R Manual for QCA

Mello, Patrick A. (2021) *Qualitative Comparative Analysis: An Introduction to Research Design and Application*, Washington, DC: Georgetown University Press, *R* Manual for QCA (online appendix). https://doi.org/10.7910/DVN/KYF7VJ

Version 3.0 (October 2023)

The book is available at: http://press.georgetown.edu/book/qualitative-comparative-analysis

Contents

1 Introduction 2
2 Installing *R* and RStudio 3
Online Material 3
Getting to Know *R* and RStudio 4
Working Directory 6
Reading Data 7
Installing Packages 8
Citing Packages 8
3 Data Management 9
Pre-Existing Data Sets 9
Creating New Data Sets 10
Modifying Data Sets 11
Saving Data Sets 12
4 Descriptive Statistics 12
5 Set-Theoretic Analysis 16
Set Operations 16
Venn Diagrams 17
Creating Raw Data 18
Calibrating Sets 19
Necessary Conditions 21
Truth Tables 22
Minimizing the Truth Table 22
6 Visualizing Results 27
7 References 33

1 Introduction

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

Robert I. Kabacoff [1]

This *R* Manual for QCA complements *Qualitative Comparative Analysis: An Introduction to Research Design and Application* (Mello 2021). The focus is on functions to prepare and analyze data and to conduct set-theoretic analyses. For this, we use the *R* software environment (R Core Team 2020), the RStudio text editor (2020), and we work primarily with the *R* packages *QCA* (Duşa 2019), *SetMethods* (Oana & Schneider 2018), and *ggplot2* (Wickham 2016). The *R* Manual is complemented by an *R* Script that contains the code used in this document (see below for download instructions).

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, this *R* Manual aims to *get you started* towards conducting your own QCA study and to reproduce the analytical steps introduced in the book. Unlike texts that assume prior knowledge of *R*, the Manual also devotes space to basic questions of working with the *R* editor, getting the data into shape, and many of the issues that can arise during the analysis.

As for QCA functions, this manual covers one complete QCA *research cycle* as presented in the book (Mello 2021, 6), including the calibration of different kinds of raw data, the analysis of necessary conditions, the creation of truth tables, their minimization to derive solution terms, and ways to customize the analysis, concluding with the visualization of QCA results. That said, this *R* Manual does not aim to replace a comprehensive introduction to *R* and its functionality. Nor can we cover all the potential ways of working with the QCA packages. Hence, I recommend consulting at least one general introduction to *R* and data management. There are plenty of textbooks available and some are tailored towards usage in specific fields (e.g. Field et al.; Elff 2021; 2012; Gaubatz 2015; Imai 2017; Kabacoff 2015; Pollock & Edwards 2018). For questions about further analytical functions and alternative settings, readers are advised to consult the comprehensive book-length treatment of the *QCA* package by Adrian Duşa (2019), and the guide to QCA using *R* by Nena Oana, Carsten Schneider, and Eva Thomann (Oana et al. 2021), which provides further advice, particularly on functions of the *SetMethods* package (Oana & Schneider 2018).

[1] Kabacoff (2015, 532).

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* questions, also on the QCA-related packages.[3] When **errors occur**, it is often a good idea to start looking for solutions 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.

2 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 *R* and RStudio, and 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 interruptions in your workflow, as when functions are discontinued or renamed, or when default settings in a package have been changed. In most cases, however, updating should work seamlessly and backward compatibility should be ensured, so that older functions should still be operable.

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 (`RManualforQCA.R`) and data sets that are used throughout the following sections. This material is also available on Harvard Dataverse: https://doi.org/10.7910/DVN/KYF7VJ.

---

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

[3] A general forum is: https://stackoverflow.com/.

There are also QCA discussion groups on Google (https://groups.google.com/g/qcawithr) and Facebook (https://www.facebook.com/groups/483487988377003/).

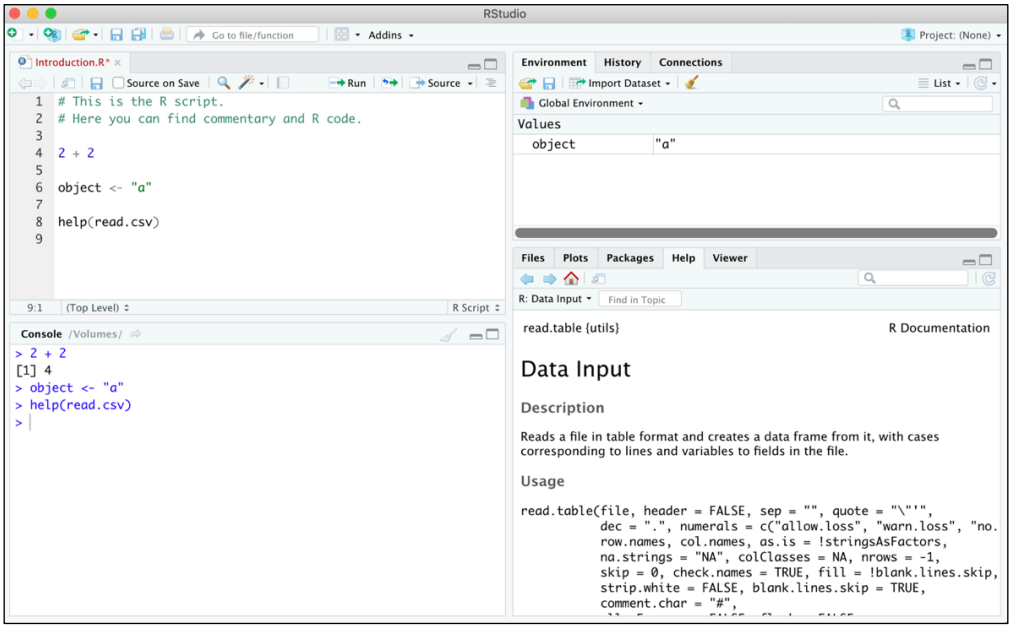
Getting to Know *R* and RStudio

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 runs on the *R* installation. For all future steps we will always work from within RStudio.

