Qualitative Comparative Analysis: An Introduction to Research Design and Application
Patrick A. Mello

Online Appendix

R Manual

Mello, Patrick A. (2021) *Qualitative Comparative Analysis: An Introduction to Research Design and Application,* Washington, DC: Georgetown University Press, Online Appendix.

Version: June 2021

The great (or perhaps infuriating) thing about R is that there's always more to learn.

Robert I. Kabacoff [1]

This online appendix contains a concise R Manual, focusing on functions to prepare and analyze data and to conduct set-theoretic analyses. For this, we will use the R software environment (R Core Team 2020) and the RStudio text editor (2020), as well as the R packages "QCA" (Duşa 2019) and "SetMethods" (Oana et al. 2020; Oana et al. 2021). This is complemented by an R Script that contains all of the code used in the R Manual (see the instructions below).

As indicated in the above quote by Robert Kabacoff, one is hardly ever done learning R. Yet, for new users, the main hurdle is starting with R in the first place. With this in mind, the R Manual aims to *get you started* towards conducting your own QCA study and to reproduce the analytical steps introduced in this book (Mello 2021). Unlike texts that assume prior knowledge of R, the Manual devotes adequate space to basic questions of working with the R editor, getting the data into shape, and many of the issues that arise during the analysis.

As for QCA functions, this manual covers one complete QCA *research cycle* as presented in this book (Figure 1.1), including the calibration of different kinds of raw data, the creation of truth tables, their minimization to derive solution terms, and ways to customize the analysis. That

[1] Kabacoff (2015, 532).

said, the R Manual cannot replace a comprehensive introduction to R and its functionality. Nor can we cover all of the potential ways of working with the QCA packages. Hence, I recommend consulting at least one general introduction to R. There are plenty of textbooks available and these are often tailored towards usage in a specific area of research (e.g. Field et al. 2012; Gaubatz 2015; Imai 2017; Kabacoff 2015; Pollock and Edwards 2018). For questions about further analytical functions and alternative settings readers are advised to consult the comprehensive book-length guide to the "QCA" package by Adrian Duşa (2019), and the forthcoming "beginner's guide" to QCA using R, by Nena Oana, Eva Thomann, and Carsten Schneider (Oana et al. 2021).

Another way of getting started with R is by using one of the many freely available online resources, as for example the "swirl" package.[2] There are also various online discussion forums, where you can find answers to all kinds of R-related questions.[3] When errors occur, it is often a good idea to start looking for a solution by copying the error message into a search engine. Most often, others experienced similar problems and there may be an easy fix for the issue, especially when it is about common tasks like installing R and its packages, reading data, or working with data frames.

Installing R and RStudio

Before we can start working, we first need to *install* R and RStudio. R is the software environment and RStudio is an editor that enhances the use of R with a graphical interface and many integrated functions. We begin by installing R and *only then* install RStudio (typically, an open-source license for RStudio suffices), because the latter runs on the former. The latest versions of the software can be accessed at the following websites, which also contain platform-based instructions (depending on whether you use Windows, Mac OS X, or Linux):

R: https://cran.r-project.org/
RStudio: www.rstudio.com/products/rstudio/

One thing to keep in mind is that there are frequent *updates* for both R and RStudio, and also for many of the R packages. This means that you should regularly check for updates, especially when you have not used the software in a while. Occasionally, this may lead to some interruptions in your workflow, as when functions are discontinued or renamed, or when default settings in a package are changed. In most cases, however, updating should work seamlessly and there's backward compatibility, in that older functions and previous names are still operable.

[2] This can be reached at: https://swirlstats.com/.

[3] One of these is: https://stackoverflow.com/.

Mello, Patrick A. (2021) *Qualitative Comparative Analysis: An Introduction to Research Design and Application,* Washington, DC: Georgetown University Press, Online Appendix.

Online Material

Complementary online material can be found on my website https://patrickmello.com under the drop-down menu "Publications" and "Qualitative Comparative Analysis: Research Design and Application". The files include an R Script ("RManual.R") and data sets that will be used throughout the following sections.

