R Manual for QCA

Patrick A. Mello

Version 4.2, September 2026

Abstract

The *R* Manual for QCA is a step-by-step guide to preparing and conducting a Qualitative Comparative Analysis (QCA) study in *R*. It provides a concise introduction to *R*, and covers data management, descriptive statistics, set-theoretic operations, calibration, the analysis of necessary conditions, truth table analysis, solution types, the visualization of results, cluster analysis, and robustness tests, using the *QCA* and *SetMethods* packages. For each step, the Manual introduces and discusses the respective *R* code. It is accompanied by an *R* Script that can be customized as needed and by two sample data sets, for the set-theoretic analysis and for robustness tests. The Manual accompanies the book *Qualitative Comparative Analysis: An Introduction to Research Design and Application* (Georgetown University Press, 2021) and can also be used on its own.

Cite as: Mello, Patrick A. 2026. *R Manual for QCA*, version 4.2. Harvard Dataverse. https://doi.org/10.7910/DVN/KYF7VJ

Contents

Introduction 1

1 Introduction to *R* 2
- 1.1 Online Material 2
- 1.2 Getting to Know *R* and RStudio 2
- 1.3 Working Directory 4
- 1.4 Installing Packages 5
- 1.5 Citing Packages 6

2 Data Management 6
- 2.1 Reading Data 6
- 2.2 Pre-Existing Data Sets 7
- 2.3 Creating New Data Sets 7
- 2.4 Modifying Data Sets 8
- 2.5 Saving Data Sets 9

3 Descriptive Statistics 10
- 3.1 Descriptive Measures 10
- 3.2 Histogram (Template) 12

4 Set-Theoretic Operations 14
- 4.1 Boolean Operators 14
- 4.2 Compute Function 14
- 4.3 Venn Diagrams 15

5 Calibrating Sets 16
- 5.1 Creating Raw Data 16
- 5.2 Direct Method of Calibration 16
- 5.3 Fuzzy-Value Assignment 17
- 5.4 Threshold Finder 18
- 5.5 Checking for Ambiguous Cases 19
- 5.6 Skewness Check 19

6 Necessary Conditions 19

7 Truth Table Analysis 20

8 Solution Types 22
- 8.1 Conservative Solution 22
- 8.2 Parsimonious Solution 22
- 8.3 Intermediate Solutions 23
- 8.4 Analyzing the Nonoutcome 25

9 Visualizing Results 27
- 9.1 Preparation 28
- 9.2 Basic X–Y Plot 28
- 9.3 Refined X–Y Plot (Template) 30
- 9.4 Combining X–Y Plots (Template) 32
- 9.5 Pimplot Function 35

10 Cluster Analysis 36

11 Robustness Tests 39
- 11.1 Preparation 39
- 11.2 Sensitivity Ranges 40
- 11.3 Fit-Oriented Robustness 41
- 11.4 Case-Oriented Robustness 42

12 References 44

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 preparing and analyzing data and conducting set-theoretic analyses, using the *R* software environment (R Core Team 2026) and the RStudio text editor (Posit team 2026). We work with the *R* packages *QCA* (Duşa 2019), *SetMethods* (Oana & Schneider 2018), *ggplot2* (Wickham 2016), *ggrepel* (Slowikowski 2026), *gridExtra* (Auguie 2026), and *venn* (Duşa 2026). 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*, though for new users the main hurdle is starting with *R* in the first place. This *R* Manual aims to *get you started* to help you conduct your own QCA study and reproduce the book's analytical steps. Unlike texts assuming prior knowledge of *R*, this text also covers basic questions of working with the *R* editor, shaping data, and issues that can arise along the way.

As for QCA functions, the manual covers one QCA *research cycle* as presented in the book (Mello 2021, 6), covering calibration, necessary conditions, truth tables, solution minimization, and result visualization. That said, this *R* Manual does not aim to replace a comprehensive introduction to *R*, nor every way of working with the QCA packages. I recommend consulting a general introduction to *R* and data management. Several textbooks are available, some tailored to specific fields (e.g., Field et al. 2012; Elff 2021; Gaubatz 2015; Imai 2017; Kabacoff 2015; Pollock & Edwards 2018). For further analytical functions, see the comprehensive treatment of the *QCA* package by Adrian Duşa (2019), and the inclusive guide to QCA using *R* by Ioana-Elena Oana, Carsten Schneider, and Eva Thomann (Oana et al. 2021), with a detailed discussion of the various functions of the *SetMethods* package (Oana & Schneider 2018).

Another way of getting started with *R* is through the many free online resources, such as the *swirl* package.[2] Various online forums also answer all kinds of *R* questions, including on QCA-related packages.[3] When **errors occur**, a good first step used to be pasting the error message into a search engine, since others have often hit the same issue, especially for common tasks like installing *R* and its packages, reading data, or working with data frames. As AI models have become more powerful, these can often provide more efficient help with coding issues than search engines.

---

[1] Kabacoff (2015, 532).

[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/).

1 Introduction to *R*

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 it with a graphical interface and integrated functions. We install *R* first and *only then* install RStudio (an open-source license usually suffices), since RStudio runs on top of *R*. The latest versions can be accessed at the following websites, which include platform-specific instructions for Windows, macOS, or Linux:

R: https://cran.r-project.org/
RStudio: https://posit.co/download/rstudio-desktop/

One thing to keep in mind is that there are frequent updates for *R* and RStudio, and for many *R* packages, so you should regularly **check for updates**, especially when returning to a project after weeks or months. Updates occasionally rename or discontinue functions or change default settings, though most updates do not disrupt existing code.

1.1 Online Material

Complementary online material can be found on my website https://patrickmello.com under the drop-down menu QCA. 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.

1.2 Getting to Know *R* and RStudio

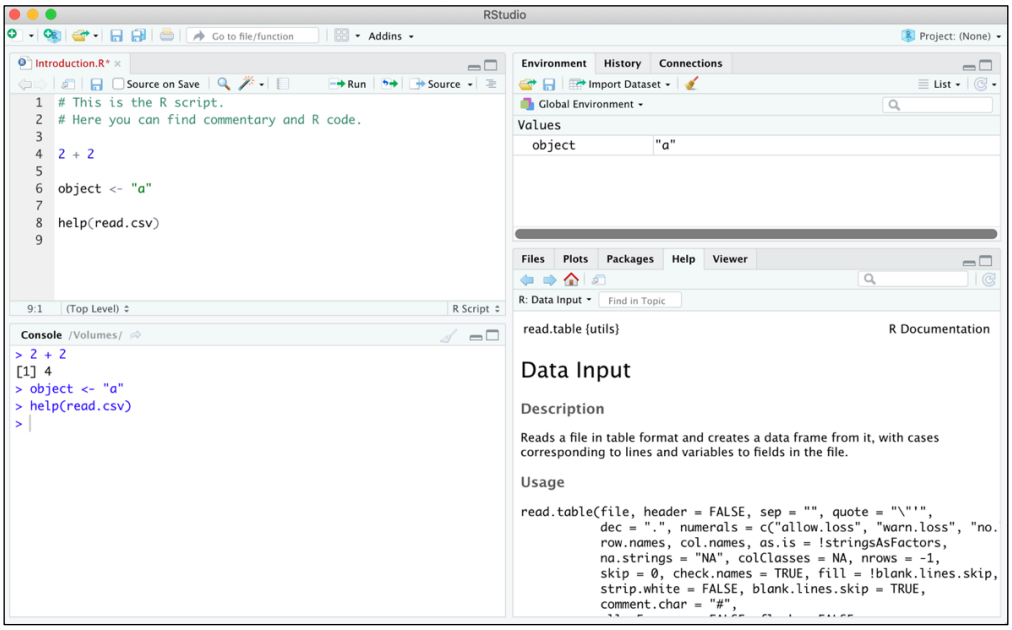
Once we have installed *R* and RStudio, we can start a new session by opening RStudio. It is not necessary to open *R*, because RStudio runs on the *R* installation. For all future steps we will always work from within RStudio.

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`). The window on the lower left shows the CONSOLE. This area shows the output that *R* produces. Whenever we ask *R* to compute something, results appear in the CONSOLE window: printed in black, while blue font repeats the script's commands.

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.

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 shown. The first two lines are **commentary**, always preceded by a hash sign (#); anything after a hash on that line is ignored by *R*. RStudio highlights commentary in a different color to set it apart from code (by default, `green`, though like most things in RStudio, this can be customized under PREFERENCES).

Illustration 1: Layout of RStudio



Commentary is part of good coding practice: short comments help us remember what a piece of *R* code does, which matters when revisiting your own code months later, and this also makes it transparent and reproducible to others. Many academic journals now require the submission of *R* code along with the data used, so writing clear, documented code from the outset saves work later. It is good practice to use white space and to break long lines up for readability. RStudio can do this for you, simply select the code and choose REFORMAT CODE (in RStudio 2024.12 and later: REFORMAT SELECTION) from the CODE menu, or press CTRL+SHIFT+A (CMD+SHIFT+A on a Mac).

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: select code and click the RUN button, or select the code and press CONTROL + ENTER (Windows) or COMMAND + ENTER (macOS), usually the faster option. With the cursor simply placed in a line (no selection), the *entire line of script* runs; select more code to run more, or select and run an entire *R* script.

For the remainder of this manual 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.

1.3 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
```

Unless specified otherwise, *R* **does not recognize subfolders**, so all files you want to work with have to be in the same folder. Hence, it is a good idea to create a dedicated *R* project folder and place all files and scripts there. Errors often come from the folder structure or from misspelled file names.

Note that you can also set up the working directory manually in RStudio. Simply navigate to the lower right window and the FILES tab. 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.

To keep our project folder tidy, this manual saves everything the script produces in a subfolder named `output`, which we create with the *dir.create* function. We name the subfolder in the file name whenever we save a file, for example `"output/XYplot.pdf"`. The data files we read stay in the working directory itself:

```
dir.create("output", showWarnings = FALSE)
```

1.4 Installing Packages

R comes with a multitude of different functions. Yet, one of the great advantages of *R* is the existence of pre-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 the set-theoretic analysis we work with the packages *QCA* (Duşa 2019) and *SetMethods* (Oana & Schneider 2018), for visualization we use *ggplot2* (Wickham 2016), *ggrepel* (Slowikowski 2026), and *gridExtra* (Auguie 2026), and for Venn diagrams we use *venn* (Duşa 2026). We install the six together, using *install.packages* and the following syntax (we could also run separate commands for each package):

```
install.packages(c("QCA", "SetMethods", "ggplot2", "ggrepel",
                   "gridExtra", "venn"))
```

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 six packages, we use the following:

```
library("QCA")
library("SetMethods")
library("ggplot2")
library("ggrepel")
library("gridExtra")
library("venn")
```

Note that you can always check which packages are installed and loaded in point-and-click fashion, by navigating the cursor to the PACKAGES tab 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()
```

1.5 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")
citation("ggrepel")
citation("gridExtra")
citation("venn")
```

For the packages *QCA* (Duşa 2019) and *SetMethods* (Oana & Schneider 2018), this yields the following reference information (adapted for cross-reference consistency):

```
Dusa, Adrian. 2019. QCA with R. A Comprehensive Resource. Springer
International Publishing.