When we open RStudio, we see four windows, similar to what is shown in *Illustration 1*. The top left window shows our R SCRIPT (to see a script, open the file RManualforQCA.R). Eventually, the script area should include our entire QCA analysis. The window on the lower left shows the CONSOLE. This area shows the output that *R* produces, based on the commands we enter. Whenever we ask *R* to compute something, the results will appear in the CONSOLE window. These are printed in black, while blue font repeats the commands given in the script.

In the top right corner, we can see several tabs, including ENVIRONMENT. This area shows, among others, objects we created and 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 do – to get us started.

Illustration 1: Layout of RStudio



You can reproduce the example in *Illustration 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. Note that the first two lines are **commentary**. This is always preceded by a hash sign (#). Anything 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 many other things in RStudio, 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 wrote it and try to understand why a particular command was needed. It also makes your code transparent and reproducible to others. Many academic journals now require the submission of *R* code, together with data used for the 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. It is also considered good practice to use white space between arguments and to break up longer lines of code into multiple lines, so that it is easier to inspect the code. This can be done manually or by asking RStudio to do this for us: simply select the respective segment of code and then select the function REFORMAT CODE from the CODE rider in the menu.

The next few lines in *Illustration 1* include small examples of *R* code. For instance, 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 simultaneously press CONTROL + ENTER (Windows) or COMMAND + ENTER (macOS) on your keyboard. The latter will usually be the faster approach. Note that when the cursor is placed in a line, then the *entire line of script* will be run (including additional lines if the code segment runs over several lines). 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 with `GRAY BACKGROUND` is meant to be written in the upper left script window of RStudio (you also find the entire script in the RManualforQCA.R file). From RStudio, you can 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 <- "Amsterdam"
```

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). In any event, you can always

list everything in your environment with the *ls* function. This is followed by empty brackets, to produce the default function of *ls*, without any parameters or arguments:

```
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 once we have more objects, the results may run over several lines, each of which would be preceded by an index number, so that separate objects can be clearly identified by their index number.

Working Directory

When starting to work with *R*, we specify a **working directory** from which *R* reads data and scripts, and where it will save any output 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 in our working directory with the function *list.files*:

```
list.files()
```

To change the working directory, we use the function *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 trouble when setting your working directory, note that you can also do this manually in click-and-point fashion in RStudio. 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, 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` (part of the online material referred to above) and create a new data frame that we choose to call `fuzzy.data` (we could assign any other name). The data then appears 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 running its name (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 the information has been read as a single column because the **column separator** has not been recognized. We can solve this problem by telling *R* that our data is separated by semi-colons rather than commas, using the *sep* argument:

```
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, using the *row.names* argument. Most often, this will be the first row in our data frame. We modify our script once more and now we should have data with 26 cases and 4 conditions in our environment (C1, C2, C3, and OUT), while the column with row names is shaded in gray, which indicates that it is not treated as a variable:

```
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 computer language settings, decimals can be indicated by a comma or a period. This may cause an incorrect reading of the data. If such an error appears, we can solve this with the *dec* argument (I mention this for illustrative purposes only – our data would wrongly be read as a single column with the following code):

```
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 you encounter **errors in reading your data**, the first thing to check are these two arguments, together with *row.names* to separate case labels from data.

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 mainly work with the packages *QCA* (Duşa 2019), *SetMethods* (Oana & Schneider 2018), and *ggplot2* (Wickham 2016). We install the three together, using *install.packages* and the following syntax (we could also run separate commands for each package):

```
install.packages("QCA", "SetMethods", "ggplot2")
```

Occasionally, *install.packages* yields an **error message** stating that the package is "not writable" on the system library (because your account may not have write permissions on the computer) and asking whether you would like to use a "personal library" instead. In that case, you can type "yes" to proceed and install the package. You can also solve the issue by changing the permissions settings on your computer or by using a different file directory.

While packages only need to be installed once, they must be **loaded at the start of each session**, using the *library* function. To load all three packages, we use the following:

```
library("QCA", "SetMethods", "ggplot2")
```

Note that you can always check which packages are installed and loaded in point-and-click fashion, by navigating the cursor to the rider PACKAGES in the lower right window. There you find INSTALL and UPDATE buttons, which may be more convenient than running code to achieve the same aims.

For summary information on your *R* environment, run *sessionInfo*. This gives you info on your *R* version, platform and operating system, general settings, and attached packages and their version numbers:

```
sessionInfo()
```

Citing Packages

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 (on how to cite the *R* version), and together with the package name to cite specific packages:

```
citation()
citation("QCA")
citation("SetMethods")
citation("ggplot2")
```

For the first line, this yields the following information:

```
To cite R in publications use:
R Core Team (2023). R: A Language and Environment for Statistical
Computing. R Foundation for Statistical Computing, Vienna, Austria. URL
https://www.R-project.org/.
```

3 Data Management

Pre-Existing Data Sets

Some packages already contain data sets that may be useful for exercises. For example, let us take a look at the data entailed in *SetMethods* (Oana & Schneider 2018):

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

This yields a list of data sets, which we can load with the *data* function. To explore, we load the data from a QCA study by Barbara Vis (2009). The data will appear in our environment:

```
data(VISF)
```

To examine this data set, we can either click on the object in the environment (this will open a new rider) or simply type its name, which will yield the entire data set, printed in the console. But this may not be convenient for large data sets. To simply inspect the structure and see whether the data was read correctly 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** with *R*. For a simple data set, we start by creating several vectors. These are one-dimensional arrays with numeric, character, or logical data. For a start, we only need numerical data, which shall represent raw data values for our cases. We use the *combine* function `c()` to create vectors with three values, separated by commas. We create the vectors `a`, `b`, and `c` and assign values to each (with decimals for vectors `b` and `c`):