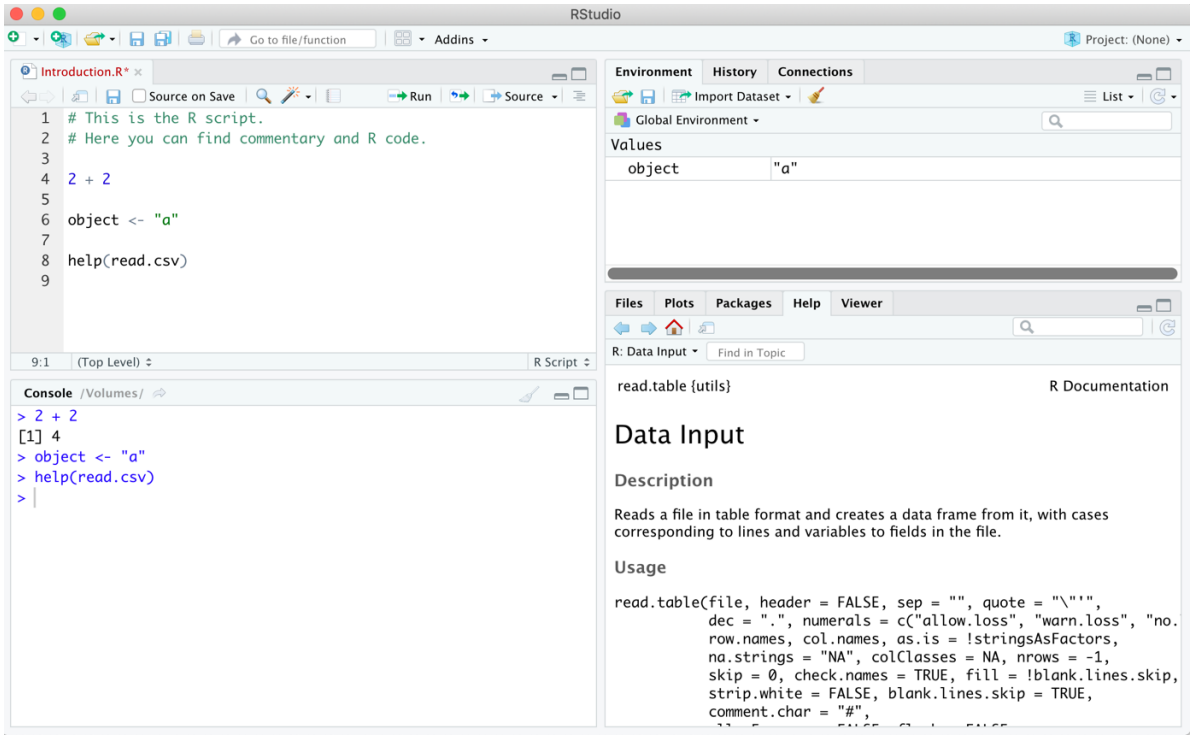
Getting to know R and RStudio

Once we have installed R and RStudio, we can start a new session by opening RStudio. It is not necessary to open R, because RStudio does that for us. So, for all of the future steps we will always work from within RStudio.

When we open RStudio, we see four windows, similar to what is shown in Figure 1. The top left window shows our *R script* (open the file "RManual.R"). Eventually, this should include our entire QCA analysis. Below, there is the console. This area shows the *output* that R produces, based on the commands that we enter. So, whenever we ask R to compute something, the result will show up in the console window.

In the top right corner, we can see several tabs, including *Environment*. This area shows objects we created, or data sets we read into R. Finally, in the lower right corner there is another window with several tabs including *Help*. Here we can find documentation on R functions. There are many more functions, but the ones mentioned should suffice to get us started.

Figure 1: Layout of RStudio



You can reproduce the example in Figure 1 by creating a new script (select *New File* and then *R Script* from the drop-down menu *File*) and writing the commands as shown in the illustration. Note that the first two lines show *commentary*. This is always preceded by a hash sign (#). In short, anything that you write in the same line after a hash is not treated as code and will not be processed by R. RStudio highlights this by assigning a different color to commentary, so it can be distinguished from code (the standard color is green, but like everything this can be customized under *Preferences*).

Commentary is part of good coding practice. Writing short comments before or after a function helps us remember what a certain part of the R code was meant to achieve and why we chose certain values. This may become important when you revisit your own code weeks or months after you have written it and try to understand why a particular command was needed. More importantly, it makes your code *transparent* and *reproducible* to others. Many academic journals now require the submission of R code, together with any data used for empirical analysis. Hence, we can spare ourselves some extra work by writing transparent and well-documented code in the first place, so that the code does not have to be rewritten or heavily edited once a manuscript is accepted.

The next few lines in Figure 1 include small examples of R code. For example, you can use R to do mathematical calculations, the result of which will be shown in the console. Here, we write "2 + 2" and then "run" this code to get the result.

Running code can be done in two ways. You can either select a part of the code and manually click on the *Run* button at the top of the script window. Or you can select the code and then press "Command" and "Enter" simultaneously on your keyboard, which usually will be the faster approach. Note that when the cursor is placed in a line, then the entire line of script will be run. If you want to run more code, then you need to select a larger part of the script. Conveniently, you can also select and run an entire R script.

For the remainder of this chapter and unless stated otherwise, text shaded in gray is meant to be written in the upper left script window of RStudio (or copy-pasted there from this text). From RStudio, you can then run the code and manipulate it to practice the different functions and arguments along the way.

The R language is based on *objects*. We create objects by using a leftward arrow sign. Once an object is created, it appears in the "environment" shown in the top right window. For example, we can create the object "A", which shall equal "2 + 2":

```
A <- 2 + 2
```

We can also assign names to objects, by placing the characters in quotation marks:

```
City <- "Berlin"
```

Note that R is *case-sensitive*, so "City" and "city" are treated as different objects (calling the second object will return an error because we have not created it). You can always list everything in your environment with the following function:

```
ls()
```

This yields the following output in the console:

```
[1] "A" "City"
```

Note that the number in brackets "[1]", shown in the output given in the console, is an index number. In this example, there are only two objects but with more objects, the result may run over several lines, each of which would be preceded by an index number, so that separate objects can be clearly identified.

