Performing fuzzy- and crisp set QCA with R

A user-oriented beginner's guide

version 07.03.2018

Authors:

Dr. Eva Thomann

Ioana-Elena Oana

Stefan Wittwer

Bamberg, March 18

Performing QCA with R

Table of contents

Infoboxes

How to (not) understand and use this manual – please read!

This manual is a teaching resource that accompanies the introductory course to Qualitative Comparative Analysis (QCA) held by Dr. Eva Thomann. It is written by users for users and offers a hands-on guide to performing a crisp-set or fuzzy-set QCA analysis with the R packages QCA and SetMethods, including a Standard Analysis, Enhanced Standard Analysis, and some important diagnostics. The sole purpose of this manual is to guide users through the different analytic steps of a QCA and explain, in intuitive words, the, in our view, most important things they need to know about what they are doing. Nothing more – we keep it simple. The manual should help you to "make R your friend" and exploit its remarkably powerful toolkit to produce a sophisticated, fully replicable QCA analysis.

As such this manual neither provides a full nor systematic guide to all the relevant packages (see the respective CRAN package documentations for this), nor does it cover everything that could be done. Additional packages relevant for QCA are not covered here (see http://www.compasss.org/software.htm). There are often many ways in which things can be done with R – perhaps more elegant or parsimonious than the ones presented here. Given the continuous and rapid development of the QCA methodology and software, the advice presented here is not set into stone. Commands and functions can change and expand quickly. However, the packages we use should offer full backwards compatibility.

The manual follows the structure of a QCA analysis, and is preceded by a short and selective introduction to basic features of R which you should read before getting started. In this manual we use commands in order to "talk" with R and tell it what we want it to do.[1] In the introduction, we outline a few basic features of the "language" that R understands. However, the good news is that you neither need to learn this whole language, nor all the commands by heart, to perform QCA with R. You can simply look up the commands for what you want to do in the respective chapter of this manual, copy-paste the command, and replace the interchangeable parts of the command with the respective features of your own research. Throughout the manual, R commands are listed in grey font colour. The commands are numbered, e.g. c 1. Those parts of the commands that you need to replace to make the commands fit your own purposes are marked **bold**. For example:

[1] The QCA package does offer a graphical, user-friendly GUI interface to perform QCA, see *Box 2*. In this manual we have chosen to work with written commands instead since these commands offer users the full functionality and flexibility for performing even advanced analytic steps.

- variables are called **var**
- datasets are called **mydata**
- sets are called **MYSET**
- conditions are called **COND**
- outcomes are called **OUTCOME** .

This should be self-explanatory; copy-paste the command you need and simply insert the name of your variable, your outcome, your dataset, etc. Importantly, note that sets, conditions and outcomes are always denoted with UPPERCASE LETTERS. Make sure you also label them with uppercase letters in your dataset. This is because some commands interpret sets written in lowercase letters as the negation of the set. If sets, conditions or outcomes are written in lowercase letters in this manual, they denote the negation of the set, condition or outcome. Also, make sure that you perform the analytic steps in sections 6-8 with a dataset that only contains the calibrated sets. Some of these commands do not work if the variables in the dataset do not range from 0-1.

Some helpful resources:

- COMPASSS (http://www.compasss.org/software.htm)
- CRAN documentation for QCA package (https://cran.r-project.org/web/packages/QCA/QCA.pdf) and for SetMethods package (https://cran.r-project.org/web/packages/SetMethods/SetMethods.pdf)
- UCLA (http://www.ats.ucla.edu/stat/r)
- Quick R (http://www.statmethods.net /)
- R Graph Gallery (http://rgraphgallery.blogspot.ch/)
- Forums: www.stackoverflow.com (http://www.stackoverflow.com) ; www.stats.stackexchange.com (http://www.stats.stackexchange.com); http://www.talkstats.com (http://www.talkstats.com)
- Try out R with the TryR Code School (http://tryr.codeschool.com/)

This manual is intended as an "open source" working document that is continuously improved based on new developments and the feedback of users. As such, it is and remains work in progress. We do our very best to ensure the accuracy of the listed commands, but errors (e.g. typos in commands) remain possible. The manual is updated at relatively regular intervals (ca. 4 months). Suggestions how to improve this manual or notifications on errors are more than welcome – please send them via e-mail to Eva Thomann (escriba[at]hotmail.ch). Please note that we only consider those suggestions that we deem relevant for the analytic steps described in this manual, and that are formulated in a concrete manner. So far, we would like to wholeheartedly thank the following colleagues for their useful input:

Adrian Dusa
Carsten Q. Schneider
Yixian Sun
Koen van der Krieken
Alrik Thiem
Daniel Belling

What is written in this manual represents the opinion of the authors and not the opinion of those who helped us with their feedback. All remaining errors are our own responsibility.

The latest version of the manual is available online at http://www.evathomann.com/links/qca-r-manual.

Please cite this manual:

Thomann, Eva, Ioana-Elena Oana and Stefan Wittwer (2018). *Performing fuzzy- and crisp set QCA with R: A user-oriented beginner's guide*. **Version 07.03.2018**. URL: http://www.evathomann.com/links/qca-r-manual [date of access: **insert here**].

Eva Thomann

Nena Oana
Stefan Wittwer Bamberg, 7 March 2018

1 Introduction to R[2]

1.1 Basics

Performing the operations described in this manual requires downloading the following software:

- **R** (freely available at http://cran.rstudio.com/)
- **RStudio** (freely available at http://www.rstudio.com/products/rstudio/download/)

Once installed, you don't have to open R; it suffices to open Rstudio.

On the upper right, the *workspace* tab shows all the active objects (see 1.5). On the *lower right*, the history tab shows a list of commands used so far. The files tab shows all the files and folders in your default workspace. The plots tab will show all your graphs. The packages tab will list a series of packages or add-ons needed to run certain processes. For additional info see the help tab. On the left-hand side, you see the *console*, which is where you can see output (i.e. the results of what you do). You can also type commands there. However, you don't only want to give instructions to R, but you also want to save these instructions, so that you can repeat them any time you want, continue your work the next day, check and modify what you already did, show what you did to colleagues, make everything you did fully replicable, etc. Therefore, you want to save your commands in a script, which is written with the *editor*. Always use the editor, not the console, to enter code. Once code grows, a good and clear editor becomes indispensable (e.g. to identify errors or to allow comments). To open the editor and start a new script, click *file -> new file – R script*. Now, the editor window appears in the upper left part of the interface; and the console is now below the editor. The editor is where you enter your codes and documentation. Save your script frequently: *file -> save* or *save as*.

It is advisable to use comments extensively in your script to document what you do and why. Comments in the script start with # (on Mac: "alt+3"): everything after # in the line is ignored by R, that is, it is not treated as a command.

```
c 1
#whatever
```

If you use several #'s before and after the comment, Rstudio recognizes it as a section title to

[2] This introductory chapter is partly based on material from the Seminar "Data Analysis with R" by Rudi Farys and Paul Bauer (University of Bern, WISO) and Chapter 2 of Thiem and Dusa (2012).

which you can navigate:

```
c 2
##### mytitle ######
```

To navigate to a title, use the tiny bar on the lower end of the editor.

Once you have typed a command, R will not run it until you send the code to the Console by marking it and then pressing "STRG + R" (Windows) or "cmd + Enter" (Mac). Within the RStudio script, you can also just press STRG and Enter to run the command line where your cursor lies. You can also mark those commands you want to run and press STRG and Enter.

Box 1: Update R and Rstudio from time to time

New versions of R and Rstudio are issued regularly. Situations can arise in which packages rely on these new versions, while you still have an older version installed. This may be one possible reason in case you encounter problems. The easiest way to avoid this is to download the most recent versions before you get going, if you haven't used R for a while.

1.2 Working directory

To tell R where you want to save your files, you have to set the working directory. You can find out what the currently set working directory is:

```
c 3
getwd()
```

Then, set your working directory with setwd(""). The easiest way to do this is to click *session -> set working directory -> choose directory*. Browse to the file location where you want to store the relevant files for your analysis. The path to the working directory will then appear in the console. Copy-paste it into the script (without the >), so you can set the working directory just by running that line the next time:

```
c 4
setwd("/Users/mypath")
```

You can also simply copy-paste the path to the location from the header of your windows folder into the quotation mark of this command. If you do this, remember to replace \ by / or \\ in the copy-pasted folder location.

Display the content of (= the files in) the working directory:

```
c 5
dir()
```

1.3 Help!

Especially in the beginning, things often do not work immediately in R. The most common cause for this is that R is case-sensitive. This means that as soon as something is not typed the way R expects it, it does not recognize it. Check uppercase and lowercase notation; whether commas, dots and quotation marks are set correctly; and generally for typos. R uses different quotation marks than word. If R studio keeps crashing down, check whether changing the file location, splitting up your codes into several, smaller scripts, or enlarging the "plots" window (lower right) helps.

To get help, just put "?" in front of a command. The ? command gives you the relevant information and examples for a specific command. For example, to get help for the function getwd():

```
c 6
?getwd
```

The ?? command gives you a list of possible help sources for some keyword (in case you don't know the command, but need help on a topic):

```
c 7
??qca
```

use "" for several words:

```
c 8
??"descriptive statistics"
```

R offers excellent online documentation: help yourself with google! You are never alone with your R-problems.

1.4 Packages

If R is the kitchen, packages are the kitchenware. There are packages for everything. For example, the packages QCA and SetMethods can be used for performing QCA with R. Some packages are loaded permanently by default („base" packages) while others must be installed:

```
c 9
install.packages("packagename")
```

So, after starting R studio and opening the editor, copy-paste the following command into the editor, mark it, and hit STRG and Enter:

```
c 10
install.packages(c("arm", "car", "gmodels", "Hmisc", "MASS", "memisc", "polycor", "psych",
"reshape", "VIM", "lattice", "XML", "xtable", "foreign", "directlabels", "betareg", "plyr",
"dplyr", "QCA", "SetMethods"), dependencies = TRUE)
```

Probably a window will pop up, where you need to choose a server to download the packages.

The dependencies option ensures that, if a package depends on the existence of another package, that other package is installed too.

You only need to install a package once. But every time you want to use it, you have to load it when starting the R session. Load an installed package (without quotes!):

```
c 11
 library(packagename)
```

To work with this manual, you will have to load the following packages:

```
c 12
library(lattice); library(arm); library(xtable); library(foreign); library(psych);
library(directlabels); library(betareg); library(VIM); library(base); library(plyr);
library(dplyr); library(QCA); library(SetMethods)
```

See which packages are loaded:

```
c 13
search()
```

You are now ready to go.

1.5 Workspace, operators and objects

R works with objects. Objects can be everything. For example, datasets are objects, variables are objects. You can even store your results as an object, and recall them any time you want. Data are contained in objects of different size and format (*object classes*) such as data sets, a list of numbers, or only a single number or a name. Functions use the content of an object and produce results.

1.5.1 Workspace

Objects are stored in the *workspace*. There (upper right), you can see which data sets are loaded and which results or other objects you stored. Always keep it clear and well-arranged.

As R is object-oriented, not only functions can be treated as objects, but also results from operations using the functions can themselves be saved as objects again. To generate an object, we first give it a name and then use a backward arrow <- to define its content and store it in the workspace. For example, here we create an object that consists of the phrase “hi there”:

```
c 14
myobject <-  "hi there"
```

You can always display the content of the object in the console by simply typing its name:

```
c 15
myobject
```

R tells you: "hi there"

But the content of the object can also be numerical, of course (for numbers, you don’t need the quotation marks):

```
c 16
myobject <-  37
myobject
```

R tells you: 37

c() is a function which combines its arguments (c stands for "concatenate"). This helps you to list different elements. This function is often used with R. In the example below, we create an object that consists of the numbers 1, 2, and 3.

```
c 17
myobject <- c(1, 2, 3)
```

You can also list words or phrases, instead of numbers, using quotation marks:

```
c 18
myobject <- c("hi", "what’s up?", "coffee please.")
```

The dollar sign (see also 1.6.3) is needed if you want to “tie” an object (e.g. a variable) to another object (e.g., the dataset). For example, you create a new variable (=the tied object) in the dataset (the object to which it is tied). Without doing this, the variable will only be in the workspace but not in the dataset. You have to specify the content of the variable. Here we assign it a value of NA (no answer):

```
c 19
mydata$newvar <- NA
```

Save objects from the workspace in a file:

```
c 20
save(myobject1, myobject2, file="filename" )
```

You can remove specific objects from the workspace:

```
c 21
rm(myobject)
```

or delete (clear) the whole workspace:

```
c 22
rm(list=ls())
```

1.5.2 Operators

You can use R much like a calculator. We might be using the following *operators*:

Arithmetic operators:

Plus: +
Minus: –
Multiplication: *
Division: /
To the power of: ^

Logical operators:
AND: &
OR: |
EQUALS: == (attention: you need a double equal sign!)[3]
NOT: !
DOES NOT EQUAL: !=
SMALLER THAN: <
GREATER THAN: >
SMALLER THAN OR EQUAL TO: <=
GREATER THAN OR EQUAL TO: >=

Functions (e.g.): sqrt(), exp(), log()

To use the results of operations for later analysis, store them as an object (see 1.5.1). Otherwise, the result only appears in the console. For example, here we create an object that is the result of the operation ((1+3)/7*18)^2:

[3] The simple equal sign = is equivalent to the backwards arrow <- that we use for storing information into objects.

c 23
```
myobject <- ((1+3)/7*18)^2
```

You can also use the operators to compare differing objects or perform operations on them. Say, you define object "a" as:

```
a <- (5+5)/2*2
```

You may want to know whether a is greater than 10:

```
a > 10
```

R will tell you no: FALSE

Does a equal 10?

```
a == 10
```

R will tell you yes: TRUE

1.5.3 Classes of objects

In R, objects have different *classes* (see http://adv-r.had.co.nz/Data-structures.html for further information):

- Vectors: integer, numerical, logical, character.
- Matrix (two dimensions),
- List (list of different objects),
- Data frame.