```
a <- c(1, 3, 17)
b <- c(24.1, 18.0, 10.3)
c <- c(6.34, 3.78, 0.61)
```

Vectors represent **columns** in our QCA data frame (cases' values for a specific condition). The vectors can be tied together with the *data.frame* function:

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

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

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

We can also assign **new column names** (instead of the generic `a`, `b`, and `c`) 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", "size")
colnames(our.data) <- columns
```

Let's examine our new data set by typing `our.data` in the console:

```
our.data
         rank population    size
Shanghai    1       24.1    6.34
Karachi     3       18.0    3.78
Seoul      17       10.3    0.61
```

We can see that we have created three cases (Shanghai, Karachi, and Seoul) and three corresponding conditions (rank, population, and size). This could be the starting point for a larger data set to be used with QCA. We would still need to calibrate this data, since it is not yet in crisp or fuzzy-set format. To be sure, most QCA studies rather import their data from Excel and the like, instead of creating it from scratch within R. But with the code above, you know how to create a data frame in case that is needed. Next, let's see how data can be modified.

Modifying Data Sets

Elements in a data frame can be modified. This may be needed, for instance, 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 we want to change. This number is placed in square brackets. For example, we can apply *colnames* to `our.data` and change the name of column two by assigning a new name to it. Note that this will overwrite any existing name for the respective column:

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

On other occasions, you may want to **add a case**, which can be done by attaching a vector to the existing data frame as an additional row, using the *nrow* argument and specifying its values after the *list* argument (here we choose the values 27, 8.7, and 1.57):

```
our.data[nrow(our.data) + 1,] = list(27, 8.7, 1.57)
```

We **assign a name** to the new case using the *row.names* function and specifying the data set (`our.data`), the row number (in square brackets), and by inserting the name of the new case:

```
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 cell in this way and assign a new value of 11.2 to it:

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

Finally, we can **delete rows and columns**. To effectively delete a row we can create a subset of our data (placing the result in a new data frame), by using 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 are selected. Hence, this creates a new data frame without row 2 and the values for all the columns across it:

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

Deleting a column from a data frame can be done in the same manner, for instance to create a subset that excludes columns 2 to 3 (or any other range of columns). For this to work, we need to place the column range in regular brackets, inside the square brackets:

```
new.data <- our.data[,-(2:3)]
```

Another way to delete a column is by stating 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 `our.data` we can 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 larger numbers of cases. Such options are explored in greater detail, for instance, in Field, Miles & Field (2012), Kabacoff (2015), and Elff (2021).

Saving Data Sets

The old adage *save early, save often* also holds true for *R*. Upon closing 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* function. When we run the following line, the new file will appear in the working directory of our computer (we can check this with the *list.files* function, introduced above):

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

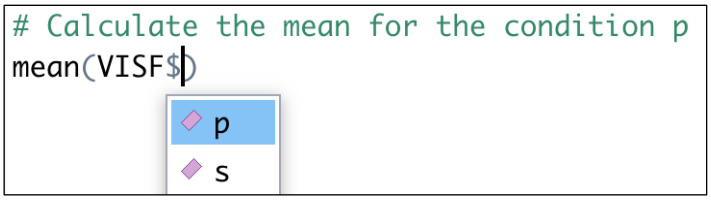
4 Descriptive Statistics

Before engaging in set-theoretic analysis, it is useful to examine data with descriptive statistics. We can use the Vis (2009) data entailed in the *SetMethods* package (see above). To **select a column** within a data frame, we use the `$` operator. For example, to get the *mean* for the condition `p` and the *median* for condition `s` of the `VISF` data set, we use the code below. Note that when you write this code in the script window, RStudio shows the available conditions that can follow the Dollar sign (see *Illustration 2*). This convenient feature is helpful to prevent

misspellings when recalling condition names. But it is also useful because it shows us the contents entailed in complex objects (we will make use of this feature below):

```
mean(VISF$p)
[1] 0.354
median(VISF$s)
[1] 0.4
```

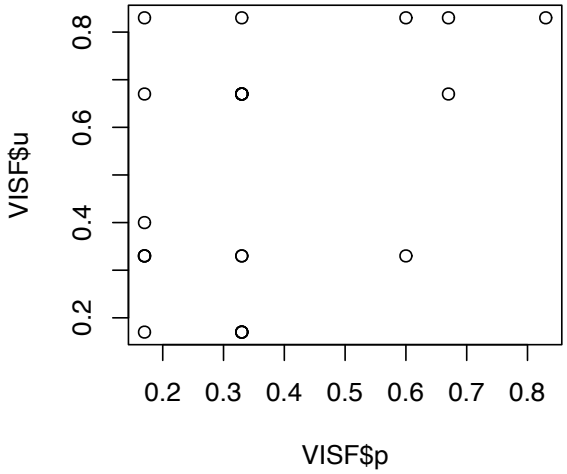
Illustration 2: Dollar Sign in RStudio



We can also **visualize** the relationship between conditions by plotting one against the other (XY plot), using the basic *plot* function of *R*. All we need to do is specify two conditions in the data frame, using the syntax introduced earlier. The output will appear in the PLOTS window on the lower right of RStudio (*Illustration 3*):

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

Illustration 3: XY Plot



To save the plot as a file on our computer, we can select EXPORT at the top of the PLOTS window and specify the file type, format, and resolution. Another way to save plots is to include such a function directly in our script. For instance, to save our plot as a PDF file, we run the following code. The first line calls upon *R*'s graphics device driver to produce PDF files, specifying the name of the file we want to create. The second line is identical to the code we used before. Finally, the command *dev.off* closes the pdf graphics device, so that the file is created:

