = as_factor(overall)
)

# A tibble: 4 × 3
  name   feedback overall
  <chr>    <dbl> <fct>
1 Thomas      6 excellent
2 Unkown      5 very good
3 Helen       5 unrated
4 Unkown      4 unrated

```

We successfully replaced all the missing values with values of our choice. Only for `overall` we have to perform two steps. If you feel adventurous, you could also warp the `as_factor()` function around the `replace_na()` one. This would result in using only one line of code for this variable.

```

student_feedback |>
  mutate(name = replace_na(name, "Unkown"),
         feedback = replace_na(feedback, 0),
         overall = as_factor(replace_na(as.character(overall), "unrated"))
  )

# A tibble: 4 × 3
  name   feedback overall
  <chr>    <dbl> <fct>
1 Thomas      6 excellent
2 Unkown      5 very good
3 Helen       0 unrated
4 Unkown      4 unrated

```

This approach to handling missing data can be beneficial when addressing these values explicitly, e.g. in a plot. While one could keep the label of `NA`, it sometimes makes sense to use a different label that is more meaningful to your intended audience. Still, I would abstain from replacing `NAs` with an arbitrary value unless there is a good reason for it.

The second method of replacing missing values is less arbitrary and requires the function `fill()`. It replaces missing values with the previous or next non-missing value. The official help page¹⁷ for this function offers some excellent examples of when one would need it, i.e. when specific values are only recorded once for a set of data points. For example, we might want to know how many students are taking certain lectures, but the module ID is only recorded for the first student on each module.

¹⁷<https://tidyverse.org/reference/fill.html>

```
# The dataset which contains student enrollment numbers
student_enrollment <- tibble(module = c("Management", NA, NA,
                                         NA, "Marketing", NA),
                               name = c("Alberto", "Frances", "Martin",
                                       "Alex", "Yuhui", "Himari"))
student_enrollment

# A tibble: 6 x 2
  module     name
  <chr>      <chr>
1 Management Alberto
2 <NA>        Frances
3 <NA>        Martin
4 <NA>        Alex
5 Marketing   Yuhui
6 <NA>        Himari
```

Here it is unnecessary to guess which values we have to insert because we know that `Alberto`, `Frances`, `Martin`, and `Alex` are taking the `Management` lecture. At the same time, `Yuhui` and `Himari` opted for `Marketing`. The `fill()` function can help us complete the missing values accordingly. By default, it completes values top-down, i.e. it takes the last value that was not missing and copies it down until we reach a non-missing value.

```
student_enrollment |> fill(module)
```

```
# A tibble: 6 x 2
  module     name
  <chr>      <chr>
1 Management Alberto
2 Management Frances
3 Management Martin
4 Management Alex
5 Marketing   Yuhui
6 Marketing   Himari
```

If you have to reverse the direction of how *R* should fill in the blank cells in your dataset, you can use `fill(.direction = "up")`. In our example, this would make not much sense, but for the sake of demonstration, here is what would happen:

```
student_enrollment |> fill(module, .direction = "up")
```

```
# A tibble: 6 x 2
  module      name
  <chr>      <chr>
1 Management Alberto
2 Marketing Frances
3 Marketing Martin
4 Marketing Alex
5 Marketing Yuhui
6 <NA>       Himari
```

This method can be helpful if the system you use to collect or produce your data is only recording changes in values, i.e. it does not record all values for all observations in your dataset.

7.7.5 Main takeaways regarding dealing with missing data

Handling missing data is hardly ever a simple process. Do not feel discouraged if you get lost in the myriad of options. While there is some guidance on how and when to use specific strategies to deal with missing values in your dataset, the most crucial point to remember is: Be transparent about what you did. As long as you can explain why you did something and how you did it, everyone can follow your thought process and help improve your analysis. However, ignoring the fact that data is missing and not acknowledging it is more than just unwise.

7.8 Latent constructs and their reliability

Social Scientists commonly face the challenge that we want to measure something that cannot be measured directly. For example, '*happiness*' is a feeling that does not naturally occur as a number we can observe and measure. The opposite is true for '*temperature*' which is naturally measured in numbers. At the same time, '*happiness*' is much more complex of a variable than '*temperature*', because '*happiness*' can unfold in various ways and be caused by different triggers (e.g. a joke, an unexpected present, tasty food, etc.). To account for this, we often work with '*latent variables*'. These are defined as variables that are not directly measured but are inferred from other variables. In practice, we often use multiple questions in a questionnaire to measure one latent variable, usually by computing the mean of those questions.

7.8.1 Computing latent variables

The `gep` dataset from the `r4np` package includes data about students' social integration experience (`si`) and communication skills development (`cs`). Data were obtained using the Global Education Profiler (GEP)¹⁸. I extracted three different questions (also called '*items*') from the questionnaire, which measure `si`, and three items that measure `cs`. This dataset only consists of a randomly chosen set of responses, i.e. 300 out of over 12,000.

```
glimpse(gep)
```

```
Rows: 300
Columns: 12
$ age <dbl> 22, 26, 21, 23, 25, 27, 24, 23, 21, 24, ~
$ gender <chr> "Female", "Female", "Female", "Female", ~
$ level_of_study <chr> "UG", "PGT", "UG", "UG", "PGT", "PGT", "~"
$ si_socialise_with_people_exp <dbl> 6, 3, 4, 3, 4, 2, 5, 1, 6, 6, 3, 3, 4, 5~
$ si_supportive_friends_exp <dbl> 4, 4, 5, 4, 4, 2, 5, 1, 6, 6, 3, 3, 4, 6~
$ si_time_socialising_exp <dbl> 5, 2, 4, 4, 3, 2, 6, 3, 6, 6, 3, 2, 4, 6~
$ cs_explain_ideas_imp <dbl> 6, 5, 5, 5, 5, 5, 4, 5, 6, 6, 5, 5, 4, 6~
$ cs_find_clarification_imp <dbl> 4, 5, 5, 6, 6, 5, 4, 6, 6, 6, 5, 5, 4, 5~
$ cs_learn_different_styles_imp <dbl> 6, 5, 6, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 5~
$ cs_explain_ideas_exp <dbl> 6, 5, 2, 5, 6, 5, 6, 6, 6, 6, 4, 4, 3, 6~
$ cs_find_clarification_exp <dbl> 6, 5, 4, 6, 6, 5, 6, 6, 6, 6, 2, 5, 4, 5~
$ cs_learn_different_styles_exp <dbl> 6, 4, 5, 4, 6, 3, 5, 6, 6, 6, 2, 4, 2, 5~
```

For example, if we wanted to know how each student scored with regards to social integration (`si`), we have to

- compute the mean (`mean()`) of all related items (i.e. all variables starting with `si_`),
- for each row (`rowwise()`) because each row presents one participant.

The same is true for communication skills (`cs_imp` and `cs_exp`). We can compute all three variables in one go. For each variable, we compute `mean()` and use `c()` to list all the variables that we want to include in the mean:

```
# Compute the scores for the latent variable 'si' and 'cs'
gep <-
  gep |>
  rowwise() |>
  mutate(si = mean(c(si_socialise_with_people_exp,
                     si_supportive_friends_exp,
                     si_time_socialising_exp
```

¹⁸<https://warwick.ac.uk/gep>

```

        )
      ),
  cs_imp = mean(c(cs_explain_ideas_imp,
                  cs_find_clarification_imp,
                  cs_learn_different_styles_imp
                  )
                ),
  cs_exp = mean(c(cs_explain_ideas_exp,
                  cs_find_clarification_exp,
                  cs_learn_different_styles_exp
                  )
                )
)
)

glimpse(gep)

```

Rows: 300
 Columns: 15
 Rowwise:
\$ age <dbl> 22, 26, 21, 23, 25, 27, 24, 23, 21, 24, ~
\$ gender <chr> "Female", "Female", "Female", "Female", ~
\$ level_of_study <chr> "UG", "PGT", "UG", "UG", "PGT", "PGT", "~
\$ si_socialise_with_people_exp <dbl> 6, 3, 4, 3, 4, 2, 5, 1, 6, 6, 3, 3, 4, 5~
\$ si_supportive_friends_exp <dbl> 4, 4, 5, 4, 4, 2, 5, 1, 6, 6, 3, 3, 4, 6~
\$ si_time_socialising_exp <dbl> 5, 2, 4, 4, 3, 2, 6, 3, 6, 6, 3, 2, 4, 6~
\$ cs_explain_ideas_imp <dbl> 6, 5, 5, 5, 5, 5, 4, 5, 6, 6, 5, 5, 4, 6~
\$ cs_find_clarification_imp <dbl> 4, 5, 5, 6, 6, 5, 4, 6, 6, 6, 5, 5, 4, 5~
\$ cs_learn_different_styles_imp <dbl> 6, 5, 6, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 5~
\$ cs_explain_ideas_exp <dbl> 6, 5, 2, 5, 6, 5, 6, 6, 6, 6, 4, 4, 3, 6~
\$ cs_find_clarification_exp <dbl> 6, 5, 4, 6, 6, 5, 6, 6, 6, 6, 2, 5, 4, 5~
\$ cs_learn_different_styles_exp <dbl> 6, 4, 5, 4, 6, 3, 5, 6, 6, 6, 2, 4, 2, 5~
\$ si <dbl> 5.000000, 3.000000, 4.333333, 3.666667, ~
\$ cs_imp <dbl> 5.333333, 5.000000, 5.333333, 5.000000, ~
\$ cs_exp <dbl> 6.000000, 4.666667, 3.666667, 5.000000, ~

Compared to dealing with missing data, this is a fairly straightforward task. However, there is a caveat. Before we can compute the mean across all these variables, we need to know and understand whether all these scores reliably contribute to one single latent variable. If not, we would be in trouble and make a significant mistake. This is commonly known as *internal consistency* and is a measure of reliability, i.e. how much we can rely on the fact that multiple questions in a questionnaire measure the same underlying, i.e. latent, construct.

7.8.2 Checking internal consistency of items measuring a latent variable

By far, the most common approach to assessing *internal consistency* of latent variables is Cronbach's α . This indicator looks at how strongly a set of items (i.e. questions in your questionnaire) are related to each other. For example, the stronger the relationship of all items starting with `si_` to each other, the more likely we achieve a higher Cronbach's α . The `psych` package has a suitable function to compute it for us.

Instead of listing all the items by hand, I use the function `starts_with()` to pick only the variables whose names start with `si_`. It certainly pays off to think about your variable names more thoroughly in advance to benefit from such shortcuts (see also Section 7.3).

```
gep |>
  select(starts_with("si_")) |>
  psych::alpha()

Reliability analysis
Call: psych::alpha(x = select(gep, starts_with("si_")))

  raw_alpha std.alpha G6(smc) average_r S/N   ase mean   sd median_r
          0.85      0.85      0.8      0.66 5.8 0.015  3.5 1.3      0.65

  95% confidence boundaries
    lower alpha upper
Feldt     0.82  0.85  0.88
Duhachek 0.82  0.85  0.88

Reliability if an item is dropped:
  raw_alpha std.alpha G6(smc) average_r S/N alpha se
  si_socialise_with_people_exp 0.82    0.82    0.70    0.70 4.6  0.021
  si_supportive_friends_exp    0.77    0.77    0.63    0.63 3.4  0.027
  si_time_socialising_exp     0.79    0.79    0.65    0.65 3.7  0.025
                           var.r med.r
  si_socialise_with_people_exp   NA  0.70
  si_supportive_friends_exp     NA  0.63
  si_time_socialising_exp      NA  0.65

Item statistics
  n raw.r std.r r.cor r.drop mean   sd
  si_socialise_with_people_exp 300  0.86  0.86  0.75  0.69  3.7 1.4
  si_supportive_friends_exp    300  0.90  0.89  0.81  0.75  3.5 1.6
  si_time_socialising_exp     300  0.88  0.88  0.79  0.73  3.2 1.5
```

```
Non missing response frequency for each item
      1   2   3   4   5   6 miss
si_socialise_with_people_exp 0.06 0.17 0.22 0.24 0.19 0.12    0
si_supportive_friends_exp     0.13 0.18 0.20 0.18 0.18 0.13    0
si_time_socialising_exp      0.15 0.21 0.22 0.20 0.12 0.10    0
```

The function `alpha()` returns a lot of information. The most important part, though, is shown at the very beginning:

```
raw_alpha std.alpha G6(smc) average_r S/N ase mean   sd median_r
  0.85       0.85      0.8       0.66 5.76 0.01 3.46 1.33      0.65
```

In most publications, researchers would primarily report the `raw_alpha` value. This is fine, but it is not a bad idea to include at least `std.alpha` and `G6(smc)` as well.

In terms of interpretation, Cronbach's α scores can range from 0 to 1. The closer the score to 1, the higher we would judge its reliability. Nunally (1967) originally provided the following classification for Cronbach's α :

- between 0.6 and 0.5 can be sufficient during the early stages of development,
- 0.8 or higher is sufficient for most basic research,
- 0.9 or higher is suitable for applied research, where the questionnaires are used to make critical decisions, e.g. clinical studies, university admission tests, etc., with a 'desired standard' of 0.95.

However, a few years later, Nunally (1978) revisited his original categorisation and considered 0.7 or higher as a suitable benchmark in more exploratory-type research. This gave grounds for researchers to pick and choose the 'right' threshold for them (Henson 2001). Consequently, depending on your research field, the expected reliability score might lean more towards 0.7 or 0.8. Still, the higher the score, the more reliable you latent variable is.

Our dataset shows that `si` scores a solid $\alpha = 0.85$, which is excellent. We should repeat this step for `cs_imp` and `cs_exp` as well. However, we have to adjust `select()`, because we only want variables included that start with `cs_` and end with the facet of the respective variable, i.e. `_imp` or `_exp`. Otherwise we compute the Cronbach's α for a combined latent construct.

```
# Select variables which start with 'cs_'
gep |>
  select(starts_with("cs_")) |>
  glimpse()
```

```
Rows: 300
Columns: 8
```

```
Rowwise:
$ cs_explain_ideas_imp      <dbl> 6, 5, 5, 5, 5, 5, 4, 5, 6, 6, 5, 5, 4, 6~
$ cs_find_clarification_imp <dbl> 4, 5, 5, 6, 6, 5, 4, 6, 6, 6, 5, 5, 4, 5~
$ cs_learn_different_styles_imp <dbl> 6, 5, 6, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 5~
$ cs_explain_ideas_exp      <dbl> 6, 5, 2, 5, 6, 5, 6, 6, 6, 4, 4, 3, 6~
$ cs_find_clarification_exp <dbl> 6, 5, 4, 6, 6, 5, 6, 6, 6, 2, 5, 4, 5~
$ cs_learn_different_styles_exp <dbl> 6, 4, 5, 4, 6, 3, 5, 6, 6, 6, 2, 4, 2, 5~
$ cs_imp                      <dbl> 5.333333, 5.000000, 5.333333, 5.000000, ~
$ cs_exp                      <dbl> 6.000000, 4.666667, 3.666667, 5.000000, ~
```

Therefore it is necessary to define two conditions to select the correct items in our dataset using `ends_with()` and combine these two with the logical operator `&` (see also Table 5.1).

```
# Do the above but include only variables that also end with '_imp'
gep |>
  select(starts_with("cs_") & ends_with("_imp")) |>
  glimpse()
```

```
Rows: 300
Columns: 4
Rowwise:
$ cs_explain_ideas_imp      <dbl> 6, 5, 5, 5, 5, 5, 4, 5, 6, 6, 5, 5, 4, 6~
$ cs_find_clarification_imp <dbl> 4, 5, 5, 6, 6, 5, 4, 6, 6, 6, 5, 5, 4, 5~
$ cs_learn_different_styles_imp <dbl> 6, 5, 6, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 5~
$ cs_imp                      <dbl> 5.333333, 5.000000, 5.333333, 5.000000, ~
```

We also created a variable that starts with `cs_` and ends with `_imp`, i.e. the latent variable we computed, which also has to be removed.

```
# Remove 'cs_imp', because it is the computed latent variable
gep |>
  select(starts_with("cs_") & ends_with("_imp"), -cs_imp) |>
  glimpse()
```

```
Rows: 300
Columns: 3
Rowwise:
$ cs_explain_ideas_imp      <dbl> 6, 5, 5, 5, 5, 5, 4, 5, 6, 6, 5, 5, 4, 6~
$ cs_find_clarification_imp <dbl> 4, 5, 5, 6, 6, 5, 4, 6, 6, 6, 5, 5, 4, 5~
$ cs_learn_different_styles_imp <dbl> 6, 5, 6, 4, 4, 4, 4, 6, 6, 6, 4, 5, 5, 5~
```

Thus, we end up computing the Cronbach's α for both variables in the following way:

```
gep |>
  select(starts_with("cs_") & ends_with("_imp"), -cs_imp) |>
  psych::alpha()

Reliability analysis
Call: psych::alpha(x = select(gep, starts_with("cs_") & ends_with("_imp"),
  -cs_imp))

  raw_alpha std.alpha G6(smc) average_r S/N   ase mean   sd median_r
  0.86      0.87     0.82      0.68 6.5 0.014     5 0.98     0.65

  95% confidence boundaries
    lower alpha upper
Feldt     0.83  0.86  0.89
Duhachek 0.84  0.86  0.89

  Reliability if an item is dropped:
    raw_alpha std.alpha G6(smc) average_r S/N
  cs_explain_ideas_imp       0.78      0.78      0.65      0.65 3.6
  cs_find_clarification_imp 0.76      0.76      0.62      0.62 3.2
  cs_learn_different_styles_imp 0.88      0.88      0.79      0.79 7.4
    alpha se var.r med.r
  cs_explain_ideas_imp      0.025     NA  0.65
  cs_find_clarification_imp 0.028     NA  0.62
  cs_learn_different_styles_imp 0.014     NA  0.79

  Item statistics
    n raw.r std.r r.cor r.drop mean   sd
  cs_explain_ideas_imp      300  0.90  0.90  0.85  0.77  5.2 1.0
  cs_find_clarification_imp 300  0.91  0.91  0.87  0.79  5.1 1.1
  cs_learn_different_styles_imp 300  0.86  0.85  0.71  0.67  4.8 1.2

  Non missing response frequency for each item
    1   2   3   4   5   6 miss
  cs_explain_ideas_imp      0.01 0.01 0.06 0.14 0.27 0.51     0
  cs_find_clarification_imp 0.01 0.02 0.05 0.14 0.30 0.47     0
  cs_learn_different_styles_imp 0.01 0.03 0.09 0.19 0.31 0.36     0

  gep |>
  select(starts_with("cs_") & ends_with("_exp"), -cs_exp) |>
  psych::alpha()
```

```

Reliability analysis
Call: psych::alpha(x = select(gep, starts_with("cs_") & ends_with("_exp"),
                               -cs_exp))

  raw_alpha std.alpha G6(smc) average_r S/N    ase mean   sd median_r
  0.81        0.82     0.76      0.6 4.4 0.019  4.2 1.1      0.58

  95% confidence boundaries
  lower alpha upper
Feldt    0.77  0.81  0.85
Duhachek 0.78  0.81  0.85

Reliability if an item is dropped:
  raw_alpha std.alpha G6(smc) average_r S/N
  cs_explain_ideas_exp          0.69      0.69      0.53      0.53 2.3
  cs_find_clarification_exp     0.73      0.74      0.58      0.58 2.8
  cs_learn_different_styles_exp 0.81      0.81      0.68      0.68 4.2
  alpha se var.r med.r
  cs_explain_ideas_exp         0.035     NA  0.53
  cs_find_clarification_exp    0.031     NA  0.58
  cs_learn_different_styles_exp 0.022     NA  0.68

Item statistics
  n raw.r std.r r.cor r.drop mean   sd
  cs_explain_ideas_exp       300  0.88  0.88  0.80   0.71  4.2 1.3
  cs_find_clarification_exp 300  0.85  0.86  0.76   0.68  4.5 1.2
  cs_learn_different_styles_exp 300  0.84  0.82  0.67   0.61  4.0 1.4

Non missing response frequency for each item
  1   2   3   4   5   6 miss
  cs_explain_ideas_exp      0.04 0.09 0.11 0.33 0.26 0.17  0
  cs_find_clarification_exp 0.02 0.05 0.13 0.27 0.32 0.21  0
  cs_learn_different_styles_exp 0.06 0.09 0.21 0.24 0.22 0.17  0

```

Similarly, to `si`, `cs_imp` and `cs_exp` show very good internal consistency scores: $\alpha_{cs_imp} = 0.86$ and $\alpha_{cs_exp} = 0.81$. Based on these results we could be confident to use our latent variables for further analysis.

7.8.3 Confirmatory factor analysis

However, while Cronbach's α is very popular due to its simplicity, there is plenty of criticism. Therefore, it is often not enough to report the Cronbach's α , but undertake additional steps. Depending on the stage of development of your measurement instrument (e.g. your questionnaire), you likely have to perform one of the following before computing the α scores:

- *Exploratory factor analysis (EFA)*: Generally used to identify latent variables in a set of questionnaire items. Often these latent variables are not yet known. This process is often referred to as *dimension reduction*.
- *Confirmatory factor analysis (CFA)*: This approach is used to confirm whether a set of items truly reflect a latent variable which we defined ex-ante.

During the development of a new questionnaire, researchers often perform EFA on a pilot dataset to see which factors would emerge without prior assumptions. Once this process has been completed, another round of data would be collected to perform a CFA. This CFA ideally confirms the results from the pilot study. It is also not unusual to split a larger dataset into two parts and then perform EFA and CFA on separate sub-samples. However, this latter approach has the disadvantage that researchers would not have the opportunity to improve the questionnaire between EFA and CFA.

Since the gep data is based on an established measurement tool, we perform a CFA. To perform a CFA, we use the popular `lavaan` (*Latent Variable Analysis*) package. The steps of running a CFA in *R* include:

1. Define which variables are supposed to measure a specific latent variable (i.e. creating a model).
2. Run the CFA to see whether our model fits the data we collected.
3. Interpret the results based on various indicators.

The code chunk below breaks down these steps in the context of the `lavaan` package.

```
library(lavaan)

#1: Define the model which explains how items relate to latent variables
model <- '
social_integration =~
si_socialise_with_people_exp +
si_supportive_friends_exp +
si_time_socialising_exp

comm_skills_imp =~
cs_explain_ideas_imp +
cs_find_clarification_imp +
cs_learn_different_styles_imp

comm_skills_exp =~
cs_explain_ideas_exp +
cs_find_clarification_exp +
```

```
cs_learn_different_styles_exp
'

#2: Run the CFA to see how well this model fits our data
fit <- cfa(model, data = gep)

#3a: Extract the performance indicators
fit_indices <- fitmeasures(fit)

#3b: We tidy the results with enframe() and
#     pick only those indices we are most interested in
fit_indices |>
  enframe() |>
  filter(name == "cfi" |
         name == "srmr" |
         name == "rmsea") |>
  mutate(value = round(value, 3))    # Round to 3 decimal places

# A tibble: 3 × 2
  name   value
  <chr> <lvn.vctr>
1 cfi    0.967
2 rmsea  0.081
3 srmr   0.037
```

The `enframe()` function is useful to turn a named vector (i.e. a named sequence of values) into a tibble, which we can further manipulate using the functions we already know. However, it is not required to take this step to inspect the results. This can be done out of pure convenience. If we want to know where `name` and `value` came from we have to inspect the output from `enframe(fit_indices)`.

```
enframe(fit_indices)

# A tibble: 46 × 2
  name      value
  <chr>    <lvn.vctr>
  1 npar    2.100000e+01
  2 fmin    1.177732e-01
  3 chisq   7.066390e+01
  4 df      2.400000e+01
  5 pvalue  1.733810e-06
  6 baseline.chisq 1.429729e+03
  7 baseline.df    3.600000e+01
```

```

8 baseline.pvalue 0.000000e+00
9 cfi           9.665187e-01
10 tli          9.497780e-01
# i 36 more rows

```

These column names were generated when we called the function `enframe()`. I often find myself working through chains of analytical steps iteratively to see what the intermediary steps produce. This also makes it easier to spot any mistakes early on. Therefore, I recommend slowly building up your `dplyr` chains of function calls, especially when you just started learning *R* and the `tidyverse` approach of data analysis.

The results of our CFA appear fairly promising:

- The `cfi` (*Comparative Fit Index*) lies above 0.95 ,
- The `rmsea` (*Root Mean Square Error of Approximation*) appears slightly higher than desirable, which usually is lower than 0.6 , and
- The `srmr` (*Standardised Root Mean Square Residual*) lies well below 0.08 (West et al. 2012; Hu and Bentler 1999).

Overall, the model seems to suggest a good fit with our data. Combined with the computed Cronbach's α , we can be reasonably confident in our latent variables and perform further analytical steps.

7.8.4 Reversing items with opposite meaning

Questionnaires like the `gep` consist of items that are all phrased in the 'same direction'. What I mean by this is that all questions are asked so that a high score would carry the same meaning. However, this is not always true. Consider the following three questions as an example:

- Q1: '*I enjoy talking to people I don't know.*'
- Q2: '*I prefer to stay at home.*'
- Q3: '*Making new friends is exciting.*'

Q1 and Q3 both refer to aspects of socialising with or getting to know others. Q2, however, seems to ask for the opposite, i.e. not engaging with others and staying at home. Intuitively, we would assume that a participant who scores high on Q1 and Q2 would score low on Q3. Q3 is a so-called *reversed item*, i.e. the meaning of the question is reversed compared to other questions in this set. This strategy is commonly used to test for respondents' engagement with a questionnaire and break a possible response pattern. For example, if a participant scores 5 out of 6 on Q1, Q2 and Q3, we have to assume that this person might not have paid much attention to the statements because someone who scores high on Q1 and Q3 would likely not score high on Q2. However, their usefulness is debated across disciplines. When designing your

data collection tool, items must be easy to understand, short, not double-barreled and reflect a facet of a latent variable appropriately. These aspects are far more important than reversing the meaning of items.

Suppose we work with a dataset containing reversed items. In that case, we have to recode the answers for each participant to reflect the same meaning as the other questions before computing the Cronbach's α or performing an EFA or CFA.

Let's assume we ask some of our friends to rate Q1, Q2 and Q3 on a scale from 1-6. We might obtain the following results:

```
df_social <- tibble(name = c("Agatha Harkness", "Daren Cross", "Hela",
                            "Aldrich Killian", "Malekith", "Ayesha"),
                      q1 = c(6, 2, 4, 5, 5, 2),
                      q2 = c(2, 5, 1, 2, 3, 6),
                      q3 = c(5, 1, 5, 4, 5, 1))
```

`df_social`

	name	q1	q2	q3
	<chr>	<dbl>	<dbl>	<dbl>
1	Agatha Harkness	6	2	5
2	Daren Cross	2	5	1
3	Hela	4	1	5
4	Aldrich Killian	5	2	4
5	Malekith	5	3	5
6	Ayesha	2	6	1

While, for example, Agatha Harkness and Aldrich Killian are both very social, Ayesha seems to be rather the opposite. For all of our respondents, we can notice a pattern that if the value of `q1` is high, usually `q3` is high too, but `q2` is low. We would be very disappointed and probably puzzled if we now computed the Cronbach's α of our latent variable `social`.

```
# Cronbach's alpha without reverse coding q2
df_social |>
  select(-name) |>
  psych::alpha()
```

```
Warning      in      psych::alpha(select(df_social,
                                         - name)): Some items were negatively correlated with the first principal component and probably should be reversed.
To do this, run the function again with the 'check.keys=TRUE' option
```

Some items (q2) were negatively correlated with the first principal component and probably should be reversed.

To do this, run the function again with the 'check.keys=TRUE' option

```
Reliability analysis
Call: psych::alpha(x = select(df_social, -name))

raw_alpha std.alpha G6(smc) average_r   S/N ase mean   sd median_r
-2.2      -1.7      0.7     -0.27 -0.64 1.7  3.6 0.69      -0.8

95% confidence boundaries
      lower alpha upper
Feldt    -12.47 -2.18  0.52
Duhachek -5.49 -2.18  1.13

Reliability if an item is dropped:
      raw_alpha std.alpha G6(smc) average_r   S/N alpha se var.r med.r
q1     -21.00    -21.04   -0.91     -0.91 -0.95  17.947   NA -0.91
q2      0.94     0.95    0.91      0.91 19.70   0.042   NA  0.91
q3     -7.61    -8.03   -0.80     -0.80 -0.89  6.823   NA -0.80

Item statistics
      n raw.r std.r r.cor r.drop mean   sd
q1 6  0.93  0.94  0.95   0.29  4.0 1.7
q2 6 -0.58 -0.61 -0.89  -0.88  3.2 1.9
q3 6  0.83  0.84  0.93  -0.22  3.5 2.0

Non missing response frequency for each item
      1   2   3   4   5   6 miss
q1 0.00 0.33 0.00 0.17 0.33 0.17   0
q2 0.17 0.33 0.17 0.00 0.17 0.17   0
q3 0.33 0.00 0.00 0.17 0.50 0.00   0
```

The function `alpha()` is clever enough to make us aware that something went wrong. It even tells us that Some items (q2) were negatively correlated with the total scale and probably should be reversed. Also, the Cronbach's α is now negative, which is another indicator that reversed items might be present in the dataset. To remedy this problem, we have to recode q2 to reflect the opposite meaning. Inverting the coding usually implies we have to mirror the scores, i.e. the largest value become the smallest one and vice versa. Since we use a 6-point Likert scale, we have to recode values in the following way:

- 6 -> 1
- 5 -> 2
- 4 -> 3

- 3 -> 4
- 2 -> 5
- 1 -> 6

As is so often the case in *R*, there is more than just one way of achieving this:

- Compute a new variable with `mutate()` only,
- Apply `recode()` to the variable in question, or
- Use the `psych` package.

I will introduce you to all three methods, because each comes with their own benefits and drawbacks. Thus, for me, it is very context-specific.

7.8.4.1 Reversing items with `mutate()`

Let us begin with the method that is my favourite because it is very quick to execute and only requires one function. With `mutate()`, we can calculate the reversed score for an item by applying the following formula:

$$score_{reversed} = score_{max} + score_{min} - score_{obs}$$

In other words, the reversed score $score_{reversed}$ of an item equals the theoretical maximum score on a given scale $score_{max}$ plus the theoretical minimum score on this scale $score_{min}$, and then we subtract the observed score $score_{obs}$, i.e. the actual response by our participant. For example, if someone in our study scored a 5 on q2, we could compute the reversed score as follows: $6 + 1 - 5 = 2$ because our Likert scale ranges from 1 to 6. In *R*, we can apply this idea to achieve this for multiple observations at once.

```
df_social_r <-  
  df_social |>  
  mutate(q2_r = 7 - q2)  
  
df_social_r |>  
  select(q2, q2_r)
```

```
# A tibble: 6 x 2  
  q2    q2_r  
  <dbl> <dbl>  
1     2     5  
2     5     2  
3     1     6  
4     2     5  
5     3     4  
6     6     1
```

This technique is simple, flexible and easy to read and understand. We can

change this formula to fit any scale length and make adjustments accordingly. I strongly recommend starting with this approach before opting for one of the other two.

7.8.4.2 Reversing items with `recode()`

Another option to reverse items is `recode()`. Conceptually, `recode()` functions similarly to `fct_recode()`, only that it can handle numbers and factors. Therefore, we can apply a similar strategy as we did before with factors (see Section 7.5.1).

```
df_social_r <-
  df_social |>
  mutate(q2_r = recode(q2,
    "6" = "1",
    "5" = "2",
    "4" = "3",
    "3" = "4",
    "2" = "5",
    "1" = "6")
  )

df_social_r
```

	name	q1	q2	q3	q2_r
	<chr>	<dbl>	<dbl>	<dbl>	<chr>
1	Agatha Harkness	6	2	5	5
2	Daren Cross	2	5	1	2
3	Hela	4	1	5	6
4	Aldrich Killian	5	2	4	5
5	Malekith	5	3	5	4
6	Ayesha	2	6	1	1

While this might seem convenient, it requires more coding and is, therefore, more prone to errors. If we had to do this for multiple items, this technique is not ideal. However, it is a great option whenever you need to change values to something entirely customised, e.g. every 6 needs to be come a 10 and every 3 needs to become a 6. However, such cases are rather rare in my experience.

Be aware, that after using `recode()`, your variable becomes a `chr`, and you likely have to convert it back to a numeric data type. We can adjust the above code to achieve recoding and converting in one step.

```
df_social_r <-
  df_social |>
  mutate(q2_r = as.numeric(recode(q2,
    "6" = "1",
    "5" = "2",
    "4" = "3",
    "3" = "4",
    "2" = "5",
    "1" = "6")))
)
```

df_social_r

```
# A tibble: 6 x 5
  name          q1     q2     q3   q2_r
  <chr>        <dbl>  <dbl>  <dbl> <dbl>
1 Agatha Harkness     6      2      5      5
2 Daren Cross         2      5      1      2
3 Hela                 4      1      5      6
4 Aldrich Killian     5      2      4      5
5 Malekith             5      3      5      4
6 Ayesha              2      6      1      1
```

7.8.4.3 Reversing items with the `psych` package

The last option I would like to share with you is with the `psych` package. This option can be helpful when reversing multiple items at once without having to create additional variables. The function `reverse.code()` takes a dataset with items and requires the specification of a coding key, i.e. `keys`. This coding key indicates whether an item in this dataset should be reversed or not by using `1` for items that should not be reversed and `-1` for items you wish to reverse.

```
items <- df_social |> select(-name)

df_social_r <- psych::reverse.code(keys = c(1, -1, 1),
                                     items = items,
                                     mini = 1,
                                     maxi = 6)

df_social_r
```

```
q1 q2- q3
[1,] 6   5   5
[2,] 2   2   1
```

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

7.8.4.4 A final remark about reversing item scores

No matter which method we used, we successfully converted the scores from q2 and stored them in a new variable q2_r or, in the case of the psych package, as q2-. We can now perform the internal consistency analysis again.

```
# Cronbach's alpha after reverse coding q2
df_social_r |>
  select(-name, -q2) |>
  psych::alpha()

Reliability analysis
Call: psych::alpha(x = select(df_social_r, -name, -q2))

  raw_alpha std.alpha G6(smc) average_r S/N   ase mean   sd median_r
    0.95      0.95      0.95      0.87   21 0.033  3.8 1.8      0.91

  95% confidence boundaries
    lower alpha upper
  Feldt     0.80  0.95  0.99
  Duhachek  0.89  0.95  1.02

  Reliability if an item is dropped:
    raw_alpha std.alpha G6(smc) average_r S/N alpha se var.r med.r
  q1        0.95      0.95      0.91      0.91   21    0.037   NA  0.91
  q3        0.88      0.89      0.80      0.80    8    0.092   NA  0.80
  q2_r      0.94      0.95      0.91      0.91   20    0.042   NA  0.91

  Item statistics
    n raw.r std.r r.cor r.drop mean   sd
  q1  6  0.94  0.94  0.91   0.87  4.0 1.7
  q3  6  0.98  0.98  0.98   0.96  3.5 2.0
  q2_r 6  0.95  0.95  0.91   0.88  3.8 1.9

  Non missing response frequency for each item
    1   2   4   5   6 miss
  q1  0.00 0.33 0.17 0.33 0.17    0
  q3  0.33 0.00 0.17 0.50 0.00    0
  q2_r 0.17 0.17 0.17 0.33 0.17    0
```

Now everything seems to make sense, and the internal consistency is excellent. We can assume that all three questions reflect one latent construct. From here, we can return to computing our latent variables (see Section 7.8.1) and continue with our analysis.

When working with datasets, especially those you have not generated yourself, it is always important to check whether any items are meant to be reverse coded. Missing this aspect will likely throw many surprising results and can completely ruin your analysis. I vividly remember my PhD supervisor's story about a viva where a student forgot about reversing their scores. The panellists noticed that the correlations presented were counter-intuitive, and only then it was revealed that the entire analysis had to be redone. Tiny mistakes can often have an enormous ripple effect. Make sure you have 'reversed items' on your checklist when performing data cleaning and wrangling.

7.9 Once you finished with data wrangling

Once you finished your data cleaning, I recommend writing (i.e. exporting) your cleaned data out of *R* into your '*01_tidy_data*' folder. You can then use this dataset to continue with your analysis. This way, you do not have to run all your data wrangling and cleaning code every time you open your *R* project. To write your tidy dataset onto your hard drive of choice, we can use `readr` in the same way as we did at the beginning to import data. However, instead of `read_csv()` we have to use another function that writes the file into a folder. The function `write_csv()` first takes the name of the object we want to save and then the folder and file name we want to save it to. We only made changes to the *wvs* dataset and after cleaning and dealing with missing data, we saved it in the object *wvs_nona*. So we should save it to our hard drive and name it, for example, *wvs_nona.csv*, because it is not only clean, but also has 'no NA', i.e. no missing data.

```
write_csv(wvs_clean, "01_tidy_data/wvs_nona.csv")
```

This chapter has been a reasonably long one. Nonetheless, it only covered the basics of what can and should be done when preparing data for analysis. These steps should not be rushed or skipped. It is essential to have the data cleaned appropriately. This process helps you familiarise yourself with the dataset in great depth, and it makes you aware of limitations or even problems of your data. In the spirit of Open Science¹⁹, using *R* also helps to document the steps you undertook to get from a raw dataset to a tidy one. This is also beneficial if

¹⁹<https://osf.io>

you intend to publish your work later. Reproducibility has become an essential aspect of transparent and impactful research and should be a guiding principle of any empirical research.

Now that we have learned the basics of data wrangling, we can finally start our data analysis.

7.10 Exercises

If you would like to test your knowledge or simply practice what we covered in this chapter you can copy and paste the following code if you have the `r4np` installed already.

```
# Option 1:  
learnr::run_tutorial("ex_data_wrangling", package = "r4np")
```

If you have not yet installed the `r4np` package, you will have to do this first using by using this code chunk:

```
# Option 2:  
devtools::install_github("ddrauber/r4np")  
learnr::run_tutorial("ex_data_wrangling", package = "r4np")
```



8

Descriptive Statistics

The best way to understand how participants in your study have responded to various questions or experimental treatments is to use *descriptive statistics*. As the name indicates, their main purpose is to ‘describe’. Most of the time, we want to describe the composition of our sample and how the majority (or minority) of participants performed.

In contrast, we use *inferential statistics* to make predictions. In Social Sciences, we are often interested in predicting how people will behave in certain situations and scenarios. We aim to develop models that help us navigate the complexity of social interactions that we all engage in but might not fully understand. We cover *inferential statistics* in later chapters of this book.

In short, descriptive statistics are an essential component to understand your data. To some extent, one could argue that we were already describing our data when we performed various data wrangling tasks (see Chapter 7). The following chapters focus on essential descriptive statistics, i.e. those you likely want to investigate in 99.9 out of 100 research projects.

This book takes a ‘visualised’ approach to data analysis. Therefore, each section will entail data visualisations and statistical computing. A key learning outcome of this chapter is to plot your data using the package `ggplot2` and present your data’s characteristics in different ways. Each chapter will ask questions about our dataset that we aim to answer visually and computationally. However, first, we need to understand how to create plots in *R*.

8.1 Plotting in *R* with `ggplot2`

Plotting can appear intimidating at first but is very easy and quick once you understand the basics. The `ggplot2` package is a very popular package to generate plots in *R*, and many other packages are built upon it. This makes it a very flexible tool to create almost any data visualisation you could imagine. If you want to see what is possible with `ggplot2`, you might want to consider

looking at `#tidytuesday`¹ on Twitter, where novices and veterans share their data visualisations every week.

To generate any plot, we need to define three components at least:

- a dataset,
- variables we want to plot, and
- a function to indicate how we want to plot them, e.g. as lines, bars, points, etc.

Admittedly, this is a harsh oversimplification, but it will serve as a helpful guide to get us started. The function `ggplot()` is the one responsible for creating any type of data visualisation. The generic structure of a `ggplot()` looks like this:

```
ggplot(data, aes(x = variable_01, y = variable_02))
```

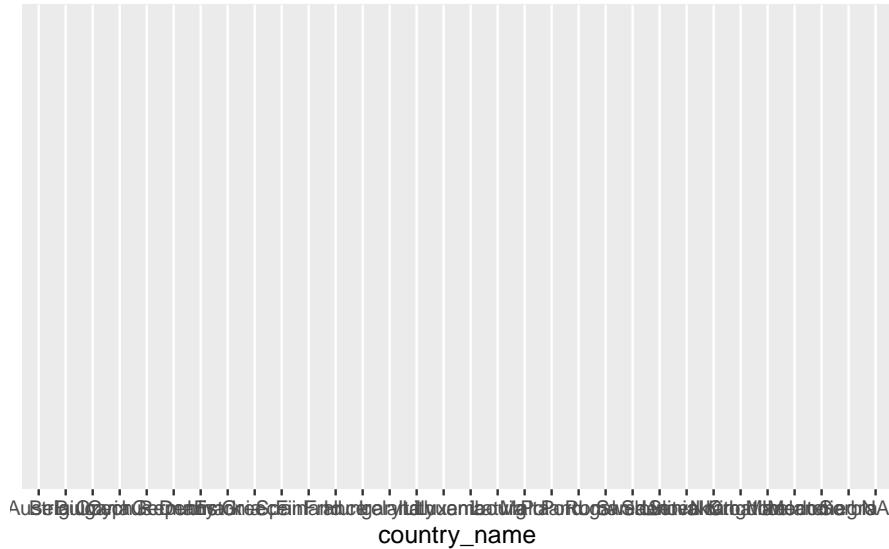
In other words, we first need to provide the dataset `data`, and then the aesthetics (`aes()`). Think of `aes()` as the place where we define our variables (i.e. `x` and `y`). Let's have a look at an example of how to create a plot and how they matter when conducting descriptive statistics.

As mentioned earlier, when we collect data, it is essential to analyse descriptive statistics to better understand the characteristics of our study's participants. For instance, if we aim to assess whether people express trust in others, this might depend on factors such as the country of origin. In such cases, it is crucial to first determine how many individuals in our dataset participated from each country, as this understanding could influence our analysis and the conclusions we draw. The dataset `eqls_2011` from the `r4np` package holds information about how much people think one can trust others and includes data from various European countries. To understand which countries are included and how many people completed the questionnaire we can create a barplot. For our `ggplot()` function we first define the components of the plot as follows:

- our data is `eqls_2011`, and
- our variable of interest is `country_name`.

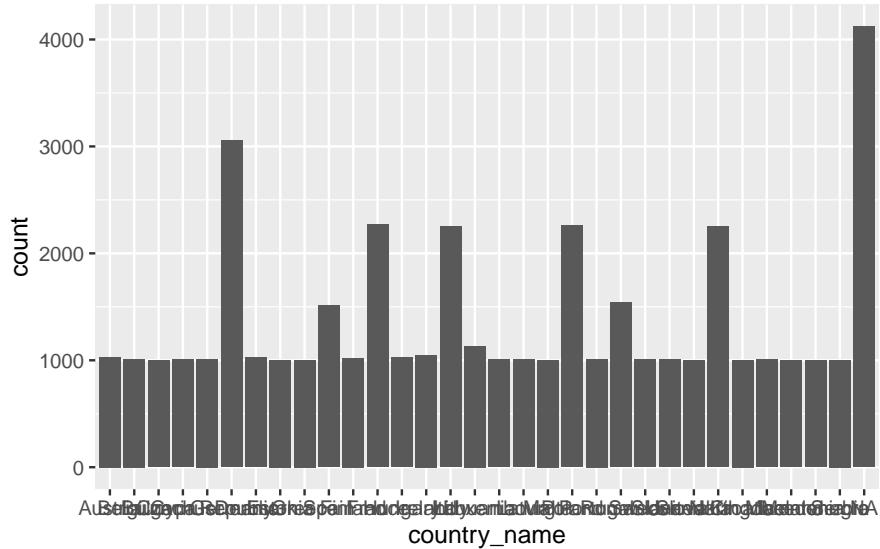
```
ggplot(eqls_2011, aes(x = country_name))
```

¹<https://twitter.com/search?q=%23tidytuesday>



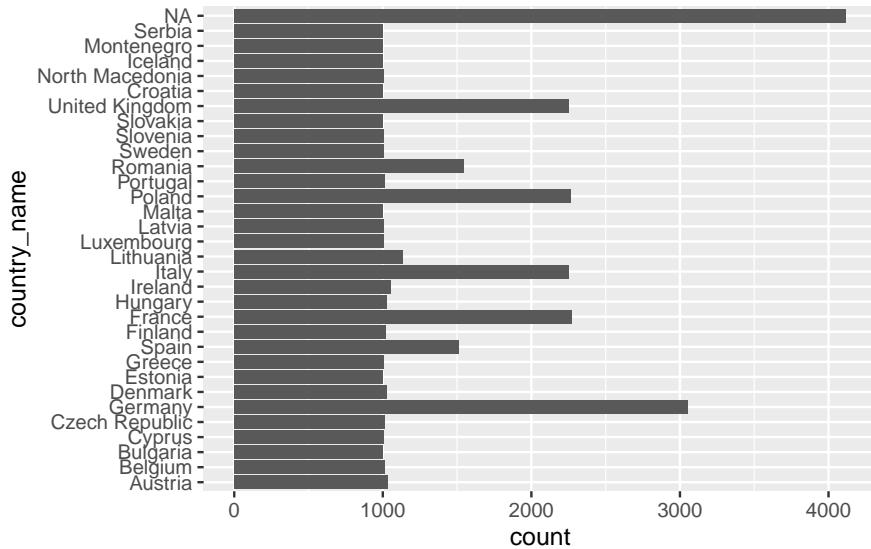
Running this line of code will produce an empty plot. We only get labels for our x-axis since we defined it already. However, we have yet to tell `ggplot()` how we want to represent the data on this canvas. Your choice for how you want to plot your data is usually informed by the type of data you use and the statistics you want to represent. For example, plotting the mean of a factor is not meaningful, e.g. computing the mean of `country_name`. On the other hand, we can count how often specific genres appear in our dataset. One way of representing a factor's count (or frequency) is to use a bar plot. To add an element to `ggplot()`, i.e. bars, we use `+` and append the function `geom_bar()`, which draws bars. The `+` operator works similar to `|>` and allows to chain multiple functions one after the other as part of a `ggplot()`.

```
ggplot(eqls_2011, aes(x = country_name)) +  
  geom_bar()
```



With only two lines of coding, we created a great looking plot. Well, not quite. When working with factors, the category names can be rather long. In this plot, we have lots of categories, and all labels are bit too close to each other to be legible. This might be an excellent opportunity to use `coord_flip()`, which rotates the entire plot by 90 degrees, i.e. turning the x-axis into the y-axis and vice versa. This makes the labels much easier to read.

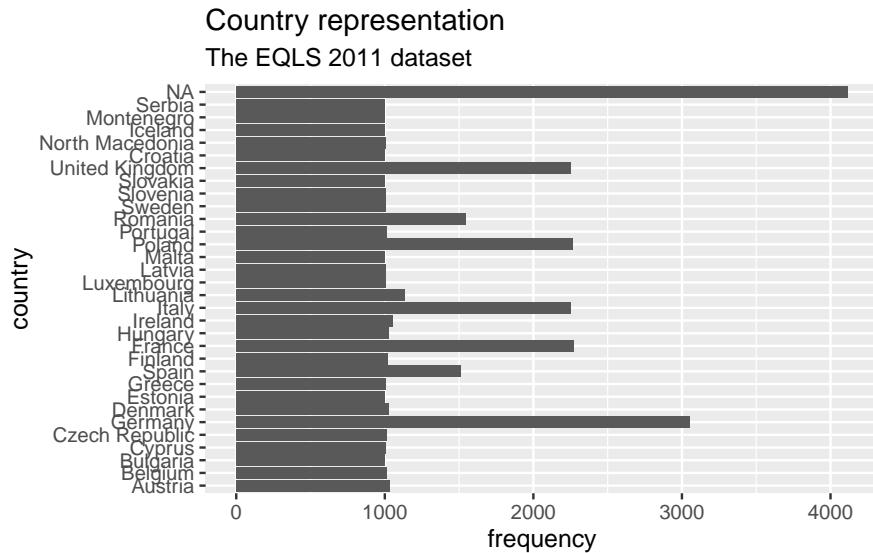
```
ggplot(eqls_2011, aes(x = country_name)) +
  geom_bar() +
  coord_flip()
```



Now we can see that `NA` is by far the longest bar, followed by `Germany` and `France`, `Italy`, `Poland` and the `United Kingdom`. Thus, a large amount of participants did not indicate which country they are from. Also, it seems there are much more Germans included in this dataset than other countries. For this large dataset this would not necessarily constitute a threat to representativeness of the data. It is, however, concerning that for a large group of participants the country-of-origin remains unknown. If we were to analyse the data further, it would be good to acknowledge such a shortcoming.

Our plot, while informative, is not finished, because it lacks formatting. Plots intended for any publication should be self-explanatory and reader-friendly. We can use `+` to add other elements to our plot, such as a title, subtitle, and proper axes labels. Here are some common functions to further customise our plot:

```
ggplot(eqls_2011, aes(x = country_name)) +
  geom_bar() +
  coord_flip() +
  ggtitle("Country representation",           # Add a title
          subtitle = "The EQLS 2011 dataset") + # Add a sub-title
  xlab("country") +                         # Rename x-axis
  ylab("frequency")                         # Rename y-axis
```



Our plot is almost perfect, but we should take one more step to make reading and understanding this plot even easier. At the moment, the bars are ordered randomly. Unfortunately, this is hardly ever a useful way to order your data. Instead, we might want to sort the data by frequency, showing the largest bar at the top. To achieve this, we could either sort the countries by hand (see Section 7.5.2) or slightly amend what we have coded so far.

The problem you encounter when rearranging a `geom_bar()` with only one variable is that we do not have an explicit value to indicate how we want to sort the bars. Our current code is based on the fact that `ggplot` does the counting for us. So, instead, we need to do two things:

- create a table with all countries and their frequency, and
- use this table to plot the countries by the frequency we computed

```
# Step 1: The frequency table only
eqls_2011 |> count(country_name)
```

```
# A tibble: 32 x 2
  country_name     n
  <fct>       <int>
1 Austria      1032
2 Belgium      1013
3 Bulgaria     1000
4 Cyprus        1006
5 Czech Republic 1012
6 Germany      3055
```

```

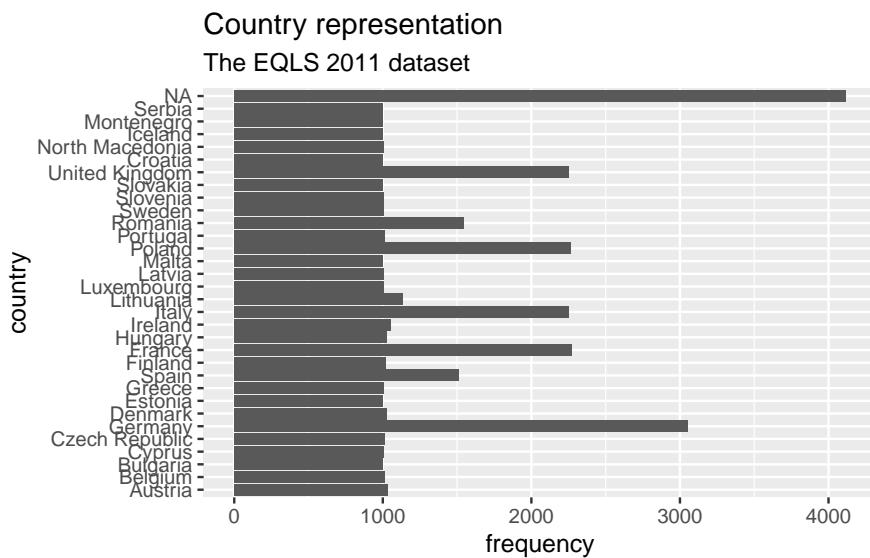
7 Denmark      1024
8 Estonia       1002
9 Greece        1004
10 Spain        1512
# i 22 more rows

```

```

# Step 2: Plotting a barplot based on the frequency table
eqls_2011 |>
  count(country_name) |>
  ggplot(aes(x = country_name, y = n)) +
    # Use geom_col() instead of geom_bar()
    geom_col() +
    # Rotate plot by 180 degrees
    coord_flip() +
    # Add titles for plot
    ggtitle("Country representation",           # Add a title
            subtitle = "The EQLS 2011 dataset") + # Add a sub-title
    xlab("country") +                         # Rename x-axis
    ylab("frequency")                        # Rename y-axis

```

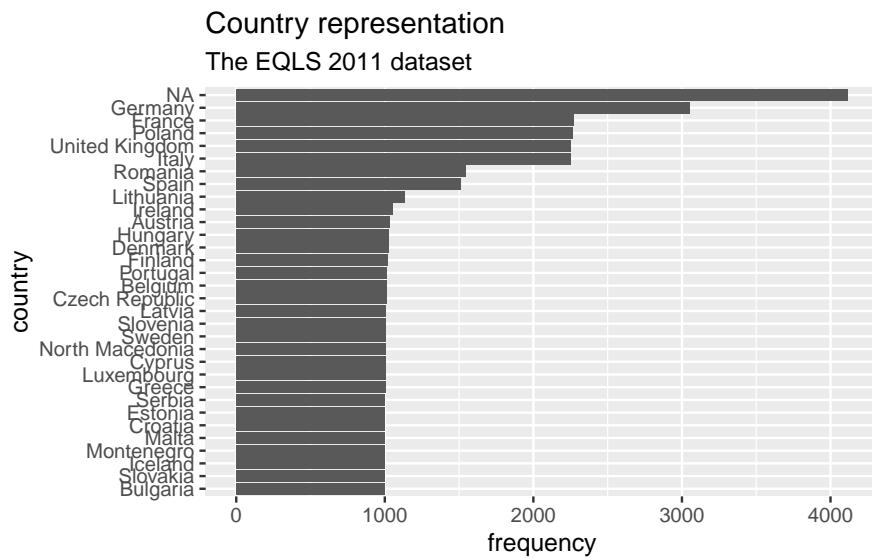


```

#Step 3: reorder() country_name by frequency, i.e. by 'n'
eqls_2011 |>

```

```
count(country_name) |>
  ggplot(aes(x = reorder(country_name, n),
             y = n)) +
  geom_col() +
  coord_flip() +
  ggtitle("Country representation",
          subtitle = "The EQLS 2011 dataset") +
  xlab("country") +
  ylab("frequency")
```



Step 3 is the only code you need to create the desired plot. The other two steps only demonstrate how one can slowly build this plot, step-by-step. You might have noticed that I used `dplyr` to chain all these functions together (i.e. `|>`), and therefore, it was not necessary to specify the dataset in `ggplot()`.

There are also two new functions we had to use: `geom_col()` and `reorder()`. The function `geom_col()` performs the same step as `geom_bar()` which can be confusing. The easiest way to remember how to use them is: If you use a frequency table to create your barplot, use `geom_col()`, if not, use `geom_bar()`. The function `geom_col()` requires that we specify an x-axis and a y-axis (our frequency scores), while `geom_bar()` works with one variable.

In many cases, when creating plots, you have to perform two steps:

- generate the statistics you want to plot, e.g. a frequency table, and
- plot the data via `ggplot()`

Now you have learned the fundamentals of plotting in *R*. We will create a lot more plots throughout the next chapters. They all share the same basic structure, but we will use different `geom`s to describe our data. By the end of this book, you will have accrued enough experience in plotting in *R* that it will feel like second nature. If you want to deepen your knowledge of `ggplot`, you should take a look at the book ‘`ggplot`: Elegant Graphics for Data Analysis’² or ‘`R` Graphics Cookbook’³ which moves beyond `ggplot2`. Apart from that, you can also find fantastic packages which extend the range of ‘`geoms`’ you can use in the ‘`ggplot2` extensions gallery’⁴.

8.2 Frequencies and relative frequencies: Counting categorical variables

One of the most fundamental descriptive statistics includes frequencies, i.e. counting the number of occurrences of a factor level, string or numeric value. We already used the required function to compute such a frequency when we looked at the representation of countries in the `eqls_2011` dataset (see Section 8.1), i.e. `count()`.

```
eqls_2011 |> count(country_name)
```

```
# A tibble: 32 x 2
  country_name     n
  <fct>        <int>
  1 Austria      1032
  2 Belgium      1013
  3 Bulgaria     1000
  4 Cyprus       1006
  5 Czech Republic 1012
  6 Germany      3055
  7 Denmark      1024
  8 Estonia       1002
  9 Greece        1004
  10 Spain        1512
# i 22 more rows
```

The result indicates that our participants are from 32 different countries. If we are only interested in top countries, we could `filter()` our dataset and include only countries that had more than 1500 participants. My choice of

²<https://ggplot2-book.org>

³<https://r-graphics.org>

⁴<https://exts.ggplot2.tidyverse.org/gallery/>

`1500` is entirely arbitrary and you can change the value to whatever you prefer and is meaningful to your analysis.

```
eqls_2011 |>
  count(country_name) |>
  filter(n >= 1500)

# A tibble: 8 x 2
  country_name     n
  <fct>           <int>
1 Germany          3055
2 Spain             1512
3 France            2270
4 Italy              2250
5 Poland             2262
6 Romania            1542
7 United Kingdom    2252
8 <NA>              4119
```

There are in total 7 countries which had more than `1500` people represented in the dataset.

Besides *absolute frequencies*, i.e. the actual count of occurrences of a value, we sometimes wish to compute *relative frequencies* to make it easier to compare categories more easily. *Relative frequencies* are more commonly known as *percentages* which indicate the ratio or proportion of an occurrence relative to the total number of observations. Let's have a look at `employment` in the `eqls_2011` dataset and find out how many participants were employed, retired, a student, etc..

```
eqls_2011 |> count(employment)

# A tibble: 7 x 2
  employment           n
  <fct>             <int>
1 Student              2490
2 Retired              12853
3 Unemployed            3540
4 Unable to work - disability/illness   897
5 Employed (includes on leave)      19376
6 Other                  666
7 Homemaker             3814
```

The table indicates that `Employed (includes on leave)` and `Retired` constitute the largest categories. In a next step we can compute the absolute and relative

frequencies at once by creating a new variable with `mutate()` and the formula to calculate percentages, i.e. `n / sum(n)`.

```
eqls_2011 |>
  count(employment) |>

  # add the relative distributions, i.e. percentages
  mutate(perc = n / sum(n))

# A tibble: 7 x 3
  employment      n     perc
  <fct>     <int>   <dbl>
1 Student        2490  0.0571
2 Retired       12853  0.295 
3 Unemployed    3540  0.0811
4 Unable to work - disability/illness 897  0.0206
5 Employed (includes on leave) 19376  0.444 
6 Other          666  0.0153
7 Homemaker     3814  0.0874
```

It seems we have a very uneven distribution of employment categories. Thus, our analysis is affected by having more 44.4% of employed and 29.5% of retired participants in our dataset. Together, these two groups account for 73.9% of our entire sample. Consequently, any further computation we perform with this dataset would predominantly reflect the opinion of `Employed (includes on leave)` and `Retired` survey takers. This is important to know because it limits the representativeness of our results. The insights gained from primarily employed participants in this study will not be helpful when trying to understand unemployed individuals in the population. Therefore, understanding who the participants are in your study is essential before undertaking any further analysis. Sometimes, it might even imply that we have to change the focus of our study entirely.

8.3 Central tendency measures: Mean, Median, Mode

The *mean*, *median* and *mode* (the 3 Ms) are all measures of central tendency, i.e. they provide insights into how our data is distributed. Measures of central tendency help to provide insights into the most frequent/typical scores we can observe in the data for a given variable. All the 3Ms summarise our data/variable by a single score which can be helpful but sometimes also terribly misleading.

8.3.1 Mean

The mean is likely the most known descriptive statistics and, at the same time, a very powerful and influential one. For example, the average ratings of restaurants on Google might influence our decision on where to eat out. Similarly, we might consider the average rating of movies to decide which one to watch in the cinema with friends. Thus, what we casually refer to as the ‘average’ is equivalent to the ‘mean’.

In *R*, it is simple to compute the mean using the function `mean()`. We used this function in Section 5.3 and Section 7.8 already. However, we have not looked at how we can plot the mean.

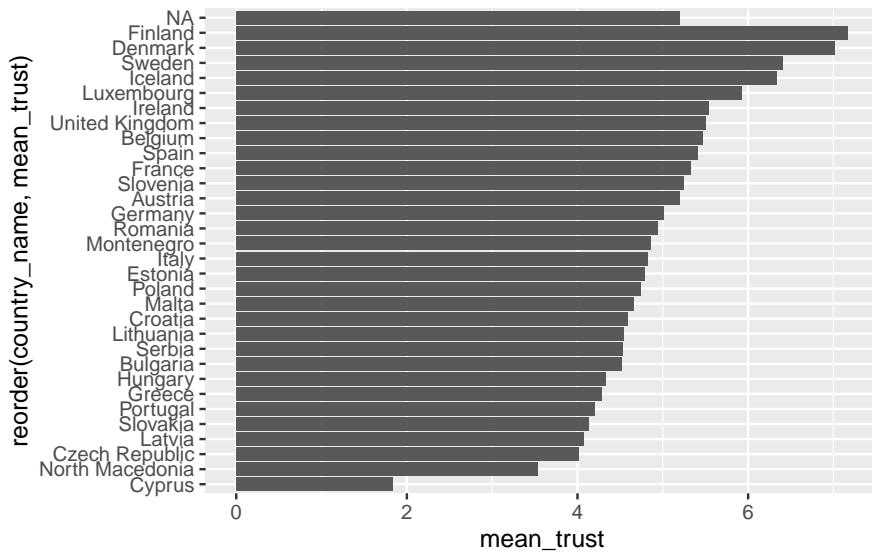
Assume we are curious to know in which country people feel they can trust others the most, i.e. a similar question we raised in Chapter Section 8.1 . The mean could be a good starting point to answer this question because it provides the ‘average’ level of trust people have in others on a scale from 1 (*You cannot be too careful*) to 10 (*Most people can be trusted*). The simplest approach to investigating this is using a bar plot, like in Section 8.1. So, we first create a table that contains the means of the variable `trust_people` for each country in `country_name`. Then we use this table to plot a bar plot with `geom_col()`.

```
eqls_2011 |>

  # Group data by country
  group_by(country_name) |>

  # Compute the mean for each group (remove NAs via na.rm = TRUE)
  summarise(mean_trust = mean(trust_people, na.rm = TRUE)) |>

  # Create the plot
  ggplot(aes(x = reorder(country_name, mean_trust), y = mean_trust)) +
  geom_col() +
  coord_flip()
```



You will have noticed that we use the function `summarise()` instead of `mutate()`. The `summarise()` function is a special version of `mutate()`. While `mutate()` returns a value for each row, `summarise` condenses our dataset, e.g. turning each row of observations into scores for each genre. Here is an example of a simple dataset which illustrates the difference between these two functions.

```
# Create a simple dataset from scratch with tibble()
(data <- tibble(number = c(1, 2, 3, 4, 5),
                 group = factor(c("A", "B", "A", "B", "A"))))

# A tibble: 5 × 2
  number group
  <dbl> <fct>
1     1 A
2     2 B
3     3 A
4     4 B
5     5 A

# Return the group value for each observation/row
data |>
  group_by(group) |>
  mutate(sum = sum(number))

# A tibble: 5 × 3
# Groups:   group [2]
  number group  sum
  <dbl> <fct> <dbl>
1     1 A      1.0
2     2 B      2.0
3     3 A      3.0
4     4 B      4.0
5     5 A      5.0
```

```

number group   sum
<dbl> <fct> <dbl>
1     1 A      9
2     2 B      6
3     3 A      9
4     4 B      6
5     5 A      9

# Returns the group value once for each group
data |>
  group_by(group) |>
  summarise(sum = sum(number))

```

```

# A tibble: 2 x 2
  group   sum
  <fct> <dbl>
1 A      9
2 B      6

```

Considering the results from our country plot, it appears as if Finland and Denmark are far ahead of the rest. On average, both countries score around 7/10 with regards to being able to trust other people in their country. We can retrieve the exact means by removing the plot from the above code.