Oana, Ioana-Elena and Carsten Q. Schneider. 2018. SetMethods: An Add-on
R Package for Advanced QCA. The R Journal 10(1): 507-33.
```

When working with this manual, please cite these two packages alongside the manual itself. The sections on cluster analysis (10) and robustness tests (11) follow the discussion in Oana, Schneider, and Thomann (2021) and the protocol of Oana and Schneider (2024), which should be cited when these functions are used.

2 Data Management

2.1 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 is comma-separated files (.csv). Here, we use the function *read.csv* to read the file `QCAdata.csv` (part of the online material referred to above) and create a new data frame that we choose to call `QCA.data` (we could assign any other name). The data then appears in our Environment in the upper right corner of RStudio.

```
QCA.data <- read.csv("QCAdata.csv")
```

However, when we examine the new `QCA.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 semicolons rather than commas, using the *sep* argument:

```
QCA.data <- read.csv("QCAdata.csv", sep = ";")
```

The result looks better, but the case-name column has been treated as a variable. We fix this with the *row.names* argument, specifying which column holds case names (usually the first). With this change we get 35 cases and 4 conditions in our environment (A, B, C, and OUT), and the row-name column is shaded gray to show it is not treated as a variable:

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

The *read.csv* function can be further customized: *row.names* specifies the column with case names, *header* whether the first row holds variable names, *sep* the separator used (comma, semicolon, space), and *dec* the decimal sign (comma or period). These are not always needed, since the defaults often read the data correctly, but they are the first things to check if you run into errors. For more, consult the documentation:

```
?read.csv
```

2.2 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 tab) 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
```

2.3 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 `v1`, `v2`, and `v3` and assign values to each (with decimals for vectors `v2` and `v3`):

```
v1 <- c(1, 3, 17)
v2 <- c(24.1, 18.0, 10.3)
v3 <- 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(v1, v2, v3)
```

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 `v1`, `v2`, and `v3`) 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 conditions (rank, population, and size), the starting point for a larger data set for QCA. We would still need to calibrate it, since it is not yet in crisp or fuzzy-set format. Most QCA studies import their data from Excel and the like rather than creating it from scratch in *R*, but the code above shows how, if needed. Next, let's see how data can be modified.

2.4 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* function to address the row after the last one, and *list* to supply its values (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 delete a row, we create a subset of our data (placing the result in a new data frame) using a minus sign inside square brackets with the row number. Leaving the space after the comma empty selects all columns, so this creates a new data frame without row 2 and its values across every column:

```
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:

```
our.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 et al. (2012), Kabacoff (2015), and Elff (2021).

2.5 Saving Data Sets

The old adage *save early, save often* also holds true for *R*. Closing RStudio prompts us to save changes to our *R* script, and we can save our workspace to pick up where we left off. We should

also save our data regularly, which just requires naming the object to save. 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. Running this line creates the new file in the output folder, which we can check with `list.files("output")`:

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

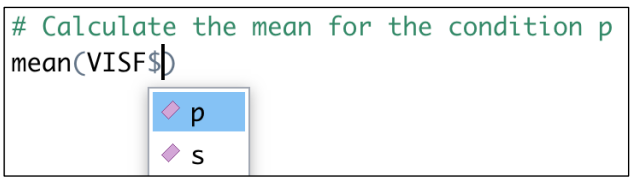
3 Descriptive Statistics

3.1 Descriptive Measures

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. RStudio shows the available conditions that can follow the dollar sign as you type (see *Illustration 2*). This helps prevent misspellings and shows the contents of complex objects (used again below):

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

Illustration 2: Dollar Sign in RStudio



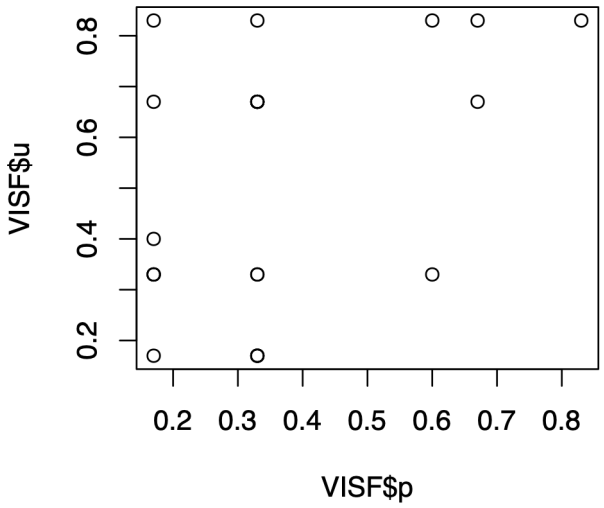
It is also useful to obtain summary statistics for an entire data set at once, using the *summary* function. This returns the minimum, maximum, quartiles, and mean for every column:

```
summary(VISF)
```

We can **visualize** the relationship between conditions by plotting one against the other (X–Y plot), using the basic *plot* function of *R*. We simply need to specify two conditions in the data frame. The output appears in the `Plots` window on the lower right of RStudio (*Illustration 3*):

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

Illustration 3: X–Y Plot



To save the plot, we can select EXPORT at the top of the PLOTS window and specify the file type, format, and resolution. But we can also do this in the 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 to produce PDF files, specifying the name of the target file. The second line is what we used before. Finally, the command *dev.off* closes the pdf graphics device, so that the file is created:

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

A PDF is a vector graphic that can be scaled to any size without loss of quality, which makes it the best format for print. Many journals and word processors, however, expect image files. The *png* device works in the same way and produces a PNG file, a lossless image format; here we specify the size in inches and a resolution (*res*) of 300 dots per inch:

```
png(filename = "output/XYplot.png", width = 7, height = 7,
    units = "in", res = 300)
plot(VISF$p, VISF$u)
dev.off()
```

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

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

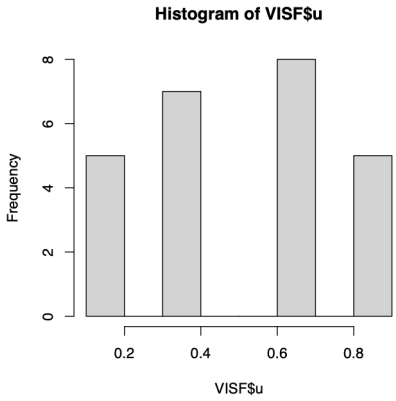
An X–Y plot serves to check the relationship between a condition and the outcome. It can also be used to plot raw against calibrated data. While the *plot* function can be tweaked, for refined visualizations, I recommend the *ggplot2* package (Wickham 2016).[4] Examples are provided in later sections. **Histograms** are another helpful descriptive tool to show the distribution of values

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

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)
```

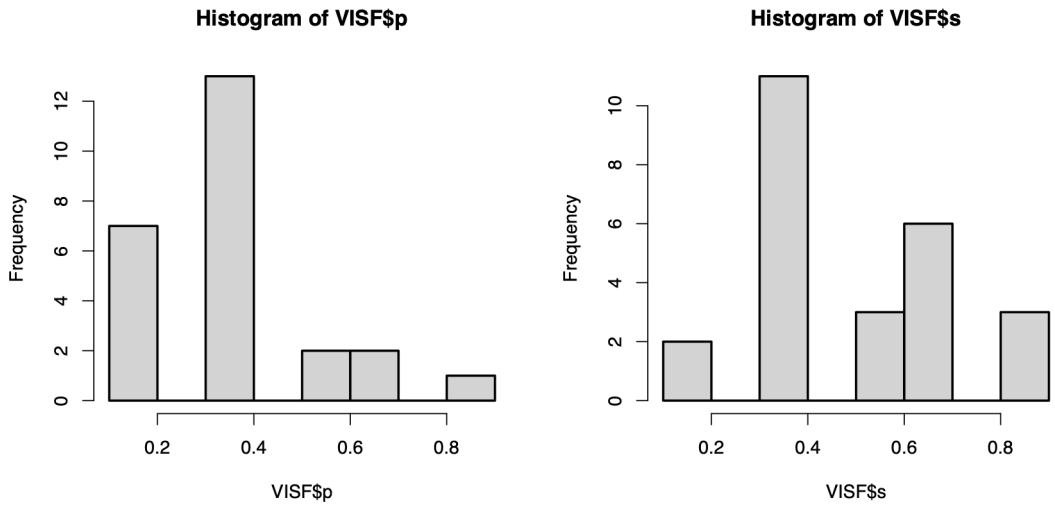
Illustration 4: Histogram



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):

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

Illustration 5: Histograms, Side by Side



3.2 Histogram (Template)

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

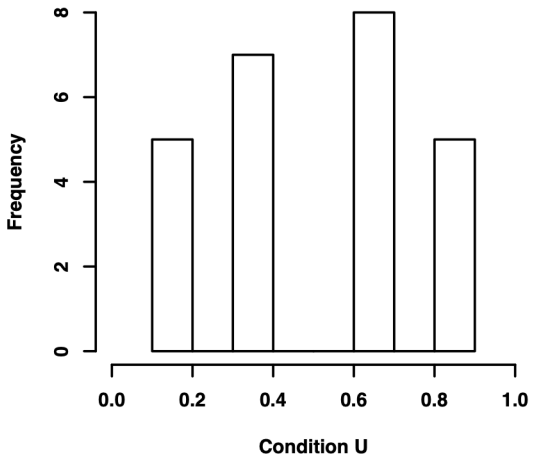
the axis numbers (`font = 2, font.lab = 2`), a solid line type (`lty = 1`), increased line thickness for the axes (`lwd = 2`), and an empty title (`main = ""`). The result is shown in *Illustration 6*. In the same manner, we could also adapt the code to produce several histograms and plot them on the 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 and Mello (2022): https://doi.org/10.1080/13523260.2022.2059226

4 Set-Theoretic Operations

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).

4.1 Boolean Operators

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` using 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 matches the calculations in 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
```

4.2 Compute Function

The *QCA* package (Duşa 2019) includes a *compute* function that calculates Boolean expressions without requiring us to manually combine *pmin*, *pmax*, and subtraction as shown above. This becomes especially useful once we work with more complex expressions, such as those returned

by the *minimize* function in later sections. For instance, using the `VISF` data set introduced earlier, we can create a new condition that reflects membership in the Boolean expression `p*s*~r`:

```
data(VISF)
VISF$newcondition <- compute("p*s*~r", data = VISF)
```

The expression syntax follows standard QCA notation: an asterisk (*) for the logical AND, a plus sign (+) for the logical OR, and a tilde (~) to negate a condition. We will use the *compute* function when visualizing results, where it lets us calculate membership in a solution term.

