Qualitative Comparative Analysis (QCA) Using R: A Beginner's Guide

Getting started with R - Online Appendix

Ioana-Elena Oana, Carsten Q. Schneider, Eva Thomann
Cambridge University Press (2021)

Contents

1 Getting started with R *page* 2

1.1 A very short introduction to R 2

1.2 Dealing with datasets in R 13

Bibliography 25

1

Getting started with R

1.1 A very short introduction to R

This chapter introduces the reader to the most basic features of the R software environment and how to navigate it step-by-step, without dealing with actual QCA analyses yet. Its learning goals are for the reader to:

Box 1.1: Learning goals - Getting started with R

- gain a basic understanding on what R is and what it can be used for;
- become familiar with setting up R and RStudio for performing basic analyses (installation, setting the working directory, loading datasets, etc.);
- be able to write simple R commands and implement basic R syntax;

1.1.1 Basics

Before getting started, it is useful to know a few things about R and why we want to use it for performing QCA.

'Talking' to R

Unlike many other software for QCA, in R we use programming language to tell the software what we want it to do. This can be a bit of a challenge for the beginner user, since they will have to learn this language first. However, it also has several major advantages. First of all, it means that we can give R very precise and customized instructions, rather than being limited to a few standardized options. Hence, we are actually much more flexible to implement advanced tools for a state-of-the-art QCA than with other programs. Finally,

we can write down and store every command we gave R in a so-called script. This in turn means that our work is fully traceable and replicable, and that we can interrupt and continue any time. We can share the script, review, and check it for errors. All of this is not only an advantage, but even a necessity in times when strict transparency standards are becoming the norm in social science.

The good news is that the commands we use in R always follow a similar, actually quite simple, logic and structure. It is a bit like writing a recipe for cooking. (Spoiler: while R can do almost everything, unfortunately at the time of writing it cannot cook.) Let us imagine for a moment we would want to prepare a poached egg. We would tell R to boil an egg for three minutes without shell:

C 1.1

```
boil(egg, duration = 3, shell = FALSE)
```

That is, we begin by writing the actual command (also called "function") that specifies the activity we want R to perform. Then we use brackets to tell R, first, what "ingredients" to use for executing the command (for example a dataset, a condition, an outcome), and then specifying options. We use commas when specifying more than one option. Moreover, we often use the logical values TRUE or FALSE to say whether options should be turned on or not. In the example, we had an option for duration and one for excluding the shell. R considers all options we specify until we close the bracket. R functions often have many possible options, not all of which we need to specify because there are default settings. For example, the usual way of boiling eggs is with shell, so the default setting would probably be shell=TRUE. The R package documentation tells us the default settings for all options. Below we explain the logic and structure of different R commands in more detail.

Installing and running the software

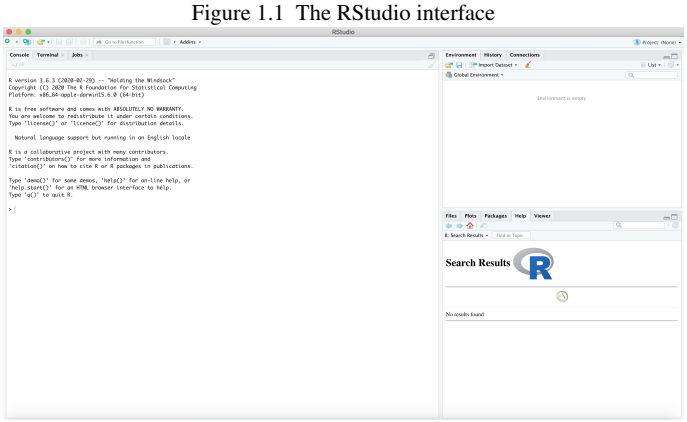
R is freeware and can be downloaded directly from the following link for Windows, Mac, or Linux:

https://cran.r-project.org/

Installation instructions are available online. In this book we use R within the open source RStudio interface, which can be downloaded for free after R was installed using the following link:

http://www.rstudio.com/products/rstudio/download/

Figure 1.1 The RStudio interface



Once both are installed, it is no longer necessary to separately start R; it suffices to open RStudio. When doing so, Figure 1.1 illustrates how the interface looks like. Initially, three windows appear. On the upper right, the *Environment* tab shows all the currently active objects in your working "environment" (see Figure 1.1). The *History* tab shows a list of commands used so far. On the lower right, there are a variety of tabs. The *Files* tab shows all the files and folders in the default workspace. The *Plots* tab displays all graphs that are produced. The *Packages* tab will list a series of packages or add-ons needed to run certain processes. The *Help* tab offers additional information.

On the left-hand side is the *Console*, where we see the so-called *output* (lower left-hand side in Figure 1.1. The output is basically where our activities are shown, as well as the results that they produce. It is also possible to type and execute commands in the console. However, beyond just giving R instructions, we want to save these instructions, repeat them any time we want, continue our work the next day, check and modify what we already did, show it to colleagues, replicate our work, and so on. To this end, we use the *Editor* in order to enter commands and documentation and save them in an *R script*. It is generally better to use the editor to enter code, rather than the console.