```

eqls_2011 |>
  group_by(country_name) |>
  summarise(mean_trust = mean(trust_people, na.rm = TRUE)) |>
  arrange(desc(mean_trust))

# A tibble: 32 x 2
  country_name   mean_trust
  <fct>           <dbl>
1 Finland        7.17
2 Denmark        7.01
3 Sweden          6.41
4 Iceland         6.33
5 Luxembourg      5.92
6 Ireland          5.54
7 United Kingdom  5.50
8 Belgium          5.47
9 Spain            5.41
10 France          5.32
# i 22 more rows

```

In the last line, I used a new function called `arrange()`. It allows us to sort

rows in our dataset by a specified variable (i.e. a column). By default, `arrange()` sorts values in ascending order, putting the country with the highest `mean_trust` last. Therefore, we have to use another function to change the order to descending with `desc()` to see the country with the highest score first. The function `arrange()` works similarly to `reorder()` but is used for sorting variables in data frames and less so for plots.

Lastly, I want to show you another very handy function that allows you to create tables and plots which only feature the Top 3/5/10 countries in our list. This can sometimes be handy if there are many categories, but you want to focus the attention of your analysis on the biggest ones (or the smallest ones). Here is what we can do:

```
eqls_2011 |>
  group_by(country_name) |>
  summarise(mean_trust = mean(trust_people, na.rm = TRUE)) |>
  slice_max(order_by = mean_trust,
            n = 5)

# A tibble: 5 x 2
  country_name mean_trust
  <fct>          <dbl>
1 Finland        7.17
2 Denmark        7.01
3 Sweden         6.41
4 Iceland         6.33
5 Luxembourg     5.92
```

The new function from `dplyr` called `slice_max()` allows us to pick the top 5 countries in our data. So, if you have many rows in your data frame (remember there are 32 countries), you might want to be ‘picky’ and report only the top 3, 4 or 5. As you can see, `slice_max()` requires at least two arguments: `order_by` which defines the variable your dataset should be sorted by, and `n` which defines how many rows should be selected, e.g. `n = 3` for the top 3 countries or `n = 10` for the top 10 countries. If you want to pick the lowest observations in your dataframe, you can use `slice_min()`. There are several other ‘slicing’ functions that can be useful and can be found on the corresponding `dplyr` website⁵.

In conclusion, Nordic countries (`Finland`, `Denmark`, `Sweden` and `Iceland`) report the highest levels of trust in other people in Europe. However, even though the mean is a great tool to understand tendencies in your data, it can be heavily affected by other factors and lead to misleading results. Some of these issues will be exemplified and discussed in Chapter 9.

⁵<https://dplyr.tidyverse.org/reference/slice.html>

8.3.2 Median

The ‘*median*’ is the little, for some, lesser-known and used brother of the ‘*mean*’. However, it can be a powerful indicator for central tendency because it is not so much affected by outliers. With outliers, I mean observations that lie way beyond or below the average observation in our dataset.

Assume we are curious to find out whether people in Germany, France or Greece work longer (not harder!) hours per week. We first `filter()` our dataset to only show participants who are from one of the selected countries and provided information about their working hours. The latter we can achieve by using the opposite (i.e. !) of the function `is.na()` for the variable `working_hrs`.

```
# Select only Nordic countries with no missing data for 'work_hrs'
eqls_comp <-
  eqls_2011 |>
  filter(country_name == "Germany" |
         country_name == "Finland" |
         country_name == "Greece"
    ) |>
  filter(!is.na(work_hrs))
```

We now can compute the means for each country and obtain the average working hours, like we did in our previous example.

```
# Mean working hours for Nordic countries
eqls_comp |>
  group_by(country_name) |>
  summarise(mean = mean(work_hrs, na.rm = TRUE)) |>
  arrange(desc(mean))

# A tibble: 3 × 2
  country_name   mean
  <fct>       <dbl>
1 Greece        44.3
2 Finland       40.4
3 Germany       38.1
```

Based on these insights, we have to conclude that **Greece** is the country where people work the longest, up to about 6 hours more per week than in **Germany**. However, if use a different measure of central tendency, for example the median, the results change drastically.

```
# Median working hours for Nordic countries
eqls_comp |>
```

```
group_by(country_name) |>
summarise(median = median(work_hrs, na.rm = TRUE)) |>
arrange(desc(median))

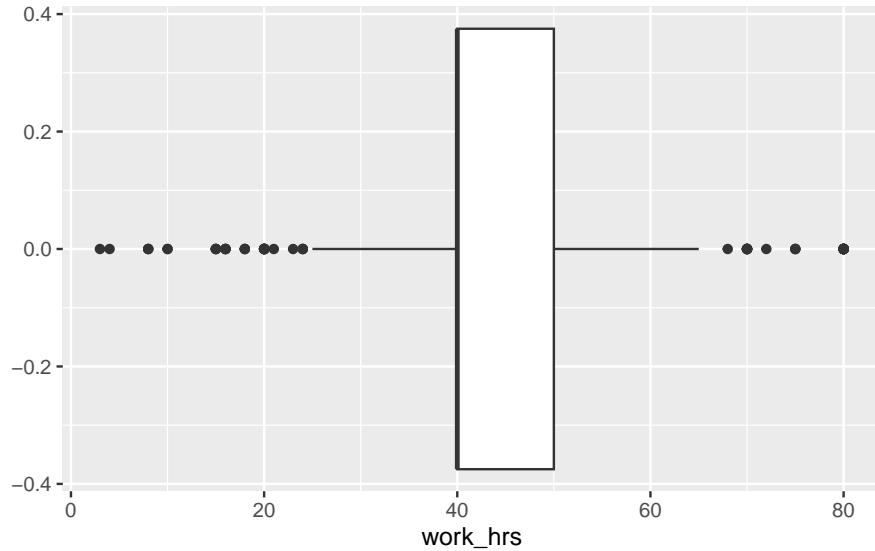
# A tibble: 3 × 2
  country_name median
  <fct>        <dbl>
1 Germany       40
2 Greece        40
3 Finland       40
```

The result is striking. The results based on the median suggest that all three countries work the same amount of working hours. This is because the median sorts a dataset, e.g. by `work_hrs` and then picks the value that would cut the data into two equally large halves. It does not matter which value is the highest or lowest in our dataset. What matters is the value that is ranked right in the middle of all values. The median splits your dataset into two equally large datasets.

You might feel confused now and wonder which measure you should choose to accurately report your research findings. In general, when we report means, it is advisable to report the median as well. If a mean and median differ substantially, it could imply that your data ‘suffers’ from outliers. So we have to detect them and think about whether we should remove them for further analysis (see Section 9.6). I hope it is evident that reporting the mean only could lead to wrong conclusions because of data that is biased.

We can visualise medians using boxplots. Boxplots are a very effective tool to show how your data is distributed in one single data visualisation, and it offers more than just the median. It also shows the spread of your data (see Section 8.4) and possible outliers. Let’s create one for `Greece` to better understand why the differences between mean and median are so large.

```
eqls_comp |>
filter(country_name == "Greece") |>
ggplot(aes(x = work_hrs)) +
geom_boxplot()
```



To interpret this boxplot consider Figure Figure 8.1. Every boxplot consists of a ‘box’ and two whiskers (i.e. the lines leading away from the box). The distribution of data can be assessed by looking at different ranges:

- `MINIMUM` to `Q1` represents the bottom 25% of our observations,
- `Q1` to `Q3`, also known as the `interquartile range` (IQR) defines the middle 50% of our observations, and
- `Q3` to `MAXIMUM`, contains the top 25% of all our data.

Thus, with the help of a boxplot, we can assess whether our data is evenly distributed across these ranges or not. If observations fall outside a certain range (i.e. $1.5 * \text{IQR}$) they are classified as outliers. We cover outliers in great detail in Section 9.6.

Considering our boxplot, it shows that we have about 16 observations that are considered outliers. If we wanted to see each of the participants’ score mapped on top of this boxplot, we can overlay another `geom_`. We can represent each participant as a point by using `geom_point()`. This function requires us to define the values for the x and y-axis. Here it makes sense to set `y = 0`, which aligns all the dots in the middle of the boxplot.

```
eqls_comp |>
  filter(country_name == "Greece") |>
  ggplot(aes(x = work_hours)) +
  geom_boxplot() +
```

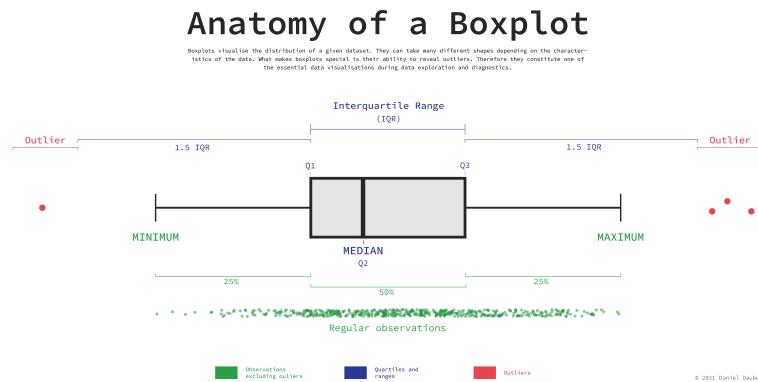
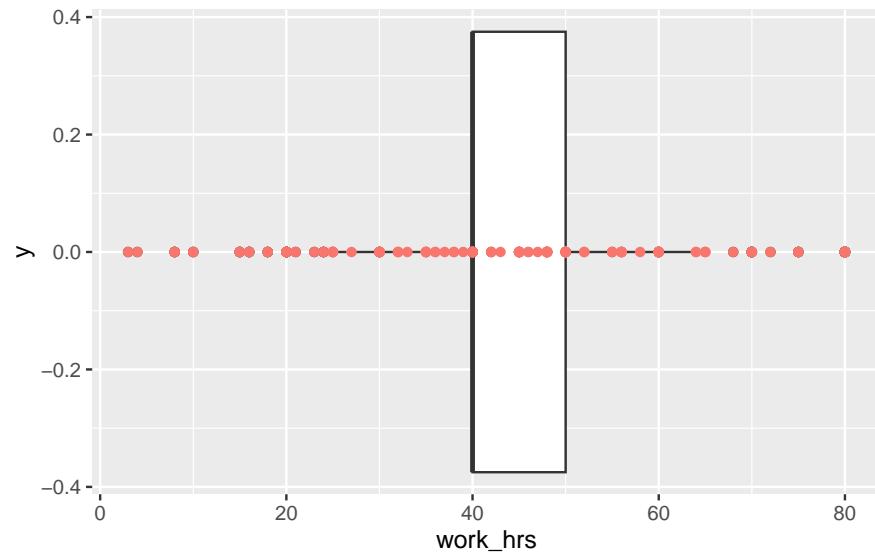


Figure 8.1: Anatomy of a Boxplot

```
# Add an additional layer of data visualisation
geom_point(aes(y = 0, col = "red"),
            show.legend = FALSE)
```



To make the dots stand out more, I changed the colour to "red". By adding the `col` attribute (`color` and `colour` also work), ggplot2 would automatically

generate a legend. Since we do not need it, we can specify it directly in the `geom_point()` function. If you prefer the legend, remove `show.legend = FALSE`.

This plot nicely demonstrates why boxplots are so popular and helpful: They provide so many insights, not only into the central tendency of a variable, but also highlight outliers and, more generally, give a sense of the spread of our data (more about this in Section 8.4).

In conclusion, the median is an important descriptive and diagnostic statistic and should be included in most empirical quantitative studies. Failing to do so could not only harm the reliability of your results, but also your reputation as an analyst.

8.3.3 Mode

Finally, the ‘mode’ indicates which value is the most frequently occurring value for a specific variable. This can apply to any type of variable, numeric, categorical, etc. and makes it a special indicator of central tendency. For example, we might be interested in knowing which age category is the most frequently represented one in our `eqls_2011` dataset.

When trying to compute the mode in *R*, we quickly run into a problem because there is no function available to do this straight away unless you search for a package that does it for you. However, before you start searching, let’s reflect on what the mode does and why we can find the mode without additional packages or functions. As a matter of fact, we found the mode multiple times in this chapter already, but we were just not aware of it. The mode is based on the frequency of the occurrence of a value. Thus, the most frequently occurring value would be the one that is listed at the top of a frequency table. We have already created several frequency tables in this book, and we can create another one to find the answer to our question.

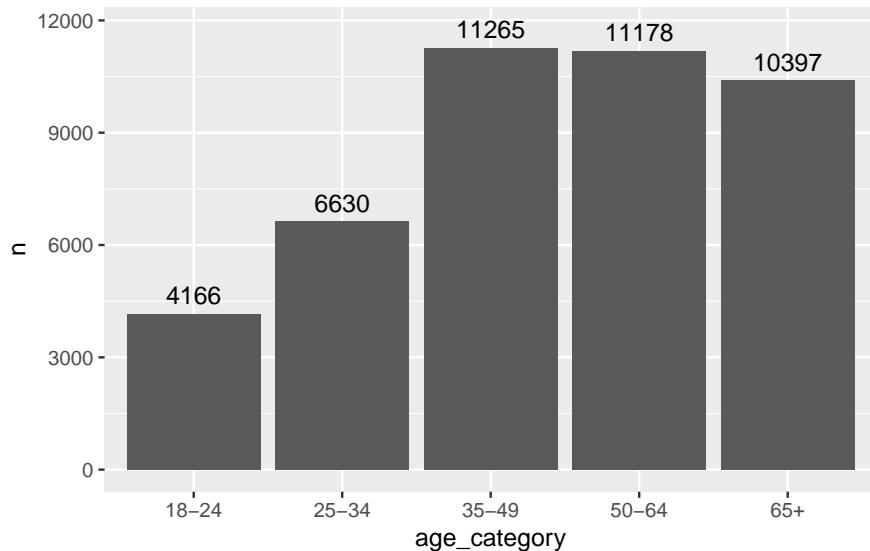
```
eqls_2011 |> count(age_category)
```

```
# A tibble: 5 x 2
  age_category     n
  <ord>        <int>
1 18-24          4166
2 25-34          6630
3 35-49          11265
4 50-64          11178
5 65+            10397
```

We can also easily visualise this frequency table in the same way as before. To make the plot a bit more ‘fancy’, we can add labels to the bar which reflect the frequency of the rating. We need to add the attribute `label` and also add

a `geom_text()` layer to display them. Because the numbers would overlap with the bars, I ‘nudge’ the labels up by 500 units on the y-axis.

```
eqls_2011 |>
  count(age_category) |>
  ggplot(aes(x = age_category,
             y = n,
             label = n)) +
  geom_col() +
  geom_text(nudge_y = 500)
```



The frequency table and the plot reveal that the mode for `age_category` is 35-49. In addition, we also get to know how many participants fall into this category, i.e. 11265. The table and plot provide much more information than receiving only a single score and helps better interpret the importance of the mode as an indicator for a central tendency. Consequently, there is generally no need to compute the mode if you can have a frequency table instead. Still, if you are keen to have a function that computes the mode, you will have to write your own function, e.g. as shown in this post on stackoverflow.com⁶ or search for a package that coded one already.

As a final remark, it is also possible that you can find two or more modes in your data. For example, if the age category 50-64 also included 11265 participants, both would be considered a mode.

⁶<https://stackoverflow.com/questions/2547402/how-to-find-the-statistical-mode>

8.4 Indicators and visualisations to examine the spread of data

Understanding how your data is spread out is essential to get a better sense of what your data is composed of. We already touched upon the notion of spread in Section 8.3.2 through plotting a boxplot. Furthermore, the spread of data provides insights into how homogeneous or heterogeneous our participants' responses are. The following will cover some essential techniques to investigate the spread of your data and investigate whether our variables are normally distributed, which is often a vital assumption for specific analytical methods (see Chapter 9). In addition, we will aim to identify outliers that could be detrimental to subsequent analysis and significantly affect our modelling and testing in later stages.

8.4.1 Boxplot: So much information in just one box

The boxplot is a staple in visualising descriptive statistics. It offers so much information in just a single plot that it might not take much to convince you that it has become a very popular way to show the spread of data.

For example, we might be interested to know how much alcohol people drink on average. We could imagine that there are some people who barely drink alcohol, some who might drink a moderate amount and others who consume an unhealthy amount. If we assumed that a person drinks about 250-500 ml of alcohol every month, e.g. a beer, we could expect an average consumption of 3-6 litres over the course of a year. One approach to get a definitive answer is a boxplot, which we used before and the dataset `alcohol_2019` in the `r4np` package.

```
# Text for annotations
most_alcohol <-
  alcohol_2019 |>
  filter(consumption == max(consumption)) |>
  select(country)

least_alcohol <-
  alcohol_2019 |>
  filter(consumption == min(consumption)) |>
  select(country)

# Creating a single string of country names
# because there are multiple countries sharing
# the minimum score.
```

```

least_alcohol_as_string <- paste(least_alcohol$country, collapse = "\n")

# Create the plot
alcohol_2019 |>
  ggplot(aes(x = consumption)) +
  geom_boxplot() +
  annotate("text",
    label = glue:::glue("{most_alcohol$country}
({max(alcohol_2019$consumption)} litres)", 
    x = 16,
    y = 0.05,
    size = 2.5) +
  annotate("text",
    label = glue:::glue("{least_alcohol_as_string}
({min(alcohol_2019$consumption)} litres)", 
    x = 0,
    y = 0.1,
    size = 2.5)
  
```

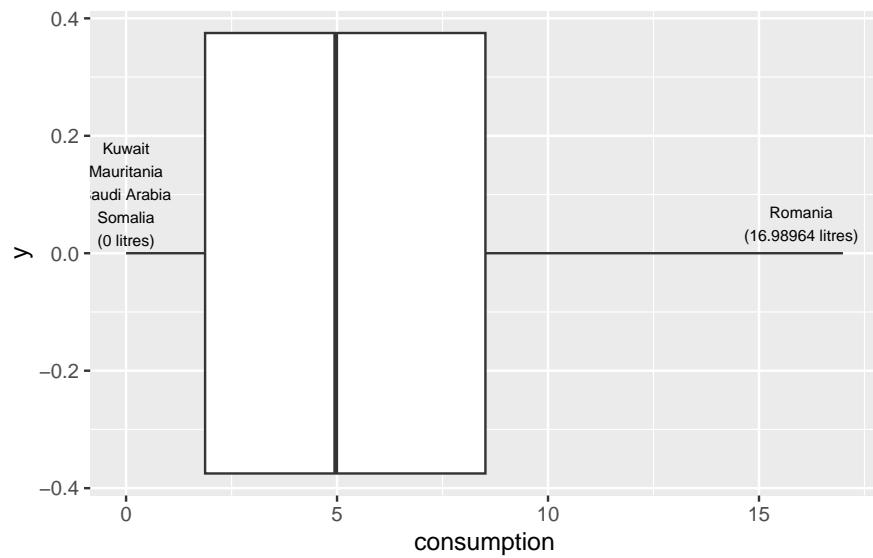


Figure 8.2: A boxplot

The results indicate that most people drink between 2 and 8 litres of alcohol. Our intuition was correct. We find that one country reports a much larger amount of alcohol consumption, i.e. almost 20 litres: Romania. In contrast, the

least amount of alcohol is consumed in Kuwait, Mauritania, Saudi Arabia and Somalia, where people seem to drink no alcohol at all. I added annotations using `annotate()` to highlight these countries in the plot. A very useful package for annotations is `glue`, which allows combining text with data to label your plots. So, instead of looking up the countries with the highest and lowest alcohol consumption, I used the `filter()` and `select()` functions to find them automatically. This has the great advantage that if I wanted to update my data, the name of the countries might change. However, I do not have to look it up again by hand and my plot will reflect the changes automatically.

As we can see from our visualisation, none of the countries would count as outliers in our dataset (see Section 9.6.2 for more information on the computation of outliers).

8.4.2 Histogram: Do not mistake it as a bar plot

Another frequently used approach to show the spread (or distribution) of data is the histogram. The histogram easily gets confused with a bar plot. However, you would be very mistaken to assume that they are the same. Some key differences between these two types of plots is summarised in Table 8.1.

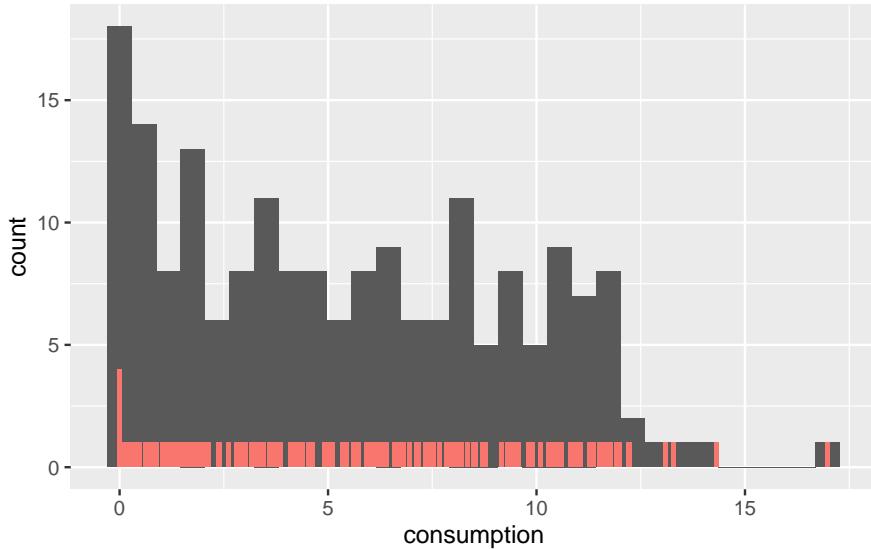
Table 8.1: Histogram vs bar plot

Histogram	Bar plot
Used to show the distribution of non-categorical data	Used for showing the frequency of categorical data, i.e. <code>factors</code>
Each bar (also called ‘bin’) represents a group of observations.	Each bar represents one category (or level) in our <code>factor</code> .
The order of the bars is important and cannot/should not be changed.	The order of bars is arbitrary and can be reordered if meaningful.

Let’s overlap a bar plot with a histogram for the same variable to make this difference even more apparent.

```
alcohol_2019 |>
  ggplot(aes(x = consumption)) +
  geom_histogram() +
  geom_bar(aes(fill = "red"), width = 0.11, show.legend = FALSE)

`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
Warning: `position_stack()` requires non-overlapping x intervals.
```



There are a couple of important observations to be made:

- The bar plot has much shorter bars because each bar represents the frequency of a single unique score. Thus, the alcohol consumption of 2.5 is represented as a bar, as is the alcohol consumption of 2.7. Only identical observations are grouped together. As such, the bar plot is based on a frequency table, similar to what we computed before.
- In contrast, the histogram's 'bars' are higher because they group together individual observations based on a specified range, e.g. one bar might represent alcohol consumption between 2 and 3 litres. These ranges are called **bins**.

We can control the number of bars in our histogram using the `bins` attribute. `ggplot` even reminds us in a warning that we should adjust it to represent more details in the plot. Let's experiment with this setting to see how it would look like with different numbers of `bins`.

```
alcohol_2019 |>
  ggplot(aes(consumption)) +
  geom_histogram(bins = 5) +
  geom_bar(aes(fill = "red"), width = 0.11, show.legend = FALSE) +
  ggtitle("bins = 5")
```

Warning: `position_stack()` requires non-overlapping x intervals.

```
alcohol_2019 |>
  ggplot(aes(consumption)) +
  geom_histogram(bins = 20) +
  geom_bar(aes(fill = "red"), width = 0.11, show.legend = FALSE) +
  ggtitle("bins = 20")
```

Warning: `position_stack()` requires non-overlapping x intervals.

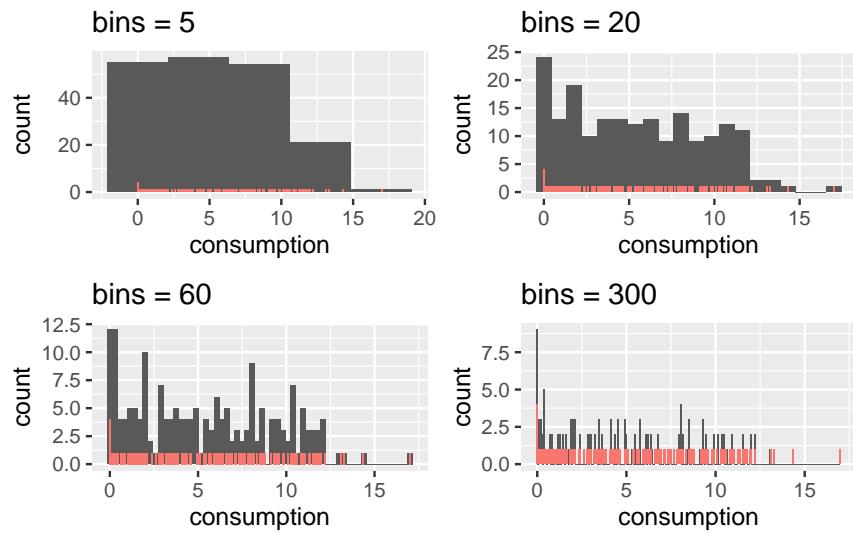
```
alcohol_2019 |>
  ggplot(aes(consumption)) +
  geom_histogram(bins = 60) +
  geom_bar(aes(fill = "red"), width = 0.11, show.legend = FALSE) +
  ggtitle("bins = 60")
```

Warning: `position_stack()` requires non-overlapping x intervals.

```
alcohol_2019 |>
  ggplot(aes(consumption)) +
  geom_histogram(bins = 300) +
  geom_bar(aes(fill = "red"), width = 0.11, show.legend = FALSE) +
  ggtitle("bins = 300")
```

Warning: `position_stack()` requires non-overlapping x intervals.

Warning: `position_stack()` requires non-overlapping x intervals.



As becomes evident, if we select a large enough number of bins, we can achieve almost the same result as a bar plot. This is the closest a histogram can become to a bar plot, i.e. if you define the number of `bins` so that each observation is captured by one `bin`. While theoretically possible, practically, this rarely makes much sense.

We use histograms to judge whether our data is normally distributed. A normal distribution is often a requirement for assessing whether we can run certain types of analyses or not (for more details, see Section 9.4). In short: It is imperative to know about it in advance. The shape of a normal distribution looks like a bell (see Figure 8.3). If our data is equal to a normal distribution, we find that

- the mean and the median are the same value and we can conclude that
- the mean is a good representation for our data/variable.

Let's see whether our data is normally distributed using the histogram we already plotted and overlay a normal distribution. The coding for the normal distribution is a little more advanced. Do not worry if you cannot fully decipher its meaning just yet. To draw such a reference plot, we need to:

- compute the `mean()` of our variable,
- calculate the standard deviation `sd()` of our variable (see Section 8.4.6),
- use the function `geom_func()` to plot it and,
- define the function `fun` as `dnorm`, which stands for 'normal distribution'.

It is more important to understand what the aim of this task is, rather than

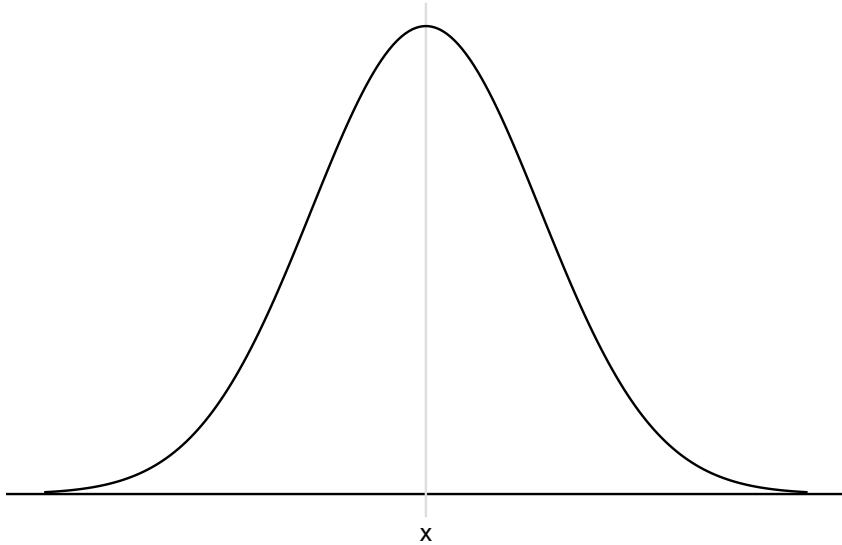
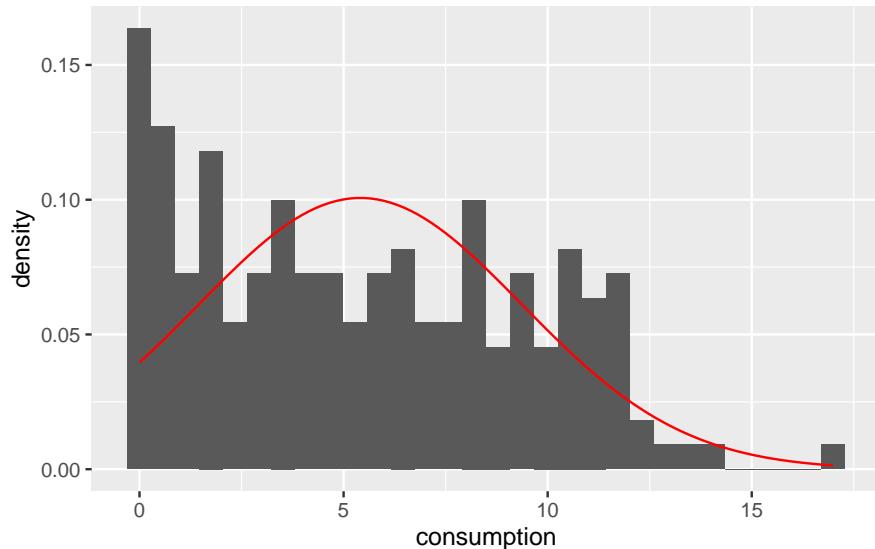


Figure 8.3: A normal distribution

fully comprehending the computational side in *R*: we try to compare our distribution with a normal one.

```
alcohol_2019 |>
  ggplot(aes(consumption)) +
  geom_histogram(aes(y = ..density..), bins = 30) +
  # The following part creates a normal curve based on
  # the mean and standard deviation of our data
  geom_function(fun = dnorm,
    n = 103,
    args = list(mean = mean(alcohol_2019$consumption),
                sd = sd(alcohol_2019$consumption)),
    col = "red")
```



However, there are two problems if we use histograms in combination with a normal distribution reference plot: First, the y-axis needs to be transformed to fit the normal distribution (which is a density plot and not a histogram). Second, the shape of our histogram is affected by the number of `bins` we have chosen, which is an arbitrary choice we make. Besides, the lines of code might be tough to understand because we have to ‘hack’ the visualisation to make it work, i.e. using `aes(y = ..density..)`. There is, however, a better way to do this: Density plots.

8.4.3 Density plots: Your smooth histograms

Density plots are a special form of the histogram. It uses ‘kernel smoothing’, which turns our blocks into a smoother shape. Better than trying to explain what it does, it might help to see it. We use the same data but replace `geom_histogram()` with `geom_density()`. I also saved the plot with the normal distribution in a separate object, so we can easily reuse it throughout this chapter.

```
# Ingredients for our normality reference plot
mean_ref <- mean(alcohol_2019$consumption)
sd_ref <- sd(alcohol_2019$consumption)

# Create a plot with our reference normal distribution
n_plot <-
  alcohol_2019 |>
  ggplot(aes(consumption)) +
  geom_density()
  # Add a normal distribution reference
  stat_function(fun = dnorm,
               args = list(mean = mean_ref, sd = sd_ref),
               color = "red")
```

```

geom_function(fun = dnorm,
              n = 103,
              args = list(mean = mean_ref,
                          sd = sd_ref),
              col = "red")

# Add our density plot
n_plot +
  geom_density()

```

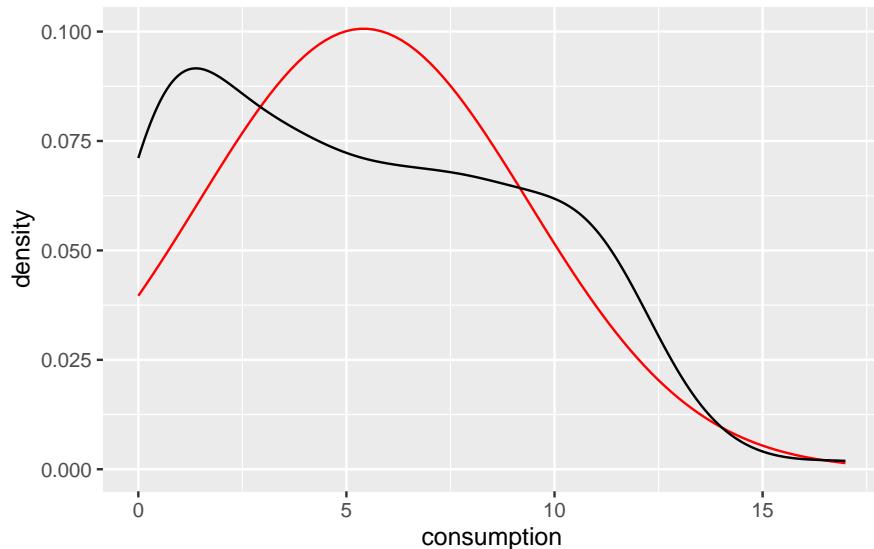


Figure 8.4: Density plot vs normal distribution

We can also define how big the `bins` are for density plots, which make the plot more or less smooth. After all, the density plot is a histogram, but the transitions from one bin to the next are ‘smoothed’. Here is an example of how different `bw` settings affect the plot.

```

# bw = 18
n_plot +
  geom_density(bw = 18) +
  ggtitle("bw = 18")

```

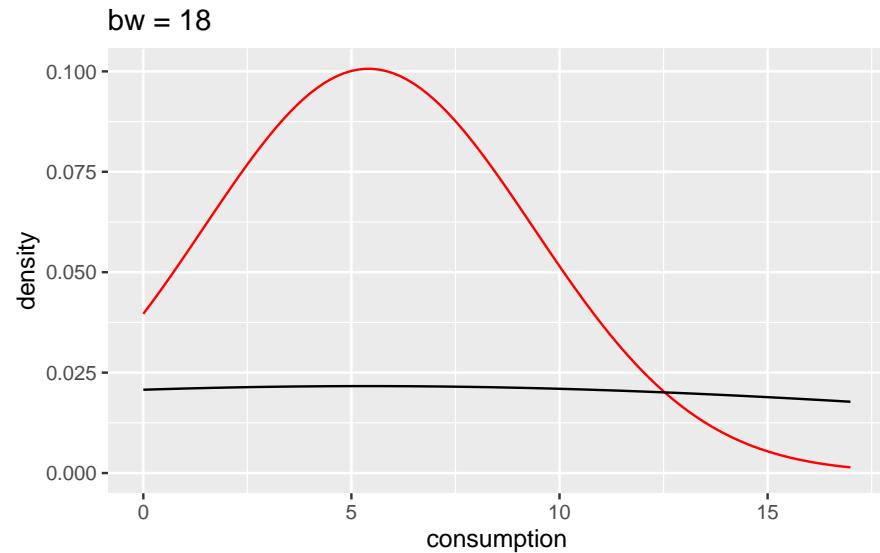


Figure 8.5

```
# bw = 3  
n_plot +  
  geom_density(bw = 3) +  
  ggtitle("bw = 3")
```

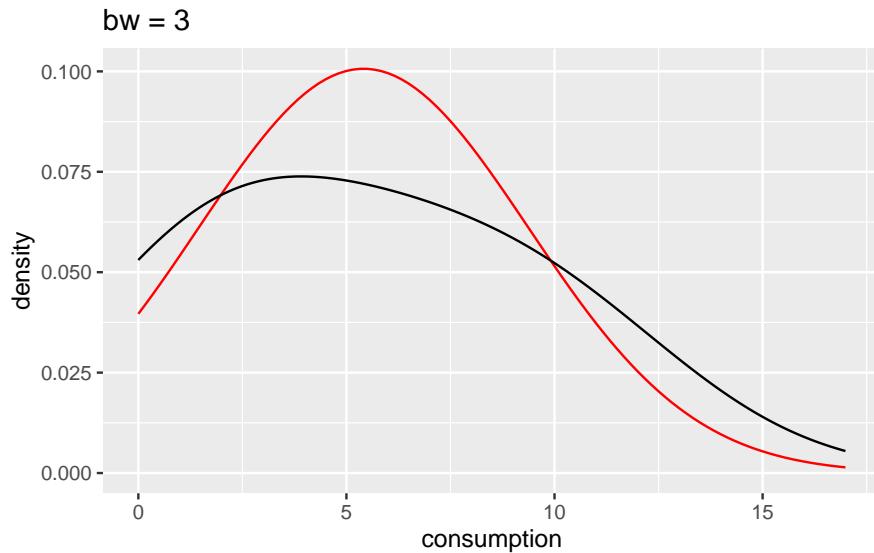


Figure 8.6

The benefits of the density plot in this situation are obvious: It is much easier to see whether our data is normally distributed or not when compared to a reference plot. However, we still might struggle to determine normality just by these plots because it depends on how high or low we set `bw`. In Section 9.4 we will look into more reliable methods of determining the fit of normal distributions to our data.

8.4.4 Violin plot: Your smooth boxplot

If a density plot is the sibling of a histogram, the violin plot would be the sibling of a boxplot, but the twin of a density plot. Confused? If so, then let's use the function `geom_violin()` to create one.

```
alcohol_2019 |>
  ggplot(aes(x = consumption,
             y = 0)) +
  geom_violin()
```

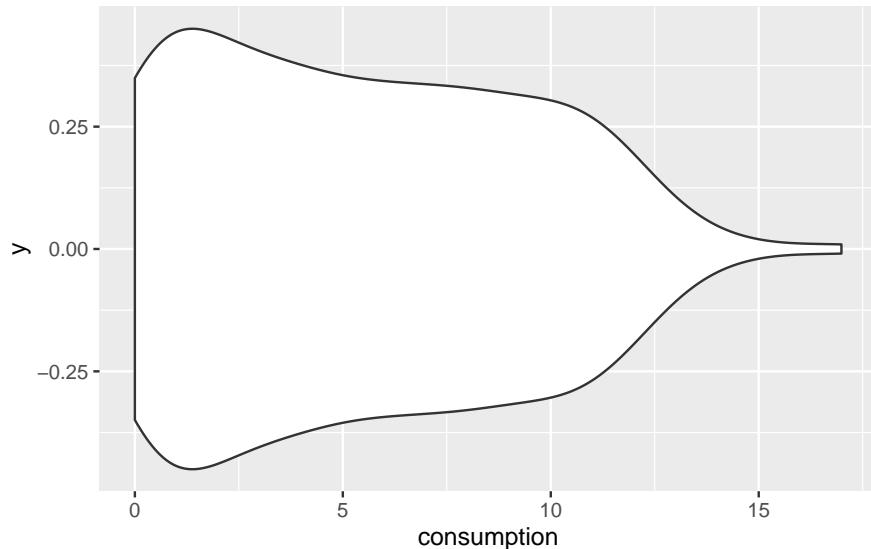


Figure 8.7

Looking at our plot, it becomes evident where the violin plot got its name from, i.e. its shape. The reason why it is also a twin of the density plot becomes clear when we only plot half of the violin with `geom_violinhalf()` from the `see` package.

```
alcohol_2019 |>
  ggplot(aes(x = 0,
             y = consumption)) +
  see::geom_violinhalf() +
  coord_flip()
```

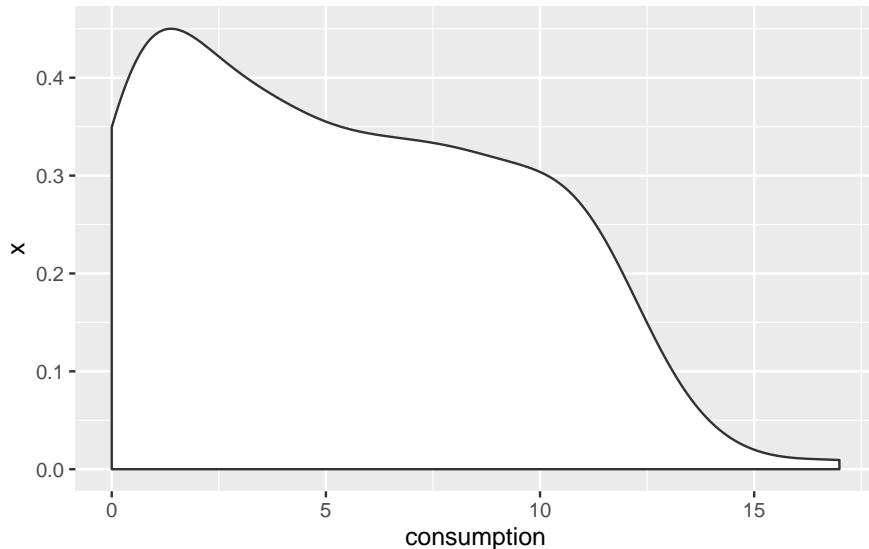


Figure 8.8

It looks exactly like the density plot we plotted earlier (see Figure 8.4). The interpretation largely remains the same to a density plot as well. The relationship between the boxplot and the violin plot lies in the symmetry of the violin plot.

At this point, it is fair to say that we enter ‘fashion’ territory. It is really up to your taste which visualisation you prefer because they are largely similar but offer nuances that some datasets might require. Each visualisation, though, comes with caveats, which results in the emergence of new plot types. For example, ‘*sina*’ plots which address the following problem: A boxplot by itself will never be able to show bimodal distributions like a density plot. On the other hand, a density plot can show observations that do not exist, because they are smoothed. The package *ggforce* enables us to draw a ‘*sina*’ plot which combines a violin plot with a dot plot. Here is an example of overlaying a *sina* plot (black dots) and a violin plot (faded blue violin plot).

```
alcohol_2019 |>
  ggplot(aes(y = consumption, x = 0)) +
  geom_violin(alpha = 0.5, col = "#395F80", fill = "#C0E2FF") +
  ggforce::geom_sina() +
  coord_flip()
```

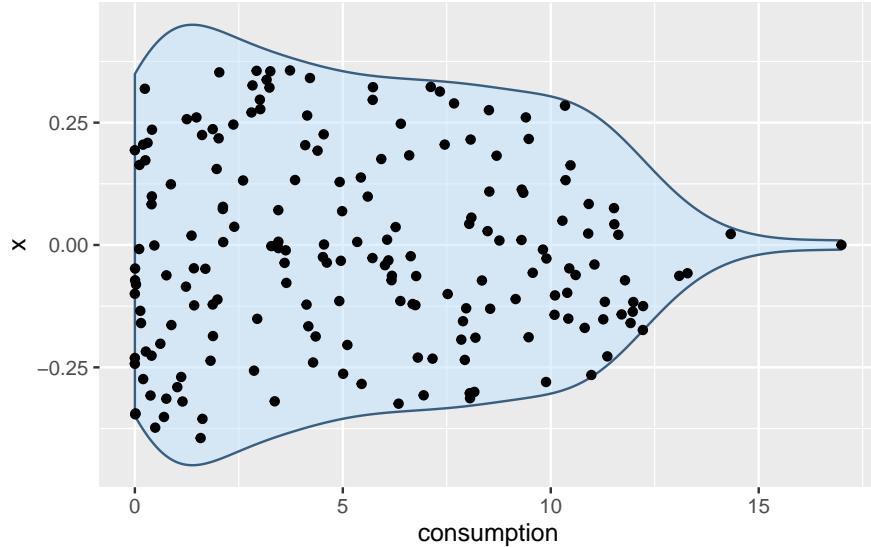


Figure 8.9

Lastly, I cannot finish this chapter without sharing with you one of the most popular uses of half-violin plots: The rain cloud plot. It combines a dot plot with a density plot, and each dot represents a country in our dataset. This creates the appearance of a cloud with raindrops. There are several packages available that can create such a plot. Here I used the `see` package.

```
alcohol_2019 |>
  ggplot(aes(x = year,
             y = consumption,
             fill = as_factor(year))) +
  see::geom_violindot(fill_dots = "blue",
                       size_dots = 8,
                       show.legend = FALSE) +
  see::theme_modern() +
  coord_flip()
```

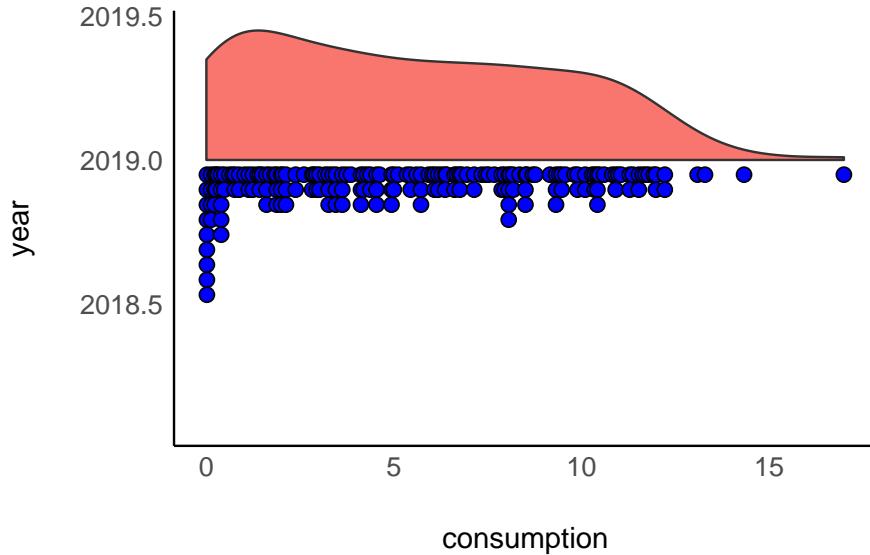


Figure 8.10

8.4.5 QQ plot: A ‘cute’ plot to check for normality in your data

The QQ plot is an alternative to comparing distributions to a normality curve. Instead of a curve, we plot a line that represents our data’s quantiles against the quantiles of a normal distribution. The term ‘quantile’ can be somewhat confusing, especially after learning about the boxplot, which shows quartiles. However, there is a relationship between these terms. Consider the following comparison:

- 1st Quartile = 25th percentile = 0.25 quantile
- 2nd Quartile = 50th percentile = 0.50 quantile = median
- 3rd Quartile = 75th percentile = 0.75 quantile

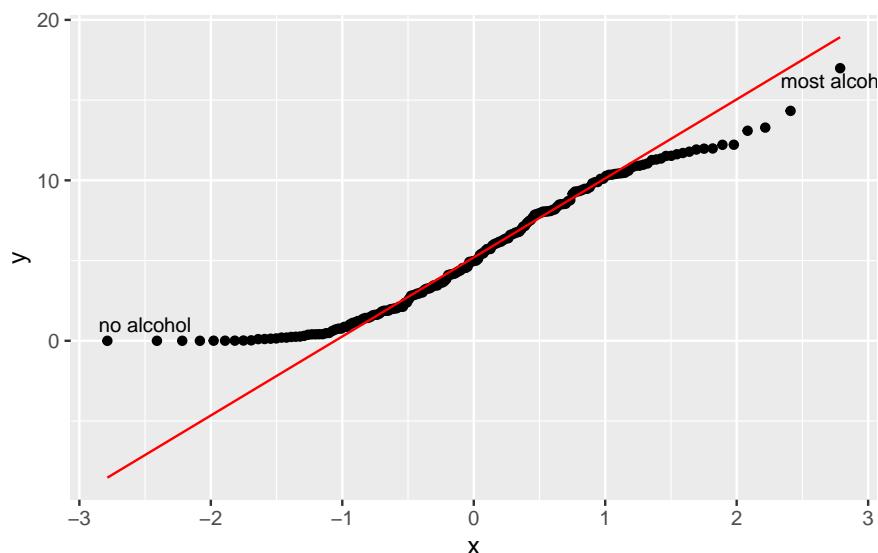
With these definitions out of the way, let’s plot some quantiles against each other.

```
alcohol_2019 |>
  ggplot(aes(sample = consumption)) +
  geom_qq() +
  geom_qq_line(col = "red") +
  annotate("text",
    label = "most alcohol",
```

```

x = 2.77,
y = 16.2,
size = 3) +
annotate("text",
label = "no alcohol",
x = -2.5,
y = 1,
size = 3)

```



The function `geom_qq()` creates the dots, while `geom_qq_line` establishes a reference for a normal distribution. The reference line is drawn in such a way that it touches the quartiles of our distribution. Ideally, we would want that all dots are firmly aligned with each other. Unfortunately, this is not the case in our dataset. At the top and the bottom, we have points that deviate quite far from a normal distribution. Remember the countries that consume no alcohol at all, i.e. $y = 0$ in the QQ plot? They are fairly far away from the rest of the other countries in our dataset.

8.4.6 Standard deviation: Your average deviation from the mean

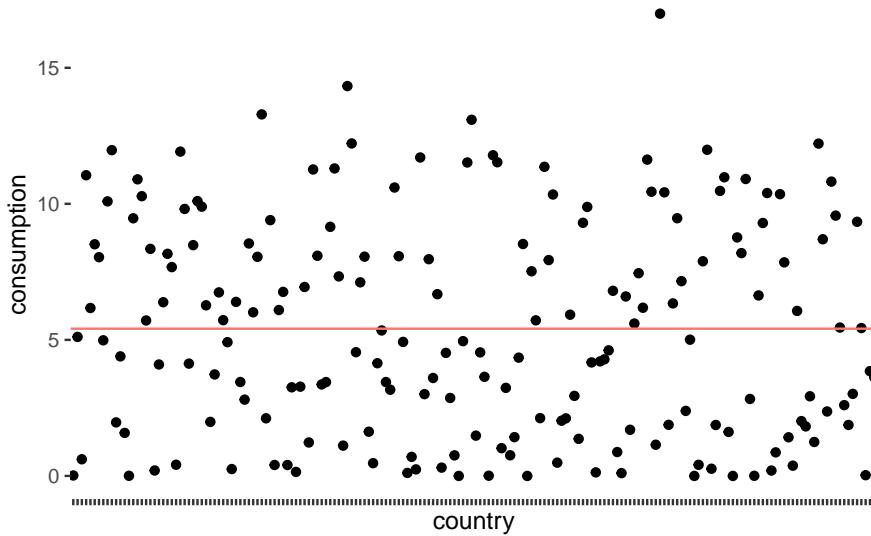
I left the most commonly reported statistics for the spread of data last. The main reason for this is that one might quickly jump ahead to look at the standard deviation without ever considering plotting the distribution of variables in the first place. Similar to the mean and other numeric indicators, they could

potentially convey the wrong impression. Nevertheless, the standard deviation is an important measure.

To understand what the standard deviation is, we can consider the following visualisation:

```
alcohol_mean <- mean(alcohol_2019$consumption)

alcohol_2019 |>
  select(country, consumption) |>
  ggplot(aes(x = country, y = consumption)) +
  geom_point() +
  geom_hline(aes(yintercept = alcohol_mean, col = "red"), show.legend = FALSE) +
  
  # Making the plot a bit more pretty
  theme(axis.text.x = element_blank(),           # Removes country names
        panel.grid.major = element_blank(),       # Removes grid lines
        panel.background = element_blank()        # Turns background white
      ) +
  ylab("consumption") +
  xlab("country")
```



The red line (created with `geom_hline()`) represents the mean consumption of alcohol across all countries in the dataset, which is 5.41 litres per year. We notice that the points are falling above and below the mean, but not directly on it. In other words, there are not many countries where people drink 5.41 litres of alcohol. We make this visualisation even more meaningful if we sorted

the countries by their alcohol consumption. We can also change the shape of the dots (see also Chapter 10) by using the attribute shape. This helps to plot many dots without having them overlap.

```
alcohol_2019 |>
  select(country, consumption) |>
  ggplot(aes(x = reorder(country, consumption), y = consumption)) +
  geom_point(shape = 124) +
  geom_hline(aes(yintercept = alcohol_mean, col = "red"), show.legend = FALSE) +
  theme(axis.text.x = element_blank(),
        panel.grid.major = element_blank(),
        panel.background = element_blank()
      ) +
  ylab("consumption") +
  xlab("country")
```

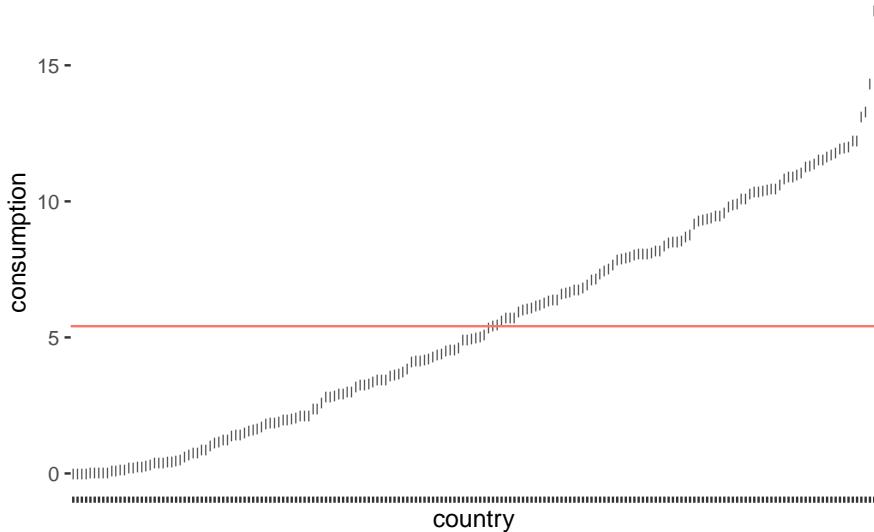


Figure 8.11: Deviation of alcohol consumption from the meand consumption levels

As we can see, only a tiny fraction of countries are close to the mean. If we now consider the distance of each observation from the red line, we know how much each of them, i.e. each country, deviates from the mean. The standard deviation tells us how much the alcohol consumption deviates on average. To be more specific, to compute the standard deviation by hand, you would:

- compute the difference between each observed value of `consumption` and the mean of `consumption` in your dataset, which is also called ‘*deviance*’, i.e. $\text{deviance} = \text{consumption} - \text{mean}_{\text{all countries}}$
- square the deviance to turn all scores into positive ones, i.e. $\text{deviance}_{\text{squared}} = (\text{deviance})^2$,
- then we take the sum of all deviations (also known as ‘*sum of squared errors*’) and divide it by the number of countries in our dataset ($n = 188$) minus 1, which results in the ‘*variance*’, i.e. $\text{variance} = \frac{\sum(\text{deviations}_{\text{squared}})}{188-1}$. The variance reflects the average dispersion of our data.
- Lastly, we take the square root of this score to obtain the standard deviation, i.e. $\text{sd} = \sqrt{\text{variance}}$.

While the deviation and variance are interesting to look at, the standard deviation has the advantage that it provides us with the average deviation (also called ‘*error*’) based on the units of measurement of our variable. Thus, it is much easier to interpret its size.

We could compute this by hand if we wanted, but it is much simpler to use the function `sd()` to achieve the same.

```
sd(alcohol_2019$consumption)
```

```
[1] 3.963579
```

The result shows that countries tend to report about 3.96 litres more or less than the average country. This seems quite a lot. However, we must be aware that this score is also influenced by the outliers we detected before. As such, if standard deviations in your data appear quite large, it can be due to outliers, and you should investigate further. Plotting your data will undoubtedly help to diagnose any outliers.

8.5 Packages to compute descriptive statistics

There is no one right way of how you can approach the computation of descriptive statistics. There are only differences concerning convenience and whether you have intentions to use the output from descriptive statistics to plot them or use them in other ways. Often, we only want to take a quick look at our data to understand it better. If this is the case, I would like to introduce you to two packages that are essential in computing descriptive statistics and constitute an excellent starting point to understanding your data:

- `psych`

- `skimr` (see also Section 7.2)

8.5.1 The `psych` package for descriptive statistics

As its name indicates, the `psych` package is strongly influenced by how research is conducted in the field of Psychology. The package is useful in many respects, and we already used it in Section 7.8 to compute Cronbach's α .

Another useful function is `describe()`. We can use this function to compute summary statistics for a variable. For example, we might wish to inspect the descriptive statistics for `consumption`.

```
psych::describe(alcohol_2019$consumption)
```

```
vars n mean sd median trimmed mad min max range skew kurtosis se
X1 1 188 5.41 3.96 4.97 5.23 4.9 0 16.99 16.99 0.33 -0.94 0.29
```

If we want to see descriptive statistics per group, we use `describeBy()`, which takes a factor as a second argument. So, for example, we could compute the descriptive statistics for `consumption` separately for each year by using the larger dataset called `alcohol`.

```
descriptives <- psych::describeBy(alcohol$consumption,
                                   alcohol$year)

# # The output for the last three years
descriptives[3:5]

$`2010`
vars n mean sd median trimmed mad min max range skew kurtosis se
X1 1 188 5.56 4.17 5.15 5.3 5.16 0 17.62 17.62 0.44 -0.78 0.3

$`2015`
vars n mean sd median trimmed mad min max range skew kurtosis se
X1 1 188 5.49 4.05 5.1 5.28 5.07 0 16.85 16.85 0.35 -0.91 0.3

$`2019`
vars n mean sd median trimmed mad min max range skew kurtosis se
X1 1 188 5.41 3.96 4.97 5.23 4.9 0 16.99 16.99 0.33 -0.94 0.29
```

The advantage of using `psych` is the convenience of retrieving several descriptive statistics with just one function instead of chaining together several functions to achieve the same.

8.5.2 The `skimr` package for descriptive statistics

While the `psych` package returns descriptive statistics for a single numeric variable, `skimr` takes this idea further and allows us to generate descriptive statistics for all variables of all types in a data frame with just one function, i.e. `skim()`. We already covered this package in Section 7.2, but here is a brief reminder of its use for descriptive statistics.

```
skimr::skim(alcohol_2019)
```

```
> skimr::skim(alcohol_2019)
-- Data Summary --
Name                           alcohol_2019
Number of rows                 188
Number of columns                4
Column type frequency:
  factor                         2
  numeric                        2
Group variables                 None
-- Variable type: factor --
skim_variable n_missing complete_rate ordered n_unique top_counts
1 country                      0          1 FALSE        188 Afg: 1, Alb: 1, Alg: 1, And: 1
2 code                          0          1 FALSE        188 AFG: 1, AGO: 1, ALB: 1, AND: 1
-- Variable type: numeric --
skim_variable n_missing complete_rate      mean     sd    p0    p25    p50    p75    p100 hist
1 year                          0          1 2019     0  2019  2019  2019  2019  2019  2019  ┌─┐
2 consumption                   0          1  5.41  3.96     0   1.87   4.97   8.52  17.0  └─┘
```

In addition to providing descriptive statistics, `skim()` sorts the variables by data type and provides different descriptive statistics meaningful to each kind. Apart from that, the column `n_missing` can help spot missing data. Lastly, we also find a histogram in the last column for `numeric` data, which I find particularly useful.

When just starting to work on a new dataset, the `psych` and the `skimr` package are solid starting points to get an overview of the main characteristics of your data, especially in larger datasets. On your *R* journey, you will encounter many other packages which provide useful functions like these.

9

Sources of bias: Sampling, outliers, normality and other ‘conundrums’

‘*Bias*’ in your analysis is hardly ever a good thing unless you are a qualitative researcher. Whether you consider it positive or negative, we have to be aware of issues preventing us from performing a specific type of analysis. All of the statistical computations discussed in the following chapters can easily be affected by different sources of bias and, therefore, need to be discussed in advance. The lack of certain biases can be an assumption of particular statistical tests. Thus, violating these assumptions would imply that the analytical technique we use will produce wrong results, i.e. biased results. Field (2013) summarises three main assumptions we have to consider:

- Linearity and additivity,
- Independence,
- Normality, and
- Homogeneity of variance, i.e. homoscedasticity.

Most *parametric tests* require that all assumptions are met. If this is not the case, we have to use alternative approaches, i.e. *non-parametric tests*. The distinction is essential since parametric and non-parametric tests are based on different computational methods, leading to different outcomes.

Lastly, Field (2013) also mentions outliers as an important source of bias. Irrespective of whether your data fulfils the assumptions for parametric tests, outliers tend to be a significant problem. They usually lead to wrong results and, consequently, to misinterpretations of our findings. We will also cover how we can identify and handle such outliers in our data at the end of this chapter.

9.1 The origins of biases: A short introduction to sampling theory

While we might have a solid understanding that biases are an issue for quantitative researchers, we might be wondering where they emerge from or whether they show up at random to make our lives miserable. While there are many different reasons why we have a biased dataset, there are two that are particularly noteworthy. One the one hand, what might seem like a bias to us could simply be the nature of the population. Not every population follows a normal distribution (see Section 8.4.2 and Section 9.4) and not every relationship between variables is linear (see Section 9.2). So, no matter how carefully we choose our participants, we might still violate some of the assumptions for certain techniques. On the other hand, and much more of a concern, are sampling strategies we pick to collect our data. This section will elaborate more on the issues of choosing a viable sampling strategy and why it matters. In many ways, it is the building block of any analysis. Only good and reliable data can produce good and reliable results.

Let’s take a step back and consider the purpose of our analysis. We primarily conduct analysis to identify patterns in our world that can hopefully be replicated across different contexts. We often conduct research based on a smaller group of people (i.e. a sample) to generalise to a much larger group of people (i.e. the population). Under most circumstances It would be impossible for most studies to collect data from the entire population. Thus, we try to infer from our small sample how things are in the entire population. For this to work reliably, we need to ensure that our sample is sufficiently large to reflect the entire population. By ‘reflect’ I mean that the data is large enough to produce results that closely approximate what we would observe if data were collected from the entire population. Researchers often refer to this as the ‘representativeness’ of the sample. The lack of ‘representativeness’ is often referred to as ‘selection bias’ (see also (Lohr 2021)).

In order to achieve ‘representativeness,’ we need to ensure that our sample matches the characteristics of the population closely enough. A major factor that affects this is *sampling strategy*. The sampling strategy determines the method we use to select individuals from a population. Unfortunately, not all strategies are considered good choices and introduce various degrees of selection bias. Bear in mind, in qualitative research, selection bias is not a real issue and strategies that help identify participants that could valuable contribute to the study would always be given preference over randomly selecting people for the sake of representativeness. Table 9.1 briefly summarises some of the most common sampling strategies deployed in Social Sciences and their likelihood of introducing selection bias.

Table 9.1: Overview of some of the most common sampling strategies used in Social Sciences (adopted from (Lohr 2021))

Sampling strategy	Description	Likelihood of selection bias
<i>Simple Random Sampling</i>	Every participant in the population has the same chance of being selected for the study. However, it is usually difficult to achieve this approach in practice since it requires a complete list of the population, also known as the sampling frame, which is difficult to obtain. Also, in very small samples it can happen that small subgroups in a population might get missed as their chance of being included is also small.	Low
<i>Stratified Random Sampling</i>	The population is divided into subgroups, so-called ‘strata’ from which participants are selected using random sampling. This technique ensures that all subgroups of a population can be included, regardless of their size. In those scenarios, this technique can be superior to simple random sampling.	Low

Sampling strategy	Description	Likelihood of selection bias
<i>Cluster Sampling</i>	The population is divided into clusters (e.g. countries, schools, departments, etc.), and a random sample of clusters is selected. After that, either all individuals in the chosen clusters are asked to participate (i.e. one-stage cluster sampling), or a random sample is drawn from within each of the sampled clusters (i.e. two-stage cluster sampling). This technique is often used in large and geographically dispersed populations to make the research more feasible.	Low to Moderate
<i>Quota Sampling</i>	The population is divided into subgroups (quota classes) and convenience sampling is used to fill quotas for each subgroup based on specific characteristics. A main difference of this technique to stratified sampling is that non-probability techniques are used to choose members for the subgroups.	Moderate

Sampling strategy	Description	Likelihood of selection bias
<i>Purposive/ Judgement Sampling</i>	Participants are selected based on specific criteria that the researcher believes are important for identifying individuals who are most relevant to the study.	Moderate to High
<i>Snowball Sampling</i>	Future participants are chosen based on recommendations from current participants. For example, after conducting an interview, the interviewer might ask whether they can recommend other possible participants.	High
<i>Convenience Sampling</i>	Participants are chosen because they are easy for the researcher to access and are willing to take part in the study.	High
<i>Self-selected Sampling</i>	Individuals choose by themselves whether they want to take part in a study or not. Many online questionnaires or review platforms use self-selected samples. Be wary of those Amazon reviews.	High

While all this might sound terribly theoretical, let me provide you with an example. Imagine we are interested in understanding how much time students spend on their weekly readings in our department. Also, assume that our department consists of 1000 students. We now wish to draw a sample of students for our analysis.

The dataset `reading_time` from the `r4np` package contains all the information we are after. Since the dataset contains information about the entire popu-

182 9 Sources of bias: Sampling, outliers, normality and other ‘conundrums’

lation we can identify important characteristics of our population which we later can compare against our sample. We can use the trusty `count()` function to obtain information about distributions in our population, for example for gender.

```
# Gender distribution
reading_time |>
  count(gender) |>
  mutate(perc = n/sum(n))

# A tibble: 3 x 3
  gender      n   perc
  <chr>     <int> <dbl>
1 female      450  0.45
2 male        450  0.45
3 non-binary  100  0.1
```

We can see that in our department 45% of students identify as `female`, 45% as `male` and 10% as `non-binary`. Consequently, when we draw a sample from this population we would ideally wish to end up with a similar distribution. The percentages mentioned here are also indicative of how likely someone will be selected based on `gender`, i.e. there is only a 10% chance that we pick a student who identifies as `non-binary`, while there is an even chance of 45% to pick a `female` or `male` student.

Now that we know how our sample should look like with regards to `gender` we can start selecting people. Let’s begin by drawing nine samples of 20 students.