Working Directory

When starting to work with R, we should set a *working directory* from which R will read data and scripts, and where it will save any output that we generate. To do this, we first check what the current working directory is, using the function *getwd*:

```
getwd()
```

This should return a pathname on your computer, ideally with all relevant files in it. We can check the folder content with *list.files*:

```
list.files()
```

To change the directory, we use *setwd*:

```
setwd("/mydirectory/mydata")  # Adapt this to match your own pathname
```

Note that R *does not recognize subfolders*, so all relevant files you want to work with in a single R session have to be in the same folder. Hence, it is a good idea to create a dedicated R project folder and place all data files and scripts there before working with R. Errors often come from the folder structure or from misspelled file names. In both cases, R will not find the right data file, and hence cannot read the data to begin with.

Should you run into troubles when setting your working directory, note that you can also do this manually in "click-and-point" fashion. Simply navigate to the lower right window and the rider *Files* (depending on your operating system, this may also be in another window). In the files view, you can steer to the desired folder on your computer, then select *More* and *Set As Working Directory*. Use the *getwd* function from above to confirm your settings.

Reading Data

To work with R, we need to import our data first. R can read various file formats, but for QCA, the most common format are comma-separated files (.csv). Here, we use the function *read.csv* to read the file "FuzzyData.csv" (found in the online material) and create a new data frame that we call "fuzzy.data" (we could assign any other name). The data should appear in our *Environment* in the upper right corner of RStudio.

```
fuzzy.data <- read.csv("FuzzyData.csv")
```

However, when we examine the new *fuzzy.data* object by calling its name and *run* (or by clicking on it in the environment), we can see that the code from the previous line has not read the data correctly. All of the information has been read as a single column. We can solve this problem by stating that the data is separated by *semi-colons* rather than commas:

```
semi.colon.data <- read.csv("FuzzyData.csv", sep = ";")
```

The result looks better, but we can see that the name column for our cases has been treated as a variable column. To change this, we specify which column includes case names. Most often, this will be the first row. We modify our script once more and now we should have data with 26 cases and 4 conditions (C1, C2, C3, and OUT) in our environment:

```
fuzzy.data <- read.csv("FuzzyData.csv", sep = ";", row.names = 1)
```

Note that another potential source of error may be the incorrect treatment of decimals. Depending on language settings, decimals are sometimes indicated by a comma, and at other times by a period. This may cause an incorrect reading of the data. We can solve this with the *dec* argument:

```
dec.comma.data <- read.csv("FuzzyData.csv", dec = ",")
```

The arguments *sep* and *dec* are not always needed (often, the standard settings correctly read the data frame). However, in case your data comes out wrong, the first thing to check are these two arguments, together with *row.names*.

Installing Packages

R comes with a multitude of different functions. Yet, one of the great advantages of R is the existence of user-defined *packages* that include functions and code for specific purposes. Rather than writing a long patch of code each time we seek to conduct a sequence of analytical steps, a function from a specific package may do just that with a single command.

To work with packages, we first need to install them on our local computer system. For our purposes, we use the packages "QCA" (Duşa 2019) and "SetMethods" (Oana et al. 2020). We install these separately, using *install.packages*:

```
install.packages("QCA")
install.packages("SetMethods")
```

While packages only need to be installed once, they must be loaded at the start of each session, using the *library* function:

```
library("QCA")
library("SetMethods")
```

Note that you can always check which packages are installed and loaded by navigating the cursor to the rider *Packages* in the lower right window. There are also *Install* and *Update* buttons, which may be more convenient than running code to achieve the same aims.

For comprehensive information on your R environment, run *sessionInfo*. This provides info on your R version, platform and operating system, general settings, and all of the loaded and attached R packages and their version number:

```
sessionInfo()
```

Note that R and R packages, like other publications, should be cited when used in your own work. To retrieve the full reference, use *citation* with empty brackets (for the R version) or together with the package name to cite specific packages:

```
citation()
citation("QCA")
```

Data Management

Pre-Existing Data Sets

Some packages already contain data sets that may be useful for exercises. Let us take a look at the data entailed in "SetMethods" (Oana et al. 2020):

```
data(package="SetMethods")
```

This should yield a list of data sets, which we can load with the "data" function. For instance, we may load the fuzzy-set data from Vis (2009), which will appear in our environment:

```
data(VISF)
```