Scripts and comments

To open the editor and start a new script, click the following tabs in the top bar:

```
file -> new file ->  R script
```

Now, the editor window appears in the upper left-hand side of the interface; and the console is now below the editor. It is advisable to save the script frequently:

```
file -> save or save as
```

Especially as code grows, a good and clear script becomes indispensable so as to identify errors, add *comments* explaining what was done, and so on. It is advisable to use comments extensively in the script to explain what was done and why. Comments in the script start with # hashtag. R ignores everything after # in the line, that is, it does not treat it as a command. When using four hashtags (####) before and after the comment, Rstudio recognizes it as a section title to which one can navigate. To navigate to a title, there is a tiny bar on the lower end of the editor. Section titles are very useful especially for long scripts, in order to facilitate orientation.

The first command we routinely execute before starting new work with R serves to clear the workspace. To do so, we tell R to remove everything that is listed in the workspace using the rm() command, see below. After typing a command in the script, R will not run it until the code is sent to the console. To execute the command, put the cursor in the line where command is and either

- hit STRG and Enter or
- hit "STRG + R" (Windows) or
- hit "cmd + Enter" (Mac).

Several commands can also be selected and run at once. Additionally, RStudio offers flexible options for running specific chunks of code when clicking the tab Code in the top bar.

C 1.2

```
#Clear the workspace:
rm(list = ls())
```

1.1.2 Working directory, packages, and the help function

Working directory

In a next step, we tell R to read files from and save files to a specific location on the computer or drive we use, for example, the desktop or a project folder. This location is the so-called *working directory*. Typically we set the working directory in the folder where the datasets we use for the analysis are stored. The getwd() function tells us what the currently set working directory is. The command to set a new working directory is setwd(""), where the path to the folder can be entered within the quotation marks. However, the easiest way to set the working directory is to click the following in the top bar.

```
session -> set working directory -> choose directory
```

One can then browse to the file location that should be the working directory. After doing this, the path to the working directory will appear in the console. When copy-pasting this line into the script (without the >), the next time the working directory can be set just by running that line, without having to click and choose a directory.

C 1.3

```
# Identify current working directory
getwd()

# Set a new working directory
setwd("/Users/desktop")

# Display the content of (= the files in) the working directory
dir()
```

Packages

If R is the kitchen, *packages* are the kitchenware. There are packages for pretty much everything. For example, in this book we use the packages QCA (Dusa, 2018) and SetMethods (Oana and Schneider, 2018) for performing QCA with R. Several recent methodological innovations in QCA have advanced implementations in these packages for performing some of the most important and cutting-edge analyses in applied set-theoretic research. Both packages also provide access to a wide variety of example datasets for exercises, example scripts, and visualization tools. Where appropriate, we will introduce commands from further R packages relevant for QCA. A full documentation covering all possible commands and their options including examples is available on CRAN for each R package (https://cran.r-project.org/web/packages/), and it can be accessed via the help function (see below). These package documentations are structured alphabetically to treat command after command. Contrary to this, this book is structured along the analytic process of conducting a QCA. As a result, it neither provides a full nor systematic guide to all the relevant packages, nor does it cover everything that could be done.

Some packages are loaded permanently by default (so-called 'base' packages), while others must be installed. After starting RStudio and opening the editor, we can install the packages we need with the `install.packages()` function. We can list several packages to install. To this end, we use the `c()` function which combines its arguments (c stands for 'concatenate') and helps us list different elements. This function is often used with R. When listing more than one package, each package needs to come in quotation marks. For example, below we install the two packages QCA and SetMethods. We specify the option to also install "dependencies", that is, all other packages that might be

required for performing the functions in QCA and SetMethods. When running the command, sometimes a window pops up that asks us to choose the geographically closest server (CRAN mirror) to download the packages.

We need to install a package only once. After installation, we only need to load the packages when starting the R session using the library() function. Here, we do not need to use the quotes when listing the package to be loaded. We can see which packages are currently loaded at any time using the search() function. Having set a working directory and loaded the packages, we are ready to go.

C 1.4

```
# Install the packages QCA and SetMethods
install.packages(c("QCA", "SetMethods"),
dependencies=TRUE)

# Load the packages
library(QCA)
library(SetMethods)

# See which packages are loaded
search()
```

Troubleshooting with R

At this point, the reader might notice that especially in the beginning, things often do not work immediately in R. This can be a nuisance, but the good news is that none of these problems are ever unsolvable and that we are seldom alone with our R problems. Also, most of the problems will disappear quickly as we get acquainted with working with R.