```
pdf(file = "XYplot.pdf")
plot(VISF$p, VISF$u)
dev.off()
```

Like most functions, the PDF graphics device can be customized in myriad ways (e.g., to specify *width*, *height*, and *fonts* used). How this is done can be seen in the documentation:

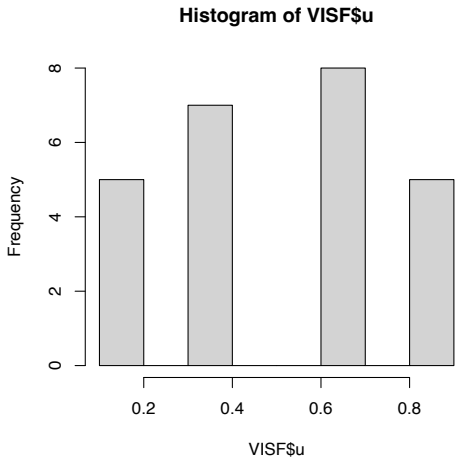
```
help("pdf")
```

A basic XY plot 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 to a certain extent, for more refined visualizations and different kinds of plots, I recommend the *ggplot2* package (Wickham 2016).[4] Examples are provided in later sections of this *R* Manual.

Histograms are another helpful descriptive tool. For any condition, a histogram shows the distribution of values in the data. Histograms are created with the *hist* function, where we need to specify the condition we seek to examine. The result is shown in *Illustration 4*:

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

Illustration 4: Histogram

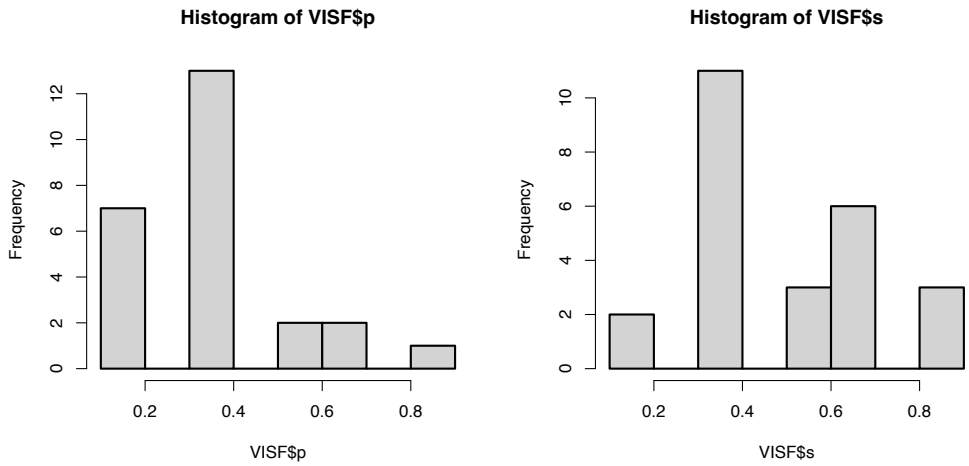


The *hist* function can be customized. Suppose we want to place several histograms on the same page, to document them in the appendix to our QCA study. This can be achieved with the *par* function to customize graphical parameters in *R*. The argument *mfrow* specifies the number of rows and columns. In this case, we want to place two histograms on one row with two columns (*Illustration 5*). The argument *lwd* specifies line width (thicker containers):

[4] The *R Graph Gallery* contains a large collection of charts done with *R*: https://www.r-graph-gallery.com/.

```
par(mfrow = c(1, 2), lwd = 2)
hist(VISF$p)
hist(VISF$s)
```

Illustration 5: Histograms, Side by Side



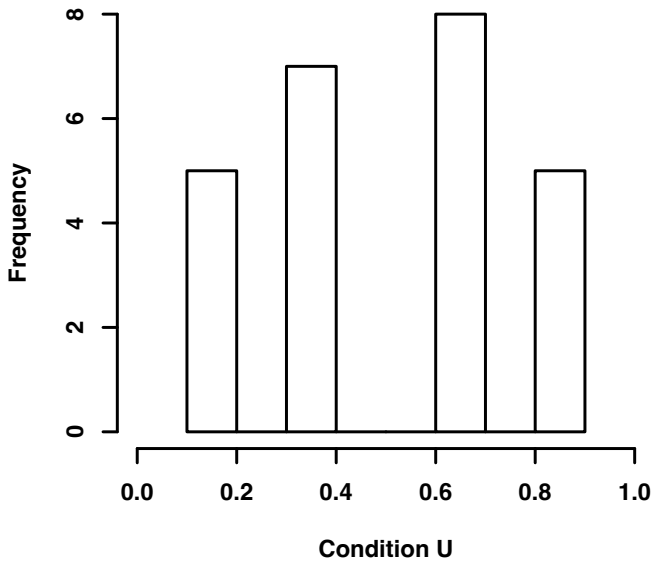
Finally, we may want to further **customize histograms**. For example, the following code produces white containers (`col = "white"`), an x-axis from 0 to 1 (`xlim = c(0, 1)`), an adapted label for the x-axis (`xlab = "Condition U"`), same font size for the axis labels and the axis numbers (`font = 2, font.lab = 2`), a solid line type (`lty = 1`), increased line thickness for the axes [`lwd = 2`], and an empty title [`main = ""`]. The result is shown in *Illustration 6*. In the same manner, we could also adapt the code to produce several histograms and plot them on the same page:

```
hist(
  VISF$u,
  col = "white",
  xlim = c(0, 1),
  xlab = "Condition U",
  font = 2,
  font.lab = 2,
  lty = 1,
  lwd = 2,
  main = ""
)
```

To reset the graphical parameters to a single column and row, we can use the following:

```
par(mfrow=c(1, 1))
```

Illustration 6: Histogram, Customized



The layout in *Illustration 6* works well for QCA purposes. Hence, we could use it as our standard layout to document the data distribution for all conditions and the outcome (ideally all histograms should be shown on a single PDF page). Note that in the above script, the x-axis is limited to scores between 0 and 1. When raw data is displayed, then the limits would need to be adjusted accordingly, similarly for the y-axis (depending on the frequencies in the data). For an example from a published QCA study, see the supplemental file to Meissner & Mello (2022): https://doi.org/10.1080/13523260.2022.2059226

5 Set-Theoretic Analysis

The following sections describe the *R* functions 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 & Schneider 2018).

Set Operations

For **Boolean operations** with crisp and fuzzy sets, we use the *pmin* and *pmax* functions of base *R*. These reflect the calculation of the minimum (Boolean AND) and the maximum (Boolean OR). To negate set membership scores, we simply use subtraction (Mello 2021, 49-53).

We first create a data frame `Boole` with the information from Table 3.3 (Mello 2021, 51):