To examine this data set, we can either click on it (which will open a new rider) or simply type its name, which will yield the entire data set, printed in the console. This may not be convenient for large data sets. To simply inspect the structure and see whether the data was correctly read into R, we use the *head* function, which yields the first six lines from the data set, as shown below the command "head(VISF)". We see that the data set contains government cabinets as cases, and four fuzzy-set conditions ("u" is the outcome):

```
head(VISF)
```

```
              p    s   r    u
Lubbers1    0.33 0.83 1.0 0.83
Lubbers2    0.17 0.33 1.0 0.33
Lubbers3    0.33 0.67 0.6 0.67
Kok1        0.17 0.40 0.4 0.67
Kok2        0.33 0.33 0.4 0.17
Balkenende2 0.67 0.67 1.0 0.83
```

Creating New Data Sets

We can also create new data sets within R. For a simple data set, we start by creating several *vectors*. These are one-dimensional arrays with either numeric, character, or logical data in them. For a start, we only need numerical data, which shall represent raw data values for our cases. The *combine* function "c()" is used to create vectors, with values separated by commas. We create the vectors "a" and "b" and assign three values to each (with decimals for vector "b"):

```
a <- c(1, 3, 17)
b <- c(24.1, 18.0, 10.3)
```

Vectors represent columns in our QCA data frame (cases' values for a specific condition). They can be combined to create a data frame, which is the format we use for QCA data sets.

```
our.data <- data.frame(a, b)
```

To work with the data, we should first assign row names, as the rows represent our cases:

```
row.names(our.data) <- c("Shanghai", "Karachi", "Seoul")
```

We can also assign new column names (instead the generic "a" and "b"), by creating a new character vector and placing it inside our data frame. These are the conditions that we use in QCA (including the outcome condition):

```
columns <- c("rank", "population")
colnames(our.data) <- columns
```

Let us examine our new data set by typing its name and running the code:

```
our.data
         rank population
Shanghai    1       24.1
Karachi     3       18.0
Seoul      17       10.3
```

We can see that we have created three cases (cities) and two corresponding conditions (rank and population). This could be the starting point for a larger data set to be used with QCA. We would also need to calibrate this data, since it is not yet in crisp or fuzzy-set format.

Modifying Data Sets

Elements in a data frame can be modified. This may be needed when we decide to drop a case or condition, or when we want to assign a new value to a case. Modifying column names can be done with the function *colnames* and a number that specifies the column that we want to change. This number is placed in square brackets. For example, we can apply the function colnames to the "our.data" data set from above and change the name of column two by assigning a new name to it. Note that this will *overwrite* any existing name for that column:

```
colnames(our.data)[2] <- "new name"
```

It can also be useful to *add a case*, which can be done by attaching a vector to the existing data frame as an additional row, here with values of "27" and "8.7":

```
our.data[nrow(our.data) + 1,] = list(27, 8.7)
```

Now *we assign a name* to this new case, which is the fourth row of our data frame:

```
row.names(our.data)[4] <- "London"
```

Sometimes, we may want to assign *new values to specific cells* in the data frame. For example, we can attribute a value of "11.2" to the cell in row 4 and column 2, which is London's value in the condition represented by the second column (the one we relabeled to "new name"). To do this, we specify the data frame ("our.data") and use square brackets to indicate the value we want to change. Inside the square brackets, the first number refers to the rows and the second number refers to columns. These are always separated by a comma. We specify the target in this way and assign a new value of "11.2" to it:

```
our.data[4,2] <- 11.2
```

Finally, we can also delete rows and columns. To *delete a row* and place the result in a new data frame, we use a minus sign inside square brackets and the number of the row to be deleted in regular brackets. Note that in the following code, square brackets are used without a number after the comma. This means that *all* columns will be selected. Hence, this code deletes the data for row 2 and all of the columns (effectively deleting case number 2):

```
new.data <- our.data[-(2),]
```

Deleting a column from a data frame can be done by giving its number in square brackets and assigning a value of NULL to it. For example, to delete the second column and all respective rows from "new.data" we could use the following code:

```
new.data[,2] <- NULL
```

These examples help address some basic questions that arise when working with QCA data frames. That said, with R there are always *multiple* ways how to solve an issue and there may be more economical ways to achieve the same aims in your R script, especially when manifold operations are required on a large number of cases. Such options are explored in greater detail, for instance, in Field et al. (2012) and Kabacoff (2015).

Saving Data Sets

The old adage *save early, save often* also holds true for R. However, with RStudio we are prompted to save changes we made to an R script and we can also save our workspace to continue at the same place where we left off. Apart from these, we should regularly save our data to keep progress we made. This is straight-forward, as we simply need to specify the object we want to save and assign it a name of our choosing. Here, we decide to save the data frame we created earlier ("our.data") as a new comma-separated file "mydata.csv", using the *write.csv* command. When we run the following line, a new file will appear in our working directory:

```
write.csv(our.data, "mydata.csv")
```

Descriptive Statistics

Before engaging in set-theoretic analysis, it is useful to examine the data with some descriptive statistics. To illustrate this, we can use the Vis (2009) data set that is entailed in the SetMethods package (see above). To *select a column* within that data frame, we use the "$" sign. For example, to get the *mean* for the condition "p" and the *median* for the condition "s" of the VISF data set, we use the following code. Note that when you write this code in the script window, RStudio

shows the available conditions that can follow the Dollar sign. This can be helpful to remember condition names and to prevent misspellings:

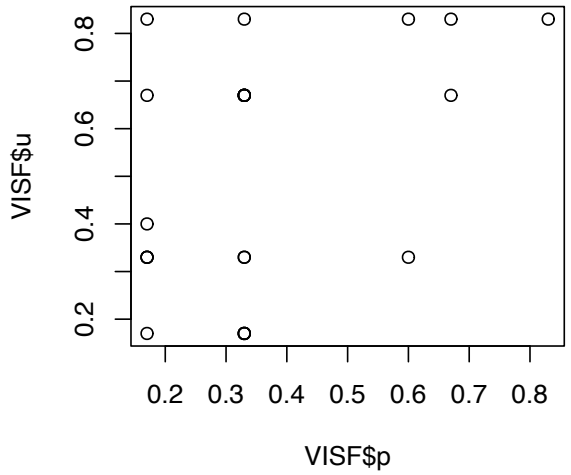
```
mean(VISF$p)
[1] 0.354

median(VISF$s)
[1] 0.4
```

We can also visualize the relationship between conditions by plotting one against another, using the basic *plot* function. For this, all we need to do is specify the two conditions in the data frame, using the syntax introduced earlier. The output will appear in the *Plots* window on the lower right (Figure 2). There we can also select *Export* and specify if we want to save the file in a certain format or resolution:

```
plot(VISF$p, VISF$u)
```

Figure 2: XY Plot



This is a basic plot that serves to check the relationship between a condition and the outcome. It can also be used to plot raw data against calibrated data. While the *plot* function can be tweaked somewhat, for more refined visualizations, and different kinds of plots, I recommend the "ggplot2" package (Wickham 2016), which offers extensive functionality and customization.[4]

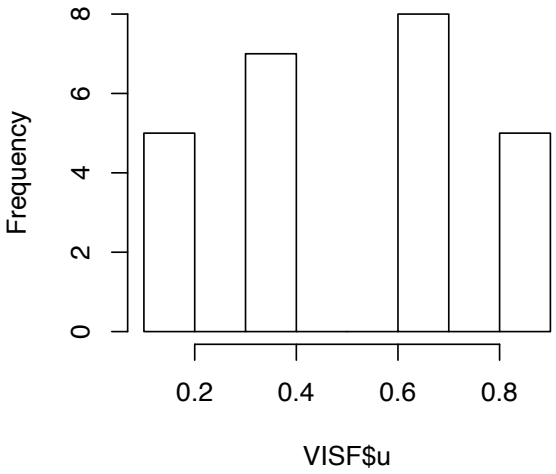
Another helpful descriptive tool are histograms. These can show the distribution of the data for each condition, to see how often certain values appear in the raw or calibrated data. We

[4] A collection of illustrations done with R is compiled at: https://www.r-graph-gallery.com/ (last accessed: December 12, 2020).

create histograms with the *hist* function by specifying the condition we want to examine. The resulting plot is shown in Figure 3:

```
hist(VISF$u)
```

Figure 3: Histogram



Set-Theoretic Analysis

The following sections describe the R functions that are needed to reproduce the analytical steps discussed in Mello (2021). The functions require that you have installed and loaded the packages "QCA" (Duşa 2019) and "SetMethods" (Oana et al. 2020), as described above.

Creating Raw Data

Before we can calibrate sets (Chapter 5), we need raw data to work with. To illustrate the procedure, we create a data frame with random data that we can calibrate. We use the function *runif* to draw from a uniform distribution 30 random numbers between 0 and 60 and place these inside the vector "Raw1". We repeat this for "Raw2" and then bind the vectors together in the data frame "DT":

```
Raw1 <- runif(30, min = 0, max = 60)
Raw2 <- runif(30, min = 0, max = 60)
DT <- data.frame(Raw1, Raw2)
```

For convenience, and also since we do not need precision beyond two decimal scores, we round up the data frame to two digits:

```
DT <- round(DT, digits = 2)
```

Now, we can examine the data frame with the *head* function, which returns the first six lines (note that you will get different values because these were randomly created):

```
head(DT)
```

```
Raw1  Raw2
1 11.00 13.56
2  2.48 51.07
3  3.50  3.88
4 48.51 56.38
5 31.80 33.35
6  4.35 36.92
```

Calibrating Sets

To calibrate a new crisp or fuzzy set, we need a data frame with raw data. This can then be transformed using the *calibrate* function of the QCA package, as shown in the next example. In brackets, we specify the raw data column of the data frame, the type of target set ("crisp" or "fuzzy"), the method of calibration ("direct" or "indirect"), and a vector with empirical anchors for full non-membership, the cross-over, and full membership in the target set). Note that these are always stated in sequence, starting with the value that reflects being "fully outside" the target set.