When talking to R, it is good to remember that *R does not think on its own at all*. Most importantly, R will not have any tolerance for typos or the like. Also, R gets confused if we use the wrong symbols, such as a dot instead of a comma, or the wrong type of quotation marks, or if a bracket was forgotten. Moreover, the R language is case sensitive, so an uppercase letter A has a different meaning than the lowercase letter a. In short, in most cases when things do not work in R, it is because we did not talk to R in a way it understands. In these cases, R will give us error messages that might give us a clue about what went wrong. Checking for spelling errors, typos, and forgotten brackets or commas is always a good way to start problem-solving. If RStudio keeps crashing down, the user could check whether changing the file location, splitting up the codes into several, smaller scripts, or enlarging the 'plots' window (lower right) helps. If R does not react for a long time, it is possible that it is busy computing something. This could be the case for example when per-

forming logical minimization (as we will see in Chapter 4) or when factorizing Boolean expressions (as we will see in Chapter 2).

Conveniently, R has an extensive and in-built *help function*. To get help about a specific function, we just put a question mark (?) in front of a command. For example, to get help for the function `getwd()`, we simply type `?getwd`. The help function gives us all the relevant information and examples for a specific command. If we need help about a topic rather than a function, we can use two question marks: ??. If the topic we need help on consists of more than one keyword, we enter them into quotation marks, see below. The output of the help function will appear on the lower right window.

C 1.5

```
# Get help for a specific function (here: the getwd() function)
?getwd

# Get help for a keyword (here: the keyword "QCA")
??qca

# Get help about a topic (here: the topic "descriptive
    statistics")
??"descriptive statistics"
```

Finally, there is excellent *online documentation* as well as a variety of online help forums for pretty much any question that can be asked about R. It is worth simply searching for a solution on any online search engine. Throughout the book, we will selectively point out common error sources to the reader, as well as ways to deal with them. If everything fails, simply uninstalling and re-installing R, RStudio, and the relevant packages can sometimes solve the problem as a last resort.

1.1.3 Workspace, objects, and operators

Objects and workspace

If R is the kitchen, and packages are the kitchenware, then *objects* are the ingredients. R is an object-oriented language, meaning that there is an almost infinite variety of objects in R. Functions use the content of an object and produce results. For example, datasets are objects, variables are objects, and we can even save results as objects to recall them any time for further use. Data are contained in objects of different size and format (object classes) such as datasets, a list of numbers, or only a single number or a name. Objects are stored in the *workspace*. There (on the upper right-hand side of the RStudio interface), we can see all objects that are being used, for example, currently loaded datasets. We can create new objects as well, and label them as we wish. To generate an

object, we first give it a label and then use a backward arrow <- to define its content and store it in the workspace. For example, below we create an object named 'salute' that consists of the phrase 'hi there'. As soon as phrases are involved, we need to use quotation marks in order for R to recognize this as a string (i.e. a sequence of characters). After creating an object, we can return its content in the console by typing the name of the object and running it like we would run a command. Below, we type salute, and R tells us 'hi there'.

We are now beginning to have a conversation with R. We can also list several words or phrases using the `c()` function and quotation marks. For example, below we create an object that contains a short conversation: 'hi', 'what's up?', 'coffee please.' We have created a so-called *vector*: A vector is a sequence of data elements of the same basic type. Here, we were dealing with so-called *character strings*. When working with QCA, the content of the object or vector is more often numerical, of course. (It could also be logical, see below). For numerical objects and logical objects containing values of `TRUE` or `FALSE`, quotation marks are not needed. For example, below we create a numerical object with the number 37. When returning the object, R tells us "37". We can also list more than one numerical element in an object, again using the `c()` function. In the example below, we create an object labelled "count" that lists the numbers 1, 2, and 3. Finally, we can delete an object we have created at any point in time, using the `rm()` command. For example, below we removed the object "count" which will then no longer be visible in the workspace. As mentioned earlier, it is also possible to delete all objects from the workspace, which we do in the end. We like our workspace to be tidy and well-arranged.

Box 1.2: Classes of R objects

- Vectors: integer, numerical, logical, character;
- Matrix (two dimensions) - homogeneous content;
- Data frame (two dimensions) - heterogeneous content;
- List (list of different objects);
- Special classes introduced in specific R packages (e.g. class 'qca' - a QCA solution formula)

An example of a logical object would be, say, object `logicalobject` below, whose content is that the value of `numericalobject` is greater than 36. This, of course, is TRUE: 37 is indeed greater than 36. Logical objects typically take on the values `TRUE` or `FALSE`. It is useful to know that you can use the command `as.numeric()` to convert logical values to numerical values (1

for TRUE, 0 for FALSE). In QCA, we might want to use this, for example, to identify cases that fulfill certain conditions. If unsure which class we are dealing with (see different classes of R objecs in Box 1.2), we can check what type an object is using the `class()` function.