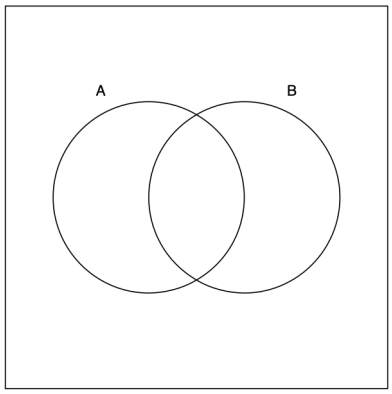
4.3 Venn Diagrams

Set operations on crisp sets can be visualized with **Venn diagrams**. The *venn* package (Duşa 2026) allows the drawing of Venn diagrams with up to 7 sets. This package is installed together with the *QCA* package (Duşa 2019) and was loaded in Section 1.4 above.

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

```
venn(2)
```

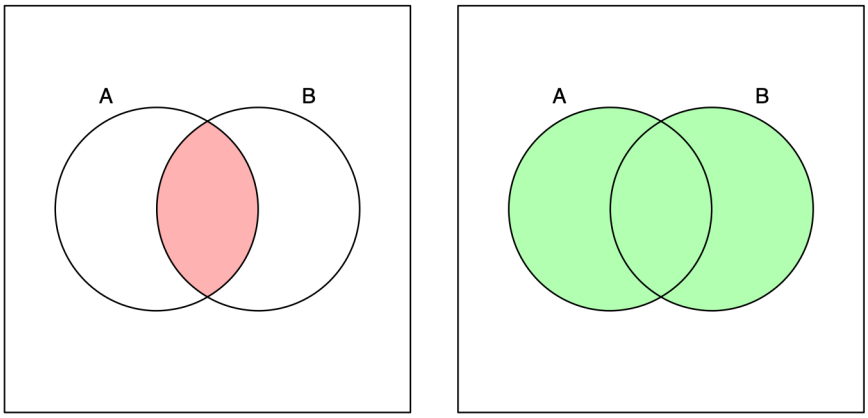
Illustration 7: Venn Diagram with Two Conditions



We can also create Venn diagrams where the intersection and the union are highlighted, as shown 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)



5 Calibrating Sets

5.1 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`. Because *runif* draws new numbers each time it is called, we first set a seed with the *set.seed* function, which makes the random draw reproducible. The number in brackets can be chosen freely, but the same number always yields the same draw:

```
set.seed(123)
Raw1 <- runif(30, min = 0, max = 60)
Raw2 <- runif(30, min = 0, max = 60)
DT <- data.frame(Raw1, Raw2)
```

Now we can examine the data frame with the *head* function, which returns the first six lines (because of the seed, you will get **the same values** on your computer):

```
head(DT)
      Raw1      Raw2
1 17.25465 57.781454
2 47.29831 54.137943
3 24.53862 41.442317
4 52.98104 47.728045
5 56.42804  1.476821
6  2.73339 28.667758
```

5.2 Direct Method of Calibration

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 crossover, 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**, once all conditions and the outcome have been calibrated, to round the entire data frame to two decimals. This suffices to have a meaningful differentiation and more fine-grained scores yield no additional benefit for the analysis. The function requires the data frame and number of digits to round to:

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

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

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

5.3 Fuzzy-Value Assignment

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
```

We continue 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 *crossover*, *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] <- 0.7  # "More in than out"
Qual1[DT$Raw1 == 30] <- 0.50               # Crossover
Qual1[DT$Raw1 < 30 & DT$Raw1 > 10] <- 0.3  # "More out than in"
Qual1[DT$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:

```
DT$Qual1 <- Qual1
head(DT)
   Raw1  Raw2 Fuzzy1 Crisp1 Qual1
1 17.25 57.78   0.13      0   0.3
2 47.30 54.14   0.93      1   0.7
3 24.54 41.44   0.31      0   0.3
4 52.98 47.73   0.97      1   1.0
5 56.43  1.48   0.98      1   1.0
6  2.73 28.67   0.02      0   0.0
```

We can now compare the calibration approaches used to create three sets from the `Raw1` data. Qualitatively, `Fuzzy1`, `Crisp1`, and `Qual1` all fall on the same side of the 0.50 crossover. Numerically they differ, though, because each uses a different scale (two values for the crisp set, five for the qualitative fuzzy set, and continuous values for the direct method), as discussed in Chapter 5 (Mello 2021).

It is also worth clarifying how these approaches transform raw scores onto fuzzy values. The qualitative approach above assigns a set of fixed values (0, 0.3, 0.5, 0.7, 1) to entire ranges of raw scores, producing a step-wise calibration: cases within the same range receive an identical fuzzy score. The direct method, by contrast, fits raw scores onto a continuous logistic (S-shaped) curve anchored at the three thresholds, so cases receive distinct fuzzy scores that vary smoothly with their distance from the thresholds. It therefore preserves more fine-grained information, whereas the qualitative approach may suit cases where only ordinal distinctions between ranges are analytically meaningful.

5.4 Threshold Finder

Identifying appropriate thresholds is the most important and also most challenging step in the calibration process. The *findTh* function of the *QCA* package automatically proposes thresholds for a given number of anchors, based on gaps in the distribution of the raw data. This can be a useful starting point, but it **should never replace substantive reasoning** about meaningful empirical anchors that reflect conceptual differences. In any event, the use of the *findTh* function should always be accompanied by **robustness tests checking the sensitivity ranges** of the calibration anchors (see Section 11 of this *R* Manual). The following are the scores we receive when asking for three thresholds for the `Raw1` data we created above:

```
findTh(DT$Raw1, n = 3)
[1] 22.11 37.04 50.14
```

With other raw data, the thresholds will differ, but the logic is the same. The *findTh* function looks for natural gaps in the distribution of the raw data and proposes anchors close to those gaps.

5.5 Checking for Ambiguous Cases

After calibration, it is good practice to check whether any cases were assigned a fuzzy value of 0.5, since these cases are at the "point of maximum ambiguity" (Ragin 2008) and that means that they cannot be assigned to any truth table row, which complicates the analysis. For this, we can use the *ambig.cases* function of the *SetMethods* package (Oana & Schneider 2018), as in the following example for the condition `Fuzzy1`:

```
ambig.cases(DT$Fuzzy1)
[1] "There are no cases with fuzzy-set scores of 0.5."
```

For our random data, none of the cases receive a calibrated score exactly at the crossover point, so no further adjustment is needed before moving on to the analysis. Had any cases turned up here, it would have been worth revisiting the calibration thresholds or examining those cases more closely before proceeding.

5.6 Skewness Check

Calibrated sets should also be checked for "skewness", meaning a strong imbalance between the number of cases more in and more out of the set (Oana et al. 2021). Substantial skew can affect consistency and coverage scores in ways that are easy to overlook. The *skew.check* function of the *SetMethods* package reports this for one or more calibrated conditions (Oana & Schneider 2018):

```
skew.check(DT[, 3:5])
[1] "Set Fuzzy1 - Cases > 0.5 / Total number of cases: 19 / 30 = 63.33
%"
[2] "Set Crisp1 - Cases > 0.5 / Total number of cases: 19 / 30 = 63.33
%"
[3] "Set Qual1 - Cases > 0.5 / Total number of cases: 19 / 30 = 63.33
%"
```

All three calibrated conditions show an identical pattern, with 19 of the 30 cases above the crossover point (because the same thresholds have been used for all three calibrations). This means there is little skewness for this particular draw of raw data. A much more lopsided pattern (for instance with 27 out of 30 cases on one side) would be a signal to reconsider the calibration or conceptualization before treating the set as analytically meaningful.

6 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 QCAdata.csv file from the online material:

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

For the *QCAfit* function we need to specify the columns with our calibrated conditions in square brackets (here columns 1 through 3), the outcome (QCA.data$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 nonoutcome, hence the negative outcome is set to FALSE:

```
QCAfit(QCA.data[, 1:3], QCA.data$OUT, necessity = TRUE, neg.out =
FALSE)
   Cons.Nec Cov.Nec   RoN
A     0.566   0.848 0.915
B     0.639   0.955 0.973
C     0.761   0.645 0.582
~A    0.606   0.552 0.576
~B    0.538   0.491 0.545
~C    0.389   0.666 0.858
```

None of the three conditions or their negations reaches the conventional 0.90 consistency threshold for necessity with respect to the outcome (see Mello 2021, Chapter 6). In the next step, we analyze the nonoutcome, setting the respective parameter to TRUE:

```
QCAfit(QCA.data[, 1:3], QCA.data$OUT, necessity = TRUE, neg.out = TRUE)
   Cons.Nec Cov.Nec   RoN
A     0.358   0.409 0.736
B     0.270   0.309 0.703
C     0.745   0.482 0.489
~A    0.867   0.604 0.606
~B    0.960   0.670 0.649
~C    0.452   0.591 0.832
```

Here, the results for ~B can be interpreted as indicating a necessary condition, with scores above 0.90 for consistency and above 0.60 for coverage and relevance, respectively.

7 Truth Table Analysis

We create truth tables with the *truthTable* function of the *QCA* package (Duşa 2019), specifying the data frame (QCA.data), the outcome ("OUT"), whether to include logical remainder rows (TRUE), whether to show cases (TRUE), and the consistency cutoff (0.75 or higher) that determines which rows enter the minimization. Additionally, we could also use a PRI cutoff (*pri.cut*) and a frequency cutoff (*n.cut*). Finally, we sort the table by consistency ("incl") and case count ("n"):

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

Once created, we can examine the truth table by calling upon the object:

```
TT
  OUT: output value
    n: number of cases in configuration
 incl: sufficiency inclusion score
  PRI: proportional reduction in inconsistency
    A  B  C    OUT    n   incl  PRI
4   0  1  1     1     9   0.995 0.991
6   1  0  1     1     4   0.994 0.989
7   1  1  0     1     6   0.944 0.916
5   1  0  0     0     4   0.594 0.165
2   0  0  1     0     12  0.497 0.239
1   0  0  0     ?     0     -     -
3   0  1  0     ?     0     -     -
8   1  1  1     ?     0     -     -
```

Case identifiers, which `show.cases = TRUE` *adds to the output, have been omitted here and in the solution outputs below for reasons of space.*

Three configurations pass the 0.75 consistency threshold and are thus coded 1 on the `OUT` column. Two configurations fall below the threshold, and three configurations have no empirical cases at all (marked `"?"`); these logical remainder rows will become relevant once we derive the parsimonious and intermediate solutions below.

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(QCA.data, "OUT", complete = TRUE,
                  show.cases = TRUE, incl.cut = 0.75,
                  conditions = c("A", "B"),
                  sort.by = "incl, n")
TT2
  OUT: output value
    n: number of cases in configuration
 incl: sufficiency inclusion score
  PRI: proportional reduction in inconsistency
    A  B    OUT    n   incl  PRI
2   0  1     1     9   0.987 0.978
4   1  1     1     6   0.956 0.931
3   1  0     1     8   0.798 0.679
1   0  0     0     12  0.456 0.197
```

Restricting the truth table to A and B effectively merges the rows that previously differed by C, which is why the case counts and consistency scores here differ from those in the full truth table

above. Three of the four possible combinations of A and B pass the 0.75 threshold, and only the combination where both conditions are absent (row 1) falls short, at 0.456.