You can check for the object type:

c 24
```
class(myobject)
```

Of course, some commands only work with some object classes. For example, you cannot calculate the mean of a logical object, but only of a numerical one.

Logical objects typically take on the values TRUE or FALSE. It is nice to know that you can use the command as.numeric() to display logical values as numerical values (1 for TRUE, 0 for FALSE).

c 25
```
as.numeric(logicalobject)
```

This enables you to, for example, count the number of cases that fulfil a certain logical condition, see e.g. section 3.2.

1.6 Working with real data

1.6.1 Open and save datasets

To import a data set, make sure to set the working directory right (getwd()). Before you start working on a new analysis with a new dataset[4], you want to clear workspace to avoid that the software "gets confused".

c 26
```
rm(list=ls())
```

For performing a QCA analysis, we usually work with .csv files.

To load the dataset, we build an object that reads the .csv file:

c 27
```
mydata  <- read.csv("mydata.csv", row.names = 1, header = TRUE, sep = ";",  dec = ",")
mydata
```

This is the option for excel files from countries that use a comma as decimal point and a semicolon as field separator (sep = ";", dec = ","), the excel convention for CSV files in some Western European locales. If this does not work, it may be that your excel is formatted differently (e.g. US version). Try the option sep = "", dec = "." instead, or sep = ",", dec = ".". If you want to find out how your CSV file is structured, open it in your working directory with the "Editor" (Windows) or "TextEdit" (Mac). The option row.names = 1 tells R that the first column in our dataset contains the names of the rows, which in a QCA analysis are usually the names of the cases. The option header = TRUE tells R that the first row in our dataset represents the names of the columns, which in a QCA analysis are usually the names of our variables or of our sets after calibration. If your dataset is structured differently, modify these options accordingly. For example, if you do not have the names of the cases stored in the dataset, omit the option row.names = 1 from the read.csv() command. In that case, R will automatically assign the rows a number as the case name.

You can also read the dataset in other formats, e.g. simply as a table:

c 28
```
mydata <- read.table ("mydata.csv", sep =";", dec=",", header=TRUE)
```

[4] In case your analysis requires operations with more than one dataset you can, of course, omit clearing the workspace in order to allow having multiple datasets simultaneously stored.

For STATA files (package "foreign"), use read.dta; for SPSS, read.spss .

To save your dataset in the working directory:

```
c 29
write.csv2(mydata, "mydata.csv")
```

write.csv2 uses a comma for the decimal point and a semicolon for the separator. If you want to save your csv in the US format, use the write.csv instead of write.csv2. This uses "." for the decimal point and a comma for the separator.

Check

```
?write.table
```

for other possible data export.

You can also create your own datasets by "tying" variables together. Suppose we have, in a dataset mydata, three variables named name, age, and height (N.B. they must be of the same size or, in other words, have the same number of elements) which we store into objects by using the backward arrow <- and the c() function. You can now create a dataset called people containing these three variables.

```
c 30
people <- as.data.frame(cbind(mydata$names, mydata$age, mydata$height))
```

Similarly, you can save calibrated sets in a new dataset by selecting just a subset of your initial dataset, here called myrawdata (see also sections 1.6.2 and 4.1). Assume, for example, that you only want to save three calibrated sets from your dataset as a new dataset "fuzzydata". You may want to do this because QCA has some commands that only work when all data ranges from 0-1. One possibility is using the subset() command (package: base):

```
c 31
myfuzzydata <- subset(myrawdata, select = c("MYSET1", "MYSET2", "MYSET3"))
write.csv2(myfuzzydata, "myfuzzydata.csv")
```

You can also save a subset of your data that fulfils certain conditions – you are incredibly flexible there, using the logical operators in 1.5.2. Note that if you have a variable consisting of character vectors (=words rather than numbers), you should use quotation marks for indicating which value you mean. Say, for example, you want to create a new dataset that contains only those cases that have the value "Eastern Europe" (=character vector) for the variable "region",

and that either have a value of bigger than 0.5 on variable 2, or a value of 0 on variable 3:

```
c 32
mynewdata <- subset(myolddata, region == "Eastern Europe" & (var2 >= 0.5 | var3 == 0))
mynewdata
write.csv2(mynewdata, "mynewdata.csv")
```

Alternatively, using colnames(), in this example you saw that the calibrated sets are the columns 8 to 10. You can tell R to save columns 8 to 10 in a new dataset:

```
c 33
write.csv2(myrawdata[,8:10], "myfuzzydata.csv")
```

1.6.2 Inspect and describe data

Much of what is discussed in this section presumes you have installed the package “dplyr”.

The View() (attention: “V” as a capital letter) command makes you see your dataset as if you opened it normally.

```
c 34
View(mydata)
```

Attention: you cannot make any changes in the dataset using the cursor!

Check how many variables you have in your dataset:

```
c 35
length(mydata)
```

Look up the names of the variables contained in your dataset:

```
c 36
names(mydata)
```

You can also use colnames() or rownames() to get the names of the columns (variables) or rows (cases), respectively:

```
c 37
colnames(mydata)
```

```
c 38
rownames(mydata)
```

Each column and each row has a number. In the console, R lists the number of the first element (row or column, depending on which command you chose above) in every new line in square

brackets. This helps you figure out which number a specific case or variable has in the dataset.

Get a first impression of how your dataset looks like (first 6 rows):

```
c 39
head(mydata)
```

Look up specific variables in your dataset (here: 2 variables, but it can be less or more, of course):

```
c 40
select(mydata, var1, var2)
```

Apart from select (which only works for variables), you have three basic options to access elements or subsets of your dataset:

1. *By using $*

The $ sign means something like “and therein”, for example “variable in mydata”:

```
c 41
mydata$var
```

2. *By using objectname[rows,columns]*

You can use square brackets to specify which part of the dataset you want to see. Before the comma, you specify the number of the row(s), and after the comma, the column(s). If unspecified, all rows / columns are displayed. Here, we look up the third column (i.e. variable) of mydata:

```
c 42
mydata[,3]
```

Now, we look up the value of the first case for the third variable:

```
c 43
mydata[1,3]
```

You can do the exactly same thing by specifying the name of the variable in the dataset:

```
c 44
mydata$var[1]
```

You can also list several elements you want to see. For example, you can look up how the first, second and eighth variable look like for rows (cases) number 2-4:

c 45
```
mydata[2:4, c(1,2,8)]
```

3. *By using subset().*

The subset command is extremely useful to identify subsets of your data that fulfil certain logical conditions. For example, we want to see only those cases that have a value of greater than 5 in the first variable and a value smaller than or equal 60 in the second variable.

c 46
```
mynewdata <- subset(mydata, var1 > 5 & var2 <= 60)
mynewdata
```

You could save these cases in a new dataset:

c 47
```
write.csv2(mynewdata, "mynewdata.csv")
```

Sort a variable in ascending order:

c 48
```
mydata[order(mydata$var, decreasing=FALSE), ]
```

Sort in descending order:

c 49
```
mydata[order(mydata$var, decreasing=TRUE), ]
```

Get descriptive statistics for your dataset:

c 50
```
describe(mydata)
```

Or perform descriptive statistics separately:

c 51
```
mean(mydata$var)
median(mydata$var)
sd(mydata$var)
```

Check if there are missings in your dataset or a specific variable (see also section 3.2). R will give you, for each case, the answer TRUE (is missing) or FALSE (not missing).

c 52
```
is.na(mydata)
```

c 53
```
is.na(mydata$var)
```

1.6.3 Recoding, renaming and deleting variables

You can attribute values to cases. For example, you want the first case to have a missing value (NA, 'Not Available') for the third variable:

```
c 54
mydata[1,3] <- NA
```

For recoding variables, you have two basic options:

1. *Using the recode() function (package: QCA)*

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

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

In the example below, we recoded var into newvar, such that values of 1, 3 and values between 5 and 7 obtain the value 1; values of 4, 5, 8 and 9 obtain value 2; and all other values retain the same values as the old variable (else=**copy**).

```
c 55
mydata$newvar <- recode(mydata$var, "1,3,5:7=1; 4,5,8,9=2; else=copy")
```

2. *Using logical and other operators*

This second option looks a bit more complicated than the first option, but you are more flexible in the rules you can use. You can create a new variable in the dataset to recode another variable. For example, we dichotomize a variable: values above 6.33 are recoded into 1, all others into 0. We use $ to ensure that the new variable (object) is tied into the dataset.

First, we create a new variable, containing only missings.

```
c 56
mydata$newvar <- NA
```

Instead of creating a new variable with missings, you can also create a new variable that equals the old variable:

```
c 57
mydata$newvar <- mydata$var
```

Assign the new variable a value of 0 if the values of the old variable are lower than or equal to 6.33:

```
c 58
mydata$newvar[mydata$var <= 6.33] <- 0
```

Then, we assign the value 1 to values of the old variable that are higher than 6.33:

```
c 59
mydata$newvar[mydata$var > 6.33] <- 1
```

Needless to say, you can use all logical operators (see 1.5.2) for this and any values you like.

You can also perform operations with variables to create a new variable. For example, we create a new variable which divides variable 2 by variable 1.

```
c 60
mydata§newvar <- mydata$var1/mydata$var2
```

Always check whether the recoding worked. Compare the old and the new, recoded variable:

```
c 61
select(mydata, var, newvar)
```

If you wish to rename a variable (here: rename “var” with the new variable name “newvar”):

```
c 62
names(mydata)[names(mydata)=="var"] <- "newvar"
```

or

```
c 63
mydata <- rename(mydata, newvar = var)
```

Alternatively, to rename a variable, you can simply create a new variable that equals the old variable, and then remove the old variable:

```
c 64
mydata$newvar <- mydata$var
mydata$var <- NULL
```

And in fact, you can delete any variable any time (attention: you won’t be able to undo this!):

```
c 65
mydata$var <- NULL
```

2 Getting started for the QCA analysis

The manual is based on the use of R Studio. **In case you did not already do this (section 1.4),** you first have to install the QCA package and the SetMethods package required to perform QCA, as well as some additional auxiliary packages with all dependencies. Open R studio and simply copy-paste the following command into the console (left-hand side):

```
c 66
install.packages(c("arm", "car", "gmodels", "Hmisc", "MASS", "memisc", "polycor", "psych",
"reshape", "VIM", "lattice", "XML", "xtable", "foreign", "directlabels", "betareg", "plyr",
"dplyr", "QCA", "SetMethods"), dependencies = TRUE)
```

To execute the command, mark the command and either

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

Within the RStudio script, you can also just press STRG and Enter to run the command line where your cursor lies.

Probably a window will pop up, where you need to choose a server to download the packages.

Click *file, new file, R script* to start writing an R code (upper left side). You can save the script clicking *file, save* or *file, save as…* .

Clear your workspace before starting to work:

```
c 67
rm(list = ls())
```

Once the packages are installed, load the required packages:

```
c 68
library(lattice); library(arm); library(xtable); library(foreign); library(psych);
library(directlabels); library(betareg); library(VIM); library(base); library(plyr);
library(dplyr); library(QCA); library(SetMethods)
```

Set your working directory (see 1.2). It should be set to the same folder to which you safe your R script. All datasets used should be saved in that folder, too (attention: all of these should be saved in the top level of the folder and not in other subfolders!). For example:

```
c 69
setwd("C:/Users/Eva/Work/ Data")
```

Box 2: Working with the visual GUI interface

Some people find it complicated to work with command lines; they prefer having a visual interface where they can click buttons to go through their analysis. In fact, the QCA package offers this option, using the runGUI() command. The interface is very intuitive and has many similarities with that of Ragin's fs/QCA software. You will be able to do a basic QCA and produce nice graphs with it. However, it is also still in development and therefore does not yet offer the full functionality we employ in this manual (and according to the QCA manual, there can still be bugs). To find out more about this, type:

?runGUI

3 Opening, preparing and saving datasets

Some steps described in this chapter are repetitions of section 1.6.