C 1.6

```
# Create an object "salute" with the phrase "hi there"
salute <-  "hi there"

# Return the object
salute

# Create an object containing a list of phrases
conversation <- c("hi", "what's up?", "coffee please.")
conversation

# Create a numerical object
numericalobject <-  37
numericalobject

# Create an object containing a list of numbers
count <- c(1, 2, 3)
count

# Remove specific objects from the workspace
rm(count)

# Create a logical object
logicalobject <- numericalobject>36

# Return a logical object as numerical
as.numeric(logicalobject)

# Identify the class type of the object
class(numericalobject)

# Delete all content (clear) in the whole workspace
rm(list=ls())
```

Operators

R integrates a variety of arithmetic and logical operators - see Table 1.1 for common examples. This means among other things that we can use it as a calculator. We can save the results of the operation(s) in an object; otherwise, they appear only in the console. Below we create an object z which is the operation 2 + 7. If we return z, R gives us the result 9. We can also combine several operations of the same type (arithmetic or logical). For instance, below we create an object x that is the result of the operation $((1+3)/7*18)^2$. The value of x is 105.7959. We can moreover use the operators to compare differing objects or perform operations on them. For example, we could add x to z and

Table 1.1 *Common operators in R*

Operator	**Meaning**
Arithmetic operators	
+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation
Logical operators	
&	AND
|	OR
==	EQUAL (attention: it is a double equal sign)
!	NOT
!=	IS NOT EQUAL
<	SMALLER THAN
>	GREATER THAN
<=	SMALLER THAN OR EQUAL
>=	GREATER THAN OR EQUAL

would get 114.7959. We can ask `R` whether z equals 12, and the answer will be FALSE. We can even ask `R` whether z does not equal x AND is smaller than x. The answer will be `TRUE`. Finally, we can perform several functions on numerical vectors. For example, we can derive a square root with `sqrt()`, compute the exponential function with `exp()`, or compute natural logarithms with `log()`. For example, below we derive the square root from z, which is 3.

C 1.7

```
# Perform arithmetic operations
z <- 2 + 7
z
x <- ((1+3)/7*18)^2

# Perform operations on objects
y <- z + x
y
z ==12
(z!=x)&(z<=x)

# Perform functions on objects
sqrt(z)
```

1.1.4 Maintaining, organizing, and documenting your software and work

Every now and then, a new version of R becomes available. Moreover, R packages are updated regularly in order to improve existing functions, integrate new innovations, or fix bugs. For the researcher, this means two things. First, it is a good idea to update R, Rstudio, and packages every now and then, especially after not using R for a while. Second, it is important make a note of which package version one has used at the time of analysis. Even though the packages QCA and setMethods should ensure backwards compatibility, it may be possible that functions written in an older package version can no longer be run with a new package version. It is possible, however, to use an older package version in order to replicate a given script, if the relevant package version is known.

One of the big advantages of doing QCA in R is that it offers the possibility to systematically and precisely document our work. Using R scripts, we can replicate analyses at any time and check our own as well as the work of others. In order to capitalize on this advantage, we recommend several good practices. First, it makes our lives considerably easier if we store all of the source files for a project in one directory and use relatively short and easy-to-remember paths to access them. Moreover, it is a neat way of making analytic steps and decisions traceable to save different versions of R scripts using version control that allows us to track changes over time—for example, by including the date in the file name. Rather than saving all codes in one huge file, it may be beneficial to save different chunks of code in different files. Within R scripts, it is a good idea to begin the script with a sentence or two on what it does; use headings in order to structure the document according to different analytical steps; and name and style code consistently. Moreover, researchers should (concisely) explain the rationale for analytic steps and decisions using the comments (#). It is also generally advisable to give documents, variables, cases, objects, et cetera labels that are short, intuitive, and not unnecessarily complex. To avoid confusion of objects, always start the a new session with a clean environment using the rm(list = ls()) function. An online search of good practices for writing R code can be very instructive. Following such practices considerable enhances the user experience.

1.2 Dealing with datasets in R

This book does not cover the process of data collection. Rather, in what follows we elaborate on how to handle data that is already collected and stored in a dataset file using a popular format (for example .csv,.sps, .sas). We can easily import such data, store it as an object, and use it for various analyses in R.

As previously mentioned, objects created in R can take different levels of complexity. Objects can store single numbers, variables containing multiple values, and even data frames containing multiple variables (see Box 1.2). Thefore, we usually import datasets as so-called data frame objects into R. Data frames are 2-dimensional objects in that they look like a table with columns and rows on which values are stored. In a data frame the columns represent different variables, while the rows represent different units of analysis (or cases) on which information is collected.

1.2.1 Opening and saving datasets

One of the most popular formats for dataset files is '.csv' (comma-separated values). Depending on the settings specified, this format uses a comma or a semicolon to separate values in a delimited, table-like text file. Most of the exercises in this book and its online appendix will be based on datasets of this format. Typically we will need to import the dataset into R before starting the analysis. We now explain how to do so by importing .csv files. Additionally, Box 1.3 provides more advanced guidance on how to import datasets of other popular formats. Finally, to make the reader's life easier, we have actually integrated all datasets used in this book directly as objects into the `SetMethods` package. Further below we will explain how to find and retrieve them in order to replicate the scripts in this book.