```
# ANALYSIS -----
plots <- sampling_sim(seeds = c(123456, 123457, 123458,
                                123451, 123455, 123465,
                                123471, 123585, 123495),
                      sample_size = 20,
                      df = reading_time)

combined_plot <-
  reduce(plots$plots, `+`) +
  plot_annotation(
    title = "SIMULATION OF SIMPLE RANDOM SAMPLING",
    subtitle = "(Sample size = 20) \n",
    theme = theme(
      plot.title = element_text(size = 16, face = "bold", hjust = 0.5),
      plot.subtitle = element_text(size = 12, face = "italic", hjust = 0.5),
    ))
```

```

) +
plot_layout(guides = "collect") # Collects legends from plots

combined_plot

```

SIMULATION OF SIMPLE RANDOM SAMPLING

(*Sample size = 20*)

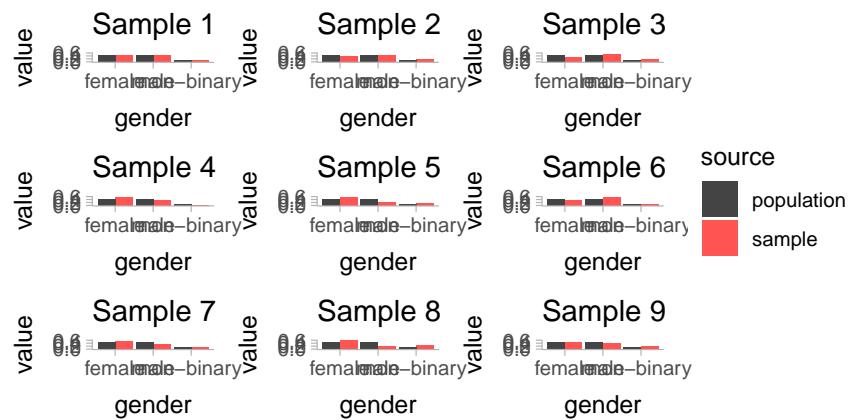


Figure 9.1: Simulation of simple random sampling ($n = 20$)

Looking at the visualisation, we can see the distribution of gender in our sample in red compared to the one in our population coloured in grey. You might be surprised to see that not all samples seem equally good in being representative even though we used simple random sampling. **Sample 1**, **Sample 2**, and **Sample 9** seem to match the gender distribution of our population fairly well. However, **Sample 4**, **Sample 5** and **Sample 8** are rather poor representations of our population. Thus, only 3 out of 9 samples would provide us with a dataset that is somewhat representative.

An attentive reader might wonder: Is a sample of 20 students not too small? And the answer would be a resounding ‘yes’. Even with simple random sampling, we would have to achieve a large enough sample to ensure no matter how often we draw a sample, we end up with a dataset that is representative enough of our population. Let’s draw nine more samples, but this time we opt to select 300 students for each sample.

```
# ANALYSIS -----
plots <- sampling_sim(seeds = c(123456, 123457, 123458,
                                123451, 123455, 123465,
                                123471, 123585, 123495),
                      sample_size = 300,
                      df = reading_time)

combined_plot <-
  reduce(plots$plots, `+`) +
  plot_annotation(
    title = "SIMULATION OF SIMPLE RANDOM SAMPLING",
    subtitle = "(Sample size = 300) \n",
    theme = theme(
      plot.title = element_text(size = 16, face = "bold", hjust = 0.5),
      plot.subtitle = element_text(size = 12, face = "italic", hjust = 0.5),
      )
    ) +
  plot_layout(guides = "collect") # Collects legends from plots

combined_plot
```

SIMULATION OF SIMPLE RANDOM SAMPLING

(*Sample size = 300*)

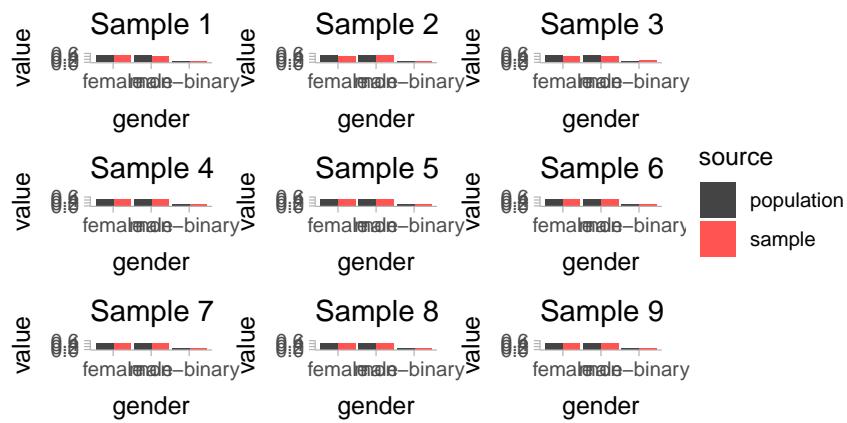


Figure 9.2: Simulation of simple random sampling ($n = 300$)

This time, all of our 9 samples seem to represent the population very well with

regards to the gender distribution. In short: your sampling strategy and sample size matter significantly to achieve a dataset which can provide insights into the chosen population. This is true irrespective of whether correct analytical techniques were chosen to analyse the data afterwards, which of course would still produce wrong results.

Lastly, let me show you how results could differ when samples are not comparable by considering the mean and median reading time (`min_reading`) by students in our department. I made a couple more modifications to enable us to read the final results better. So don't feel irritated by the code I am showing you below.

```
# Reading time based on entire population (n = 1000)
population <- reading_time |>
  summarise(mean = mean(min_reading),
            median = median(min_reading)) |>
  mutate(sample_type = "population",
        .before = everything())

# Reading time based on representative sample (n = 300)
set.seed(123456)

rep_sample_300 <-
  reading_time |>
  slice_sample(n = 300) |>
  summarise(mean = mean(min_reading),
            median = median(min_reading)) |>
  mutate(sample_type = "representative sample (n = 300)", .before = everything())

# Reading time based on representative sample (n = 20)
set.seed(123456)

rep_sample_20 <-
  reading_time |>
  slice_sample(n = 20) |>
  summarise(mean = mean(min_reading),
            median = median(min_reading)) |>
  mutate(sample_type = "representative sample (n = 20)",
        .before = everything())

# Reading time based on non-representative sample (n = 20)
set.seed(123585)

non_rep_sample <-
```

```

reading_time |>
slice_sample(n = 20) |>
summarise(mean = mean(min_reading),
median = median(min_reading)) |>
mutate(sample_type = "non-representative sample (n = 20)", .before = everything())

comparison <- rbind(population, rep_sample_300, rep_sample_20, non_rep_sample)

comparison

# A tibble: 4 × 3
  sample_type      mean   median
  <chr>        <dbl>    <dbl>
1 population     362.    359.
2 representative sample (n = 300) 364.    364.
3 representative sample (n = 20)   354.    376
4 non-representative sample (n = 20) 335.    328.

```

In light of everything we covered in this chapter, it is not surprising to find that the representative samples are much closer to the true `mean` and `median` of the population than the non-representative one. In addition, the bigger representative sample also performed much better in capturing the central tendency of `min_reading` in our population than the smaller one.

In conclusion, even with the right analytical tools, our results depend heavily on how we selected a sample from the population to ensure it represents the group we are interested in. This highlights the importance of clearly defining your population as a crucial first step in the process. Consequently, choosing a sampling strategy that leads to less representative datasets implies that your findings are likely not applicable to people outside your sample. While this is not necessarily a problem, it is a limitation that you should be transparent about.

9.2 Linearity and additivity

The assumption of linearity postulates that the relationship of variables represents a straight line and not a curve or any other shape. Figure 9.3 depicts examples of how two variables could be related to each other. Only the first one demonstrates a linear relationship, and all other plots would represent a violation of linearity.

Data visualisations are particularly useful to identify whether variables are

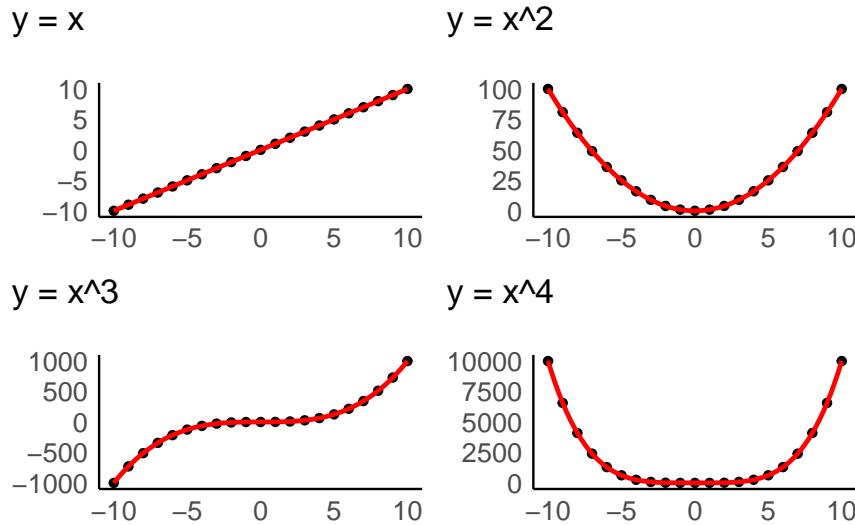


Figure 9.3: Examples of linear and non-linear relationships of two variables

related to each other in a linear fashion. The examples above were all created with `geom_point()`, which creates a dot plot that maps the relationship between two variables. In Chapter 10, we will look more closely at the relationship of two variables in the form of *correlations*, which measure the strength of a linear relationship between variables.

Additivity is given when the effects of all independent variables can be added up to obtain the total effect they have on a dependent variable. In other words, the effect that multiple variables have on another variable can be added up to reflect their total effect.

If we assume that we have a dependent variable Y which is affected by other (independent) variables X , we could summarise additivity and linearity as a formula:

$$Y = \beta_1 * X_1 + \beta_2 * X_2 + \dots + \beta_n * X_n$$

The β (beta) stands for the degree of change in a variable X causes in Y . Or, in simpler terms, β reflects the impact an independent variable has on the dependent variable. We will return to this equation and terminology in Chapter 13, where we create a linear model via regression.

9.3 Independence

The notion of independence is an important one. It assumes that each observation in our dataset is independent of other observations. For example, imagine my wife Fiona and I take part in a study that asks us to rank movies by how much we like them. Each of us has to complete the ranking by ourselves, but since we both sit in the same living room, we start chatting about these movies. By doing so, we influence each other’s rankings and might even agree on the same ranking. Thus, our scores are not independent from each other. On the other hand, suppose we were both sitting in our respective offices and rank these movies. In that case, the rankings could potentially still be very similar, but this time the observations are independent of each other.

There is no statistical measurement or plot that can tell us whether observations are independent or not. Ensuring independence is a matter of data collection and not data analysis. Thus, it depends on how you, for example, designed your experiment, or when and where you ask participants to complete a survey, etc. Still, this criterion should not be downplayed as being a ‘soft’ one, just because there is no statistical test, but should remain on your radar throughout the planning and data collection stage of your research.

9.4 Normality

We touched upon the notion of ‘normality’ and ‘normal distributions’ before in Section 8.4 because it refers to the spread of our data. Figure 9.4 should look familiar by now.

However, we have yet to understand why it is essential that our data follows a normal distribution. Most parametric tests are based on means. For example, if we want to compare two groups with each other, we would compute the mean for each of them and then see whether their means differ from each other in a significant way (see Chapter 11). Of course, if our data is not very normally distributed, means are a poor reference point for most of the observations in this group. We already know that outliers heavily affect means, but even without outliers, the mean could be a poor choice. Let me provide some visual examples.

```
Warning in geom_text(aes(x = mean(income) - 15, y = 2e-04, label = "mean", : All aesthetics have length 1, but the data has 11 rows.
i Please consider using `annotate()` or provide this layer with data containing
a single row.
```

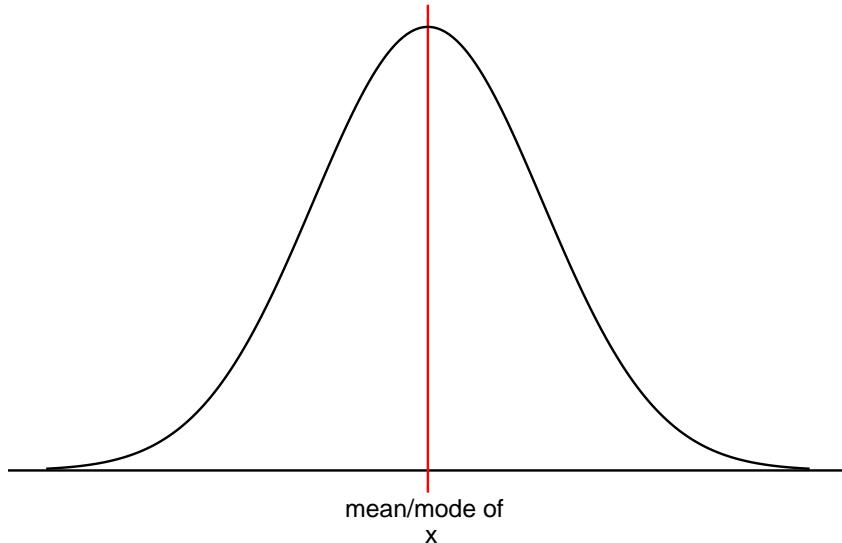


Figure 9.4: A normal distribution

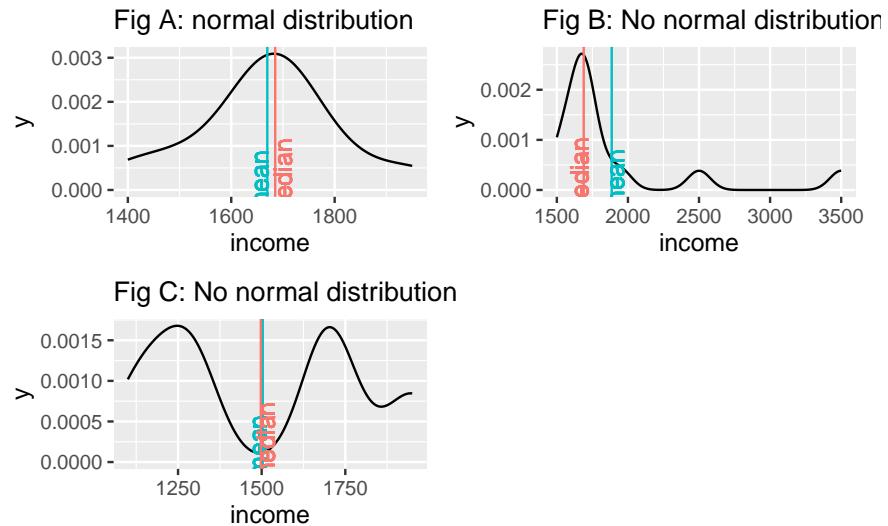
```
Warning in geom_text(aes(x = median(income) + 15, y = 0.00024, label = "median", : All aesthetics have length 1
i Please consider using `annotate()` or provide this layer with data containing
a single row.

Warning in geom_text(aes(x = mean(income) + 25, y = 2e-04, label = "mean", : All aesthetics have length 1, but the data has 13 rows.
i Please consider using `annotate()` or provide this layer with data containing
a single row.

Warning in geom_text(aes(x = median(income) - 30, y = 0.00024, label = "median", : All aesthetics have length 1, but the data has 13 rows.
i Please consider using `annotate()` or provide this layer with data containing
a single row.

Warning in geom_text(aes(x = mean(income) - 15, y = 2e-04, label = "mean", : All aesthetics have length 1, but the data has 12 rows.
i Please consider using `annotate()` or provide this layer with data containing
a single row.

Warning in geom_text(aes(x = median(income) + 15, y = 0.00024, label = "median", : All aesthetics have length 1
i Please consider using `annotate()` or provide this layer with data containing
a single row.
```



We notice that neither the median nor the mean by themselves is a reliable indicator for normality. Figure A and Figure C both show that the median and mean are almost identical, but only Figure A follows a normal distribution. The median and mean in Figure C are not reflective of the average observation in this dataset. Most scores lie below and above the mean/median. Therefore, when we analyse the normality of our data, we usually are not interested in the normality of a single variable but the normality of the sampling distribution. However, we cannot directly assess the sampling distribution in most cases. As such, we often revert to testing the normality of our data. There are also instances where we would not expect a normal distribution to exist. Consider the following plots:

The first plot clearly shows that data is not normally distributed. If anything, it looks more like the back of a camel. The technical term for this distribution is called '*bimodal distribution*'. In cases where our data has even more peaks we consider it as a '*multimodal distribution*'. If we identify distributions that look remotely like this, we can assume that there must be another variable that helps explain why there are two peaks in our distribution. The plot below reveals that gender appears to play an important role. Drawing the distribution for each subset of our data reveals that `income` is now normally distributed for each group and has two different means. Thus, solely focusing on normal distributions for a single variable would not be meaningful if you do not consider the impact of other variables. If we think that `gender` plays an important role to understand `income` levels, we would have to expect that the distribution is not normal when looking at our data in its entirety.

Determining whether data is normally distributed can be challenging when

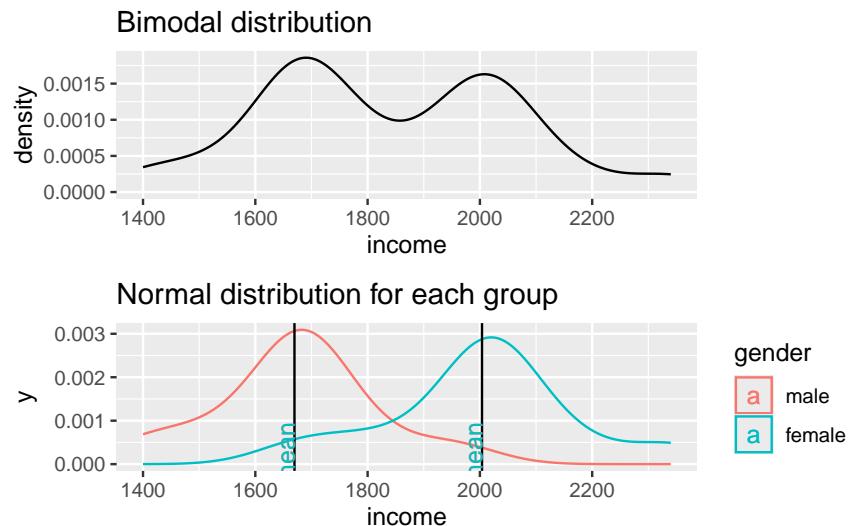


Figure 9.5: Two normal distributions in one dataset.

only inspecting plots, e.g. histograms, density plots or QQ plots. Luckily, there is also a statistical method to test whether our data is normally distributed: *The Shapiro-Wilk test*. This test compares our distribution with a normal distribution (like in our plot) and tells us whether our distribution is **significantly different** from it. Thus, if the test is not significant, the distribution of our data is not significantly different from a normal distribution, or in simple terms: It is normally distributed. We can run the test in R as follows for the dataset that underpins Fig A above:

```
shapiro.test(data$income)
```

```
Shapiro-Wilk normality test
data: data$income
W = 0.94586, p-value = 0.5916
```

This result confirms that the data is normally distributed, because it is not significantly different ($p > 0.05$). The chapter on correlations looks at significance and its meaning more thoroughly (see Section 10.3).

In terms of computing normality, it can become very troublesome, very quickly. When we compare different groups of people (like in Section 11.3.1) we have to check normality for each group and each variable of interest separately,

192 9 Sources of bias: Sampling, outliers, normality and other ‘conundrums’

which means we first need to create subsets of our data for each group and then perform `shapiro.test()` separately for each group. This might be still manageable with two or three groups, but can become very time-consuming if there are more groups in a larger dataset. Luckily, I included some helpful function in `r4np` that can help make this a lot easier:

- `ntest()`: This function allows to compute multiple `shapiro.test()`s for multiple variables.
- `ntest_by()`: This function allows to compute multiple `shapiro.test()`s for multiple variables based on a group variable.

Let me show you how these two functions extremely handy when testing normality in bulk.

```
# Perform 'shapiro.test()' for multiple variables at once
ic_training |> ntest(cols = c(communication,
                                teamwork,
                                leadership))
```

```
# A tibble: 3 x 4
  row_name      statistic p.value method
  <chr>          <dbl>     <dbl> <chr>
1 communication  0.970  0.0255 Shapiro-Wilk normality test
2 teamwork       0.963  0.00890 Shapiro-Wilk normality test
3 leadership     0.962  0.00702 Shapiro-Wilk normality test
```

Here is an example where we compute the normality for all three variables again, but this time we consider two different subsets of our dataset.

```
# Perform 'shapiro.test()' for multiple variables and groups
ic_training |> ntest_by(cols = c(communication,
                                    teamwork,
                                    leadership),
                           group = test
                         )
```



```
# A tibble: 6 x 5
  test           variable      statistic p.value method
  <fct>         <chr>          <dbl>     <dbl> <chr>
1 pre_training   communication  0.978    0.517  Shapiro-
Wilk normality test
2 pre_training   teamwork      0.950    0.0398 Shapiro-Wilk normality test
3 pre_training   leadership     0.957    0.0747 Shapiro-Wilk normality test
4 post_training  communication  0.924    0.00415 Shapiro-
Wilk normality test
```

```
5 post_training teamwork      0.944 0.0240 Shapiro-Wilk normality test
6 post_training leadership    0.937 0.0124 Shapiro-Wilk normality test
```

I use the `ntest_by()` function regularly, because it saves so much time and does not require me to create subsets of my data. It is probably the most useful analytical function I have ever written.

In conclusion, the normality of data is an essential pre-test for any of our studies. If we violate the assumption of normality, we will have to fall back to non-parametric tests. However, this rule has two exceptions: *The Central Limit Theorem* and using ‘robust’ measures of parametric tests. The Central Limit Theorem postulates that as our sample becomes larger, our sampling distribution becomes more and more normal around the mean of the underlying population. For example, Field (2013) (p.54) refers to a sample of 30 as a common rule of thumb. As such, it is possible to assume normality for larger datasets even though our visualisation and the Shapiro-Wilk test tell us otherwise. The second exception is that many parametric tests offer a ‘robust’ alternative, often via *bootstrapping*. *Bootstrapping* refers to the process of subsampling your data, for example, 2000 times, and look at the average outcome.

Admittedly, this raises the question: Can I ignore normality and move on with my analysis if my sample is large enough or I use bootstrapping? In short: Yes. This fact probably also explains why we hardly ever find results from normality tests in journal publications since most Social Science research involves more than 30 participants. However, if you find yourself in a situation where the sample size is smaller, all of the above needs to be checked and thoroughly considered. However, the sample size also has implications with regards to the power of your test/findings (see Chapter 12). Ultimately, the answer to the above questions remains, unsatisfyingly: ‘It depends’.

9.5 Homogeneity of variance (homoscedasticity)

The term ‘variance’ should sound familiar, because we mentioned it in Section 8.4.6 where we looked at the standard deviation derived from the variance.

Homogeneity of variance implies that the variance of, for example, two subsets of data, is equal or close to being equal. Let’s look at how close the observed values are to the mean for the two groups identified in Figure 9.5.

```
data4 |>
  ggplot(aes(x = gender, y = income, col = gender)) +
  geom_jitter(width = 0.1) +
```

```
geom_hline(yintercept = group_means$mean[1], color = "red") +
  geom_hline(yintercept = group_means$mean[2], color = "turquoise")
```

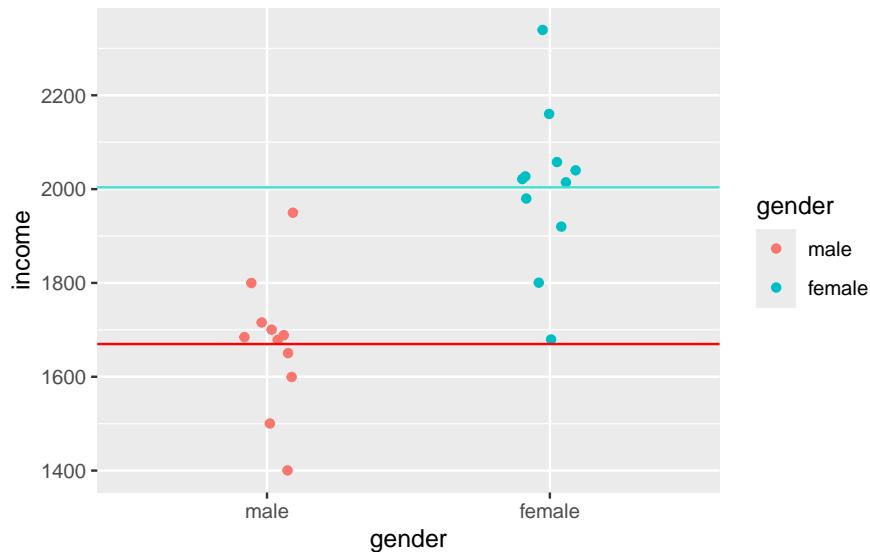
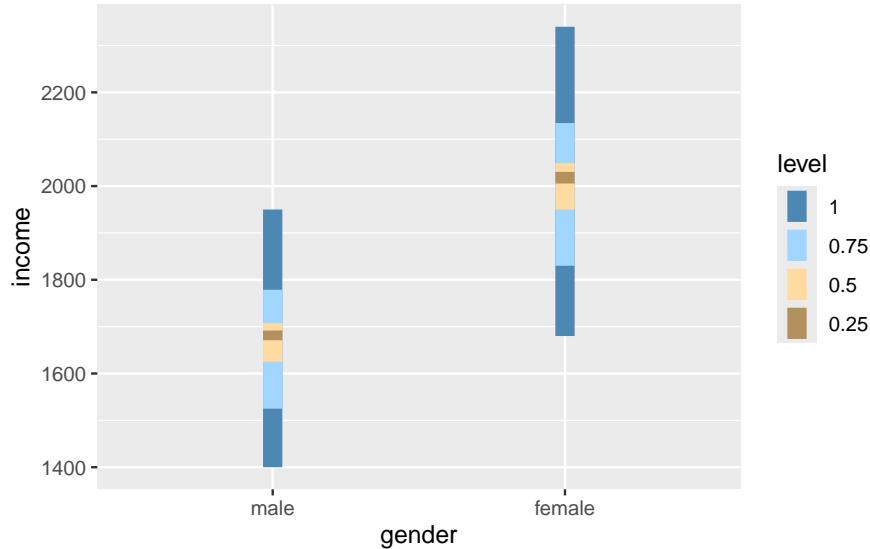


Figure 9.6

Judging by eye, we could argue that most values lie around the mean for each respective group. However, some observations are a bit further off. Still, using this visualisation, it is tough to judge whether the spread is about the same. However, boxplots can help with this, or even better, a boxplot reflected by a bar. The package `ggdist` has an excellent plotting function called `stat_interval()`, which allows us to show a boxplot in the form of a bar.

```
data4 |>
  ggplot(aes(x = gender, y = income, group = gender)) +
  ggdist::stat_interval(aes(y = income),
                        .width = c(0.25, 0.5, 0.75, 1)) +
  # Let's add some nice complementary colours
  scale_color_manual(values = c("#4D87B3", "#A1D6FF", "#FFDA81", "#B3915F"))
```

**Figure 9.7**

If we compare the bars, we can tell that the variance in both groups looks very similar, i.e. the length of the bars appear to be about the same height. Furthermore, if we compare the IQR for both groups, we find that they are quite close to each other.

```
data4 |>
  group_by(gender) |>
  summarise(iqr = IQR(income))
```

```
# A tibble: 2 x 2
  gender   iqr
  <fct>   <dbl>
1 male     82.5
2 female   99
```

However, to test whether the variance between these two groups is truly similar or different, we have to perform a *Levene's test*. The Levene's test follows a similar logic as the Shapiro-Wilk test. If the test is significant, i.e. $p < 0.05$, we have to assume that the variances between these groups are significantly different from each other. However, if the test is not significant, then the variances are similar, and we can proceed with a parametric test - assuming other assumptions are not violated. The interpretation of this test follows the one for the Shapiro-Wilk test. To compute the Levene's test we can use the function `leveneTest()` from the `car` package. However, instead of using , to

196 9 Sources of bias: Sampling, outliers, normality and other ‘conundrums’

separate the two variables, this function expects a formula, which is denoted by a `~`. We will cover formulas in the coming chapters.

```
car::leveneTest(gep$age ~ gep$gender)
```

```
Levene's Test for Homogeneity of Variance (center = median)
  Df F value Pr(>F)
group   2  0.8834 0.4144
297
```

The Levene’s test shows that our variances are similar and not different from each other because $p > 0.05$. This is good news if we wanted to continue and perform a group comparison, like in Chapter 11.

9.6 Outliers and how to deal with them

In Chapter 8, I referred to outliers many times but never eluded to the aspects of handling them. Dealing with outliers is similar to dealing with missing data. However, it is not quite as straightforward as one might think.

In a first step, we need to determine which values count as an outlier. Aguinis, Gottfredson, and Joo (2013) reviewed 232 journal articles and found that scholars had defined outliers in 14 different ways, used 39 different techniques to detect them and applied 20 different strategies to handle them. It would be impossible to work through all these options in this book. However, I want to offer two options that have been frequently considered in publications in the field of Social Sciences:

- The standard deviation (SD), and
- The inter-quartile range (IQR).

9.6.1 Detecting outliers using the standard deviation

Let us assume we are interested in understanding whether there are certain areas in England that score particularly high/low on `happiness` and whether the majority of areas report levels around the `mean`. This is a deviation analysis similar to our previous plot regarding alcohol consumption (see Figure 8.11). We can simply adapt it to our new research aim. By using the `hie_2021` we can first recreate the same plot. As before, I will create a base plot `outlier_plot` first so that we do not have to repeat the same code over and over again. We then use `outlier_plot` and add more layers as we see fit.

```
# Create the base plot
outlier_plot <-
  hie_2021 |>
  select(area_name, happiness) |>
  ggplot(aes(x = reorder(area_name, happiness),
              y = happiness)) +
  geom_point(shape = 124) +
  theme(axis.text.x = element_blank(),
        panel.grid.major = element_blank(),
        panel.background = element_blank())
) +
  xlab("Area in England") +
  ylab("Happiness")

outlier_plot
```

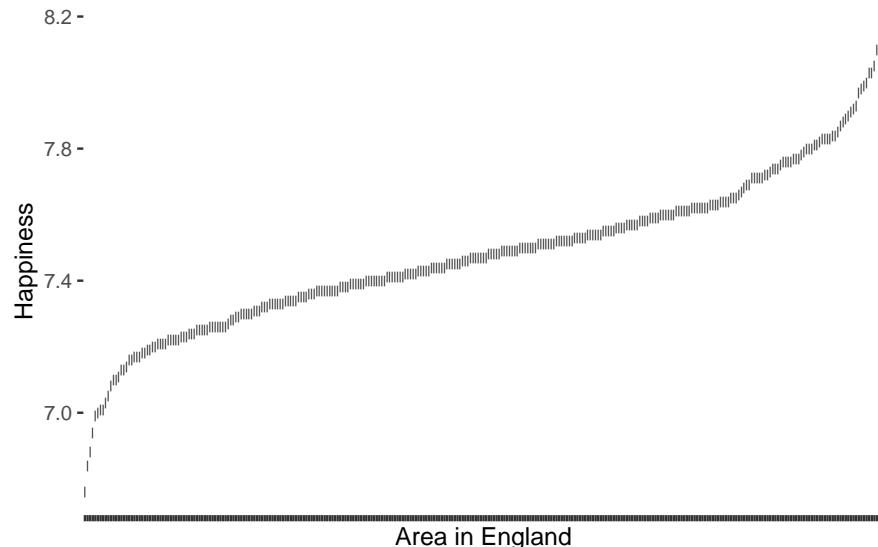
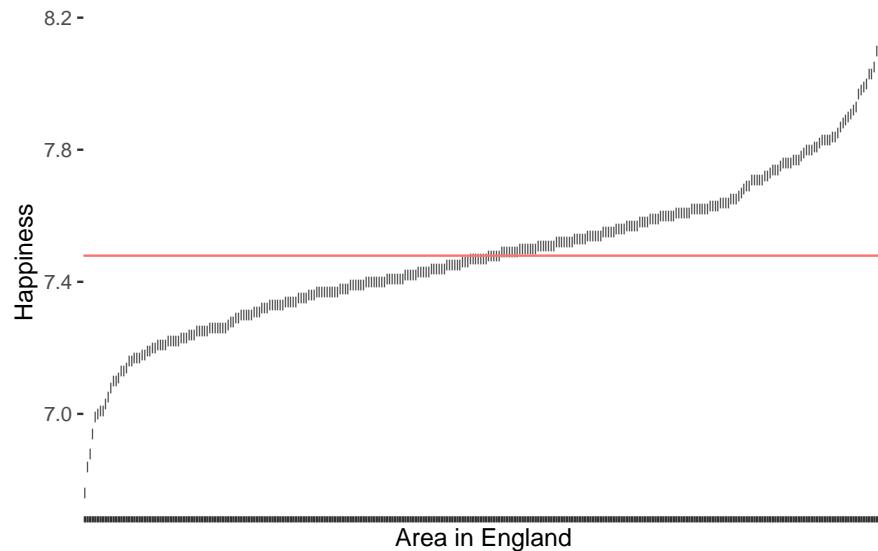


Figure 9.8

Now we can compute the mean of happiness and add it as a new layer to our existing plot.

```
# Compute the mean
happiness_mean <- mean(hie_2021$happiness)
```

```
# Add the mean reference line
outlier_plot +
  geom_hline(aes(yintercept = happiness_mean, col = "red"),
             show.legend = FALSE)
```

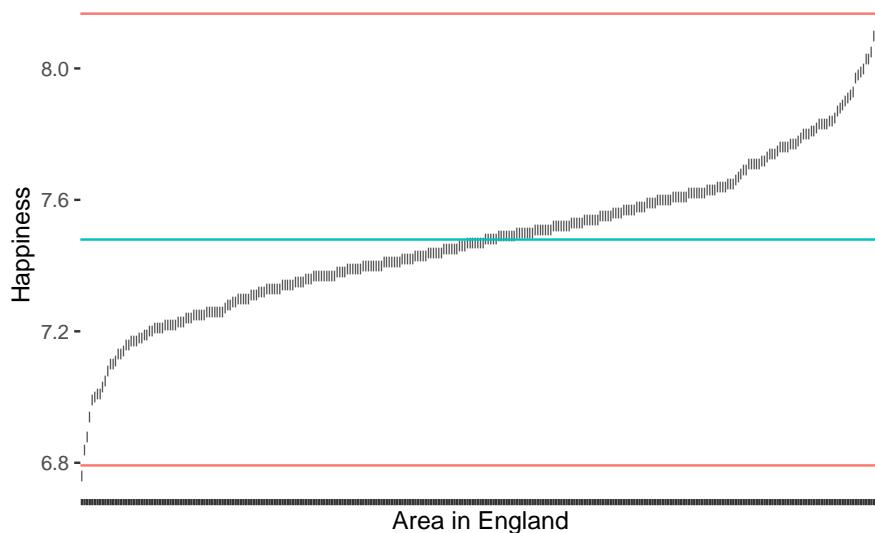


We can see that our distribution follows an S-curve and there seem to be some areas that score very high and some that score very low on happiness. Those observations at both ends of this plot which seem further away from the rest could likely be outliers. So, how can we determine which observations are truly outliers?

A very frequently used approach to detecting outliers is the use of the standard deviation. Usually, scholars use multiples of the standard deviation to determine thresholds. For example, a value that lies 3 standard deviations above or below the mean could be categorised as an outlier. Unfortunately, there is quite some variability regarding how many multiples of the standard deviation counts as an outlier. Some authors might use 3, and others might settle for 2 (see also Leys et al. (2013)). Let’s stick with the definition of 3 standard deviations to get us started.

```
# Compute the mean and the thresholds
happiness_mean <- mean(hie_2021$happiness)
sd_upper <- happiness_mean + 3 * sd(hie_2021$happiness)
sd_lower <- happiness_mean - 3 * sd(hie_2021$happiness)
```

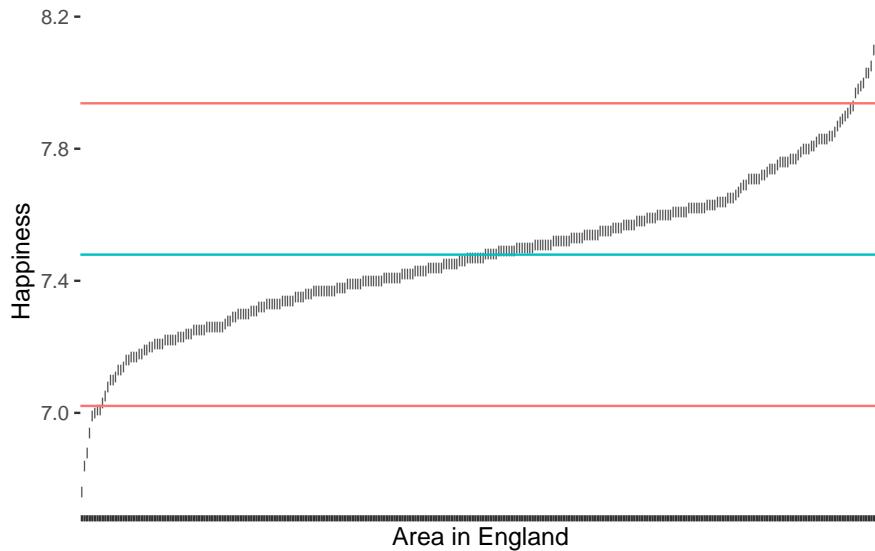
```
# Add the thresholds and mean
outlier_plot +
  geom_hline(aes(yintercept = happiness_mean, col = "red"),
             show.legend = FALSE) +
  geom_hline(aes(yintercept = sd_upper, col = "blue"),
             show.legend = FALSE) +
  geom_hline(aes(yintercept = sd_lower, col = "blue"),
             show.legend = FALSE)
```



The results suggest that only very few outliers would be detected if we chose these thresholds, i.e. one at the very bottom that sits right under the red line. However, what would we find if we applied the stricter threshold of 2?

```
# Compute the standard deviation
sd_upper <- happiness_mean + 2 * sd(hie_2021$happiness)
sd_lower <- happiness_mean - 2 * sd(hie_2021$happiness)

# Create our plot
outlier_plot +
  geom_hline(aes(yintercept = happiness_mean, col = "red"),
             show.legend = FALSE) +
  geom_hline(aes(yintercept = sd_upper, col = "blue"),
             show.legend = FALSE) +
  geom_hline(aes(yintercept = sd_lower, col = "blue"),
             show.legend = FALSE)
```



As we would expect, we identify some more areas in England as being outliers regarding their happiness. It certainly feels somewhat arbitrary to choose a threshold of our liking. Despite its popularity, there are additional problems with this approach:

- outliers affect our mean and standard deviation too,
- since we use the mean, we assume that our data is normally distributed, and
- in smaller samples, this approach might result in not identifying outliers at all (despite their presence) (Leys et al. 2013) (p. 764).

Leys et al. (2013) propose an alternative approach because medians are much less vulnerable to outliers than the mean. Similarly to the standard deviation, it is possible to calculate thresholds using the *median absolute deviation* (MAD). Best of all, the function `mad()` in *R* does this automatically for us. Leys et al. (2013) suggest using 2.5 times the MAD as a threshold. However, if we want to compare how well this option performs against the standard deviation, we could use 3 as well.

```
# Compute the median and thresholds
happiness_median <- median(hie_2021$happiness)
mad_upper <- happiness_median + 3 * mad(hie_2021$happiness)
mad_lower <- happiness_median - 3 * mad(hie_2021$happiness)

# Create our plot
outlier_plot +
  geom_hline(aes(yintercept = happiness_median, col = "red"),
             linetype = "solid", size = 1),
  geom_hline(aes(yintercept = mad_upper, col = "red"),
             linetype = "solid", size = 1),
  geom_hline(aes(yintercept = mad_lower, col = "red"),
             linetype = "solid", size = 1)
```

```

    show.legend = FALSE) +
geom_hline(aes(yintercept = mad_upper, col = "blue"),
show.legend = FALSE) +
geom_hline(aes(yintercept = mad_lower, col = "blue"),
show.legend = FALSE)

```

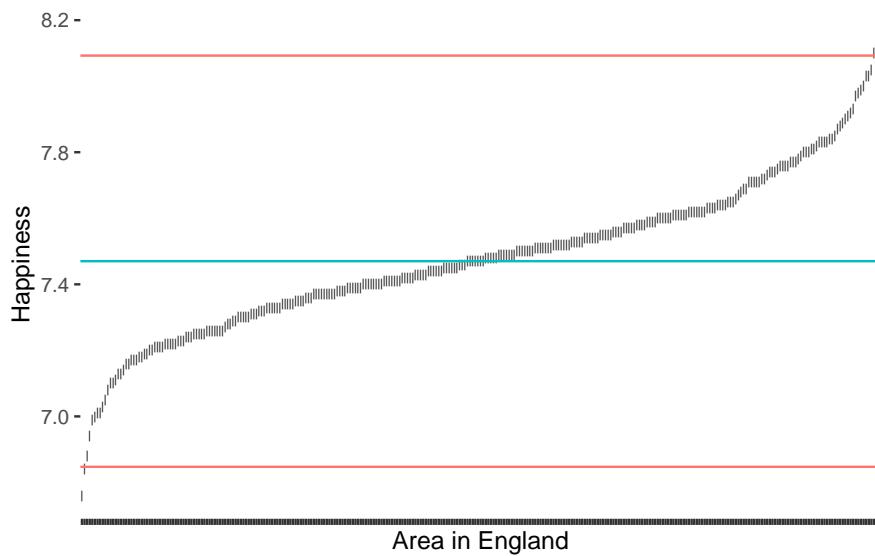


Figure 9.9: Outlier detection via MAD using $3 \times \text{MAD}$ as a threshold

Compared to our previous results, we notice that the median approach was much better in detecting outliers at the upper range of happiness. Because the median is not affected so much by the ‘extraordinarily happy areas of England’. Thus, the results have improved and seem more balanced. Still, one could argue that there are at least two more observations at the lower range of happiness that seem a little further away from the rest of the points on this S-curve. If we chose the criterion of $2.5 \times \text{MAD}$, we would also capture these outliers (see Figure 9.10).

Which approach to choose very much depends on the nature of your data. For example, one consideration could be that if the median and the mean of a variable are dissimilar, choosing the median might be the better option. However, we should not forget that the outliers we need to identify will affect the mean. Hence, it appears that in many cases, the median could be the better choice. Luckily, there is yet another highly popular method based on two quartiles (rather than one, i.e. the median).

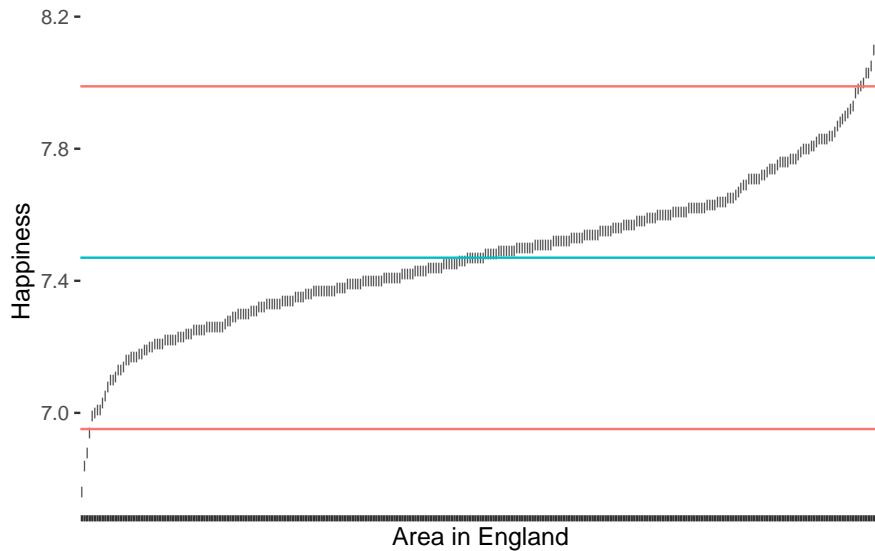


Figure 9.10: Outlier detection via MAD using $2.5 \times \text{MAD}$ as a threshold

9.6.2 Detecting outliers using the interquartile range (IQR)

Another approach to classifying outliers is the use of the *interquartile range* (*IQR*). The IQR is used in boxplots and creates the dots at its ends to indicate any outliers. This approach is straightforward to implement because the computation of the IQR is simple:

$$IQR = Q_3 - Q_1$$

Therefore, we can create new thresholds for the detection of outliers. For the IQR, it is common to use $\pm 1.5 \times IQR$ as the lower and upper thresholds measured from Q_1 and Q_3 respectively. To compute the quartiles we can use the function `quantile()`, which returns *Minimum, Q_1 , Q_2 , Q_3 , and Q_4*

```
# Compute the quartiles
(happiness_quantiles <- quantile(hie_2021$happiness))
```

```
0%    25%    50%    75%   100%
6.760 7.335 7.470 7.610 8.160
```

```
# Compute the thresholds
iqr_upper <- happiness_quantiles[4] + 1.5 * IQR(hie_2021$happiness)
iqr_lower <- happiness_quantiles[2] - 1.5 * IQR(hie_2021$happiness)
```

```
# Create our plot
outlier_plot +
  geom_hline(aes(yintercept = happiness_median, col = "red"),
             show.legend = FALSE) +
  geom_hline(aes(yintercept = iqr_upper, col = "blue"),
             show.legend = FALSE) +
  geom_hline(aes(yintercept = iqr_lower, col = "blue"),
             show.legend = FALSE)
```

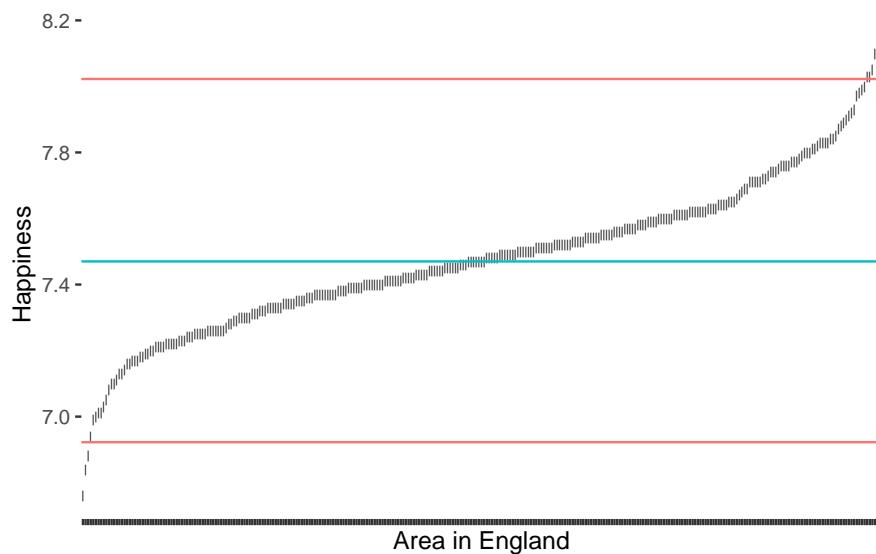


Figure 9.11

As we can tell, the IQR method detects about the same outliers for our data as the median absolute deviation (MAD) approach. The outliers we find here are the same as shown in Figure 9.12 below.

```
hie_2021 |>
  ggplot(aes(x = happiness)) +
  geom_boxplot()
```

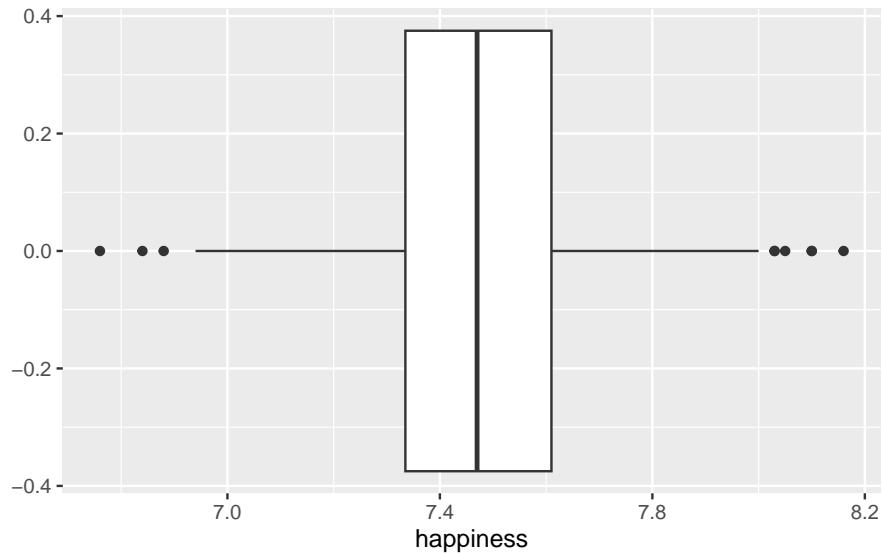


Figure 9.12: Outliers identified via IQR shown in boxplot

For our data, I would argue that the IQR and MAD method by Leys et al. (2013) produced the ‘best’ selection of outliers. However, we have to acknowledge that these classifications will always be subjective because how we position the thresholds depends on the researcher’s choice. Still, the computation is standardised, and we can plausibly explain the process of identifying outliers, we just have to make it explicit to the audience of our research.

9.6.3 Removing or replacing outliers

Now that we have identified our outliers, we are confronted with the question of what we should do with them. Like missing data (see Section 7.7), we can either remove them or replace them with other values. While removal is a relatively simple task, replacing it with other ‘reasonable’ values implies finding techniques to create such values. The same methods we used for missing data, especially multiple imputation (see Cousineau and Chartier 2010), can be used for replacing outliers. Of course, changing our dataset implies we are changing our results, so it is good practice to consider your analysis with and without outliers (e.g. P. Cohen, West, and Aiken (2014)). Consequently, running your analysis with two datasets can be a meaningful way to understand how much outliers affect your outcomes. Simply removing data without knowing how this might impact one’s results would be a poor decision. We should not forget that, especially with human participants, outliers can be genuine observations and should not simply be marginalised. If we know that our outliers are mistakes or respondents have intentionally provided wrong ratings

in our dataset, we can certainly remove them. However, this is not always evident when detecting outliers. Thus, I recommend proceeding with caution when removing outliers. Besides, there are many alternatives to analysing your data, which are less biased by outliers, such as analytical tools that rely on the median instead of the mean. In this book, we primarily focus on conducting analysis without outliers. However, performing the analysis twice (with and without outliers) would be meaningful for your own research projects.

Irrespective of whether we remove or replace outliers, we somehow need to single them out of the crowd. Since the IQR strategy worked well for our data, we can use the thresholds we defined before, i.e. `iqr_upper` and `iqr_lower`. Therefore, an observation (i.e. an area in England) is considered an outlier if

- its value lies above `iqr_upper`, or
- its value lies below `iqr_lower`

It becomes clear that we somehow need to define a condition because if it is an outlier, it should be labelled as one, but if not, then obviously we need to label the observation differently. Ideally, we want a new column in our dataset which indicates whether an area in England is an outlier (i.e. `outlier == TRUE`) or not (`outlier == FALSE`). *R* offers a way for us to express such conditions with the function `ifelse()`. It has the following structure:

```
ifelse(condition, TRUE, FALSE)
```

Let's formulate a sentence that describes our scenario as an `ifelse()` function:

- If an area's `happiness` is higher than `iqr_upper`, or
- if an area's `happiness` is lower than `iqr_lower`,
- classify this area as an outlier (i.e. `TRUE`),
- otherwise, classify this area as not being an outlier (i.e. `FALSE`).

We already know from Section 5.1 how to use logical and arithmetic operators. All we have to do is put them together in one function call and create a new column with `mutate()`.

```
hie_2021_outliers <-
  hie_2021 |>
  mutate(outlier = ifelse(happiness > iqr_upper |
                           happiness < iqr_lower,
                           TRUE, FALSE))
```

Since we have now a classification, we can more thoroughly filter for and inspect our outliers, i.e. we can see which areas in England are the ones that are lying outside our defined norm. We can `arrange()` our outliers by `happiness` to give the output even more meaning and improve readability

206 9 Sources of bias: Sampling, outliers, normality and other ‘conundrums’

```
hie_2021_outliers |>
  filter(outlier == "TRUE") |>
  select(area_name, happiness, outlier) |>
  arrange(happiness)
```

```
# A tibble: 9 × 3
  area_name  happiness outlier
  <fct>      <dbl>   <lgl>
1 Colchester    6.76  TRUE
2 Redditch      6.84  TRUE
3 Norwich       6.88  TRUE
4 Blaby          8.03  TRUE
5 Ryedale        8.03  TRUE
6 Hambleton     8.05  TRUE
7 Pendle         8.1   TRUE
8 Lichfield      8.1   TRUE
9 Torridge       8.16  TRUE
```

Given that the median of happiness for this dataset is 7.47, we can postulate that `Colchester` and `Redditch` are places in England that score considerably below the majority of the rest. However, places like `Torridge` and `Lichfield` can be considered particularly happy places in England. This list of areas in England represents to the answer to our question at the beginning of this section.

From here, it is simple to remove these areas in England (i.e. keep the areas that are not outliers) or set their values to `NA`. If we `replace()` the values with `NA`, we can continue with one of the techniques demonstrated for missing values in Section 7.7. Both approaches are shown in the following code chunk.

```
# Keep all observations but outliers
hie_2021_outliers |>
  filter(outlier == "FALSE") |>
  select(area_name, outlier)
```

```
# Replace values with NA
hie_2021_outliers |>
  mutate(happiness = replace(happiness, outlier == "TRUE", NA)) |>
  select(area_name, happiness)
```

```
# A tibble: 307 × 2
  area_name              happiness
  <fct>                  <dbl>
1 Hartlepool            7.54
```

```

2 Middlesbrough           7.41
3 Redcar and Cleveland    7.46
4 Stockton-on-Tees        7.23
5 Darlington               7.35
6 Halton                   7.36
7 Warrington                7.38
8 Blackburn with Darwen   7.36
9 Blackpool                 7.43
10 Kingston upon Hull, City of 7.38
# i 297 more rows

```

The `replace()` function is very intuitive to use. It first needs to know where you want to replace a value (`happiness`), then what the condition for replacing is (`outlier == "TRUE"`), and lastly, which value should be put instead of the original one (`NA`).

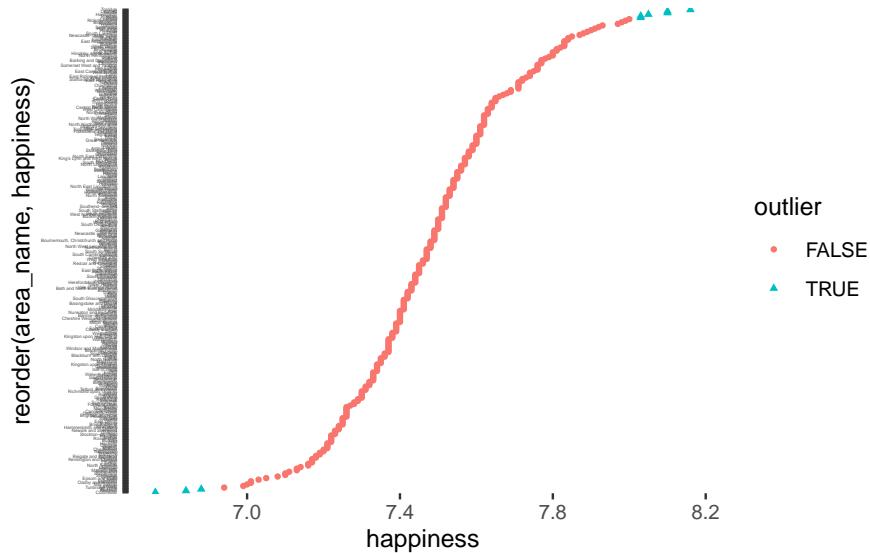
9.6.4 Concluding remarks about outliers

As you hopefully noticed, understanding your data requires some effort, but it is important to know your data well before proceeding to any further analysis. You can experiment with different data visualisations and design them in a way that best reflects the message you want to get across. For example, because we have added a new variable that classifies outliers, we can do more with our data visualisation than before and highlight them more clearly.

```

hie_2021_outliers |>
  ggplot(aes(x = reorder(area_name, happiness),
             y = happiness,
             col = outlier,
             shape = outlier))
  ) +
  geom_point(size = 1) +
  theme(panel.background = element_blank(),
        axis.text.y = element_text(size = 2)) +
  coord_flip()

```



To round things off, let me share with you one more visualisation, which demonstrates that outliers might also have to be defined based on specific groupings of your data. For example, if we look at the `gep` dataset that we used before, we could compare distributions based on `level_of_study` e.g. UG, PGT, PGR or Other. Thus, it might be essential to consider outliers in light of sub-samples of your data rather than the entire dataset, especially when comparing groups (see Chapter 11).

```
# The wesanderson package can make your plots look more
# 'Wes Anderson'
colour_pal <-
  wesanderson::wes_palette("Darjeeling1",
    11,
    type = "continuous")

gep |>
  ggplot(aes(x = reorder(level_of_study,
    si_socialise_with_people_exp,
    FUN = median),
    y = si_socialise_with_people_exp,
    col = level_of_study)
  ) +
  geom_boxplot(alpha = 0,
    show.legend = FALSE) +
  geom_jitter(width = 0.1,
    size = 0.5,
```

```
alpha = 0.5,  
show.legend = FALSE  
) +  
scale_color_manual(values = colour_pal) +  
coord_flip() +  
theme(panel.background = element_blank()) +  
xlab("Level of study") +  
ylab("Socialising with people") +  
ggtitle("Distribution of 'socialising with people' in the GEP dataset")
```



Figure 9.13



10

Correlations

Sometimes, counting and measuring means, medians, and standard deviations is not enough because they are all based on a single variable. Instead, we might have questions about the relationship between two or more variables. In this section, we will explore how correlations can (and kind of cannot - see Section 10.4.3) provide insights into the following questions:

- Do people who eat healthier also feel happier?
- Are happier people also more satisfied with their life?
- Is eating healthier related to other health factors, such as life expectancy, blood pressure and overweight/obesity?

There are many different ways to compute the correlation, and it partially depends on the type of data you want to relate to each other. The '*Pearson's correlation*' is by far the most frequently used correlation technique for normally distributed data. On the other hand, if data is not normally distributed, we can opt for the '*Spearman's rank*' correlation. One could argue that the relationship between these two correlations is like the mean (Pearson) to the median (Spearman). Both approaches require numeric values to be computed correctly. If our data is ordinal or, worse, dichotomous (like a logical variable), we have to choose different options. Table 10.1 summarises different methods depending on the data type.

Table 10.1: Different ways of computing correlations

Correlation	Used when ...
<i>Pearson</i>	... variables are numeric and parametric
<i>Spearman</i>	... variables are numeric and non-parametric
<i>Polychoric</i>	... investigating two ordinal variables
<i>Tetrachoric</i>	... both variables are dichotomous, e.g. 'yes/no', 'True/False'.
<i>Rank-biserial</i>	... one variable is dichotomous and the other variable is ordinal

There are many more variations of correlations, which you can explore on the package's website¹ we will use in this chapter: `correlation`. However, we will primarily focus on Pearson and Spearman because they are the most commonly used correlation types in academic publications to understand the relationship between two variables. In addition, we also look at '*partial correlations*', which allow us to introduce a third variable into this mix.

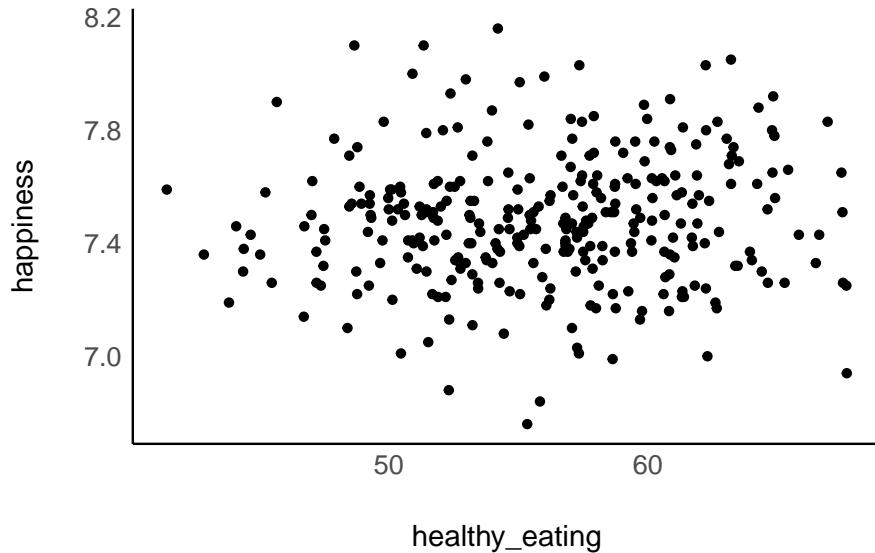
10.1 Plotting correlations

Since correlations only show the relationship between two variables, we can easily put one variable on the x-axis and one on the y-axis, creating a so-called 'scatterplot'. We used two functions to plot scatterplots before, i.e. `geom_point()` and `geom_jitter`. Let's explore our first research question: Do people who eat healthier also feel happier? One assumption might be that healthier eating habits contribute to better physical well-being, which in turn enhances happiness. However, a counter-argument could be that diet does not significantly impact happiness levels, as everyone is subject to various life circumstances that influence our mood. Either way, we first need to identify the two variables of interest, which we can find in the `hie_2021` dataset. It contains measures of national health in various areas of England collected in 2021, such as:

- `happiness`, which measures the average happiness of respondents when asked how they felt the day before on a scale from 0 (not at all) to 10 (completely), and
- `healthy_eating`, which represents the percentage of adults who report eating five or more portions of fruit and vegetables on a regular day.

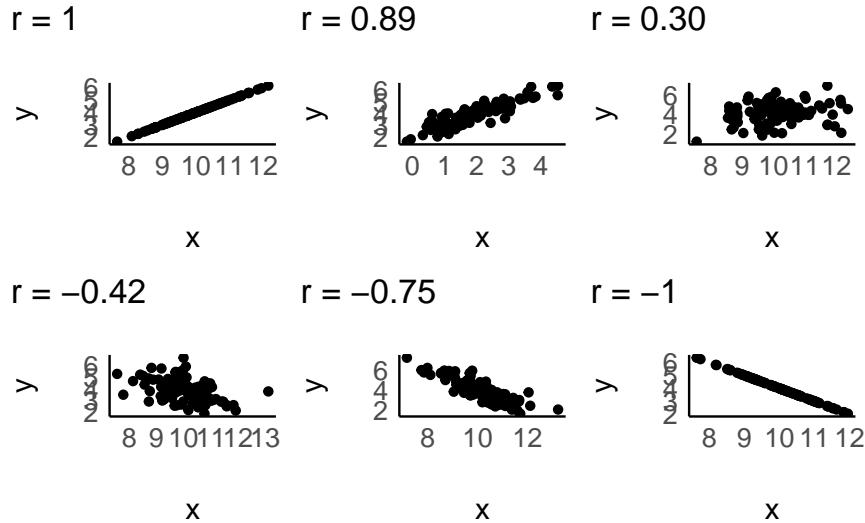
```
hie_2021 |>
  ggplot(aes(healthy_eating, happiness)) +
  geom_point() +
  see::theme_modern()
```

¹<https://easystats.github.io/correlation/articles/types.html>



The results from our scatterplot are, well, somewhat random. We can see that in some areas of England people report very high `happiness` scores as well as high levels of `healthy_eating`. However, some regions score high on `happiness` but low regarding `healthy_eating`. Overall, the points look like they are randomly scattered all over our canvas. The only visible pattern we can notice is that there are slightly more scores at the center of this cloud of dots than on its edges.

Since correlations only explain linear relationships, a perfect correlation would be represented by a straight line. Consider the following examples of correlations:



A correlation can be either positive or negative, and its value (i.e. r in case of the Pearson correlation) can range from -1 to 1 :

- -1 defines a perfectly negative correlation,
- 0 defines no correlation (completely random), and
- 1 defines a perfectly positive correlation.

In other words, the further the score is away from 0 , the stronger is the relationship between variables. We also have benchmarks that we can use to assess the strength of a relationship, for example, the one by J. Cohen (1988). The strength of the relationship is also called ‘*effect size*’. Table 10.2 shows the relevant benchmarks. Note that effect sizes are always provided as absolute figures. Therefore, -0.4 would also count as a moderate relationship.

Table 10.2: Assessing effect size of relationships according to J. Cohen (1988).

effect size	interpretation
$r < 0.1$	very small
$0.1 \leq r < 0.3$	small
$0.3 \leq r < 0.5$	moderate
$r \geq 0.5$	large

10.2 Computing correlations

If we compare the plot from our data with the sample plots, we can conclude that the relationship is weak, and therefore the r must be close to zero. We can test this with the Pearson correlation using the function `correlation()` from the `correlation` package. By default, it will perform a Pearson correlation, which is only applicable for parametric data. As outlined before, if our data violates the assumptions for parametric tests, we can use Spearman's correlation instead. For the rest of the chapter, we assume that our data is parametric, but I will demonstrate how to compute both in the following code chunk:

```
library(correlation)

# Pearson correlation
hie_2021 |>
  select(happiness, healthy_eating) |>
  correlation()

# Correlation Matrix (pearson-method)

Parameter1 |      Parameter2 |      r |      95% CI | t(305) |      p
-----
happiness | healthy_eating | 0.07 | [-0.04, 0.18] | 1.24 | 0.215

p-value adjustment method: Holm (1979)
Observations: 307

# Spearman correlation
hie_2021 |>
  select(happiness, healthy_eating) |>
  correlation(method = "spearman")

# Correlation Matrix (spearman-method)

Parameter1 |      Parameter2 |      rho |      95% CI |      S |      p
-----
happiness | healthy_eating | 0.09 | [-0.03, 0.20] | 4.41e+06 | 0.135

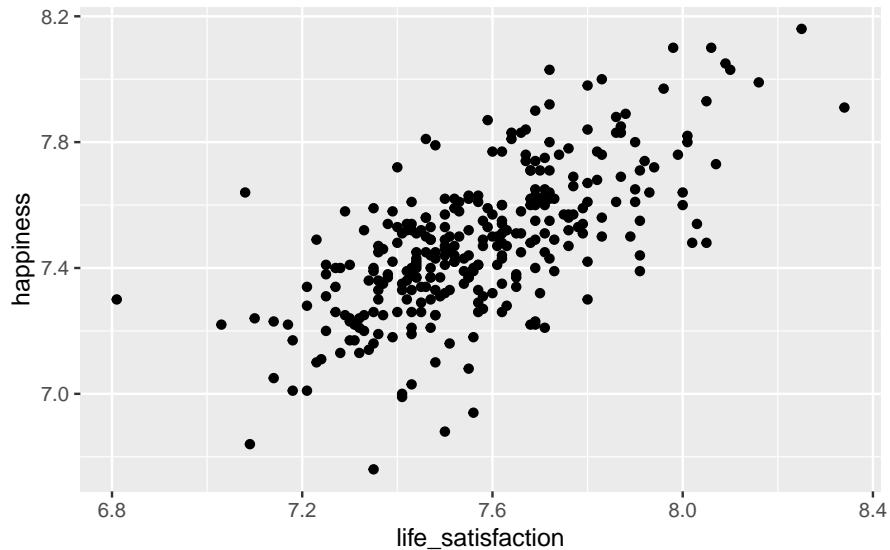
p-value adjustment method: Holm (1979)
Observations: 307
```

Indeed, our analysis reveals that the effect size is very small ($r = 0.07 < 0.1$).

Therefore, `healthy_eating`, seems to not make us happier. In all fairness, the foods that make us smile are often not fruits and veggies, but other, more guilty, pleasures.

Another question we posed at the beginning was: Are people who are happier also more satisfied with their life? Our intuition might say ‘yes’. A person who consistently feels happy is more likely to assess their overall life positively, because their day-to-day experiences are enjoyable. Let’s create another scatterplot to find out.

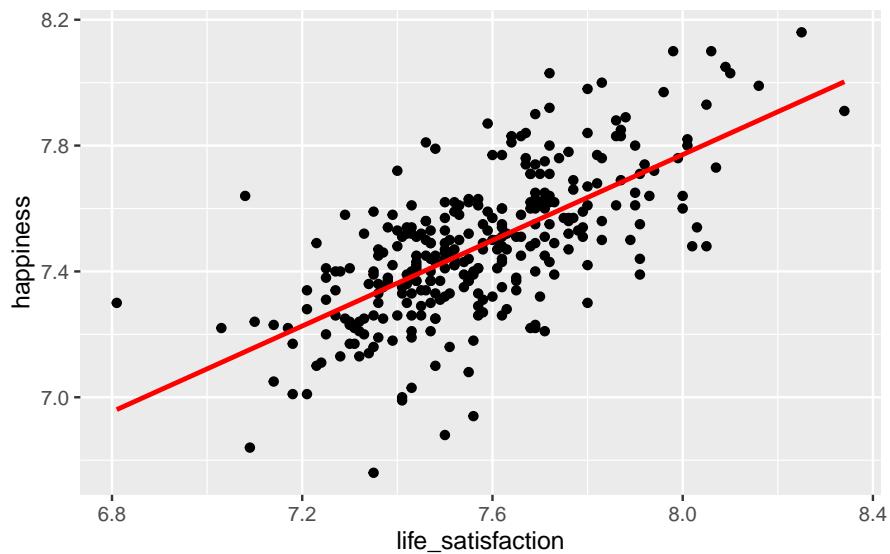
```
hie_2021 |>
  ggplot(aes(x = life_satisfaction, y = happiness)) +
  geom_point()
```



The scatterplot shows a somewhat positive trend which suggests that we are right: Higher `happiness` is linked to higher `life_satisfaction`. Often, it can be tough to see the trend. To improve our plot, we can use the function `geom_smooth()`, which can help us draw a straight line that best fits our data points. We need to set the `method` for drawing the line to `lm`, which stands for ‘linear model’. Remember, correlations assume a linear relationship between two variables, i.e. we model a linear relationship in our scatterplot. As mentioned before, the stronger the relationship the closer each dot in the plot will be to the red line.