Once created, we can examine the truth table by calling the object (`TT or TT2`), or extract the truth table proper to save separately: the object `tt` (lowercase) sits inside the larger truth table object, which also holds additional information the *R* package needs. You can always explore an object's structure with *R*'s *str* function.

```
TT                                  # Display the truth table TT
write.csv(TT$tt, "output/TT.csv") # Save the truth table in a new file
```

8 Solution Types

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 were coded 1 on the `OUT` column, that is, those rows that pass the consistency threshold we defined when we created the truth table. If we had used additional thresholds like PRI or frequency, then these would also have to be met to be coded 1 on the `OUT` column.

8.1 Conservative Solution

To derive the conservative solution, we need a truth table object (`TT`) and indicate that only rows coded `"1"` on the output column should be included in the minimization. Setting *details* to `TRUE` displays measures of fit:

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

Let us look at this solution by calling upon the object `sol.cons`:

```
sol.cons   # Examine the solution
M1: A*B*~C + A*~B*C + ~A*B*C -> OUT
           inclS   PRI   covS   covU
------------------------------------
1  A*B*~C  0.944  0.916  0.270  0.179
2  A*~B*C  0.994  0.989  0.312  0.167
3  ~A*B*C  0.995  0.991  0.397  0.239
------------------------------------
      M1   0.978  0.970  0.746
```

The conservative solution consists of three sufficient paths: `A*B*~C`, `A*~B*C`, and `~A*B*C`. Each is highly consistent (all above 0.94), and together they have an overall coverage of 0.746.

8.2 Parsimonious Solution

The parsimonious solution requires a complete truth table, meaning one that includes logical remainder rows. The main difference lies in the *include* argument, which now indicates that logical remainders should be considered in addition to those coded 1 on the `OUT` column:

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

As expected, the parsimonious solution differs from the conservative solution:

```
sol.pars
M1: B + A*C -> OUT
       inclS   PRI   covS   covU
---------------------------------
1    B  0.955  0.937  0.639  0.443
2  A*C  0.994  0.991  0.362  0.167
---------------------------------
    M1  0.964  0.953  0.805
```

Where the conservative solution contained three paths, the parsimonious solution has only two paths: `B` on its own, and `A*C`. The single condition `B` already covers both groups of cases from the conservative solution's first and third paths, which is only possible because the minimization algorithm was allowed to draw on logical remainder rows as simplifying assumptions. Consistency and PRI are a bit lower than in the conservative solution, while coverage increases slightly. This illustrates a common pattern for the relation between the conservative and parsimonious solution.

Now we check which logical remainder rows were used as simplifying assumptions (SA) for the parsimonious solution:

```
sol.pars$SA       # Logical remainder rows 3 and 8 were used as SAs
$M1
  A B C
3 0 1 0
8 1 1 1
```

Rows 3 (`~A*B*~C`) and 8 (`A*B*C`) were treated as simplifying assumptions: the algorithm assumed that these logical remainders, which have no cases in our data, would also lead to the outcome if they existed empirically.

8.3 Intermediate Solutions

Finally, we can derive the intermediate solution from the truth table. As discussed in Mello (2021, Chapter 7), there are two ways to create this type of solution: we can tell the algorithm to exclude certain logical remainder rows from the minimization – those we deem implausible on substantive or logical grounds – or we can formulate directional expectations that the algorithm then uses when selecting among logical remainders.

To make an informed choice, let us look again at the complete truth table above: rows 1, 3, and 8 are logical remainders (marked `"?"` in `TT`). Row 3 (`~A*B*~C`) has no cases in that particular configuration. We may deem it implausible on substantive grounds and choose not to include it as a counterfactual.

To exclude this single row, we use the *exclude* argument (since version 3.7 of the *QCA* package, *exclude* is documented as an argument of *truthTable*, but *minimize* still accepts it and passes it on):

```
sol.int1 <- minimize(TT, include = "?", details = TRUE,
                     exclude = c(3))
sol.int1
M1: A*B + A*C + B*C -> OUT
        inclS   PRI   covS   covU
---------------------------------
1  A*B  0.956  0.931  0.375  0.179
2  A*C  0.994  0.991  0.362  0.167
3  B*C  0.994  0.991  0.434  0.239
---------------------------------
    M1  0.978  0.971  0.781
```

As the name implies, the intermediate solution is placed between the conservative and parsimonious solutions in complexity. It entails three paths, as in the conservative solution, but each path now consists of only two conditions rather than three (`A*B`, `A*C`, and `B*C`).

Let us check which logical remainder rows were used as simplifying assumptions:

```
sol.int1$SA
$M1
  A B C
8 1 1 1
```

Only row 8 was used, since row 3 was explicitly excluded from consideration.

Alternatively, we can formulate directional expectations (*dir.exp*) to derive an intermediate solution, rather than excluding rows by number. For example, we may expect that the presence of A and C, respectively, leads to the outcome:

```
sol.int2 <- minimize(TT, include = "?", details = TRUE,
                     dir.exp = "A, C")
sol.int2
From C1P1:
M1:    A*B + A*C + B*C -> OUT
        inclS   PRI   covS   covU
---------------------------------
1  A*B  0.956  0.931  0.375  0.179
2  A*C  0.994  0.991  0.362  0.167
3  B*C  0.994  0.991  0.434  0.239
---------------------------------
    M1  0.978  0.971  0.781
```

This solution is identical to `sol.int1` above. When directional expectations are used, the simplifying assumptions need to be checked differently, through the *i.sol* component, which combines the conservative and parsimonious solutions (*C1P1*) and further distinguishes easy counterfactuals (*EC*), difficult counterfactuals (*DC*), and non-simplifying easy counterfactuals (*NSEC*). For this solution, the relevant check is:

```
sol.int2$i.sol$C1P1$EC
  A B C
8 1 1 1
```

This confirms that row 8 was treated as an easy counterfactual and used as a simplifying assumption. In this case, the approach yields the same result as the *exclude* approach we used above.

Conjunctural directional expectations can also be specified. For example, we may expect the combination `A*C`, together with `B` on its own, to lead to the outcome:

```
sol.int3 <- minimize(TT, include = "?", details = TRUE,
                     dir.exp = "A*C, B")
sol.int3
From C1P1:
M1:    B + A*C -> OUT
        inclS   PRI   covS   covU
---------------------------------
1    B  0.955  0.937  0.639  0.443
2  A*C  0.994  0.991  0.362  0.167
---------------------------------
    M1  0.964  0.953  0.805
```

This solution happens to be identical to the parsimonious solution above.

```
sol.int3$i.sol$C1P1$EC
  A B C
3 0 1 0
8 1 1 1
```

Here, both logical remainder rows 3 and 8 were treated as easy counterfactuals, which effectively matches the simplifying assumptions used for the parsimonious solution.

As we can see, there are many different ways to derive solution terms. Beyond the standard analysis illustrated here, there are also what Schneider and Wagemann (2013) introduced as "enhanced standard analysis" (ESA) and "theory-guided enhanced standard analysis" (TESA), which differ in how they treat logical remainders (see also Mello 2021, Chapter 7). What is essential, regardless of the approach chosen, is that logical remainder rows are always treated in a conscious and transparent manner, with the simplifying assumptions checked and reported alongside the solution itself.

8.4 Analyzing the Nonoutcome

Because of causal asymmetry, the solution for the outcome cannot simply be inverted to explain the nonoutcome (Mello 2021, 70–71; Oana et al. 2021, 15). If the nonoutcome is of interest, it requires a separate analysis that follows the same steps: necessary conditions (Section 6, where we found `~B` to be necessary for the nonoutcome), a truth table, and minimization. For the

truth table, the only change in the code is a tilde in front of the outcome name. We use the standard consistency threshold 0.75:

```
TTn <- truthTable(QCA.data, "~OUT", complete = TRUE,
                  show.cases = TRUE, incl.cut = 0.75,
                  sort.by = "incl, n")
TTn

    A  B  C    OUT    n   incl  PRI
5   1  0  0     1     4   0.920 0.835
2   0  0  1     1     12  0.812 0.716
4   0  1  1     0     9   0.432 0.009
6   1  0  1     0     4   0.422 0.011
7   1  1  0     0     6   0.382 0.084
1   0  0  0     ?     0     -     -
3   0  1  0     ?     0     -     -
8   1  1  1     ?     0     -     -
```

Two configurations are sufficient for the nonoutcome (rows 2 and 5). The conservative and parsimonious solutions can be derived as before:

```
sol.consn <- minimize(TTn, include = "1", details = TRUE)
sol.consn

M1: A*~B*~C + ~A*~B*C -> ~OUT

            inclS   PRI   covS   covU
-------------------------------------
1  A*~B*~C  0.920  0.835  0.257  0.152
2  ~A*~B*C  0.812  0.716  0.734  0.629
-------------------------------------
        M1  0.832  0.751  0.886

sol.parsn <- minimize(TTn, include = "?", details = TRUE)
sol.parsn

M1: ~A*~B + ~B*~C -> ~OUT

          inclS   PRI   covS   covU
----------------------------------
1  ~A*~B  0.792  0.693  0.850  0.545
2  ~B*~C  0.901  0.805  0.416  0.110
----------------------------------
      M1  0.790  0.707  0.960

sol.parsn$SA
$M1
```

```
  A B C
1 0 0 0
```