3.1 Opening, describing and saving the dataset

See also section 1.6.

Load your csv dataset. Modify the options row.names, header, sep, and dec according to your own dataset, see section 1.6.1.

```
c 70
mydata <- read.csv("mydata.csv", row.names=1, header = TRUE, sep = ";",  dec = ",")
```

Check the labels of your variables:

```
c 71
names(mydata)
```

Check the labels of your cases:

```
c 72
rownames(mydata)
```

Sort the values for a variable in ascending order:

```
c 73
mydata[order(mydata$var, decreasing=FALSE), ]
```

Compare several (here: two) variables with each other:

```
c 74
select(mydata, var1, var2)
```

Most important descriptive stats:

```
c 75
describe(mydata)
```

Save your dataset:

```
c 76
write.csv2(mydata, "mydata.csv")
```

Save a subset of your dataset (here: sets 1, 2 and 3; for other options, see 1.6.1):

```
c 77
mynewdata <- subset(myolddata, select = c("MYSET1", "MYSET2", "MYSET3"))
write.csv2(mynewdata, "mynewdata.csv")
```

3.2 Dealing with missing data

R only recognizes values of "NA" for missing values. If missing data is denoted by a different sign – e.g., "-99", you can (and have to) convert these into NAs:

```
c 78
mydata[mydata==-99] <- NA
```

QCA cannot attribute cases with missing values to the truth table, so you have to exclude cases with missing values from the analysis.

You can check whether there are missing observations in your dataset (TRUE = missing, FALSE = not missing):

```
c 79
is.na(mydata)
```

You can also check this for specific variables:

```
c 80
is.na(mydata$var)
```

And you can check whether the cases are complete, i.e. do NOT contain missings (TRUE = no missing values, FALSE = has one or several missing values):

```
c 81
complete.cases(mydata)
```

You can obtain the *number of cases with missing values* (0) and *with no missing values* (1):

c 82
```
nomissings <- as.numeric(complete.cases(mydata))
table(nomissings)
```

and based on this you can obtain the percentage of cases with (left-hand side of crosstable) and without (right-hand side) missing values:

c 83
```
prop.table(table(nomissings))
```

Obtain the number of cases with missing values on a specific variable. This time, these will be cases with value 1 on the object, because we are asking for those cases with the value NA ("is.na"):

c 84
```
varmiss <- as.numeric(is.na(mydata$var))
table(varmiss)
```

Percentage of cases with missings, on the right-hand side of the crosstable:

c 85
```
varmiss <- as.numeric(is.na(mydata$var))
prop.table(table(varmiss))
```

Identify the names of the cases that have missing values on the variable:

c 86
```
varmiss <- as.numeric(is.na(mydata$var))
rownames(subset(mydata, varmiss==1))
```

You can also check this for several variables simultaneously. The example below presumes that we first used as.numeric(is.na()) (see above) for identifying the cases with missing values for variables 1, 2 and 3. So you can then identify those cases that have missing values on one or several variables of interest to you.:

c 87
```
rownames(subset(mydata, var1miss==1 | var2miss==1 | var3miss==1))
```

If you wish to exclude cases with missing values from your dataset, you can simply only include cases with no missing values on any variable in the dataset:

c 88
```
nomissings <- as.numeric(complete.cases(mydata))
mydata = mydata[nomissings== 1,]
```

OR you can first identify cases with missing values for certain variables (here: variables 1, 2 and 3), and then only include only those cases *without* missing values in the dataset:

```
c 89
var1miss <- as.numeric(is.na(mydata$var1))
var2miss <- as.numeric(is.na(mydata$var2))
var3miss <- as.numeric(is.na(mydata$var3))
mydata = mydata[var1miss == 0,]
mydata = mydata[var2miss == 0,]
mydata = mydata[var3miss == 0,]
```

You can then save this new dataset (see 1.6.1).

3.3 Recoding variables

Package: QCA; see also section 1.6.3. Say you want to recode values of 4 to values of 3 for a given variable. You can EITHER create a new variable:

```
c 90
mydata$varnew <- recode(mydata$var, "4=3; else=copy")
```

OR you could simply overwrite the existing variable (but then, the old values are lost):

```
c 91
recode(mydata$var, "4=3; else=copy")
```

4 Calibration of and operations with sets

References:

Ragin (2008b, 2009), Schneider and Wagemann (2012: 32-52, 232-244), Thiem and Dusa (2012: 27-32, 51-62, 2013b: 89).

4.1 Calibration

In order to decide about the appropriate calibration threshold (section 4.1.3), you want to look at your data (section 1.6.2), play around with it (section 1.6.3), and visualize different calibration options and how they affect skewness (section 4.2). The nice thing about R is that you can very flexibly try out different things and go back and forth between these different steps. After calibration, you want to save what you did (section 1.6.1). Always label your sets

using uppercase notation.

4.1.1 Fuzzy sets

Direct method of calibration

Simply enter the thresholds for full nonmembership "e" (here: 1), the crossover point "c" (here: 2.5), and full membership "i" (here: 4).

```
c 92
mydata$MYFUZZYSET <- calibrate(mydata$rawvar, type = "fuzzy", thresholds = "e=1, c=2.5, i=4", logistic = TRUE)
```

By default this command uses a logistic function and values of 0.05 and 0.95 as thresholds for full (non)membership. If you set logistic=FALSE, then a linear function is used for transforming the raw data into a set (Package: QCA).

You can additionally tell the software to code cases with certain values on the raw variable as a certain set membership. Here, for example, after calibration we want to code cases with a value of 3 on a five-point Likert scale (3 = neither agree nor disagree) of the raw variable as fully out of the (already calibrated) fuzzy set (0.05):

```
c 93
mydata$MYFUZZYSET[mydata$rawvar == 3] <- 0.05
```

Box 3: Rounding sets (and any other variable)

By default, R calibrates sets with as many decimals as there may be. However, you can round calibrated sets (here: 2 decimals):

```
c 94
MYFUZZYSETROUNDED <- round(MYFUZZYSET, digits = 2)
```

or you can directly round when calibrating, e.g.:

```
c 95
mydata$MYFUZZYSET <- round(calibrate(mydata$myrawvar, type = "fuzzy", thresholds = "e=1, c=2.5, i=4", logistic = TRUE), digits=2)
```

Note that whether or not you round the sets slightly affects the parameters of fit.

Fuzzy sets with multiple degrees/Theoretical calibration (so-called multi-value fuzzy sets)[5]

As described in Ragin (2009: 91), especially when we are not dealing with fine-grained interval-level data, we often resort to fuzzy sets with multiple, qualitatively defined degrees (e.g., 5-value fuzzy sets or 7-value fuzzy sets), based on a qualitative classification of observations into fuzzy set membership scores – in other words, a procedure of coding / recoding raw data. This procedure is often referred to as "indirect calibration", but to be precise, it describes only the first step of a two-step, quantitative procedure which is rarely applied in practice (see also Ragin 2008b; Schneider and Wagemann 2012: 35ff). When using this procedure, be careful not to assign values of 0.5 to empirical cases.

In order to code fuzzy sets with multiple degrees, you can assign a fuzzy set membership for specific values on the raw variable – essentially, recode a variable into a set (see for more details section 1.6.3). Here, for example, we create a four-value fuzzy set (0, 0.3, 0.7, 1) out of a raw variable that ranges from 0 to 3:

```
c 96
mydata$MYFUZZYSET <- recode(mydata$rawvar, "0=0; 1=0.3; 2=0.7; 3=1; else=NA")
mydata$MYFUZZYSET
```

We can also assign certain fuzzy set values to cases that fall within a given range of the raw variable that you can define as you see fit. In the example here, raw values from 1 to 3 result in a fuzzy value of 0, raw values greater than 3 but smaller than 5 are assigned a fuzzy value of 0.33, and raw values equal or greater than 5 get a fuzzy value of 1:

```
c 97
mydata$MYFUZZYSET <- NA
mydata$MYFUZZYSET[(mydata$rawvar >= 1)&(mydata$rawvar <=3)] <- 0
mydata$MYFUZZYSET[(mydata$rawvar >3)&( mydata$rawvar < 5)] <- 0.33
mydata$MYFUZZYSET(mydata$rawvar >= 5) <- 1
mydata$MYFUZZYSET
```

Depending on what you prefer, you can do this using the recode() function or using logical operations (see 1.6.3 and 3.3). The calibrate() function also offers a subsets of the possibilities described here if the option type= "crisp", see the QCA package manual.

[5] Multi-value fuzzy sets, capturing ordinal or higher-scaled data, should not be confused with (by definition multinomial) multi-value sets used in multi-value set QCA (mvQCA). The latter are not covered in this manual.

4.1.2 Crisp sets

We can calibrate a crisp set by simply indicating the crossover point (here: 50) (package: QCA):

```
c 98
mydata$MYCRISPSET <- calibrate(mydata$rawvar, type = "crisp", thresholds = 50,
include = TRUE)
```

If you set include = TRUE, values of 50 will be calibrated as set membership 1. If it is set to FALSE, values of 50 will be counted as set membership 0.

Calibrating crisp sets can also be done by recoding the data, see 4.1.1 (and 1.6.3). Again, we can either assign concrete raw values to concrete crisp set memberships. For example, we assign raw values of 1 and 3 a crisp set membership of 0, and raw values of 2, a crisp set value of 1:

```
c 99
mydata$MYCRISPSET <- recode(mydata$rawvar, "1=0; 3=0; 2=1; else=copy")
```

Or we can define how specific ranges of the raw values result in crisp set memberships. For example, we assign raw values between 0 and 2 a crisp set membership 0, and raw values above 2, a crisp set membership 1:

```
c 100
mydata$MYCRISPSET <- recode(mydata$rawvar, "0:2=0; else=copy")
mydata$MYCRISPSET(mydata$rawvar > 2) <- 1
```

4.1.3 Tips for calibration (read this)

The following tips will make your life a lot easier.

1. Labelling sets

Future versions of QCA will have a lot of commands that allow you to negate sets using lowercase notation. In this scenario, whether you use upper- or lowercase letters is really decisive. To avoid problems and confusion, always use *uppercase letters for labelling your sets, and lowercase letters for their negation*.

QCA analyses are often perceived as very complex and hard to understand by outsiders. It considerably reduces the perceived complexity of QCA results if you *use short labels for your sets*. It also makes it easier to fit results into tables and figures. Ask yourself: do I need more than 1 letter to describe my set, and how many more do I minimally need?

2. Finding calibration thresholds

The QCA package does offer data-driven ways to find calibration thresholds. They are not included in this manual because the resulting sets are very hard to interpret. The most important analytic choice is that of the threshold that establishes the difference in kind. *Whenever possible, use conceptual and theoretical criteria for the crossover point and avoid using purely empirical criteria such as descriptive statistics*. *Avoid using the median as crossover point*: it can usually not be interpreted other than the set of "cases with values equal as or higher than 50% of the other cases". Similarly, if you use the sample mean (e.g., for unemployment) as crossover point, the conceptual meaning of the set is "unemployment above average in the cases observed". All this does not mean that empirical criteria are not important for determining calibration thresholds. In particular, you should *avoid overly skewed sets (4.2.2 and 8.1) and empirical cases on the crossover point (4.2.3).*

Technically, it is easy to calibrate sets. However, calibration is essentially a process of concept formation and definition that interacts decisively with your theory, research design and results. Therefore, calibration is one of the most demanding analytic phases of a QCA analysis. *Take **a lot** of time for calibration, and try out different options*. If you find that several different crossover points are equally plausible, *try them all and see how that affects the analysis*. This is called a robustness test and a very good thing to do (see also Maggetti and Levi-Faur 2013). See 4.2.1 and the online appendix of Hinterleitner, Sager and Thomann (2016) for an illustration.

3. Saving calibrated data

After calibrating your raw data, *save the calibrated sets in a new dataset*. Some commands do not work if the dataset does not contain only variables that range from 0-1. We have seen in 1.6.1 how this works:

```
c 101
myfuzzydata <- subset(myrawdata, select = c("MYSET1", "MYSET2", "MYSET3"))
write.csv2(myfuzzydata, "myfuzzydata.csv")
```

4.2 Calibration diagnostics

During calibration, you want to see how your data is distributed, and know if you have (and avoid) overly skewed sets and empirical cases with set membership 0.5.

4.2.1 Visualization

You can visualize the calibration with an XY plot, which will equally show you not only if there are cases on the 0.5 threshold, but also how the cases distribute in the set (see also section 5 on graphs).