```
A <- c(1, 1, 0)
B <- c(0, 1, 0)
C <- c(0.9, 0.7, 0.2)
D <- c(0.3, 0.8, 0.4)
```

```
Boole <- data.frame(A, B, C, D)
```

For the **Boolean AND** we use the *pmin* function across the respective conditions. We place the results into new conditions of the data frame:

```
Boole$AandB <- pmin(A, B)
Boole$CandD <- pmin(C, D)
```

Likewise, for the **Boolean OR** we use the *pmax* function across the respective conditions, also placing the results into new conditions:

```
Boole$AorB <- pmax(A, B)
Boole$CorD <- pmax(C, D)
```

Finally, for **Boolean negation** (NOT), we use subtraction, creating further conditions:

```
Boole$notA <- 1-A
Boole$notC <- 1-C
```

This results in a data frame that reflects all the calculation of Table 3.3 (Mello 2021, 51):

```
Boole
  A B   C   D AandB CandD AorB CorD notA notC
1 1 0 0.9 0.3     0   0.3    1  0.9    0  0.1
2 1 1 0.7 0.8     1   0.7    1  0.8    0  0.3
3 0 0 0.2 0.4     0   0.2    0  0.4    1  0.8
```

Venn Diagrams

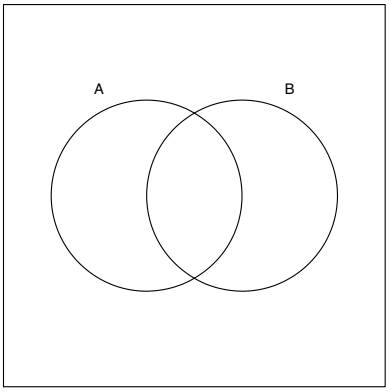
Set operations on crisp sets can be visualized with **Venn diagrams**. The *venn* package (Duşa 2022) allows the drawing of Venn diagrams with up to 7 sets. This package is installed as part of the *QCA* package (Duşa 2019), so usually we wouldn't need to install it separately. In any event, we install and load the package:

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

For a simple Venn diagram, we only need to specify the number of sets to be included. The result is show in *Illustration 7*:

```
venn(2)
```

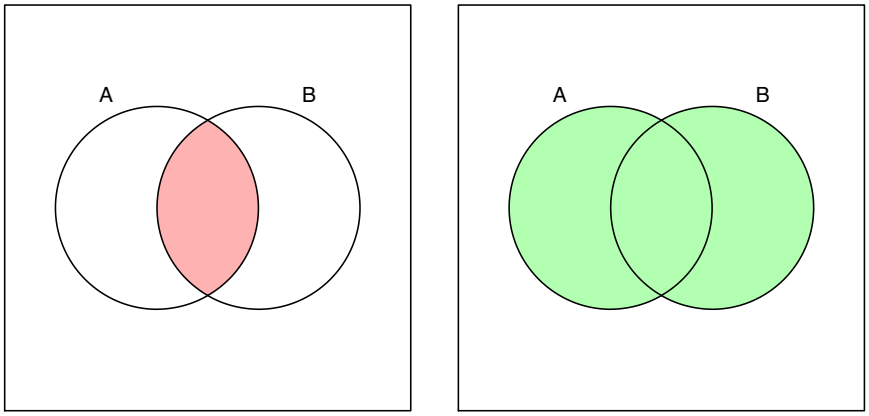
Illustration 7: Venn Diagram with Two Conditions



We can also create Venn diagrams where the intersection and the union are highlighted, as show in *Illustration 8* (we place them side by side with the *par* function introduced above):

```
par(mfrow = c(1, 2))
venn("A*B", zcol = "red")
venn("A+B", zcol = "green")
```

Illustration 8: Venn Diagrams (Intersection and Union)



Creating Raw Data

Before we can calibrate sets, we need raw data to work with. To illustrate the procedure, we create a data frame with random data for our calibration. 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** on your computer 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 be transformed using the *calibrate* function of the *QCA* package (Duşa 2019), 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 the empirical anchors for full non-membership, the cross-over, and full membership in the target set. These are termed `thresholds` in the *QCA* package. Note that the thresholds are always stated in the same sequence, starting with the value that reflects being *fully outside* the target set.

For example, we can 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 reflects *full set membership*). This is the **direct method of calibration**:

```
DT$Fuzzy1 <- calibrate(DT$Raw1, type = "fuzzy",
                       method = "direct", thresholds = "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 and approach. 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", threshold = "30")
```

Alternatively, we can also use a **qualitative approach** (of sorts), 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 calibration 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 (full exclusion). 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 data frame – creating a column with the same name – and have a look at the data. Again, 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 compare the results of the calibration approaches 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 apart from this similarity, there are numerical differences because the approaches differ in the scales that were used for the calibration (two values for the crisp set, five values for the qualitative fuzzy set chosen here, and continuous fuzzy values for the direct method of calibration), as discussed in Chapter 5 (Mello 2021).

Necessary Conditions

There are different ways to assess necessary conditions, but the recommended function for most purposes is *QCAfit* from the *SetMethods* package (Oana & Schneider 2018). 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 from the 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` (we could also test for sufficiency by setting this to `FALSE`), and that we want to analyze the outcome (rather than the non-outcome, hence the negative outcome is set to `FALSE`):