To import a dataset, we should first make sure to set the working directory to the folder where we have stored the dataset (see Section 1.1.2). To load a .csv file from that folder we use the function `read.csv()`. Inside the brackets, we input the name of the file and its extension using quotation marks (here: "data.csv"). We can store this into an object named 'mydata".

Beside the name of the data file, the `read.csv()` function has some other options that are important. The `header` option is a logical option to define whether the first line of the file contains the names of the variables, as is usually the case in empirical datasets. It can take on the values of `TRUE` or `FALSE`. The `row.names` option set to value 1 allows us to tell R that the names of our rows (that is, the labels of our cases) are located in the first column of the file. The reader should note that while .csv files are usually delimited by a "," and store

decimals as a ".", it happens that .csv files are sometimes stored using other separators and decimals. The options SEP (separator) and `dec` (decimal) allow us to specify these. In case of problems with importing a .csv file, it is worth trying to change these settings and to see if it resolves the issue. There are other options which we do not introduce here, but these can be all checked by just using the `?` in front of the function name, as shown below. We can save a dataset, for example after having recoded variables or calibrated sets, using the `write.csv()` function, which has basically the same options for specifying the format of the .csv file.

C 1.8

```
mydata <- read.csv("asylumraw.csv", header = TRUE, row.names =
    1, sep = ",", dec = ".")
```

C 1.9

```
?read.csv
```

Box 1.3: Going further - Importing dataset files in other formats

For importing files in other formats, we can install the "foreign" package, which contains specific functions for each file type:

C 1.10

```
install.packages("foreign")
library(foreign)
```

- **Octave:** `read.octave()`
- **STATA:** `read.dta()`
- **SAS:** `read.ssd()`
- **SPSS:** `read.spss()`
- **Minitab:** `read.mtp()`
- **Systat:** `read.systat()`

Finally, as mentioned earlier, we can also retrieve some dataset files directly from R packages. For example, the `SetMethods` package contains plenty of crisp, fuzzy, and multi-value datasets, which we can use for QCA exercises. Among these there are also the two example studies that we introduce below. We can import these datasets into R by just using the `data()` command with the name of dataset (as saved in `SetMethods`).

C 1.11

```
data("LIPC")
```

1.2.2 Create and save datasets from scratch

We can also create data frames from scratch and save them as .csv files in R. For example, using the `c()` function, we can create two numerical vectors, x and y, and a logical vector, z, each storing 4 values. Since the vectors all contain the same number of values, we can combine them in a data frame called 'newdt' (or whichever name you want to give to your data) using the function `data.frame()`. The functions `names()` and `rownames()` allow us to look up and change the names of the columns or the rows of a dataset. By simply entering the name of the dataset in the brackets, we get the current labels of the variables or cases. If we want to change these labels, we can simply overwrite the previous column names by assigning them new labels using the `c()` function. When changing the names of the columns and of the rows we must be careful to input within the `c()` functions exactly as many values as there are columns or rows. Finally, we can save this new dataset in a .csv file by using the `write.csv()` function. We first input the name of the object within which we stored the dataset (here: newdt) and then we input the name that we want to give to the .csv file within quotation marks. By running this function, we store the new 'my3vectors.csv' file in the working directory.

C 1.12

```
# We create 2 numerical vectors
x <- c(1, 2, 3, 4)
y <- c(4, 7, 100, -2)

# We create 1 logical vector
z <- c(TRUE, FALSE, FALSE, TRUE)

# We create the dataframe newdt containing the three vectors
newdt <- data.frame(x, y, z)

# We can look up the current labels of the columns and rows
names(newdt)
rownames(newdt)

# We can rename the columns
names(newdt) <- c("vector1", "vector2", "vector3")

# We can rename the rows
rownames(newdt) <- c("c1","c2","c3","c4")

# We can save the data frame as a .csv file
write.csv(newdt, "my3vectors.csv")
```

1.2.3 Inspecting and describing data

The raw dataset contains the values that the cases, or units of analysis, take on for the different variables that go into a QCA analysis. Following Gerring (2004), we define the unit of analysis as a "spatially bounded phenomenon—e.g., a nation-state, revolution, political party, election, or person—observed at a single point in time or over some delimited period of time".

After having our dataset loaded into R we can start inspecting and describing the dataset. Usually, the first thing we do after loading the dataset is to look at the first few rows of the data in order to check whether everything was imported properly. For doing this we use the `head()` command with the name we gave earlier to the dataset object (here: newdt). This will display in the console window in RStudio the first 5 rows of the data, together with the names of these rows (denoting the cases or units of analysis that are compared in the QCA) and the names of the columns (denoting the variables or sets). We can also check the names of the columns and of all the rows separately by using the `names()` and the `rownames()` functions. We can check how many columns and how many rows a dataset has by using the `dim()` function. Additionally, we can also check the structure of the dataframe object we have created (here: newdt). The function `str()` will tell us that the object called 'newdt' is a dataframe that includes 4 observations (rows) on 3 variables (columns). The function will also list the three columns (vector1, vector2, and vector3) separately. The dollar sign $ helps us to retrieve only certain elements from a dataset (or from any other complex object, for that matter). The dollar sign basically tells R to choose, from a given object (such as our dataset), a certain element. Therefore, if we want to have a look only at variable vector1 in the dataset, we simply write the name of the dataset, 'newdt', followed by a $ and then the name of the variable, 'vector1'. This will give us the values of all cases on this variable.

