

Adrian Duşa

QCA with R

A Comprehensive Resource

 Springer

QCA with R

Adrian Duşa

QCA with R

A Comprehensive Resource

 Springer

Adrian Duşa
Romanian Social Data Archive
University of Bucharest
Bucharest, Romania

ISBN 978-3-319-75667-7 ISBN 978-3-319-75668-4 (eBook)
<https://doi.org/10.1007/978-3-319-75668-4>

Library of Congress Control Number: 2018933067

© Springer International Publishing AG, part of Springer Nature 2019

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by the registered company Springer International Publishing AG part of Springer Nature.

The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preamble

The initial idea to write this book emerged in August 2016, with the intention to update the description of package *QCA* in R. A lot of things have changed since the appearance of the previous user guide 5 years ago, when the package reached version 1.0-0.

Most of the commands are backwards compatible, meaning that all examples from the previous guide still work with the current version of the package. There are some minor but important differences; for instance, the function `calibrate()` has its default changed from `type = "crisp"` to `type = "fuzzy"`. Examples still work with previous versions, but for crisp calibration the argument `type = "crisp"` should be explicitly specified. All changes, from all versions of the package, are specified in the `ChangeLog` file which is located in the package's installation directory (details in chapter 2).

This book is correlated with the release of version 3.3 of the package. There are many new, cutting-edge features that were not available before: dramatically improved functionality and of course a graphical user interface to enable R beginners to use this package using a point and click method. If previously it was all about the command line, now it is all about the interconnection between the command line and the graphical user interface.

To advance the presentation from section 2.4, the point of the graphical user interface is not to replace the command line but to offer a more accessible way to learn how to use the commands. No graphical user interface, irrespective of how user friendly, can replace the complete flexibility of the command line. The variety of available written functions, from the base R and specifically from package *QCA*, is beyond the possibilities of any standardized graphical (therefore confined) user interface.

But there is one thing that makes this user interface special: each and every mouse click on the various options triggers an immediate construction of the corresponding command. The real purpose of the graphical user interface is

to demonstrate one possible way to write commands, while there are many others that users are invited to discover.

The structure of this book is different from the former user guide. It will of course touch on the same topics and present the same (updated) package, but instead of organizing chapters on the distinction between crisp, multi-value and fuzzy sets, a better approach is to organize the book on QCA-related analyses: calibration, analysis of necessity, analysis of sufficiency, constructing and minimizing a truth table, etc.

It has grown to be much more than a guide, but it is also not intended as a complete theoretical material. This is valid for both R-related information (chapter 1 being a very short introduction) and QCA-related topics. There are entire books to cover each, and this book cannot replace them all, but instead it shifts the focus on how to perform QCA using R.

There are dedicated chapters combining theory for all QCA variants (cs, mv and fs, also extensions) and R, with detailed instructions on how to perform each of the analyses using both command line and the new graphical user interface. Readers are nevertheless invited to supplement their knowledge from other sources, most being introduced in the specific sections.

To say that writing this book has been a challenge is an understatement. It started as a challenge, but it has developed into something I definitely did not anticipate. The text in the book is written for an imaginary character with whom I had many interesting discussions emerging from natural questions. This character can be both a QCA expert who is interested in using R and an R expert who is interested in QCA, or perhaps the most common scenario someone who is neither a QCA nor an R expert. But the questions are the same:

- What exactly is this? and/or
- How do I do it with R?

Many sections and paragraphs, sometimes down to each word, had to be rewritten while conversing with this imaginary character and trying to anticipate what kind of questions and what kind of answers are expected from this book. During all this process of describing and explaining, I have often realized that many things that I took for granted were not exactly trivial. This was practically an invitation to rethink and rewrite the structure of the commands to better answer possible questions, while at the same time increasing the user experience.

Much like a sculptor who has a concrete idea about the potential of a statue to be carved out from a stone, the initial book plan gradually adapted to the new structure to accommodate all ideas that have emerged while writing it. In a way, the book wrote itself, just as the statue that already exists inside the stone and the sculptor only removes the excess material to reveal it. I am extremely pleased to have actively experienced this process.