```
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 (because its coverage and relevance of necessity also pass the customary thresholds, see Chapter 6). In the next step, we analyze the non-outcome, setting the respective parameter to `TRUE`. The result shows that no condition can be considered 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 (Duşa 2019). In the brackets, we need to specify our data frame (FD), the outcome ("OUT"), whether we want to have a complete truth table with logical remainder rows (TRUE), whether cases should be shown (TRUE), and what our consistency cut-off shall be (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, incl.cut = 0.75,
                  conditions = c("C1", "C2"),
                  sort.by = "incl, n")
```

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

```
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 (Duşa 2019). This function 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 it requires that we refer to a truth table object (TT) and that we specify 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. This syntax provides us with 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 plenty of information. At the top, we see the distribution of cases. The minimization included 12 cases on rows that consistently led to the outcome (rows on or above the consistency threshold we specified above), while 14 cases were on rows that did not meet the threshold. The solution would further list if any case were covered by multiple solution paths (which is not the case here). `M1: C1*C2*~C3 + ~C1*C2*C3 => OUT` summarizes the solution in Boolean notation. Below, we can see details on the two solutions paths, starting with the configurations that comprise each path, their consistency (`inclS`), 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 indicates the consideration of logical remainders (`"?"`).

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

As expected, the parsimonious 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).

To examine which logical remainder rows were used as **simplifying assumptions** (`SA`) for the parsimonious solution, we use the following syntax:

```
sol.pars$SA
```

The output in the console tells us that logical remainder rows 3 and 8 were used as simplifying assumptions and thus as **counterfactuals** for the parsimonious solution:

```
  C1 C2 C3
3  0  1  0
8  1  1  1
```

Finally, we can derive the **intermediate solution** from our truth table. As discussed in Chapter 7 (Mello 2021), 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 the argument `omit` must be used) or whether you want to exclude several rows (using the `exclude` argument). This may change in future versions of the *QCA* 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 Mello 2021, 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 yields 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* (`dir.exp`) 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 & Wagemann (2013) introduced as *enhanced standard analysis* and *theory-guided enhanced standard analysis*, which differ in the treatment of logical remainders (see the discussion in Mello 2021, 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. As mentioned above, for the parsimonious solution and intermediate solutions based on the exlusion of logical remainders, this is done by calling upon the `SA` element, which is already entailed in the solution object produced by the *QCA* package (Duşa 2019). For example, for the first intermediate solution the simplifying assumptions are as such:

```
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.

When using **directional expectations** for the intermediate solution, the simplifying assumptions need to be checked in a different manner (due to how the *QCA* package works). Any intermediate solution that used the `dir.exp` argument creates an `i.sol` component, which in turns entails the combination of the conservative and parsimonious solutions (`C1P1`) and subcomponents with easy counterfactuals (`EC`), difficult counterfactuals (`DC`), and non-simplifying easy counterfactuals (`NSEC`). To find out which logical remainders have been used as simplifying assumptions for an intermediate solution with directional expectations, the `EC` component needs to be examined. For example, for intermediate solution #3 from above:

```
sol.int3$i.sol$C1P1$EC
```

We can see that logical remainder row 8 was treated as an easy counterfactual and used as a simplifying assumption for this intermediate solution:

```
  C1 C2 C3

8  1  1  1
```

6 Visualizing Results

When applying fuzzy-set QCA, it is recommended to construct an **XY plot** that displays set-membership in the solution term against set-membership in the outcome. As described above, this can be done with the basic *plot* function of *R*. Moreover, the *SetMethods* package (Oana & Schneider 2018) includes the function *xy.plot,* which produces plots with integrated measures of fit. While this is convenient, the resulting plots include axes lines that go beyond 0 and 1, have a rectangular shape, and are prone to overplotting labels. Hence, I recommend building a plot from the ground up and to customize it, so that it fits your research purposes. The best option is to work with the *ggplot2* package (Wickham 2016). The downside of working directly with *ggplot2* is that customization can require quite a bit of script and it may take a while until you have a plot you're satisfied with. The upshot is that once you have created your script, you can adapt it for other projects, which will take considerably less time. The

following section provides you with a script to use and adapt for your own purposes. That said, before engaging with *ggplot2*, it's a good idea to first have a look at the documentation of the package: https://ggplot2.tidyverse.org/

Then, if we haven't yet done so, we start by installing the package:

```
install.packages("ggplot2")
library("ggplot2")
```

Now we read the sample data entailed in the file `plot-data.csv` (on the website, see the download instructions above):

```
plot.data <- read.csv("plot-data.csv", row.names = 1)
```

When creating a plot with *ggplot2*, we first specify a target object (`our.plot`), followed by the *ggplot* function and information about the data source (`plot.data`) and the aesthetics (`aes`) to be plotted as x and y (in our case the columns `SOL` and `OUT` from `plot.data`). Next, we indicate that the *geom_point* function should be used (to create a scatterplot, also known as XY plot). We also specify color `fill` for our points (here `dodgerblue`, but we could use any other color), the `size` of the points, their `shape`, the `color` of the surrounding lines (`black`), and their line `stroke` (thickness). This concludes the first six lines of script:

```
our.plot <- ggplot(plot.data, aes(x = SOL, y = OUT)) +
  geom_point(fill = "dodgerblue",
             size = 7,
             shape = 21,
             color = "black",
             stroke = 1) +
```

(continued below)

The next part of the script specifies the **layout of the non-data components** of our plot. We use `theme_classic` for a minimal layout without gridlines or background. The next four lines specify the border box around the plot (`panel.border`), indicating color, thickness, and type of line. This is followed by `scale_x_continuous` and `scale_y_continuous`, which attach names to the x-axis and the y-axis, and also indicate that these should run from 0 to 1 with tick marks at each 0.1 break. `Coord_cartesian` defines the coordinate system. By setting the limits slightly outside 0 and 1, respectively, we enlarge the plot so that cases in the corners become better visible. Finally, we add a diagonal line to the plot (`geom_abline`), so that it is easier to identify set-relations:

(continued from above)

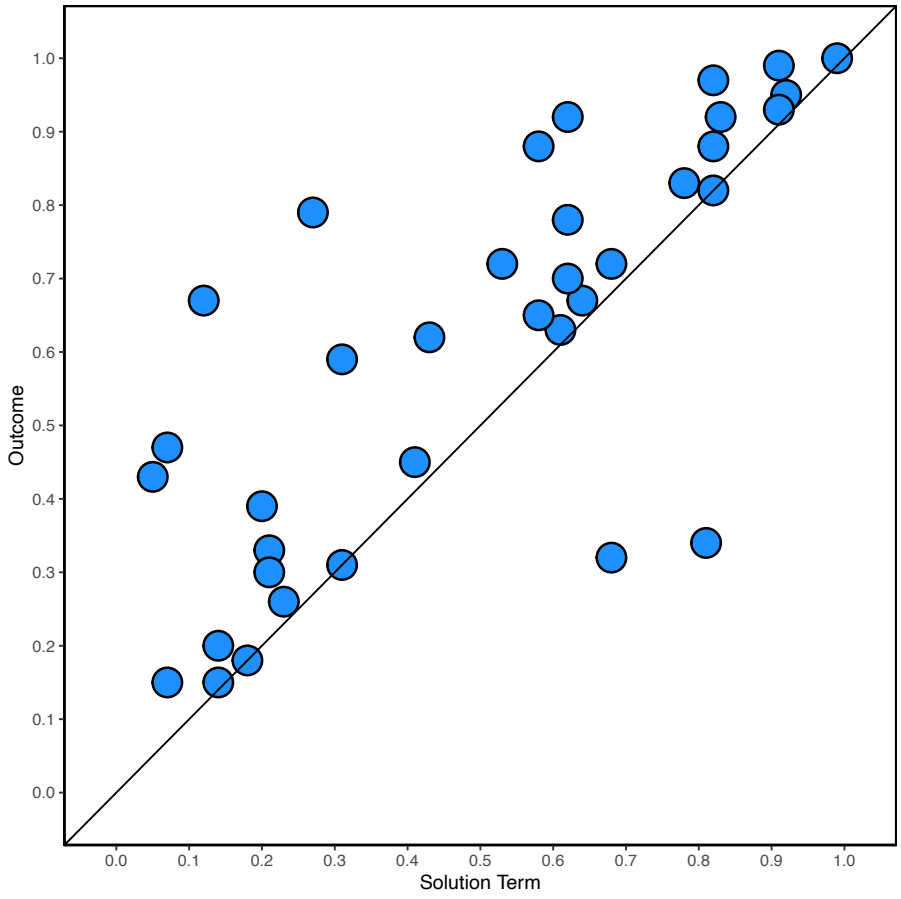
```
  theme_classic() +
  theme(panel.border = element_rect
        (fill = NA,
         color = "black",
         linewidth = 1,
         linetype = 1)) +
  scale_x_continuous(name = "Solution Term",
                     breaks = seq(0.0, 1.0, 0.1)) +
  scale_y_continuous(name = "Outcome",
                     breaks = seq(0.0, 1.0, 0.1)) +
  coord_cartesian(xlim = c(-0.02, 1.02),
                  ylim = c(-0.02, 1.02)) +
  geom_abline(intercept = c(0, 0),
              slope = 1)
our.plot
```

This script yields a basic XY plot with *ggplot2*, the result of which is shown in *Illustration 9* found below. We can further use the function *ggsave* to save the plot as a PDF file (`xyplot.pdf`) with specified measurements (`width`, `height`) and resolution (`dpi`):

```
ggsave("xyplot.pdf",
       plot = our.plot,
       path = NULL,
       scale = 1,
       width = 7,
       height = 7,
       dpi = 300)
```

The XY plot of *Illustration 9* is fine, yet it can still be improved. The text elements should be larger, and we could add horizontal and vertical lines to indicate qualitative differences. We may also want to label **unaccounted cases** and **deviant cases**. Moreover, we could include a shaded triangle in the upper right corner for observations that are **typical cases** (see Mello 2021: 194). This can be done with the following script, which is described below (and entailed in `RManualforQCA.R`).

Illustration 9: Basic XY Plot with ggplot2



The following is the script for an **enhanced XY Plot**, which could also serve as a template to be customized as needed (this is entailed in the RManualforQCA.R file). In the first steps, subsets are created for unaccounted and deviant cases, and a polygon shape is constructed for the shaded triangle that is used in the XYplot:

```
# Create subsets with unaccounted and deviant cases
# (In a similar way, you could also create a subset for typical cases)

Unacc <- subset(plot.data,
                OUT > 0.5 & SOL < 0.5) # Subset of unaccounted cases
Devia <- subset(plot.data,
                OUT < 0.5 & SOL > 0.5) # Subset of deviant cases

# Create triangular (polygon) shape for upper right triangle
poly <- data.frame(x = c(0.5, 0.5, 1.1),
                   y = c(0.5, 1.1, 1.1))
```

(continued on next page)
(continued from previous page)