C 1.13

```
# Checking the first 5 rows of the data
head(newdt)

# Checking the names of the columns
names(newdt)

# Checking the names of the rows
rownames(newdt)

# Checking the dimension of the dataset
dim(newdt)

# Checking the structure of the dataframe object
str(newdt)
```

```
# Looking at variable x in our dataframe
newdt$vector1
```

We can narrow down our inspection even more and retrieve specific values from a dataset. One way to do this is by using square brackets after the name of the object. To do this, we either need to know the order of the cases (rows) and columns (variables) in the dataset that we want to see, or their names. For example, we can use the names of the respective cases or variables directly in the square brackets, as shown below. We use a comma within the square brackets to delimit the rows (before the comma) and the columns (after the comma). However, sometimes we just want to see all values for a variable, or for a case. This is possible by simply using a blank space. For instance, if we want to retrieve all the values for case c1 in our dataset, we simply place c1 before the comma (case c1) and leave the space after it blank (all the columns). Alternatively, if we want to retrieve the values of all cases on the variable "vector1", we leave the space before the comma blank (all the rows) and place vector1 after it to specify the column..Alternatively, say that we want to know the value of the fourth case in the second variable. We can also retrieve multiple rows and columns by using the `c()` function (for combining different columns/rows) or a ':' (for ranges of columns/rows) in the square brackets. In the example below, we first look at the values of all cases in the dataset on the first and the third variable; and then, from the first to the third variable.

C 1.14

```
# Checking the values of case 'c1' on all variables
newdt["c1",]

# Checking the values of variable 'vector1' on all cases
newdt[,"vector1"]

# Checking the value in the fourth row and second column
newdt[4,2]

# Checking the values on the second row of the data
newdt[2,]

# Checking the values of the third column of the data
newdt[,3]

# Checking the value on the first and the third column of the
    data
newdt[,c(1,3)]

# Checking the values from the first to the third column
newdt[,c(1:3)]
```

Another way of inspecting a dataset is to use the `subset()` function, if we

can define specific subsets of our dataset for which we are looking for information. For example, say that we look at the dataset by Sager and Thomann (2017) (saved as 'asylumraw.csv').[1] We want to look only at those cantons that have a strong left (variable 'L') AND a strong right (variable 'RM'), in order to see to what extent they integrate asylum seekers into the labor market. The `subset()` function allows us the specify a range of cases we want to look at in a given dataset using logical expressions (here: those with values above 0.5 for the sets L and RM), and specify which variables we want to look at using the select option.

C 1.15

```
subset(mydata, (L > 0.5) & (RM > 0.5), select = INT)
```

After having imported the dataset properly, we can start to describe and summarize the data. Here we introduce the most common functions for performing basic descriptive statistics (Wonnacott and Wonnacott, 1990). Later in in Chapters 2, 3, and 4 we will look at various visual ways of exploring data. The generic function `summary()` implemented in base `R` is a really useful feature for summarizing the data. It gives us the minimum, maximum, median, mean, first, and third quantiles for each numerical variable. For variables with character or logical values, `summary()` indicates the frequencies for each of the available values. Other common ways for numerically describing our data are the mean (`mean()`, standard deviation (`sd()`), sum of all values in a variable (`sum()`), and the frequency of each value in variable (`table()`). We usually use these four functions for specific variables, while we can apply `summary()` to describe either the entire dataframe, or a selected variable therein. Notice that we use the dollar sign $ in order to access an element of an object. For example, below we tell `R` to take the column 'vector1' in the data 'newdt' using the dollar sign.

C 1.16

```
# Check descriptive statistics for the entire dataframe
summary(newdt)

# Check descriptive statistics for column vector1
summary(newdt$vector1)

# Check the mean of column vector1
mean(newdt$vector1)

# Check the standard deviation of column vector1
sd(newdt$vector1)
```

[1] Note that, for didactical purposes, this is a modified version on the original dataset.

```
# Check the sum of the values in column vector1
sum(newdt$vector1)

# Check the frequency of each value in column vector1
table(newdt$vector1)
```

1.2.4 Adding, renaming, deleting, and recoding variables and cases

After having a dataset imported into R we may want to add, recode, rename, or delete variables or cases. To add a new variable (column) to a dataset we can simply write the name of the dataset, the dollar sign $, and the label of the new variable we would like to add. Using the `c()` function we can insert into this new column as many values as there are rows in the dataset. We can also *rename* a variable by using the `names()` function followed by square brackets where we enter the number or name of that particular variable; we then assign a new label to the column using the backward arrow and quotation marks. Finally, we can delete a variable (say, column 4) by just placing into our dataset object the old dataset object in which we delete the column by subtracting it within the square brackets, after the comma.