A basic way of doing this can be to plot a set against its raw scores and set a horizontal line at the crossover point, as well as a vertical line at the raw value that indicates the crossover point (here: 25) (package: graphics).

```
c 102
plot(mydata$rawvar, mydata$MYSET, pch=18,  col="black",
    main='MYSET',
    xlab=' Raw score ',
    ylab=' Fuzzy score ')
abline(h=0.5, col="black")
abline(v= 25, col="black")
```

In the example below, we plot the calibration of "MYSET". The crossover point is set at 25, and indicated by a vertical black line (v= 25); we have also added a horizontal black line at set membership 0.5 (h=0.5 – optional). In addition, the last two command lines (optional) add two dotted vertical lines to the graph to indicate two alternative plausible crossover points (18 and 50) that we decided could be tested for robustness. The plot shows us whether, for example, changing the crossover point from 25 to 50 would change the qualitative set membership of an empirical case (indicated by a dot in-between the black and the left-hand side dotted line).

```
c 103
plot(mydata$rawvar, mydata$MYSET, pch=18,  col="black",
    main='MYSET',
    xlab=' Raw score ',
    ylab=' Fuzzy score ')
abline(h=0.5, col="black")
abline(v= 25, col="black")
abline(v= 18, col="black", lty="dotted")
abline(v= 50, col="black", lty="dotted")
```

In the following plot, we only test for one possible alternative crossover point (25 = regular crossover point, 18 = alternative crossover point). The line for the alternative crossover point is shaded grey, and we also add the data curve for the alternative calibration using the points option – again, shaded grey.

```
c 104
plot(mydata$rawvar, mydata$MYSET, pch=18,  col="black",
    main=' MYSET ',
    xlab=' Raw score ',
    ylab=' Fuzzy score ')
points(mydata$rawvar, mydata$MYALTERNATIVESET, pch=18, col="grey",
lty="dotted")
abline(h=0.5, col="black")
abline(v= 25, col="black")
abline(v= 18, col="grey", lty="dotted")
```

4.2.2 Skewness

To check the skewness of your set, you can identify the number of cases that have set membership above 0.5 and check whether there is a disproportionate amount of them in your data.

```
c 105
skewMYSET <- as.numeric(mydata$MYSET > 0.5)
sum(skewMYSET)
```

Obtain the percentage of cases with set membership above 0.5 (right-hand side value):

```
c 106
prop.table(table(skewMYSET))
```

Identify the names of the cases with set membership above 0.5:

```
c 107
rownames(subset(mydata, MYSET > 0.5))
```

4.2.3 Cases on crossover point

You can check the number of cases that have membership of 0.5 in a set:

```
c 108
checkMYSET <- as.numeric(mydata$MYSET == 0.5)
sum(checkMYSET)
```

And their percentage:

```
c 109
prop.table(table(checkMYSET))
```

It should be 0. If it is not, find out which cases have membership 0.5 on a set:

```
c 110
rownames(subset(mydata, checkMYSET==1))
```

You can also check this for several sets simultaneously:

```
c 111
rownames(subset(mydata, checkMYSET1==1 | checkMYSET2==1 | checkMYSET3==1))
```

You can exclude all cases that have values of 0.5 on one of your sets (here: outcome and 3 conditions):

```
c 112
mydata = mydata[mydata$OUTCOME != 0.5,]
mydata = mydata[mydata$COND1 != 0.5,]
mydata = mydata[mydata$COND2 != 0.5,]
mydata = mydata[mydata$COND3 != 0.5,]
mydata
```

Box 4: A hands-on template code for calibration and its diagnostic

To cut a long story short, for your convenience, here is a ready-made, hands-on template code for what you could use as a standard procedure when calibrating a fuzzy set using the direct calibration method:

```
c 113

# descriptive statistics
describe(mydata$var)

# check for missings (% of cases with missings)
varmiss <- as.numeric(is.na(mydata$var))
prop.table(table(varmiss))

# calibration (fuzzy set, direct calibration method, rounded to 2 digits)
mydata$MYFUZZYSET <- round(calibrate(mydata$var, type = "fuzzy", thresholds =
"e=1, c=2.5, i=4", logistic = TRUE), digits=2)

# Number of cases on the crossover point
checkMYSET <- as.numeric(mydata$MYSET == 0.5)
sum(checkMYSET)

# check for skewness (% of cases with membership > 0.5)
skewMYSET <- as.numeric(mydata$MYSET > 0.5)
```

```
prop.table(table(skewMYSET))

# visualize calibration
plot(mydata$var, mydata$MYSET, pch=18,  col="black",
    main='MYSET',
    xlab=' Raw score ',
    ylab=' Fuzzy score ')
abline(h=0.5, col="black")
abline(v= 25, col="black")
```

4.3 Calculating membership in operations on sets

If you create a new set (a negated set, a disjunction or a conjunction, or a combination of these), you may want to store it as an object (NEWSET <- operation), so that you can use it for further analysis. You can also (but don't have to) tie it to the dataset by using the dollar sign (mydata$NEWSET <- operation), so that it appears as a new variable in the dataset.

You have several options to calculate the cases' membership in combined sets.

1. *Automatically, using compute() (package: QCA)*

The compute() command allows you to directly calculate the membership in any expression you wish to indicate, e.g. in truth table rows or the solution term. You can use either the tilde ~ sign or lowercase notation to negate sets, but be consistent. You can also skip the * sign if you prefer. Below we calculate the cases' membership in the set "set1*SET2 + set3*SET4*set5 + SET6" and store the values as an object "sol" for further use:

```
c 114
sol <- compute("myset1*MYSET2 + myset3*MYSET4*myset5 + MYSET6",
data=mydata)
```

2. *Via logical operators, using 1-, fuzzyand(), fuzzyor()(package: QCA)*

To calculate the cases' membership in a negated set, you can simply subtract the set from 1:

```
c 115
mydata$myset <- 1-mydata$MYSET
```

Note that here we use lowercase notation to label the negated set.

This way, you can also directly negate the set *within* a different command, without previously creating the negated set as a separate object. We do this with set 3 in the next example.

You can combine several sets (e.g., SET1*SET2*set3 that together form path 1 of the solution

formula) with the logical AND (which implements the minimum rule):

```
c 116
path1 <- fuzzyand(mydata$MYSET1, mydata$MYSET2, 1-mydata$MYSET3)
```

You can combine several sets (here: path1 + path2) with the logical OR using the maximum rule:

```
c 117
myunion <- fuzzyor(path1, path2)
```

You can use both fuzzyand()/pmin() and fuzzyor()/pmax() with either sets from your dataset (like mydata$MYSET1) or sets you created as objects (like path1).

You can also combine these commands in the same expression. For example, here we calculate membership in "set1*SET2 + set3*SET4*set5 + SET6" (but note: compute() does the same much faster!):

```
c 118
fuzzyor(fuzzyand(1 - MYSET1, MYSET2), fuzzyand(1 - MYSET3, MYSET4, 1 - MYSET5), MYSET6)
```

Box 5: Aggregating sets when you have missing values

You may encounter a situation in which you want to combine different sets using the logical OR or the logical AND, but one (or several) of the component sets has missing values. In this case, you may want the composed set to take on the (minimum or maximum) value of the component set(s) that does not have a missing value. In fact, this way, building composed sets can be a nice way to reduce sample dropout due to missing data.

The functions fuzzyand() and fuzzyor() , however, will not do that for you. You can use the functions pmin() (logical AND) and pmax() (logical OR) instead (package: base) and specify the option na.rm=TRUE:

```
c 119
MYSET <- pmin(mydata$MYSET1, mydata$MYSET2, 1-mydata$MYSET3, na.rm=TRUE)
```

```
c 120
MYSET <- pmax(mydata$MYSET1, mydata$MYSET2, 1-mydata$MYSET3, na.rm=TRUE)
```

3. *For solution terms: using $pims*

If you have calculated a QCA solution using minimize() (see section 7) and stored the solution as an object (e.g. psOUTCOME), then the cases' membership in the different paths of the solution term are stored in the solution object and can be looked up using

```
c 121
psOUTCOME$pims
```

You can select the path you wish to work with and, for example, store it as an object for further use. Here, we use path 1 which is "COND1*cond2":

```
c 122
path1 <- psOUTCOME$pims$`COND1*cond2`
```

You could do this for all paths of the solution term and then calculate membership in the whole solution term sol:

```
c 123
sol <- fuzzyor(path1, path2, path3)
```

Based on all this, you can make XY-plots of truth table rows or of your necessary or sufficient conditions and the outcome, see section 5.

5 XY-plots[6]

References:

Schneider and Rohlfing (2013), Schneider and Wagemann (2012: 305-312), Thiem and Dusa (2012: 80-83).

See also section 4.2.1 for visualizing calibration, and 8.1 for visualizing skewness checks.

You have four commands at disposal to produce xy-plots with R. For QCA, we will usually work with either xy.plot() (for plotting single conditions or customized predefined sets; XYplot() from the QCA package equally works) or pimplot() (for plotting the solution term).

[6] Venn-Diagrams can be performed with R, but are not covered in this manual. See https://cran.r-project.org/web/packages/VennDiagram/VennDiagram.pdf .

See *Box 6* for other options for making XY plots.

1. *xy.plot (package: SetMethods)*

The xy.plot command has the advantage that it enables us to identify cases in the plot (case.lab = TRUE):

c 124
xy.plot(**mydata$myx**, **mydata$myy**, case.lab = TRUE, labs = rownames(**mydata**), necessity=**TRUE**)

It also automatically integrates dotted lines for the quadrants, and a black diagonal; and it indicates the parameters of fit (consistency, coverage, PRI or RoN), which can be set to necessity (necessity=TRUE) or sufficiency (skip the necessity option or set necessity=FALSE). Use "1- "for negating sets, for example 1-**mydata$myy**. This plot also indicates you Haesebrouck's consistency value, in addition to the other parameters of fit (see Haesebrouck 2015).

You can also integrate labels for the x- and y-axis and for the entire plot:

c 125
xy.plot(**mydata$myx**, **mydata$myy**, ylab = "**Label of my Y**", xlab = "**Label of my X**", main = "**Label for the entire plot**", case.lab = TRUE, labs = rownames(**mydata**))

1. *pimplot (package: SetMethods)*

You can directly plot a solution term that was calculated with the minimize() function and stored as an object, e.g. named as "psOUTCOME" (see section 7) using the pimplot() function for XYplots

c 126
pimplot(data=**mydata**, results=**psOUTCOME,** outcome = "**OUTCOME**", neg.out = **TRUE**, sol=**1,** case_labels **= TRUE,** all_labels **= FALSE**)

The option sol= can be used to specify which model you want to plot in case of model ambiguities[7]. In the example, we used the first solution (sol=1). When performing pimplots (or radar charts, see below) for the negated outcome you should use the results of the minimize() function from the sufficiency analysis of the negated outcome in the

[7] For more complicated structures of model ambiguity, the intermediate solution can also be specified by using a character string of the form "c1p3i2" where c = conservative solution, p = parsimonious solution and i = intermediate solution.

argument results, *together* with option neg.out set to TRUE. Changing the name in the argument outcome or using a tilde is not necessary. Option case_labels in pimplot allows you to display the labels of the relevant cases (left hand-side for sufficiency and upper quadrants for necessity, while option all_labels allows you to display labels for all the cases in your dataset.

This is very useful as a diagnostic tool. It will give you an XY plot of each solution term, and of the whole solution, including parameters of fit (consistency, coverage, PRI, Haesebrouck's (2015) consistency) and case labels. In order to display the different plots, move the cursor again to the command and hit enter again. You can also use pimplot() to plot truth table rows, see *Box **7***, and to plot compound necessary conditions, see section 6.2.

You can also use the and the QCAradar() function for obtaining radar charts:

c 127
```
QCAradar(results=psOUTCOME, outcome = "OUTCOME", fit=TRUE, sol=1)
```

The options for QCAradar() work like those for pimplot().

It is useful to know that you can combine several plots in one graph. Just specify the rules before typing the command for the first plot. For example, here we specify that we want to have 3 rows of plots, each row displaying 2 plots:

c 128
```
par(mfrow=c(3, 2))
```

And then simply list the commands for your 6 plots below that. Note that R will keep applying this rule until you undo it. If you want to return to a single plot, just specify par(mfrow=c(**1**, **1**)) before running the command for that plot.