```
# Plot
our.plot2 <- ggplot(plot.data, aes(x = SOL, y = OUT)) +
  geom_polygon(inherit.aes = FALSE, aes(x = x, y = y),  # Uses triangle
               data = poly,
               fill = "gray95") +
  geom_point(fill = "dodgerblue",       # Aesthetics for the cases
             size = 7,
             shape = 21,
             color = "black",
             stroke = 1) +
  theme_classic() +
  theme(panel.border = element_rect(fill = NA,
                                    color = "black",
                                    size = 1,
                                    linetype = 1)) +
  theme(axis.text.y = element_text(size = 18,
                                   color = "black"),
        axis.title = element_text(face = "plain",
                                  size = 20)) +
  theme(axis.text.x = element_text(size = 18,
                                   color = "black"),
        axis.title = element_text(face = "plain",
                                  size = 20)) +
  scale_x_continuous(name = "Solution Term",
                     breaks = seq(0.0, 1.0, 0.1)) +
  scale_y_continuous(name = "Outcome",
                     breaks = seq(0.0, 1.0, 0.1)) +
  coord_cartesian(xlim = c(-0.02, 1.02),
                  ylim = c(-0.02, 1.02)) +
  geom_hline(aes(yintercept = 0.5),             # Horizontal line
             color = "black", linetype = 5) +
  geom_vline(aes(xintercept = 0.5),             # Vertical line
             color = "black", linetype = 5) +
  geom_abline(intercept = c(0, 0),              # Diagonal line
              slope = 1) +
```

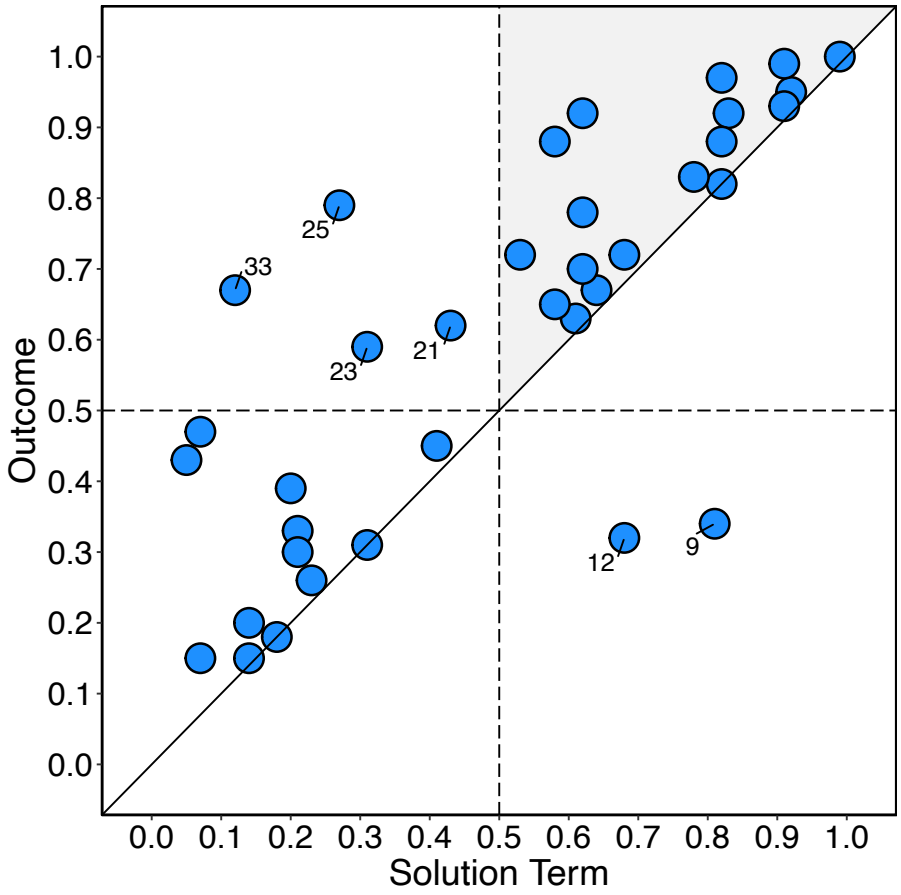
(continued on next page)
(continued from previous page)

```
  geom_text_repel(data = Unacc,              # Label unaccounted cases
            aes(label = row.names(Unacc)),
            box.padding = 0.5,
            size = 5) +
  geom_text_repel(data = Devia,             # Label deviant cases
            aes(label = row.names(Devia)),
            box.padding = 0.5,
            size = 5)
our.plot2

ggsave("xyplot2.pdf",                  # Save the plot as PDF file
       plot = our.plot2,
       path = NULL,
       scale = 1, width = 7,
       height = 7, dpi = 300)
```

This produces the XY plot shown in *Illustration 10* on the next page. In case we wanted to visualize the constituent paths of a QCA solution, we could also combine several plots on the same page layout, using the *par* function introduced above (for example, if we had three "paths" or "recipes", showing the overall solution in one XY plot and the constituent paths in three further XY plots). The argument *geom_text_repel* serves to avoid overplotting labels. But this can also be customized, for instance if you prefer to not have lines indicating the positions of the cases. This is just one example of how QCA results can be visualized and how the *ggplot2* package (Wickham 2016) can be utilized for that purpose. I also recommend having a look at visualization options explored by Rubinson (2019) and Oana, Schneider & Thomann (2021), among others.

Illustration 10: Enhanced XY Plot with ggplot2



7 References

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

Duşa, A. 2022. *Venn: Draw Venn Diagrams*. R Package Version 1.11.

Elff, M. 2021. *Data Management in R*. London: Sage.

Field, A., Miles, J., & Field, Z. 2012. *Discovering Statistics Using R*. Los Angeles: Sage.

Gaubatz, K. T. 2015. *A Survivor's Guide to R: An Introduction for the Uninitiated and the Unnerved*. Los Angeles, CA: Sage.

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

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

Meissner, K. L., & Mello, P. A. 2022. The Unintended Consequences of UN Sanctions: A Qualitative Comparative Analysis. *Contemporary Security Policy* 43 (2): 243-73.

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

Oana, I.-E., & Schneider, C. Q. 2018. SetMethods: An Add-on R Package for Advanced QCA. *The R Journal* 10 (1): 507-33.

Oana, I.-E., Schneider, C. Q., & Thomann, E. 2021. *Qualitative Comparative Analysis Using R: A Beginner's Guide*. New York: Cambridge University Press.

Pollock, P. H., & Edwards, B. C. 2018. *An R Companion to Political Analysis*. Thousand Oaks: Sage.

R Core Team (2020) *R: A Language and Environment for Statistical Computing*, Vienna, R Foundation for Statistical Computing.

RStudio (2020) *RStudio: Integrated Development Environment for R*, Boston.

Rubinson, C. 2019. Presenting Qualitative Comparative Analysis: Notation, Tabular Layout, and Visualization. *Methodological Innovations* 12 (2): 1-22.

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

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

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