```
hie_2021 |>
  ggplot(aes(x = life_satisfaction, y = happiness)) +
```

```
geom_point() +
  geom_smooth(method = "lm",
              formula = y ~ x,
              se = FALSE,
              col = "red")
```



To find a definitive answer to this research question we can compute the correlation for both variables.

```
hie_2021 |>
  select(life_satisfaction, happiness) |>
  correlation()

# Correlation Matrix (pearson-method)

Parameter1      | Parameter2 |    r |    95% CI | t(305) |      p
-----|-----|-----|-----|-----|-----|
---|---|---|---|---|---|
life_satisfaction |  happiness | 0.67 | [0.60, 0.73] | 15.73 | < .001***
```

p-value adjustment method: Holm (1979)
Observations: 307

With an r of 0.67, we did not only find a positive relationship, but one that is considered *large*, according to Cohen (1988).

10.3 Significance: A way to help you judge your findings

One of the most common pitfalls of novice statisticians is the interpretation of what counts as significant and not significant. Most basic tests offer a ‘*p value*’, which stands for ‘*probability value*’. I have referred to this value multiple times in this book already, but did not thoroughly explain why it is important.

To understand p-values and significance, we need to briefly discuss *hypotheses*. A hypothesis is an assumption we make about the data, such as whether `healthy_eating` leads to more `happiness` (as in Section 10.2). In contrast, the null hypothesis is a special type of hypothesis that assumes that no effect or relationship exists between variables, i.e. the opposite of what we are testing. When you read that a hypothesis has been “rejected” or “accepted,” it is often the null hypothesis being tested. The p-value helps determine whether we should reject or retain the null hypothesis based on the data.

The p-value can range from 1 (for 100%) to 0 (for 0%) and helps us assess whether to accept or reject the null hypothesis:

- $p = 1$, there is a 100% probability that the result is due to pure chance, supporting (accepting) the null hypothesis.
- $p = 0$, there is a 0% probability that the result is due to pure chance, suggesting that the null hypothesis has to be rejected.

Technically, we would not find that p is ever truly zero and instead denote very small p-values with $p < 0.01$ or even $p < 0.001$, indicating that our null hypothesis is unlikely to be true.

There are also commonly considered and accepted thresholds for the p-value:

- $p > 0.05$, the result is not significant.
- $p \leq 0.05$, the result is significant, i.e. there is less than a 5% chance that the null hypothesis should have been accepted.
- $p \leq 0.01$, the result is highly significant.

Most frequently, confusion about the p-value emerges because we test FOR the null hypothesis and not what we are actually interested in. Returning to our example of eating healthy and its impact on happiness, we would first formulate our hypothesis and the null hypothesis:

- Hypothesis (H_1): Healthy eating and happiness are significantly positively related to each other.
- Null Hypothesis (H_0): There is no significant relationship between healthy eating and happiness.

Based on our research question, we hope to accept our hypothesis and, therefore, reject the null hypothesis. Hardly ever will you find that researchers are interested in not finding relationships/effects in their data. Consequently, a *significant* p-value would tell us to reject the null hypothesis and accept our hypothesis.

Let's revisit the results from our earlier analysis where we performed a Pearson correlation to test our assumption.

```
# Pearson correlation
hie_2021 |>
  select(happiness, healthy_eating) |>
  correlation()

# Correlation Matrix (pearson-method)

Parameter1 |      Parameter2 |      r |      95% CI | t(305) |      p
-----+-----+-----+-----+-----+-----+-----+
happiness | healthy_eating | 0.07 | [-0.04, 0.18] | 1.24 | 0.215

p-value adjustment method: Holm (1979)
Observations: 307
```

Besides the correlation coefficient (r), we also find $t_{(305)} = 1.24$, which is the test statistic. The test statistic helps us determine if our data shows something unusual based on our hypothesis. A test statistic close to 0 suggests that our data is consistent with the null hypothesis, i.e. no effect/relationship. The further the test statistic moves from 0, the more evidence we have against the null hypothesis.

In our analysis of the relationship between `happiness` and `healthy_eating`, we obtained a p-value of 0.215. This indicates that there is a 21.5% probability of finding a t-value of 1.24 or larger if the null hypothesis is true. You probably agree that these are not good odds. Since this p-value is greater than the conventional significance level of $p \leq 0.05$ we do not reject the null hypothesis. This means we do not have sufficient evidence to conclude that a significant relationship exists between `happiness` and `healthy_eating`.

Lastly, I also want to visually explain what the test statistic is and how it relates to p-values. We already learned about the normal distribution, as such, the visualisation in Figure 10.1 will look familiar.

What we can see here is the distribution of test statistic scores (t-values) related to our correlation analysis. We can see that all t-values that fall within the blue area of the distribution would not count as significantly different from the null hypothesis, which means we have to accept that there is no effect/relationship to be found. However, if the t-value falls into one of the

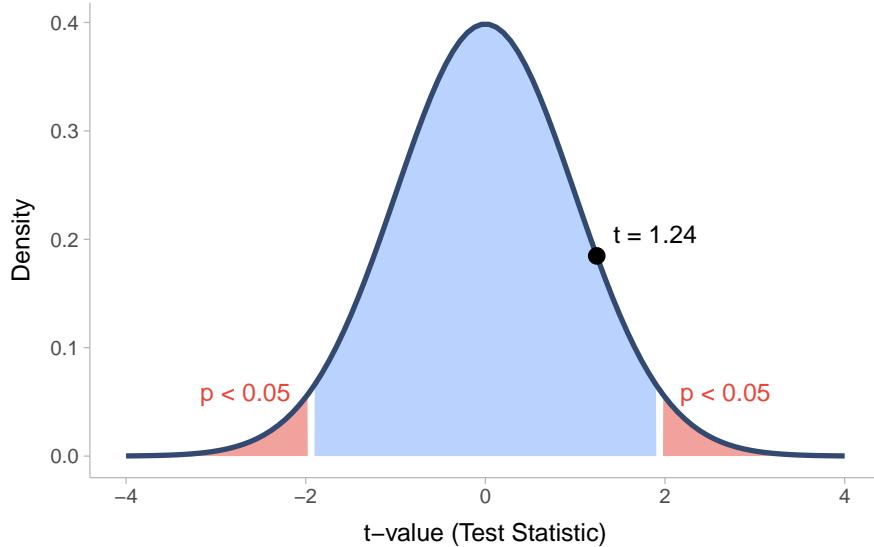


Figure 10.1: t-Distribution with test statistic

two red areas, we can reject the null hypothesis and be excited that the effect/relationship we found is *significant*.

While we clarified how the t-value is related to the p-value, you might still be confused how these two values relate to the correlation coefficient (r) in our correlation results. The answer is fairly simple: we need r to compute our t-value. The mathematical formula is as follows:

$$t = \frac{r\sqrt{n-2}}{\sqrt{1-r^2}}$$

With this knowledge, we can reproduce $t_{(305)} = 1.24$ writing the following lines of code:

```
# Define the correlation coefficient and sample size
r <- 0.07099542 # not rounded
n <- 307

# Calculate the t-value
t_value <- (r * sqrt(n - 2)) / sqrt(1 - r^2)
t_value
```

[1] 1.243018

We will cover more about the p-value in Chapter 11 and Chapter 13. For now,

it is important to know that a significant correlation is one that we should look at more closely. Usually, correlations that are not significant suffer from small effect sizes. However, different samples can lead to different effect sizes and significance levels. Consider the following examples:

```
# Correlation Matrix (pearson-method)

Parameter1 | Parameter2 |     r |      95% CI |   t(2) |      p
-----
x          |           y |  0.77 | [-0.73, 0.99] |  1.73 |  0.225

p-value adjustment method: Holm (1979)
Observations: 4

# Correlation Matrix (pearson-method)

Parameter1 | Parameter2 |     r |      95% CI |   t(10) |      p
-----
x          |           y |  0.77 | [0.36, 0.93] |  3.87 | 0.003**

p-value adjustment method: Holm (1979)
Observations: 12
```

In both examples $r = 0.77$, but the sample sizes are different (4 vs 12), and the p-values differ. In the first example, $p = 0.225$, which means the relationship is not significant, while in the second example, we find that $p < 0.01$ and is, therefore, highly significant. As a general rule, the bigger the sample, the more likely we find significant results, even though the effect size is small. Therefore, it is crucial to interpret correlations based on at least three factors:

- the p-value, i.e. significance level,
- the r-value, i.e. the effect size, and
- the n, i.e. the sample size

The interplay of all three can help determine whether a relationship is important. Therefore, when we include correlation tables in publications, we have to provide information about all three indicators.

Let's put all this information into practice and investigate our third research question, which asks whether eating healthy relates to other health factors such as life expectancy, blood pressure and overweight/obesity. While we could compute three correlations independently to find an answer to our question, it is much more common to include multiple variables simultaneously and look at the results conjointly.

```

hie_2021 |>
  select(healthy_eating, life_expectancy, high_blood_pressure, overweight_and_obesity_in_adults) |>
  correlation()

# Correlation Matrix (pearson-method)

Parameter1      |           Parameter2 |   r |    95% CI | t(305) |      p
-----+-----+-----+-----+-----+-----+-----+
-----+-----+
healthy_eating   |           life_expectancy |  0.70 | [ 0.64,  0.75] |  17.14 | < .001***  

healthy_eating   |           high_blood_pressure | -0.05 | [-0.16,  0.06] | -0.85 |  0.395  

healthy_eating   |           overweight_and_obesity_in_adults | -0.45 | [-0.54, -0.36] | -8.85 | < .001***  

life_expectancy  |           high_blood_pressure | -0.08 | [-0.19,  0.03] | -1.45 |  0.295  

life_expectancy  |           overweight_and_obesity_in_adults | -0.56 | [-0.63, -0.48] | -11.81 | < .001***  

high_blood_pressure |           overweight_and_obesity_in_adults |  0.10 | [-0.01,  0.21] |  1.76 |  0.237

p-value adjustment method: Holm (1979)
Observations: 307

```

If you have seen correlation tables before, you might find that `correlation()` does not produce the classic table by default. If you want it to look like the tables in publications, which are more compact but offer less information, you can use the function `summary()`. Which one you use it entirely up to you (or your supervisor).

```

hie_2021 |>
  select(healthy_eating, life_expectancy, high_blood_pressure, overweight_and_obesity_in_adults) |>
  correlation() |>
  summary()

# Correlation Matrix (pearson-method)

Parameter      | overweight_and_obesity_in_adults | high_blood_pressure | life_expectancy
-----+-----+-----+-----+
-----+-----+
healthy_eating  |                               -0.45*** |          -  

0.05 |           0.70***  

life_expectancy |                               -0.56*** |        -0.08 |  

high_blood_pressure |                           0.10 |          |

```

p-value adjustment method: Holm (1979)

This table provides a clear answer to our final question. While `healthy_eating` positively relates to `life_expectancy` ($r = 0.70, p < 0.001$) and negatively to `overweight_and_obesity_in_adults` ($r = -0.45, p < 0.001$), it does not correlate with `high_blood_pressure` ($r = 0.05, p < 0.395$). In the classic correlation table, you often see *. These stand for the different significant levels:

- *, i.e. $p < 0.05$
- **, i.e. $p < 0.01$
- ***, i.e. $p < 0.001$ (you might find some do not use this as a separate level)

In short, the more * there are attached to each value, the more significant a result. Therefore, we likely find the same relationships in newly collected data if there are many * and we can trust our findings.

10.4 Limitations of correlations

Correlations are helpful, but only to some extend. The three most common limitations you should be aware of are:

- Correlations are not causal relationships
- Correlations can be spurious
- Correlations might only appear in sub-samples of your data

10.4.1 Correlations are not causal relationships

Correlations do not offer insights into causality, i.e. whether a change in one variable causes change in the other variable. Correlations only provide insights into whether these two variables tend to change when one of them changes. Still, sometimes we can infer such causality by the nature of the variables. For example, in countries with heavy rain, more umbrellas are sold. Buying more umbrellas will not cause more rain, but if there is more rain in a country, we rightly assume a higher demand for umbrellas. If we can theorise the relationship between variables, we would rather opt for a regression model instead of a correlation (see Chapter 13).

10.4.2 Correlations can be spurious

Just because we find a relationship between two variables does not necessarily mean that they are truly related. Instead, it might be possible that a third vari-

able is the reason for the relationship. We call relationships between variables that are caused by a third variable ‘*spurious correlations*’. This third variable can either be part of our dataset or even something we have not measured at all. The latter case would make it impossible to investigate the relationship further. However, we can always test whether some of our variables affect the relationship between the two variables of interest. This can be done by using *partial correlations*. A *partial correlation* returns the relationship between two variables minus the relationship to a third variable. Figure 10.2 depicts this visually. While *a* and *b* appear to be correlated, the correlation might only exist because they correlate with *x*.

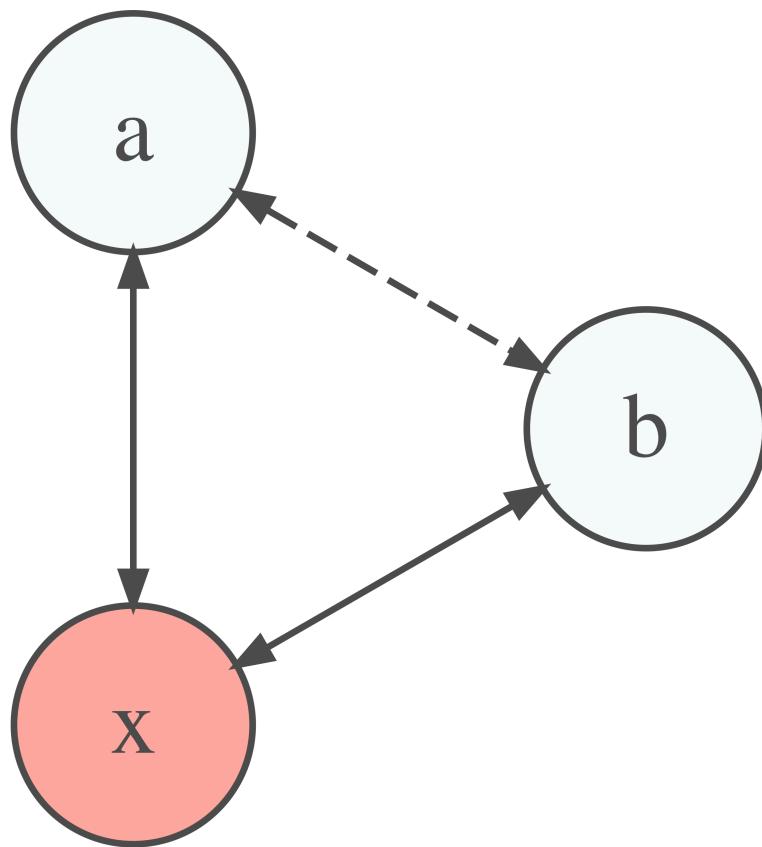


Figure 10.2: Illustration of a spurious correlation

Let’s return to our first research questions. We found that `healthy_eating` is not related to `happiness`, but to `life_satisfaction`. `Life_satisfaction`, however, is related to `happiness`.

```
# Correlation between variables
hie_2021 |>
  select(healthy_eating, happiness, life_satisfaction) |>
  correlation()

# Correlation Matrix (pearson-method)

Parameter1 | Parameter2 |   r |      95% CI | t(305) |      p
-----+-----+-----+-----+-----+-----+
-----+
healthy_eating |      happiness | 0.07 | [-0.04, 0.18] |  1.24 | 0.215
healthy_eating | life_satisfaction | 0.23 | [ 0.12, 0.33] |  4.05 | < .001***  

happiness     | life_satisfaction | 0.67 | [ 0.60, 0.73] | 15.73 | < .001***
```

p-value adjustment method: Holm (1979)
Observations: 307

However, if we consider the impact of `life_satisfaction` on the relationship between `heathly_eating` and `happiness` when we compute the correlation, we retrieve a very different result.

```
# Correlation between variables, considering partial correlations
hie_2021 |>
  select(healthy_eating, happiness, life_satisfaction) |>
  correlation(partial = TRUE)

# Correlation Matrix (pearson-method)

Parameter1 | Parameter2 |   r |      95% CI | t(305) |      p
-----+-----+-----+-----+-----+-----+
-----+
healthy_eating |      happiness | -0.11 | [-0.22, 0.00] | -1.95 | 0.053
healthy_eating | life_satisfaction | 0.24 | [ 0.13, 0.34] |  4.33 | < .001***  

happiness     | life_satisfaction | 0.67 | [ 0.61, 0.73] | 15.86 | < .001***
```

p-value adjustment method: Holm (1979)
Observations: 307

Suddenly, `eating_healthy` and `happiness` show a slight negative correlation ($r = -0.11$). In other words, by controlling for the effect of `life_satisfaction` on both variables, we reversed the relationship from positive to negative. You might wonder whether this implies that eating healthily makes us less happy. If we were to take this result superficially, it would indicate exactly this. However, as mentioned before, when we look at correlations we have to consider not only r but also the p-value and the sample size. While sample size

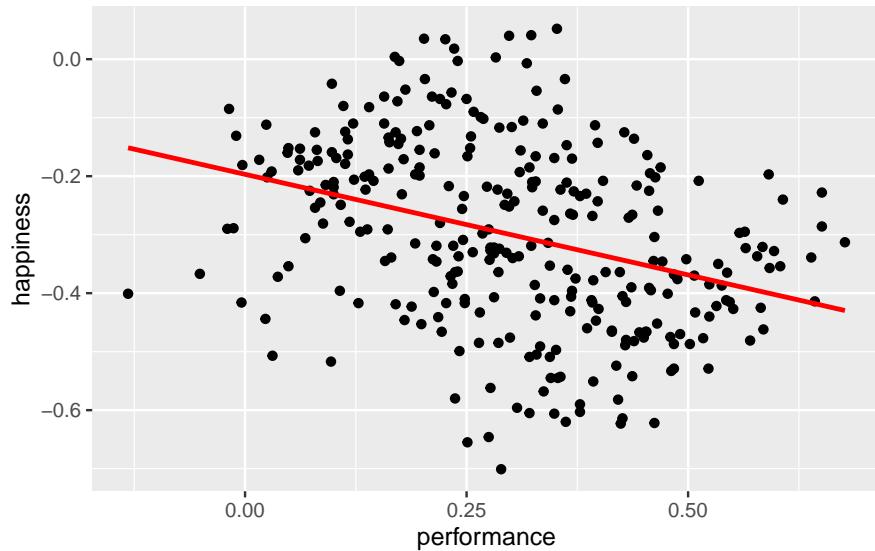
seems less problematic ($n = 307$), the p-value is only $p = 0.053$, which is just shy of the required minimum significance level of 0.05 . If we strictly applied the rules, this relationship would not be classified as ‘significant’. Thus, the relationship would not be strong enough to derive meaningful conclusions. Even if the p-value were slightly lower, the effect size would still be very small.

10.4.3 Simpson’s Paradox: When correlations betray you

The final limitation is so important that it even has its own name: the ‘*Simpson’s Paradox*’. Let’s find out what is so paradoxical about some correlations. For this demonstration, we have to make use of a different dataset: `simpson` of the `r4np` package. It contains information about changes in student performance and changes in happiness. The dataset includes responses from three different groups: Teachers, Students and Parents.

We would assume that an increase in students’ performance will likely increase the happiness of participants. After all, all three have stakes in students’ performance.

```
simpson |>
  ggplot(aes(x = performance,
             y = happiness)) +
  geom_point() +
  geom_smooth(method = "lm",
              formula = y ~ x,
              se = FALSE,
              color = "red")
```



```

simpson |>
  select(performance, happiness) |>
  correlation()

# Correlation Matrix (pearson-method)

Parameter1 | Parameter2 |      r |      95% CI |   t(298) |      p
-----+-----+-----+-----+-----+-----+
  performance |  happiness | -0.34 | [-0.44, -0.24] | -6.32 | < .001***
```

p-value adjustment method: Holm (1979)
 Observations: 300

It appears we were terribly wrong. It seems as if `performance` and `happiness` are moderately negatively correlated with each other. Thus, the more a student improves their `performance`, the less happy teachers, parents, and students are. I hope you agree that this is quite counter-intuitive. However, what could be the cause for such a finding?

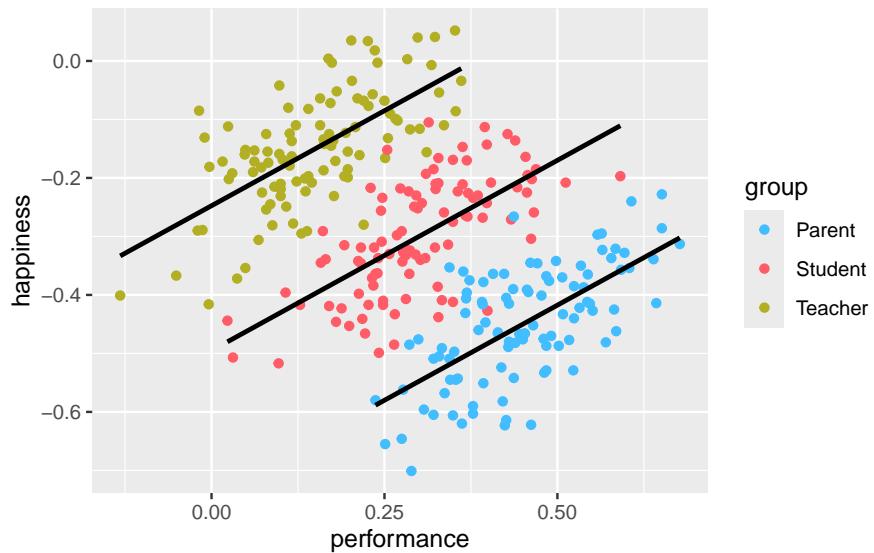
If you have the instincts of a true detective², you would think that maybe we should look at each group of participants separately. So, let's plot the same scatterplot again but colour the responses of each participant group differently and also compute the correlation for each subset of our data.

²<https://www.imdb.com/title/tt2356777/episodes?season=1>

```

simpson |>
  ggplot(aes(x = performance,
             y = happiness,
             group = group,
             col = group)) +
  geom_point() +
  geom_smooth(method = "lm",
              formula = y ~ x,
              se = FALSE,
              col = "black") +
  # Choose custom colours for the dots
  scale_color_manual(values = c("#42BDFF", "#FF5C67", "#B3AF25"))

```



```

simpson |>
  group_by(group) |>
  correlation()

```

```
# Correlation Matrix (pearson-method)
```

Group	Parameter1	Parameter2	r	95% CI	t(98)	p
Parent	performance	happiness	0.65	[0.52, 0.75]	8.47	< .001***
Student	performance	happiness	0.65	[0.52, 0.75]	8.47	< .001***

```
Teacher | performance | happiness | 0.65 | [0.52, 0.75] | 8.47 | <.001***
```

p-value adjustment method: Holm (1979)

Observations: 100

The results have magically been inverted. Instead of a negative correlation, we now find a strong positive correlation among all groups. This result seems to make much more sense.

When computing correlations, we need to be aware that subsets of our data might show different directions of correlations. Sometimes insignificant correlations might suddenly become significant. This is what the *Simpson's paradox* postulates.

If your study relies solely on correlations to detect relationships between variables, which it hopefully does not, it is essential to investigate whether the detected (or undetected) correlations exist. Of course, such an investigation can only be based on the data you obtained. The rest remains pure speculation. Nevertheless, correlations are beneficial to review the bilateral relationship of your variables. It is often used as a pre-test for regressions (see Chapter 13) and similarly more advanced computations. As a technique to make inferences, correlations are a good starting point but should be complemented by other steps, if possible.



11

Comparing groups

Social Sciences is about the study of human beings and their interactions. As such, we frequently want to compare two or more groups of human beings, organisations, teams, countries, etc., with each other to see whether they are similar or different from each other. Sometimes we also want to track individuals over time and see how they may have changed in some way or other. In short, comparing groups is an essential technique to make inferences and helps us better understand the diversity surrounding us.

If we want to perform a group comparison, we have to consider which technique is most appropriate for our data. For example, some of it might be related to the type of data we have collected, and other aspects might be linked to the distribution of the data. More specifically, before we apply any statistical technique, we have to consider at least the following:

- missing data (see Section 7.7),
- outliers (see Section 9.6), and
- the assumptions made by analytical techniques about our data.

While we covered missing data and outliers in previous chapters, we have yet to discuss assumptions. For group comparisons, there are three main questions we need to answer:

- Are the groups big enough to be compared, i.e. are they comparable?
- Is my data parametric or non-parametric? (see Chapter 9)
- How many groups do I wish to compare?
- Are these groups paired or unpaired?

In the following, we will look at group comparisons for parametric and non-parametric data in each category and use the `wvs_nona` dataset, i.e. the `wvs` data frame after we performed imputation (see also Section 7.7.3). Since we already covered how to test whether data is parametric or non-parametric, we will forgo this step out of pure convenience and remain succinct. We also ignore any potential outliers. Thus, parametric and non-parametric tests will be demonstrated with the same dataset and the same variables.

11.1 Comparability: Apples vs Oranges

Before we can jump into group comparisons, we need to make ourselves aware of whether our groups can be compared in the first place. ‘*Comparability*’ should not be confused with ‘are the groups equal’. In many cases, we don’t want groups to be equal in terms of participants, e.g. between-subject studies. On the other hand, we might wish for groups to be perfectly equal when we perform within-subject studies. Thus, asking whether groups are comparable is unrelated to whether the subjects in our study are the same. Instead, we are looking at the characteristics of our groups. Some commonly considered features include:

- *Size*: Are the groups about equally large?
- *Time*: Was the data collected around the same time?
- *Exogenous variables*: Is the distribution of characteristics we are not interested in approximately the same across groups?

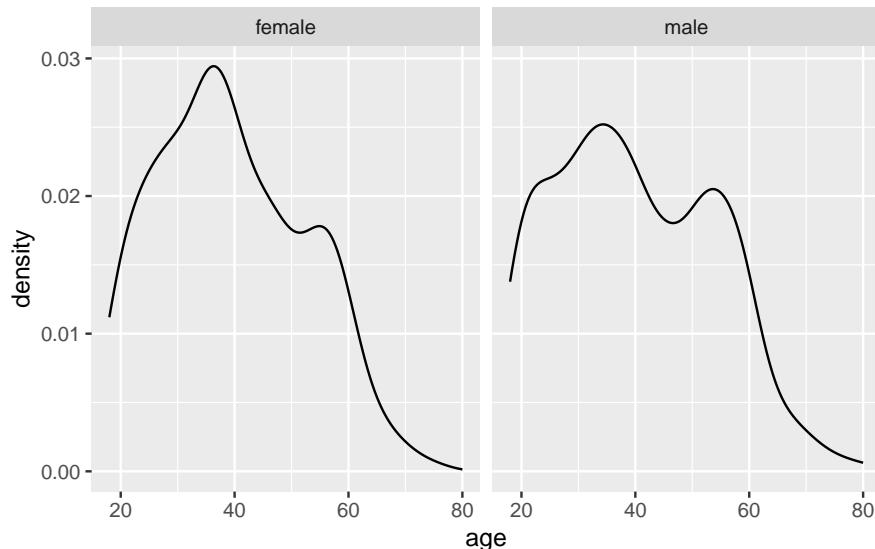
When we compare groups, we want to minimise the systematic differences that are not the primary focus of our study. Using the right sampling technique can help with this matter. For example, using a random sample and performing a random allocation to groups can help achieve comparable groups and remove systematic differences in a way no other sampling strategy can. However, there is still no guarantee that they will be comparable Berger (2006). Besides, we also face the challenge that in Social Sciences, we do not always have the option of random sampling. For example, International Business studies heavily rely on lists provided by others (e.g. the European Union, Fortune 500, etc.), personal judgement and convenience sampling. Only a small proportion perform probability sampling (Yang, Wang, and Su 2006). In short, there is no reason to worry if your sampling technique is not random. However, it emphasises the need to understand your sample and your groups thoroughly. To inspect characteristics of groups we wish to compare, we can use descriptive statistics as we covered them in Chapter 8). However, this time, we apply these techniques to subsets of our data and not the entire dataset.

For example, we might wish to compare `female` and `male` Egyptians (see Section 11.2.1). If we wanted to make sure these two groups can be compared, we might have to check (among other characteristics) whether their age is distributed similarly. We can use the functions we already know to create a plot to investigate this matter and apply the function `facet_wrap()` at the end.

```
# Only select participants from 'Egypt'  
comp <-  
  wvs_nona |>
```

```
filter(country == "Egypt")

# Plot distribution using facet_wrap()
comp |>
  ggplot(aes(x = age)) +
  geom_density() +
  facet_wrap(~ gender)
```

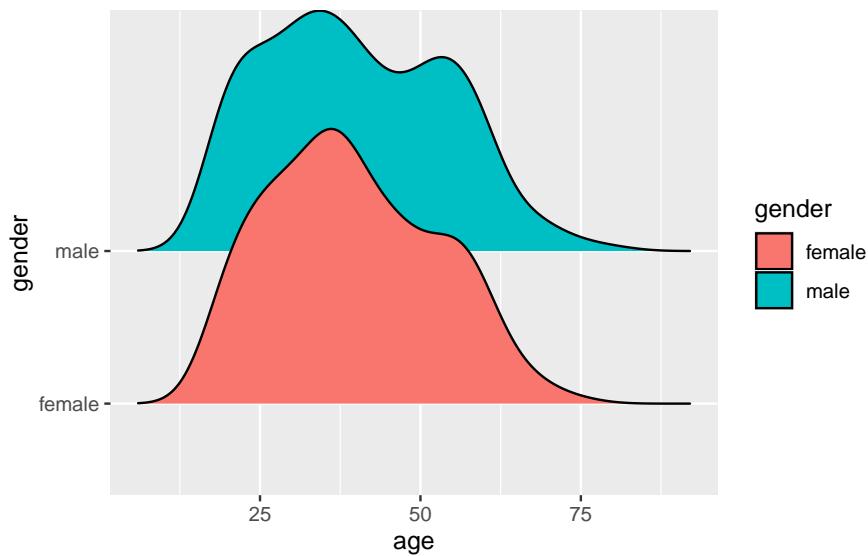


The function `facet_wrap(~ gender)` allows us to create two plots based on a grouping variable, i.e. gender. We have to use a tilde (~) to indicate the group. This character is related to writing formulas in R, which we will cover in more detail in the following chapters. Being able to place plots next to each other can be very beneficial for comparison. This way, inspecting the normality across multiple groups can be done within seconds. On the other hand, creating separate plots for each group can take a long time, for example, comparing 48 countries.

Alternatively, we could also create density plots using the `ggridges` package, which puts all the plots onto the same x-axis, making it even easier to see the differences across groups.

```
# Plot distributions
comp |>
  ggplot(aes(x = age,
              y = gender,
```

```
fill = gender)) +
ggridges::geom_density_ridges(bandwidth = 4)
```



In light of both data visualisations, we can conclude that the distribution of age across both gender groups is fairly similar and likely not different between groups. Of course, we could also statistically explore this using a suitable test before performing the main group comparison. However, we first have to understand how we can perform such tests.

In the following chapters, we will primarily rely on the package `rstatix`, which offers a pipe-friendly approach to using the built-in functions of *R* to perform our group comparisons. However, you are welcome to try the basic functions, which you can find in the Appendix.

```
library(rstatix)
```

11.2 Comparing two groups

The simplest of comparisons is the one where you only have two groups. These groups could either consist of different people (unpaired) or represent two measurements of the same individuals (paired). We will cover scenarios in

the following chapter because they require slightly different computational techniques.

11.2.1 Two unpaired groups

An unpaired group test assumes that the observations in each group are not related to each other, for example, observations in each group are collected from different individuals.

Our first comparison will be participants from Egypt, and we want to understand whether `male` and `female` citizens in this country perceive their `freedom_of_choice` differently or equally.

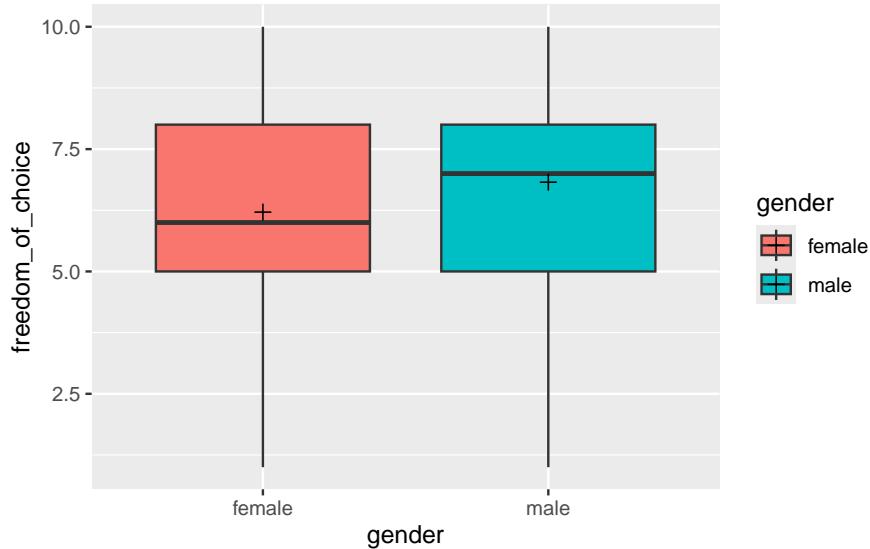
We first can compare these two groups using our trusty `geom_boxplot()` (or any variation) and use different `fill` colours for each group.

```
# Compute the mean for and size of each group
group_means <-
  comp |>
    group_by(gender) |>
    summarise(mean = mean(freedom_of_choice),
              n = n())

group_means

# A tibble: 2 × 3
  gender   mean     n
  <fct>   <dbl> <int>
1 female   6.21    579
2 male     6.82    621

# Create our data visualisation
comp |>
  ggplot(aes(x = gender, y = freedom_of_choice, fill = gender)) +
  geom_boxplot() +
  # Add the mean for each group
  geom_point(data = group_means,
             aes(x = gender, y = mean),
             shape = 3,
             size = 2)
```



While the distribution looks similar, we notice that the median and the mean (marked by the cross inside the boxplot) are slightly higher for `male` participants. Thus, we can suspect some differences between these two groups, but we do not know whether these differences are significant or not. Therefore, to consider the significance (remember Section 10.3) and the effect size (see Table 10.2), we have to perform statistical tests.

Table 11.1 summarises the different tests and functions to perform the group comparison computationally. It is important to note that the parametric test compares the means of two groups, while the non-parametric test compares medians. All of these tests turn significant if the differences between groups are large enough. Thus, significant results can be read as ‘these groups are significantly different from each other’. Of course, if the test is not significant, the groups are considered to be not different. For parametric tests, i.e. `t_test()`, it is also essential to indicate whether the variances between these two groups are equal or not. Remember, the equality of variances was one of the assumptions for parametric tests. The *Welch t-test* can be used if the variances are not equal, but all other criteria for normality are met. By setting `var.equal = TRUE`, a regular *T-Test* would be performed. By default, `t_test()` assumes that variances are not equal. Make sure you test for *homogeneity of variance* before making your decision (see Section 9.5).

Table 11.1: Comparing two unpaired groups

Assumption	Test	Function	Effect size	Function
Parametric	T-Test	<code>t_test(var.equal = TRUE)</code>	Cohen's d	<code>cohens_d()</code>
	Welch			
	T-Test	<code>t_test(var.equal = FALSE)</code>		
Non-parametric	Mann-Whitney U	<code>wilcox_test(paired = FALSE)</code>	Wilcoxon R	<code>wilcox_effsize()</code>

With this information in hand, we can start comparing the `female` Egyptians with the `male` ones using both the parametric and the non-parametric test for illustration purposes only. By setting `detailed = TRUE`, we can obtain the maximum amount of information for certain comparisons. In such cases, it is advisable to use `glimpse()`. This will make the output (a `tibble`) easier to read because each row presents one piece of information, rather than having one row with many columns.

```
# T-Test
comp |> t_test(freedom_of_choice ~ gender,
             var.equal = TRUE,
             detailed = TRUE) |>
  glimpse()
```

```
Rows: 1
Columns: 15
$ estimate      <dbl> -0.6120414
$ estimate1     <dbl> 6.212435
$ estimate2     <dbl> 6.824477
$ .y.           <chr> "freedom_of_choice"
$ group1        <chr> "female"
$ group2        <chr> "male"
$ n1            <int> 579
$ n2            <int> 621
$ statistic     <dbl> -4.75515
$ p              <dbl> 2.22e-06
$ df             <dbl> 1198
$ conf.low       <dbl> -0.864566
$ conf.high      <dbl> -0.3595169
$ method         <chr> "T-test"
$ alternative    <chr> "two.sided"
```

```
# Welch t-test (var.equal = FALSE by default)
comp |> t_test(freedom_of_choice ~ gender,
              var.equal = FALSE,
              detailed = TRUE) |>
  glimpse()
```

```
Rows: 1
Columns: 15
$ estimate      <dbl> -0.6120414
$ estimate1     <dbl> 6.212435
$ estimate2     <dbl> 6.824477
$ .y.           <chr> "freedom_of_choice"
$ group1        <chr> "female"
$ group2        <chr> "male"
$ n1            <int> 579
$ n2            <int> 621
$ statistic     <dbl> -4.756287
$ p             <dbl> 2.21e-06
$ df            <dbl> 1193.222
$ conf.low      <dbl> -0.8645067
$ conf.high     <dbl> -0.3595762
$ method         <chr> "T-test"
$ alternative   <chr> "two.sided"
```

```
# Mann-Whitney U test
comp |>
  wilcox_test(freedom_of_choice ~ gender,
               detailed = TRUE) |>
  glimpse()
```

```
Rows: 1
Columns: 12
$ estimate      <dbl> -0.999948
$ .y.           <chr> "freedom_of_choice"
$ group1        <chr> "female"
$ group2        <chr> "male"
$ n1            <int> 579
$ n2            <int> 621
$ statistic     <dbl> 149937.5
$ p             <dbl> 4.97e-07
$ conf.low      <dbl> -0.9999694
$ conf.high     <dbl> -5.79946e-05
$ method         <chr> "Wilcoxon"
$ alternative   <chr> "two.sided"
```

You might notice that the notation within the functions for group tests looks somewhat different to what we are used to, i.e. we use the \sim ('tilde') symbol. This is because some functions take a formula as their attribute, and to distinguish the dependent and independent variables from each other, we use \sim . A more generic notation of how formulas in functions work is shown below, where **DV** stands for '*dependent variable*' and **IV** stands for '*independent variable*':

```
function(formula = DV ~ IV)
```

Even for multiple groups, group comparisons usually only have one independent variable, i.e. the grouping variable. Grouping variables are generally of the type **factor**. In the case of two groups, we have two levels present in this factor. For example, our variable **gender** contains two levels: **female** and **male**. If there are multiple groups, the factor comprises various levels, e.g. the variable **country** includes levels of 48 countries.

No matter which test we run, it appears as if the difference is significant. However, how big/important is the difference? The effect size provides the answer to this. The interpretation of the effect size follows the explanations in Chapter 10, where we looked at the strength of the correlation of two variables. However, different analytical techniques require different effect size measures, implying that we have to use different benchmarks. To help us with the interpretation, we can use the **effectsize** package and their set of **interpret_***() functions (see also Indices of Effect Sizes¹). Sometimes, there are even more than one way of computing the effect size. For example, we can choose between the classic Wilcoxon R or the rank-biserial correlation coefficient for the Mann-Whitney test. In practice, you have to be explicit about how you computed the effect size. The differences between the two measures are often marginal and a matter of taste (or should I say: Your reviewers' taste). Throughout this chapter, I will rely on the effect sizes most commonly found in Social Sciences publications. However, feel free to explore other indices as well, especially those offered in the **effectsize** package.

```
# After parametric test
(d <-
  comp |>
  cohens_d(freedom_of_choice ~ gender,
            var.equal = TRUE,
            ci = TRUE))

# A tibble: 1 x 9
.y.      group1 group2 effsize   n1   n2 conf.low conf.high magnitude
* <chr>    <chr>  <chr>  <dbl> <int> <int>  <dbl>    <dbl>    <dbl> <ord>
```

¹<https://easystats.github.io/effectsize/articles/interpret.html>

```
1 freedom_of_cho~ female male   -0.275  579  621   -0.39   -0.16 small

effectsize::interpret_cohens_d(d$effsize)

Cohen's d
"small"
(Rules: cohen1988)

# After non-parametric test
(wr <-
  comp |>
  wilcox_effsize(freedom_of_choice ~ gender,
                  ci = TRUE))

# A tibble: 1 × 9
.y.      group1 group2 effsize   n1   n2 conf.low conf.high magnitude
* <chr>    <chr>  <chr>   <dbl> <int> <int>  <dbl>    <dbl> <ord>
1 freedom_of_cho~ female male     0.145  579   621     0.09     0.2 small

effectsize::interpret_r(wr$effsize)

Effect size (r)
"small"
(Rules: funder2019)
```

Looking at our test results, the `female` Egyptians perceive `freedom_of_choice` differently from their `male` counterparts. This is in line with our boxplots. However, the effect sizes tend to be small, which means the differences between the two groups is marginal. Similar to correlations, group comparisons need to be analysed in two stages, answering two questions:

1. Is the difference between groups significant?
2. If it is significant, is the difference small, medium or large?

Combining both analytical steps gives us a comprehensive answer to our research question and enables us to derive meaningful conclusions. This applies to all group comparisons covered in this book.

11.2.2 Two paired groups

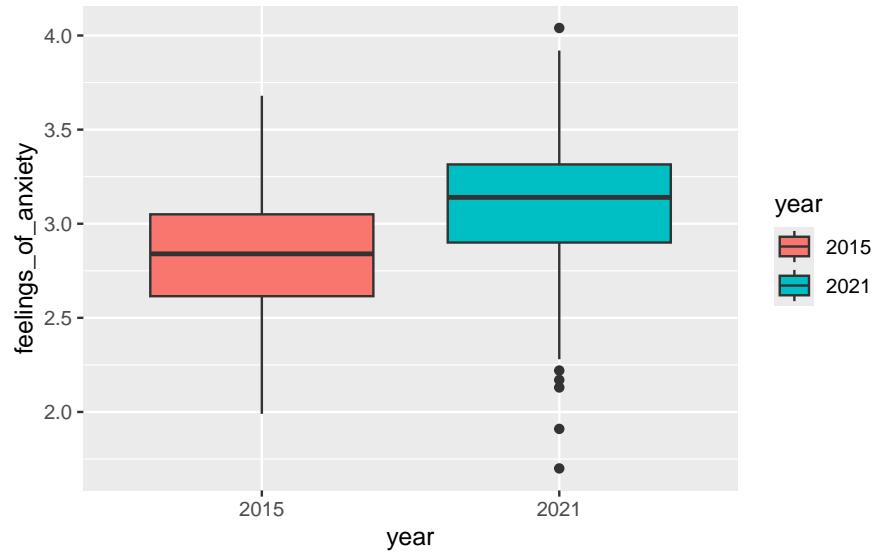
Sometimes, we are not interested in the difference between subjects, but within them, i.e., we want to know whether the same person provides similar or different responses at two different times. Thus, it becomes evident that observations need to be somehow linked to each other. Paired groups are frequently

used in longitudinal and experimental studies (e.g., pre-test vs post-test). For example, we might be interested to know whether people in England experienced different levels of anxiety over time. By analysing data provided in `hie_15_21`, we can examine changes in anxiety levels within the same population between 2015 and 2021 and find an answer to our question.

Let us begin our investigative journey by creating boxplots to compare the levels of anxiety from 2015 with those from 2021. However, in a first step, we need to convert the variable `year`, which is a numeric value in this dataset, into a factor. This will allow us to treat each year as a grouping variables for the entirety of our analysis.

```
hie_comp <-
  hie_15_21 |>
  # Convert numeric data into categorical data
  mutate(year = as_factor(year))

hie_comp |>
  # Create boxplots
  ggplot(aes(x = year,
             y = feelings_of_anxiety,
             fill = year)) +
  geom_boxplot()
```



Considering this plot, we can ascertain that there is a slight increase in `feelings_of_anxiety` which is concerning. However, it is tough to say whether

these differences are large, significant and, therefore, important. Instead we likely have to find a way to assess this statistically by crunching the numbers.

Table 11.2 summarises which tests and functions need to be performed when our data is parametric or non-parametric. In both cases, the functions are the same as those of the unpaired group comparisons, but we need to add the attribute `paired = TRUE`. Still, the interpretations between the unpaired and paired tests remain the same. Also, be aware that some tests have changed in name, e.g. the Mann-Whitney U test has become the Wilcoxon Signed Rank Test. Even though we use the same functions as before, by changing the attribute `paired` to `TRUE`, we also change the computational technique to obtain the results. Thus, remember that the same function can perform different computations which are not comparable. While this practice is handy, because it is easier to remember a function, it can also easily lead to mistakes if the wrong parameters were set.

Table 11.2: Comparing two paired groups

Assumption Test	Function	Effect size	Function
Parametric T-Test	<code>t_test(paired = TRUE)</code>	Cohen's d	<code>cohens_d()</code>
Non-parametric Wilcoxon Signed Rank Test	<code>wilcox_test(paired = TRUE)</code>	Wilcoxon r	<code>wilcox_effsize()</code>

Let's apply these functions to find out whether the differences we can see in our plot matter.

```
# Paired T-Test
hie_comp |>
  t_test(feelings_of_anxiety ~ year,
         paired = TRUE,
         var.equal = TRUE,
         detailed = TRUE) |>
  glimpse()
```

```
Rows: 1
Columns: 13
$ estimate    <dbl> -0.2539739
$ .y.        <chr> "feelings_of_anxiety"
$ group1     <chr> "2015"
$ group2     <chr> "2021"
$ n1          <int> 307
$ n2          <int> 307
```

```
$ statistic <dbl> -10.6258
$ p          <dbl> 1.18e-22
$ df         <dbl> 306
$ conf.low   <dbl> -0.3010063
$ conf.high  <dbl> -0.2069416
$ method     <chr> "T-test"
$ alternative <chr> "two.sided"

# Wilcoxon Signed Rank Test
hie_comp |>
  wilcox_test(feelings_of_anxiety ~ year,
               paired = TRUE) |>
  glimpse()
```

```
Rows: 1
Columns: 7
$ .y.      <chr> "feelings_of_anxiety"
$ group1   <chr> "2015"
$ group2   <chr> "2021"
$ n1       <int> 307
$ n2       <int> 307
$ statistic <dbl> 8151
$ p         <dbl> 1.9e-22
```

Irrespective of whether we use a parametric or non-parametric test, the results ($p < 0.05$) clearly point towards a significant change in levels of anxiety experienced in the England between 2015 and 2021: People seem to be more anxious than they used to be. Therefore, our next question has to be: How big are these differences, and are they important? In other words, we need to find out how large the effect size is.

```
## After T-Test
d <- cohens_d(feelings_of_anxiety ~ year,
               data = hie_15_21,
               paired = TRUE,
               var.equal = TRUE)

glimpse(d)
```

```
Rows: 1
Columns: 7
$ .y.      <chr> "feelings_of_anxiety"
$ group1   <chr> "2015"
$ group2   <chr> "2021"
$ effsize   <dbl> -0.6064463
```

```
$ n1      <int> 307
$ n2      <int> 307
$ magnitude <ord> moderate

effectsize::interpret_cohens_d(d$effsize)

Cohen's d
"medium"
(Rules: cohen1988)

# After Wilcoxon Signed Rank Test
wr <- 
  hie_15_21 |>
  wilcox_effsize(feelings_of_anxiety ~ year,
                  paired = TRUE, ci = TRUE)

glimpse(wr)

Rows: 1
Columns: 9
$ .y.      <chr> "feelings_of_anxiety"
$ group1   <chr> "2015"
$ group2   <chr> "2021"
$ effsize   <dbl> 0.5574338
$ n1       <int> 307
$ n2       <int> 307
$ conf.low <dbl> 0.47
$ conf.high <dbl> 0.64
$ magnitude <ord> large

effectsize::interpret_r(wr$effsize, rules = "cohen1988")
```

```
Effect size (r)
"large"
(Rules: cohen1988)
```

The results further confirm that `feelings_of_anxiety` have gone up substantially within six years, reporting either a `moderate` effect for the parametric test and even a `large` effect for the non-parametric equivalent. While both tests point in the right direction, it is noteworthy that our interpretations might slightly differ, given the interpretation of effect sizes. This highlights the importance of determining the appropriate method before conducting any analysis. Like many other statistical software programs, *R* can perform various analyses, but that does not necessarily mean it is the correct approach.

While we completed all analytical steps diligently, one might still be puzzled by this outcome. What exactly caused an increase in anxiety in England? Unfortunately, the numbers would not provide any insights into this question. Instead, considering the political changes and the COVID-pandemic during this time might offer some possible explanations.

11.3 Comparing more than two groups

Often we find ourselves in situations where comparing two groups is not enough. Instead, we might be faced with three or more groups reasonably quickly. For example, the `wvs_nona` dataset allows us to look at 48 different countries, all of which we could compare very effectively with just a few lines of code. In the following chapters, we look at how we can perform the same type of analysis as before, but with multiple unpaired and paired groups using *R*. Similarly to the two-samples group comparison, we cover the parametric and non-parametric approaches.

11.3.1 Multiple unpaired groups

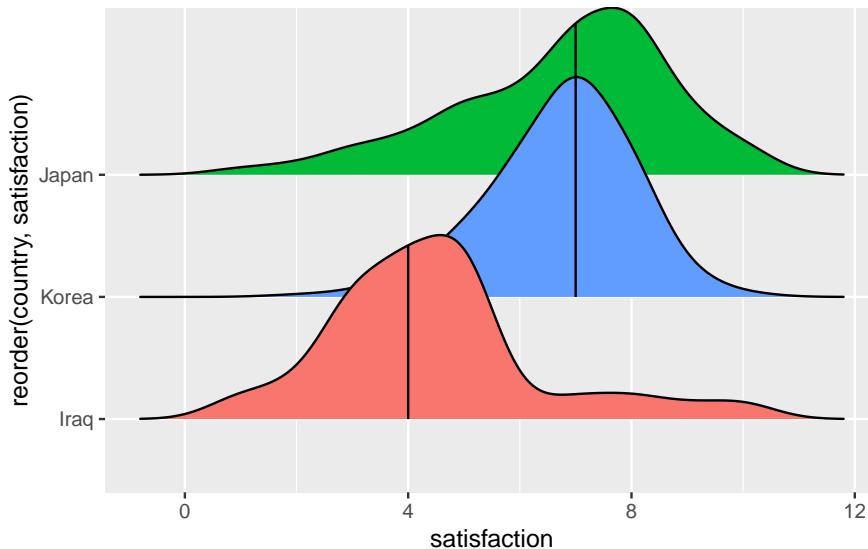
Have you ever been wondering whether people in different countries are equally satisfied with their lives? You might have a rough guess that it is not the case because the social, economic and political environment might play an important role. For example, if you live in a country affected by social conflicts, one's life satisfaction might be drastically lower. In the following, we take a look at three countries Iraq, Japan and Korea. I chose these countries out of personal interest and because they nicely demonstrate the purpose of the chapter, i.e. finding out whether there are differences in the perception of satisfaction across three countries. At any time, feel free to remove the `filter()` function to gain the results of all countries in the dataset, but prepare for slightly longer computation times. We first create the dataset, which only contains the three desired countries.

```
mcomp <-
  wvs_nona |>
  filter(country == "Iraq" |
         country == "Japan" |
         country == "Korea")
```

Similar to before, we can use the `ggridges` package to draw density plots for each group. This has the added benefit that we can compare the distribution of data for each group and see whether the assumption of normality is likely

met or not. On the other hand, we lose the option to identify any outliers quickly. You win some, and you lose some.

```
mcomp |>
  group_by(country) |>
  ggplot(aes(x = satisfaction,
              y = reorder(country, satisfaction),
              fill = country)) +
  ggridges::stat_density_ridges(bandwidth = 0.6,
                                 quantile_lines = TRUE,    # adds median indicator
                                 quantiles = (0.5)) +
  # Remove legend
  theme(legend.position = "none")
```



The plot shows us that Japan and Korea appear to be very similar, if not identical (based on the median), but Iraq appears to be different from the other two groups. When performing a multiple group comparison, we can follow similar steps as before with two groups, i.e.

- perform the comparison,
- determine the effect size, and
- interpret the effect size.

Table 11.3 summarises which test needs to be chosen to compare multiple unpaired groups and their corresponding effect size measures.

Table 11.3: Comparing multiple unpaired groups (effect size functions from package `effectsize`)

Assumption	Test	Function for test	Effect size	Function for effect size ²
Parametric	ANOVA	<ul style="list-style-type: none"> • <code>anova_test</code> (assumes equal variances) • <code>oneway.test(var.equal = TRUE/FALSE)</code> 	Eta squared	<code>eta_squared()</code>
Non-parametric	Kruskall-Wallis test	<code>kruskal_test()</code>	Epsilon squared (rank)	<code>rank_epsilon_squared()</code>

Let's begin by conducting the group comparison. As you will notice, `rstatix` currently does not support a parametric test where `var.equal = FALSE`. Therefore we need to fall back to the underlying function `oneway.test(var.equal = FALSE)`

```
# ANOVA
## equal variances assumed
mcomp |>
  anova_test(satisfaction ~ country,
              detailed = TRUE)
```

ANOVA Table (type II tests)

Effect	SSn	SSd	DFn	DFd	F	p	p<.05	ges
1 country	4258.329	11314.68	2	3795	714.133	5.74e-264	*	0.273

```
## Equal variances not assumed
(oneway_test <- oneway.test(satisfaction ~ country,
                           data = mcomp,
                           var.equal = FALSE))
```

```
One-way analysis of means (not assuming equal variances)

data: satisfaction and country
F = 663.17, num df = 2.0, denom df = 2422.8, p-value < 2.2e-16
```

²These functions are taken from the `effectsize` package.

```
# Kruskall-Wallis test
mcomp |> kruskal_test(satisfaction ~ country)

# A tibble: 1 × 6
  .y.          n statistic   df      p method
  <chr>     <int>    <dbl> <int>    <dbl> <chr>
1 satisfaction 3798     1064.     2 1.11e-231 Kruskal-Wallis
```

While `anova_test()` does provide the effect size automatically, i.e. generalised eta squared (ges), this is not the case for the other two approaches. Therefore, we have to use the `effectsize` package to help us out. Packages often can get you a long way and make your life easier, but it is good to know alternatives if a single package does not give you what you need.

```
# After ANOVA with var.equal = FALSE
effectsize::eta_squared(oneway_test)
```

```
# Effect Size for ANOVA

Eta2 |      95% CI
-----
0.35 | [0.33, 1.00]

- One-sided CIs: upper bound fixed at [1.00].
```

```
# effect size rank epsilon squared
effectsize::rank_epsilon_squared(mcomp$satisfaction ~ mcomp$gender)
```

```
Epsilon2 (rank) |      95% CI
-----
2.12e-03 | [0.00, 1.00]

- One-sided CIs: upper bound fixed at [1.00].
```

The results show that there is a significant and large difference between these groups. You might argue that this is not quite true. Considering our plot, we know that Japan and Korea do not look as if they are significantly different. Multiple group comparisons only consider differences across all three groups. Therefore, if one group differs from the other groups, the test will turn significant and even provide a large enough effect size to consider it essential. However, these tests do not provide information on which differences between groups are significant. To gain more clarification about this, we need to incorporate another step called '*post-hoc tests*'. These tests compare two groups at a time, which is why they are also known as '*pairwise comparisons*'. Com-

pared to regular two-sample tests, these perform corrections of the p-values for multiple testing, which is necessary. However, there are many different ‘*post-hoc*’ tests you can choose. Field (2013) (p.459) nicely outlines the different scenarios and provides recommendations to navigate this slightly complex field of *post-hoc* tests. Table 11.4 provides an overview of his suggestions.

Table 11.4: Different post-hoc tests for different scenarios (parametric)

Equal sample size	Equal variance	Post-hoc tests	Functions in R
YES	YES	<ul style="list-style-type: none"> • <code>mutoss::regwq()</code>³ REGWQ <code>statix::tukey_hsd()</code> • <code>pairwise.t.test(p.adjust.method = Tukey, "bonferroni")</code> • 	
NO (slightly different)	YES	<ul style="list-style-type: none"> Bonferroni Gabriel 	
YES	YES	<ul style="list-style-type: none"> • Not available in R and should not be confused Hochberg with <code>pairwise.t.test(p.adjust.method = GT2, "hochberg")</code>, which is based on Hochberg (1988). The GT2, however, is based on Hochberg (1974). 	
NO (not ideal for small samples)	NO	<ul style="list-style-type: none"> • <code>rstatix::games_howell_test()</code> Games-Howell 	

You might be surprised to see that there are also *post-hoc* tests for parametric group comparisons when equal variances are not assumed. Would we not have to use a non-parametric test for our group comparison instead? Well, empirical studies have demonstrated that ANOVAs tend to produce robust results, even if the assumption of normality (e.g. Blanca Mena et al. 2017) is not given, or there is some degree of heterogeneity of variance between groups (Tomarken and Serlin 1986). In other words, there can be some leniency (or flexibility?) when it comes to the violation of parametric assumptions. If you want to reside on the save side, you should ensure you know your data and its properties. If in doubt, non-parametric tests are also available.

If we want to follow up the Kruskal-Wallis test, i.e. the non-parametric equivalent to the one-way ANOVA, we can make use of two *post-hoc* tests:

³In order to use this package, it is necessary to install a series of other packages found on bioconductor.org⁴

- Dunn Test: `rstatix::dunn_test()` (Dinno 2015)
- Pairwise comparison with Bonferroni (and other) correction: `pairwise.wilcox.test()`.

Below are some examples of how you would use these functions in your project. However, be aware that some of the *post-hoc tests* are not well implemented yet in *R*. Here, I show the most important ones that likely serve you in 95% of the cases.