Naturally, improving the code in the package implied further and better explanations in the book, while describing the concepts many times involved rewriting the code, into an energy-draining spiral that lasted for almost an entire year. On top of the book, and on top of the code, every change meant even more additional effort to update the graphical user interface. In the absence of a formal background in programming, writing the package and especially the graphical user interface was a Herculean effort.

In order to circumvent differences between operating systems, the natural choice is to open dialogs into the most universal environment, a web page. Thankfully, using the excellent package *shiny* made this effort manageable, although I still had to learn HTML, Javascript and SVG in the process. It has now reached a rather stable and easy to manage backend, while the front end is very close to an ideal, combining point and click dialogs with a web-based command line, all in the same workspace.

The only important thing that it still lacks, and will certainly stay on the development list for the next versions, is a code editor. There are a number of very good choices to pick from, but this is something I simply did not have the time to do for this book. Most likely, the next versions of the package will be geared towards completing the graphical user interface.

R is a great environment and few people are aware that it can be used to write books. Not surprisingly, this book was written using R. There are not enough words to describe the amazing work of Yihui Xie and all the team of engineers from *RStudio*, who provide this public service and especially for the packages *knitr*, *rmarkdown* and *bookdown* that allow this form of HTML, PDF and ebook publication (among others). It made the writing of this book a whole lot easier than I had expected, by automatically transforming the markdown code into LaTeX and finally rendered into an eye-candy PDF version.

In the book, there are many places where theoretical information is exemplified using R commands. These are placed into specific grey boxes, and they have colours associated (for instance, all functions have red colour, arguments are green and text input is blue). With the exception of very few commands that are only illustrative, all of them have an output which is printed immediately below, using a monospaced font.

I am deeply thankful to a select group of people who contributed feedback while writing the book, and even provided ideas to further improve the package functionality: to Eva Thomann, Omar Bautista González and Monica Şerban for the very careful reading of the entire book text, and to Ioana-Elena Oana and Carsten Schneider for all their work making the package *SetMethods* work smoothly with package *QCA*.

There have been numerous other suggestions, bug reports and feature requests from too many people to mention here, and I very much appreciate all their interest.

I would like to dedicate this book to my wife Adriana Rotar, who has been overly patient with me while writing it. I have taken time out from too many common activities, and I am grateful beyond words for being so understanding.

Bucharest, Romania
November 2017

Adrian Duşa

Contents

1	R Basics	1
1.1	Working Directory	3
1.2	Workspace	4
1.3	Object Types	5
1.4	Mathematical and Logical Operations	8
1.5	Indexing and Subsetting	10
1.6	Data Coercion	13
1.7	User Interfaces	15
1.8	Working with Script Files	17
2	The <i>QCA</i> Package	19
2.1	Installing the <i>QCA</i> Package	19
2.2	Structure	20
2.3	Command Line Mode	22
2.3.1	Getting Help	22
2.3.2	Function Arguments	23
2.4	The Graphical User Interface (GUI)	26
2.4.1	Description	27
2.4.2	Starting the Graphical User Interface	29
2.4.3	Creating an Executable Icon	30
2.4.4	Command Construction	32
2.4.5	The Web R Console	34
2.4.6	Graphics	36
2.4.7	The Data Editor	39
2.4.8	Import, Export and Load Data	41
3	Set Theory	47
3.1	The Binary System and the Boolean Algebra	48
3.2	Types of Sets	48
3.2.1	Bivalent Crisp Sets	49
3.2.2	Multivalent Crisp Sets	50
3.2.3	Fuzzy Sets	52