Box 6: More options for making XY-plots

1. *plot (package: graphics)*

The plot command is the basic.

c 129
```
plot(mydata$myx, mydata$myy, pch=18, col="black")
```

You can label the plot (main) and the axes (xlab and ylab):

c 130

```
plot(mydata$myx, mydata$myy, pch=18, col="black",
    main=' Title of my plot ',
    xlab=' Label of my X ',
    ylab=' Label of my Y ')
```

You can add a number of features. For example, using xlim and ylim, you can define that the x-axis and/or the y-axis has a certain range (by default, the axis uses the sample range). Furthermore, with the abline option you can add horizontal (h) and vertical (v) lines to the xy-plot, which can be full or dotted. In the example below, we define a range for the x-axis from 0 to 3, and a range for the y-axis from 10-150. We add a horizontal, full line at where Y equals 80; and a vertical, dotted line where X equals 2.75 (for another example see section 4.2.1).

c 131

```
plot(mydata$myx, mydata$myy, pch=18,  xlim=c(0, 3), ylim=c(10, 150), col="black",
    main=' Title of my plot ',
    xlab=' Label of my X ',
    ylab=' Label of my Y ')
abline(h=80, col="black")
abline(v= 2.75, col="black", lty="dotted")
```

1. *XYplot (package: QCA)*

The XYplot command produces a plot that looks like those produced with xy.plot, but that can jitter the points. For sufficiency, set relation = "suf"; for necessity, set relation = "nec". Both XYplot() and xy.plot() allow the user to specify other graphical parameters in the basic plot() function. These commands only work if all variables in the dataset used have values between 0 and 1.

c 132

```
XYplot(myx, myy, mydata, relation = "nec", mguides = TRUE, jitter = TRUE, xlab =
"label of my x", ylab = "label of my Y", clabels =rownames(mydata))
```

The great thing about R is that you can directly export the graphs, good-looking and ready-made. Above the graph (which is displayed in the lower right quadrant of the R Studio interface), click the option *export -> save as image…* and then you can specify the size of the graph as well as its format (JPEG, TIFF, etc.). The disadvantage of all the ways to produce xy-plots with R is that if one data point describes several cases, the case labels will overlap and hence become illegible. In this case, use the separate excel template provided in the course material.

6 Analysis of necessity

References:

Goertz (2006), Ragin (1987, 2000, 2006), Schneider and Wagemann (2012: 69-76), Thiem (2016), Thiem and Dusa (2012: 32-38, 62-68, 2013b: 90-91).

6.1 Testing for single necessary conditions

If you want to "deductively" test the necessity for single necessary conditions or theoretically defined disjunctions representing higher-order constructs, the pof command (package: QCA) can be used if the option relation = "nec" is set (if it is set to "suf", then the command tests for the sufficiency of the listed conditions). If you want to test for several conditions, create the list the conditions for which you want to test first. For negating conditions or the outcome, use either a tilde or 1-. This command gives you the consistency, coverage and relevance (RoN) of necessity for each listed condition.[8] In the example below, we test the necessity of conditions 1, 2, and 3 for the negated outcome.

```
c 133
conds <- subset(mydata, select = c("COND1", "COND2", "COND3"))
pof(conds, ~OUTCOME, mydata, relation = "nec")
```

You can also use this command for testing the necessity of only one condition:

```
c 134
pof(COND, ~OUTCOME, mydata, relation = "nec")
```

If you want to test the necessity of the negated conditions for this outcome, you can simply subtract the conditions from 1. Here we check the necessity of the negated conditions for the positive outcome:

```
c 135
pof(1-COND, OUTCOME, mydata, relation = "nec")
```

You can also use pof() for complex condition sets. You can use either lowercase notation or tildes to negate conditions. For example, we want to know whether the condition "COND1 + cond2*COND3" is necessary (<=) for the outcome:

[8] Similarly, you can use the QCAfit() command (package: SetMethods), which equally works for both necessity and sufficiency, check ?QCAfit and section 7.1.

```
c 136
pof("COND1 + cond2*COND3 <= OUTCOME", data = mydata)
```

By using =>, you can test the same for sufficiency.

You can also make an XY plot that integrates the parameters of fit. This can be done for necessity (necessity=**TRUE**), but also for sufficiency (necessity=**FALSE**). This has the advantage that you can make a visual diagnostic of contradictory cases (in the upper left quadrant) and trivialness.

```
c 137
xy.plot(mydata$COND, mydata$OUTCOME, necessity=TRUE)
```

You can integrate case labels (see also section 5 on XY-plots):

```
c 138
xy.plot(mydata$COND, mydata$OUTCOME, case.lab = TRUE, labs =
rownames(mydata), necessity=TRUE)
```

6.2 SuperSubset procedure: Finding all superset of the outcome

The superSubset() command (package: QCA) "inductively" identifies all supersets of the outcome, both single conditions, conjunctions and disjunctions (unions) of sets. So using this command, you can basically skip the steps described in 6.1. The option incl.cut serves to specify a minimal consistency threshold that the conditions need to pass (for fuzzy sets, usually 0.9; typically 1.0 for crisp sets). Using cov.cut, you can also specify a coverage cutoff (here: 0.6), below which the necessary conditions and are deemed trivial and will not appear in the list. Use a tilde ~ to negate the outcome, e.g. outcome = "**~OUTCOME**".

```
c 139
superSubset(mydata, outcome = "OUTCOME",
        conditions = "COND1, COND2, COND3",
        incl.cut = 0.9, cov.cut = 0.6)
```

This command will give you a lot of supersets, but to decide whether they can be deemed necessary, you will still have to 1) check for deviant cases consistency in kind (plot the result), 2) check for empirical relevance / trivialness (Goertz 2006), and 3) identify whether the superset makes theoretical sense as a necessary condition, that is, whether the sets combined with the logical OR represent some higher-order concept (see Schneider and Wagemann 2012). Note also that the different supersets produced by this command are alternatives to each other: for

example, if A*B is necessary, in the output you will find A*B, but also A, and also B. In summary, this command is useful because it gives you all potential necessary conditions in one go; but do not use it for mindless data-mining.

If you store the results of superSubset() as an object (here: nec), you can plot all compound necessary conditions by using pimplot():

```
c 140
nec <- superSubset(mydata, outcome = "OUTCOME",
        conditions = "COND1, COND2, COND3",
        incl.cut = 0.9, cov.cut = 0.6)

c 141
pimplot(data=mydata, results=nec, outcome= "OUTCOME", necessity=TRUE,
case_labels=TRUE, all_labels=FALSE)
```

Use the backward arrow above the plot window to view all plots. The latter also indicate all parameters of fit (including RoN) and the cases with membership > 0.5 in the outcome are labelled.

Alternatively, you can also use command c 114 for calculating membership in the compound condition, and command c 138 for the plot.

7 Analysis of sufficiency

References:

Baumgartner (2015), Baumgartner and Thiem (2015), Ragin (1987, 2000, 2008a, 2009), Rihoux and Ragin (2009), Schneider and Wagemann (2012: 91-220, 2013, 2015, Thiem and Dusa (2012: 38-49, 68-80, 2013a: 513-519, 2013b: 91-95).

7.1 Testing for single sufficient conditions

While the QCA technique *always* uses the truth table procedure for the analysis of sufficiency, in principle the sufficiency (consistency, coverage and PRI) of single conditions can also be ("deductively") tested using the pof() command of the QCA package (just specify relation = "suf"), and visualized using, for instance, the xy.plot() command (again, specify necessity=FALSE) (see section 6.1). Alternatively, you can use QCAfit() of the SetMethods

package (which additionally gives you the Haesebrouck consistency and also works for necessity, specify necessity=TRUE). You can negate the outcome by specifying neg.out=TRUE. If you don't want to label your condition, skip cond.lab= "**COND**".

```
c 142
QCAfit(mydata$COND, mydata$OUTCOME, cond.lab= "COND", necessity=FALSE, neg.out=FALSE)
```

You can also do this for several conditions at once. Just build an object (here: conds) first that contains your conditions.

```
c 143
conds <- subset(mydata, select = c("COND1", "COND2", "COND3"))
QCAfit(conds, mydata$OUTCOME, cond.lab= c("COND1", "COND2", "COND3"), necessity=FALSE, neg.out=FALSE)
```

Alternatively, you can select the conditions from your dataset using square brackets.

```
c 144QCAfit(mydata[,1:3], mydata$OUTCOME, cond.lab= c("COND1", "COND2", "COND3"), necessity=FALSE, neg.out=FALSE)
```

7.2 Building the truth table and logical minimization

First you want to figure out where to set the raw consistency threshold. For this you can produce a basic truth table, with raw consistencies (sorted by descending) and indicating individual cases (package: QCA).

```
c 145
ttOUTCOME <- truthTable(data=mydata, outcome = "OUTCOME",
                        conditions = "COND1, COND2, COND3",
                        incl.cut=1.00, sort.by="incl, n", complete=FALSE, show.cases=TRUE)
ttOUTCOME
```

If you want to analyze the negated outcome, simply set a tilde before the outcome, e.g., "~**OUTCOME**".

The truth table command provides several options. We set the raw consistency threshold using incl.cut. If you skip this option, then all outcomes are coded 0 except for rows with consistency 1. n.cut is used to specify a frequency threshold (by default: 1). The sort.by option can be set such that the rows are ordered by raw consistency ("incl"), or by the N ("n"), or by both (as done here). With decreasing = FALSE, the truth table rows would be ordered in ascending

(instead of descending) order, e.g., by ascending raw consistencies. With the show.cases option, we can choose whether we want to indicate the single cases contained in a truth table row (TRUE) or not (FALSE). If the complete option is set to TRUE, then all logical remainders are also displayed; if FALSE, then only empirically observed truth table rows are displayed. The command below produces a truth table for the negated outcome, with raw consistency threshold 0.828, frequency threshold 1, sorted by descending raw consistency (and, raw consistency being equal, by N), showing only empirically observed rows, and showing individual cases.

```
c 146
ttoutcome <- truthTable(mydata, outcome="~OUTCOME",
        conditions = "COND1, COND2, COND3",
        incl.cut=0.828, n.cut=1, sort.by="incl, n", decreasing=TRUE,
        complete=FALSE, show.cases=TRUE)
ttoutcome
```

Note that we used lowercase letters here for labelling the truth table object because it analyzes the negated outcome. You will know that ttOUTCOME is the truth table for the positive outcome, and ttoutcome, for the negated outcome.

The minimize() command (package: QCA) performs logical minimization of the truth table (i.e., the object we created with the truth table command, below: ttOUTCOME). First we calculate the **conservative solution** and obtain all details (consistency, raw and unique coverage). We also want to display the cases (show.cases) contained in the prime implicants (optional). The use.tilde option can be set to FALSE if you prefer using uppercase and lowercase notation for sets and their negation; and to TRUE if your prefer to denote negated sets with a tilde. It can be skipped and then upper-/lowercase notation is used.

```
c 147
csOUTCOME <- minimize(ttOUTCOME, details=TRUE, show.cases=TRUE,
row.dom=TRUE, all.sol=FALSE, use.tilde=FALSE)
csOUTCOME
```

To calculate the **parsimonious solution**, we tell the software to include logical remainders (those with outcome "?") representing simplifying assumptions.

```
c 148
psOUTCOME <- minimize(ttOUTCOME, include="?", details=TRUE, show.cases=TRUE,
row.dom=TRUE, all.sol=FALSE)
psOUTCOME
```

Box 7: Useful tools for truth table analysis

Plotting truth table rows

Once you see the truth table, in order to find the appropriate raw consistency threshold, you may want to plot different truth table rows. You can plot truth table rows easily using pimplot(), including the names of cases with membership > 0.5 in the row and all parameters of fit (consistency, coverage, PRI, Haesebrouck's consistency). However, note that this requires you to already have calculated a solution which you stored as an object (here: psOUTCOME; you could use any raw consistency to begin with, just to get such a provisional solution object to work with). Here we plot the truth table rows number 1 and 26. Use the backward arrow above the plot window in order to get inspect all the different plots:

```
c 149
pimplot(data=mydata, results=psOUTCOME, ttrows=c("1", "26"), outcome=
"OUTCOME")
```

You can also simply plot all truth table rows above a certain raw consistency level (here: 0.8) at once. The resulting plots have the row number as label of the X axis:

```
c 150
pimplot(data=mydata, results=psOUTCOME, incl.tt=0.8, outcome= "OUTCOME")
```

If, for some reason, you do not wish to calculate a solution object yet, you can also build the sets of different truth table rows (see section 4.3) and plot these rows (see section 5). For your convenience, here is a repetition on how you can do this. Here we plot the truth table row number 1, which is COND1*cond2*COND3:

```
c 151
row1 <- compute("COND1*cond2*COND3", data=mydata)
xy.plot(row1, mydata$OUTCOME, case.lab = TRUE, labs = rownames(mydata))
```

Assessing the consequences of setting a frequency threshold

If you set a frequency threshold of more than 1 and you want to see which rows are now treated as logical remainders by imposing such a threshold, type

```
c 152
ttoutcome$excluded
```

Exporting the truth table

You can save the truth table as a csv file. This will make it very easy to then copy-paste the table into your word file (e.g. the online appendix).

```
c 153
write.csv(ttOUTCOME$tt, "mytt.csv")
```

Box 8: Default settings for logical minimization

The row.dom option for logical minimization, if set to TRUE, is used to further eliminate redundant prime implicants when solving the PI chart, applying the principle of row dominance: if a prime implicant X covers the same configurations as another prime implicant Y and in the same time covers other configurations which Y does not cover, then Y is redundant and eliminated. By setting all.sol=**TRUE**, you can derive all possible solutions, irrespective of the number of prime implicants.