Both paths contain ~B, in line with the necessity analysis in Section 6. For an intermediate solution, directional expectations must be formulated for the nonoutcome; they are not simply the reverse of those used for the outcome. Expecting the absence of A and of C to contribute to the nonoutcome (`dir.exp = "~A, ~C"`) returns the same solution as the parsimonious one, because the single remainder used (row 1) is an easy counterfactual under these expectations:

```
sol.intn <- minimize(TTn, include = "?", details = TRUE,
                     dir.exp = "~A, ~C")
sol.intn$i.sol$C1P1$EC
  A B C
1 0 0 0
```

Finally, the two analyses must not rest on contradictory simplifying assumptions, that is, logical remainders treated as sufficient for both the outcome and the nonoutcome (Mello 2021, 143–144). The *findRows* function of the *QCA* package lists such rows (*type* = 2); the equivalent in *SetMethods* is `LR.intersect(sol.pars, sol.parsn)`:

```
findRows(obj = TT, type = 2)
numeric(0)
```

The parsimonious solution for the outcome used logical remainder rows 3 and 8 and the one for the nonoutcome used row 1, so there was no overlap between these. If there had been an overlap, then we could use the *exclude* argument (Section 8.3) to derive an intermediate solution that excludes all implausible remainders and thereby ensure that no remainder is used for both the analysis of the outcome and its negation. The *findRows* function with *type* = 3 can further be used to identify rows that pass the consistency threshold for both the outcome and the nonoutcome (simultaneous subset relations), which are not present in our data.

9 Visualizing Results

When applying fuzzy-set QCA, it is recommended to construct an *X–Y plot* that displays set-membership in the solution term against set-membership in the outcome, using the basic *plot* function of *R* or the *SetMethods* package's *xy.plot*, which adds integrated measures of fit. However, *xy.plot*'s default axes extend beyond 0 and 1 and its plots are prone to overplotting labels, so this manual instead builds plots from the ground up with *ggplot2* (Wickham 2016). Customizing *ggplot2* takes more script, but the advantage is that you get a template that you can reuse across projects with only small adjustments. The following sections provide such a script, show how to combine several plots on one page with *gridExtra*, and introduce the ready-made *pimplot* function from *SetMethods* for quick X–Y plots. Before working with *ggplot2*, it's worth browsing its documentation: https://ggplot2.tidyverse.org/.

9.1 Preparation

Before creating any plots, we need a column that captures membership in the solution term, which will form the x-axis of the X–Y plot. Membership in the outcome, the y-axis, is already the `OUT` column we have used throughout this manual. We add the new column not to `QCA.data` itself but to a copy named `plot.data`, which is used solely for the plots. This leaves the original data frame as it is, so that the analyses of the previous sections can be re-run at any time. Otherwise, a truth table built from `QCA.data` would treat the added columns as conditions. The solution term is calculated with the *compute* function introduced above, using the intermediate solution `sol.int1` derived in the previous chapter (`A*B + A*C + B*C`):

```
plot.data <- QCA.data
plot.data$SOL <- compute("A*B + A*C + B*C", data = plot.data)
```

Because *ggplot2* cannot use the row names of a data frame directly, we also copy them into a column `case_name`, which the plots below use to label the cases:

```
plot.data$case_name <- row.names(plot.data)
head(plot.data[, c("OUT", "SOL", "case_name")])
```

This confirms that both columns were added correctly, with case names ready to label individual cases in the plots that follow.

9.2 Basic X–Y Plot

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`. Next, we indicate that the *geom_point* function creates a scatterplot. We set the point fill color (here `dodgerblue`), their size, shape, and outline color (`black`), and the line stroke (thickness). *theme_classic* gives a clean layout without gridlines, and the *theme* call that follows draws a border around the plot. *scale_x_continuous* and *scale_y_continuous* label the axes with ticks every 0.1. *coord_cartesian* extends the plotting limits slightly past 0 and 1 so corner cases stay visible. Finally, *geom_abline* adds the diagonal reference line for set-relations:

```
our.plot <- ggplot(plot.data, aes(x = SOL, y = OUT)) +
  geom_point(
    fill = "dodgerblue",
    size = 7,
    shape = 21,
    color = "black",
    stroke = 1
  ) +
  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 = 0,
              slope = 1)
our.plot
```

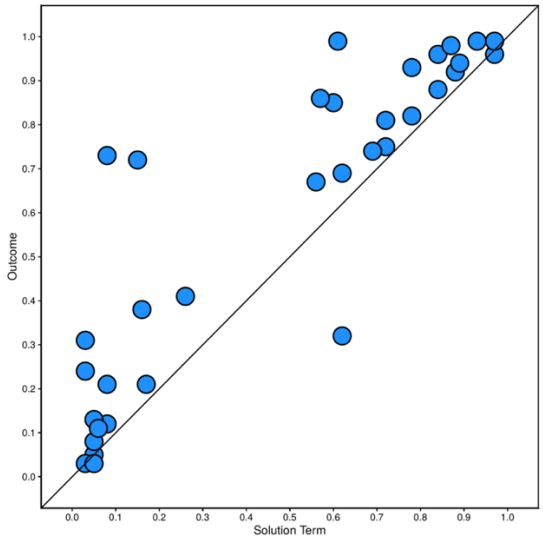
This script yields a basic X–Y plot with *ggplot2*, the result of which is shown in *Illustration 9* below. We can further use the *ggsave* function to save the plot as a PDF file (`xyplot.pdf`) in the output folder (*path*) with specified measurements (*width*, *height*):

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

To obtain a PNG file instead, we change the file extension and set the resolution with *dpi* (the file below has 2100 × 2100 pixels):

```
ggsave("xyplot.png", plot = our.plot, path = "output",
       width = 7, height = 7, dpi = 300)
```

Illustration 9: Basic X–Y Plot with *ggplot2*



9.3 Refined X–Y Plot (Template)

The X–Y plot of *Illustration 9* is fine, yet it can be improved: larger text, lines marking qualitative differences, labels for typical, unaccounted, deviant, and irrelevant cases, and shading for the upper-right triangle where cases are jointly consistent with the solution term and outcome (see Mello 2021, 194). The following script achieves this and can serve as a template (also in the `RManualforQCA.R` file). First, we create subsets for typical, unaccounted, deviant, and irrelevant cases, and construct the shaded triangle's polygon shape:

```
Typ <- subset(plot.data,
              OUT > 0.5 & SOL > 0.5)
Unacc <- subset(plot.data,
                OUT > 0.5 & SOL < 0.5)
Devia <- subset(plot.data,
                OUT < 0.5 & SOL > 0.5)
Irr <- subset(plot.data,
              OUT < 0.5 & SOL < 0.5)

poly <- data.frame(x = c(0.5, 0.5, 1.1),
                   y = c(0.5, 1.1, 1.1))
```

The plot itself follows the same logic as `our.plot` above, with a few additions: the shaded triangle (*geom_polygon*), horizontal and vertical reference lines at 0.5 (*geom_hline*, *geom_vline*), larger axis text, and four calls to *geom_text_repel*, one for each subset of cases, which label the cases while avoiding overlapping text:

```
our.plot2 <- ggplot(plot.data, aes(x = SOL, y = OUT)) +
  geom_polygon(
    inherit.aes = FALSE,
    aes(x = x, y = y),
    data = poly,
    fill = "gray95"
  ) +
  geom_point(
    fill = "dodgerblue",
    size = 7,
    shape = 21,
    color = "black",
    stroke = 1
  ) +
  theme_classic() +
  theme(
    panel.border = element_rect(
      fill = NA,
      color = "black",
      linewidth = 1,
      linetype = 1
```

```
    ),
    axis.text = 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(yintercept = 0.5,
             color = "black",
             linetype = 5) +
  geom_vline(xintercept = 0.5,
             color = "black",
             linetype = 5) +
  geom_abline(intercept = 0,
              slope = 1) +
  geom_text_repel(data = Unacc,
                  aes(label = case_name),
                  box.padding = 0.5,
                  size = 5,
                  max.overlaps = 20) +
  geom_text_repel(data = Devia,
                  aes(label = case_name),
                  box.padding = 0.5,
                  size = 5,
                  max.overlaps = 20) +
  geom_text_repel(data = Typ,
                  aes(label = case_name),
                  box.padding = 0.5,
                  size = 5,
                  max.overlaps = 20) +
  geom_text_repel(data = Irr,
                  aes(label = case_name),
                  box.padding = 0.5,
                  size = 5,
                  max.overlaps = 20)
our.plot2
```

This produces the X–Y plot shown in *Illustration 10* below. We save it the same way as before:

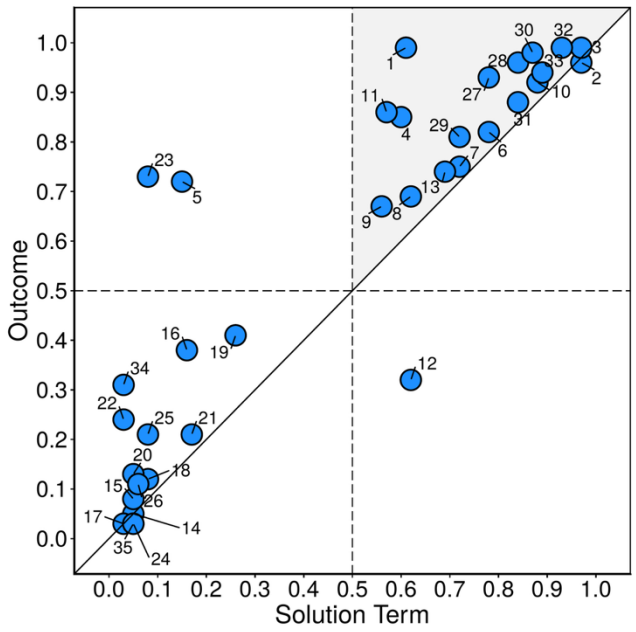
```
ggsave(
  "xyplot2.pdf",
  plot = our.plot2,
  path = "output",
```

```
  scale = 1,
  width = 7,
  height = 7
)
```

And again as a PNG file:

```
ggsave("xyplot2.png", plot = our.plot2, path = "output",
       width = 7, height = 7, dpi = 300)
```

Illustration 10: Enhanced X–Y Plot with *ggplot2*



In case we want to visualize the constituent paths of a QCA solution, we can combine several plots on the same page layout, as shown in the next two sections. The *geom_text_repel* function serves to avoid overplotting labels; this too can be customized, for instance if you prefer not to have lines indicating the positions of the cases.