```
# POST-HOC TEST FOR PARAMETRIC DATA
# Bonferroni
pairwise.t.test(mcomp$satisfaction,
                 mcomp$country,
                 p.adjust.method = "bonferroni")
```

```
Pairwise comparisons using t tests with pooled SD

data: mcomp$satisfaction and mcomp$country
```

```
Iraq    Japan
Japan <2e-16 -
Korea <2e-16 1
```

```
P value adjustment method: bonferroni
```

```
# Tukey
mcomp |> tukey_hsd(satisfaction ~ country)
```

```
# A tibble: 3 x 9
  term   group1 group2 null.value estimate conf.low conf.high      p.adj
* <chr>  <chr>  <chr>       <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
1 country Iraq    Japan        0     2.29     2.13     2.45  0.0000000141
2 country Iraq    Korea       0     2.26     2.10     2.43  0.0000000141
3 country Japan   Korea      -0.0299   -0.189    0.129  0.898
# i 1 more variable: p.adj.signif <chr>
```

```
# Games-Howell
mcomp |> games_howell_test(satisfaction ~ country)
```

```
# A tibble: 3 x 8
  .y.      group1 group2 estimate conf.low conf.high  p.adj p.adj.signif
* <chr>    <chr>  <chr>    <dbl>    <dbl>    <dbl>    <dbl>    <chr>
1 satisfaction Iraq    Japan     2.29     2.11     2.47    0      ****
```

```

2 satisfaction Iraq    Korea     2.26      2.11      2.42  5.71e-11 ****
3 satisfaction Japan   Korea    -0.0299   -0.178     0.118 8.84e- 1 ns

# POST-HOC TEST FOR NON-PARAMETRIC DATA
mcomp |> dunn_test(satisfaction ~ country)

# A tibble: 3 x 9
.y. group1 group2 n1 n2 statistic      p   p.adj p.adj.signif
<chr> <chr> <chr> <int> <int> <dbl> <dbl> <dbl> <chr>
1 satisfac~ Iraq    Japan    1200 1353    29.2 4.16e-187 1.25e-186 ****
2 satisfac~ Iraq    Korea    1200 1245    27.6 8.30e-168 1.66e-167 ****
3 satisfac~ Japan   Korea    1353 1245   -1.02 3.10e- 1 3.10e- 1 ns

# or
pairwise.wilcox.test(mcomp$satisfaction,
                      mcomp$country,
                      p.adjust.method = "holm")

```

Pairwise comparisons using Wilcoxon rank sum test with continuity correction

```

data: mcomp$satisfaction and mcomp$country

Iraq    Japan
Japan < 2e-16 -
Korea < 2e-16 0.00093

P value adjustment method: holm

```

As we can see, no matter which function we use, the interpretation of the results remain the same on this occasion. This is, of course reassuring to know, because it implies that the method chosen does not change the outcome.

11.3.2 Multiple paired groups

When comparing multiple paired groups, matters become slightly more complicated. On the one hand, our groups are not really groups anymore because our data refer to the same group of people, usually over an extended period of time. In experimental studies, this can also refer to different '*treatments*' or '*conditions*'. This is similar to comparing two paired groups. On the other hand, we also have to deal with yet another assumption: *Sphericity*.

11.3.2.1 Testing the assumption of sphericity

Sphericity assumes that the variance of covariate pairs (i.e. any combination of the groups/treatments) are roughly equal. This might sound familiar to the assumption of *homogeneity of variance* in between-subject ANOVAs, only that we look at differences between pairs and not between two different participant groups. Thus, it seems less surprising that the parametric test to compare multiple paired groups is also called '*repeated measures ANOVA*'.

To illustrate the concept of *sphericity*, let's look at an example. Assume we conduct a longitudinal study that involves five university students who started their studies in a foreign country. We ask them to complete a survey that tests their level of integration into the local community at three points in time: month 1 (`m1`), month 4 (`m4`) and month 8 (`m8`). This gives us the following dataset:

```
Warning: There was 1 warning in `mutate()`.  
i In argument: `across(where(is.double), round, 0)`.  
Caused by warning:  
! The `...` argument of `across()` is deprecated as of dplyr 1.1.0.  
Supply arguments directly to `.fns` through an anonymous function instead.  
  
# Previously  
across(a:b, mean, na.rm = TRUE)  
  
# Now  
across(a:b, \((x) mean(x, na.rm = TRUE))
```

acculturation

```
# A tibble: 5 x 4  
  name      m1     m4     m8  
  <chr>    <dbl>  <dbl>  <dbl>  
1 Waylene     2      3      5  
2 Nicole      1      3      6  
3 Mikayla     2      3      5  
4 Valeria     1      3      5  
5 Giavanni    1      3      5
```

If each month measured a different group of participants, we would compare the differences between each month regarding homogeneity. However, since we look at paired data, we need to consider the differences in pairs of measures. In this study, the following combinations are possible:

- `m1` and `m4`
- `m4` and `m8`

- `m1` and `m8`

Let's add the differences between measures for each participant based on these pairs and compute the variances across these values using the function `var()`.

```
# Compute the differences across all three pairs of measurements
differences <-
  acculturation |>
  mutate(m1_m4 = m1 - m4,
         m4_m8 = m4 - m8,
         m1_m8 = m1 - m8)

differences

# A tibble: 5 x 7
  name      m1     m4     m8 m1_m4 m4_m8 m1_m8
  <chr>   <dbl>  <dbl>  <dbl> <dbl> <dbl> <dbl>
1 Waylene    2      3      5    -1    -2    -3
2 Nicole     1      3      6    -2    -3    -5
3 Mikayla    2      3      5    -1    -2    -3
4 Valeria    1      3      5    -2    -2    -4
5 Giovanni   1      3      5    -2    -2    -4

# Compute the variance for each pair
differences |>
  summarise(m1_m4_var = var(m1_m4),
            m4_m8_var = var(m4_m8),
            m1_m8_var = var(m1_m8))

# A tibble: 1 x 3
  m1_m4_var m4_m8_var m1_m8_var
  <dbl>      <dbl>      <dbl>
1     0.3       0.2       0.7
```

We can tell that the differences across groups are relatively small when comparing `m1_m4_var` and `m4_m8_var`. However, the value for `m1_m8_var` appears bigger. So, how can we be sure whether the assumption of sphericity is violated or not? Similar to many of the other assumptions we covered, a significance test provides the answer to this question. For multiple paired groups, we use *Mauchly's Test of Sphericity*. If the test is significant, the variances across groups are not equal. In other words, a significant Mauchly test implies a violation of sphericity. The `rstatix` package includes Mauchly's Test of Sphericity in its `anova_test()`. Thus, we get both with one function. However, our data in this example is not tidy (remember the definition from Section 7.2?), because there are multiple observation per row for each individual and the same

variable. Instead, we need to ensure that each observation for a given variable has its own row. In short, we first need to convert it into a tidy dataset using the function `pivot_longer()`. Feel free to ignore this part for now, because we will cover pivoting datasets in greater detail in Section 11.4.

```
# Convert into tidy data
acc_long <- 
  acculturation |>
  pivot_longer(cols = c(m1, m4, m8),
               names_to = "month",
               values_to = "integration")

acc_long

# A tibble: 15 x 3
  name    month integration
  <chr>   <chr>      <dbl>
1 Waylene m1            2
2 Waylene m4            3
3 Waylene m8            5
4 Nicole  m1            1
5 Nicole  m4            3
6 Nicole  m8            6
7 Mikayla m1            2
8 Mikayla m4            3
9 Mikayla m8            5
10 Valeria m1           1
11 Valeria m4           3
12 Valeria m8           5
13 Giavanni m1          1
14 Giavanni m4          3
15 Giavanni m8          5

# Perform repeated-measures ANOVA
# plus Mauchly's Test of Sphericity
acc_long |>
  rstatix::anova_test(dv = integration,
                      wid = name,
                      within = month,
                      detailed = TRUE)
```

ANOVA Table (type III tests)

\$ANOVA

```

Effect DFn DFd   SSn SSD      F          p p<.05    ges
1 (Intercept)  1   4 153.6 0.4 1536 2.53e-06     * 0.987
2       month   2   8 36.4 1.6    91 3.14e-06     * 0.948

$`Mauchly's Test for Sphericity`
Effect      W      p p<.05
1 month 0.417 0.269

$`Sphericity Corrections`
Effect GGe   DF[GG]  p[GG] p[GG]<.05 HFe   DF[HF]  p[HF] p[HF]<.05
1 month 0.632 1.26, 5.05 0.000162      * 0.788 1.58, 6.31 3e-05     *

```

Not only do the results show that the differences across measures are significant, but that our Mauchly's Test of Sphericity is not significant. Good news on all fronts.

The output from `anova_test()` also provides information about `Sphericity Corrections`. We need to consider these scores if the sphericity assumption is violated, i.e. Mauchly's Test of Sphericity is significant. We commonly use two methods to correct the degrees of freedom in our analysis which will change the p-value of our tests: *Greenhouse-Geisser correction* (Greenhouse and Geisser 1959) and *Huynh-Field correction* (Huynh and Feldt 1976). Field (2013) explains that if the Greenhouse-Geisser estimate (i.e. `GGe`) is greater than `0.74`, it is advisable to use the Huynh-Field correction instead. For each correction, the `anova_test()` provides corrected p-values. In our example, we do not have to worry about corrections, though. Still, it is very convenient that this function delivers the corrected results as well.

11.3.2.2 Visualising and computing multiple paired group comparisons

So far, we blindly assumed that our data are parametric. If our assumptions for parametric tests are violated, we can draw on a non-parametric equivalent called `Friedman Test`. Table 11.5 summarises the parametric and non-parametric tests as well as their respective effect sizes.

Table 11.5: Comparing multiple paired groups (effect size functions from package `effectsize`)

Assumption	Test	Function	Effect size	
			Eta squared	Function
Parametric	Repeated measures ANOVA	<code>anova_test()</code>	Eta squared	<code>eta_squared()</code>
Non-parametric	Friedman Test	<code>friedman_test()</code>	Kendall's R	<code>kendalls_w()</code>

Instead of using a fictional example as we did in the previous chapter, let's draw on real observations using the dataset `wvs_waves`. This dataset contains similar data to `wvs` but offers multiple measures as indicated by the variable `wave` for several countries. For example, we might be interested to know whether `satisfaction` changed over the years. This time, our unit of analysis is not individuals, but countries. Therefore, we compare the same countries (not individuals) over time.

Let's start by visualising the data across all time periods using `geom_boxplot()`. I also added the mean for each wave since this is what we will eventually compare in our repeated-measures ANOVA.

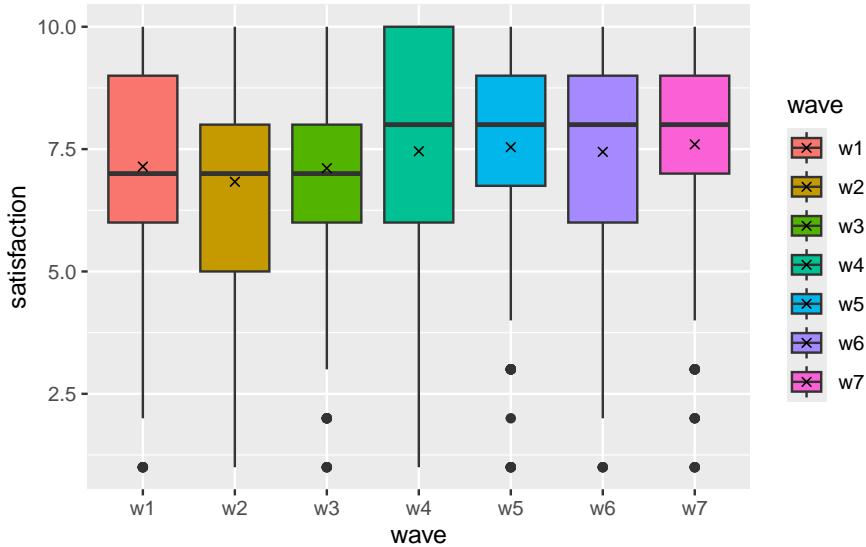
```
# Compute the mean for each wave
wave_means <-
  wvs_waves |>
  group_by(wave) |>
  summarise(w_mean = mean(satisfaction))

wave_means

# A tibble: 7 x 2
  wave   w_mean
  <fct> <dbl>
1 w1     7.14
2 w2     6.83
3 w3     7.11
4 w4     7.46
5 w5     7.54
6 w6     7.44
7 w7     7.60

# Plot the boxplots
wvs_waves |>
  ggplot(aes(x = wave,
             y = satisfaction,
             fill = wave)) +
  geom_boxplot() +

  # Add the means
  geom_point(data = wave_means,
             aes(x = wave,
                 y = w_mean),
             shape = 4)
```



In total, we plotted seven boxplots that represent each wave of data collection. We can tell that `satisfaction` with life has improved slightly, especially from wave 3 (w3) to wave 4 (w4).

Next, we would have to check the assumptions for parametric tests. Since we have covered this in Chapter 9, we only consider Mauchly's Test of Sphericity by running `anova_test()`.

```
# Compute repeated measures ANOVA and
# Mauchly's Test of Sphericity
wvs_waves |>
  rstatix::anova_test(dv = satisfaction,
                      wid = id,
                      within = wave)
```

ANOVA Table (type III tests)

```
$ANOVA
Effect  DFn   DFd      F      p p<.05    ges
1  wave     6 1794  5.982 3.33e-06      * 0.015

$`Mauchly's Test for Sphericity`
Effect      W      p p<.05
1  wave 0.903 0.067

$`Sphericity Corrections`
Effect  GGe      DF[GG]    p[GG] p[GG]<.05    HFe      DF[HF]    p[HF]
```

```
1  wave 0.968 5.81, 1736.38 4.57e-06      * 0.989 5.94, 1774.66 3.7e-06
   p[HF]<.05
1      *
```

We first look at the sphericity assumption which we did not violate, i.e. $p > 0.05$. Thus, using the ANOVA test is appropriate. Next, we inspect the ANOVA test results and find that the differences are significant as well, i.e. $p < 0.05$. Still, the effect size `ges` is small.

From our plot, we know that some of the waves are fairly similar, and we need to determine which pair of waves are significantly different from each other. Therefore, we need a post-hoc test that allows us to perform pairwise comparisons. By now, this should sound familiar (see also Table 11.4).

Since we assume our data is parametric and the groups are equally large for each wave ($n = 300$), we can use T-Tests with a Bonferroni correction.

```
pairwise.t.test(wvs_waves$satisfaction,
                 wvs_waves$wave,
                 p.adjust.method = "bonferroni",
                 paired = TRUE)
```

Pairwise comparisons using paired t tests

```
data: wvs_waves$satisfaction and wvs_waves$wave

      w1     w2     w3     w4     w5     w6
w2 1.00000 -      -      -      -      -
w3 1.00000 1.00000 -      -      -      -
w4 1.00000 0.01347 0.78355 -      -      -
w5 0.30433 0.00033 0.11294 1.00000 -      -
w6 1.00000 0.00547 0.68163 1.00000 1.00000 -
w7 0.05219 0.00023 0.03830 1.00000 1.00000 1.00000
```

P value adjustment method: bonferroni

Based on these insights, we find that mainly `w2` shows significant differences with other waves. This is not a coincidence because `w2` has the lowest mean of all waves. Similarly, `w7`, which reports the highest mean for satisfaction, also reports several significant differences with other groups.

Given the above, we can confirm that our people's `satisfaction` with life in each country has changed positively, but the change is minimal (statistically).

To finish this chapter, I want to share the non-parametric version of this test: the Friedman test and its function `friedman_test()` from the `rstatix` package.

```
# NON-PARAMETRIC COMPARISON
wvs_waves |> rstatix::friedman_test(satisfaction ~ wave | id)

# A tibble: 1 × 6
  .y.      n statistic   df      p method
  <chr>  <int>     <dbl> <dbl>    <dbl> <chr>
1 satisfaction  300      43.4     6 0.0000000966 Friedman test

# Effect size
(kw <- effectsize::kendalls_w(satisfaction ~ wave | id,
                                 data = wvs_waves))

Kendall's W |      95% CI
-----
0.02 | [0.02, 1.00]

- One-sided CIs: upper bound fixed at [1.00].
```

```
effectsize::interpret_kendalls_w(kw$Kendalls_W)
```

[1] "slight agreement"
(Rules: landis1977)

The non-parametric test confirms the parametric test from before. However, the interpretation of the effect size Kendall's W requires some explanation. First, the effect size for Kendall's W can range from 0 (small effect size) to 1 (large effect size). Usually, this effect size is considered when comparing inter-rater agreement, for example, the level of agreement of a jury in ice skating during the Olympic games. Thus, the interpretation offered by the function `interpret_kendalls_w()` follows this logic. As such, it is worth considering a slightly amended interpretation of Kendall's W if our data is not based on ratings. This is shown in Table 11.6 based on Landis and Koch (1977) (p. 165). Thus, we could report that the effect size of the significant differences is ‘very small’.

Table 11.6: Interpretation benchmarks for the effect size Kendall's W

Kendall's W	Interpretation (Landis and Koch 1977)	Alternative interpretation
0.00-0.02	Slight agreement	Very small
0.21-0.40	Fair agreement	Small
0.41-0.60	Moderate agreement	Moderate
0.61-0.80	Substantial agreement	Large

Kendall's W	Interpretation (Landis and Koch 1977)	Alternative interpretation
0.81-1.00	Almost perfect agreement	Very large

11.4 Comparing groups based on factors: Contingency tables

So far, our dependent variable was always a numeric one. However, what if both independent and dependent variables are categorical? In such cases, we can only count the number of occurrences for each combination of the categories.

For example, we might recode our previous dependent variable `satisfaction` into a dichotomous one, i.e. participants are either happy or not. This is also known as a binary/logical variable (see Section 7.4). Since the original scale ranges from 1-10, we assume that participants who scored higher than 5 are `satisfied` with their lives, while those who scored lower are categorised as `unsatisfied`.

```
# Create a dichotomous/binary variable for satisfaction
wvs_nona <-
  wvs_nona |>
    mutate(satisfaction_bin = as_factor(ifelse(satisfaction > 5,
                                                 "satisfied",
                                                 "unsatisfied")))
  )
```

Your first intuition is likely to ask: Are there more `satisfied` or `unsatisfied` people in my sample?

```
wvs_nona |> count(satisfaction_bin)
```

```
# A tibble: 2 x 2
  satisfaction_bin     n
  <fct>             <int>
1 unsatisfied        3087
2 satisfied          5477
```

The results reveal that considerably more people are `satisfied` with their life than there are `unsatisfied` people. In the spirit of group comparisons, we might wonder whether gender differences might exist among the `satisfied`

and `unsatisfied` group of people. Thus, we want to split the `satisfied` and `unsatisfied` responses into `male` and `female` groups. We can do this by adding a second argument to the function `count()`.

```
wvs_nona |> count(satisfaction_bin, gender)
```

```
# A tibble: 4 x 3
  satisfaction_bin gender     n
  <fct>           <fct>   <int>
1 unsatisfied     female  1554
2 unsatisfied     male    1533
3 satisfied       female  2794
4 satisfied       male    2683
```

This `tibble` reveals that there are more `female` participants who are satisfied than `male` ones. However, the same is true for the category `unsatisfied`. Using absolute values is not very meaningful when the sample sizes of each group are not equal. Thus, it is better to use the relative frequency instead and adding it as a new variable.

```
ct <- 
  wvs_nona |>
  count(satisfaction_bin, gender) |>
  group_by(gender) |>
  mutate(perc = round(n / sum(n), 3)) # compute the percentages
```

```
ct
```

```
# A tibble: 4 x 4
# Groups:   gender [2]
  satisfaction_bin gender     n  perc
  <fct>           <fct>   <int> <dbl>
1 unsatisfied     female  1554 0.357
2 unsatisfied     male    1533 0.364
3 satisfied       female  2794 0.643
4 satisfied       male    2683 0.636
```

The relative frequency (`perc`) reveals that `female` and `male` participants are equally satisfied and unsatisfied. In other words, we could argue that gender does not explain satisfaction with life because the proportion of `male` and `female` participants is almost identical.

A more common and compact way to show such dependencies between categorical variables is a contingency table. To convert our current table into a contingency table, we need to map the levels of `satisfaction_bin` as rows

(i.e. `satisfied` and `unsatisfied`), and for `gender`, we want each level represented as a column (i.e. `male` and `female`). This can be achieved with the function `pivot_wider()`. It turns our ‘long’ data frame into a ‘wide’ one. Therefore, we basically perform the opposite of what the function `pivot_longer()` did earlier. However, this means that our data is not ‘tidy’ anymore. Let’s take a look at the output first to understand better what we try to achieve.

```
ct |> pivot_wider(id_cols = satisfaction_bin,
                    names_from = gender,
                    values_from = perc)

# A tibble: 2 x 3
  satisfaction_bin female male
  <fct>          <dbl> <dbl>
1 unsatisfied     0.357 0.364
2 satisfied       0.643 0.636
```

The resulting `tibble` looks like a table as we know it from Excel. It is much more compact than the long format. However, what are the different arguments I passed to the function `pivot_wider()`? Here is a more detailed explanation of what we just did:

- `id_cols` refers to one or more columns that define the groups for each row. In our case, we wanted to group observations based on whether these reflect the state of `satisfied` or `unsatisfied`. Thus, each group is only represented once in this column. The underlying concept is comparable to using the function `group_by()` together with `summarise()`, which happens to be the same as using `count()`.
- `names_from` defines which variable should be translated into separate columns. In other words, we replace the column `gender` and create a new column for each level of the factor, i.e. `male` and `female`.
- `values_from` requires us to specify which variable holds the values that should be entered into our table. Since we want our percentages included in the final output, we use `perc`.

It is essential to note that we did not change any values but rearranged them. However, we lost one variable, i.e. `n`. Where has it gone? When using `pivot_wider()` we have to make sure we include all variables of interest. By default, the function will drop any other variables not mentioned. If we want to keep `n`, we can include it as another variable that is added to `values_from`.

```
ct |> pivot_wider(id_cols = satisfaction_bin,
                    names_from = gender,
                    values_from = c(perc, n))
```

```
# A tibble: 2 x 5
  satisfaction_bin perc_female perc_male n_female n_male
  <fct>           <dbl>      <dbl>     <int>    <int>
1 unsatisfied     0.357      0.364     1554    1533
2 satisfied       0.643      0.636     2794    2683
```

The table has become even wider because we have two more columns to show the absolute frequency per gender and `satisfaction_bin`.

As you already know, there are situations where we want to turn a ‘wide’ data frame into a ‘long’ one. We performed such a step earlier and there is an in-depth example provided in Section 13.2.1.5. Knowing how to pivot dataset is an essential data wrangling skill and something you should definitely practice to perfection.

Contingency tables are like plots: They provide an excellent overview of our data structure and relationships between variables. However, they provide no certainty about the strength of relationships. Therefore, we need to draw on a set of statistical tests to gain further insights.

Similar to previous group comparisons, we can distinguish between paired and unpaired groups. I will cover each comparison in turn and allude to plotting two or more categories in a `ggplot()`.

11.4.1 Unpaired groups of categorical variables

In the introduction to this chapter, we covered a classic example of an unpaired comparison of two groups (`male` and `female`) regarding another categorical variable, i.e. `satisfaction_bin`. The tables we produced offered detailed insights into the distribution of these categories. However, it is always helpful to have a visual representation. Plotting two categorical variables (i.e. factors) in `ggplot2` can be achieved in many different ways. Three commonly used options are shown in Figure 11.1).

```
# Plot with absolute frequencies (stacked)
absolute_stacked <-
  wvs_nona |>
  ggplot(aes(x = satisfaction_bin,
             fill = gender)) +
  geom_bar() +
  ggtitle("Stacked absolute frequency") +
  theme(legend.position = "none")

# Plot with absolute frequencies (grouped)
absolute_grouped <-
  wvs_nona |>
```

```

ggplot(aes(x = satisfaction_bin,
            fill = gender)) +
  geom_bar(position = "dodge") +
  ggtitle("Grouped absolute frequency") +
  theme(legend.position = "none")

# Plot with relative frequencies
relative <-
  wvs_nona |>
  ggplot(aes(x = satisfaction_bin,
            fill = gender)) +
  geom_bar(position = "fill") +
  ggtitle("Relative frequency") +
  theme(legend.position = "none")

# Plot all three plots with 'patchwork' package
absolute_stacked + absolute_grouped + relative + plot_spacer()

```

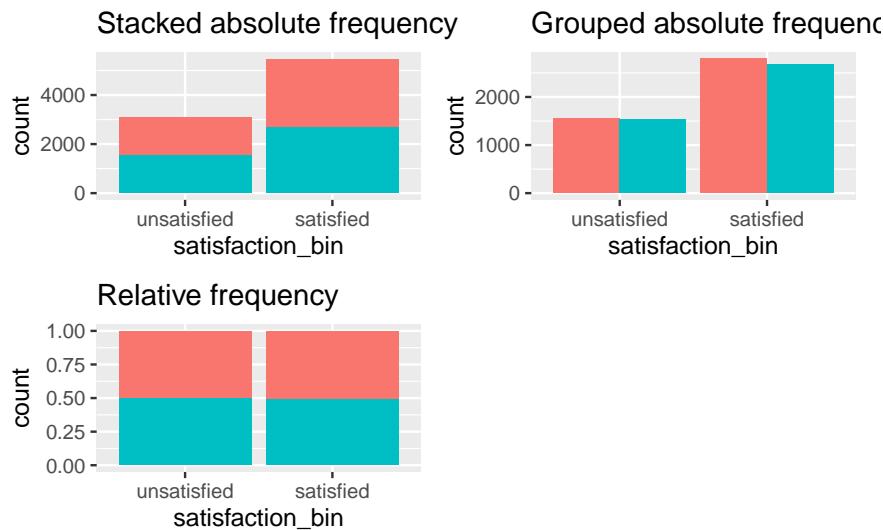


Figure 11.1: Three ways to plot frequencies.

A more elegant and compact way of visualising frequencies across two or more categories are *mosaic plots*. These visualise relative frequencies for both variables in one plot. For example, consider the following mosaic plot created with the package `ggridges` and the function `geom_mosaic()`.

```
library(ggmosaic)
wvs_nona |>
  ggplot() +
  geom_mosaic(aes(x = product(satisfaction_bin, gender),
                  fill = gender)) +
  theme_mosaic()
```

Warning: The `scale_name` argument of `continuous_scale()` is deprecated as of ggplot2 3.5.0.

Warning: The `trans` argument of `continuous_scale()` is deprecated as of ggplot2 3.5.0.
i Please use the `transform` argument instead.

Warning: `unite_()` was deprecated in tidyverse 1.2.0.

i Please use `unite()` instead.

i The deprecated feature was likely used in the ggmosaic package.

Please report the issue at <<https://github.com/haleyjeppson/ggmosaic>>.

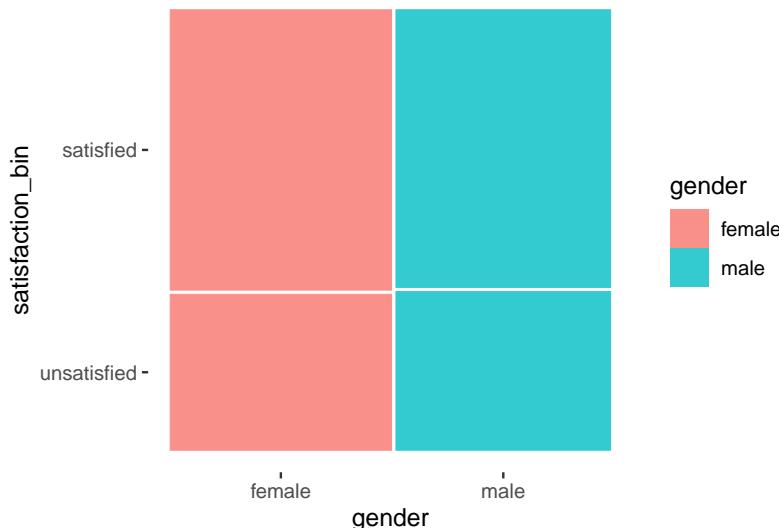
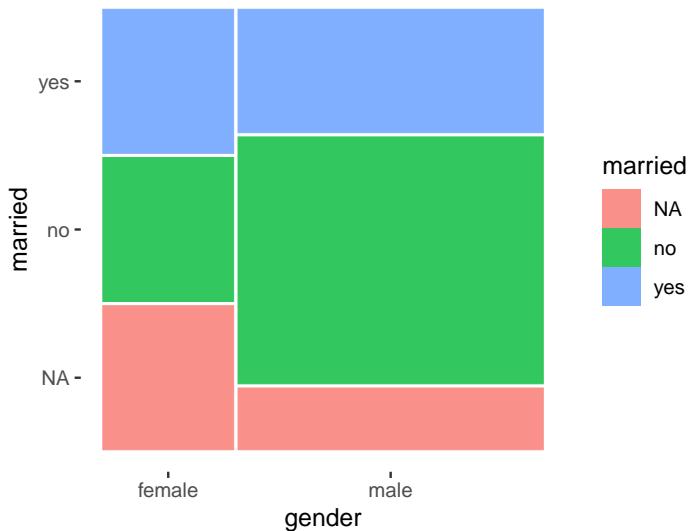


Figure 11.2: A mosaic plot which visualises the relationship of two categorical variables.

It might not be evident from our data, but frequencies of each variable determine the bars' height and width. In other words, the width of the bars (i.e. x-axis) is determined by the relative frequency of `female` and `male` participants in our sample. On the other hand, the height of each bar (i.e. the y-axis) is specified by the relative frequency of `satisfied` and `unsatisfied`. This re-

sults in a square block that is divided by the respective relative distributions. Let me share an example where it is more apparent what a mosaic plot aims to achieve.

```
data |>
  ggplot() +
  geom_mosaic(aes(x = product(married, gender),
                  fill = married)) +
  theme_mosaic()
```



In this example, we can tell that there were more `male` participants than females because the bar for `male` is much wider. Apart from that, we notice that female participants have more missing values `NA` for the variable `married`. However, more `female` participants reported that they are `married`, i.e. answered with `yes`. While mosaic plots make for impressive visualisations, we must be mindful that more complex visualisations are always more challenging to understand. Thus, it is wise to plan the usage of such plots carefully depending on your audience.

Throughout the remaining chapters, I will use the mosaic plot to illustrate distributions. The main difference to regular bar plots is the function `product()`, which allows us to define different variables of interest instead of using `x` and `y` in the `aes()` of `ggplot()`. It is this function that enables a mosaic visualisation. Apart from that, the same `ggplot2` syntax applies.

When it comes to the computational side of things, we have to distinguish whether our two variables create a 2-by-2 matrix, i.e. both variables only have

two levels. Depending on which scenario applies to our analysis, a different statistical test has to be performed. For some tests, a minimum frequency for each value in our contingency table (i.e. cells) also needs to be achieved. This is usually a matter of sample size and diversity in a sample. Table 11.7 provides an overview of the different scenarios and indicates the functions we have to use to conduct our analysis in *R*.

Table 11.7: Statistical tests to compare two unpaired categorical variables. Effect sizes are computed using the `effectsize` package.

MatrixCondition	Test	Function in R	Effect size
2x2 < 10 obs.	Fisher's Exact Test	<code>fisher.test()</code>	<code>phi()</code>
2x2 > 10 obs.	Chi-squared Test with Yate's Continuity Correction	<code>infer::chisq_test(correct = TRUE)</code> ⁵	<code>phi()</code>
n x > 5 obs. n (80% of cells)	Chi-squared Test	<code>infer::chisq_test()</code>	<code>cramers_v()</code>

To cut a long story short, we only need to be worried about 2x2 contingency tables where the values in some (or all) cells are lower than 10. In those cases, we need to rely on the *Fisher's Exact Test*. In all other cases, we can depend on the *Pearson Chi-squared Test* to do our bidding. The function `infer::chisq_test()` is based on the function `chisq.test()`, which automatically applies the required *Yate's Continuity Correction* if necessary.

For every test that involves two categorical variables, we have to perform three steps:

1. Compute a contingency table and check the conditions outlined in Table 11.7.
2. Conduct the appropriate statistical test.
3. If the test is significant, compute the effect size and interpret its value.

Considering our example from the beginning of the chapter, we are confronted with a 2x2 table that has the following distribution:

⁵`infer` is an R package which is part of `tidymodels`.

```
ct |> pivot_wider(id_cols = satisfaction_bin,
                    names_from = gender,
                    values_from = n)
```

```
# A tibble: 2 x 3
  satisfaction_bin female male
  <fct>          <int> <int>
1 unsatisfied     1554  1533
2 satisfied       2794  2683
```

We easily satisfy the requirement for a regular Chi-squared test with Yate's Continuity Correction because each row has at least a value of 10. However, for demonstration purposes, I will show how to compute both the Chi-squared test and the Fisher's Exact test for the same contingency table.

```
# Fisher's Exact Test
fisher.test(wvs_nona$satisfaction_bin, wvs_nona$gender) |>
  # Make output more readable
  broom::tidy() |>
  glimpse()
```

```
Rows: 1
Columns: 6
$ estimate    <dbl> 0.9734203
$ p.value     <dbl> 0.5584163
$ conf.low    <dbl> 0.8903427
$ conf.high   <dbl> 1.064261
$ method      <chr> "Fisher's Exact Test for Count Data"
$ alternative <chr> "two.sided"
```

```
## Effect size
(phi <- effectsize::phi(wvs_nona$satisfaction_bin, wvs_nona$gender))
```

```
Phi (adj.) | 95% CI
-----
0.00 | [0.00, 1.00]

- One-sided CIs: upper bound fixed at [1.00].
```

```
effectsize::interpret_r(phi$phi, rules = "cohen1988")
```

```
[1] "very small"
(Rules: cohen1988)
```

```
# Chi-squared test with Yate's continuity correction
wvs_nona |> infer::chisq_test(satisfaction_bin ~ gender,
                                correct = TRUE)

# A tibble: 1 × 3
  statistic chisq_df p_value
  <dbl>     <int>    <dbl>
1 0.332       1    0.565

## Effect size
(cv <- effectsize::cramers_v(wvs_nona$satisfaction_bin, wvs_nona$gender))

Cramer's V (adj.) |      95% CI
-----| [0.00, 1.00]
- One-sided CIs: upper bound fixed at [1.00].
```

```
effectsize::interpret_r(cv$Cramers_v, rules = "cohen1988")
```

```
[1] "very small"
(Rules: cohen1988)
```

Both tests reveal that the relationship between our variables is not significant ($p > 0.05$), and the effect sizes are very small. This result aligns with our visualisation shown in Figure 11.2 because everything looks very symmetrical.

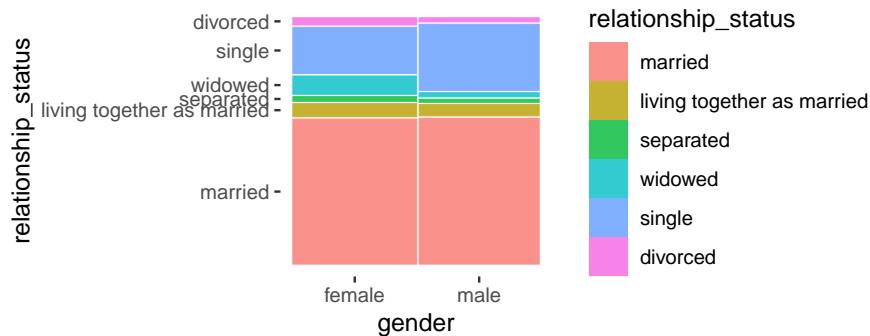
Contingency tables do not always come as 2x2 matrices. Therefore, it makes sense to look at one more example where we have a much larger matrix. Bear in mind that the larger your matrix, the larger your dataset has to be to produce reliable results.

Let's explore an example for a contingency table that is a 6x2 matrix. Imagine yourself at a soirée⁶, and someone might raise the question: Is it true that men are less likely to be married than women? To give you an edge over others at this French evening party, we can take a closer look at this matter with our `wvs_nona` dataset. We can create a mosaic plot and contingency table which feature `relationship_status` and `gender`.

As usual, let's start with a plot first. We create a mosaic plot to visualise all six levels of the factor `relationship_status` against the two levels of `gender`.

⁶A fancy way of saying ‘evening party’ in French.

```
wvs_nona |>
  ggplot() +
  geom_mosaic(aes(x = product(relationship_status, gender),
                  fill = relationship_status)) +
  theme_mosaic()
```



While this plot is a lot more colourful than our previous ones, it still suggests that differences across categories are not particularly large. Especially the category `married` seems to indicate that `male` and `female` participants do not differ by much, i.e. their relative frequency is very similar. The only more considerable difference can be found for `widowed` and `single`. Apparently, `female` participants more frequently indicated being `widowed` while more `male` participants indicated that they are `single`.

Next, we compute the numbers underpinning this plot, comparing the absolute distribution to check our conditions and the relative distribution to interpret the contingency table correctly.

```
# Check whether we fulfill the criteria
rel_gen <-
  wvs_nona |>
  count(relationship_status, gender)

rel_gen
```

A tibble: 12 x 3

relationship_status	gender	n
<fct>	<fct>	<int>
1 married	female	2688
2 married	male	2622
3 living together as married	female	242
4 living together as married	male	202
5 separated	female	87
6 separated	male	55
7 widowed	female	339
8 widowed	male	74
9 single	female	858
10 single	male	1188
11 divorced	female	134
12 divorced	male	75

```
# Compute relative frequency
rel_gen |>
  group_by(gender) |>
  mutate(perc = round(n / sum(n), 3)) |>
  pivot_wider(id_cols = relationship_status,
              names_from = gender,
              values_from = perc)
```

relationship_status	female	male
<fct>	<dbl>	<dbl>
1 married	0.618	0.622
2 living together as married	0.056	0.048
3 separated	0.02	0.013
4 widowed	0.078	0.018
5 single	0.197	0.282
6 divorced	0.031	0.018

The contingency table with the relative frequencies confirms what we suspected. The differences are very minimal between `male` and `female` participants. In a final step, we need to perform a Chi-squared test to see whether the differences are significant or not.

```
# Perform Chi-squared test
wvs_nona |> infer::chisq_test(relationship_status ~ gender)
```

	statistic	chisq_df	p_value
	<dbl>	<int>	<dbl>
1	250.	5	6.77e-52

```
# Compute effect size
(cv <- effectsize::cramers_v(wvs_nona$relationship_status,
                               wvs_nona$gender))
```

Cramer's V (adj.)	95% CI
0.17	[0.15, 1.00]

- One-sided CIs: upper bound fixed at [1.00].

```
effectsize::interpret_r(cv$Cramers_v, rules = "cohen1988")
```

```
[1] "small"
(Rules: cohen1988)
```

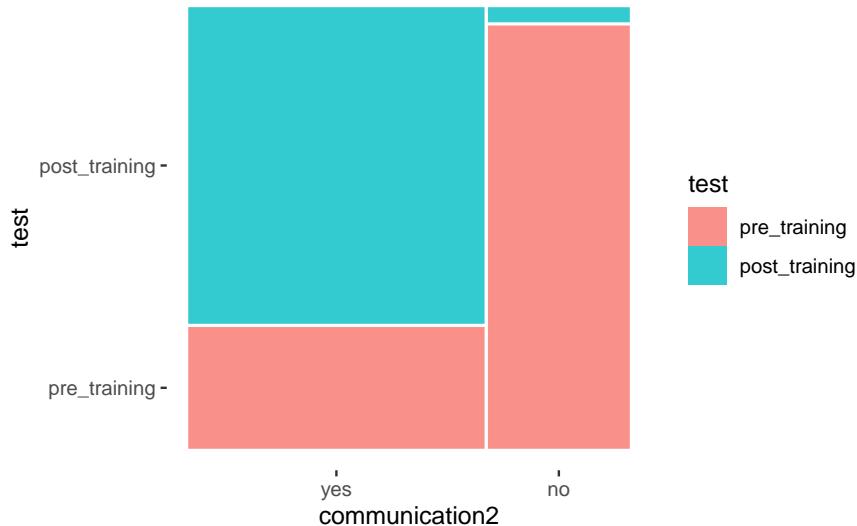
The statistical results further confirm that the relationship between `relationship_status` and `gender` is weak but significant. Therefore, at your next soirée, you can confidently say: “This is bogus. The World Value Survey, which covers 48 different countries, shows that such a relationship is weak considering responses from over 69,000 individuals.” #cheers-to-that

11.4.2 Paired groups of categorical variables

Similar to paired group comparisons, contingency tables may show paired data. For example, we could be interested in knowing whether an intercultural training we delivered improved participants skills (or at least their personal perception of having improved).

The dataset `ic_training` looks at participants’ confidence in `communication` with people from diverse cultural backgrounds before and after an intercultural training session. Thus, we might be curious to know whether the training was effective. The dataset contains multiple versions of the same variable measured in different ways. Let’s begin with the variable `communication2`, which measures improvement as a factor with only two levels, i.e. `yes` and `no`. We can start by plotting a mosaic plot.

```
ic_training |>
  ggplot() +
  geom_mosaic(aes(x = product(test, communication2),
                  fill = test)) +
  theme_mosaic()
```



The mosaic plot shows that more participants indicate to be more confident in communicating with culturally others **post-training**. There were only very few participants who indicated that they have not become more confident.

In the next step, we want to compute the contingency table. However, there is one aspect we need to account for: The responses are paired, and therefore we need a contingency table of paired responses. This requires some modification of our data. Let's start with a classic contingency table as we did for unpaired data.

```
# Unpaired contingency table
ic_training |>
  count(test, communication2) |>
  group_by(test) |>
  pivot_wider(id_cols = test,
              names_from = communication2,
              values_from = n) |>
  mutate(across(where(is.double), round, 2))
```

```
# A tibble: 2 × 3
# Groups:   test [2]
  test       yes     no
  <fct>     <int> <int>
1 pre_training    18     30
2 post_training   47      1
```

This contingency table reflects the mosaic plot we created, but it does not

show paired answers. If we sum the frequencies in each cell, we receive a score of 96, which equals the number of observations in `ic_training`. However, since we have two observations per person (i.e. paired responses), we only have 48 participants. Thus, this frequency table treats the pre and post-training group as independent from each other, which is obviously incorrect.

The current dataset includes two rows for each participant. Therefore, if we want to create a contingency table with paired scores, we first need to convert our data so that each participant is reflected by one row only, i.e. we need to make our data frame wider with `pivot_wider()`.

```
# Creating data frame reflecting paired responses
paired_data <-
  ic_training |>
  pivot_wider(id_cols = name,
              names_from = test,
              values_from = communication2)

paired_data
```

		pre_training	post_training
1	Hamlin, Christiana	no	no
2	Horblit, Timothy	no	yes
3	Grady, Justin	no	yes
4	Grossetete-Alfaro, Giovana	no	yes
5	Dingae, Lori	no	yes
6	el-Sabir, Saamyya	no	yes
7	Reynolds, Tylor	no	yes
8	Aslami, Andrew	no	yes
9	Alexander, Kiri	no	yes
10	Guzman Pineda, Timothy	no	yes
	# i 38 more rows		

In simple terms, we converted the column `communication2` into two columns based on the factor levels of `test`, i.e. `pre-training` and `post-training`. If we now use these two columns to create a contingency table and mosaic plot, we understand of how the paired distributions between `pre-training` and `post-training` look like.

```
# Contignency table with paired values
ct_paired <-
  paired_data |>
  count(pre_training, post_training) |>
```

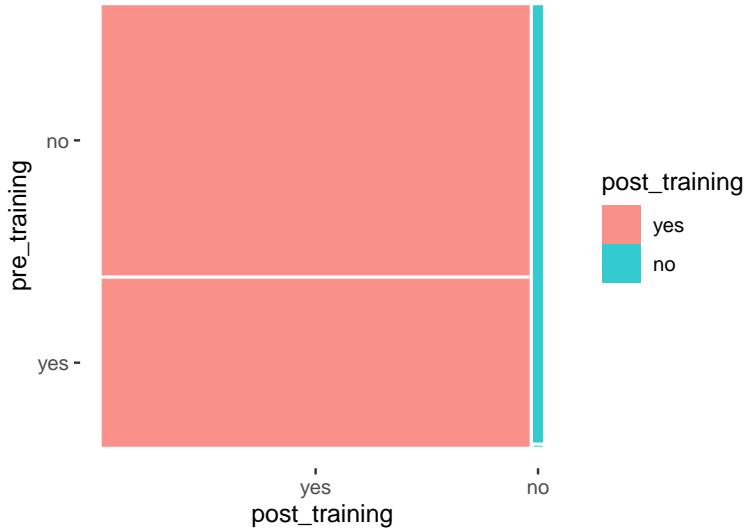
```
pivot_wider(names_from = post_training,
            names_prefix = "post_training_",
            values_from = n,
            values_fill = 0) # fill empty cells with '0'
ct_paired

# A tibble: 2 x 3
#>   pre_training post_training_yes post_training_no
#>   <fct>           <int>             <int>
#> 1 yes                18                  0
#> 2 no                 29                  1

# Contingency table with paired percentage values
ct_paired |>
  mutate(post_training_yes = post_training_yes / nrow(paired_data),
         post_training_no = post_training_no / nrow(paired_data)) |>
  mutate(across(where(is.double), round, 2))

# A tibble: 2 x 3
#>   pre_training post_training_yes post_training_no
#>   <fct>           <dbl>             <dbl>
#> 1 yes              0.38               0
#> 2 no               0.6                0.02

# Mosaic plot with paired values
paired_data |>
  ggplot() +
  geom_mosaic(aes(x = product(pre_training, post_training),
                  fill = post_training)
               ) +
  theme_mosaic()
```



With these insights, we can refine our interpretation and state the following:

- There were 19 participants (40%) for whom the training caused no change, i.e. before and after the training their response was still yes or no.
- However, for the other 29 participants (60%), the training helped them change from no to yes. This is an excellent result.
- We notice that no participant scored no after the training was delivered, which is also a great achievement.

Lastly, we need to perform a statistical test to confirm our suspicion that the training significantly impacted participants. For paired 2x2 contingency tables, we have to use *McNemar's Test*, using the function `mcnemar.test()`. To determine the effect size, we have to compute *Cohen's g* and use the function `cohens_g()` from the `effectsize` package to compute it. Be aware that we have to use the `paired_data` as our data frame and not `ic_training`.

```
#McNemar's test
mcnemar.test(paired_data$pre_training, paired_data$post_training)
```

McNemar's Chi-squared test with continuity correction

```
data: paired_data$pre_training and paired_data$post_training
McNemar's chi-squared = 27.034, df = 1, p-value = 1.999e-07
```

```
# Effect size
effectsize::cohens_g(paired_data$pre_training, paired_data$post_training)

Cohen's g |      95% CI
-----|-----
0.50    | [0.38, 0.50]
```

To correctly interpret the effect size J. Cohen (1988) suggest the following benchmarks:

- $g < 0.05 = \text{negligible}$,
- $0.05 \leq g < 0.15 = \text{small}$,
- $0.15 \leq g < 0.25 = \text{medium}$,
- $0.25 \leq g = \text{large}$.

Thus, our effect size is very large, and we can genuinely claim that the intercultural training had a significant impact on the participants. Well, there is only one flaw in our analysis. An eager eye will have noticed that one of our cells was empty, i.e. the top-right cell in the contingency table. The conditions to run such a test are similar to the Chi-squared test. Thus, using the McNemar test is not entirely appropriate in our case. Instead, we need to use the ‘*exact McNemar test*’, which compares the results against a binomial distribution and not a chi-squared one. More important than remembering the name or the distribution is to understand that the exact test produces more accurate results for smaller samples. However, we have to draw on a different package to compute it, i.e. `exact2x2` and its function `mcnemar.exact()`.

```
library(exact2x2)
mcnemar.exact(paired_data$pre_training, paired_data$post_training)
```

Exact McNemar test (with central confidence intervals)

```
data: paired_data$pre_training and paired_data$post_training
b = 0, c = 29, p-value = 3.725e-09
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
0.0000000 0.1356472
sample estimates:
odds ratio
0
```

```
detach("package:exact2x2", unload = TRUE)
```

Even after using a more robust computation for our data, the results are still significant. Thus, there is no doubt that the intercultural training helped participants to improve.

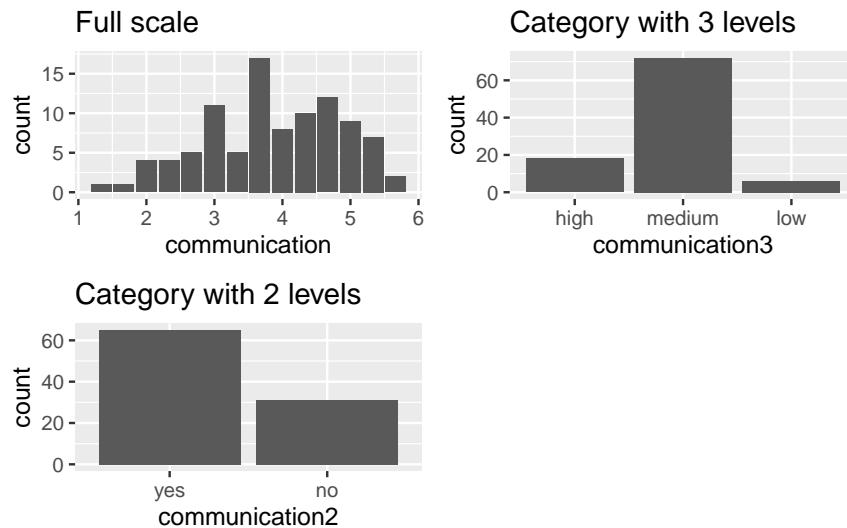
There are many other combinations of contingency tables. For example, the McNemar test can be extended to a 3x3 or higher matrix, but the rows and columns must be the same length. If you want to compare multiple categorical variables which do not follow a square matrix, it would be necessary to look into a loglinear analysis. While this book does not cover this technique, an excellent starting point is provided by Field (2013) (p. 732ff).

11.4.3 A final remark about comparing groups based on categorical data

Admittedly, I hardly ever find studies in the field of Social Sciences which work with primarily categorical data. Political Science might be an exception because working, for example, with polling data often implies working with categorical data. A reason you likely will not find yourself in the situation to work with such data and analytical techniques is measurement accuracy. If you look at the dataset `ic_training`, the variable `communication2` was artificially created to turn numeric data into a `factor`. This is usually not good practice because you lose measurement accuracy, and it can exaggerate differences between groups. Consider the following plots:

```
p1 <-  
  ic_training |>  
  ggplot(aes(x = communication)) +  
  geom_bar() +  
  ggtitle("Full scale")  
  
p2 <-  
  ic_training |>  
  ggplot(aes(x = communication3)) +  
  geom_bar() +  
  ggtitle("Category with 3 levels")  
  
p3 <-  
  ic_training |>  
  ggplot(aes(x = communication2)) +  
  geom_bar() +  
  ggtitle("Category with 2 levels")
```

```
# Combine plots into one single plot with 'patchwork' package
p1 + p2 + p3 + plot_spacer()
```



The more we aggregate the data into fewer categories, the more likely we increase differences between groups. For example, the plot `Category with 3 levels` shows that most participants fall into the category `medium`. However, reducing the categories to two, some participants are classified as `yes` and some as `no`. Thus, some respondents who were classified as `medium` are now in the `no` category. In reality, we know from our non-categorical measures that several participants will still have improved in confidence but are considered with those who have not improved. This is a major problem because it seems more people are not confident about communicating with people from different cultural backgrounds than there actually are. The accuracy of our measurement scale is poor.

In short, I recommend to only use the techniques outlined in this chapter if your data is truly categorical in nature. Thus, when designing your data collection tool, you might also wish to abstain from measuring quantitative variables as categories, for example `age`, which is a very popular choice among my students. While it might be sometimes more appropriate to offer categories, it is always good to use multiple categories and not just two or three. Every additional level in your factor will increase measurement accuracy and provides access to more advanced analytical techniques. Remember, it is always possible to convert `numeric` data into a `factor`, but not the other way around.

11.5 Reviewing group comparisons

In Social Sciences, we often compare individuals against each other or with themselves over time. As such, understanding how to compare groups of participants is an essential analytical skill. The techniques outlined in this chapter merely provide a solid starting point and likely cover about 90% of the datasets you will encounter or collect. For the other 10% of cases, you might have to look further for more niche approaches to comparing groups. There is also more to know about these techniques from a theoretical and conceptual angle. However, as far as the computation in *R* is concerned, you should be able to tackle your research projects confidently.

12

Power: You either have it or you don't

The most frequently asked question by my students is: How much data do I have to collect? My answer is always the same: '*It depends*'. From a student's perspective, this must be one of the most frustrating answers. Usually, I follow this sentence up with something more helpful to guide students on their way. For example, there is a way to determine how large your sample has to be to detect specific relationships reliably. This procedure is called *power analysis*.

While *power analysis* can be performed before and after data collection, it seems rather pointless to do it afterwards when collecting more data is not only challenging but sometimes impossible. Thus, my explanations in this chapter will primarily focus on the question: How much data is enough to reliably detect relationships between variables?

Power analysis is something that applies to all kinds of tests, such as correlations (Chapter 10), group comparisons (Chapter 11) and regressions (Chapter 13). Therefore, it is important to know about it. There is nothing worse than presenting a model or result that is utterly 'underpowered', i.e. the sample is not large enough to detect the desired relationship between variables reliably. In anticipation of your question about whether your analysis can be 'overpowered', the answer is 'yes'. Usually, there are no particular statistical concerns about having an overpowered analysis - more power to you (#pun-intended). However, collecting too much data can be an ethical concern. If we consider time as a scarce resource, we should not collect more data than we need. Participants' time is valuable, and being economical during data collection is crucial. We do not have to bother more people with our research than necessary because not everyone is as excited about our study as we are. Besides, if everyone conducts studies on many people, effects such as survey fatigue can set in, making it more difficult for others to carry out their research. Thus, being mindful of the required sample size is always important - I mean: essential.

12.1 Ingredients to achieve the power you deserve

We already know that sample size matters, for example, to assume normality when there is none in your sample (see Central Limit Theorem in Section 9.4). Also, we should not mistakenly assume that a relationship exists where there is none and vice versa. I am referring to so-called Type I and Type II errors (see Table 12.1).

Table 12.1: Type I and Type II error defined

Error	Meaning
Type I	<ul style="list-style-type: none"> • We assume a relationship exists between variables where there is none in the population. • We also refer to this as '<i>false positive</i>'. • While we find a significant relationship between variables in our sample, a larger sample would not find such a relationship. • Is represented by the Greek letter α (alpha). • The acceptance level of this error is equivalent to the p-value, i.e. significance level.
Type II	<ul style="list-style-type: none"> • We assume that there is no relationship between variables even though there is one in the population. • We also refer to this as '<i>false negative</i>'. • While our sample does not reveal a significant relationship between variables, a larger sample would show that the relationship is significant. • Is represented by the Greek letter β (beta).

Power analysis aims to help us avoid type II errors and, therefore, power is defined as the opposite of it, i.e. $1 - \beta$, i.e. a 'true positive'. To perform such a power analysis, we need at least three of the following four ingredients:

- the power level we want to achieve, i.e. the probability that we find a '*true positive*',
- the expected effect size (r) we hope to find,
- the significance level we set for our test, i.e. how strict we are about deciding when a relationship is significant, and
- the sample size.

As I mentioned earlier, it makes more sense to use a *power analysis* to determine the sample size. However, in the unfortunate event that you forgot to do

so, at least you can find out whether your results are underpowered by providing the sample size you obtained. If you find that your study is underpowered, you are in serious trouble and dire need of more participants.

12.2 Computing power

So, how exactly can we compute the right sample size, and how do we find all the numbers for these ingredients without empirical data at our disposal? First, we need a function that can compute it because computing a power analysis manually is quite challenging. The package `pwr` was developed to do precisely that for different kinds of statistical tests. Table Table 12.2 provides an overview of functions needed to perform power analysis for various tests covered in this book.

Table 12.2: Power analysis via the package ‘`pwr`’ for different methods covered in this book

Statistical method	Chapter	function in <code>pwr</code>
Correlation	Chapter 10	<code>pwr.r.test()</code>
T-test	Chapter 11	<code>pwr.t.test()</code> <code>pwr.t2n.test()</code> (for unequal sample sizes)
ANOVA	Section 11.3	<code>pwr.anova.test()</code>
Chi-squared test	Section 11.4	<code>pwr.chisq.test()</code>
Linear regression	Chapter 13	<code>pwr.f2.test()</code>

Since all functions work essentially the same, I will provide an example of last chapter’s correlation analysis. You likely remember that we looked at whether `happiness` and `life_satisfaction` are correlated with each other. Below are the results from the correlation analysis, which revealed a relationship in our sample based on the Health Index England (`hie_2021`) dataset.

```
hie_2021 |>
  select(happiness, life_satisfaction) |>
  correlation()

# Correlation Matrix (pearson-method)
```

Parameter1	Parameter2	r	95% CI	t(305)	p

happiness	life_satisfaction	0.67	[0.60, 0.73]	15.73	< .001***

p-value adjustment method: Holm (1979)
 Observations: 307

Let's define our ingredients to compute the ideal sample size for our correlation:

- *Power*: J. Cohen (1988) suggests that an acceptable rate of type II error is $\beta = 0.2$ (i.e. 20%). Thus, we can define the expected power level as $power = 1 - 0.2 = 0.8$.
- *Significance level* (i.e. α): We can apply the traditional cut-off point of 0.05 (see also Section 10.3).
- *Effect size*: This value is the most difficult to judge but substantially affects your sample size. It is easier to detect big effects than smaller effects. Therefore, the smaller our r, the bigger our sample size has to be. We know from our correlation that the effect size is 0.67, but what can you do to estimate your effect size if you do not have data available. Unfortunately, the only way to acquire an effect size is to either use simulated data or, as most frequently is the case, we need to refer to reference studies that looked at similar phenomena. I would always aim for slightly smaller effect sizes than expected, which means I need a larger sample. If the effect is bigger than expected, we will not find ourselves with an underpowered result. Thus, for our example, let's assume we expect $r = 0.6$.

All there is left to do is insert these parameters into our function `pwr.r.test()`.

```
pwr::pwr.r.test(r = 0.6,
                 sig.level = 0.05,
                 power = 0.8)
```

approximate correlation power calculation (arctanh transformation)

```
n = 18.63858
r = 0.6
sig.level = 0.05
power = 0.8
alternative = two.sided
```

To find the effect we are looking for, we only need 19 participants (we always round up!), and our dataset contains 307 observations. Thus, we are considerably overpowered for this type of analysis. What if we change the parameters

slightly and expect $r = 0.5$ and increase our threshold of accepting a true relationship, i.e. $p = 0.01$?

```
pwr::pwr.r.test(r = 0.5,
                 sig.level = 0.01,
                 power = 0.8)
```

```
approximate correlation power calculation (arctangh transformation)

n = 41.28159
r = 0.5
sig.level = 0.01
power = 0.8
alternative = two.sided
```

The results show that if we are stricter with our significance level and look for a smaller effect, we need about two times more participants in our sample, i.e. at least 42.

Given that we already know our sample size, we could also use this function to determine the power of our result ex-post. This time we need to provide the sample size n and we will not specify $power$.

```
pwr::pwr.r.test(n = 307,
                 r = 0.67,
                 sig.level = 0.01)
```

```
approximate correlation power calculation (arctangh transformation)

n = 307
r = 0.67
sig.level = 0.01
power = 1
alternative = two.sided
```

The results reveal that our power level is equivalent to cosmic entities¹, i.e. extremely powerful. In other words, we would not have needed a sample this large to find this genuine relationship.

¹https://marvelcinematicuniverse.fandom.com/wiki/Cosmic_Entities

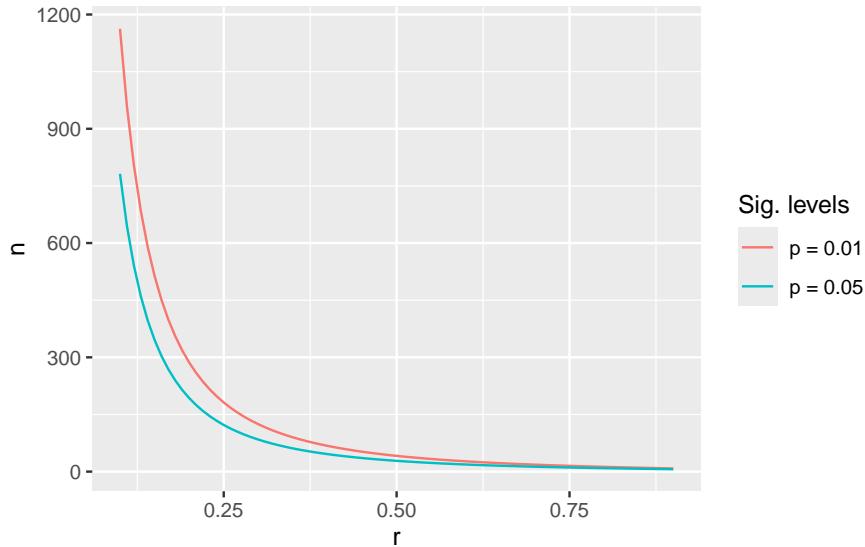
12.3 Plotting power

Lastly, I want to show you how to create a plot that reveals the different sample sizes needed to achieve $power = 0.8$. It visually demonstrates how much more data we need depending on the effect size. For the plot, I set a power level (i.e. 0.8) and significance levels of 0.01 and 0.05. I only varied effect sizes to see the required sample size.