We use the "DT" data frame we created in the previous section to calibrate the raw data for condition 1 ("Raw1"), into a fuzzy-set condition "Fuzzy1", using the thresholds 10, 30, and 50 (the latter being full set membership):

```
DT$Fuzzy1 <- calibrate(DT$Raw1, type = "fuzzy",
                               method = "direct", c(10, 30, 50))
```

For presentational purposes, it may be helpful to use the rounding function again after the fuzzy calibration (see previous), to round to two decimals. This suffices to have a meaningful differentiation and more fine-grained scores yield no additional benefit for the analysis.

We can also calibrate a crisp set based on the same data. For this, we only need to specify the cross-over point that distinguishes exclusion from inclusion:

```
DT$Crisp1 <- calibrate(DT$Raw1, type = "crisp", method = "direct", c(30))
```

Alternatively, we can also use a "qualitative" approach, where we assign fuzzy values to specific ranges of scores in the raw data. This procedure is more complicated because we need to

proceed stepwise, starting by creating an empty vector. This serves as a "container" to be filled with values throughout the process:

```
Qual1 <- NA
```

Now we start by assigning values to cases that pass the threshold for full inclusion. We refer to the raw data column we used before and assign a value of 1 to all cases that have a raw data of 50 or higher.

```
Qual1[DT$Raw1>=50] <- 1                          # Full inclusion
```

In the next steps, we fill the vector sequentially with values for being "more in than out", the cross-over, "more out than in", and the threshold for being fully outside the set. The arrow on the right side indicates which fuzzy values are to be assigned to which range of raw data values:

```
Qual1[DT$Raw1<50 & DT$Raw1>30] <- 0.67 # "More in than out"
Qual1[DT$Raw1==30] <- 0.50              # Cross-over
Qual1[DT$Raw1<30 & DT$Raw1>10] <- 0.33  # "More out than in"
Qual1[DT$Raw1<=10] <- 0                 # Full exclusion
```

Finally, we add the new vector to the existing data frame and have a look at the data (your numbers will differ because these were drawn randomly):

```
DT$Qual1 <- Qual1
head(DT)
```

```
  Raw1  Raw2 Fuzzy1 Crisp1 Qual1
1 32.73 24.90   0.60      1  0.67
2 13.69 19.77   0.08      0  0.33
3 15.54 43.81   0.11      0  0.33
4 26.92 45.25   0.39      0  0.33
5  1.80 26.08   0.02      0  0.00
6 43.23 30.55   0.88      1  0.67
```

We can now see the results of the three different calibration approaches that we used to create three sets from the "Raw1" data. Qualitatively, the values for "Fuzzy1", "Crisp1", and "Qual1" are all either above or below the cross-over of 0.50. But beyond that, there are numerical differences because the approaches differ in the scales that are used for calibration (two values for the crisp set, five values for the qualitative fuzzy set chosen here, and continuous fuzzy values), as discussed in Chapter 5.

Necessary Conditions

There are different ways to assess necessary conditions, but the recommended function for most purposes is *QCAfit* from the SetMethods package. It tests whether a specified number of conditions, in their absence or presence, are individually necessary for the outcome. We start by reading the "FuzzyData.csv" file (online material):

```
FD <- read.csv("FuzzyData.csv", row.names = 1, sep = ";")
```

For the QCAfit function we need to specify the columns with our calibrated conditions in square brackets (here column 1 through 3, separated by a colon), the outcome ("FD$OUT"), that we want to test for necessity, and that the analysis should be for the outcome (rather than the non-outcome):

```
QCAfit(FD[,1:3], FD$OUT, necessity = TRUE, neg.out = FALSE)

    Cons.Nec Cov.Nec   RoN
C1     0.526   0.843 0.932
C2     0.947   0.863 0.851
C3     0.737   0.570 0.543
~C1    0.820   0.616 0.550
~C2    0.398   0.465 0.705
~C3    0.346   0.523 0.804
```

We can see that Condition "C2" passes the 0.90 threshold, so formally speaking it can be considered a necessary condition (also due to high coverage and relevance of necessity, see Chapter 6). In the next step, we analyze the non-outcome, setting the respective parameter to TRUE. The result shows that no condition is necessary for the non-outcome:

```
QCAfit(FD[,1:3], FD$OUT, necessity = TRUE, neg.out = TRUE)
```

Truth Tables

We create truth tables with the *truthTable* function of the QCA package (note the capitalization in the function). In the brackets, we need to specify our data frame, the outcome, whether we want to have a complete truth table with logical remainder rows, whether cases should be shown, and our consistency cut-off (typically 0.75 or higher), to determine which rows should be included in the minimization procedure afterwards. Finally, we sort the truth table by consistency ("incl") and the number of cases per row ("n"):

```
TT <- truthTable(FD, "OUT", complete = TRUE, show.cases = TRUE,
                             incl.cut = 0.75, sort.by = "incl, n")
```

This code assumes that we are working with a data frame that only includes calibrated conditions and an outcome. But sometimes we may want to specify the conditions to be included. For an alternative truth table ("TT2"), we can use the "conditions" argument and a vector that lists all the conditions we want to include:

```
TT2 <- truthTable(FD, "OUT", complete = TRUE, show.cases = TRUE,
                             conditions = c("C1", "C2"),
                             incl.cut = 0.75, sort.by = "incl, n")
```