To obtain a subset of the solution space, set row.dom=**TRUE** and all.sol=**FALSE**. This is the default setting that the QCA package implements if you do not specify these two options (as done here in this manual). Conversely, for revealing the full extent of model ambiguity, set row.dom=**FALSE** and all.sol=**TRUE** (see Baumgartner 2015 and Baumgartner and Thiem 2015). The usage of all.sol = TRUE does not represent the opinion of the QCA package author, where the default option is FALSE.

By presenting the templates using the default options, we do not intend to make a recommendation. It is good to be aware of the fact that there are often many possible solutions, and that you have several possibilities how to deal with this.

7.3 Standard Analysis: Specifying directional expectations

To derive an **intermediate solution** using Standard Analysis (Ragin 2008), we may want to specify directional expectations. Just like with the parsimonious solution, we tell the software to include simplifying assumptions; but then we specify the directional expectations (dir.exp) for each condition *in the same order as the conditions were listed when creating the truth table* (see section 7.2). In the example, we assume that condition 1 contributes to the outcome when present (1); condition 2 contributes to the outcome when absent (0); and we have no directional expectation ("-") for condition 3.

```
c 154
isOUTCOME <- minimize(ttOUTCOME, include = "?", details=TRUE,
show.cases=TRUE,  row.dom=TRUE, all.sol=FALSE, dir.exp = "1, 0, -")
isOUTCOME
```

7.4 Enhanced Standard Analysis: Excluding truth table rows

When deriving an intermediate solution and performing an Enhanced Standard Analysis, we may or may not specify directional expectations; but in addition, we exclude those truth table

rows from the analysis of sufficiency that a) display a negated necessary condition for the same outcome, b) display a sufficient condition for the negated outcome, or c) are logically implausible (the "pregnant man"). You have several possibilities to do this. In the running text, we discuss two of them; see *Box 10* for more options.

7.4.1 Excluding truth table rows before logical minimization

We can build a new truth table where we simply tell the software to code those rows with outcome 0 that display a certain configuration of conditions – we can do this only for logical remainders, or for all truth table rows, whether they are empirically observed or logical remainders. This possibility, which we apply in class, is an easy and transparent way if you think that neither your observations nor your simplifying assumptions should contradict prior findings and / or logic, and you want to see what you did to the truth table before turning to logical minimization.

In a first step, we build the truth table (here: for the negated outcome and for 5 conditions) that also displays the logical remainders (complete=TRUE):

```
c 155
ettoutcome <- truthTable(mydata, outcome="~OUTCOME",
        conditions = "COND1, COND2, COND3, COND4, COND5",
        incl.cut=0.8, n.cut=1, sort.by="incl, n", decreasing=TRUE,
        complete=TRUE, show.cases=TRUE)

ettoutcome
```

Now we tell the software which truth table rows we want to exclude. In the example, we want to exclude the configurations "cond1*COND2 + COND1*COND2*cond3" that were sufficient for "OUTCOME" and hence, constitute an untenable assumption.

Furthermore, we found that "COND4 + COND5" is necessary for "outcome". Hence, we want to exclude the negation of this necessary condition ("cond4*cond5") from the sufficient conditions for outcome.

In a next step, you identify all truth table rows that display these configurations (package: QCA). You can use either lowercase notation or the tilde sign to negate the conditions (remember: label your conditions with uppercase letters to indicate their presence, already when calibrating). You have to use the Boolean AND (*) for conjunctions. You can choose if you want to do this only for logical remainders (remainders = **TRUE)** or for all truth table rows, also empirically observed ones **(**remainders = **FALSE).**

```
c 156
rows <- findRows("cond1*COND2 + COND1*COND2*cond3 + cond4*cond5",
ettoutcome, remainders = FALSE)
rows
```

This gives you the numbers of all the truth table rows that display one of these combinations.

Box 9: Identifying untenable assumptions contradicting the statement of necessity

If you have identified a necessary condition, and you don't want to employ deMorgan's law yourself when negating it, R's deMorgan() command can do that for you (package: QCA). You can use either the tilde or lowercase notation for negated conditions, so long as you are consistent. Equally, you can either use the Boolean AND (*) for conjunctions, or you can skip the *, but be consistent. Obviously, you do not have to do this if you did not find a necessary condition.

```
c 157
nec <- deMorgan("COND4 + COND5")
nec
```

This will give you the untenable assumption, cond4*cond5.

Note that, if you identified several (single or compound) necessary conditions, in order to avoid that they "disappear" from the sufficient solution term, you have to assume that they are connected with the logical AND – although with less-than-perfect subset relations, their conjunction may empirically no longer pass the consistency threshold for necessity. So, if you found that A ← Y and that B+C ← Y, type:

```
c 158
nec <- deMorgan("A * (B + C)")
nec
```

After that, you can first code these truth table rows with outcome 0:

```
c 159
ettoutcome$tt[as.character(rows), "OUT"] <- 0
ettoutcome
```

In the new truth table, these truth table rows should now have the outcome 0.

Now, using the new, coded truth table, we can use the minimize command to calculate the **enhanced conservative solution**:

```
c 160
ecsoutcome <- minimize(ettoutcome, details=TRUE, show.cases=TRUE, row.dom=TRUE,
all.sol=FALSE)
ecsoutcome
```

The **enhanced parsimonious solution** (which, in fact, is one of many possible intermediate solutions):

```
c 161
epsoutcome <- minimize(ettoutcome, include="?", details=TRUE, show.cases=TRUE,
row.dom=TRUE, all.sol=FALSE)
epsoutcome
```

and the **enhanced intermediate solution**:

```
c 162
eisoutcome <- minimize(ettoutcome, include = "?", details=TRUE, show.cases=TRUE,
row.dom=TRUE, all.sol=FALSE, dir.exp = "0, 1, -")
eisoutcome
```

7.4.2 Differentiating between empirically observed rows and logical remainders

Sometimes you may want to exclude only empirically observed rows, or only logical remainders; or you do not wish to apply the same rules to them. For example, you want to 1) exclude remainders that contain a path of the solution for OUTCOME "cond1*COND2 + COND1*COND2*cond3"; 2) exclude remainders that contradict the necessary condition for outcome, "COND4"; and 3) exclude truth table rows number 5 and 6, which contain cases that you do not want to include into the analysis.

First, you will build the truth table (in the example, again, for the negated outcome):

```
c 163
ttoutcome <- truthTable(mydata, outcome="~OUTCOME",
        conditions = "COND1, COND2, COND3, COND4, COND5",
        incl.cut=0.8, n.cut=1, sort.by="incl, n", decreasing=TRUE,
        complete=TRUE, show.cases=TRUE)
ttoutcome
```

Then, you can use the esa() function to specify these coding rules (package: SetMethods). Below we use the above truth table (oldtt=ttoutcome), and code it into a new enhanced truth

table ett**outcome**.

The option "nec_cond" allows you to simply insert the necessary condition; R automatically negates it for you without further ado. The option imposs_LR allows you to insert your untenable assumptions using a logical expression (note: these can be impossible remainders or otherwise implausible assumptions – despite its label, all that this option does is to code these logical remainders with outcome 0). For example, you can use this argument for excluding remainder rows that were included in the minimization procedure for the opposite outcome – simply enter the Boolean expression for the solution for the opposite outcome. For both options, use the tilde sign for negating conditions (lowercase letters do not work), and use the * sign for the logical AND. These two options are only applied to logical remainders, not to empirically observed rows. In our example, we specify as untenable assumptions all remainders that are members of the set ~COND1*COND2 + COND1*COND2*~COND3**.**

Finally, the option contrad_rows allows you to exclude specific empirically observed rows or specific logical remainders, in our example: rows 5 and 6 (note: this options works for ALL rows in the truth table). You will have to identify their row number first, and insert the respective truth table row numbers - you cannot apply a "rule" here. You can exclude with contrad_rows, for example, empirically observed rows that have a very low PRI which indicates that they might be included in the minimization of the negated outcome. Additionally, you can also use this for excluding rows that, despite having a high consistency threshold, have too many deviant cases consistency in kind and which are therefore deemed insufficient.

```
c 164
ett outcome <- esa(oldtt = ttoutcome, nec_cond = "COND4", imposs_LR =
"~COND1*COND2 + COND1*COND2*~COND3", contrad_rows=c("5", "6"))
ett outcome
```

If you have more than one necessary condition, you can enter them like this, for example: nec.cond= c("**COND1**", "**COND2**"). Negated necessary conditions are combined with the logical OR; that is, it is assumed that the necessary conditions are combined with the logical AND. (Note however that with less-than-perfect subset relations, sometimes the conjunction of two individually necessary conditions may actually no longer pass the consistency threshold for necessity). Equally, you can enter compound necessary conditions. For example, if your necessary condition is COND1 + COND2, enter: nec.cond= "**COND1** + **COND2**".

Based on this new truth table, you can then perform logical minimization, see commands c 160,

c 161 and c 162.

Box 10: Alternative options for flexibly coding and omitting truth table rows

Omitting truth table rows during logical minimization

You can also skip the step of explicitly coding truth table rows with outcome 0 first, and directly omit these rows when performing logical minimization with minimize() with the QCA package. This possibility is faster, but it has the disadvantage that you cannot check the truth table visually first. Build the truth table:

```
c 165
ttoutcome <- truthTable(mydata, outcome="~OUTCOME",
        conditions = "COND1, COND2, COND3, COND4, COND5",
        incl.cut=0.8, n.cut=1, sort.by="incl, n", decreasing=TRUE,
        complete=TRUE, show.cases=TRUE)

ttoutcome
```

Identify those rows that you wish to omit (see example above):

```
c 166
rows <- findRows("cond1*COND2 + COND1*COND2*cond3 + cond4*cond5",
ttoutcome, remainders = FALSE)
rows
```

Again, this can be done for all truth table rows (remainders = **FALSE**), or only for logical remainders (remainders = **TRUE**).

Omit these rows when calculating the **enhanced conservative solution**:

```
c 167
ecsoutcome <- minimize(ttoutcome, details=TRUE, show.cases=TRUE, row.dom=TRUE,
all.sol=FALSE, omit = rows)
```

You can also use omit for calculating the **enhanced intermediate** and the **enhanced parsimonious solution**:

```
c 168
eisoutcome <- minimize(ttoutcome, include = "?", details=TRUE, show.cases=TRUE,
row.dom=TRUE, all.sol=FALSE, dir.exp = "0, 1, -", omit = rows)
eisoutcome


c 169
epsoutcome <- minimize(ttoutcome, include="?", details=TRUE, show.cases=TRUE,
row.dom=TRUE , all.sol=FALSE, omit = rows)
epsoutcome
```

Excluding or omitting specific identified rows

Sometimes you may not want to apply a rule for excluding truth table rows, but you want to exclude specific rows from logical minimization. Build a truth table first:

```
c 170
ttoutcome <- truthTable(mydata, outcome="~OUTCOME",
        conditions = "COND1, COND2, COND3, COND4, COND5",
        incl.cut=0.8, n.cut=1, sort.by="incl, n", decreasing=TRUE,
        complete=TRUE, show.cases=TRUE)
ttoutcome
```

By checking the truth table, you can identify those rows you wish to exclude – let us say these are rows 8, 13 and 27. Now you have two options.
You can code these rows with outcome 0 in the truth table – which enables you to visually check the truth table first:

```
c 171

ettoutcome <- ttoutcome
ettoutcome$tt[c('8','13','27'), "OUT"] <- 0
ettoutcome
```

And then you can calculate the different solution terms based on this new truth table, without using the omit option; see commands c 160, c 161 and c 162.

If you want to skip this step, you can also directly calculate a new solution that omits these rows from the truth table during logical minimization. Here we omit rows 8, 13 and 27 and calculate the **enhanced conservative solution**:

```
c 172
ecsoutcome <- minimize(ttoutcome, details=TRUE, show.cases=TRUE, row.dom=TRUE,
all.sol=FALSE, omit = c(8, 13, 27))
```

You can do this for the **enhanced intermediate** and the **enhanced parsimonious solution**:

```
c 173
eisoutcome <- minimize(ttoutcome, include = "?", details=TRUE, show.cases=TRUE,
row.dom=TRUE, all.sol=FALSE, dir.exp = "0, 1, -", omit = c(8, 13, 27))
eisoutcome

c 174
epsoutcome <- minimize(ttoutcome, include="?", details=TRUE, show.cases=TRUE,
row.dom=TRUE , all.sol=FALSE, omit = c(8, 13, 27))
epsoutcome
```

7.5 Identifying simplifying assumptions and easy counterfactuals

With R we can easily identify the simplifying assumptions, that is, all remainders that were assumed to contribute to the outcome for the parsimonious solution that we calculated above.

```
c 175
SAOUTCOME <- psOUTCOME$SA
SAOUTCOME
```

This will give you a table with all remainders that were assumed to be sufficient for the outcome (for each parsimonious model, if there was ambiguity). If you had several models due to ambiguity, then you can specify for which model you opt. You should choose the model that also was the basis for your intermediate solution (the parsimonious model that was used will be shown in upper part of the output when you calculate the intermediate solution). For example, if your intermediate solution is based on the 8th parsimonious solution, then specify

```
c 176
psOUTCOME$SA$M08.
```

Now, not all of these simplifying assumptions are also necessarily easy counterfactuals, used for calculating your intermediate solution (here: eisOUTCOME). The easy counterfactuals can be obtained as follows:

```
c 177
ECOUTCOME <- eisOUTCOME$i.sol$C1P1$EC
ECOUTCOME
```

Here we calculated the easy counterfactuals for the intermediate model 1. If we had several intermediate solutions, we could also do so, e.g., for model 8:

```
c 178
ECOUTCOME <- eisOUTCOME$i.sol$C1P8$EC
```

Having done this, we can check whether all simplifying assumptions are also easy counterfactuals:

```
c 179
identical(rownames(ECOUTCOME), rownames(SAOUTCOME))
```

If this is not the case (FALSE), we can identify which counterfactuals are both simplifying and easy. Under ESA, the result of this intersection should always be identical to the easy counterfactuals as these are a subset of the simplifying assumptions.

```
c 180
intersect(rownames(EC**OUTCOME**), rownames(SA**OUTCOME**))
```

This only works if you have chosen the simplifying assumptions and easy counterfactuals of a specific model.