C 1.17

```
# Adding a column called 'vector4' to the newdt dataframe
newdt$vector4 <- c("red","blue","green","yellow")
head(newdt)

# Renaming the 4th column in newdt as 'colours'
names(newdt)[4] <- "colours"

# Deleting the 4th column in newdt
newdt <- newdt[,-4]
```

Recoding variables can be performed by using the name of the dataframe and square brackets. Before the comma within the square brackets we write the conditions that values have to fulfill to be changed, while after the comma we write the name of the column in which we recode. For example, the code below replaces all values of `FALSE` in column 'vector3' with values of `TRUE`. All values larger than or equal to 5 in column 'vector3' are also replaced with the expression 'bignumber'. We can also just recode specific values in a dataset by specifying their location using the square brackets. This can be done by either writing the number of the row (before the comma) and of the column (after the comma) of that particular values, or their names using the quotation marks.

The `QCA` package offers another function for recoding variable, which enables us to specify complex rules for recoding. The `recode()` function allows

us to tell R which variable you want to recode, and according to what rules you want to attribute (=) which value. Several rules can be combined using semicolons ; . Rules can be:

- a single value, for example 1=0
- a range of values, for example 2:5=1
- a set of values, for example 6,7,10=2
- everything that is not specified in the defined rule: for example, else=0

In the example below, we used the dataset by Sager and Thomann (2017). Suppose that we want to recode the outcome set, cantonal policies enabling labor market integration, which is measured using an index called "LABRES" ranging from 0 to 15. We could recode LABRES into LABRES2, such that values of 1, 3, and values between 5 and 7 obtain the value 1; values of 4, 8 and 9 obtain value 2; and all other values retain the same values as the old variable (else=copy).

C 1.18

```
# Recoding FALSE to TRUE in column vector 3
newdt[newdt$vector3==FALSE,"vector3"] <- TRUE

# Recoding numbers >= 5 in 'vector2' with the expression '
    bignumber'
newdt[newdt$vector2>=5,"vector2"] <- "bignumber"

# Recoding a particular values
newdt[2,3] <- 'anewvalue'
newdt["c1","vector1"] <- 'anewvalue'

# Recoding a variable using the recode() function
mydata$LABRES2 <- recode(mydata$LABRES, "1,3,5:7=1; 4,8,9=2;
    else=copy")
```

Adding cases (rows) to a dataset can be done with the `rbind()` function. With this function we replace our dataset with the dataframe to which we bind the values that the new case (here: c5) has on the different variables using the `c()` function. We must add as many values as there are columns in our data. Deleting a case (row) from the data is just like deleting a column: we simply subtract the number of the row before (rather than after) the comma within the square brackets.

C 1.19

```
# Adding a new row named 'c5' to newdt
newdt <- rbind(newdt, "c5" = c(1, 2, FALSE))
head(newdt)

# Deleting the 5th row from newdt
newdt <- newdt[-5,]
```

1.2.5 Dealing with missing values

When working with empirical datasets, it is often the case that the dataset does not contain values for some variables. Researchers working with data they collected themselves for a small- or intermediate-N QCA might not encounter this situation, in which case the reader can safely ignore this section. However, as soon as we use secondary datasets, especially large datasets such as survey data, it is imperative to check how complete the dataset is and decide on how to deal with missing values. This is important not least because, unlike some statistical analyses, truth table analysis does require us to have an empirical value for all sets included in the analysis. QCA cannot attribute cases with missing values to the truth table; they will result into altogether missing cases.

There are different approaches to how missing values are coded in social science datasets, often depending on the statistical software for which they were originally intended. For example, missing values might be denoted by a value of NA (or na) which means "no answer". Or a special value may have been reserved for them that does not otherwise occur in the data under question, such as 99. Sometimes we find simply a dot (.) for missing values. If available, the first step would be to check the codebook for the given dataset to identify the value it uses. If a codebook is not available, we might want to visually look at the dataset for example by using the head() function to see which value is used for missings. R recognizes values of 'NA' for missing values. If missing data is denoted by a different sign, '–' or '99' for example, we have to recode them into NA. Using the is.na() function, we can find out whether there are missing observations in our dataset or for a specific variable or case (TRUE = missing, FALSE = not missing). In a very similar vein, the complete.cases() function tells us whether the cases are complete, that is, they do NOT contain missings (TRUE = no missing values, FALSE = has one or several missing values).