```
glimpse(power_df)

Rows: 81
Columns: 3
Rowwise:
$ r      <dbl> 0.10, 0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19, 0.2~
$ n_0_01 <tibble[,3]> <tbl_df[26 x 3]>
$ n_0_05 <tibble[,3]> <tbl_df[26 x 3]>

# Plot the effect sizes against the sample size
power_df |>
  ggplot() +
  # The line for p = 0_01
  geom_line(aes(x = r,
                 y = n_0_01$n,
                 col = "p = 0.01")) +
  # The line for p = 0_05
  geom_line(aes(x = r,
                 y = n_0_05$n,
                 col = "p = 0.05")) +
  # Add nice labels
  labs(col = "Sig. levels",
       x = "r",
       y = "n")
```



The results reveal that the relationship between n and r is logarithmic and not linear. Thus, the smaller the effect size, the more difficult it is to reach reliable results.

The *R* code to produce this plot features `broom::tidy()` and `first()`. The function `tidy()` from the `broom` package converts the result from `pwr.r.test()` from a list to a tibble, and the function `first()` picks the first entry in this tibble, i.e. our sample sizes. Don't worry if this is confusing at this stage. We will return to `tidy()` in later chapters, where its use becomes more apparent.

12.4 Concluding remarks about power analysis

Conducting a *power analysis* is fairly straightforward for the techniques we cover in this book. Therefore there is no excuse to skip this step in your research. The challenging part for conducting a *power analysis* in advance is the definition of your effect size. If you feel you cannot commit to a single effect size, specify a range, e.g. $0.3 < \text{effectsize} < 0.5$. This will provide you with a range for your sample size.

For statistical tests like group comparisons (Chapter 11) and regressions (Chapter 13), the number of groups or variables also plays an important role. Usually, the more groups/variables are included in a test/model, the larger the sample has to be.

As mentioned earlier, it is not wrong to collect slightly more data, but you need

to be cautious not to waste ‘resources’. In other fields, like medical studies, it is absolutely critical to know how many patients should be given a drug to test its effectiveness. It would be unethical to administer a new drug to 300 people when 50 would be enough. This is especially true if the drug turns out to have an undesirable side effect. In Social Sciences, we are not often confronted with health-threatening experiments or studies. Still, being mindful of how research impacts others is essential. A *power analysis* can help to determine how you can carry out a reliable study without affecting others unnecessarily.



13

Regression: Creating models to predict future observations

Regressions are an exciting area of data analysis since it enables us to make very specific predictions, incorporating different variables simultaneously. As the name implies, regressions ‘regress’, i.e., draw on past observations to make predictions about future observations. Thus, any analysis incorporating a regression makes the implicit assumption that the past best explains the future.

I once heard someone refer to regressions as driving a car by looking at the rear-view mirror. As long as the road is straight, we will be able to navigate the car successfully. However, if there is a sudden turn, we might drive into the abyss. This makes it very clear when and how regressions can be helpful. Regressions are also a machine learning method, which falls under *models with supervised learning*. If you find machine learning fascinating, you might find the book “*Hands-on Machine Learning with R*” (Boehmke and Greenwell 2019) very insightful and engaging.

In the following chapters, we will cover three common types of regressions, which are also known as *Ordinary Least Squares (OLS) regression* models:

- Single linear regression,
- Multiple regression, and
- Hierarchical regression, as a special type of multiple regression.

These three types will allow you to perform any OLS regression you could imagine. We can further distinguish two approaches to modelling via regressions:

- *Hypothesis testing*: A regression model is defined ex-ante.
- *Machine learning*: A model is developed based on empirical data.

In the following chapters, we will slightly blur the lines between both approaches by making assumptions about relationships (hypothesising) and make informed decisions based on our data (exploring).

All our regressions will be performed using the `covid` dataset of the `r4np` package to investigate whether certain factors can predict COVID numbers in different countries. I felt this book would not have been complete without

covering this topic. After all, I wrote this piece during the pandemic, and it likely will mark a darker chapter in human history.

13.1 Single linear regression

A single linear regression looks very similar to a correlation (see Chapter 10), but it is different in that it defines which variable affects another variable, i.e. a directed relationship. I used the terms *dependent variable (DV)* and *independent variable (IV)* previously when comparing groups (see Chapter 11), and we will use them here again. In group comparisons, the independent variable was usually a `factor`, but in regressions, we can use data that is not a categorical variable, i.e. `integer`, `double`, etc.

While I understand that mathematical equations can be confusing, they are fairly simple to understand with regressions. Also, when writing our models in *R*, we will continuously use a `formula` to specify our regression. Thus, it is advisable to understand them. For example, a single linear regression consists of one independent variable and one dependent variable:

$$DV = \beta_0 + IV * \beta_1 + error$$

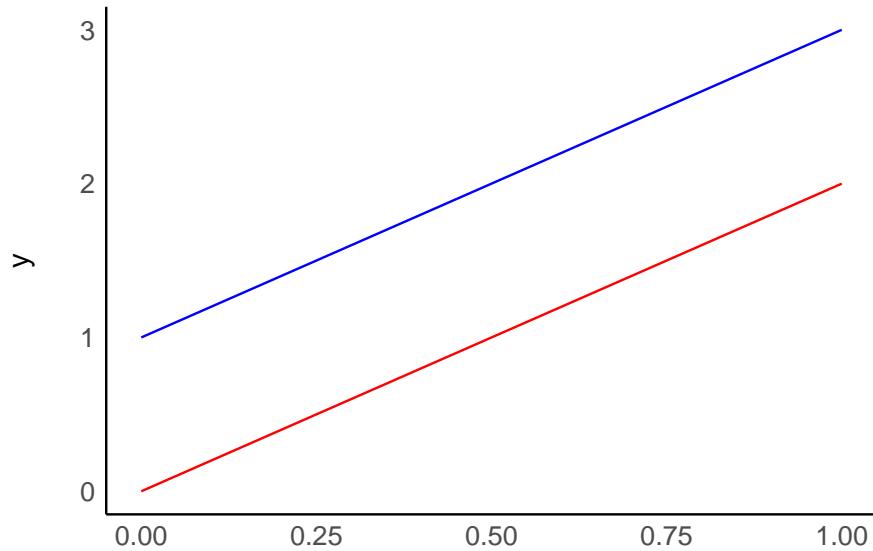
Beta (β) represents the coefficient of the independent variable, i.e. how much a change in IV causes a change in DV. For example, a one-unit change in IV might mean that DV changes by two units of IV:

$$DV = \beta_0 + IV * 2 + error$$

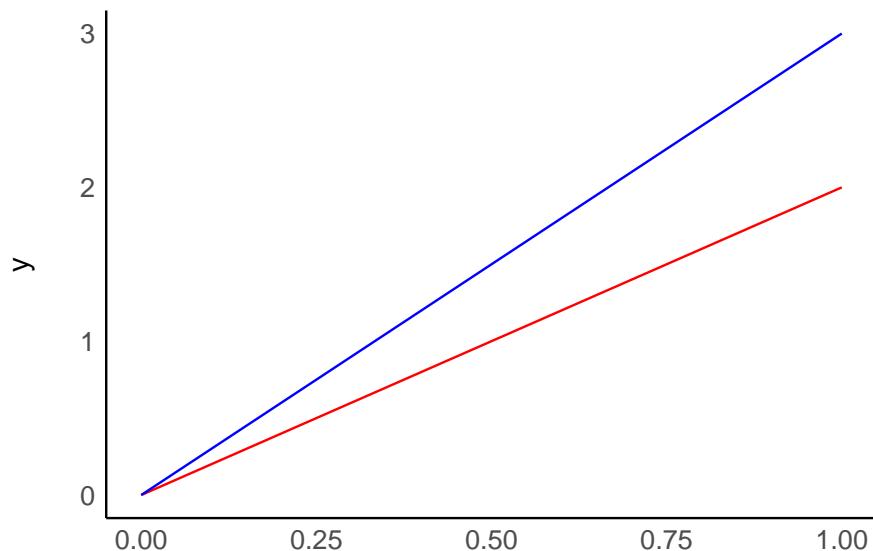
If we ignore β_0 and *error* for a moment, we find that if $IV = 3$, our $DV = 3 * 2 = 6$. Similarly, if $IV = 5$, we find that $DV = 10$, and so on. According to this model, DV will always be twice as large as IV.

You might be wondering what β_0 stands for. It indicates an offset for each value, also called the intercept. Thus, no matter which value we choose for IV, DV will always be β_0 different from IV. It is a constant in our model. This can be best explained by visualising a regression line. Pay particular attention to the expressions after `function(x)`

```
# A: Two models with different beta(0)
ggplot() +
  geom_function(fun = function(x) 0 + x * 2, col = "red") +
  geom_function(fun = function(x) 1 + x * 2, col = "blue") +
  see::theme_modern()
```



```
# B: Two models with the same beta(0), but different beta(1)
ggplot() +
  geom_function(fun = function(x) 0 + x * 2, col = "red") +
  geom_function(fun = function(x) 0 + x * 3, col = "blue") +
  see:::theme_modern()
```



Plot A shows two regressions where only β_0 differs, i.e. the intercept. On the

other hand, plot B shows what happens if we change β_1 , i.e. the slope. The two models in plot B have the same intercept and, therefore, the same origin. However, the blue line ascends quicker than the red one because its β_1 is higher than for the red model.

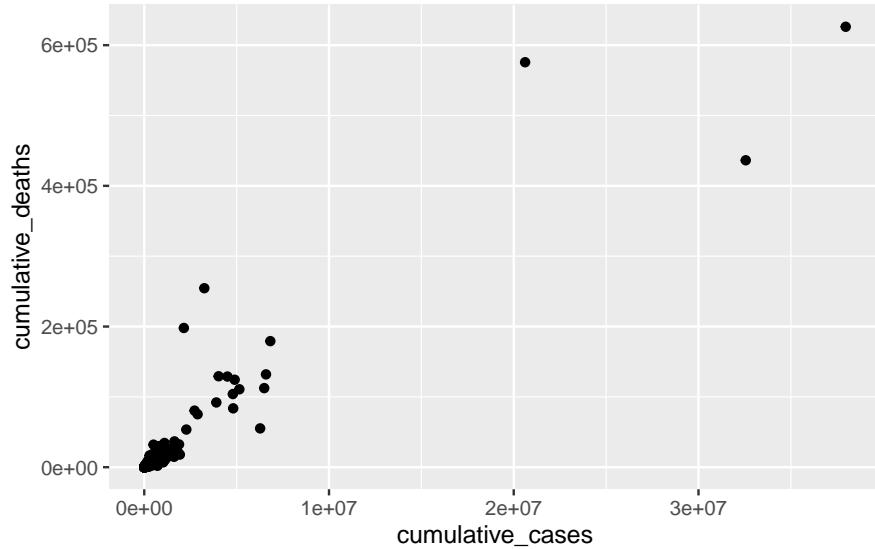
Lastly, the *error* component in the regression model refers to the deviation of data from these regression lines. Ideally, we want this value to be as small as possible.

13.1.1 Fitting a regression model by hand, i.e. trial and error

If everything so far sounds all awfully theoretical, let's try to fit a regression model by hand. First, we need to consider what our model should be able to predict. Let's say that the number of COVID-19 cases predicts the number of deaths due to COVID-19. Intuitively we would assume this should be a linear relationship because the more cases there are, the more likely we find more deaths caused by it.

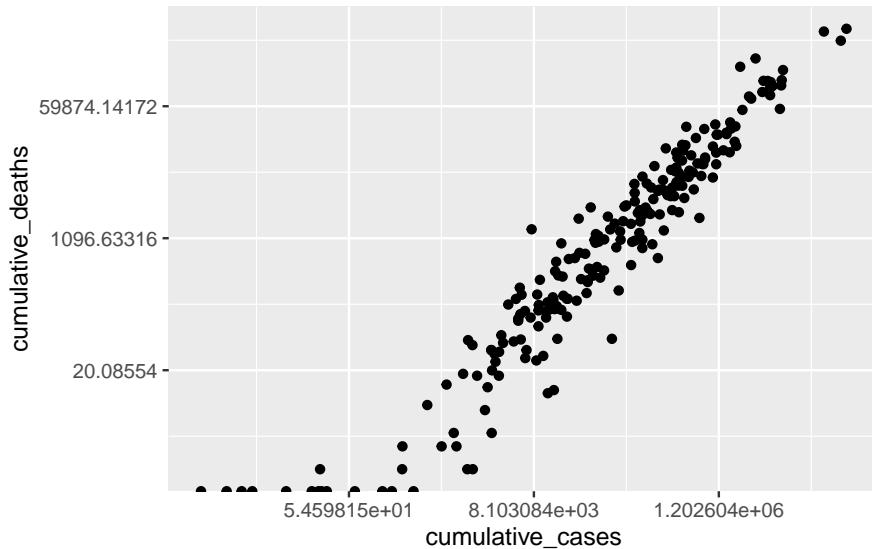
```
# We only select most recent numbers, i.e. "2021-08-26"
# and countries which have COVID cases
covid_sample <-
  covid |>
  filter(date_reported == "2021-08-26" &
         cumulative_cases != 0)

covid_sample |>
  ggplot(aes(x = cumulative_cases,
             y = cumulative_deaths)) +
  geom_point()
```



This data visualisation does not show us much. For example, we can see three countries, which appear to have considerably more cases than most other countries. Thus, all other countries are crammed together in the bottom left corner. To improve this visualisation without removing the outliers, we can rescale the x- and y-axis using the function `scale_x_continuous()` and `scale_y_continuous()` and apply a "log" transformation.

```
covid_sample |>
  ggplot(aes(x = cumulative_cases,
             y = cumulative_deaths)) +
  geom_point() +
  scale_x_continuous(trans = "log") +
  scale_y_continuous(trans = "log")
```



As a result, the scatterplot is now easier to read, and the dots are more spread out. This reveals that there is quite a strong relationship between `cumulative_cases` and `cumulative_deaths`. However, similar to before, we should avoid outliers when performing our analysis. For the sake of simplicity, I will limit the number of countries included in our analysis, which also removes the requirement of using `scale_x_continuous()` and `scale_y_continuous()`.

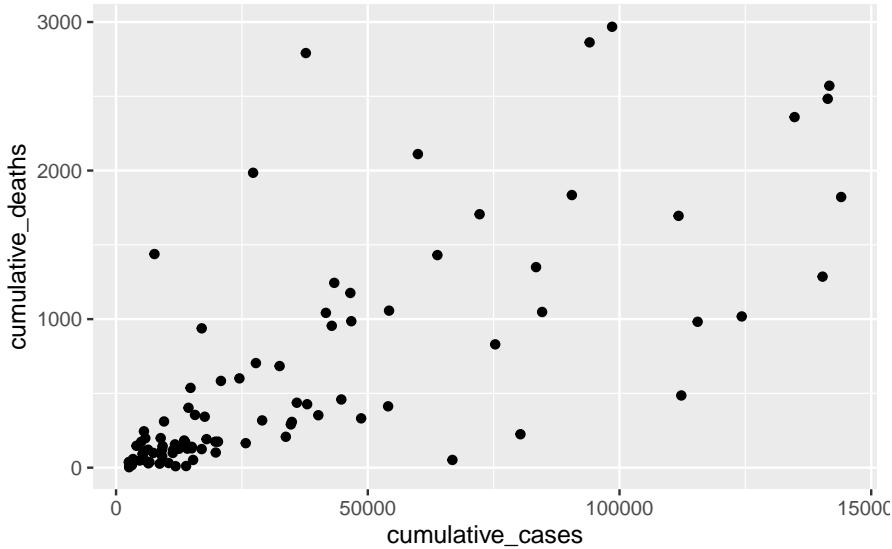
```

covid_sample <-
  covid_sample |>
  filter(date_reported == "2021-08-26" &
         cumulative_cases >= 2500 &
         cumulative_cases <= 150000 &
         cumulative_deaths <= 3000)

plot <- covid_sample |>
  ggplot(aes(x = cumulative_cases,
             y = cumulative_deaths)) +
  geom_point()

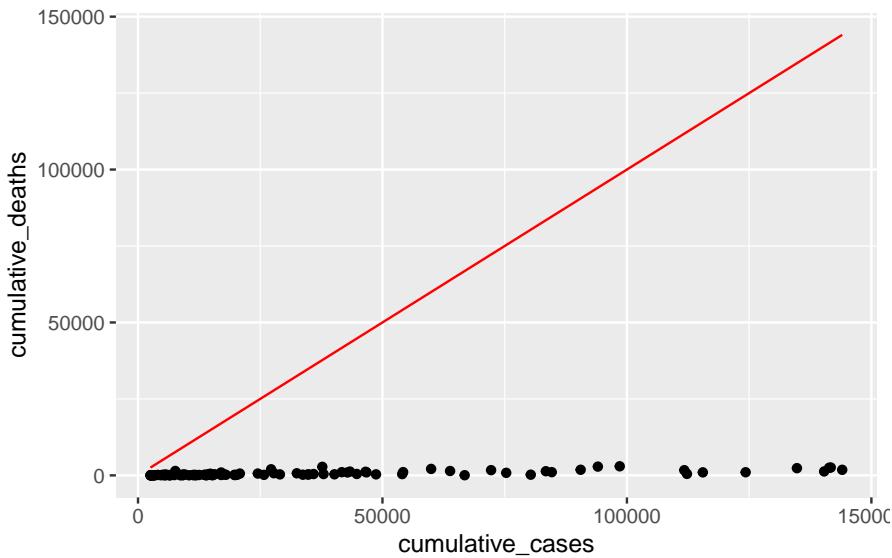
plot

```



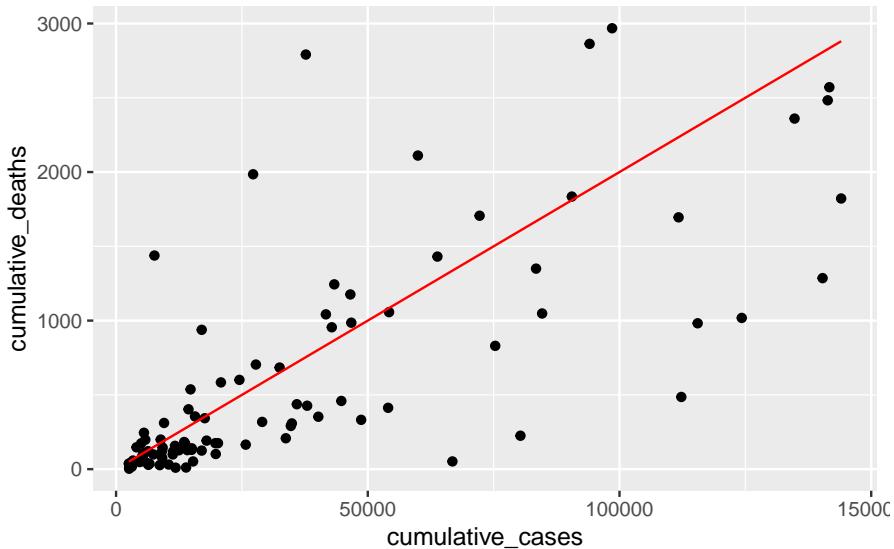
We can try to fit a straight line on top by adjusting the beta values through trial and error. This is effectively what we hope to achieve with a regression: the β values, which best explain our data. Let's start with the basic assumption of $y = x$ without specific β s, i.e. they are 0.

```
plot +
  geom_function(fun = function(x) x, col = "red")
```



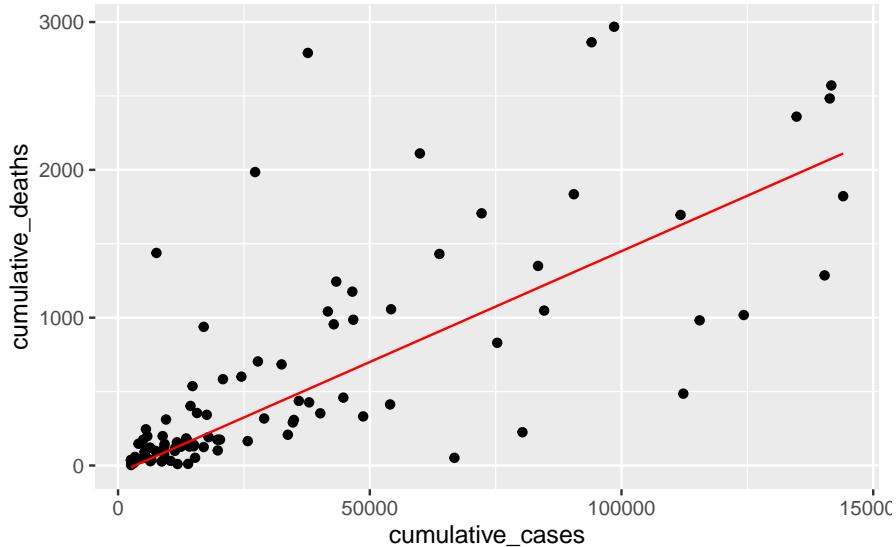
What we try to achieve is that the red line fits nicely inside the cloud of dots. Our simple model provides a very poor fit to our data points, because the dots are considerably below it. This makes sense because $y = x$ would imply that every COVID-19 case leads to a death, i.e. everyone with COVID did not survive. From our own experience, we know that this is luckily not true. Ideally, we want the line to be less steep. We can do this by adding a β_1 to our equation. Maybe only 2% of people who got COVID-19 might not have recovered, i.e. $\beta_1 = 0.02$.

```
plot +
  geom_function(fun = function(x) 0.02 * x, col = "red")
```



This time the line looks much more aligned with our observations. However, one could argue that it might have to move a little to the right to cover the observations at the bottom better. Therefore, we should add a β_0 to our equation, e.g. -50. This moves the line to the right. I also adjusted β_1 ever so slightly to make it fit even better.

```
plot +
  geom_function(fun = function(x) -50 + 0.015 * x, col = "red")
```



We finished creating our regression model. If we wanted to express it as a formula, we would write $DV = -5 + 0.015 * IV$. We could use this model to predict how high COVID cases will likely be in other countries not included in our dataset.

13.1.2 Fitting a regression model computationally

Estimating a regression model by hand is not ideal and far from accurate. Instead, we would compute the β s based on our observed data, i.e. `cumulative_cases` and `cumulative_deaths`. We can use the function `lm()` to achieve this. I also rounded (`round()`) all numeric values to two decimal places to make the output easier to read. We also use `tidy()` to retrieve a cleaner result from the computation.

```
# We define our model
m0 <- lm(cumulative_deaths ~ cumulative_cases, data = covid_sample)

# We clean the output to make it easier to read
broom::tidy(m0) |>
  mutate(across(where(is.numeric),
    round, 2)
)
```

Warning: There was 1 warning in `mutate()`.
 i In argument: `across(where(is.numeric), round, 2)`.
 Caused by warning:

! The `...` argument of `across()` is deprecated as of dplyr 1.1.0.
Supply arguments directly to ` `.fns` through an anonymous function instead.

```
# Previously
across(a:b, mean, na.rm = TRUE)

# Now
across(a:b, \(x) mean(x, na.rm = TRUE))

# A tibble: 2 × 5
  term      estimate std.error statistic p.value
  <chr>     <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept) 88.1      70.5      1.25    0.21
2 cumulative_cases 0.01       0       10.7     0
```

We first might notice that the `p.value` indicates that the relationship between `cumulative_death` and `cumulative_cases` is significant. Thus, we can conclude that countries with more COVID cases also suffer from higher numbers of people who do not recover successfully from it. However, you might be wondering where our β scores are. They are found where it says `estimate`. The standard error (`std.error`) denotes the error we specified in the previous equation. In the first row, we get the β_0 , i.e. the intercept (88.10). This one is larger than what we estimated, i.e. -50. However, β_1 is 0.01, which means we have done a very good job guessing this estimate. Still, it becomes apparent that it is much easier to use the function `lm()` to estimate a model than ‘eyeballing’ it.

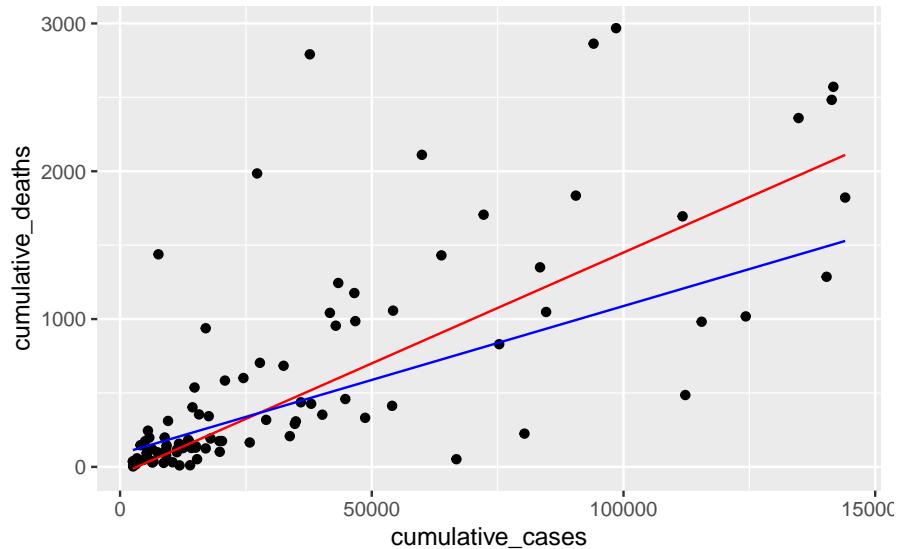
Let me also briefly explain what the function `mutate(across(where(is.numeric), round, 2))` does in our computation:

- `mutate()` implies we are changing values of variables,
- `across()` indicates that whatever function is placed insight this function will be applied ‘across’ certain columns. We specified our columns with
- `where(is.numeric)`, which indicates that all columns should be selected if they are `numeric`.
- `round, 2` represents the function `round()` and is applied with the parameter `2` to round numbers to two decimal places.

While this might seem like a more advanced method of handling your data, it is good to get into the habit of writing more efficient code, because it tends to be less prone to errors.

Returning to our data, we can now visualise the computed model (in blue) and our guessed model (in red) in one plot and see the differences. The plot shows that both regression lines are fairly close to each other. Admittedly, it was relatively easy to fit this model by hand. However, it is much more difficult to do so when more than two variables are involved.

```
plot +
  geom_function(fun = function(x) -50 + 0.015 * x, col = "red") +
  geom_function(fun = function(x) 88.1 + 0.01 * x, col = "blue")
```



With our final model computed, we also need to check its quality in terms of predictive power based on how well it can explain our observed data. We tested models before when we looked at confirmatory factor analyses for latent variables (see Section 7.8). This time we want to know how accurate our model is in explaining observed data and, therefore, how accurate it predicts future observations. The package `performance` offers a nice shortcut to compute many different indicators at once:

- `check_model()`: Checks for linearity, homogeneity, collinearity and outliers
- `model_performance()`: Tests the quality of our model.

For now, we are mainly interested in the performance of our model. So, we can compute it the following way:

```
performance::model_performance(m0)
```

```
# Indices of model performance
```

AIC		AICc		BIC		R2		R2 (adj.)		RMSE		Sigma
<hr/>												
1472.626		1472.887		1480.319		0.548		0.543		502.573		507.891

We are presented with quite a number of performance indicators, and here is how to read them:

- AIC stands for *Akaike Information Criterion*, and the lower the score, the better the model.
- BIC stands for *Bayesian Information Criterion*, and the lower the score, the better the model.
- R² stands for *R squared* (R^2) and is also known as the coefficient of determination. It measures how much the independent variable can explain the variance in the dependent variable. In other words, the higher R^2 , the better is our model, because our model can explain more of the variance. R^2 falls between 0-1, where 1 would imply that our model can explain 100% of the variance in our sample. R^2 is also considered a *goodness-of-fit measure*.
- R² (adj.) stands for *adjusted R squared*. The adjusted version of R^2 becomes essential if we have more than one predictor (i.e. multiple independent variables) in our regression. The adjustment of R^2 accounts for the number of independent variables in our model. Thus, we can compare different models, even though they might have different numbers of predictors. It is important to note that unadjusted R^2 will always increase if we add more predictors.
- RMSE stands for *Root Mean Square Error* and indicates how small or large the prediction error of the model is. Conceptually, it aims to measure the average deviations of values from our model when we attempt predictions. The lower its score, the better, i.e. a score of 0 would imply that our model perfectly fits the data, which is likely never the case in the field of Social Sciences. The *RMSE* is particularly useful when trying to compare models.
- Sigma stands for the standard deviation of our residuals (the difference between predicted and empirically observed values) and is a measure of prediction accuracy. Sigma is '*a measure of the average distance each observation falls from its prediction from the model*' (Gelman, Hill, and Vehtari 2020) (p. 168).

Many of these indices will become more relevant when we compare models. However, R^2 can also be meaningfully interpreted without a reference model. We know that the bigger R^2 , the better. In our case, it is 0.548, which is very good considering that our model consists of only one predictor. It is not easy to interpret whether a particular R^2 value is good or bad. In our simple single linear regression, R^2 is literally '*r squared*', which we already know from correlations and their effect sizes (see Table 10.2). Thus, if we take the square root of R^2 we can retrieve the correlation coefficient, i.e. $r = \sqrt{R^2} = \sqrt{0.548} = 0.740$. According to J. Cohen (1988), this would count as a large effect size.

However, the situation is slightly more complicated for multiple regressions, and we cannot use Cohen's reference table by taking the square root of R^2 .

Still, the meaning of R^2 and its adjusted version remain the same for our model.

Once you have a model and it is reasonably accurate, you can start making predictions. This can be achieved by using our model object `m0` with the function `add_predictions()` from the `modelr` package. However, first, we should define a set of values for our independent variable, i.e. `cumulative_cases`, which we store in a tibble using the `tibble()` function.

```
df_predict <-
  tibble(cumulative_cases = c(100, 1000, 10000))

# Make our predictions
df_predict |>
  modelr::add_predictions(m0) |>
  mutate(pred = round(pred, 0))

# A tibble: 3 × 2
  cumulative_cases   pred
              <dbl> <dbl>
1             100     90
2            1000    102
3           10000    232
```

The predictions are stored in column `pred`. Therefore, we know how many deaths from COVID have to be expected based on our model for each value in our dataset.

Single linear regressions are simple and an excellent way to introduce novice users of R to modelling social phenomena. However, we hardly ever find that a single variable can explain enough variance to be a helpful model. Instead, we can most likely improve most of our single regression models by considering more variables and using a multiple regression technique.

13.2 Multiple regression

Multiple regressions expand single linear regressions by allowing us to add more variables. Maybe less surprising, computing a multiple regression is similar to a single regression in R because it requires the same function, i.e. `lm()`. However, we add more IVs. Therefore, the equation we used before needs to be modified slightly by adding more independent variables. Each of these variables will have its own β value:

$$DV = \beta_0 + IV_1 * \beta_1 + IV_2 * \beta_2 + \dots + IV_n * \beta_n + error$$

In the last section, we wanted to know how many people will likely not recover from COVID. However, it might be even more interesting to understand how we can predict new cases and prevent casualties from the outset. Since I live in the United Kingdom during the pandemic, I am curious whether specific COVID measures help reduce the number of new cases in this country. To keep it more interesting, I will also add Germany to the mix since it has shown to be very effective in handling the pandemic relative to other European countries. Of course, feel free to pick different countries (maybe the one you live in?) to follow along with my example. In Section 13.3.2 it will become apparent why I chose two countries (#spoiler-alert).

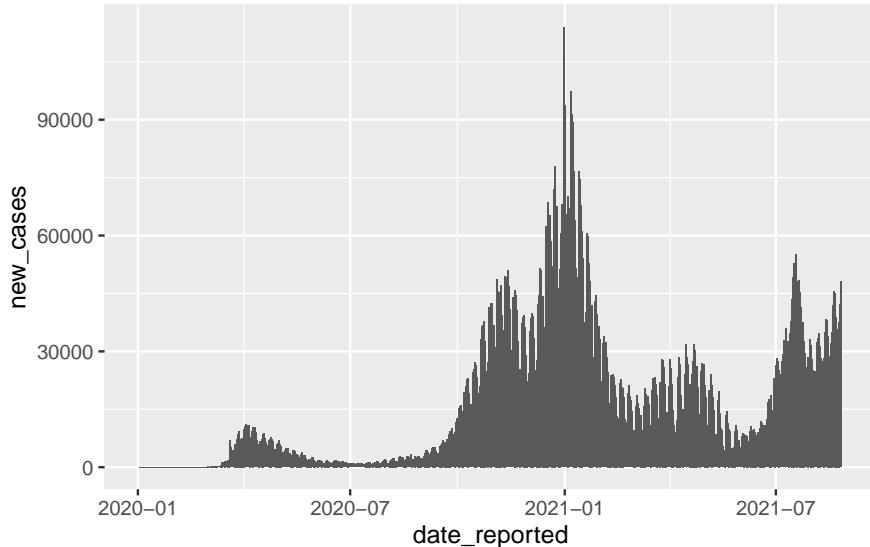
First, we create a dataset that only contains information from the United Kingdom and Germany¹, using `filter()`.

```
covid_uk_ger <-
  covid |>
  filter(iso3 == "GBR" | iso3 == "DEU")
```

In the next step, we might want to know how new cases are distributed over time. It is always good to inspect the dependent variable to get a feeling of how much variance there is in our data. Having a more extensive range of data values is ideal because the regression model will consider low and high values of the dependent variable instead of just high or low scores. If you find that your dependent variable shows minimal variance, your model will likely be ‘overfitted’. An overfitted model can very well explain the sample data but performs poorly with a new dataset. This is, of course not desirable, because the value of creating a model is to predict future observations. Let’s plot the DV `new_cases` across time to see when and how many new COVID cases had to be reported for both countries together.

```
covid_uk_ger |>
  ggplot(aes(x = date_reported,
             y = new_cases)) +
  geom_col()
```

¹Germany is called ‘Deutschland’ in German, hence the abbreviation ‘DEU’ according to the ISO country code².



We can tell that there are different waves of new cases with very low and very high values. As such, we should find variables that help us explain when `new_cases` are high and when they are low. If you have hypotheses to test, you would already know which variables to include in your regression. However, we do not have a hypothesis based on our prior reading or other studies. Thus, we pick variables of interest that we suspect could help us with modelling new COVID cases. For example, we can be fairly confident that the number of new COVID cases should be lower if more safety measures are in place - assuming that they are effective and everyone adheres to them. The `covid` dataset includes such information evaluated by the WHO³, i.e. `masks`, `travel`, `gatherings`, `schools` and `movements`. Remember, you can always find out what these variables stand for by typing `?covid` into the console. A higher value for these variables indicates that there were more safety measures in place. Scores can range from 0 (i.e. no measures are in place) to 100 (i.e. all WHO measures are in place).

We can add multiple variables by using the `+` symbol in the `lm()` function.

```
# Create our model
m0 <- lm(new_cases ~ masks + movements + gatherings +
           schools + businesses + travel,
           data = covid_uk_ger)

# Inspect the model specifications.
# I always round the p.value since I
```

³<https://covid19.who.int/info/>

```
# do not prefer the scientific notation
broom::tidy(m0) |>
  mutate(p.value = round(p.value, 3))

# A tibble: 7 x 5
  term      estimate std.error statistic p.value
  <chr>     <dbl>     <dbl>     <dbl>     <dbl>
1 (Intercept) -2585.     745.    -3.47    0.001
2 masks       -14.8      9.18    -1.61    0.108
3 movements   -54.7     15.5    -3.52     0
4 gatherings  -26.0     19.8    -1.31    0.191
5 schools      394.     29.7     13.2     0
6 businesses   93.7     13.8     6.79     0
7 travel        49.5     9.30     5.32     0

# Evaluate the quality of our model
performance::model_performance(m0)

# Indices of model performance

AIC | AICc | BIC | R2 | R2 (adj.) | RMSE | Sigma
-----|-----|-----|-----|-----|-----|-----
25277.622 | 25277.744 | 25318.262 | 0.233 | 0.229 | 10027.204 | 10056.877
```

Overall (and purely subjectively judged), the model is not particularly great because even though we added so many variables, the *adjusted R²* is not particularly high, i.e. ‘only’ 0.229. As mentioned earlier, for multiple regression, it is better to look at *adjusted R²*, because it adjusts for the number of variables in our model and makes the comparison of different models easier. There are a couple more important insights gained from this analysis:

- Not all variables appear to be significant. The predictors `masks` and `gatherings` are not significant, i.e. `p.value > 0.05`. Thus, it might be worth removing these variables to optimise the model.
- The variable, `movements` seems to reduce `new_cases`, i.e. it has a negative estimate (β).
- However, `schools`, `businesses`, and `travel` have a positive effect on `new_cases`.

Especially the last point might appear confusing. How can it be that if more measures are taken, the number of new COVID cases increases? Should we avoid them? We have not considered in our regression that measures might be put in place to reduce the number of new cases rather than to prevent them.

Thus, it might not be the case that `schools`, `businesses`, and `travel` predict higher `new_cases`, but rather the opposite, i.e. due to higher `new_cases`, the measures for `schools`, `businesses` and `travel` were tightened, which later on (with a time lag) led to lower `new_cases`. Thus, the relationships might be a bit more complicated, but to keep things simple, we accept that with our data, we face certain limitations (as is usually the case)⁴.

As a final step, we should remove the variables that are not significant and see how this affects our model.

```
m1 <- lm(new_cases ~ movements + schools + businesses + travel,
          data = covid_uk_ger)

broom:::tidy(m1) |>
  mutate(p.value = round(p.value, 3))

# A tibble: 5 x 5
  term      estimate std.error statistic p.value
  <chr>     <dbl>     <dbl>     <dbl>     <dbl>
1 (Intercept) -3024.     724.    -4.18      0
2 movements     -64.0     13.4    -4.79      0
3 schools       382.      28.8     13.3      0
4 businesses     90.5     12.9     7.01      0
5 travel        45.7      9.17     4.99      0

# Evaluate the quality of our model
performance::model_performance(m1)

# Indices of model performance
```

AIC	AICc	BIC	R2	R2 (adj.)	RMSE	Sigma
25279.519	25279.590	25309.999	0.229	0.227	10052.123	10073.343

Comparing our original model `m0` with our revised model `m1`, we can see that our `R2` (adj.) barely changed. Thus, `m1` is a superior model because it can explain (almost) the same amount of variance but with fewer predictors. The model `m1` would also be called a *parsimonious model*, i.e. a model that is simple but has good predictive power. When reading about multiple regressions, you might often hear people mention the ‘*complexity*’ of a model, which refers to the number of predictors. The complexity of a model is known as the ‘*degrees of freedom (df) of the numerator*’ and is computed as *number of predictors* – 1.

⁴Be aware that such insights might also fundamentally question the relationship between variables, i.e. are the measures truly independent variables or rather dependent ones?

For example, in our model of 4 independent variables, the df of the numerator is 3. This value is relevant when computing the power of a regression model (see also Chapter 12).

```
performance::compare_performance(m0, m1)
```

```
# Comparison of Model Performance Indices
```

Name	Model	AIC (weights)	AICc (weights)	BIC (weights)	R2	R2 (adj.)	RMSE	Sigma
<hr/>								
m0	lm	25277.6 (0.721)	25277.7 (0.716)	25318.3 (0.016)	0.233	0.229	10027.204	10056.8
m1	lm	25279.5 (0.279)	25279.6 (0.284)	25310.0 (0.984)	0.229	0.227	10052.123	10073.3

However, there are a couple of things we overlooked when running this regression. If you are familiar with regressions already, you might have been folding your hands over your face and burst into tears about the blasphemous approach to linear regression modelling. Let me course-correct at this point.

Similar to other parametric approaches, we need to test for sources of bias, linearity, normality and homogeneity of variance. Since multiple regressions consider more than one variable, we must consider these criteria in light of other variables. As such, we have to draw on different tools to assess our data. There are certain pre- and post-tests we have to perform to evaluate and develop a multiple regression model fully:

- *Pre-test*: We need to consider whether there are any outliers and whether all assumptions of OLS regression models are met.
- *Post-test*: We need to check whether our independent variables correlate very strongly with each other, i.e. are there issues of multiple collinearity.

We already covered aspects of linearity, normality and homogeneity of variance. However, outliers and collinearity have to be reconsidered for multiple regressions.

13.2.1 Outliers in multiple regressions

While it should be fairly clear by now why we need to handle outliers (remember Section 9.6), our approach is somewhat different when we need to consider multiple variables at once. Instead of identifying outliers for each variable independently, we have to consider the interplay of variables. In other words, we need to find out how an outlier in our independent variable affects the overall model rather than just one other variable. Thus, we need a different technique to assess outliers. By now, you might not be shocked to find that there is more than one way of identifying outliers in regressions and that there are many different ways to compute them in *R*. (P. Cohen, West, and Aiken

2014) distinguishes between where one can find outliers in the model as summarised in Table 13.1). I offer a selection of possible ways to compute the relevant statistics, but this list is not exhaustive. For example, many of these statistics can also be found using the function `influence.measures()`.

Table 13.1: Outlier detection in multiple regressions

Outlier in?	Measures	function in R
<i>Dependent variable</i>	• Internally studentised residuals	• <code>rstandard()</code> or <code>fortify()</code>
<i>Dependent variable</i>	• Externally studentised residuals	• <code>rstudent()</code>
<i>Independent variable</i>	• Leverage	• <code>hatvalues()</code> or <code>fortify()</code>
<i>Independent variable</i>	• Mahalanobis distance	• <code>mahalanobis()</code> ⁵ or <code>mahalanobis_distance()</code> ⁶
<i>Entire model</i>	<i>Global measures of influence</i>	<i>Global measures of influence</i>
	• DFFITS, • Cook's d	• <code>dffits()</code> • <code>cooks.distance()</code> or <code>fortify()</code>
<i>Entire model</i>	<i>Specific measures of influence:</i> • DFBETAS	<i>Specific measures of influence:</i> • <code>dfbetas()</code>

While it might be clear why we need to use different approaches to find outliers in different model components, this might be less clear when evaluating outliers that affect the entire model. We distinguish between *global measures of influence*, which identify how a single observation affects the quality of the entire model, and *specific measures of influence*, which determine how a single observation affects each independent variable, i.e. its regression coefficients denoted as β . It is recommended to look at all different outlier measures before venturing ahead to perform linear multiple regression. Going through the entire set of possible outliers would go way beyond the scope of this book. So, I will focus on five popular measures which cover all three categories:

- Dependent variable: Externally studentised residuals
- Independent variable: Leverage and Mahalanobis distance
- Entire model: Cook's d and DFBETAS

⁵This function does not take the model as an attribute, but instead requires the data, the column means and the covariance. Thus, we have to specify this function in the following way: `mahalanobis(data, colMeans(data), cov(data))`

⁶Function from `rstatix` package which automatically classifies values as outliers for us.

The approach taken is the same for the other outlier detection methods, but with different functions. Thus, it should be quite simple to reproduce these as well after having finished the chapters below.

13.2.1.1 Outliers in the dependent variable

Irrespective of whether we look at independent or dependent variables, we always want to know whether extreme values are present. Here I will use the externally studentised residual, which is the preferred statistic to use to identify cases whose (...) values are highly discrepant from their predicted values (P. Cohen, West, and Aiken 2014) (p.401).

First, we need to compute the residuals for our model as a new column in our dataset. Since we also want to use other methods to investigate outliers, we can use the function `fortify()`, which will add some of the later indicators and creates a `tibble` that only includes the variables from our model. This one handy function does a lot of things at once. In a second step, we add the studentised residuals using `rstudent()`.

```
# Create a tibble with some pre-computed stats
m1_outliers <-
  fortify(m1) |>
  as_tibble()

glimpse(m1_outliers)

Rows: 1,188
Columns: 11
$ new_cases <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ movements <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ schools <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ businesses <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ travel <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ .hat <dbl> 0.005159386, 0.005159386, 0.005159386, 0.005159386, 0.00515~
$ .sigma <dbl> 10077.22, 10077.22, 10077.22, 10077.22, 10077.22, ~
$ .cooksdi <dbl> 9.393496e-05, 9.393496e-05, 9.393496e-05, 9.393496e-~05, 9.3~
$ .fitted <dbl> -3023.617, -3023.617, -3023.617, -3023.617, -3023.617, ~
$ .resid <dbl> 3023.617, 3023.617, 3023.617, 3023.617, 3023.617, 3023.617, ~
$ .stdresid <dbl> 0.3009375, 0.3009375, 0.3009375, 0.3009375, 0.3009375, 0.30~
```

```
# Add the externally studentised residuals
m1_outliers <-
  m1_outliers |>
```

```
mutate(.studresid = rstudent(m1))

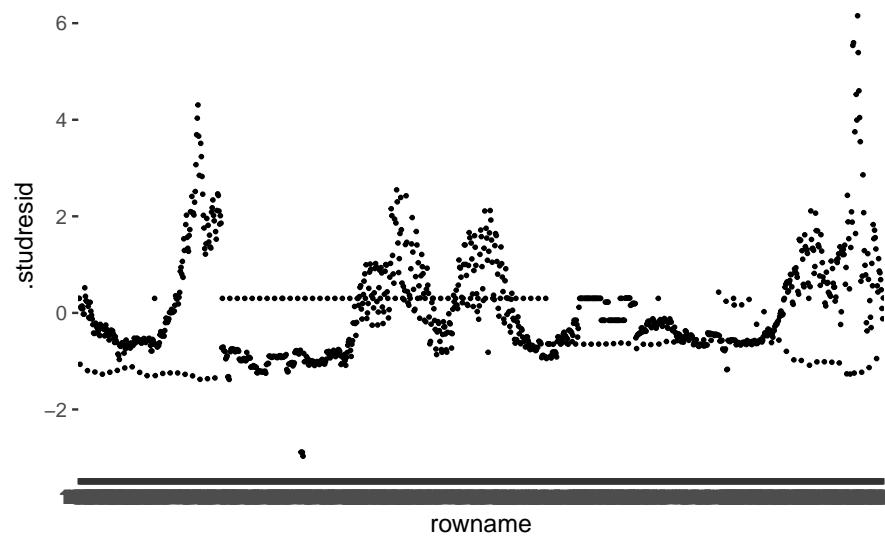
glimpse(m1_outliers)

Rows: 1,188
Columns: 12
$ new_cases <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ movements <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ schools <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ businesses <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ travel <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ .hat <dbl> 0.005159386, 0.005159386, 0.005159386, 0.005159386, 0.00515~
$ .sigma <dbl> 10077.22, 10077.22, 10077.22, 10077.22, 10077.22, 10077.22, ~
$ .cooksdi <dbl> 9.393496e-05, 9.393496e-05, 9.393496e-05, 9.393496e-
05, 9.3~
$ .fitted <dbl> -3023.617, -3023.617, -3023.617, -3023.617, -3023.617, ~
3023.617, -302~
$ .resid <dbl> 3023.617, 3023.617, 3023.617, 3023.617, 3023.617, 3023.617, ~
$ .stdresid <dbl> 0.3009375, 0.3009375, 0.3009375, 0.3009375, 0.3009375, 0.30~
$ .studresid <dbl> 0.3008218, 0.3008218, 0.3008218, 0.3008218, 0.3008218, 0.30~
```

With our dataset ready for plotting, we can do exactly that and see which observations are particularly far away from the rest of our `.studresid` values. To plot each observation separately, we need an id variable for each row. We can quickly add one by using the function `rownames_to_column()`. This way, we can identify each column and also `filter()` out particular rows. You might be able to guess why this will come in handy at a later stage of our outlier analysis (hint: Section 13.2.1.5).

```
# Create an ID column
m1_outliers <- m1_outliers |> rownames_to_column()

m1_outliers |>
  ggplot(aes(x = rowname,
             y = .studresid)) +
  geom_point(size = 0.5)
```



The values of the externally studentised residuals can be positive or negative. All we need to know is which values count as outliers and which ones do not. P. Cohen, West, and Aiken (2014) (p. 401) provides some guidance:

- general: $outlier = \pm 2$
- bigger samples: $outlier = \pm 3$ or ± 3.5 or ± 4

As you can tell, it is a matter of well-informed personal judgement. Our dataset consists of over 1200 observations. As such, the data frame certainly counts as large. We can take a look and see how many outliers we would get for each of the benchmarks.

```
out_detect <-
  m1_outliers |>
  mutate(pm_2 = ifelse(abs(.studresid) > 2, "TRUE", "FALSE"),
         pm_3 = ifelse(abs(.studresid) > 3, "TRUE", "FALSE"),
         pm_35 = ifelse(abs(.studresid) > 3.5, "TRUE", "FALSE"),
         pm_4 = ifelse(abs(.studresid) > 4, "TRUE", "FALSE"))

out_detect |> count(pm_2)

# A tibble: 2 x 2
  pm_2     n
  <chr> <int>
1 FALSE   1132
2 TRUE     56
```

```
out_detect |> count(pm_3)
```

```
# A tibble: 2 × 2
  pm_3      n
  <chr> <int>
1 FALSE    1171
2 TRUE     17
```

```
out_detect |> count(pm_35)
```

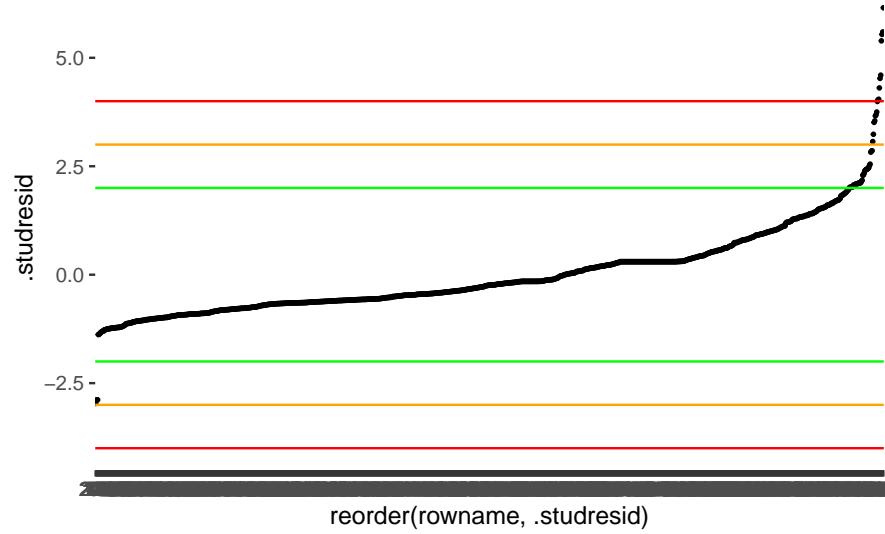
```
# A tibble: 2 × 2
  pm_35     n
  <chr> <int>
1 FALSE   1173
2 TRUE    15
```

```
out_detect |> count(pm_4)
```

```
# A tibble: 2 × 2
  pm_4      n
  <chr> <int>
1 FALSE    1179
2 TRUE     9
```

The results indicate we could have as many as 56 outliers and as little as 9. It becomes apparent that choosing the right threshold is a tricky undertaking. Let's plot the data again to make it easier to read and add some of the thresholds. I skip 3.5 since it is very close to 3. I also reorder the observations (i.e. the x-axis) based on `.studresid` using `reorder()`.

```
m1_outliers |>
  ggplot(aes(x = reorder(rowname, .studresid),
             y = .studresid)) +
  geom_point(size = 0.5) +
  geom_hline(yintercept = c(-2, 2), col = "green") +
  geom_hline(yintercept = c(-3, 3), col = "orange") +
  geom_hline(yintercept = c(-4, 4), col = "red")
```



At this point, it is a matter of choosing the threshold that you feel is most appropriate. More importantly, though, you have to make sure you are transparent in your choices and provide some explanations around your decision-making. For example, a threshold of 2 appears too harsh for my taste and identifies too many observations as outliers. On the other hand, using the orange threshold of 3 seems to capture most observations I would consider an outlier because we can also visually see how the dots start to look less like a straight line and separate more strongly. Besides, P. Cohen, West, and Aiken (2014) also suggests that a threshold of 3 is more suitable for larger datasets. Finally, since we have an ID column (i.e. `rownames`), we can also store our outliers in a separate object to easily reference them later for comparisons with other measures. Again, the purpose of doing this will become evident in Section 13.2.1.5.

```
outliers <-  
  out_detect |>  
  select(rowname, pm_3) |>  
  rename(studresid = pm_3)
```

There are still more diagnostic steps we have to take before deciding which observations we want to remove or deal with in other ways (see also Section 9.6).

13.2.1.2 Outliers in independent variables

To identify outliers in independent variables, we can use *Leverage scores* or the *Mahalanobis distances*. Both are legitimate approaches and can be computed very easily.

For the leverage scores, we can find them already in our `fortify()`-ed dataset `m1_outliers`. They are in the column `.hat`. An outlier is defined by the distance from the average leverage value, i.e. the further the distance of an observation from this average leverage, the more likely we have to classify it as an outlier. The average leverage is computed as follows:

$$\text{average leverage} = \frac{k+1}{n}$$

In this equation, k stands for the number of predictors (i.e. 6) and n for the number of observations (i.e. 1188). Therefore, our average leverage can be computed as follows:

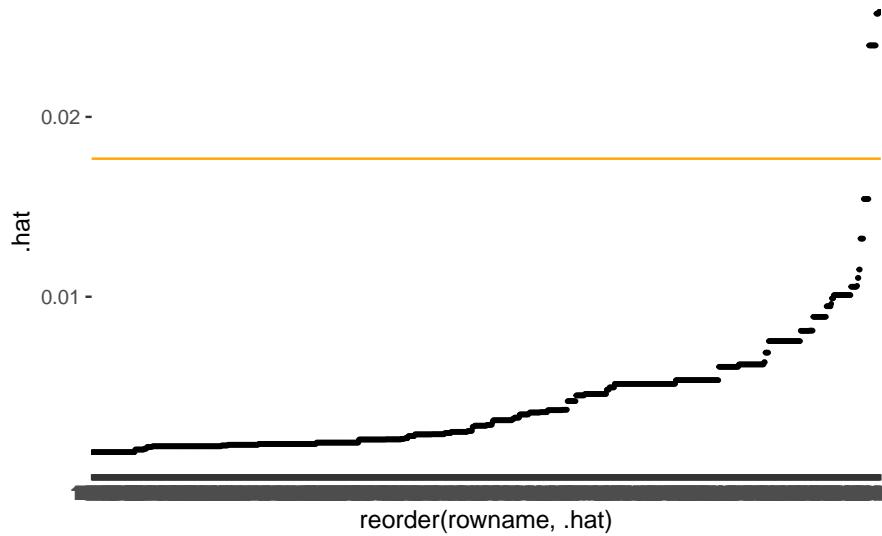
```
(avg_lvg <- (6 + 1) / 1188)
```

```
[1] 0.005892256
```

Similar to before, we find different approaches to setting cut-off points for this indicator. While Hoaglin and Welsch (1978) argue that a distance twice the average counts as an outlier, Stevens (2012) (p. 105) suggests that values three times higher than the average leverage will negatively affect the model. The rationale is the same as for the externally studentised residuals: If the thresholds are too low, we might find ourselves with many observations, which we would have to investigate further. This might not always be possible or even desirable. However, this should not imply that many outliers are not worth checking. Instead, if there are many, one would have to raise questions about the model itself and whether an important variable needs adding to explain a series of observations that appear to be somewhat ‘off’.

Let’s plot the leverages and use Stevens (2012) benchmark to draw our reference line.

```
m1_outliers |>
  ggplot(aes(x = reorder(rownames, .hat),
             y = .hat)) +
  geom_point(size = 0.5) +
  geom_hline(yintercept = 3 * avg_lvg, col = "orange")
```



As before, we want to know which observations fall beyond the threshold.

```
new_outliers <-
  m1_outliers |>
  mutate(avglvg = ifelse(.hat > 3 * avg_lvg, "TRUE", "FALSE")) |>
  select(rowname, avglvg)

# Add new results to our reference list
outliers <- left_join(outliers, new_outliers, by = "rowname")
```

Looking at our reference object `outliers`, you will notice that both methods (`studresid` and `avglvg`) detect different outliers. Thus, the detection of outliers depends on where we look, i.e. dependent variable or independent variable.

The second method I will cover in this section is the Mahalanobis distance. Luckily the `rstatix` package includes a handy function `mahalanobis_distance()` which automatically detects outliers and classifies them for us.

```
mhnbs_outliers <-
  m1_outliers |>
  select(new_cases:travel) |>
  rstatix::mahalanobis_distance() |>
  rownames_to_column() |>
  select(rowname, mahal.dist, is.outlier) |>
  rename(mhnbs = is.outlier)
```

```
# Add new results to our reference list
outliers <- left_join(outliers, mhnbs_outliers, by = "rowname")

glimpse(outliers)

Rows: 1,188
Columns: 5
$ rowname <chr> "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "1~"
$ studresid <chr> "FALSE", "FALSE", "FALSE", "FALSE", "FALSE", "FALSE", "FALS~"
$ avgldv <chr> "FALSE", "FALSE", "FALSE", "FALSE", "FALSE", "FALSE", "FALS~"
$ mahal.dist <dbl> 5.215, 5.215, 5.215, 5.215, 5.215, 5.215, 5.215, 5.2~"
$ mhnbs <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FAL~

# We need to remove mahal.dist because it does not indicate
# whether a value is an outlier #data-cleaning
outliers <- outliers |> select(-mahal.dist)
```

While it is very convenient that this function picks the cut-off point for us, it might be something we would want more control over. As we have learned so far, choosing the ‘right’ cut-off point is essential. Since the values follow a chi-square distribution, we can determine the cut-off points based on the relevant critical value at the chosen p-value. *R* has a function that allows finding the critical value for our model, i.e. `qchisq()`.

```
(mhnbs_th <- qchisq(p = 0.05,
                     df = 4,
                     lower.tail = FALSE))
```

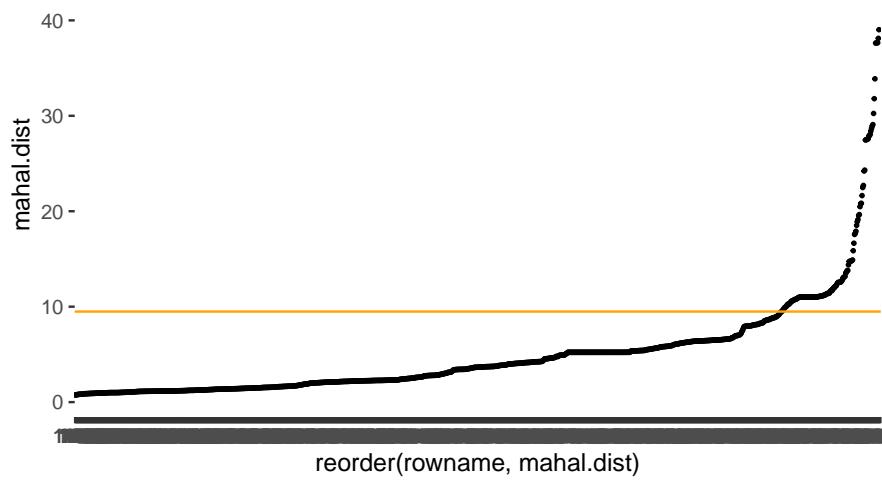
```
[1] 9.487729
```

The p-value reflects the probability we are willing to accept that our result is significant/not significant (remember Type I error in Chapter 12). The value `df` refers to the degrees of freedom, which relates to the number of independent variables, i.e. 4. Thus, it is fairly simple to identify a cut-off point yourself by choosing the p-value you consider most appropriate. The function `mahalanobis_distance()` assumes $p = 0.01$, which is certainly a good but strict choice.

If we want to plot outliers as before, we can reuse our code from above and replace it with the relevant new variables.

```
mhnbs_outliers |>
  ggplot(aes(x = reorder(rowname, mahal.dist),
```

```
y = mahal.dist)) +
geom_point(size = 0.5) +
geom_hline(yintercept = mhnbs_th, col = "orange")
```



Looking at our outliers, we notice that the Mahalanobis distance identifies more outliers than the leverage, but the same ones.

```
outliers |> filter(avglvg == "TRUE")
```

```
# A tibble: 18 x 4
  rowname studresid avglvг mhnbs
  <chr>    <chr>     <chr>   <lgl>
1 224      FALSE      TRUE    TRUE
2 225      FALSE      TRUE    TRUE
3 226      FALSE      TRUE    TRUE
4 227      FALSE      TRUE    TRUE
5 969      FALSE      TRUE    TRUE
6 970      FALSE      TRUE    TRUE
7 971      FALSE      TRUE    TRUE
8 972      FALSE      TRUE    TRUE
9 973      FALSE      TRUE    TRUE
10 974     FALSE      TRUE    TRUE
11 975     FALSE      TRUE    TRUE
12 976     FALSE      TRUE    TRUE
13 977     FALSE      TRUE    TRUE
```

```

14 978      FALSE    TRUE    TRUE
15 979      FALSE    TRUE    TRUE
16 980      FALSE    TRUE    TRUE
17 981      FALSE    TRUE    TRUE
18 982      FALSE    TRUE    TRUE

```

Thus, whether you need to use both approaches for the same study is questionable and likely redundant. Still, in a few edge cases, you might want to double-check the results, especially when you feel uncertain which observations should be dealt with later. But, of course, it does not take much time to consider both options.

Besides looking at each side of the regression separately, we might also consider whether removing an observation significantly affects all variables. This is done with outlier detection diagnostics which consider the entire model.

13.2.1.3 Outlier detection considering the entire model: Global measures

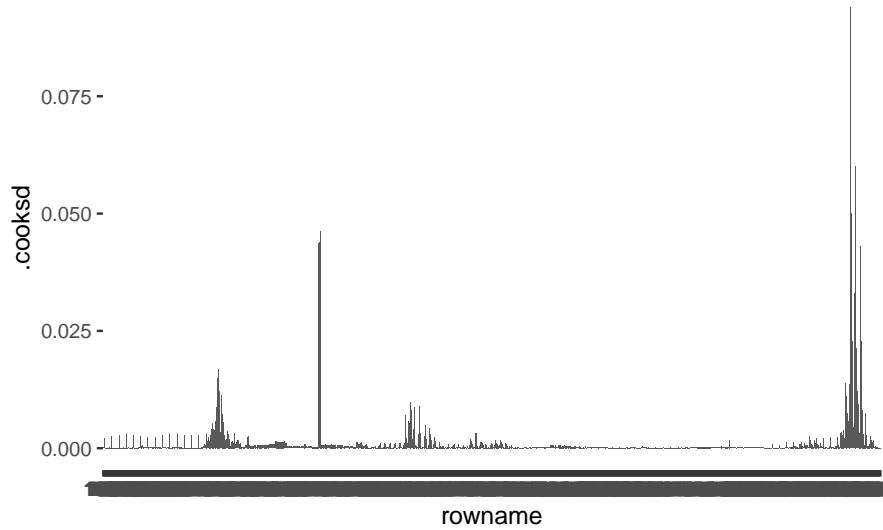
To assess the global impact of outliers on the entire regression model, *Cook's d* (Cook and Weisberg 1982) is a popular method in the Social Sciences. It measures to which extend a single observation can affect the predictive power of our model to explain all other observations. Obviously, we do not wish to keep observations that make predicting most of the other observations more challenging. However, as shown in Table 13.1, there are different approaches to this, but some are somewhat redundant. For example, P. Cohen, West, and Aiken (2014) highlight that DDFITS and Cook's d are 'interchangeable statistics' (p. 404). Thus, there is no point in demonstrating both since they function similarly. Your decision might be swayed by the preferences of a publisher, editor, lecturer, supervisor or reviewer. Here, I will focus on *Cook's d* since it is the approach I see most frequently used in my field. By all means, feel encouraged to go the extra mile and perform the same steps for the DDFITS.

The good news, `fortify()` automatically added `.cooksdi` to our dataset `m1_outliers`. Thus, we can immediately compute whether outliers exist and inspect them visually as we did before. A promising starting point to find outliers via the Cook's D is to plot its distribution.

```

m1_outliers |>
  ggplot(aes(x = rowname,
             y = .cooksdi)) +
  geom_col()

```



Inspecting this barplot, we can tell that some observations have much higher .cooksD values than any other observation. Once again, we first need to decide on a benchmark to determine whether we can consider these values as outliers. If we follow Cook and Weisberg (1982), values that are higher than 1 (i.e. $d > 1$) require reviewing. Looking at our plot, none of the observations reaches 1, and we need not investigate outliers. Alternatively, P. Cohen, West, and Aiken (2014) suggest that other benchmarks are also worth considering, for example, based on the critical value of an F distribution, which we can determine with the function `qf()`. This requires us to determine two degrees of freedom (df_1 and df_2) and a p-value. To determine these values, P. Cohen, West, and Aiken (2014) suggests $p = 0.5$ and the following formulas to determine the correct df :

$$df_1 = k + 1$$

$$df_2 = n - k - 1$$

Like the critical value for [average leverage](#), k reflects the number of predictors, and n refers to the sample size. Thus, we can determine the critical value, and therefore our cut-off point as follows:

```
qf(p = 0.5,
  df1 = 4 + 1,
  df2 = 1188 - 6 - 1)
```

```
[1] 0.8707902
```

This score would also imply that we have no particular outliers to

consider, because the highest value in `.cooksrd` is 0.09 computed via `max(m1_outliers$.cooksrd)`.

We might be led to believe that our work is done here, but P. Cohen, West, and Aiken (2014) recommends that any larger deviation is worth inspecting. We do notice that relative to other observations, some cases appear extreme. Ideally, one would further investigate these cases and compare the regression results by removing such extreme cases iteratively. This way, one can assess whether the extreme observations genuinely affect the overall estimates of our model. This would imply repeating steps we already covered earlier when performing multiple regressions with and without outliers. Thus, I will forgo this step here.

In the last chapter about outliers (Section 13.2.1.5), we will rerun the regression without outliers to see how this affects our model estimates. However, before we can do this, we have to cover one more method of detecting outliers.

13.2.1.4 Outlier detection considering the entire model: Specific measures

While Cook's d helps us identify outliers that affect the quality of the entire model, there is also a way to investigate how outliers affect specific predictors. This is achieved with DFBETAS, which, similar to previous methods, assesses the impact of outliers by removing them and measures the impact of such removal on other parts of the regression model.

The function `dfbetas()` takes our model `m1` and returns the statistics for each predictor in the model. Thus, we do not receive a single score for each observation but multiple for each specific predictor, i.e. each specific measure.

```
m1_dfbetas <-
  dfbetas(m1) |>
  as_tibble() |>
  rownames_to_column() # convert to tibble for convenience
                        # add our rownames

glimpse(m1_dfbetas)
```

Rows: 1,188
 Columns: 6
 \$ rowname <chr> "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11",~
 \$ `(Intercept)` <dbl> 0.02166365, 0.02166365, 0.02166365, 0.02166365, 0.021663~
 \$ movements <dbl> -0.0008908373, -0.0008908373, -0.0008908373, -0.00089083~
 \$ schools <dbl> -0.01412839, -0.01412839, -0.01412839, -0.01412839, -0.01412839, -0.0~

```
$ businesses      <dbl> -0.002957931, -0.002957931, -0.002957931, -
0.002957931, ~
$ travel         <dbl> -0.005546516, -0.005546516, -0.005546516, -
0.005546516, ~
```

All we have to do now is to compare the scores against a benchmark for each variable in this dataset, and we know which observations substantially affect one of the predictors. P. Cohen, West, and Aiken (2014) provides us with the following recommendations for suitable thresholds:

- small or moderate datasets: $DFBETAS > \pm 1$
- large datasets: $DFBETAS > \pm \frac{2}{\sqrt{(n)}}$

Since our dataset falls rather into the ‘large’ camp, we should choose the second option. Again, n stands for the sample size. Let’s create an object to store this value.

```
(dfbetas_th <- 2 / sqrt(1188))
```

```
[1] 0.05802589
```

For demonstration purposes, I will pick `movements` to check for outliers. If this was a proper analysis for a project, you would have to compare this indicator against each variable separately. As the benchmarks indicate, the DFBETAS value can be positive or negative. So, when we compare the calculated values with it, we can look at the absolute value, i.e. use `abs()`, which turns all values positive. This makes it much easier to compare observations against a threshold and we could do this for any assessment of outliers against a benchmark.