Once we have created the truth table, we can examine it by calling upon the object ("TT" or "TT2"). We can also *extract* the truth table to save it in a separate file. For this purpose, we use an object "tt" (lowercase) that is inside the larger truth table that we created. This is because the "TT" object contains additional information for the functionality of the R package (you can explore its *structure* with the function "str(TT)").

```
TT                          # Display the truth table TT
str(TT)                     # Display the structure of the TT object
write.csv(TT$tt, "TT.csv") # Save the truth table element in a new file
```

Minimizing the Truth Table

We derive solution terms from the truth table with the *minimize* function of the QCA package. This applies the rules of Boolean minimization to the rows that consistently lead towards the outcome. These are the rows that pass the consistency threshold we defined when we created the truth table ("incl.cut" in the code above).

The minimization can be customized, but at a minimum we need to specify the truth table object ("TT") and which types of rows should be included (either "1" for all consistent rows or "?" if we also want to include logical remainder rows). We should also set "details" and "use.tilde" to "TRUE", which means that measures of fit and other information will be displayed and that the tilde sign rather than lowercase will be used for the absence of conditions. The standard syntax gives us the *conservative solution:*

```
sol.cons <- minimize(TT, include = "1", details = TRUE,
                             use.tilde = TRUE)
```

Let us have a look at this solution by calling upon the object "sol.cons":

```
sol.cons
```

```
n OUT = 1/0/C: 12/14/0
  Total      : 26

Number of multiple-covered cases: 0

M1: C1*C2*~C3 + ~C1*C2*C3 => OUT

              inclS  PRI    covS   covU   cases
-----------------------------------------------------------
1  C1*C2*~C3  0.860  0.727  0.278  0.180  R,L,Z
2  ~C1*C2*C3  0.909  0.839  0.677  0.579  T,A,F,N,C,P,V,W,J
-----------------------------------------------------------
   M1         0.891  0.813  0.857
```

This solution contains a lot of information. At the top, we see the distribution of cases. Here, 12 cases are included in the minimization that are considered consistent (rows that are on or above the consistency threshold we specified above), while 14 cases are inconsistent. The solution would further list if any case were covered by multiple solution paths. "M1" summarizes the solution in Boolean notation. Below, we can see details on the two solutions paths, starting with the conjunction of conditions, their consistency, PRI, raw coverage ("covS"), unique coverage ("covU"), and the cases that hold membership above 0.50 in the respective configuration. Finally, the bottom line provides the measures of fit for the overall solution term.

We can also derive the *parsimonious solution*. Note that we need to work with a complete truth table for this, meaning a truth table with logical remainder rows. The main difference lies in the include argument, which now also specifies logical remainders ("?").

```
sol.pars <- minimize(TT, include = "?", details = TRUE,
                     use.tilde = TRUE)
```

As expected, this solution differs from the conservative solution:

```
sol.pars
n OUT = 1/0/C: 12/14/0
  Total      : 26

Number of multiple-covered cases: 0
```

```
M1: C2 => OUT

       inclS  PRI    covS   covU   cases
-------------------------------------------------------
1  C2  0.863  0.783  0.947    -    T,A,F,N,C,P,V,W,J; R,L,Z
-------------------------------------------------------
   M1  0.863  0.783  0.947
```

We can see that instead of the two paths of the conservative solution, the parsimonious solution contains the single condition "C2", which happens to be a superset of the two conservative solution paths (the condition is present in both of these). We can also see that consistency and PRI are lower than for the previous solution, whereas coverage goes up (as will often be the case in empirical applications).

Finally, we can derive the *intermediate solution* from our truth table. As discussed in Chapter 7, there are two ways to create this type of solution: (1) we can tell the algorithm to *exclude* certain logical remainder rows from the minimization (those we deem implausible on substantive or logical grounds), or (2) we can formulate *directional expectations* that will be used by the algorithm in its selection of logical remainders.

The QCA package requires different syntax based on whether you want to exclude a single row (here *omit* must be used) or whether you want to exclude several rows (using the *exclude* argument). This may change in future versions of the R package (Duşa 2019). To make an informed choice, let us first have a look at the complete truth table:

```
TT
    C1 C2 C3   OUT    n  incl  PRI   cases
4   0  1  1     1     9  0.909 0.839 T,A,F,N,C,P,V,W,J
7   1  1  0     1     3  0.860 0.727 R,L,Z
2   0  0  1     0     7  0.533 0.054 U,I,B,G,H,M,O
1   0  0  0     0     7  0.480 0.133 D,X,Y,S,E,Q,K
3   0  1  0     ?     0    -     -
5   1  0  0     ?     0    -     -
6   1  0  1     ?     0    -     -
8   1  1  1     ?     0    -     -
```

We can see that there are four logical remainder rows (rows 3, 5, 6, and 8). This is hypothetical data, but we may say that we deem row 3 to be implausible on substantive grounds, and that

we do not want to include it as a counterfactual case (see the discussion of counterfactual analysis in Chapter 7).