With the same command, we could also identify whether some logical remainders were used both for the calculation of the solution for the positive outcome and for the solution for its negation (we would first have to calculate the simplifying assumptions SA**outcome** and easy counterfactuals EC**outcome** for the negated outcome, of course). We could see whether we made such untenable assumptions for the parsimonious solution, and which rows are concerned, by intersecting the simplifying assumptions for the positive outcome with those for the negated outcome:

```
c 181
intersect(rownames(SA**OUTCOME**), rownames(SA**outcome**))
```

If this reveals such contradictory assumptions, we could then check whether we successfully eliminated this problem for the enhanced intermediate solution, by intersecting the easy counterfactuals for the positive outcome with those for the negated outcome:

```
c 182
intersect(rownames(EC**OUTCOME**), rownames(EC**outcome**))
```

The intersection should be empty.

Box 11: Useful tools for interpreting QCA outputs

A tool is the sop() function (package: QCA). It will help you to minimize any logical expression into its simplest equivalent logical expression:

```
c 183
sop(expression="**Ac + Abc + bc + abC + ab**", snames = "**A, C, B**")
```

The function factorize() will give you all the possible ways in which elements of the results can be factored out (package: QCA):

```
c 184
factorize ("**Ac + Abc + bc + abC + ab**", snames = "**A, C, B**")
```

snames can be useful here to ensure that a certain order of the conditions is preserved; without specifiying snames, the conditions are listed alphabetically.

You can also use the factorize function directly on a solution term, for example:

```
c 185
csOUTCOME <- minimize(ttOUTCOME, details=TRUE, show.cases=TRUE,
row.dom=TRUE, all.sol=FALSE, use.tilde=FALSE)
factorize(csOUTCOME)
```

Attention: apparently the factorize() function is computationally intensive. Sometimes R gets "hung up" when performing it, and then the subsequent codes appear to not be working. Always wait until R finishes with factorize(). If it doesn't finish, delete the command from your code / skip it to continue.

Box 12: A hands-on template code for Enhanced Standard Analysis

```
c 186

####Analysis of necessity #######
# Analyze single necessary conditions
conds <- subset(mydata, select = c("COND1", "COND2", "COND3"))
pof(conds, ~OUTCOME, mydata, relation = "nec")

# Alternatively: Identify all supersets of the outcome
superSubset(mydata, outcome = "OUTCOME",
        conditions = "COND1, COND2, COND3",
        incl.cut = 0.9, cov.cut = 0.6)

# Calculate the cases' membership in the compound necessary condition, if applicable
NEC1 <- compute("COND1 + cond2", data=mydata)
NEC2 <- compute("COND2 + cond3", data=mydata)

# Plot the results
xy.plot(mydata$NEC1, mydata$OUTCOME, case.lab = TRUE, labs =
rownames(mydata), necessity=TRUE)


####Analysis of sufficiency#######
# Build the truth table and set consistency threshold
ttOUTCOME <- truthTable(mydata, outcome="OUTCOME",
        conditions = "COND1, COND2, COND3, COND4, COND5",
        incl.cut=0.828, n.cut=1, sort.by="incl, n", decreasing=TRUE,
        complete=TRUE, show.cases=TRUE)
ttOUTCOME

# Standard Analysis
# Parsimonious solution
```

```
psOUTCOME <- minimize(ttOUTCOME, include="?", details=TRUE,
show.cases=TRUE,  row.dom=TRUE, all.sol=FALSE)
psOUTCOME

# Intermediate solution (Standard Analysis)
isOUTCOME <- minimize(ttOUTCOME, include = "?", details=TRUE,
show.cases=TRUE,  row.dom=TRUE, all.sol=FALSE, dir.exp = "0, 1, -")
isOUTCOME

#Enhancd Standard Analysis
# Excludee remainder rows contradicting necessity (assuming "COND4 + COND5" is
necessary for the OUTCOME) and the contradictory rows. In our example, these are rows 5
and 6.

ettOUTCOME <-  esa(ttOUTCOME, nec_cond = c("COND4 + COND5"),
contrad_rows=c("5", "6"))

# Enhanced conservative solution
ecsOUTCOME <- minimize(ettOUTCOME, details=TRUE, show.cases=TRUE,
row.dom=TRUE, all.sol=FALSE)
ecsOUTCOME

# Enhanced intermediate solution
eisOUTCOME <- minimize(ettOUTCOME, include = "?", details=TRUE,
show.cases=TRUE,  row.dom=TRUE, all.sol=FALSE, dir.exp = "0, 1, -")
eisOUTCOME

# Enhanced parsimonious solution
epsOUTCOME <- minimize(ettOUTCOME, include="?", details=TRUE,
show.cases=TRUE,  row.dom=TRUE, all.sol=FALSE)
epsOUTCOME

# Identify simplifying assumptions
SAOUTCOME <- psOUTCOME$SA
SAOUTCOME

# Identify easy counterfactuals
ECOUTCOME <- isOUTCOME$i.sol$C1P1$EC
ECOUTCOME

# Analysis of negated outcome
# Build the truth table and set consistency threshold

ttoutcome <- truthTable(mydata, outcome="~OUTCOME",
        conditions = "COND1, COND2, COND3",
        incl.cut=0.9, n.cut=1, sort.by="incl, n", decreasing=TRUE,
        complete=TRUE, show.cases=TRUE)
ttoutcome
```

```
# Code remainder rows contradicting necessity (assuming "COND2" is necessary for
outcom) and the contradictory rows to 0. In our example, these are rows 3 and 6. In
addition, exclude incoherent counterfactuals that contradict the sufficient solution for
OUTCOME (in this example: COND4*COND1 + COND5* ~COND2*COND3), using the
imposs_LR option.

ettoutcome <-  esa(ttoutcome, nec_cond = c("COND2"), imposs_LR = " COND4*COND1
+ COND5* ~COND2*COND3", contrad_rows=c("3", "6"))

# Enhanced conservative solution
ecsoutcome <- minimize(ettoutcome, details=TRUE, show.cases=TRUE,
row.dom=TRUE, all.sol=FALSE)
ecsoutcome

# Enhanced intermediate solution
eisoutcome <- minimize(ettoutcome, include = "?", details=TRUE, show.cases=TRUE,
row.dom=TRUE, all.sol=FALSE, dir.exp = "1, 0, -")
eisoutcome

# Enhanced parsimonious solution
epsoutcome <- minimize(ettoutcome, include="?", details=TRUE, show.cases=TRUE,
row.dom=TRUE, all.sol=FALSE)
epsoutcome

# Identify simplifying assumptions
SAoutcome <- psoutcome$SA
SAoutcome

# Identify easy counterfactuals
ECoutcome <- isoutcome$i.sol$C1P1$EC
ECoutcome

# Intersect easy counterfactuals used for intermediate solution for positive and negative
outcome
CSA <- intersect(rownames(ECOUTCOME), rownames(ECoutcome))
```

8 Advanced stuff

8.1 Ex post skewness diagnostics

Schneider and Wagemann (2012: 244-250) outline how the clustering of cases in particular

intersecting areas of the two diagonals of the XY plot can lead to flawed causal conclusions (see also Cooper and Glaesser 2011). To check if this is a problem in your analysis, you can easily produce such plots yourself, indicating the percentage of case that cluster in each intersection. Just use the command below – you can e.g. copy-paste the template into an R script and search and replace the items marked bold with the features of your own analysis.

First, you will have to calculate the cases' membership in the particular condition set (configuration, solution term, path of the solution term… here: "MYSET") for which you want to perform the test, see section 4.3. Then you calculate the percentage of cases situated in each of the four intersecting areas (myN denotes the number of cases).

c 187

```
A1 <- round(sum(as.numeric((MYSET <= mydata$OUTCOME) & (MYSET >= (1-
mydata$OUTCOME))))/(myN/100), digits = 1)
A2 <- round(sum(as.numeric((MYSET >= mydata$OUTCOME)&(MYSET >= (1-
mydata$OUTCOME))))/( myN /100), digits = 1)
A3 <- round(sum(as.numeric((MYSET >= mydata$OUTCOME)&(MYSET <= (1-
mydata$OUTCOME))))/( myN /100), digits=1)
A4 <- round(sum(as.numeric((MYSET <= mydata$OUTCOME)&( MYSET <= (1-
mydata$OUTCOME))))/( myN /100), digits = 1)
```

Then you produce the plot with the two diagonals and integrate these percentages.

c 188

```
plot(mydata$OUTCOME, MYSET, pch=18,  xlim=c(0, 1), ylim=c(0,1), col="black",
    xaxs="i", yaxs="i",
    main= ' MYSET ',
    xlab= ' condition ',
    ylab=' OUTCOME ')
abline(0,1, col="black")
abline(1, -1, col="black")
text(x=0.5, y=0.8, labels = A1)
text(x=0.8, y=0.5, labels = A2)
text(x=0.5, y=0.2, labels = A3)
text(x=0.2, y=0.5, labels = A4)
```

Note that, if some cases are placed exactly on one of the diagonals, the percentages will add up to more than 100.

8.2 Formal set-theoretic theory evaluation

Below you can find ways how to do several steps of formal theory evaluation with R (see Ragin 1987, Schneider and Wagemann 2012)

8.2.1 Calculating the logical intersections of theory and empirics

To perform the Boolean intersections of the hypotheses, the results, and their negations (Ragin 1987), you can use the commands intersection() and deMorgan() offered by QCA. At the end of this step, you will have obtained the intersection of your hypotheses T with your solution S (T*S), the intersection of the theory with what you did not observe (T*s), the intersection of what you did not expect with what you observed (t*S), and what you neither expected not observed (t*s).

Negate a Boolean expression – for example, we negate our hypotheses T: "~A*B + C*A". To use these objects later combined with the theory.evaluation function, use the tilde for negated conditions and use the Boolean AND (*) for conjunctions. The resulting expression "t" here are those results that we should not observe according to theory:

```
c 189
t <- deMorgan("~A*B + C*A")
t
```

You can also use this command to directly negate the solutions you obtained (see 7.2-7.4). In the example here, the solution has been stored as an object labeled "csOUTCOME":

```
c 190
s <- deMorgan(csOUTCOME)
s
```

In the example, this would give you everything that you did not observe in your results.

You can then intersect different statements (package: QCA). Here you have to insert the full Boolean expression inside the quotation marks – you cannot work with objects such as solution terms. Use the snames option to specify the names of the conditions you use.

```
c 191
ts <- intersection("A*~C + ~B*~C + ~A*~B", "C + A + ~B", snames = "A, B, C")
ts
```

This command can only intersect two expressions and not more at once.

In order to figure out whether and how these intersections reflect your hypotheses, you can use

the factorize function to group the intersections, see command c 184.