```
# Check whether values exceed the threshold
dfbetas_check <-
  m1_dfbetas |>
  mutate(dfbetas_movements = ifelse(abs(movements) > dfbetas_th,
                                     "TRUE",
                                     "FALSE")) |>
  select(rowname, dfbetas_movements)

# Add result to our outliers object
outliers <-
  outliers |>
  left_join(dfbetas_check, by = "rowname")

glimpse(outliers)
```

Rows: 1,188

```
Columns: 5
$ rowname      <chr> "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "~"
$ studresid    <chr> "FALSE", "FALSE", "FALSE", "FALSE", "FALSE", "FALSE"~
$ avglvg       <chr> "FALSE", "FALSE", "FALSE", "FALSE", "FALSE", "FALSE"~
$ mhnbs        <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FAL~
$ dfbetas_movements <chr> "FALSE", "FALSE", "FALSE", "FALSE", "FALSE", "FALSE"~

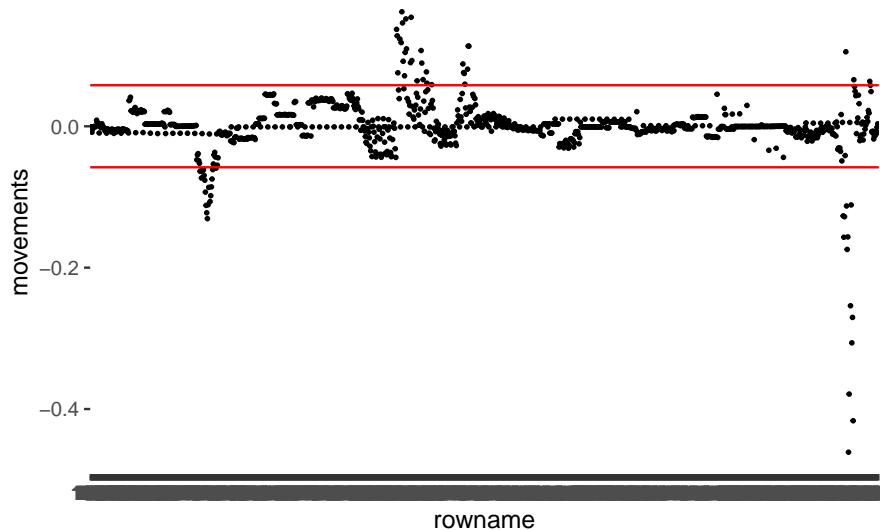
# some housekeeping, i.e. making all columns <lgl> except for rowname
outliers <-
  outliers |>
  mutate(rowname = as_factor(rowname)) |>
  mutate_if(is.character, as.logical)

glimpse(outliers)

Rows: 1,188
Columns: 5
$ rowname      <fct> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 1~
$ studresid    <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FAL~
$ avglvg       <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FAL~
$ mhnbs        <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FAL~
$ dfbetas_movements <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FAL~
```

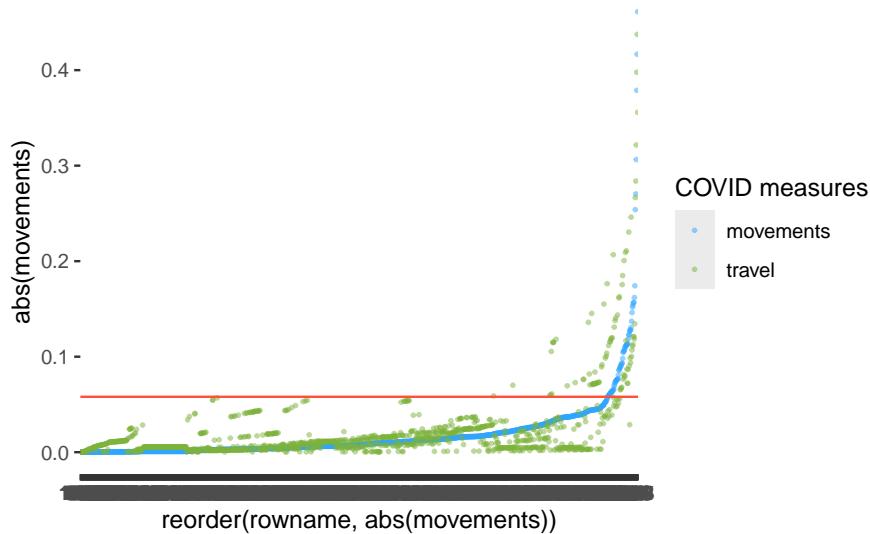
Of course, we can also visualise the outliers as we did before.

```
m1_dfbetas |>
  ggplot(aes(x = rowname,
             y = movements)) +
  geom_point(size = 0.5) +
  geom_hline(yintercept = dfbetas_th, col = "red") +
  geom_hline(yintercept = -dfbetas_th, col = "red")
```



I want to take this opportunity to show that sometimes we can make visualisations even simpler. Remember, we used `abs()` to make all values positive? We can apply the same principle here. This way, we only need one line to indicate the threshold. In addition, we could also plot multiple variables at once. Instead of defining the `aes()` inside the `ggplot()` function, we can define it independently for each `geom_point()`. What do you think about the following version of the same plot?

```
m1_dfbetas |>
  ggplot() +
  geom_point(aes(x = reorder(rowname, abs(movements)),
                 y = abs(movements),
                 col = "movements"),
             size = 0.5,
             alpha = 0.5) +
  geom_point(aes(x = rowname,
                 y = abs(travel),
                 col = "travel"),
             size = 0.5,
             alpha = 0.5) +
  geom_hline(yintercept = dfbetas_th, col = "#FF503A") +
  scale_color_manual(values = c("movements" = "#2EA5FF",
                               "travel" = "#7DB33B")) +
  # Label the legend appropriately
  labs(col = "COVID measures")
```



After `reorder()`ing the variable `movements` and plotting `travel` as well, we notice that there might be slightly fewer outliers for `movements` than for `travel`. Thus, some observations affect some predictors more strongly than others.

13.2.1.5 Reviewing the outliers

After all this hard work and what turned out to be a very lengthy chapter, we finally arrive at the point where we check which observations we might wish to remove or handle in some shape or form. First, we want to know which observations are affected. Therefore we need to review our reference data in the object `outliers`. We want to see all observations identified by one or more diagnostic tools as an outlier. There are two ways to achieve this. First, we can use what we have learned so far and `filter()` each column for the value TRUE. This is the hard way of doing it, and if you have many more columns, this will take a little while and potentially drive you insane. The easy (and clever) way of filtering across multiple columns can be achieved by turning multiple columns into a single column. In the `tidyverse`, this is called `pivot_longer()` and we performed it earlier in Section 11.3.2.1. Thus, it might seem less complicated than expected. Let's do this step-by-step as we did for `pivot_wider()` in Section 11.4. Currently, our data has five columns, of which four are different measures to detect outliers. Our goal is to create a table that has only three columns:

- `rowname`, which we keep unchanged because it is the ID that identifies each observation in our data
- `outlier_measure`, which is the variable that indicates which measure was used to find an outlier.

- `is.outlier`, which contains the values from the tibble, i.e. the cell values of `TRUE` and `FALSE`.

Here is an example of what we want to achieve. Imagine we have a smaller dataset, which contains three columns, two of which are outlier detection measures, i.e. `studresid` and `mhbns`:

```
data <- tribble(
  ~rowname,    ~ studresid,     ~mhbns,
  "1",        FALSE,          FALSE,
  "2",        TRUE,           FALSE,
  "3",        TRUE,           TRUE
)
```

```
data
```

```
# A tibble: 3 x 3
  rowname studresid mhbns
  <chr>    <lgl>     <lgl>
1 1        FALSE      FALSE
2 2        TRUE       FALSE
3 3        TRUE       TRUE
```

To filter for outliers using a single column/variable, we need to rearrange the values in `studresid` and `mhbns`. We currently have one column which captures the `rowname`, and two other columns which are both outlier detection methods. Thus, we could argue that `studresid` and `mhbns` measure the same, but with different methods. Therefore we can combine them into a factor, e.g. `is.outlier`, with two levels reflecting each outlier detection method. At the moment, we have two values recorded for every row in this dataset. For example, the first observation has the values `studresid == FALSE` and `mhbns == FALSE`. If we want to combine these two observations into a single column, we need to have two rows with the `rowname 1` to ensure that each row still only contains one observation, i.e. a tidy dataset. Therefore, we end up with more rows than our original dataset, hence, a ‘longer’ dataset. Here is how we can do this automatically:

```
data_long <-
  data |>
  pivot_longer(cols = !rowname,
               names_to = "outlier_measure",
               values_to = "is.outlier")

data_long
```

```
# A tibble: 6 x 3
  rowname outlier_measure is.outlier
  <chr>    <chr>        <lgl>
1 1       studresid     FALSE
2 1       mhbns         FALSE
3 2       studresid     TRUE
4 2       mhbns         FALSE
5 3       studresid     TRUE
6 3       mhbns         TRUE
```

Instead of three rows, we have six, and all the outlier detection values (i.e. all `TRUE` and `FALSE` values) are now in one column, i.e. `is.outlier`. However, what exactly did just happen in this line of code? In light of what we specified above, we did three things inside the function `pivot_longer()`:

- We excluded `rownames` from being pivoted, i.e. `cols = !rowname`.
- We specified a column where all the column names go to, i.e. `names_to = "outlier_measure"`.
- We defined a column where all cell values should be listed, i.e. `values_to = "is.outlier"`.

From here, it is straightforward to `count()` the number of `is.outlier` per `rowname` and `filter()` out those that return a value of `TRUE`. Everything we learned is now easily applicable without overly complicated functions or many repetitions. As mentioned earlier, pivoting datasets is an essential data wrangling skill, not matter which software you prefer.

```
data_long |>
  count(rowname, is.outlier) |>
  filter(is.outlier == "TRUE")
```

```
# A tibble: 2 x 3
  rowname is.outlier     n
  <chr>    <lgl>      <int>
1 2       TRUE           1
2 3       TRUE           2
```

Et voilà. We now counted the number of times an observation was detected considering both measures. This is scalable to more than two measures. Thus, we can apply it to our data as well.

```
outliers_true <-
  outliers |>
  pivot_longer(!rowname,
               names_to = "measure",
```

```
values_to = "is.outlier") |>
count(rownames, is.outlier) |>
filter(is.outlier == "TRUE")

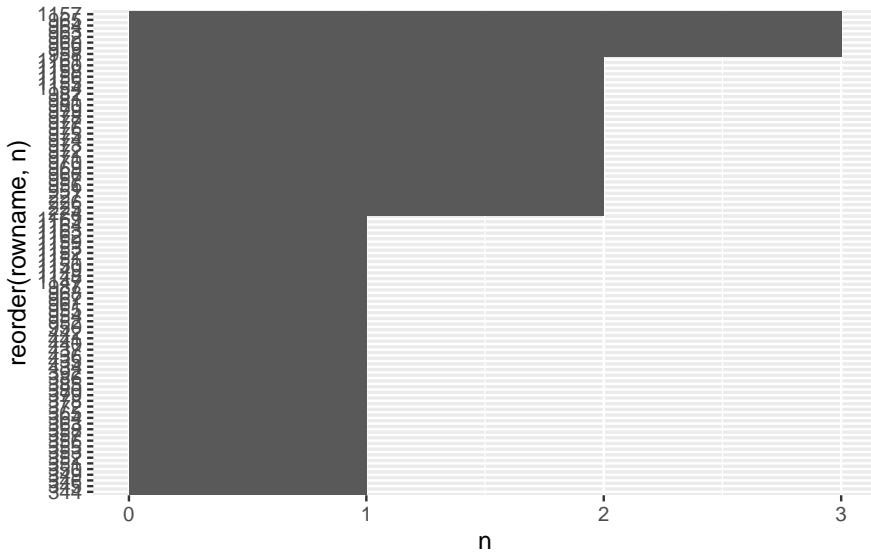
outliers_true
```

A tibble: 85 x 3
 rowname is.outlier n
 <fct> <lgl> <int>
1 224 TRUE 2
2 225 TRUE 2
3 226 TRUE 2
4 227 TRUE 2
5 344 TRUE 1
6 345 TRUE 1
7 346 TRUE 1
8 349 TRUE 1
9 350 TRUE 1
10 351 TRUE 1
i 75 more rows

The result is a `tibble` which tells us that we identified 85 distinct outliers. This might sound like a lot, but we also have to remember that our dataset consists of 1188 observations, i.e. 7% of our data are outliers.

Since the output is relatively long, we might want to plot the outcome to get an idea of the bigger picture.

```
outliers_true |>
ggplot(aes(x = reorder(rownames, n),
           y = n)) +
geom_col() +
coord_flip()
```



A few observations were detected by three out of the four methods we used to detect outliers. There are some more which were detected by two different methods. However, there are also more than half of our outliers which were only detected by one method.

As highlighted in Section 9.6, there are many ways we can go about outliers. To keep our analysis simple, I intend to remove those observations detected by multiple methods rather than only by one. Whether this approach is genuinely appropriate is a matter of further investigation. I take a very pragmatic approach with our sample data. However, it is worth reviewing the options available to transform data, as covered by Field (2013) (p.203). Data transformation can help to deal with outliers instead of removing or imputing them. Still, data transformation comes with severe drawbacks and in most cases, using a bootstrapping technique to account for violations of parametric conditions is often more advisable. Bootstrapping refers to the process of randomly resampling data from your dataset and rerun the same test multiple times, e.g. 2000 times. Thus, there will be some samples that do not include the outliers. The process is somewhat similar to multiple imputation (see Section 7.7.3). Still, instead of estimating a specific value in our dataset, we estimate statistical parameters, e.g. confidence intervals or regression coefficients (i.e. estimates). The package `rsample` allows implementing bootstrapping straightforwardly. However, bootstrapping comes at the cost of lower power because we are running the test so many times.

For now, I settle on removing outliers entirely. The easiest way to remove these outliers is to combine our `outliers` dataset with our regression (`m1_outliers`).

Now you will notice that having `rowname` as an ID allows us to match the values of each table to each other. Otherwise, this would not be possible.

```
# Include outliers which were detected by multiple methods
outliers_select <-
  outliers_true |>
  filter(n > 1)

# Keep only columns which are NOT included in outliers_select
m1_no_outliers <- anti_join(m1_outliers, outliers_select, by = "rowname")

m1_no_outliers
```

A tibble: 1,152 x 13

	rowname	new_cases	movements	schools	businesses	travel	.hat	.sigma	.cooksdf	<dbl>							
1	1	0	0	0	0	0	0.00516	10077.	9.39e-5	0	0	0	0	0	0	0	0
2	2	0	0	0	0	0	0.00516	10077.	9.39e-5	0	0	0	0	0	0	0	0
3	3	0	0	0	0	0	0.00516	10077.	9.39e-5	0	0	0	0	0	0	0	0
4	4	0	0	0	0	0	0.00516	10077.	9.39e-5	0	0	0	0	0	0	0	0
5	5	0	0	0	0	0	0.00516	10077.	9.39e-5	0	0	0	0	0	0	0	0
6	6	0	0	0	0	0	0.00516	10077.	9.39e-5	0	0	0	0	0	0	0	0
7	7	0	0	0	0	0	0.00516	10077.	9.39e-5	0	0	0	0	0	0	0	0
8	8	0	0	0	0	0	0.00516	10077.	9.39e-5	0	0	0	0	0	0	0	0
9	9	0	0	0	0	0	0.00516	10077.	9.39e-5	0	0	0	0	0	0	0	0
10	10	0	0	0	0	0	0.00516	10077.	9.39e-5	0	0	0	0	0	0	0	0
		# i 1,142 more rows															
		# i 4 more variables: .fitted <dbl>, .resid <dbl>, .stdresid <dbl>,															
		# .studresid <dbl>															

The function `anti_join()` does exactly what the name implies. It takes the first data frame (i.e. `m1_outliers`) and removes values that are included in the second data frame (i.e. `outliers_select()`). This is tremendously helpful when performing such complex outlier detection and removing them all in one go.

As the final step, we want to compare how the removal of outliers affected our model. Ideally, we managed to improve the regression. Thus, we compare our old model `m1` with a new model `m2`.

```
# Original regression
m1 <- lm(new_cases ~ movements + schools + businesses + travel,
         data = covid_uk_ger)

# Regression with outliers removed
```

```
m2 <- lm(new_cases ~ movements + schools + businesses + travel,
  data = m1_no_outliers)

# Compare the parameters between models
# I added some reformatting, because
# I prefer a regular tibble over a pre-formatted
# table. Technically you only need the first line.
parameters::compare_parameters(m1, m2) |>
  as_tibble() |>
  select(Parameter, Coefficient.m1, p.m1, Coefficient.m2, p.m2) |>
  mutate(across(where(is.double), round, 3))
```

```
# A tibble: 5 x 5
  Parameter   Coefficient.m1   p.m1   Coefficient.m2   p.m2
  <chr>          <dbl>    <dbl>          <dbl>    <dbl>
1 (Intercept) -3024.        0       -1543.    0.031
2 movements     -64.0       0       -15.8    0.174
3 schools       382.        0       320.     0
4 businesses     90.5       0       82.7     0
5 travel         45.7       0      -1.27    0.875
```

```
# Compare the performance between models
performance::compare_performance(m1, m2)
```

```
# Comparison of Model Performance Indices
```

Name	Model	AIC (weights)	AICc (weights)	BIC (weights)	R2	R2 (adj.)	RMSE	Sigma
m1	lm	25279.5 (<.001)	25279.6 (<.001)	25310.0 (<.001)	0.229	0.227	10052.123	10073.3
m2	lm	24099.6 (>.999)	24099.7 (>.999)	24129.9 (>.999)	0.179	0.177	8398.110	8416.3

If you look at R₂ (adj.) you might feel a sense of disappointment. How does the model explain less variance? If we consider AIC, BIC and RMSE, we improved the model because their values are lower for m2. However, it seems that the outliers affected R₂, which means they inflated this model indicator. In other words, our regression explains less than we hoped for. However, we can be more confident that the predictions of this model will be more accurate.

If we inspect the estimates more closely, we also notice that movements (where we removed outliers) and travel are not significant anymore (i.e. p.m2 > 0.05). We should remove these from our model since they do not help explain the variance of new_cases.

```
m3 <- lm(new_cases ~ schools + businesses,
         data = m1_no_outliers)

parameters::model_parameters(m3) |>
  as_tibble() |>
  select(Parameter, Coefficient, p) |>
  mutate(across(where(is.double), round, 3))

# A tibble: 3 x 3
  Parameter   Coefficient     p
  <chr>        <dbl> <dbl>
1 (Intercept) -1664.  0.016
2 schools       307.  0
3 businesses    78.9  0

performance::model_performance(m3)

# Indices of model performance

AIC      |     AICc     |      BIC     |      R2 | R2 (adj.) |      RMSE     |      Sigma
-----+-----+-----+-----+-----+-----+-----+-----+
24098.209 | 24098.243 | 24118.406 | 0.178 | 0.176 | 8407.515 | 8418.483
```

Similar to before, after removing the insignificant predictors, we end up with an equally good model (in terms of *adjusted R²*, but we need fewer variables to explain the same amount of variance in `new_cases`. Our final model `m3` is a parsimonious and less complex model.

13.2.2 Standardised beta (β) vs. unstandardised beta (B)

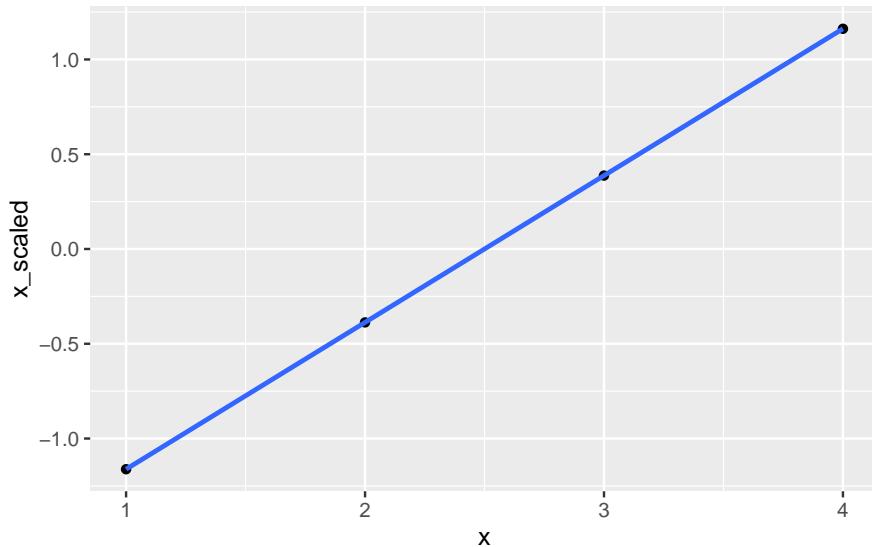
In multiple regressions, we often use variables that are measured in different ways and which use different measurement units, e.g. currency, age, etc. Thus, it is sometimes fairly difficult to interpret regression coefficients without standardising the measurement units. Standardised scores (also called *z-scores*) imply that each variable has a distribution where the mean equals 0 and the standard deviation equals 1. By transforming our variables to z-scores, they become ‘unit free’ (P. Cohen, West, and Aiken 2014) (p. 25) and are easier to interpret. Here is an example with a simplified dataset:

```
# Create some data
data <- tibble(x = c(1, 2, 3, 4))
```

```
# Compute z-scores
data <-  
  data |>  
  mutate(x_scaled = scale(x))  
  
# Compare the mean and sd for each variable  
# and put it into a nice table
tibble(  
  variable = c("x", "x_scaled"),  
  mean = c(mean(data$x), mean(data$x_scaled)),  
  sd = c(sd(data$x), sd(data$x_scaled))  
)  
  
# A tibble: 2 x 3  
  variable   mean    sd  
  <chr>     <dbl> <dbl>  
1 x         2.5    1.29  
2 x_scaled  0      1
```

If you feel that my choice of values has likely affected the outcome, please feel free to change the values for x in this code chunk to whatever you like. The results for x_scaled will always remain the same. There is something else that remains the same: the actual distribution. Because we are only rescaling our data, we are not affecting the differences between these scores. We can show this in a scatterplot and by running a correlation. Both will show that these variables are perfectly correlated. Therefore, our transformations did not affect the relative relationship of values to each other.

```
data |>  
  ggplot(aes(x = x,  
             y = x_scaled)) +  
  geom_point() +  
  geom_smooth(method = "lm",  
              formula = y ~ x,  
              se = FALSE)
```



```
correlation::correlation(data)
```

```
# Correlation Matrix (pearson-method)
```

Parameter1	Parameter2	r	95% CI	t(2)	p
x	x_scaled	1.00	[1.00, 1.00]	Inf	< .001***

p-value adjustment method: Holm (1979)

Observations: 4

It is not a bad idea for regressions to report both unstandardised (B) and standardised (β) values for each predictor, because comparing predictors based on standardised β s only could lead to misleading interpretations as the differences in coefficients could not relate to the true effect size but differences in standard deviations. Let's look at an example where this issues becomes more evident. Consider the `swm` dataset which contains the variables `salary`, `work_experience` and `mindfulness`. We are curious to know to which extend `work_experience` and `mindfulness` predict `salary`. In a first step, we run the model with scaled variables which returns standardised β coefficients.

```
# Multiple regression model with scaled variables
model <- lm(scale(salary) ~ scale(work_experience) + scale(mindfulness),
            data = swm)
```

```
summary(model)
```

Call:
`lm(formula = scale(salary) ~ scale(work_experience) + scale(mindfulness),
 data = swm)`

Residuals:

Min	1Q	Median	3Q	Max
-1.72465	-0.36714	0.05038	0.41489	2.05082

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	-3.128e-16	6.244e-02	0.000	1
scale(work_experience)	6.451e-01	6.337e-02	10.179	< 2e-16 ***
scale(mindfulness)	5.475e-01	6.337e-02	8.639	1.16e-13 ***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.6244 on 97 degrees of freedom
 Multiple R-squared: 0.6179, Adjusted R-squared: 0.6101
 F-statistic: 78.45 on 2 and 97 DF, p-value: < 2.2e-16

Considering the outcome, it would seem that both variables are good predictors for salary and their Estimates are almost the same. Thus, one could be fooled into thinking that they are equally important. We can repeat the analysis but obtain unstandardised β s.

```
# Multiple regression model with scaled variables  

model <- lm(salary ~ work_experience + mindfulness,  

  data = swm)  
  

summary(model)
```

Call:
`lm(formula = salary ~ work_experience + mindfulness, data = swm)`

Residuals:

Min	1Q	Median	3Q	Max
-29.739	-6.331	0.869	7.154	35.363

Coefficients:

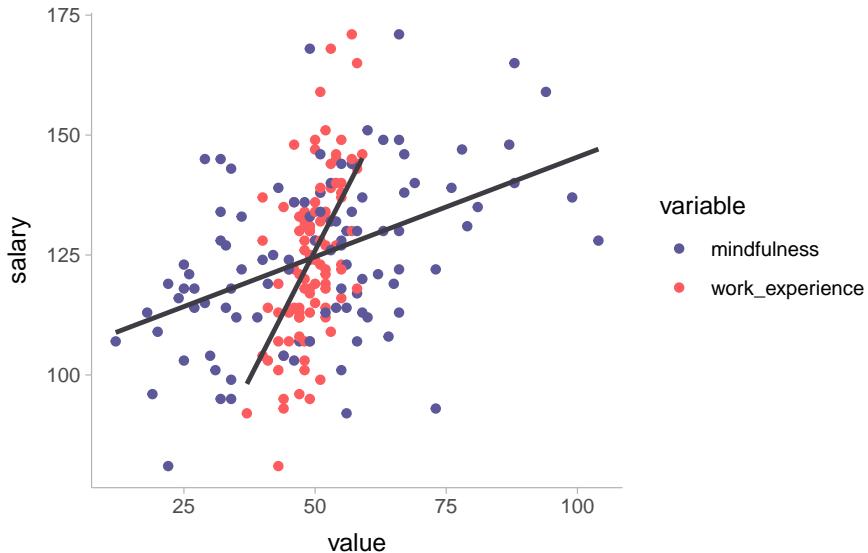
	Estimate	Std. Error	t value	Pr(> t)
--	----------	------------	---------	----------

```
(Intercept) -20.15294 12.56370 -1.604 0.112
work_experience 2.42379 0.23811 10.179 < 2e-16 ***
mindfulness 0.49650 0.05747 8.639 1.16e-13 ***
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 10.77 on 97 degrees of freedom
Multiple R-squared: 0.6179, Adjusted R-squared: 0.6101
F-statistic: 78.45 on 2 and 97 DF, p-value: < 2.2e-16
```

This time the Estimates are quite different from each other, i.e. `work_experience` = 2.42 and `mindfulness` = 0.50. Considering these results it becomes clear that `work_experience` has a much stronger impact on salary than `mindfulness`. We can further confirm this by graphing the regression lines for each variable separately.

```
swm |>
  pivot_longer(cols = c(work_experience, mindfulness),
               values_to = "value",
               names_to = "variable") |>
  ggplot(aes(x = value,
             y = salary,
             col = variable,
             group = variable)) +
  geom_point() +
  geom_smooth(method = "lm",
              formula = y ~ x,
              se = FALSE,
              col = "#3e3c42") +
  # Choose custom colours for the dots
  scale_color_manual(values = c("#5c589a", "#fc5c5d")) +
  ggdist::theme_ggdist()
```



We can see that while `work_experience` is more centralised around its regression line, `mindfulness` is much more spread out across the x-axis. In other words, the standard deviation of `mindfulness` is much larger.

```
# Comparing the standard deviation
swm |>
  summarise(work_xp_sd = sd(work_experience),
            mindfulness_sd = sd(mindfulness)
  )

# A tibble: 1 × 2
  work_xp_sd  mindfulness_sd
  <dbl>          <dbl>
1     4.59        19.0
```

We can also see that the slope of the regression line is different, indicating that a small change in `work_experience` has a much greater impact on `salary` than `mindfulness`. Thus, we need to be very careful when interpreting standardised coefficients. Of course, when in doubt (or simply confused), a data visualisation like this one can provide more certainty about the relationship of variables and their strength.

Unfortunately, this is not the end of the story because ‘standardisation’ can mean different things. We can generate standardised β after the fact (post-hoc) or decide to ‘refit’ our regression with standardised predictors, i.e. before we run the regression. Both options are appropriate, but the scores will differ slightly. However, the main idea remains the same: We want to interpret differ-

ent regression coefficients based on their standard deviations and not their absolute scores. In R, many different packages offer standardised estimates. One package I particularly recommend is `parameters`. It provides various options for returning our estimates from a linear model by specifying the `standardize` argument (see detailed documentation⁷). Of course, you can also opt to scale the variables by hand, using the `scale()` function, as we did in the previous examples. All three approaches are shown in the following code chunk.

```
# Scaling 'by hand'
m3_scaled <- lm(scale(new_cases) ~
                  scale(schools) +
                  scale(businesses),
                  data = m1_no_outliers)

parameters::model_parameters(m3_scaled)
```

Parameter	Coefficient	SE	95% CI	t(1149)	p
<hr/>					
(Intercept)	2.00e-15	0.03	[-0.05, 0.05]	7.49e-14	> .999
schools	0.29	0.03	[0.23, 0.34]	9.90	< .001
businesses	0.21	0.03	[0.16, 0.27]	7.38	< .001

Uncertainty intervals (equal-tailed) and p-values (two-tailed) computed using a Wald t-distribution approximation.

```
# Scaling using 'parameters' package with refit
# This is equivalent to scaling 'by hand'
parameters::model_parameters(m3, standardize = "refit")
```

Parameter	Coefficient	SE	95% CI	t(1149)	p
<hr/>					
(Intercept)	-4.99e-15	0.03	[-0.05, 0.05]	-1.87e-13	> .999
schools	0.29	0.03	[0.23, 0.34]	9.90	< .001
businesses	0.21	0.03	[0.16, 0.27]	7.38	< .001

Uncertainty intervals (equal-tailed) and p-values (two-tailed) computed using a Wald t-distribution approximation.

```
# Scaling using `parameters` package without refitting the model
parameters::model_parameters(m3, standardize = "posthoc")
```

⁷https://easystats.github.io/parameters/reference/model_parameters.default.html

Parameter	Std. Coef.	SE	95% CI	t(1149)	p
<hr/>					
(Intercept)	0.00	0.00	[0.00, 0.00]	-2.41	0.016
schools	0.29	0.03	[0.23, 0.34]	9.90	< .001
businesses	0.21	0.03	[0.16, 0.27]	7.38	< .001

Uncertainty intervals (equal-tailed) and p-values (two-tailed) computed using a Wald t-distribution approximation.

I prefer to standardise using the `parameters` package because it makes my life easier, and I need to write less code. I can also easily compare results before scaling my variables. The choice is yours, of course. Also, I tend to use `standardize = refit` in most regressions, since it is more in line with what most Social Scientists seem to report in their publications.

13.2.3 Multicollinearity: The dilemma of highly correlated independent variables

We finished fitting our model, and we are likely exhausted but also happy that we accomplished something. However, we are not yet done with our analysis. One of the essential post-tests for multiple regressions is a test for *multicollinearity* or sometimes referred to as *collinearity*. The phenomenon of multicollinearity defines a situation in which some of our independent variables can be explained by other independent variables in our regression model. In other words, there exists a substantial correlation (a linear relationship) between two or more independent variables. However, it is crucial to not confuse this with correlations between a dependent variable and independent variables. Remember, a regression reflects the linear relationship between the predictor variables and the outcome variable. As such, we do hope to find a correlation between these variables, just not among the independent variables.

If we have evidence that multicollinearity exists in our data, we face some problems (P. Cohen, West, and Aiken 2014; Field 2013; Grandstrand 2004):

- *Unstable regression Coefficients*: We cannot trust our regression coefficients, i.e. our estimates (β), because the standard errors could be inflated and therefore affect statistical significance.
- *Ambiguity in variable importance*: Since two independent variables can explain the same variance, it is unclear which one is important since both can be easily interchanged without affecting the model much.
- *Challenges in hypothesis testing*: Due to inflated standard errors and unstable estimates, it can be challenging to conduct reliable hypothesis tests, leading to potential confusion about the importance of predictors.

- *Underestimation of model variance*: We likely underestimate the variance our model can explain, i.e. our R^2 .
- *Increased Type II errors*: We produce more Type II errors, i.e. we likely reject significant predictors due to statistical insignificance because of the potentially inflated standard errors.

The *Variance Inflation Factor (VIF)* and its little sibling *Tolerance* are methods to identify issues of multicollinearity. Compared to detecting outliers, it is very simple to compute these indicators and surprisingly uncomplicated to interpret them. The package `performance` offers a simple function called `check_collinearity()` which provides both. The tolerance can be calculated from the VIF as $\text{tolerance} = \frac{1}{\text{VIF}}$.

```
performance::check_collinearity(m3)

# Check for Multicollinearity

Low Correlation

      Term   VIF   VIF 95% CI Increased SE Tolerance Tolerance 95% CI
schools 1.18 [1.12, 1.28]       1.09      0.85      [0.78, 0.89]
businesses 1.18 [1.12, 1.28]      1.09      0.85      [0.78, 0.89]
```

As the output already indicates, there is a `Low Correlation` between our independent variables. So, good news for us. In terms of interpretations, we find the following benchmarks as recommendations to determine multicollinearity (Field 2013) :

- `VIF > 10`: Evidence for multicollinearity
- `mean(VIF) > 1`: If the mean of all VIFs lies substantially above 1, multicollinearity might be an issue.
- `Tolerance < 0.1`: Multicollinearity is a severe concern. This is the same condition as `VIF > 10`.
- `Tolerance < 0.2`: Multicollinearity could be a concern. This is the same condition as `VIF > 5`.

However, P. Cohen, West, and Aiken (2014) warns to not rely on these indicators alone. There is sufficient evidence that lower VIF scores could also cause issues. Thus, it is always important to still investigate relationships of independent variables statistically (e.g. correlation) and visually (e.g. scatter-plots). For example, outliers can often be a cause for too high or too low VIFs. In short, the simplicity in the computation of these indicators should not be mistaken as a convenient shortcut.

Sometimes, the errors (or residuals) in a regression model can be closely related

to each other as well, which is called *autocorrelation*. Imagine if the mistakes you make when predicting someone's salary were similar every time - maybe you always underestimate how much someone earns after they reach a certain level or work experience. This pattern can be a problem because regression analysis assumes that these errors should be random and independent from one another. When they are not, it can make our results unreliable, meaning we cannot trust whether our findings are accurate. The *Durbin-Watson Test* offers an equally easy way to detect autocorrelation among residuals. We can use the `car` package and the function `durbinWatsonTest()` to retrieve the relevant information.

```
car::durbinWatsonTest(m3) |>
  broom::tidy()

# A tibble: 1 x 5
  statistic p.value autocorrelation method      alternative
  <dbl>     <dbl>        <dbl> <chr>       <chr>
1     0.135      0         0.930 Durbin-Watson Test two.sided
```

The `statistic` of the Durbin-Watson Test can range from 0 to 4, where 2 indicates no correlation. Luckily for us, we do not have to guess whether the difference from our computation is significantly different from 2 because we also get the `p.value` for such test. In our case, our model suffers from strong correlation of error terms because $p.value < 0.05$.

If we wanted to remedy autocorrelation (and to some extend multicollinearity as well), we could consider, for example:

- Revisiting the regression model and potentially dropping those variables that measure the same or similar underlying factors (P. Cohen, West, and Aiken 2014).
- We could collect more data because a larger dataset will always increase the precision of the regression coefficients (P. Cohen, West, and Aiken 2014).
- Use different modelling techniques (P. Cohen, West, and Aiken 2014) which can resolve such issues, for example, using generalised least squares (GLS) models and transform some variables of concern (Grandstrand 2004).

The cause for autocorrelation is often rooted in either '*clustered data*' or a '*serial dependency*' (P. Cohen, West, and Aiken 2014). In our dataset, both could apply. First, we have data that was collected over time, which could lead to wrong standard errors if not accounted for. Second, we have data from two different countries (`United Kingdom` and `Germany`). Thus, our data might be clustered.

To remove serial dependency effects, we would have to transform data and

account for the correlation across time. Such a technique is shown in P. Cohen, West, and Aiken (2014) (p.149). Furthermore, to counteract the issue of clustered data, we need to use multilevel regression models, also known as hierarchical regressions, which we will cover in the next chapter.

13.3 Hierarchical regression

Adding independent variables to a regression model often happens in a step-wise approach, i.e. we do not add all independent variables at once. Instead, we might first add the essential variables (based on prior research), run the regression, and then examine the results. After that, we add more variables that could explain our dependent variable and rerun the regression. This results in two models which we can compare and identify improvements.

In hierarchical regressions, we most frequently distinguish three types of independent variables, which also reflect the order in which we add variables to a multiple regression:

1. *Control variables*: These are independent variables that might affect the dependent variable somehow, and we want to make sure its effects are accounted for. Control variables tend to be not the primary focus of a study but ensure that other independent variables (the main effects) are not spurious. Control variables are added first to a regression.
2. *Main effects variables*: These are the independent variables that the researcher expects will predict the dependent variable best. Main effects variables are added after any control variables.
3. *Moderating variables*: These are variables that attenuate the strength of the relationship between independent and dependent variables. They are also referred to as *interactions* or *interaction terms*. Moderating variables are added last, i.e. after main effects variables and control variables.

In the field of Social Sciences, it is rare not to find control variables and/or moderators in multiple regressions because the social phenomena we tend to investigate are usually affected by other factors as well. Classic control variables or moderators are socio-demographic variables, such as age and gender. The following two chapters cover control variables and moderation effects separately from each other. However, it is not unusual to find both types of variables in the same regression model. Thus, they are not mutually exclusive approaches but simply different types of independent variables.

13.3.1 Regressions with control variables

Hierarchical regression implies that we add variables step-by-step, or as some call it, ‘*block-wise*’. Each block represents a group of variables. Control variables tend to be the first block of variables added to a regression model. However, there are many other ways to perform multiple regression, for example, starting with all variables and removing those that are not significant, which P. Cohen, West, and Aiken (2014) calls a ‘tear-down’ approach (p. 158). As the title indicates, we take less of an exploratory approach to our analysis because we define the hierarchy, i.e. the order, in which we enter variables. In the Social Sciences, this is often the preferred method, so I cover it in greater detail.

However, I should probably explain what the purpose of entering variables in a stepwise approach is. As we discussed in the previous chapter, we sometimes face issues of multicollinearity, which makes it difficult to understand which variables are more important than others in our model. Therefore, we can decide to enter variables that we want to control first and then create another model containing all variables. The procedure is relatively straightforward for our research focus when predicting `new_cases` of COVID-19:

1. Create a model `m1` (or whatever name you want to give it) containing the dependent variable `new_cases` and a control variable, for example `country`.
2. Inspect the results of this model and note down the performance measures.
3. Create another model `m2` and include the control variable `country` and all other independent variables of interest, i.e. `schools`, and `businesses`.
4. Inspect the results of this model and note down the performance measures.
5. Compare models `m1` and `m2` to see whether they are significantly different from each other. We can use `anova()` to perform this step.

Let’s put these steps into action and start with formulating our first model. I choose `country` as a control variable because we have sufficient evidence that clustered data could be a reason for the high autocorrelation of residuals we found in Section 13.2.3. For all computations in this section, we use the original dataset, i.e. `covid_uk_ger`, because we changed the model and therefore would have to revisit outliers from scratch, which we shall skip. We also have to remove observations with missing data to allow comparisons of models and ensure they have the same degrees of freedom (i.e. the same number of observations). So, we begin by selecting our variables of interest and then remove missing data.

```
hr_data <-
  covid_uk_ger |>
  select(new_cases, country, schools, businesses) |>
  droplevels() |>
  na.omit()
```

We have to recode factors into so-called dummy variables⁸ or indicator variables. Dummy variables represent categories as `0s` (i.e. `FALSE` for this observation) and `1s` (`TRUE` for this observation). The good news is, the `lm()` function will do this automatically for us. If you want to inspect the coding ex-ante, you can use the function `contrasts()`.

```
contrasts(hr_data$country)
```

	United Kingdom
Germany	0
United Kingdom	1

The column reflects the coding and the rows represent the levels of our factor. If we had more levels, we will find that the coding will always be the number of levels minus 1. This is a common mistake that novice analysts easily make. You might think you need to have a dummy variable for `Germany` (i.e. 0 and 1) and a dummy variable for `United Kingdom` (i.e. 0 and 1). However, all you really need is one variable, which tells us whether the `country` is the `United Kingdom` (i.e. 1) or not (i.e. 0).

If our control variable has more than two levels, for example, by adding `Italy`, the dummy coding will change to the following:

```
hr_uk_ger_ita <-
  covid |>
  filter(country == "United Kingdom" |
        country == "Germany" |
        country == "Italy") |>
  droplevels()

contrasts(hr_uk_ger_ita$country)

      Italy United Kingdom
Germany          0          0
Italy            1          0
United Kingdom  0          1
```

⁸<https://methods.sagepub.com/reference/the-sage-encyclopedia-of-educational-research-measurement-and-evaluation/i7682.xml?fromsearch=true>

With this new dataset of three countries, our regression would include more control variables because we create a new control variable for each level of the factor (minus one level!).

Returning to our hierarchical regression, we can build our first model, i.e. `m1`, which only contains our control variable `country`.

```
m1 <- lm(formula = new_cases ~ country,
          data = hr_data)

parameters::model_parameters(m1, standardize = "refit")
```

Parameter	Coefficient	SE	95% CI	t(1186)	p
(Intercept)	-0.18	0.04	[-0.26, -0.10]	-4.53 < .001	-
country [United Kingdom]	0.37	0.06	[0.25, 0.48]	6.40 < .001	

Uncertainty intervals (equal-tailed) and p-values (two-tailed) computed using a Wald t-distribution approximation.

```
performance::model_performance(m1)

# Indices of model performance
```

AIC	AICc	BIC	R2	R2 (adj.)	RMSE	Sigma
25542.724	25542.744	25557.964	0.033	0.033	11258.076	11267.564

Our control variable turns out to be significant for our model, but it explains only a small proportion of the variance in `new_cases`. If you are testing hypotheses, you would consider this a good result because you do not want your control variables to explain too much variance. At the same time, it is a significant variable and should be retained in our model. Let's construct our next model, `m2`, by adding the main effects variables and comparing our models.

```
m2 <- lm(formula = new_cases ~
          schools +
          businesses +
          country,
          data = hr_data)
```

```
parameters::compare_parameters(m1, m2, standardize = "refit") |>
  as_tibble() |>
  select(Parameter, Coefficient.m1, p.m1, Coefficient.m2, p.m2) |>
  mutate(across(where(is.double), round, 3))
```

```
# A tibble: 4 × 5
  Parameter          Coefficient.m1  p.m1 Coefficient.m2  p.m2
  <chr>                <dbl>   <dbl>       <dbl>   <dbl>
1 (Intercept)        -0.183      0     -0.235      0
2 country (United Kingdom)  0.365      0      0.469      0
3 schools             NA        NA      0.328      0
4 businesses           NA        NA      0.246      0
```

```
performance::compare_performance(m1, m2)
```

```
# Comparison of Model Performance Indices
```

Name	Model	AIC (weights)	AICc (weights)	BIC (weights)	R2	R2 (adj.)	RMSE	Sigma
m1	lm	25542.7 (<.001)	25542.7 (<.001)	25558.0 (<.001)	0.033	0.033	11258.076	11267.5
m2	lm	25234.1 (>.999)	25234.1 (>.999)	25259.5 (>.999)	0.257	0.255	9870.095	9886.7

After adding all our variables, *adjusted R²* went up from 0.033 to 0.255. While this might seem like a considerable improvement, we have to perform a statistical test to compare the two models. This leads us back to comparing groups, and in many ways, this is what we do here by using the function `anova()`, but we compare models based on the residual sum of squares, i.e. the amount of error the models produce. Thus, if the ANOVA returns a significant result, `m1` shows a significantly reduced residual sum of squares compared to `m2`. Therefore, `m2` would be the better model.

```
anova(m1, m2)
```

```
Analysis of Variance Table
```

```
Model 1: new_cases ~ country
Model 2: new_cases ~ schools + businesses + country
  Res.Df    RSS Df Sum of Sq    F    Pr(>F)
1    1186 1.5057e+11
2    1184 1.1573e+11  2 3.4839e+10 178.21 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The results confirm that our model significantly improved. You might argue that this is not surprising because we added those variables which already worked in the final model of Section 13.2.3. However, the important takeaway is that our control variable `country` helps us explain more variance in `new_cases`. Comparing the model with and without the control variable, we would find that the *adjusted R²* improves our model by about 25%. As such, it is worth keeping it as part of our final regression model. Here is evidence of this improvement:

```
m0 <- lm(formula = new_cases ~
           schools +
           businesses,
           data = hr_data)

performance::compare_performance(m0, m2)

# Comparison of Model Performance Indices

Name | Model |   AIC (weights) |   AICc (weights) |   BIC (weights) |     R2 | R2 (adj.) |     RMSE |     Sigma
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
m0  |   lm | 25309.1 (<.001) | 25309.2 (<.001) | 25329.5 (<.001) | 0.207 | 0.206 | 10195.397 | 10208.2
m2  |   lm | 25234.1 (>.999) | 25234.1 (>.999) | 25259.5 (>.999) | 0.257 | 0.255 |  9870.095 |  9886.7
```



```
anova(m0, m2)
```

Analysis of Variance Table

```
Model 1: new_cases ~ schools + businesses
Model 2: new_cases ~ schools + businesses + country
  Res.Df       RSS Df  Sum of Sq      F    Pr(>F)
1   1185 1.2349e+11
2   1184 1.1573e+11  1 7754483300 79.332 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

There are many more combinations of models we could test. For example, instead of specifying two models (`m1, m2`), we could also add independent variables one at a time and compare the resulting models to a baseline model (`m0`). Thus, we end up with more models to compare, for example:

```
m1 <- lm(formula = new_cases ~ country, data = hr_data)
m2 <- lm(formula = new_cases ~ country + schools, data = hr_data)
```

```
m3 <- lm(formula = new_cases ~ country + schools + businesses, data = hr_data)
```

All these decisions have to be guided by the purpose of your research and whether you explore your data or have pre-defined hypotheses. However, the steps remain the same in terms of computation in *R*.

13.3.2 Moderated regression

While regressions with control variables help us decide whether to include or exclude variables by controlling for another variable, moderation models imply that we expect a certain interaction between an independent variable and a so-called moderator. A moderator also enters the equation on the right-hand side and therefore constitutes an independent variable, but it is usually entered after control variables and main effects variables.

In our quest to find a model that predicts new COVID cases, let's assume that `country` is not a control variable but a moderating one. We could suspect that the measures for `businesses` and `schools` were significantly differently implemented in each country. Therefore, the strength of the relationship between `businesses/schools` and `new_cases` varies depending on the country. In other words, we could assume that measures showed different effectiveness in each of the countries.

When performing a moderation regression, we assume that the relationship between independent variables and the dependent variable differs based on a third variable, in our case, `country`. Thus, if we visualise the idea of moderation, we would plot the data against each other for each country separately and fit two (instead of one) regression lines by defining the colours of each group with `col = country`.

```
# Plotting the interaction between schools and country
hr_data |>
  ggplot(aes(x = schools,
             y = new_cases,
             col = country)) +
  geom_jitter(width = 5,
              size = 0.5,
              alpha = 0.5) +
  stat_smooth(method = "lm",
              formula = y ~ x,
              se = FALSE)
```

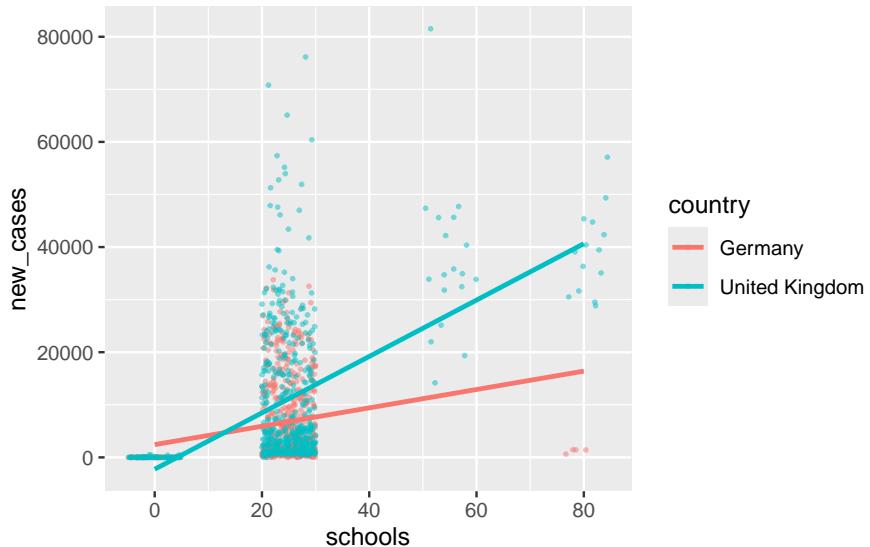


Figure 13.1: Plotting moderation effects for schools.

```
# Plotting the interaction between schools and country
hr_data |>
  ggplot(aes(x = businesses,
             y = new_cases,
             col = country)) +
  geom_jitter(width = 5,
              size = 0.5,
              alpha = 0.5) +
  geom_smooth(method = "lm",
              formula = y ~ x,
              se = FALSE)
```

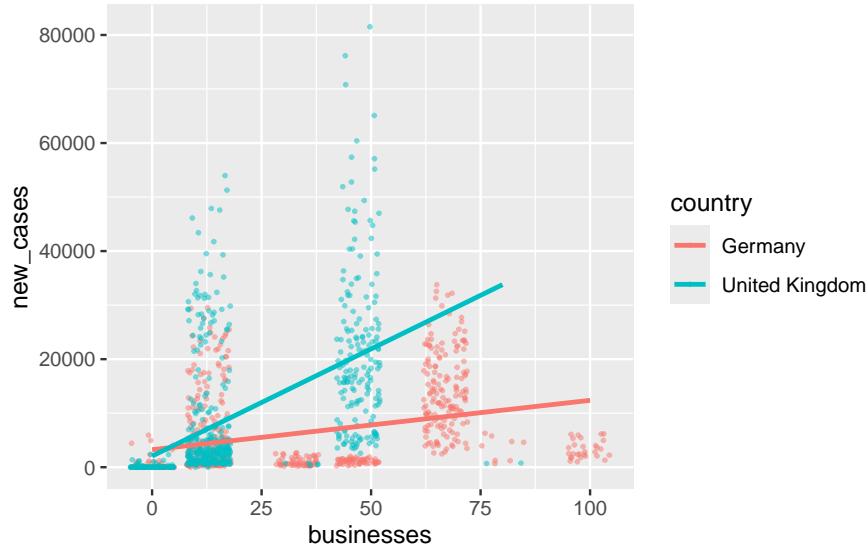


Figure 13.2: Plotting moderation effects for business.

There are three interesting insights we can gain from these plots:

1. In terms of business measures, the United Kingdom seems to have more observations at the lower end, i.e. taking less protective measures than Germany. In contrast, for schools, it is the opposite.
2. For schools, the UK reports more frequently higher numbers of new_cases even if measures taken are high, which results in a steeper regression line.
3. However, for businesses, the United Kingdom had barely any new_cases when the measures were high, but Germany still reports high numbers of new cases when tight measures were taken.

Given the difference in the slope of the regression lines, we have to assume that the β s for Germany are quite different from the ones for the United Kingdom. It seems that the relationship between new_cases and business/schools can be partially explained by country. Therefore, we could include country as a moderator and introduce the interaction of these variables with each other into a new model, i.e. m4.

```
m4 <- lm(formula = new_cases ~
           schools +
           businesses +
           schools * country + # moderator 1
```

```

businesses * country, # moderator 2
data = hr_data)

parameters::compare_parameters(m0, m4, standardize = "refit") |>
  as_tibble() |>
  select(Parameter, Coefficient.m0, p.m0, Coefficient.m4, p.m4) |>
  mutate(across(where(is.double), round, 3))

# A tibble: 6 x 5
  Parameter          Coefficient.m0 p.m0 Coefficient.m4 p.m4
  <chr>                <dbl>   <dbl>        <dbl>   <dbl>
1 (Intercept)            0         1       -0.229  0
2 schools                 0.366    0       0.077  0.104
3 businesses               0.168    0       0.183  0
4 country (United Kingdom) NA        NA      0.546  0
5 schools x country (United Kingdom) NA        NA      0.274  0
6 businesses x country (United Kingdo~ NA        NA      0.371  0

performance::compare_performance(m0, m4)

# Comparison of Model Performance Indices

Name | Model |   AIC (weights) |   AICC (weights) |   BIC (weights) |     R2 | R2 (adj.) |     RMSE |     Sigma
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
m0  | lm | 25309.1 (<.001) | 25309.2 (<.001) | 25329.5 (<.001) | 0.207 | 0.206 | 10195.397 | 10208.2
m4  | lm | 25149.9 (>.999) | 25150.0 (>.999) | 25185.5 (>.999) | 0.310 | 0.307 | 9510.456 | 9534.5

```

```
anova(m0, m4)
```

Analysis of Variance Table

```

Model 1: new_cases ~ schools + businesses
Model 2: new_cases ~ schools + businesses + schools * country + businesses *
          country
Res.Df      RSS Df Sum of Sq      F      Pr(>F)
1    1185 1.2349e+11
2    1182 1.0745e+11  3 1.6035e+10 58.795 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

The moderation effect is quite strong. Compared to our baseline model `m0`, which only contains `businesses` and `schools` as predictors, our new model `m4` considerably outperforms it with an *adjusted R²* = 0.307. Our previous

model with country as a control variable achieved an *adjusted R²* = 0.255. I would argue this is quite an improvement.

While significant moderation effects help us make better predictions with our model, they come with a caveat: Interpreting the relative importance of the main effects becomes considerably more difficult because their impact depends on the level of our moderation variable. As such, you cannot interpret the main effects without considering the moderation variable as well. You might have noticed that our variable `schools` is not significant anymore. However, this does not imply it is not essential because its significance depends on the level of our moderator, i.e. whether we look at Germany or the United Kingdom.

13.3.3 Centering predictors: Making β s more interpretable

In Section 13.2.2, we covered procedures to standardise β coefficients, which allow us to compare independent variables based on different measurement units or if we want to compare coefficients across multiple models. Whenever we perform a regression with an interaction term, for example, the moderation regression in the previous chapter, we often have to make similar adjustments called ‘centering’.

To explain why we might need centering, we should look at an example. Let’s consider a simple model using the `wvs_nona` dataset. Assume we want to know whether `freedom_of_choice` can predict `satisfaction` with life, moderated by `age`. If we formulate this as a model, we can write the following:

```
model <- lm(formula = satisfaction ~
              freedom_of_choice +
              freedom_of_choice * age,
              data = wvs_nona)

parameters::model_parameters(model) |>
  as_tibble() |>
  select(Parameter, Coefficient, t, df_error, p)

# A tibble: 4 x 5
  Parameter          Coefficient      t df_error       p
  <chr>            <dbl>    <dbl>   <int>     <dbl>
1 (Intercept)      3.42     16.3     8560 1.40e-58
2 freedom_of_choice 0.409    13.9     8560 1.50e-43
3 age              0.00815   1.78     8560 7.57e- 2
4 freedom_of_choice:age -0.000635 -0.978    8560 3.28e- 1

performance::model_performance(model)
```

```
# Indices of model performance
```

AIC	AICc	BIC	R2	R2 (adj.)	RMSE	Sigma
<hr/>						
37214.901	37214.908	37250.177	0.129	0.129	2.124	2.124

The β score of `freedom_of_choice` unfolds the most substantial effect on `satisfaction`, but assumes that `age = 0`. In regressions with interaction terms, the coefficient β of a variable reflects the change in the dependent variables if other variables are set to zero. If this is a logical value for the variables, then no further steps are required. However, if a score of zero on any predictor is not meaningful, it is recommended to engage in centering variables (see P. Cohen, West, and Aiken 2014; Field 2013). In our model, the coefficient of `freedom_of_choice` assumes that `age` is equal to 0. However, no participant in our sample will have been of age 0, which renders any interpretation of the β coefficient meaningless. Instead, it is more meaningful to retrieve a β score which is based on the average `age` of people in our sample. This is what centering can achieve.

There are mainly two very commonly used techniques in the Social Sciences to center variables: *grand mean centering* and *group mean centering*. The difference between these two approaches lies in their reference sample. For group mean centering, we compute means for each group independently and for grand mean centering, we calculate the mean for all observations irrespective of any groupings. Centered scores are almost the same as z-scores, but without using standard deviation as a measurement unit. If we divided centered scores by the standard deviation, we would have computed z-scores. The following formula summarises the computation of centered variables:

$$x_{n_{centered}} = x_n - \bar{x}$$

In other words, to achieve a centered variable ($x_{n_{centered}}$), we have to take the original observation (x_n) and subtract the mean of the variable (\bar{x}). How we define \bar{x} determines whether we perform grand mean centering (using the entire dataset) or group mean centering (using subsets of our data, i.e. groups). We already know all the functions needed to compute centered variables. Since the `age` of 0 is not meaningful in our study, we should center this variable around its mean⁹.

```
# Grand mean centering
wvs_nona_c <-
  wvs_nona |>
    mutate(age_c = age - mean(age))
```

⁹It is not required to choose the mean as a way to center variables. Any value that is meaningful can be used. However, the mean often tends to be the most meaningful value to perform centering.

```
wvs_nona_c |>
  select(age, age_c) |>
  head()

# A tibble: 6 x 2
  age   age_c
  <dbl> <dbl>
1    60   17.9
2    40   -2.14
3    25   -17.1
4    71   28.9
5    38   -4.14
6    20   -22.1

# Group mean centering by country
wvs_nona |>
  group_by(country) |>
  mutate(age_c = age - mean(age)) |>

  select(age, age_c) |>
  sample_n(1)
```

Adding missing grouping variables: `country`

```
# A tibble: 6 x 3
# Groups:   country [6]
  country   age   age_c
  <fct>   <dbl> <dbl>
1 Bolivia    21  -17.3
2 Iran       23  -16.5
3 Iraq       43   6.40
4 Japan      30  -24.8
5 Korea      49   3.37
6 Egypt      54   14.3
```

```
## The group means used
wvs_nona |>
  group_by(country) |>
  summarise(mean = round(mean(age), 0))
```

```
# A tibble: 6 x 2
  country   mean
  <fct>   <dbl>
```

```

1 Bolivia    38
2 Iran       39
3 Iraq       37
4 Japan      55
5 Korea      46
6 Egypt      40

```

The `mutate()` function is the same for both approaches. However, we used `group_by()` to compute the mean for each country and then centered the scores accordingly for group mean centering. Thus, observations from Australia will be centered around the mean score of 54, while observations from Myanmar will be centered around the mean value of 40.

Returning to our investigation, I opted to choose grand mean centering. Now we can use our newly centered variable in the regression to see the impact it had on our β s.

```

model_c <- lm(formula = satisfaction ~
               freedom_of_choice +
               freedom_of_choice * age_c,
               data = wvs_nona_c)

parameters::compare_parameters(model, model_c) |>
  as_tibble() |>
  select(Parameter, Coefficient.model, p.model, Coefficient.model_c, p.model_c) |>
  mutate(across(where(is.double), round, 3))

```

Parameter	Coefficient.model	p.model	Coefficient.model_c	p.model_c
<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1 (Intercept)	3.42	0	3.77	0
2 freedom of choice	0.409	0	0.382	0
3 age	0.008	0.076	NA	NA
4 freedom of choice × a~	-0.001	0.328	NA	NA
5 age c	NA	NA	0.008	0.076
6 freedom of choice × a~	NA	NA	-0.001	0.328

In contrast to standardising the coefficients, it is not necessary to center all predictors in your dataset. For example, we only centered `age` and not `freedom_of_choice` in our case, because a value of 0 for the latter variable seems plausible. As we can tell from the output, the β for `freedom_of_choice` has changed, but all other coefficients remained the same. In general, grand mean centering has the following effects:

- centering does not affect the β of the centered variable (i.e. `age`), because we subtract a constant (i.e. the mean) from all observations equally,

- centering does not affect the interaction term (i.e. `freedom_of_choice * age_c`), which is also known as the ‘higher-order’ predictor,
- centering changes the β of the independent variable in the interaction term (also known as ‘lower-order predictor’), (i.e. `freedom_of_choice`)
- centering does not change β s of other variables that are not interacting with a centered variable.
- centering any of the predictors changes the intercept.

To conclude, centering plays a crucial role if the interpretation of β coefficients is essential. If this is true, we should center all variables where a value of 0 is not meaningful. However, keep in mind that we do not change the model itself, i.e. we cannot improve the performance of a model by centering certain variables. Consequently, if we are only interested in the performance of our model, centering does not matter.

```
performance::compare_performance(model, model_c)

# Comparison of Model Performance Indices

Name | Model | AIC (weights) | AICc (weights) | BIC (weights) | R2 | R2 (adj.) | RMSE | Sigma
-----|-----|-----|-----|-----|-----|-----|-----|-----|
-----|-----|-----|-----|-----|-----|-----|-----|-----|
model | lm | 37214.9 (0.500) | 37214.9 (0.500) | 37250.2 (0.500) | 0.129 | 0.129 | 2.124 | 2.124
model_c | lm | 37214.9 (0.500) | 37214.9 (0.500) | 37250.2 (0.500) | 0.129 | 0.129 | 2.124 | 2.124
```

13.4 Other regression models: Alternatives to OLS

This book only covers linear regressions, which are considered *Ordinary Least Squares (OLS)* regression models. These are by far the most frequently used models in the Social Sciences. However, what can we do if the assumptions of OLS models are violated, for example, the relationship of variables looks curved or if our dependent variable is dichotomous and not continuous? In such cases, we might have to consider other types of regression models. There are several different approaches, but we most frequently find one of the following:

- Polynomial regressions*, which are used for curvilinear relationships between variables.
- Generalised Linear models*, such as *Logistic regressions* (when the predictor is a logical variable) or *Poisson regressions* (when we model count data, i.e. frequencies and contingency tables).

These are specialised models, and learning them requires a more in-depth knowledge of their mechanics which would go beyond the scope of a book that intends to cover the basics of conducting statistical analysis in *R*. However, P. Cohen, West, and Aiken (2014) offers an excellent introduction to both approaches.



14

Mixed-methods research: Analysing qualitative data in R

Conducting mixed-methods research is challenging for everyone. It requires an understanding of different methods and data types and particular knowledge in determining how one can mix different methods to improve the insights compared to a single method approach. This chapter looks at one possibility of conducting mixed-methods research in *R*. It is likely the most evident use of computational software for qualitative data.

While I consider myself comfortable in qualitative and quantitative research paradigms, this chapter could be somewhat uncomfortable if you are used to only one or the other approach, i.e. only quantitative or only qualitative research. However, with advancements in big data science, it is impossible to code, for example, two million tweets qualitatively. Thus, the presented methodologies should not be considered in isolation from other forms of analysis. For some, they might serve as a screening tool to sift through large amounts of data and find those nuggets of most significant interest that deserve more in-depth analysis. For others, these tools constitute the primary research design to allow to generalise to a larger population. In short: the purpose of your research will dictate the approach.

Analysing text quantitatively, though, is not new. The field of Corpus Analysis has been doing this for many years. If you happen to be a Corpus Analyst, then this section might be of particular interest.

This chapter will cover two analytical approaches for textual data:

- Word frequencies, and
 - Word networks with n-grams.
-

14.1 The tidy process of working with textual data

Before we dive head-first into this exciting facet of research, we first need to understand how we can represent and work with qualitative data in *R*.

Similar to working with tidy quantitative data, we also want to work with tidy qualitative data. This book adopts the notion of tidy text data from Silge and Robinson (2017), which follows the terminology used in Corpus Linguistics:

We (...) define the tidy text format as being **a table with one-token-per-row**. A token is a meaningful unit of text, such as a word, that we are interested in using for analysis, and tokenization is the process of splitting text into tokens.

In other words, what used to be an ‘observation’ is now called a ‘token’. Therefore, a token is the smallest chosen unit of analysis. The term ‘word token’ can easily be confused with ‘word type’. The first one usually represents the instance of a ‘type’. Thus, the frequency of a word type is determined by the number of its tokens. For example, consider the following sentence, which consists of five tokens, but only four types:

```
(sentence <- "This car, is my car.")
```

```
[1] "This car, is my car."
```

Because the word `car` appears twice, we have two tokens of the word type `car` in our dataset. However, how would we represent this in a rectangular dataset? If each row represents one token, we would expect to have five rows in our data frame. As mentioned above, the process of converting a text into individual tokens is called ‘tokenization’. To turn our text into a tokenized data frame, we have to perform two steps:

1. Convert our `text` object into a data frame, and
2. Split this text into individual words, i.e. tokenization.

While the first part can be achieved using `tibble()`, we need a new function to perform tokenisation. The package `tidytext` will be our primary tool of choice, and it comes with the function `unnest_tokens()`. Among many other valuable applications, `unnest_tokens()` can tokenize the text for us.

```
# Convert text object into a data frame
(df_text <- tibble(text = sentence))

# A tibble: 1 × 1
  text
```

```

<chr>
1 This car, is my car.

# Tokenization
library(tidytext)
(df_text <- df_text |> unnest_tokens(output = word,
                                         input = text))

# A tibble: 5 × 1
  word
  <chr>
1 this
2 car
3 is
4 my
5 car

```

If you paid careful attention, two elements got lost in our data during the process of tokenization: the . and , are no longer included. In most cases, commas and full stops are of less interest because they tend to carry no particular meaning when performing such an analysis. The function `unnest_tokens()` conveniently removes them for us and also converts all letters to lower-case.