To exclude a single row, like row 3, we use the *omit* argument and the number of the logical remainder in simple brackets:

```
sol.int1 <- minimize(TT, include = "?", details = TRUE, use.tilde =
                              TRUE, omit = (3))
```

This will yield the following intermediate solution (abbreviated console output):

```
          inclS  PRI    covS   covU   cases
-----------------------------------------------------
1  C1     0.843  0.629  0.526  0.180  R,L,Z
2  C2*C3  0.881  0.803  0.722  0.376  T,A,F,N,C,P,V,W,J
-----------------------------------------------------
   M1     0.857  0.770  0.902
```

As the name implies, in terms of its complexity, the intermediate solution is situated between the conservative and the parsimonious solutions. It also entails two paths, but these contain fewer conditions than the conservative solution.

We can also exclude several rows, using the *exclude* argument and a vector with the numbers of the rows that shall be excluded (here rows 5 and 8):

```
sol.int2 <- minimize(TT, include = "?", details = TRUE, use.tilde =
                              TRUE, exclude = c(5, 8))
```

Moreover, we can formulate *directional expectations* to derive an intermediate solution. For example, we may expect that C1 and the absence of C2 lead to the outcome:

```
sol.int3 <- minimize(TT, include = "?", details = TRUE, use.tilde =
                              TRUE, dir.exp = "C1, ~C2")
```

Finally, we can formulate *conjunctural directional expectations*, as for the combination of C1 and C3, in addition to C2:

```
sol.int4 <- minimize(TTdt, include = "?", details = TRUE,
                                    use.tilde = TRUE,
                                    dir.exp = "C1*C3, C2")
```

As we can see, there are many different ways to derive solution terms. Beyond the standard analysis, there are also what Schneider and Wagemann (2013) introduced as *enhanced standard*

analysis and *theory-guided enhanced standard analysis*, which differ in the treatment of logical remainders (see discussion in Chapter 7).

What is essential is that logical remainder rows are treated in a conscious manner. To do this, we should always check the *simplifying assumptions* that were used to derive a solution. This is done by calling upon the "SA" element, which is already entailed in the solution object produced by the QCA package. For example, for the first intermediate solution:

```
sol.int1$SA
$M1
  C1 C2 C3
5  1  0  0
6  1  0  1
8  1  1  1
```

This shows us that while our first intermediate solution omitted logical remainder row 3, it used logical remainder rows 5, 6, and 8 for the calculation of the solution term. If we deem some of these questionable (based on the reasoning discussed in Chapter 7), then we could adapt the settings, so that the specific rows are excluded from the minimization procedure, using some of the procedures introduced above.

References

Duşa, Adrian. 2019. *QCA with R. A Comprehensive Resource*. Cham: Springer.

Field, Andy, Jeremy Miles, and Zoë Field. 2012. *Discovering Statistics Using R*. Los Angeles: Sage.

Gaubatz, Kurt Taylor. 2015. *A Survivor's Guide to R: An Introduction for the Uninitiated and the Unnerved*. Los Angeles, CA: Sage.

Imai, Kosuke. 2017. *Quantitative Social Science: An Introduction*. Princeton, NJ: Princeton University Press.

Kabacoff, Robert I. 2015. *R In Action: Data Analysis and Graphics with R*. Shelter Island: Manning.

Mello, Patrick A. 2021. *Qualitative Comparative Analysis: An Introduction to Research Design and Application*. Washington, DC: Georgetown University Press.

Oana, Ioana-Elena Medzihorsky, Juraj, Mario Quaranta, and Carsten Q. Schneider. 2020. "SetMethods: Functions for Set-Theoretic Multi-Method Research and Advanced QCA." *R Package Version 2.6*.

Oana, Ioana-Elena, Carsten Q. Schneider, and Eva Thomann. 2021. *Qualitative Comparative Analysis Using R: A Beginner's Guide*. New York: Cambridge University Press.

Pollock, Philip H., and Barry C. Edwards. 2018. *An R Companion to Political Analysis*. Thousand Oaks: Sage.

R Core Team. 2020. "R: A Language and Environment for Statistical Computing [Computer Software]." Vienna: R Foundation for Statistical Computing. Retrieved from https://www.R-project.org

RStudio. 2020. "RStudio: Integrated Development Environment for R [Computer Software]." Boston, MA: RStudio. Retrieved from https://www.rstudio.com

Schneider, Carsten Q., and Claudius Wagemann. 2013. "Doing Justice to Logical Remainders in QCA: Moving Beyond the Standard Analysis." *Political Research Quarterly* 66 (1): 211-20.

Vis, Barbara. 2009. "Governments and Unpopular Social Policy Reform: Biting the Bullet or Steering Clear?" *European Journal of Political Research* 48: 31–57.

Wickham, Hadley. 2016. *ggplot2: Elegant Graphics for Data Analysis*. New York: Springer.