If these functions tell us that there are missing values in the dataset, we may want to obtain the number and/or percentage of cases that are complete or have missing values. We do this by combining the above functions with the as.numeric() function. This is a useful feature to transform logical values (TRUE and FALSE) into numerical values (1 or 0) which we can then count. Even more precisely, combining as.numeric() with is.na() another thing we can do is to obtain the number or percentage of cases with missing values on a specific variable. Often however, we also want to know the names of cases that have missing values on a specific variable. To do so, we can again identify the cases with missing values with is.na() and then use the subset() function that allows us to return values for a specific subset of our dataset. We then simply

ask R to tell us the names of the rows in this subset of the dataset. We can also check this for several variables simultaneously. In the example below we first use as.numeric(is.na())for identifying the cases with missing values for variable 'SPPMAJ', and then identify the names of those cases. In the same vein, using complete.cases() we can identify those cases that have no missing values on one or several variables.

There are different ways how to resolve the issue of missing values analytically. All of them raise broader questions of measurement validity and validation (Adcock and Collier, 2001). One way is to only work with those observations for which all data is available. We can simply only include cases with no missing values on any variable in the dataset using complete.cases(). Alternatively, we can first identify cases with missing values for certain variables (in the example below: SPPMAJ), and then only include only those cases without missing values in the dataset. We can then save the data containing no missings in a new dataset, see 1.2.1.

While preserving measurement validity, excluding all cases with one or several missing values has implications for case selection. It can lead the set of cases included (the sample) to be systematically distorted due to measurement error or other biases. Sometimes it can be appropriate and feasible to use techniques for establishing measurement equivalence for missing values, for example by resorting to context-specific indicators from additional data sources (Adcock and Collier, 2001). If sets are composed of several indicators, then the researcher can apply set-theoretic techniques for structuring concepts in order to aggregate indicator sets, such as the use of substitutable indicators (logical OR) or family resemblance structures (Goertz and Mahoney, 2005). We explain how to do this in Chapter 2. However, it is important to be aware of the analytic reasoning behind and consequences of such decisions, as they can lead to skewed set memberships and weaken measurement equivalence. Missing data always poses challenges and trade-offs. It is imperative to take these into account and establish full transparency (Wagemann and Schneider, 2015).

C 1.20

```
# Number of missing values per row
rowSums(is.na(newdt))

# Number of missing values per column/variable
colSums(is.na(newdt))

# Convert to missing data
newdt[newdt$vector2==4,"vector2"] <- NA

# List rows that have missing values
```

```
newdt[!complete.cases(newdt),]

# Creating a new dataset without missing data by listwise
    deletion
newdt <- na.omit(newdt)

# Check for presence of missing data in dataset
is.na(newdt)

# Check for the presence of missing data in specific variable
is.na(newdt$vector1)

# Check for the absence of missing data in dataset or variable
complete.cases(newdt)
complete.cases(newdt$vector1)
```

C 1.21

```
# Recode missing values (here: -99) into format suitable for R (
    NA)
mydata[mydata==-99] <- NA

# Check for the presence of missing data in dataset
is.na(mydata)

# Check for the presence of missing data in specific variable of
     dataset
is.na(mydata$SPPMAJ)

# Check for the absence of missing data in dataset or variable
complete.cases(mydata)
complete.cases(mydata$SPPMAJ)

# Obtain the number of cases with missing values (0) and with no
     missing values (1)
nomissings <- as.numeric(complete.cases(mydata))
table(nomissings)

# Based on this, obtain the percentage of cases with
# missing values (left-hand side of crosstable) and
# without missing values (right-hand side)
prop.table(table(nomissings))

# Identify the names of the cases that have missing values on a
    specific variable:
varmiss <- as.numeric(is.na(mydata$var))
rownames(subset(mydata, varmiss==1))

# Include only complete cases in dataset
nomissings <- as.numeric(complete.cases(mydata))
mydata = mydata[nomissings== 1,]
```

Bibliography

Adcock, Robert, and Collier, David. 2001. Measurement validity: a shared standard for qualitative and quantitative research. *American Political Science Review*, **95**(3), 529–546.

Dusa, Adrian. 2018. *QCA with R*. Springer.

Gerring, John. 2004. What is a case study and what is it good for? *American political science review*, **98**(2), 341–354.

Goertz, Gary, and Mahoney, James. 2005. Two-level theories and fuzzy-set analysis. *Sociological Methods & Research*, **33**(4), 497–538.

Oana, Ioana-Elena, and Schneider, Carsten Q. 2018. SetMethods: an Add-on R Package for Advanced QCA. *The R Journal*.

Sager, Fritz, and Thomann, Eva. 2017. Multiple streams in member state implementation: politics, problem construction and policy paths in Swiss asylum policy. *Journal of Public Policy*, **37**(3), 287–314.

Wagemann, Claudius, and Schneider, Carsten Q. 2015. Transparency standards in qualitative comparative analysis. *Qualitative & Multi-Method Research*, **13**(1), 38–42.

Wonnacott, Thomas H, and Wonnacott, Ronald J. 1990. *Introductory statistics*. Vol. 5. Wiley New York.