From here onwards we find ourselves in more familiar territory because the variable `word` looks like any other character variable we encountered before. For example, we can `count()` the number of occurrences for each word.

```
df_text |> count(word)
```

```

# A tibble: 4 × 2
  word     n
  <chr> <int>
1 car      2
2 is       1
3 my      1
4 this     1

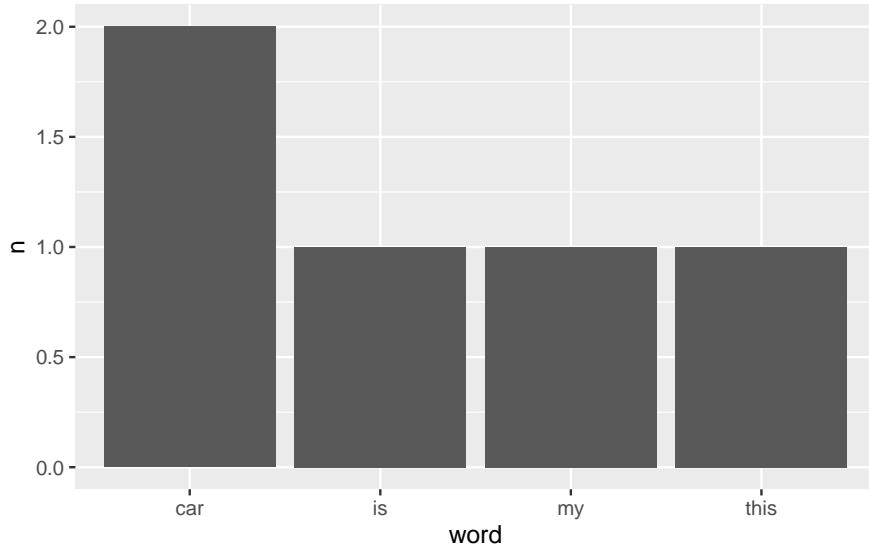
```

Since we have our data nicely summarised, we can also easily visualise it using `ggplot()`.

```

df_text |>
  count(word) |>
  ggplot(aes(x = word,
              y = n)) +
  geom_col()

```



Once we have converted our data into a data frame, all the techniques we covered for non-quantitative variables can be applied. It only requires a single function to turn ourselves into novice corpus analysts. If this sounds too easy, then you are right. Hardly ever will we retrieve a tidy dataset that allows us to work with it in the way we just did. Data cleaning and wrangling still need to be performed but in a slightly different way. Besides, there are many other ways to tokenize text than using individual words, some of which we cover in this chapter.

14.2 Stop words: Removing noise in the data

In the previous example, we already performed an important data wrangling process, i.e. tokenization. However, besides changing the text format, we also need to take care of other components in our data that are usually not important, for example, removing '*stop words*'. To showcase this step (and a little more), I will draw on the `comments` dataset, which holds data about students' study experiences.

In my industry, i.e. Higher Education, understanding students' study experiences is crucial for improving academic programmes and support services. The insights gathered from student feedback can significantly influence how a university enhances its offerings. With the help of the `comments` dataset, we can empirically investigate what themes emerge from students comments about

their social inclusion experiences. However, we first have to tidy the data and then clean it. An essential step in this process is the removal of ‘stop words’.

‘Stop words’ are words that we want to exclude from our analysis, because they carry no particular meaning. The `tidytext` package comes with a data frame that contains common stop words in English.

stop_words

```
# A tibble: 1,149 x 2
  word      lexicon
  <chr>     <chr>
  1 a        SMART
  2 a's      SMART
  3 able     SMART
  4 about    SMART
  5 above    SMART
  6 according SMART
  7 accordingly SMART
  8 across   SMART
  9 actually SMART
  10 after   SMART
# i 1,139 more rows
```

It is important to acknowledge that there is no unified standard of what constitutes a stop word. The data frame `stop_words` covers many words, but in your research context, you might not even want to remove them. For example, if we aim to identify topics in a text, stop words would be equivalent to white noise, i.e. many of them are of no help to identify relevant topics. Instead, stop words would confound our analysis. For obvious reasons, stop words highly depend on the language of your data. Thus, what works for English will not work for German, Russian or Chinese. You will need a separate set of stop words for each language.

The removal of stop words can be achieved with a function we already know: `anti_join()`. In other words, we want to subtract all words from `stop_words` from a given text. Let’s begin with the tokenization of our comments.

```
comments_token <-
  comments |>
  unnest_tokens(word, comment) |>
  select(word, social_inclusion)

comments_token
```

```
# A tibble: 3,299 x 2
```

```

word      social_inclusion
<chr>      <dbl>
1 joining          8
2 the              8
3 international    8
4 club             8
5 was              8
6 a                8
7 game             8
8 changer          8
9 i                8
10 made            8
# i 3,289 more rows

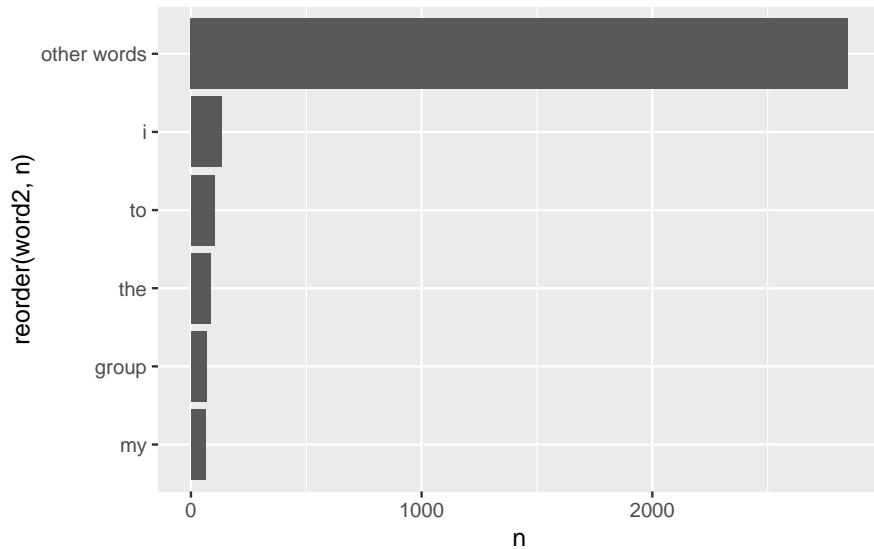
```

Our new data frame has considerably more rows, i.e. 3299, which hints at around 16 words per comment on average. If you are the curious type, like me, we already can peek at the frequency of words in our dataset.

```

comments_token |>
  mutate(word2 = fct_lump_n(word, 5,
                            other_level = "other words")) |>
  count(word2) |>
  ggplot(aes(x = reorder(word2, n),
             y = n)) +
  geom_col() +
  coord_flip()

```



The result is somewhat disappointing. None of these words carries any particular meaning because they are all stop words, except for `group`. Thus, we must clean our data before conducting such an analysis.

I also sneaked in a new function called `fct_lump_n()` from the `forcats` package. It creates a new factor level called `other` words and ‘lumps’ together all the other factor levels. You likely have seen plots before which show a category called ‘Other’. We usually apply this approach if we have many factor levels with very low frequencies. It would not be meaningful to plot 50 words as a barplot which only occurred once in our data. They are less important. Thus, it is sometimes meaningful to pool factor levels together. The function `fct_lump_n(word, 5)` returns the five most frequently occurring words and pools the other words together into a new category. There are many different ways to ‘lump’ factors. For a detailed explanation and showcase of all available alternatives, have a look at the `forcats` website¹.

In our next step, we have to remove stop words using the `stop_words` data frame and apply the function `anti_join()`.

```
comments_no_sw <- anti_join(comments_token, stop_words,
                             by = "word")

comments_no_sw
```

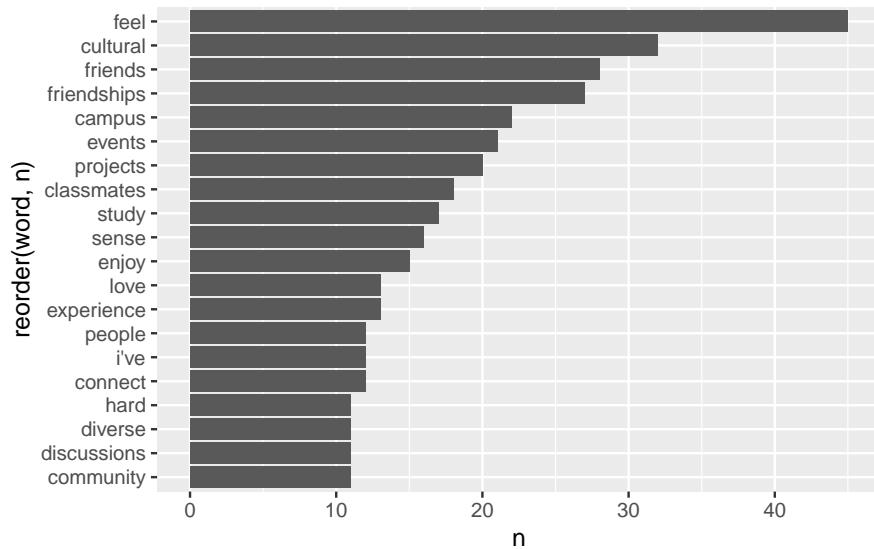
word	social_inclusion
joining	8
international	8
club	8
game	8
changer	8
friends	8
world	8
studying	8
campus	10
diversity	10

i 1,500 more rows

The dataset shrank from 3299 rows to 1510 rows. Thus, about 54% of our data was actually noise. With this cleaner dataset, we can now look at the frequency count again.

¹https://forcats.tidyverse.org/reference/fct_lump.html

```
comments_no_sw |>
  count(word) |>
  filter(n > 10) |>
  ggplot(aes(x = reorder(word, n),
             y = n)) +
  geom_col() +
  coord_flip()
```

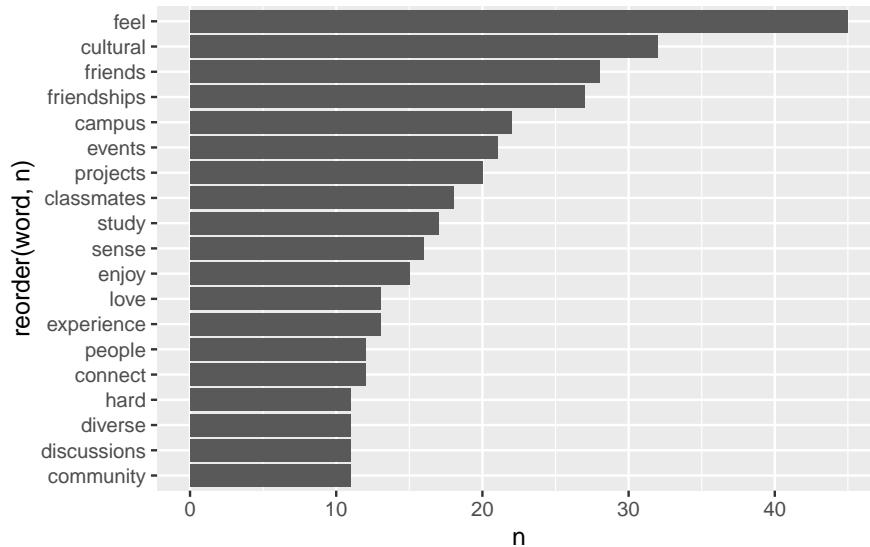


To keep the number of bars somewhat manageable, I removed those words with less than 10 occurrences in the dataset. Some of the most frequently occurring words include, for example, `feel`, `cultural`, `friends`, `campus`, `events`, `projects`. There is also a reference to the word `i've` which refers to `i have` and, therefore, constitutes a stop word. However, it seems our list of `stopwords` did not pick this one up. It is always good to remember that our tools for analysis are never perfect and some more manual data cleaning is often necessary. It is very easy to remove this occurrence of words using our trusty `filter()` function.

```
comments_no_sw <-
  comments_no_sw |>
  filter(word != "i've")

comments_no_sw |>
  count(word) |>
  filter(n > 10) |>
  ggplot(aes(x = reorder(word, n),
```

```
y = n) +
geom_col() +
coord_flip()
```



Another commonly used method for visualising word frequencies are word clouds. You likely have seen them before since they are very popular, especially for websites and poster presentations. Word clouds use the frequency of words to determine the font size of each word and arrange them so that it resembles the shape of a cloud. The package `wordcloud` and its function `wordcloud()` make this very easy. You can modify a word cloud in several ways. Here are some settings I frequently change for my data visualisations:

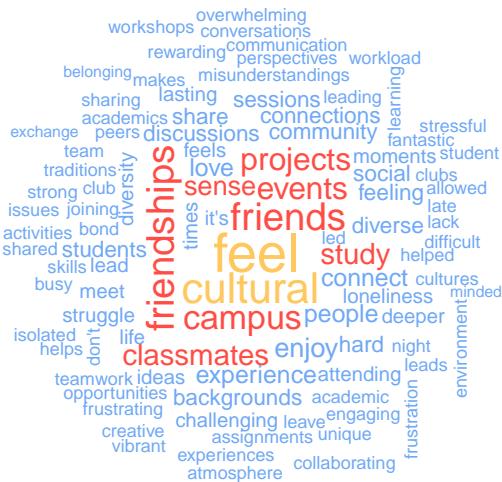
- `random.order = FALSE`: This ensures that the most important words appear in the middle of the plot.
- `scale = c()`: This attribute determines the size difference between the most and least frequent words in the dataset.
- `min.freq`: This setting determines which words are included based on their frequency, which is helpful if you have many words to plot.
- `max.words`: This attribute limits the number of words that should be plotted and is also useful in cutting down on large datasets.
- `colors`: This setting allows you to provide custom colours for different frequencies of words. This substantially improves the readability of your data visualisation.

To create a word cloud you have to provide at least a column with words and

a column with their frequencies. All other settings are entirely customisable and optional. Here is an example based on the data we just used for our bar plot.

```
# Create a dataframe with the word count
word_frequencies <-
  comments_no_sw |>
  count(word)

# Plot word cloud
wordcloud::wordcloud(words = word_frequencies$word,
                      freq = word_frequencies$n,
                      random.order = FALSE,
                      scale = c(2, 0.5),
                      min.freq = 1,
                      max.words = 100,
                      colors = c("#6FA8F5",
                                "#FF4D45",
                                "#FFC85E"))
)
```



Our word cloud nicely summarises keywords that were particularly important, assuming we consider frequency of words an indicator of importance. Students seem to talk about their feelings, culture and friends/friendships. While this constitutes a good starting point for further analysis we would require more context about what was said, for example what kind of feelings were expressed.

This is something we will explore more shortly. Before we advance to this stage, though, we can investigate which words were mostly associated with each `social_inclusion` scores, i.e. what words were most frequently used by students who scores a 1, 2, 3, etc. In other words, each `social_inclusion` will become its own little group/sub-sample that we look at.

We can still use `count()` but we have do approach this slightly differently:

1. We create the frequencies for each word with `count()` and include both variables, `social_inclusion` and `word`. This way, `count` will look for all combinations of the social inclusion scores and the word.
2. Then, we group our data based on `social_inclusion`.
3. We select the top 5 most frequently occurring words for each group.

```
comments_grouped <-
  comments_no_sw |>
  count(social_inclusion, word) |>
  group_by(social_inclusion) |>
  slice_max(n, n = 5) |>
  ungroup() |>
  arrange(desc(social_inclusion))
```

```
comments_grouped
```

```
# A tibble: 58 x 3
  social_inclusion word      n
  <dbl> <chr>     <int>
1 10   friendships  7
2 10   feel        5
3 10   friends     5
4 10   diversity   3
5 10   times       3
6 8    friends     14
7 8    friendships  14
8 8    study       6
9 8    classmates  5
10 8   love        5
# i 48 more rows
```

When you first look at this you might be slightly confused what it actually shows. It is best to think of `social_inclusion` as a grouping variable and not as a number anymore. For example, in group 10, which refers to comments that scored very high on `social_inclusion`, `friendships` are mentioned 7 times, while `feel` and `friends` are mentioned 5 times. Similarly, in group 8, `friends`

and friendships are each mentioned 14 times. The frequency is very much affected by the size of each group, i.e. how many students scored a 10 or an 8. Thus, comparing results across groups based on n is not wise. However, looking at words within each group we can get a sense of what made students score very high or low.

We can visualise this data which will make it even clearer what the value of this analysis is. However, first we have to do some housekeeping since the meaning of our variables has changed. Our variable `social_inclusion` should now be a `factor`. Let's transform it in the same way as we did at the very beginning of the book - going full circle.

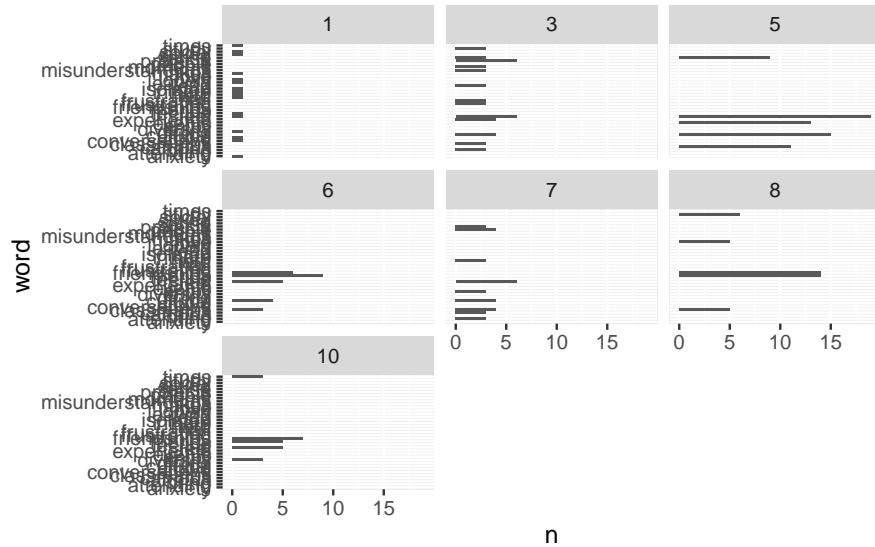
```
comments_grouped <-
  comments_grouped |>
  mutate(social_inclusion = as_factor(social_inclusion))

glimpse(comments_grouped)
```

```
Rows: 58
Columns: 3
$ social_inclusion <fct> 10, 10, 10, 10, 10, 8, 8, 8, 8, 7, 7, 7, 7, ~
$ word          <chr> "friendships", "feel", "friends", "diversity", "times~
$ n             <int> 7, 5, 5, 3, 3, 14, 14, 6, 5, 5, 6, 4, 4, 4, 3, 3, 3, ~
```

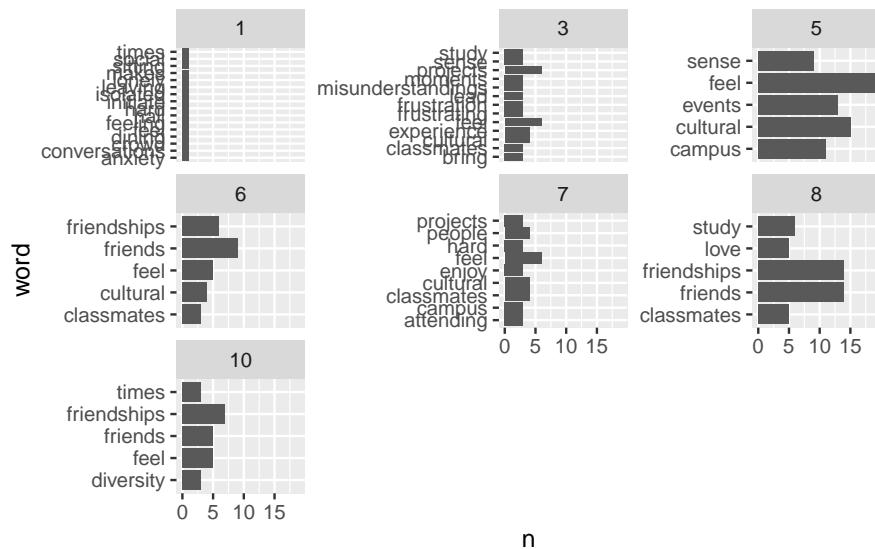
With this in place we can go ahead and create a plot for each group by using `facet_wrap()`.

```
comments_grouped |>
  ggplot(aes(x = word,
             y = n)
         ) +
  geom_col() +
  coord_flip() +
  facet_wrap(~ social_inclusion)
```



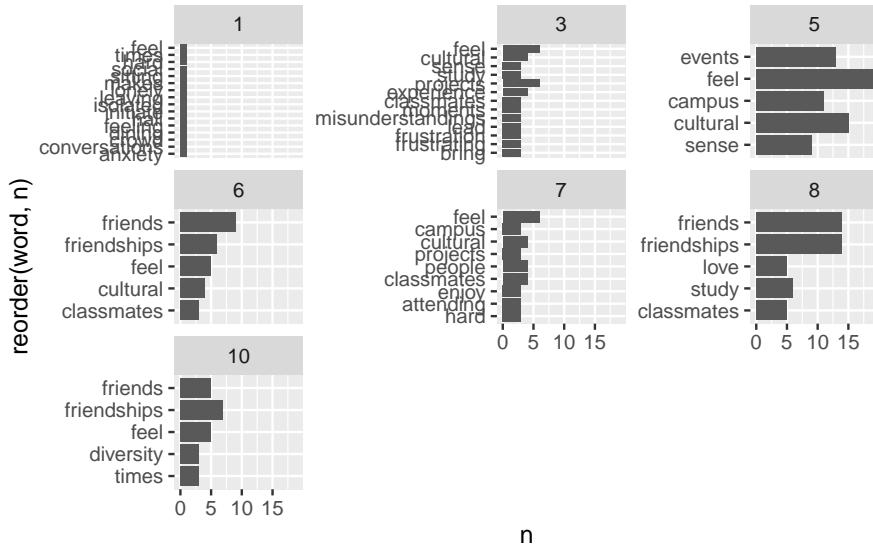
One need not be a graphic designer to understand that this is not a well-designed visualisation and very hard to understand. Let's improve it further by letting each plot have their own y-axis. In this case, this is perfectly fine, since there are different words ranked for each facet plot. We can achieve this by inserting `scales = "free_y"` into the `facet_wrap()` function.

```
comments_grouped |>
  ggplot(aes(x = word,
             y = n))
  ) +
  geom_col() +
  coord_flip() +
  facet_wrap(~ social_inclusion,
             scales = "free_y")
```



This looks a lot better and we start to see which words appeared most frequently in comments for each level of `social_inclusion`. There is, however, one more step we should take to make this visualisation even more meaningful. Ideally, we want the top word show up at the top for each plot. We can see that currently it seems each bar is in a rather random order. This problem might sound familiar, since we learnt how to `reorder()` barplots very early on in this book. So, let's apply our knowledge to this plot.

```
comments_grouped |>
  ggplot(aes(x = reorder(word, n),
             y = n)
         ) +
  geom_col() +
  coord_flip() +
  facet_wrap(~ social_inclusion,
             scales = "free_y")
```



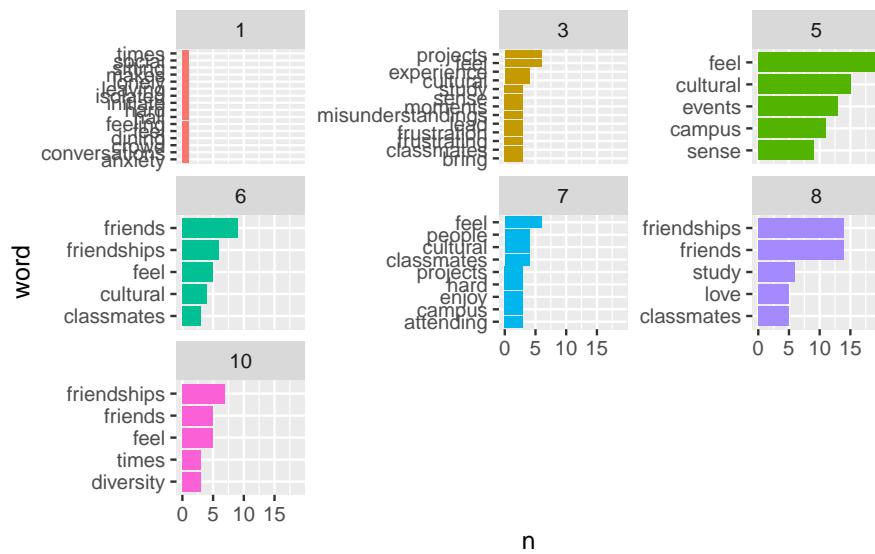
Well, this brought us a little closer, but not quite were we intended to be. Some plots are in the correct order but others are not. The reason for this odd phenomenon is that we reordered our variable `word` for the entire dataset and not for each facet in our visualisation. Unfortunately, neither `reorder()` nor `fct_reorder()` can help us in this case. Instead, we need to use two new functions from the `tidytext` package to help us perfect our plot: `reorder_within()` and `scale_x_reordered()`. The `reorder_within()` function allows us to order a variable based on another quantitative variable (`n`) for each level of a factor (`social_inclusion`). The `scale_x_reordered()` function cleans up the labels of our x-axis which were changed in the process of using `reorder_within()`. If you are curious how the plot would look like without using `scale_x_reordered()` you can simply remove it.

```
comments_grouped |>

# Reorder our data for for each group level ion social_inclusion
mutate(word = reorder_within(x = word,
                             by = n,
                             within = social_inclusion)) |>

# Create the plot
ggplot(aes(x = word,
           y = n,
           fill = social_inclusion)) +
  geom_col(show.legend = FALSE) +
  coord_flip() +
```

```
# Adjust the x-axis to obtain the correct labels
scale_x_reordered() +
  facet_wrap(~ social_inclusion,
             scales = "free_y")
```



The final visualisation, with a splash of colour added, reveals interesting clusters of words that are directly linked to the `social_inclusion` scores. For example, it seems students who score 5 or lower on `social_inclusion` never refer to `friends` or `friendships`. Most importantly, we can very quickly understand how individuals with very low scores felt: `lonely`, `isolated`, `anxiety`. This is clearly something a department would aim to improve and maybe learn from students who managed to built flourishing relationships with their peers.

While these might all be fascinating insights, we have not exhausted our options and can further expand on what we did so far by looking at more than just single, isolated, words.

14.3 N-grams: Exploring correlations of words

Besides looking at words in isolation, it is often more interesting to understand combinations of words to provide much-needed context. For example, the difference between ‘like’ and ‘not like’ can be crucial when trying to under-

stand sentiments in data. The co-occurrence of words follows the same idea as correlations, i.e. how often one word appears together with another. If the frequency of word pairs is high, the relationship between these two words is strong. The technical term for looking at tokens that represent pairs for words is ‘*bigrad*’. If we look at more than two words, we would consider them as ‘*n-gram*’, where the ‘*n*’ stands for the number of words.

Creating a bigram is relatively simple in *R* and follows similar steps as counting word frequencies:

1. Turn text into a data frame.
2. Tokenize the variable that holds the text, i.e. `unnest_tokens()`,
3. Split the two words into separate variables, e.g. `word1` and `word2`, using `separate()`.
4. Remove stop words from both variables listwise, i.e. use `filter()` for `word1` and `word2`.
5. Merge columns `word1` and `word2` into one column again, i.e. `unite()` them. (optional)
6. Count the frequency of bigrams, i.e. `count()`.

There are a lot of new functions covered in this part. However, by now, you likely understand how they function already. Let’s proceed step-by-step. For this final example about mixed-methods research, we will look at the `comment` variable in the `comments` dataset. Since we already have the text as a data frame, we can skip step one. Next, we need to engage in tokenization. While we use the same function as before, we need to provide different arguments to retrieve bigrams. As before, we need to define an `output` column and an `input` column. In addition, we also have to provide the correct type of tokenization, i.e. determine `token`, which we need to set to “`ngrams`”. We also need to define the `n` in ‘`ngrams`’, which will be `2` for bigrams. After this, we apply `count()` to our variable `bigram`, and we achieved our task. If we turn this into *R* code, we get the following (with considerably less characters than its explanation):

```
# Create bigrams from variable synopsis
comments_bigram <-
  comments |>
  unnest_tokens(bigram, comment, token = "ngrams", n = 2)

comments_bigram

# A tibble: 3,090 x 2
  social_inclusion bigram
  <dbl> <chr>
1                 8 joining the
2                 8 the international
```

```

3           8 international club
4           8 club was
5           8 was a
6           8 a game
7           8 game changer
8           8 changer i
9           8 i made
10          8 made friends
# i 3,080 more rows

# Inspect frequency of bigrams
comments_bigram |> count(bigram, sort = TRUE)

```

```

# A tibble: 2,098 x 2
  bigram          n
  <chr>        <int>
1 can be         19
2 i appreciate   18
3 i often         18
4 group work      16
5 sense of         16
6 group projects   14
7 i enjoy          14
8 but i            13
9 i love            13
10 i sometimes       13
# i 2,088 more rows

```

As before, the most frequent bigrams are those that contain stop words. Thus, we need to clean our data and remove them. This time, though, we face the challenge that the variable `bigram` has two words and not one. Thus, we cannot simply use `anti_join()` because the dataset `stop_words` only contains individual words, not pairs. To remove stop words successfully, we have to `separate()` the variable into two variables so that each word has its own column. The package `tidyverse` makes this an effortless task.

```

bigram_split <-
  comments_bigram |>
  separate(col = bigram,
    into = c("word1", "word2"),
    sep = " ")

```

```
bigram_split
```

```
# A tibble: 3,090 x 3
```

```

social_inclusion word1      word2
                  <dbl> <chr>      <chr>
1                 8 joining     the
2                 8 the         international
3                 8 international club
4                 8 club        was
5                 8 was         a
6                 8 a           game
7                 8 game        changer
8                 8 changer     i
9                 8 i           made
10                8 made        friends
# i 3,080 more rows

```

With this new data frame, we can remove stop words in each variable. Of course, we could use `anti_join()` as before and perform the step twice, but there is a much more elegant solution. Another method to compare values across vectors/columns is the `%in%` operator. It is very intuitive to use because it tests whether values in the variable on the left of `%in%` exist in the variable to the right. Let's look at a simple example. Assume we have an object that contains the names of people related to `family` and `work`. We want to know whether people in our `family` are also found in the list of `work`. If you remember Table 5.1, which lists all logical operators, you might be tempted to try `family == work`. However, this would assess the object in its entirety and not tell us which values, i.e. which names, can be found in both lists.

```

# Define the two
family <- tibble(name = c("Fiona", "Ida", "Lukas", "Daniel"))
work <- tibble(name = c("Fiona", "Katharina", "Daniel"))

# Compare the two objects using '=='
family$name == work$name

```

Warning in family\$name == work\$name: longer object length is not a multiple of shorter object length

```
[1] TRUE FALSE FALSE FALSE
```

The output suggests that all but one value in `family` and `work` are not equal, i.e. they are `FALSE`. This is because `==` compares the values in order. If we changed the order, some of the `FALSE` values would turn `TRUE`. In addition, we get a warning telling us that these two objects are not equally long because `family` holds four names, while `work` contains only three names. Lastly, the results are not what we expected because `Fiona` and `Daniel` appear in both objects. Therefore we would expect that there should be two `TRUE` values. In short, `==` is not the right choice to compare these two objects.

If we use `%in%` instead, we can test whether each name appears in both objects, irrespective of their length and the order of the values.

```
# Compare the two objects using '%in%'  
family$name %in% work$name
```

```
[1] TRUE FALSE FALSE TRUE
```

If we want *R* to return the values that are the same across both objects, we can ask: `which()` names are the same?

```
# Compare the two objects using '%in%'  
(same_names <- which(family$name %in% work$name))
```

```
[1] 1 4
```

The numbers returned by `which()` refer to the position of the value in our object. In our case, the first value in `family` is Fiona, and the fourth value is Daniel. It is `TRUE` that these two names appear in both objects. If you have many values that overlap in a large dataset, you might not want to know the row number but retrieve the actual values. This can be achieved by `slice()`ing our dataset, i.e. filtering our data frame by providing row numbers.

```
# Retrieve the values which exist in both objects  
family |>  
slice(same_names)
```

```
# A tibble: 2 x 1  
  name  
  <chr>  
1 Fiona  
2 Daniel
```

While this might be a nice little exercise, it is important to understand how `%in%` can help us remove stop words. Technically, we try to achieve the opposite, i.e. we want to keep the values in `word1` and `word2` that are not a word in `stop_words`. If we use the language of `dplyr`, we `filter()` based on whether a word in `bigram_split` is not `%in%` the data frame `stop_words`. Be aware that `%in%` requires a variable and not an entire data frame to work as we intend.

```
bigram_cleaned <-  
  bigram_split |>  
  filter(!word1 %in% stop_words$word) |>  
  filter(!word2 %in% stop_words$word)
```

```
bigram_cleaned <- bigram_cleaned |>
```

```
  count(word1, word2, sort = TRUE)
```

```
bigram_cleaned
```

```
# A tibble: 420 x 3
  word1    word2      n
  <chr>    <chr>     <int>
1 cultural events     8
2 study    sessions   8
3 lasting   friendships 7
4 late     night      5
5 night    study      5
6 campus   life       4
7 leave    feeling    4
8 campus   events     3
9 deeper   connections 3
10 feel    grateful   3
# i 410 more rows
```

The result is a clean dataset, which reveals that `cultural` and `events` are the most common bigram together with `study` and `session` in the `comments` dataset. Clearly, students spend a lot of time at social events or in study groups. Thus, when they talk about their experiences, these could be considered common places where they have them.

Sometimes we might wish to `re-unite()` the two columns that we separated. For example, when plotting the results into a `ggplot()`, we need both words in one column² to use the actual bigrams as labels.

```
bigram_cleaned |>
  unite(col = bigram,
        word1, word2,
        sep = " ")
```

```
# A tibble: 420 x 2
  bigram          n
  <chr>         <int>
1 cultural events     8
2 study sessions    8
3 lasting friendships 7
```

²With `glue::glue()` you can also combine values from two different columns directly in `ggplot()`, but it seems simpler to just use `unite()`. Remember, there is always more than one way of doing things in R.

```

4 late night      5
5 night study    5
6 campus life    4
7 leave feeling   4
8 campus events   3
9 deeper connections 3
10 feel grateful  3
# i 410 more rows

```

So far, we primarily looked at frequencies as numbers in tables or as bar plots. However, it is possible to create network plots of words with bigrams by drawing a line between `word1` and `word2`. Since there will be overlaps across bigrams, they would mutually connect and create a network of linked words.

I am sure you have seen visualisations of networks before but might not have engaged with them on a more analytical level. A network plot consists of ‘nodes’, which represent observations in our data, and ‘edges’, which represent the link between nodes. We need to define both to create a network.

Unfortunately, `ggplot2` does not enable us to easily create network plots, but the package `ggraph` offers such features using the familiar `ggplot2` syntax. Network plots are usually made using specific algorithms to arrange values in a network efficiently. Thus, if you want your plots to be reproducible, you have to `set.seed()` in advance. This way, random aspects of some algorithms are set constant.

There is one more complication, but with a simple solution. The function `ggraph()`, which is an equivalent to `ggplot()`, requires us to create a `graph` object that can be used to plot networks. The package `igraph` has a convenient function that produces a `graph_from_data_frame()`. As a final step we need to choose a layout for the network. This requires some experimentation. Details about different layouts and more for the `ggraph` package can be found on its website³.

```

library(ggraph)

# Make plot reproducible
set.seed(1234)

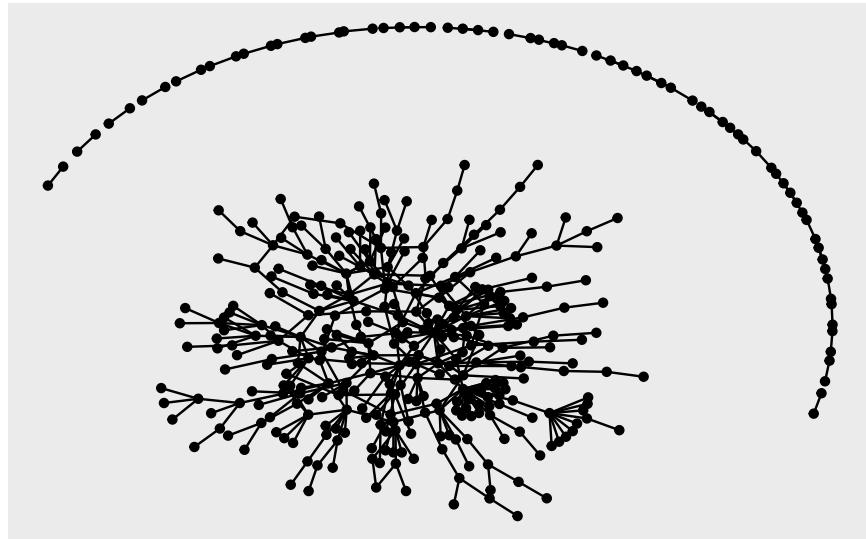
# Create the special igraph object
graph <- igraph::graph_from_data_frame(bigram_cleaned)

# Plot the network graph
graph |>

```

³<https://ggraph.data-imarinist.com>

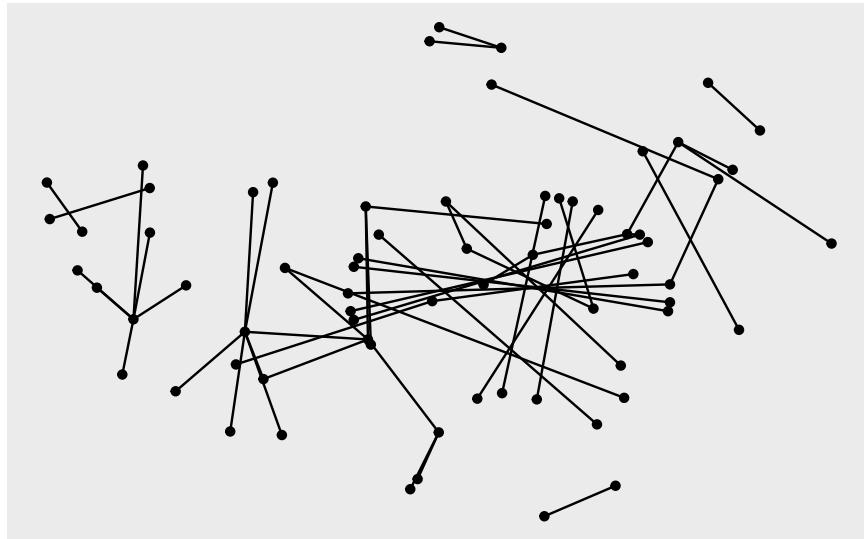
```
ggraph(layout = "kk") + # Choose a layout, e.g. "kk"
  geom_edge_link() +      # Draw lines between nodes
  geom_node_point()       # Add node points
```



The result looks like a piece of modern art. Still, it is not easy to understand what the plot shows us. Thus, we need to remove some bigrams that are not so frequent. For example, we could remove those that only appear once.

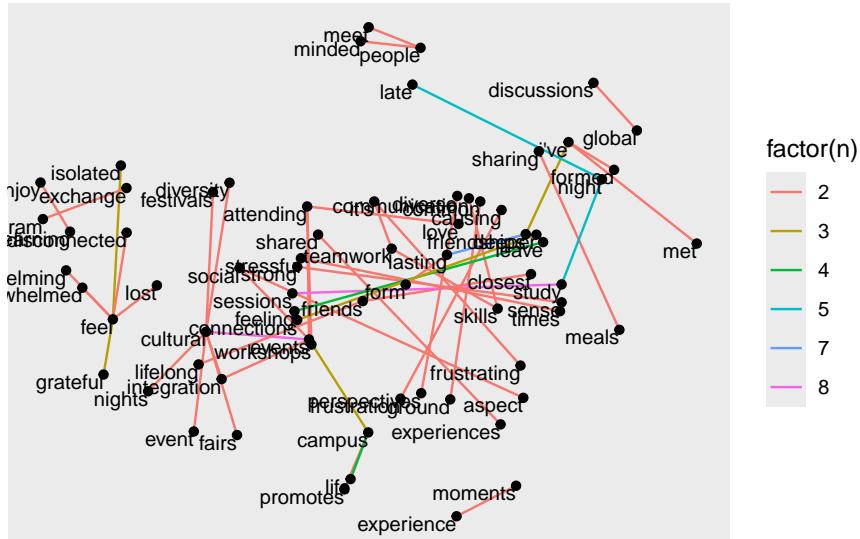
```
# Filter bigrams
network_plot <- bigram_cleaned |>
  filter(n > 1) |>
  igraph::graph_from_data_frame()

# Plot network plot
network_plot |>
  ggraph(layout = "kk") +
  geom_edge_link() +
  geom_node_point()
```



It seems that many bigrams were removed. Thus, the network plot looks much smaller and less busy. Nevertheless, we cannot fully understand what this visualisation shows because there are no labels for each node. We can add them using `geom_node_text()`. I also recommend offsetting the labels with `vjust` (vertical adjustment) and `hjust` (horizontal adjustment), making them easier to read. To enhance the network visualisation further, we could colour the edges based on the frequency of each token.

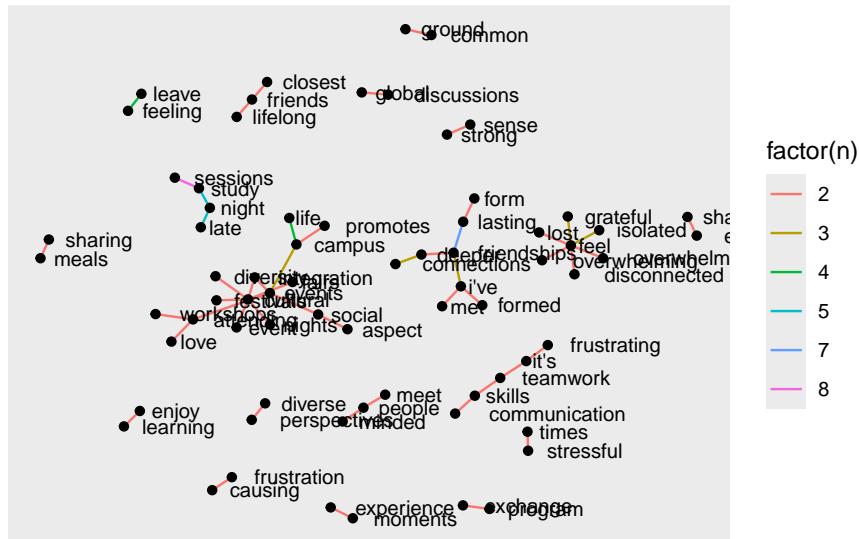
```
# Plot network plot
network_plot |>
  ggraph(layout = "kk") +
  geom_edge_link(aes(col = factor(n))) +
  geom_node_point() +
  geom_node_text(aes(label = name),
                vjust = 1,
                hjust = 1,
                size = 3)
```



This visualisation looks considerably more informative than any of the previous attempts. Still, it seems the chosen layout is not ideal to identify themes/topics in our data. Thus, it is worth experimenting with another layout, for example `graphopt`.

```
set.seed(1234)

# Plot network plot but using a different layout
network_plot |>
  ggraph(layout = "graphopt") +
  geom_edge_link(aes(col = factor(n))) +
  geom_node_point() +
  geom_node_text(aes(label = name),
                vjust = 0.5,
                hjust = -0.25,
                size = 3)
```



The final plot shows us how themes like *events*, *friendships*, and *feel* (i.e. ‘feelings’) connect to other words. They constitute important topics that students talked about when describing their study experiences. While this type of analysis cannot fully replace manual analysis, it does offer unique insights into our data from a macro perspective without spending days analysing these comments.

Our example is a simple analysis based on a relatively small ‘corpus’, i.e. a small dataset. However, the application of n-grams in larger datasets can reveal topic areas and links between them even more effectively. There are more approaches to exploring topics in large datasets, for example, ‘topic modelling’⁴, which are far more complex, but offer more sophisticated analytical insights.

14.4 Exploring more mixed-methods research approaches in R

This chapter can only be considered a teaser for mixed-methods research in *R*. There is much more to know and much more to learn. If this type of work is of interest, especially when working with social media data or historical text documents, there are several *R* packages I can recommend:

⁴<https://www.tidytextmining.com/topicmodeling.html>

- `tm`: This text mining package⁵ includes many functions that help to explore documents systematically.
- `quanteda`: This package⁶ offers natural language processing features and incorporates functions that are commonly seen in corpus analysis. It also provides tools to visualise frequencies of text, for example, wordclouds.
- `topicmodels`: To analyse topics in large datasets, it is necessary to use special Natural Language Processing techniques. This package⁷ offers ways to perform *Latent Dirichlet Allocation (LDA)* and *Correlated Topic Models (CTM)*. Both are exciting and promising pathways to large-scale text analysis.
- The `tidytext` package we used above also has a lot more to offer than what we covered, for example, performing *sentiment analyses*.

If you want to explore the world of textual analysis in greater depth, there are two fantastic resources I wholeheartedly can recommend:

- Text mining with tidy data principles⁸: This free online course introduces additional use cases of the `tidytext` package by letting you work with data interactively right in your browser. Julia Silge developed these materials.
- Supervised Machine Learning for Text Analysis in R⁹: This book is the ideal follow-up resource if you worked your way through Silge and Robinson (2017). It introduces the foundations of natural language analysis and covers advanced text analysis methods such as regression, classification, and deep learning.

I hope I managed to showcase the versatility of *R* to some degree in this chapter. Needless to say, there is also a lot I still have to learn, which makes using *R* exciting and, more than once, has inspired me to approach my data in unconventional ways.

⁵<http://tm.r-forge.r-project.org>

⁶<https://quanteda.io>

⁷<https://cran.r-project.org/web/packages/topicmodels/index.html>

⁸<https://juliasilge.shinyapps.io/learntidytext/>

⁹<https://smltar.com>



15

Where to go from here: The next steps in your R journey

The first steps have been made, but we can certainly not claim we have reached our destination on this journey. If anything, this book helped us reach the first peak in a series of mountains, giving us an overview of what else there is to explore. To provide some guidance on where to go next, I collated some resources worth exploring as a natural continuation to this book.

15.1 GitHub: A Gateway to even more ingenious *R* packages

If you feel you want more *R* or you are curious to know what other *R* packages can do to complement your analysis, then GitHub brings an almost infinite amount of options for you. Besides the CRAN versions of packages, you find development versions of *R* packages on Github. These usually incorporate the latest features but are not yet available through CRAN. Having said that, if you write your next publication, it might be best to work with packages that are released on CRAN. These are the most stable versions. After all, you don't want the software to fail on you and hinder you from making that world-changing discovery.

Still, more and more *R* packages are released every day that offer the latest advancements in statistical computations and beyond. Methods covered in Chapter 14 are a great example of what has become possible with advanced programming languages. So if you want to live at the bleeding edge of innovative research methods, then look no further.

However, GitHub also serves another purpose: To back up your research projects and work collaboratively with others. I strongly encourage you to create your own GitHub account, even just to try it. RStudio has built-in features for working with GitHub, making it easy to keep track of your analysis and ensure regular backups. Nobody wants to clean up their data over months and lose it because their cat just spilt the freshly brewed Taiwanese High Mountain Oolong Tea over one's laptop. I use GitHub for many different

things, hosting my blog (The Social Science Sofa¹), research projects, and this book.

There is much more to say about GitHub that cannot be covered in this book, but if you seek an introduction, you might find GitHub's Youtube Channel² of interest or their 'Get started'³ guide. Best of all, setting up and using your GitHub account is free.

15.2 Books to read and expand your knowledge

There are undoubtedly many more books to read, explore and use. However, my number one recommendation to follow up with this book is 'R for Data Science'⁴. It takes your *R* coding beyond the context of Social Sciences and introduces new aspects, such as '*custom functions*' and '*looping*'. These are two essential techniques if you, for example, have to fit a regression model for +20 subsets of your data, create +100 plots to show results for different countries, or need to create +1000 of individualised reports for training participants. In short, these are very powerful (and efficient) techniques you should learn sooner than later but are less essential for a basic understanding of data analysis in *R* for the Social Sciences.

If you want to expand your repertoire regarding data visualisations, a fantastic starting point represents 'ggplot2: Elegant Graphics for Data Analysis'⁵ and 'Fundamentals of Data Visualization'⁶. However, these days I am looking mostly for inspiration and new packages that help me create unique, customised plots for my presentations and publications. Therefore, looking at the source code of plots you like (for example, on GitHub) is probably the best way to learn about ggplot2 and some new techniques of how to achieve specific effects (see also Section 15.4).

If you are a qualitative researcher, you might be more interested in what else you can do with *R* to systematically analyse large amounts of textual data (as was shown in Chapter 14). I started with the excellent book 'Text Mining with R: A Tidy Approach'⁷, which introduces you in greater depth to sentiment analysis, correlations, n-grams, and topic modelling. The lines of qualitative and quantitative research become increasingly blurred. Thus, learning these

¹<http://thesocialsciencesofa.com>

²<https://www.youtube.com/user/github>

³<https://docs.github.com/en>

⁴<https://r4ds.had.co.nz>

⁵<https://ggplot2-book.org>

⁶<https://clauswilke.com/dataviz/>

⁷<https://www.tidytextmining.com>

techniques will be essential moving forward and pushing the boundaries of what is possible with textual data.

R can do much more than just statistical computing and creating pretty graphs. For example, you can write your papers with it, even if you do not write a single line of code. From Section 6.4, you might remember that I explained that *R* Markdown files are an alternative to writing *R* scripts. Suppose you want to deepen your knowledge in this area and finally let go of Microsoft Word, I encourage you to take a peek at the ‘*R* Markdown Cookbook’⁸ for individual markdown files and ‘bookdown: Authoring Books and Technical Documents with *R* Markdown’⁹ for entire manuscripts, e.g. journal paper submissions or books. I almost entirely abandoned Microsoft Word, even though it served me well for so many years - thanks.

Lastly, I want to make you aware of another open source book that covers What They Forgot to Teach you About *R*¹⁰ by Jennifer Bryan and Jim Hester. It is an excellent resource to get some additional insights into how one should go about working in *R* and *RStudio*.

15.3 Engage in regular online readings about *R*

A lot of helpful information for novice and expert *R* users is not captured in books but online blogs. There are several that I find inspiring and an excellent learning platform, each focusing on different aspects.

The most comprehensive hub for all things *R* is undoubtedly ‘R-bloggers’¹¹. It is a blog aggregator which focuses on collecting content related to *R*. I use it regularly to read more about new packages, new techniques, helpful tricks to become more ‘fluent’ in *R* or simply find inspiration for my own *R* packages. More than once, I found interesting blogs by just reading posts on ‘R-bloggers’. For example, the day I wrote this chapter, I learned about `emayili`, which allows you to write your emails from *R* using *R* markdown. So not even the sky is the limit, it seems.

Another blog worth considering is the one from ‘Tidyverse’¹². This blog is hosted and run by RStudio and covers all packages within the `tidyverse`. Posts like ‘waldo 0.3.0’¹³ made my everyday data wrangling tasks a lot easier because finding the differences between two datasets can be like searching a

⁸<https://bookdown.org/yihui/rmarkdown-cookbook/>

⁹<https://bookdown.org/yihui/bookdown/>

¹⁰<https://rstats.wtf>

¹¹<https://www.r-bloggers.com>

¹²<https://www.tidyverse.org/blog/>

¹³<https://www.tidyverse.org/blog/2021/08/waldo-0-3-0/>

needle in a haystack. For example, it is not unusual to receive two datasets that contain the same measures, but some of their column names are slightly different, which does not allow us to merge them in the way we want quickly. I previously spent days comparing and arranging multiple datasets with over 100 columns. Let me assure you, it is not really fun to do.

The blog ‘Data Imaginist’¹⁴ by Thomas Lin Pedersen covers various topics around data visualisations. He is well known for his Generative Art, i.e. art that is computationally generated. But one of my favourite packages, `patchwork`, was written by him and gained immense popularity. It has never been easier to arrange multiple plots with such little code.

Lastly, I want to share with you the blog of Cédric Scherer¹⁵ for those of you who want to learn more about high-quality data visualisations based on `ggplot2`. His website hosts many visualisations with links to the source code on GitHub to recreate them yourself. It certainly helped me improve the visual storytelling of my research projects.

15.4 Join the X/Twitter community and hone your skills

Learning *R* means you also join a community of like-minded programmers, researchers, hobbyists and enthusiasts. Whether you have a X/Twitter account or not, I recommend looking at the #RStats¹⁶ community there. Plenty of questions about *R* programming are raised and answered on X/Twitter. In addition, people like to share insights from their projects, often with source code published on GitHub. Even if you are not active on social media, it might be worth having a X/Twitter account just to receive news about developments in *R* programming.

As with any foreign language, if we do not use it regularly, we easily forget it. Thus, I would like to encourage you to take part in Tidy Tuesday¹⁷. It is a weekly community exercise around data visualisation and data wrangling. In short: You download a dataset provided by the community, and you are asked to create a visualisation as simple or complex as you wish. Even if you only manage to participate once a month, it will make you more ‘fluent’ in writing your code. Besides, there is a lot to learn from others because you are also asked to share the source code. This activity takes place on Twitter, and you can find contributions by using #TidyTuesday¹⁸. Whether you want to share your plots is up to you, but engaging with this activity will already pay

¹⁴<https://www.data-imaginist.com>

¹⁵<https://www.cedricscherer.com/top/dataviz/>

¹⁶<https://twitter.com/search?q=%23rstats>

¹⁷<https://www.tidytuesday.com>

¹⁸<https://twitter.com/search?q=%23tidytuesday>

dividends. Besides, it is fun to work with datasets that are not necessarily typical for your field.



Epilogue

Congratulations! You made it through the book, or you started reading it from the back. Either way, I hope that spending your valuable time with this book was/is enjoyable, insightful and hopefully helpful in whatever grand or small questions you intend to answer through your research.

For my part, it has been a very enjoyable experience to put these materials together for those who need them. I also learned several new aspects about *R* I did not know before, for example, how to use *bookdown* and *quarto* to write this book. My biggest hope is to convince more researchers in the Social Sciences to try something new that can enrich their work. Whether you decide to adopt *R* for all your work, intend to move on to other languages like Python, or give up on it entirely, it does not matter. At least you tried, and you are more knowledgeable. Whenever we learn something new, it requires us to push ourselves out of our comfort zone, and I hope this book helped with that.

Whichever path you choose, I hope your research will have an impact, which, of course, does not depend on the tools you use.



Appendix

Comparing two unpaired groups

Table 15.1: Comparing two unpaired groups (effect size functions from package `effectsize`, except for `wilcoxonR()` from `rcompanion`)

Assumption	Test	Function	Effect size	Function
Parametric	T-Test	<code>t.test(var.equal = TRUE)</code>	Cohen's d	<code>cohens_d()</code>
	Welch T-Test	<code>t.test(var.equal = FALSE)</code>		
Non-parametric	Mann-Whitney U	<code>wilcox.test(paired = FALSE)</code>	Rank-biserial r or Wilcoxon R	<code>rank_biserial()</code> or <code>wilcoxonR()</code>

Comparing two paired groups

Table 15.2: Comparing two unpaired groups (effect size functions from package `effectsize`, except for `wilcoxonPairedR()` from `rcompanion`)

Assumption	Test	Function for test	Effect size	Function for effect size
Parametric	T-Test	<code>t.test(paired = TRUE)</code>	Cohen's d	<code>cohens_d()</code>
Non-parametric	Wilcoxon Signed Rank Test	<code>wilcox.test(paired = TRUE)</code>	Rank-biserial r or Wilcoxon r	<code>rank_biserial()</code> or <code>wilcoxonPairedR()</code>

Comparing multiple unpaired groups

Table 15.3: Comparing multiple unpaired groups (effect size functions from package `effectsize`)

AssumptionTest	Function for test	Effect size	Function for effect size
ParametricANOVA	<ul style="list-style-type: none">• <code>aov()</code> (assumes equal variances)• <code>oneway.test(var.equal = TRUE/FALSE)</code>	Epsilon squared	<code>eta_squared()</code>
Non-parametricWallis test	Kruskall-Wallis test <code>kruskal.test()</code>	Epsilon squared (rank)	<code>rank_epsilon_squared()</code>

References

- Aguinis, Herman, Ryan K Gottfredson, and Harry Joo. 2013. “Best-Practice Recommendations for Defining, Identifying, and Handling Outliers.” *Organizational Research Methods* 16 (2): 270–301.
- Altman, Douglas G. 1985. “Comparability of Randomised Groups.” *Journal of the Royal Statistical Society: Series D (The Statistician)* 34 (1): 125–36.
- Berger, Vance W. 2006. “A Review of Methods for Ensuring the Comparability of Comparison Groups in Randomized Clinical Trials.” *Reviews on Recent Clinical Trials* 1 (1): 81–86.
- Blanca Mena, M José, Rafael Alarcón Postigo, Jaume Arnau Gras, Roser Bono Cabré, and Rebecca Bendayan. 2017. “Non-Normal Data: Is ANOVA Still a Valid Option?” *Psicothema*, 2017, Vol. 29, Num. 4, p. 552-557.
- Boehmke, Brad, and Brandon Greenwell. 2019. *Hands-on Machine Learning with r*. Chapman; Hall/CRC.
- Buuren, Stef van. 2018. *Flexible Imputation of Missing Data*. CRC press.
- Cambridge Dictionary. 2021. “Concatenate.” <https://dictionary.cambridge.org/dictionary/english/concatenate>.
- Cohen, Jacob. 1988. *Statistical Power Analysis for the Behavioral Sciences*. New York, NY: Academic Press.
- Cohen, Patricia, Stephen G West, and Leona S Aiken. 2014. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. Psychology press.
- Cook, R Dennis, and Sanford Weisberg. 1982. *Residuals and Influence in Regression*. New York: Chapman; Hall.
- Cousineau, Denis, and Sylvain Chartier. 2010. “Outliers Detection and Treatment: A Review.” *International Journal of Psychological Research* 3 (1): 58–67.
- Dinno, Alexis. 2015. “Nonparametric Pairwise Multiple Comparisons in Independent Groups Using Dunn’s Test.” *The Stata Journal* 15 (1): 292–300.
- Dong, Yiran, and Chao-Ying Joanne Peng. 2013. “Principled Missing Data Methods for Researchers.” *SpringerPlus* 2 (1): 1–17.
- Field, Andy. 2013. *Discovering Statistics Using IBM SPSS Statistics*. Sage Publications.
- Gelman, Andrew, Jennifer Hill, and Aki Vehtari. 2020. *Regression and Other Stories*. Cambridge University Press.
- Ginkel, Joost R van, Marielle Linting, Ralph CA Rippe, and Anja van der Voort. 2020. “Rebutting Existing Misconceptions about Multiple Impu-

- tation as a Method for Handling Missing Data.” *Journal of Personality Assessment* 102 (3): 297–308.
- Grandstrand, Ove. 2004. “Durbin-Watson Statistic.” In *The SAGE Encyclopedia of Social Science Research Methods*, edited by Bryman Lewis-Beck Michael S. Thousand Oaks, California. <https://methods.sagepub.com/reference/the-sage-encyclopedia-of-social-science-research-methods>.
- Greenhouse, Samuel W, and Seymour Geisser. 1959. “On Methods in the Analysis of Profile Data.” *Psychometrika* 24 (2): 95–112.
- Grolemund, Garrett, and Hadley Wickham. 2011. “Dates and Times Made Easy with lubridate.” *Journal of Statistical Software* 40 (3): 1–25. <https://www.jstatsoft.org/v40/i03/>.
- Haerpfer, Christian, Ronald Inglehart, Alejandro Moreno, Christian Welzel, Kseniya Kizilova, Jaime Diez-Medrano, Marta Lagos, Pippa Norris, Edward Ponarin, and Bi Puranen. 2022. “World Values Survey: All Rounds - Country-Pooled Datafile. Madrid, Spain & Vienna, Austria: JD Systems Institute & WVSA Secretariat. Dataset Version 3.0.0.” World Values Survey Association. <https://doi.org/10.14281/18241.17>.
- Henson, Robin K. 2001. “Understanding Internal Consistency Reliability Estimates: A Conceptual Primer on Coefficient Alpha.” *Measurement and Evaluation in Counseling and Development* 34 (3): 177–89.
- Hoaglin, David C, and Roy E Welsch. 1978. “The Hat Matrix in Regression and ANOVA.” *The American Statistician* 32 (1): 17–22.
- Hochberg, Yosef. 1974. “Some Generalizations of the t-Method in Simultaneous Inference.” *Journal of Multivariate Analysis* 4 (2): 224–34.
- . 1988. “A Sharper Bonferroni Procedure for Multiple Tests of Significance.” *Biometrika* 75 (4): 800–802.
- Hu, Li-tze, and Peter M Bentler. 1999. “Cutoff Criteria for Fit Indexes in Covariance Structure Analysis: Conventional Criteria Versus New Alternatives.” *Structural Equation Modeling: A Multidisciplinary Journal* 6 (1): 1–55.
- Huynh, Huynh, and Leonard S Feldt. 1976. “Estimation of the Box Correction for Degrees of Freedom from Sample Data in Randomized Block and Split-Plot Designs.” *Journal of Educational Statistics* 1 (1): 69–82.
- Jakobsen, Janus Christian, Christian Gluud, Jørn Wetterslev, and Per Winkel. 2017. “When and How Should Multiple Imputation Be Used for Handling Missing Data in Randomised Clinical Trials—a Practical Guide with Flowcharts.” *BMC Medical Research Methodology* 17 (1): 1–10.
- Landis, J Richard, and Gary G Koch. 1977. “The Measurement of Observer Agreement for Categorical Data.” *Biometrics*, 159–74.
- Leys, Christophe, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. 2013. “Detecting Outliers: Do Not Use Standard Deviation Around the Mean, Use Absolute Deviation Around the Median.” *Journal of Experimental Social Psychology* 49 (4): 764–66.
- Little, Roderick J. A. 1988. “A Test of Missing Completely at Random for Multivariate Data with Missing Values.” *Journal of the American Statist*

- tical Association 83 (404): 1198–1202. <https://doi.org/10.1080/01621459.1988.10478722>.
- Lohr, Sharon L. 2021. *Sampling: Design and Analysis* (3rd Ed.). Chapman; Hall/CRC. <https://doi.org/10.1201/9780429298899>.
- Müller, Kirill, and Hadley Wickham. 2021. “Column Data Types.” <https://tibble.tidyverse.org/articles/types.html>.
- Nunally, J. C. 1967. *Psychometric Theory*. New York: Mc Graw-Hill.
- . 1978. *Psychometric Theory (2nd Edition)*. New York: Mc Graw-Hill.
- Rubin, Donald B. 1976. “Inference and Missing Data.” *Biometrika* 63 (3): 581–92.
- Schafer, Joseph L. 1999. “Multiple Imputation: A Primer.” *Statistical Methods in Medical Research* 8 (1): 3–15.
- Silge, Julia, and David Robinson. 2017. *Text Mining with r: A Tidy Approach*. O’Reilly Media, Inc.”.
- Stevens, James P. 2012. *Applied Multivariate Statistics for the Social Sciences*. Routledge.
- Stobierski, Tim. 2021. “Data Wrangling: What It Is and Why It’s Important.” Harvard Business School Online. <https://online.hbs.edu/blog/post/data-wrangling>.
- Tomarken, Andrew J, and Ronald C Serlin. 1986. “Comparison of ANOVA Alternatives Under Variance Heterogeneity and Specific Noncentrality Structures.” *Psychological Bulletin* 99 (1): 90.
- West, Stephen G, Aaron B Taylor, Wei Wu, et al. 2012. “Model Fit and Model Selection in Structural Equation Modeling.” *Handbook of Structural Equation Modeling* 1: 209–31.
- Wickham, Hadley. 2014. “Tidy Data.” *Journal of Statistical Software* 59 (1): 1–23.
- . 2021. *The Tidyverse Style Guide*. <https://style.tidyverse.org>.
- Wickham, Hadley, and Garrett Grolemund. 2016. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O’Reilly Media, Inc.
- Yang, Zhilin, Xuehua Wang, and Chenting Su. 2006. “A Review of Research Methodologies in International Business.” *International Business Review* 15 (6): 601–17. [https://doi.org/https://doi.org/10.1016/j.ibusrev.2006.08.003](https://doi.org/10.1016/j.ibusrev.2006.08.003).