9.4 Combining X–Y Plots (Template)

Each of the three sufficient paths of `sol.int1` (`A*B`, `A*C`, and `B*C`, see the chapter on solution types) can be plotted the same way as the overall solution term above, showing which cases are typical, deviant, unaccounted for, or irrelevant with respect to that specific path. Rather than repeating the same block of *ggplot2* code three more times, we wrap it in a function and call it once per path:

```
make.xyplot <- function(data, xvar, xlab) {

  data$X <- data[[xvar]]

  Typ   <- subset(data, OUT > 0.5 & X > 0.5)
  Unacc <- subset(data, OUT > 0.5 & X < 0.5)
```

```
Devia <- subset(data, OUT < 0.5 & X > 0.5)
Irr   <- subset(data, OUT < 0.5 & X < 0.5)

poly <- data.frame(x = c(0.5, 0.5, 1.1),
                   y = c(0.5, 1.1, 1.1))

ggplot(data, aes(x = X, y = OUT)) +
  geom_polygon(
    inherit.aes = FALSE,
    aes(x = x, y = y),
    data = poly,
    fill = "gray95"
  ) +
  geom_point(
    fill = "dodgerblue",
    size = 7,
    shape = 21,
    color = "black",
    stroke = 1
  ) +
  theme_classic() +
  theme(
    panel.border = element_rect(
      fill = NA,
      color = "black",
      linewidth = 1,
      linetype = 1
    ),
    axis.text = element_text(size = 18, color = "black"),
    axis.title = element_text(face = "plain", size = 20)
  ) +
  scale_x_continuous(name = xlab,
                     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(yintercept = 0.5,
             color = "black",
             linetype = 5) +
  geom_vline(xintercept = 0.5,
             color = "black",
             linetype = 5) +
  geom_abline(intercept = 0,
              slope = 1) +
```

```
    geom_text_repel(data = Unacc,
                    aes(label = case_name),
                    box.padding = 0.5,
                    size = 5,
                    max.overlaps = 20) +
    geom_text_repel(data = Devia,
                    aes(label = case_name),
                    box.padding = 0.5,
                    size = 5,
                    max.overlaps = 20) +
    geom_text_repel(data = Typ,
                    aes(label = case_name),
                    box.padding = 0.5,
                    size = 5,
                    max.overlaps = 20) +
    geom_text_repel(data = Irr,
                    aes(label = case_name),
                    box.padding = 0.5,
                    size = 5,
                    max.overlaps = 20)
}
```

We then compute membership in each individual path and build one fully customized plot per path (`our.plot2`, the plot for the overall solution term, was already created above):

```
plot.data$P1 <- compute("A*B", data = plot.data)
plot.data$P2 <- compute("A*C", data = plot.data)
plot.data$P3 <- compute("B*C", data = plot.data)

plot.P1 <- make.xyplot(plot.data, "P1", "Path A*B")
plot.P2 <- make.xyplot(plot.data, "P2", "Path A*C")
plot.P3 <- make.xyplot(plot.data, "P3", "Path B*C")
```

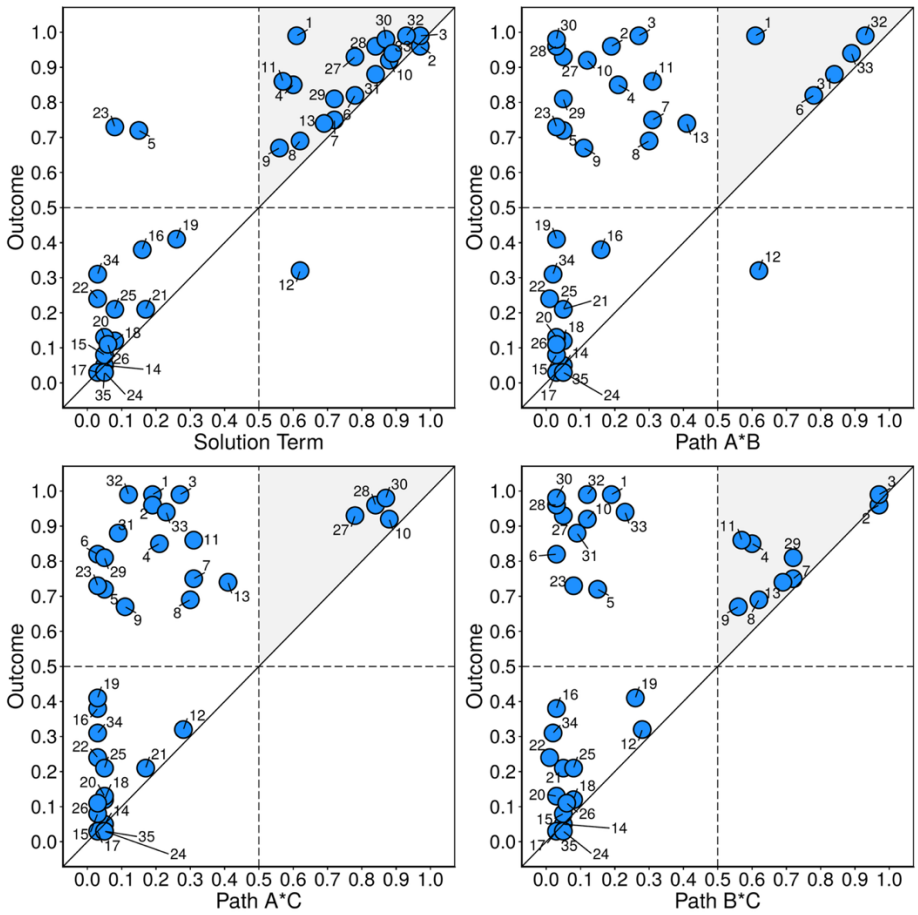
Finally, we combine the overall-solution plot and the three path plots on a single PDF page (*Illustration 11*), arranged in two rows and two columns. This requires the *gridExtra* package and its *grid.arrange* function; note that the `par(mfrow = ...)` approach used for the histograms above only affects base *R* graphics and has no effect on *ggplot2* plots such as these:

```
pdf(file = "output/xyplot_combined.pdf", width = 12, height = 12)
grid.arrange(our.plot2, plot.P1, plot.P2, plot.P3, ncol = 2, nrow = 2)
dev.off()
```

The same page can be saved as a PNG file with the *png* device introduced in Section 3.1:

```
png(filename = "output/xyplot_combined.png", width = 12, height = 12,
    units = "in", res = 300)
grid.arrange(our.plot2, plot.P1, plot.P2, plot.P3, ncol = 2, nrow = 2)
dev.off()
```

Illustration 11: Combined X–Y Plots with *gridExtra*



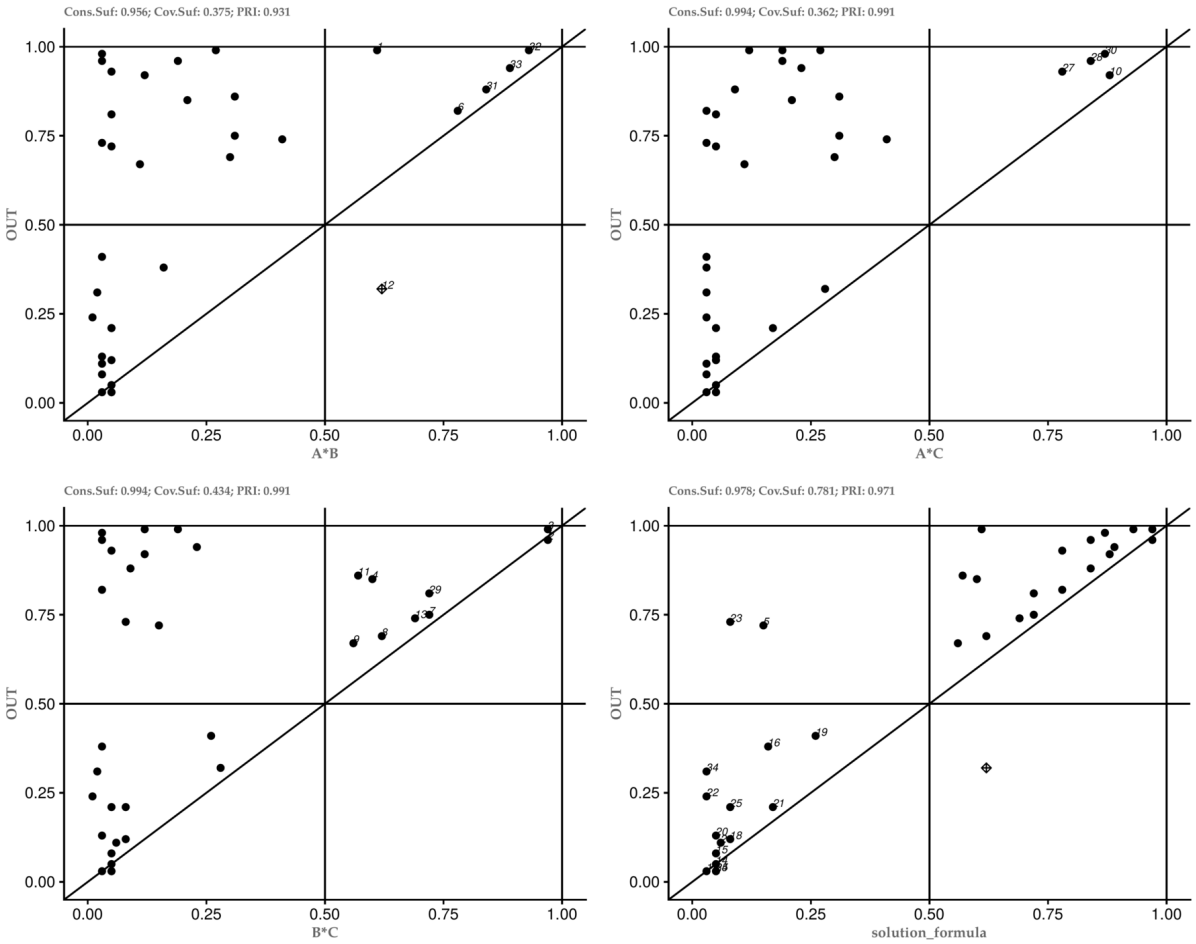
9.5 Pimplot Function

For a quicker alternative, the *pimplot* function from the *SetMethods* package automatically creates X–Y plots for each sufficient path along with the overall solution, without requiring a custom script:

```
pimplot(data = QCA.data,
        results = sol.int1,
        outcome = "OUT")
```

Unlike the approach above, *pimplot* prints each of these plots separately rather than combining them on a single page, and it offers less customization: no labeled subsets, polygon shading, or threshold lines. For compactness, *Illustration 12* below shows the four resulting plots (the overall solution term, and its three constituent paths `A*B`, `A*C`, and `B*C`) arranged on one page for this manual.