8.2.2 Calculating the membership of the cases in the intersections

Schneider and Wagemann (2012: 300-305) show how accounting for coverage can add leverage to these evaluations. Here we show how you can calculate the number and percentage of cases in certain intersections (but this is not the coverage measure – see below). Once you have obtained the relevant intersections, you can *calculate the cases' membership in these intersections* (package: SetMethods). [9]

The theory.evaluation() function (package: SetMethods) offers a fast way for calculating the cases' membership in the four intersections between your theory and your results. The function also returns the names of the cases in these intersections, their percentages, and parameters of fit for the intersections. For running the function you will use your solution as calculated with minimize() and stored in an object (here: csOUTCOME). You will also need your theory written as a Boolean expression. Specify negations with ~ and intersections with * (here: myset1*MYSET2 + myset3*MYSET4*myset5 + MYSET6). For seeing the structure of the theory.evaluation() function more easily, we can store the theory in an object, here: T (note: you will get the same by just entering the Boolean expression in the option theory= in the function). Don't forget to specify the option sol = 1 for identifying which model you are referring to in case of model ambiguities (e.g., specify sol=2 if you want to use model 2).

c 192

```
T <- "~MYSET1*MYSET2 + ~MYSET3*MYSET4*~MYSET5 + MYSET6"
TE <- theory.evaluation(theory= T, empirics = csOUTCOME, outcome = "OUTCOME", sol = 1, print.data=TRUE)
TE
```

If the option print.data is set to TRUE, the function will first return a dataframe containing all the cases' membership in the solution, the theory, and the different intersections between them (when print.data is set to FALSE this part of the output will be skipped). The second part of the output consists of the identification of cases *in each of these intersections, the outcome or the negation of the outcome, together with the percentage of cases in the intersection out of the*

[9] Each of these intersections may consist of several configurations. Some of these may cover logical remainders and hence, not represent empirical evidence. The SetMethods package currently does not allow you to check this. If you need a code for this, contact the author of this manual.

total number of cases. For fuzzy sets, this means that the cases' membership in the set is higher or smaller than 0.5. In other words, the output will give you a list of the covered cases, including information on whether they are most or least likely cases for the posited set relation, whether they have membership above or below 0.5 in the outcome, and their percentage relative to the total number of cases (package: SetMethods). Finally, the last part of the output consists of the parameters of fit (consistency, coverage, PRI, Haesebrouck's consistency) of the different intersections (see Schneider and Wagemann 2012).

Note however that these parameters must be interpreted cautiously, because the cases' membership in these intersections is typically highly skewed. The parameters of fit will not necessarily tell you beyond any doubt if an intersection is populated by deviant cases consistency in kind, for example. Hence, reporting the percentage of cases with membership above or below 0.5 in the outcome may be an attractive or even necessary complement to the parameters of fit.

You can also access just the part of the output that you like by navigating the object created with the theory.evaluation function (here: TE).

c 193

```
#For getting just the names of the cases in the intersections with percentages:

TE$cases

#For getting just the parameters of fit:

TE$fit

# Or, for getting just the dataframe:

TE$data
```

8.3 Post-QCA case selection

The SetMethods package allows for a targeted identification of most typical, most deviant and individually irrelevant cases (see Schneider and Rohlfing 2013) for the analysis of sufficiency. Most conveniently, the SetMethods package also allows you to match cases for targeted post-QCA case analysis (process tracing), see Schneider and Rohlfing (2013). For identifying both single cases and best pairs of matching cases, you just need to use the mmr() function (package SetMethods).

You need to have the results stored in an object, for example, the parsimonious solution psOUTCOME and the intermediate solution isoutcome for the negated outcome. For single case identification, the option match inside the function mmr() must be set to FALSE. The option cases can be modified with the following values for obtaining specific types of single cases. Remember that the sol = option allows you to specify which model you want to choose in case of model ambiguities (e.g., sol=3 would give you model 3). Additionally, when the outcome is negated you need to use the neg.out = TRUE option and input the result of the minimize() analysis for the negation of the outcome.

1 = typical cases

```
c 194
mmr(results = psOUTCOME, outcome = "OUTCOME", neg.out=FALSE, sol=1, match =
FALSE, cases = 1)
```

2 = typical cases for each focal conjunct in a sufficient term. Note that you can specify the term with the option term.

```
c 195
mmr(results = psOUTCOME, outcome = "OUTCOME", neg.out=FALSE, sol=1, match =
FALSE, cases = 2, term = 2)
```

3 = deviant consistency cases

```
c 196
mmr(results = isoutcome, outcome = "OUTCOME", neg.out=TRUE, sol=1, match =
FALSE, cases = 3)
```

4 = deviant coverage

```
c 197
mmr(results = psOUTCOME, outcome = "OUTCOME", neg.out=FALSE, sol=1, match =
FALSE, cases = 4)
```

5 = individually irrelevant cases

```
c 198
mmr(results = psOUTCOME, outcome = "OUTCOME", neg.out=FALSE, sol=1, match =
FALSE, cases = 5)
```

6 = you will get all of the types of cases from above

```
c 199
mmr(results = psOUTCOME, outcome = "OUTCOME", neg.out=FALSE, sol=1, match =
FALSE, cases = 6)
```

For identifying best pairs of matching cases, the option match inside the function mmr() must be set to TRUE. The option cases can again be modified with the following values for obtaining specific pairs of cases. Note that for the functions identifying pairs for focal conjuncts in sufficient terms you can specify the term for which to get the pairs through the option term. In the example below we will get pairs of typical cases for the second sufficient term in the parsimonious solution as term = **2.** Additionally, you can also specify the maximum number of pairs that you want to get through the option max_pairs (the default is set to 5 pairs).

cases = 1: pairs of typical and typical cases for each focal conjunct

```
c 200
mmr(results = psOUTCOME, outcome = "OUTCOME", neg.out=FALSE, sol=1, match =
TRUE, cases = 1, max_pairs = 3, term = 2)
```

Cases = 2: pairs of typical and individually irrelevant cases for each focal conjunct

```
c 201
mmr(results = psOUTCOME, outcome = "OUTCOME", neg.out=FALSE, sol=1, match =
TRUE, cases = 2, max_pairs = 3, term = 2)
```

Cases = 3: pairs of typical and deviant consistency cases

```
c 202
mmr(results = psOUTCOME, outcome = "OUTCOME", neg.out=FALSE, sol=1, match =
TRUE, cases = 3, max_pairs = 3)
```

Cases = 4: pairs of deviant coverage and individually irrelevant cases

```
c 203
mmr(results = psOUTCOME, outcome = "OUTCOME", neg.out=FALSE, sol=1, match =
TRUE, cases = 4, max_pairs = 3)
```

Cases = 5: you will get all of the types of pairs of cases from above

```
c 204
mmr(results = psOUTCOME, outcome = "OUTCOME", neg.out=FALSE, sol=1, match =
TRUE, cases = 5, max_pairs = 3, term = 2)
```

8.4 QCA with clustered data

References:

Garcia-Castro and Ariño (2016), Hino (2009).

The SetMethods package allows you to use QCA on panel data or other clustered data. The theory and formulae underlying these functions are in Garcia-Castro and Ariño (2016). So don't believe anyone who tells you that QCA cannot be used on panel data!

The trick is to first calculate a solution (here: isOUTCOME) on the whole dataset. For example:

```
c 205
isOUTCOME <- minimize(ttOUTCOME, include = "?", details=TRUE,
show.cases=TRUE,  row.dom=TRUE, all.sol=FALSE, dir.exp = "1, 0, -")
isOUTCOME
```

Obviously you can also perform ESA first, see section 7.4.

You can then get the pooled, within, and between consistencies for this solution. You simply have to specify the variable that constitutes your units of analysis (here: COUNTRY) and the variable that constitutes your clustering units (here: YEAR). If you have model ambiguity, you could specify which model you want, for example, model 2 would be sol=2.

```
c 206
cluster(data = mydata, results = isOUTCOME, outcome= "OUTCOME",
unit_id="COUNTRY", cluster_id="YEAR", sol=1, necessity = FALSE)
```

The pooled consistency indicates the overall consistency observed in the sample when time and individual effects are not taken into account. The between consistency is a measure of the cross-sectional consistency for each year t in the panel. The within consistency measures how consistent the set-subset relationship is across time for each particular case in the sample, in other words, the longitudinal consistency of the set-subset connection for each individual in in the panel over time (Garcia-Castro and Ariño 2016).

The function can also perform cluster diagnostics for single necessary conditions or for Boolean expressions by just modifying the results field with the appropriate information. Let's assume we want to check whether condition **myset1** is necessary across the clusters. You can just set option results = "~**MYSET1**" (note that negation is done with a ~) and option necessity = **TRUE**.

```
c 207
cluster(data = mydata, results = "~MYSET1", outcome= "OUTCOME",
unit_id="COUNTRY", cluster_id="YEAR", necessity = TRUE)
```

You can also input Boolean expressions:

c 208

cluster(data = **mydata**, results = "~**MYSET1*MYSET2 + MYSET3**", outcome= "**OUTCOME**", unit_id="**COUNTRY**", cluster_id="**YEAR",** necessity = **FALSE**)

References

Baumgartner, M. 2015. Parsimony and Causality. *Quality & Quantity* 49: 839-856.

Baumgartner, M., & A. Thiem, A. 2015. Model Ambiguities in Configurational Comparative Research. *Sociological Methods & Research*. Advance online publication. DOI: 10.1177/0049124115610351.

Garcia-Castro, R.C. & M.A. Ariño. 2016. A General Approach to Panel Data Set-Theoretic Research. *Journal of Advances in Management Sciences & Information Systems* 2: 63-76.

Cooper, B. & J. Glaesser. 2011. Paradoxes and pitfalls in using fuzzy set QCA: Illustrations from a critical review of a study of educational inequality. *Sociological Research Online* 16(3): 1-18.

Dusa, A. 2018. *QCA with R. A Comprehensive Resource*. Springer International Publishing.

Goertz, G. 2006. Assessing the Trivialness, Relevance, and Relative Importance of Necessary or Sufficient Conditions in Social Science. *Studies in Comparative International Development* 41(2): 88-109.

Haesebrouck, T. 2015. Pitfalls in QCA's Consistency Measure. *Journal of Comparative Politics* 2:65-80.

Hino, A. 2009. Time-Series QCA. *Sociological Theory and Methods* 24 (2): 247-265.

Hinterleitner, M., Sager, F. & E. Thomann. 2016. The Politics of External Approval: Explaining the IMF's Evaluation of Austerity Programs. *European Journal of Political Research*. Advance online publication. DOI: 10.1111/1475-6765.12142.

Maggetti, M. & D. Levi-Faur. 2013. Dealing with Errors in QCA. *Political Research Quarterly* 66(1): 198-204.

Medzihorsky, J., Oana, I., Quaranta, M. and C.Q. Schneider. 2017. *SetMethods: A Package Companion to "Set-Theoretic Methods for the Social Sciences". R* Package Version 2.1. URL: http://cran.r-project.org/package=SetMethods.

Ragin, C.C. 1987. *The Comparative Method: Moving Beyond Qualitative and Quantitative Strategies.* Berkeley and Los Angeles: University of California Press

Ragin, C.C. 2000. *Fuzzy-Set Social Science*. Chicago and London: University of Chicago Press.

Ragin, C.C. 2006. "Set Relations in Social Research: Evaluating Their Consistency and Coverage." *Political Analysis* 14(3): 291-310.

Ragin, C.C. 2008a. Easy Versus Difficult Counterfactuals. *Redesigning Social Inquiry: Set Relations in Social Research.* Chicago: University of Chicago Press, chapter 9.

Ragin, C.C. 2008b. Measurement versus calibration: a set-theoretic approach. *The Oxford handbook of political methodology*. Oxford Handbooks Online: 174-198.

Ragin, C.C. 2009. Qualitative Comparative Analysis Using Fuzzy Sets (fsQCA). *Configurational Comparative Methods. Qualitative Comparative Analysis (QCA) and Related Techniques*. Los Angeles, London, New Delhi and Singapore: Sage Publications, 87-121.

Rihoux, B. & C.C. Ragin. *Configurational Comparative Methods. Qualitative Comparative Analysis (QCA) and Related Techniques*. Los Angeles, London, New Delhi and Singapore: Sage Publications.

Schneider, C.Q, & I. Rohlfing. 2013. Combining QCA and process tracing in set-theoretic multi-method research. *Sociological Methods & Research* 42(4): 559-597.

Schneider, C.Q. & C. Wagemann. 2012. *Set-Theoretic Methods for the Social Sciences. A Guide to Qualitative Comparative Analysis*. New York: Cambridge University Press.

Schneider, C.Q. & C. Wagemann. 2013. Doing Justice to Logical Remainders in QCA: Moving Beyond the Standard Analysis. *Political Research Quarterly* 66(1): 211-220.

Schneider, C.Q. & C. Wagemann. 2015. Assessing ESA on What It Is Designed for: A Reply to Cooper and Glaesser. *Field Methods*. Advance online publication. DOI:1525822X15598977.

Thiem, A. 2016. Conducting Configurational Comparative Research With Qualitative Comparative Analysis A Hands-On Tutorial for Applied Evaluation Scholars and Practitioners. *American Journal of Evaluation*, DOI: 1098214016673902.

Thiem, A. & A. Dusa. 2012. *Qualitative comparative analysis with R: A user's guide*. New York: Springer Science & Business Media.

Thiem, A. & A. Dusa. 2013a. Boolean Minimization in Social Science Research: A Review of Current Software for Qualitative Comparative Analysis (QCA). *Social Science Computer Review* 31(4): 505-521.

Thiem, A., & A. Dusa. 2013b. QCA: A package for qualitative comparative analysis. *The R Journal 5*(1): 1-11.