Illustration 12: X–Y Plots with *pimplot*



This is just one example of how QCA results can be visualized and how the *ggplot2* package (Wickham 2016) can be used for that purpose. I also recommend having a look at visualization options explored by Rubinson (2019) and Oana et al. (2021), among others.

10 Cluster Analysis

The *cluster* function of the *SetMethods* package extends the QCA framework to clustered data, where cases are grouped into higher-level units such as regions or time periods. This lets us assess whether solution terms derived from the pooled analysis hold consistently across clusters, or whether they mask meaningful differences between groups.

Following Oana et al. (2021), we illustrate this using the example of the study by Paykani et al. (2018) entailed in the PAYF data set from the *SetMethods* package, a clustered example in which cases (countries) are nested within regions:

```
data(PAYF)
TTpayf <- truthTable(PAYF,
                     "HL",
                     conditions = c("HE", "GG", "AH", "HI", "HW"),
                     complete = TRUE,
```

```
                    show.cases = TRUE,
                    incl.cut = 0.75,
                    sort.by = "incl, n")
TTpayf
  OUT: output value
    n: number of cases in configuration
 incl: sufficiency inclusion score
  PRI: proportional reduction in inconsistency

     HE GG AH HI HW   OUT    n   incl  PRI
30   1  1  1  0  1     1     36  0.929 0.906
32   1  1  1  1  1     1     5   0.925 0.824
24   1  0  1  1  1     1     2   0.890 0.470
20   1  0  0  1  1     1     2   0.873 0.281
14   0  1  1  0  1     1     1   0.868 0.435
26   1  1  0  0  1     1     1   0.855 0.375
17   1  0  0  0  0     0     1   0.747 0.110
 3   0  0  0  1  0     0     30  0.332 0.036
```

(This table has been shortened to the six configurations coded 1 and two of the configurations coded 0.)

The country codes for each configuration are omitted here. Running the script with `show.cases = TRUE` reproduces the full listing.

From the truth table, we derive the conservative solution:

```
SOLpayf <- minimize(TTpayf, include = "1", details = TRUE)
SOLpayf
M1: HE*HI*HW + AH*~HI*HW + ~HE*GG*AH*~HW + HE*~GG*~AH*HI +
HE*~GG*AH*~HI + HE*GG*~AH*~HI + ~GG*~AH*HI*HW -> HL
                  inclS   PRI   covS   covU
-----------------------------------------
1       HE*HI*HW  0.862  0.688  0.334  0.075
2      AH*~HI*HW  0.863  0.820  0.638  0.435
3  ~HE*GG*AH*~HW  0.803  0.225  0.149  0.004
4  HE*~GG*~AH*HI  0.804  0.216  0.200  0.005
5  HE*~GG*AH*~HI  0.702  0.160  0.141  0.001
6  HE*GG*~AH*~HI  0.788  0.274  0.143  0.004
7  ~GG*~AH*HI*HW  0.760  0.208  0.213  0.019
-----------------------------------------
              M1  0.789  0.702  0.862
```

As with the truth table above, the case listing under each path has been omitted here for reasons of space.

The *cluster* function unpacks this solution by cluster structure, here defined by country (*unit_id*) nested within region (*cluster_id*):

```
cluster.sol <- cluster(results = SOLpayf,
                       data = PAYF,
                       outcome = "HL",
                       unit_id = "COUNTRY",
                       cluster_id = "REGION",
                       wicons = FALSE)
cluster.sol
```

The full `cluster.sol` output reports consistencies, distances, and coverages for all seven paths across the pooled sample and each of the six regions included in the study. To keep this manageable, the tables below have been shortened to two of the seven paths, chosen because they show the clearest contrast between regions. The distances information is given in full, as it condenses to a single value per path.

```
Consistencies (2 of 7 paths shown):
                                          HE*HI*HW  ~GG*~AH*HI*HW
Pooled                                       0.862          0.760
Between African Region (33)                  0.279          0.152
Between Eastern Mediterranean Region (8)     0.916          0.900
Between European Region (46)                 1.000          1.000
Between Region of the Americas (26)          0.905          0.815
Between South-East Asian Region (8)          0.798          0.822
Between Western Pacific Region (10)          0.744          0.878

Distances (From Between to Pooled, all 7 paths):
HE*HI*HW        0.125
AH*~HI*HW       0.097
~HE*GG*AH*~HW   0.147
HE*~GG*~AH*HI   0.148
HE*~GG*AH*~HI   0.13
HE*GG*~AH*~HI   0.099
~GG*~AH*HI*HW   0.15

Coverages (2 of 7 paths shown):
                                          HE*HI*HW  ~GG*~AH*HI*HW
Pooled                                       0.334          0.213
Between African Region (33)                  0.522          0.391
Between Eastern Mediterranean Region (8)     0.782          0.790
Between European Region (46)                 0.175          0.072
Between Region of the Americas (26)          0.610          0.403
Between South-East Asian Region (8)          0.691          0.711
Between Western Pacific Region (10)          0.295          0.175
```

This shows that consistency for `HE*HI*HW` and `~GG*~AH*HI*HW` holds up well in the European region (1.000 in each path) but that it drops sharply in Africa (0.279 and 0.152). Coverage is

different: `HE*HI*HW` has only 0.175 coverage in Europe, but 0.782 in the Eastern Mediterranean.

11 Robustness Tests

11.1 Preparation

Robustness tests serve to "establish the extent to which the results of the set-theoretic analysis are conditional upon specific analytical choices" (Mello 2021, 208). This can involve broader research design decisions, which would call for qualitative changes. In a narrower sense, robustness tests are used to assess the sensitivity of QCA results to analytical decisions such as calibration anchors or the consistency threshold.

We can conduct such tests with the protocol designed by Oana and Schneider (2024), implemented in *SetMethods* (see also Oana et al. 2021, 144–158). The protocol has three steps. First, sensitivity ranges show how far a single parameter (a calibration anchor, the consistency threshold, the frequency cut-off) can be moved without changing the solution formula. Second, fit-oriented robustness compares the initial solution (IS) with a test set of alternative solutions that reflect plausible alternative choices. The intersection of all test solutions is the minimal test set (minTS), their union the maximal test set (maxTS), and the part of IS that withstands all tests (its overlap with the minimal test set) is the robust core (RC). Third, case-oriented robustness identifies the cases whose classification as typical or deviant depends on these choices. Note that changes of cases or conditions are not covered by this robustness test protocol. For these situations, the analysis needs to be re-run with and without the case or condition in question (Mello 2021, 174–175).

These tests require both raw and calibrated data, so instead of the `QCAdata.csv` file used previously, we use data from a published study: Meissner and Mello (2022) on the unintended consequences of UN sanctions (`SanctionsData.csv`, included in the online material of this manual). The study and its replication data are open access, and the article's online appendix documents the robustness tests in detail. The original data has been shortened here for illustration.

Article and appendix: https://doi.org/10.1080/13523260.2022.2059226
Data: https://doi.org/10.7910/DVN/KYF7VJ

```
SD <- read.csv("SanctionsData.csv", row.names = 1)
```

We separate the raw and calibrated columns into two data frames, as required by the robustness functions, then create a truth table and derive the initial solution, which replicates the intermediate solution (with *exclude*) reported in Meissner and Mello (2022):

```
SD.raw   <- SD[, 1:6]
SD.calib <- SD[, 7:12]
TT.s <- truthTable(SD.calib,
                   "UC",
                   conditions = c("AR", "LS", "P5", "EI", "LD"),
```

```
                    complete = TRUE,
                    show.cases = TRUE,
                    incl.cut = 0.75,
                    sort.by = "incl, n")
IS <- minimize(TT.s, include = "?", details = TRUE,
               exclude = c(1, 5, 9, 17))
```

This yields the following intermediate solution:

```
IS
M1: LS*P5 + ~AR*~P5*LD + AR*P5*~EI -> UC


               inclS   PRI   covS   covU
-----------------------------------------
1       LS*P5  0.935  0.931  0.506  0.251
   cases: Libya 2; Côte d'Ivoire,Somalia; DPRK; Angola; Haiti
2  ~AR*~P5*LD  0.936  0.935  0.231  0.229
   cases: DRC; Liberia; Sierra Leone
3   AR*P5*~EI  0.910  0.902  0.366  0.111
   cases: FRY; Libya 1; DPRK; Angola
-----------------------------------------
          M1  0.922  0.919  0.847
```

The initial solution (IS) consists of three sufficient paths to unintended consequences of UN sanctions (UC), each with a consistency above 0.90. The overall solution reaches a consistency of 0.922 and a coverage of 0.847.

11.2 Sensitivity Ranges

The *rob.calibrange* function identifies the range of calibration anchors for a given condition within which the solution formula does not change. Wide ranges indicate that a calibration choice is robust, while narrow ranges signal that the result is sensitive to where the anchors were placed. Here we test the sensitivity of the outcome's anchors (full exclusion at 0.5, crossover at 0.75, full inclusion at 1.25, based on Meissner & Mello 2022, Table 1):

```
rob.calibrange(raw.data = SD.raw, calib.data = SD.calib,
               test.cond.raw = "UC_raw", test.cond.calib = "UC",
               test.thresholds = c(0.5, 0.75, 1.25),
               step = 0.01, max.runs = 100,
               outcome = "UC",
               conditions = c("AR", "LS", "P5", "EI", "LD"),
               incl.cut = 0.75, n.cut = 1, include = "?",
               exclude = c(1, 5, 9, 17))

Exclusion:  Lower bound  NA Threshold  0.5 Upper bound  0.5
Crossover:  Lower bound  0.51 Threshold  0.75 Upper bound  1.24
```

```
Inclusion:  Lower bound  0.76 Threshold  1.25 Upper bound  NA
```

The lower and upper bounds mark the values within which the solution formula remains unchanged. NA means that no change was found within the number of steps tested. The crossover anchor can be moved anywhere between 0.51 and 1.24, and the inclusion anchor from 0.76 upwards, that is, across their whole admissible range up to the neighboring anchor, without altering the solution. The exclusion anchor cannot be raised at all, but it can be lowered.

The same test can be applied to the raw consistency threshold (*rob.inclrange*) and to the frequency cut-off (*rob.ncutrange*). Both require a definition of the data, the step size, and the maximum number of steps. The remaining arguments are the same as those used for the initial solution:

```
rob.inclrange(data = SD.calib, step = 0.01, max.runs = 20,
              outcome = "UC",
              conditions = c("AR", "LS", "P5", "EI", "LD"),
              incl.cut = 0.75, n.cut = 1, include = "?",
              exclude = c(1, 5, 9, 17))
Raw Consistency T.:  Lower bound  NA Threshold  0.75 Upper bound  0.75

rob.ncutrange(data = SD.calib, step = 1, max.runs = 5,
              outcome = "UC",
              conditions = c("AR", "LS", "P5", "EI", "LD"),
              incl.cut = 0.75, n.cut = 1, include = "?",
              exclude = c(1, 5, 9, 17))
N.Cut:  Lower bound  1 Threshold  1 Upper bound  1
```

This shows that the consistency threshold cannot be raised without changing the solution. The frequency cut-off cannot be raised either. All but one configuration hold a single case, so a cut-off of 2 would remove nearly every row from the minimization.

11.3 Fit-Oriented Robustness

Beyond calibration, we can also compare the initial solution against alternative test solutions derived under different analytical choices, such as a higher consistency threshold or a different calibration for the outcome. The example below constructs two such test solutions and combines them into a test set, which is then evaluated with *rob.fit* and *rob.corefit*. The original study used more alternatives, but here we simply create one test solution based on a higher consistency threshold (TS1) and one based on a different calibration of the outcome (TS2). Note that the protocol treats the outcome as fixed (Oana & Schneider 2024), and the new anchors of TS2 remain within the sensitivity ranges found above, so TS2 is not a firm test:

```
TT.t1 <- truthTable(SD.calib, "UC",
                    conditions = c("AR", "LS", "P5", "EI", "LD"),
                    complete = TRUE, show.cases = TRUE,
                    incl.cut = 0.80, sort.by = "incl, n")
TS1 <- minimize(TT.t1, include = "?", details = TRUE,
```

```
                exclude = c(1, 5, 9, 17))

SD.calib2 <- SD.calib
SD.calib2$UC <- calibrate(SD.raw$UC_raw, type = "fuzzy",
                          method = "direct", thresholds = c(0, 1, 2))
SD.calib2$UC <- round(SD.calib2$UC, 2)

TT.t2 <- truthTable(SD.calib2, "UC",
                    conditions = c("AR", "LS", "P5", "EI", "LD"),
                    complete = TRUE, show.cases = TRUE,
                    incl.cut = 0.75, sort.by = "incl, n")
TS2 <- minimize(TT.t2, include = "?", details = TRUE,
                exclude = c(1, 5, 9, 17))
TS <- list(TS1, TS2)

rob.fit(test_sol = TS, initial_sol = IS, outcome = "UC")
               RF_cov RF_cons RF_SC_minTS RF_SC_maxTS
Robustness_Fit  0.869   0.986       0.857           1

rob.corefit(test_sol = TS, initial_sol = IS, outcome = "UC")
Cons.Suf  Cov.Suf      PRI
   0.935    0.736    0.932
```

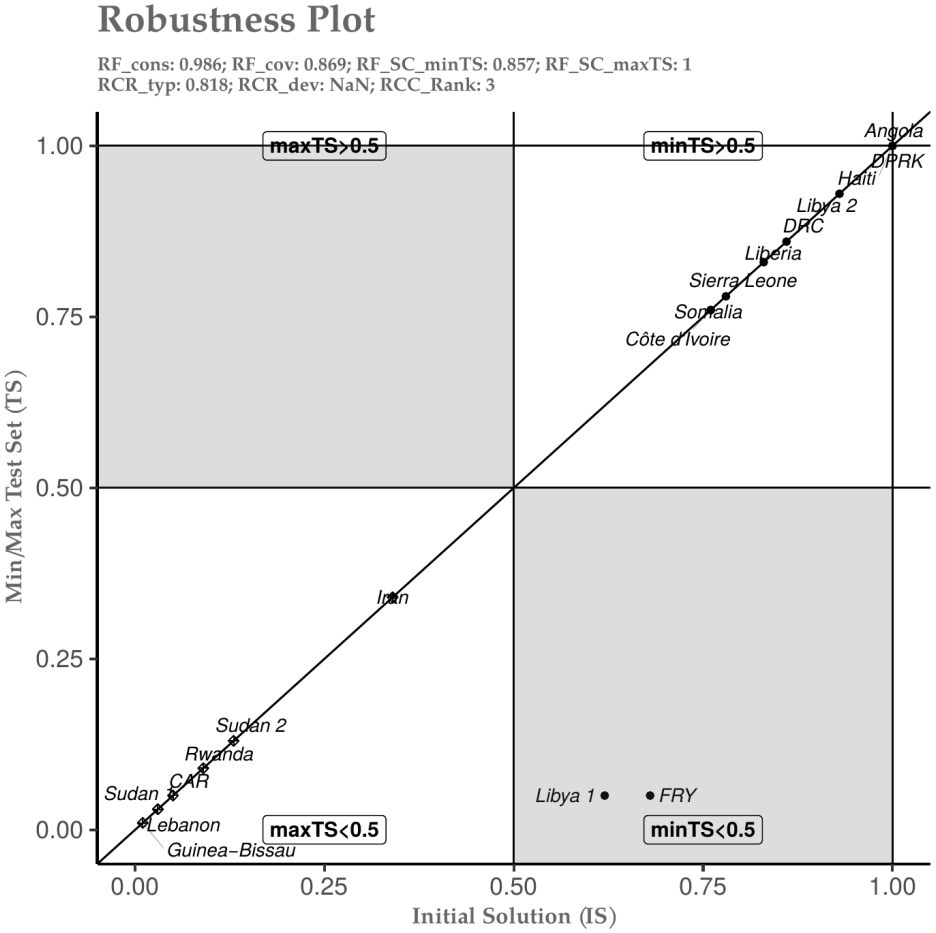
The *rob.fit* function reports four parameters between 0 and 1, where 1 means perfect robustness. `RF_cons` divides the consistency of the initial solution by that of the robust core (0.922/0.935 = 0.986). `RF_cov` divides the coverage of the robust core by that of the initial solution (0.736/0.847 = 0.869). `RF_SC_minTS` (0.857) and `RF_SC_maxTS` (1) measure the overlap between the initial solution and the minimal and the maximal test set. The *rob.corefit* function reports the parameters of fit of the robust core (consistency 0.935, coverage 0.736, PRI 0.932). Here the core consists of the first two paths of IS, `LS*P5` and `~AR*~P5*LD`, while the third path, `AR*P5*~EI`, drops out when the consistency threshold is raised to 0.80 in TS1.

11.4 Case-Oriented Robustness

The third step of the protocol looks at the qualitative positioning of cases. The *rob.xyplot* function plots each case's membership in the initial solution (x-axis) against its membership in the test sets (y-axis). Robust cases lie in the upper-right quadrant, with membership above 0.5 in both the initial solution and the minimal test set. So-called "shaky" cases (covered by IS but not by the minimal test set) fall into the lower right quadrant, and "possible" cases (covered by a test solution but not by IS) into the upper left quadrant. The options *all_labels*, *fontsize*, *jitter* and *area_lab* control the labeling. The result is shown in *Illustration 13*:

```
rob.xyplot(test_sol = TS, initial_sol = IS, outcome = "UC",
           all_labels = TRUE, fontsize = 3.5,
           jitter = TRUE, area_lab = TRUE)
```

Illustration 13: Robustness Plot with *rob.xyplot*



Nine of the eleven cases with the outcome are robust typical cases. FRY and Libya 1 are considered to be "shaky" cases. They are covered by IS through the path `AR*P5*~EI`, but not by the minimal test set, because that path disappears in TS1. No case is "possible", which is why the upper-left quadrant is empty. Note that the header of the plot also shows the measures of fit and adds case-oriented measures such as `RCR_typ` (0.818), the share of typical cases that are robust, and `RCR_dev`, which does the same for deviant consistency cases (not defined here because there are none). `RCC_Rank` summarizes the overall robustness assessment on a scale from 1 (all cases robust) to 4 (shaky and possible cases at the same time). The value 2 stands for possible cases without shaky ones, and the value 3 for shaky cases without possible ones. Finally, *rob.singletest* shows the contribution of each test solution separately, with its case rank and its set coincidence with IS:

```
rob.singletest(test_sol = TS, initial_sol = IS, outcome = "UC")
                        Model RCC_Rank    SC
1           LS*P5+~AR*~P5*LD        3 0.857
2 LS*P5+~AR*~P5*LD+AR*P5*~EI        1 1.000
```

TS1, the solution with the higher consistency threshold, is the test that reduces the robustness of the initial solution. TS2, the recalibrated outcome, leaves it intact. For further details on the calculation and interpretation of these robustness measures, see Oana and Schneider (2024).

12 References

Auguie, B. 2026. *gridExtra: Miscellaneous Functions for "Grid" Graphics*. R Package Version 2.3.1.

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

Duşa, A. 2026. *venn: Draw Venn Diagrams*. R Package Version 1.13.

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: 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*. 2nd ed. Shelter Island, NY: 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. 2024. A Robustness Test Protocol for Applied QCA: Theory and R Software Application. *Sociological Methods & Research* 53 (1): 57–88.

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

Paykani, T., Rafiey, H., & Sajjadi, H. 2018. A Fuzzy-Set Qualitative Comparative Analysis of 131 Countries: Which Configuration of the Structural Conditions Can Explain Health Better? *International Journal for Equity in Health* 17 (1): 10.

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

Posit team. 2026. *RStudio: Integrated Development Environment for R*. Boston, MA: Posit Software, PBC.

R Core Team. 2026. *R: A Language and Environment for Statistical Computing*. Vienna: R Foundation for Statistical Computing.

Ragin, C. C. 2008. *Redesigning Social Inquiry: Fuzzy Sets and Beyond*. Chicago: University of Chicago Press.

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.

Slowikowski, K. 2026. *ggrepel: Automatically Position Non-Overlapping Text Labels with 'ggplot2'*. R Package Version 0.9.8.

Vis, B. 2009. Governments and Unpopular Social Policy Reform: Biting the Bullet or Steering Clear? *European Journal of Political Research* 48 (1): 31–57.

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