3.3	Set Operations	53
3.3.1	Set Negation	53
3.3.2	Logical AND	54
3.3.3	Logical OR	57
3.4	Complex Operations	59
4	Calibration	61
4.1	Calibrating to Crisp Sets	63
4.2	Calibrating to Fuzzy Sets	71
4.2.1	Direct Assignment	72
4.2.2	Direct Method, the “S-Shape” Functions	74
4.2.3	How Does It Works: The Logistic Function	85
4.2.4	Direct Method, the “Bell-Shape” Functions	88
4.2.5	The Indirect Method	92
4.3	Calibrating Categorical Data	94
5	Analysis of Necessity	99
5.1	Conceptual Description	99
5.2	Inclusion/Consistency	105
5.3	Coverage/Relevance	110
5.4	Necessity for Conjunctions and Disjunctions	119
5.5	Exploring Possible Necessity Relations	122
6	Analysis of Sufficiency	125
6.1	Conceptual Description	126
6.2	Inclusion/Consistency	130
6.3	The PRI Score	134
6.4	Coverage: Raw and Unique	136
7	The Truth Table	139
7.1	General Considerations	139
7.2	Command Line and GUI Dialog	143
7.3	From Fuzzy Sets to Crisp Truth Tables	146
7.4	Calculating Consistency Scores	151
7.5	The OUTput Value	154
7.6	Other Details	158
8	The Logical Minimization	159
8.1	Command Line and GUI Dialog	161
8.2	Conservative (Complex) Solutions	163
8.3	What Is Explained	167
8.4	Parsimonious Solutions	172
8.5	A Note on Complexity	176
8.6	Types of Counterfactuals	178
8.7	Intermediate Solutions: SA and ESA	183
8.8	Theory Evaluation	194

- 9 Pseudo-Counterfactual Analysis** 197
 - 9.1 eQMC 198
 - 9.2 Consistency Cubes 203
 - 9.2.1 Search Space 205
 - 9.3 Include vs. Exclude 207
- 10 QCA Extensions** 209
 - 10.1 Temporal QCA 209
 - 10.2 Coincidence Analysis: CNA 214
 - 10.3 Panel/Clustered Data 225
 - 10.4 Robustness Tests 230
- 11 Less Known Features** 241
 - 11.1 Boolean Expressions 242
 - 11.2 Negate Expressions 248
 - 11.3 Factorize Expressions 250
 - 11.4 More Parameters of Fit 252
 - 11.5 XY Plots 255
 - 11.6 Venn Diagrams 263
 - 11.7 Custom Labels 271
- References** 273

Chapter 1

R Basics



In this chapter, users are introduced to a series of basic concepts in using R. There are many other, dedicated introductory books for this purpose (e.g. Verzani 2005; Dalgaard 2008; Zuur et al. 2009, among others). There is also a dedicated section containing freely available, contributed manuals on the CRAN (Comprehensive R Archive Network) website.¹

The purpose of this chapter is less about providing a complete and exhaustive guide to R, but rather to help understand at least those basic concepts that would facilitate using the *QCA* package.

Chapter 2 will help even further in this aim, especially Sect. 2.4 which shows how the graphical user interface can be used to interactively construct written commands, but for the start it is highly recommended that users would at least understand what R is all about and, perhaps most importantly, how it differs from the “usual” data analysis software.

Various object types will be briefly presented in Sect. 1.3. Each and every type of object will become relevant in the subsequent chapters, therefore a thorough understanding of those types is necessary. There are many other books that describe those objects, and such content is outside the scope of this book. Users are invited to supplement their knowledge with external reading, whenever they feel this brief introduction does not sufficiently cover the topic.

When first opening an R console, the user is immediately subjected to an inherent, perceived puzzle: there are no specific menus, and importing a simple dataset seems like an insurmountable task. When the data is somehow imported, the most general questions are:

- Where is the data?
- How do we look at the data?

¹ <https://cran.r-project.org/>.

These are natural questions for users who have worked with data analysis software like SPSS or even Excel, where the data is there to be seen immediately after opening the file. As users make progress with their R knowledge, they realize the data doesn't actually need to be seen. In fact, that is actually impossible unless the dataset is small enough to fit on the screen. There are large datasets (thousands of rows and hundreds of columns) that are only partially displayed, and users usually scroll left and right, as well as up and down through the dataset. Furthermore, R can open multiple datasets in the same time, or can create temporary copies of some datasets, this one cannot possibly "look" at everything.

When large datasets are opened, the human eye cannot possibly observe all potential problems in a dataset, many of which could remain undetected. For this purpose, R does a far better job using written commands to interactively query the data for potential problems.

Specific commands are used to obtain answers to questions such as: Are there any missing values in my data? If yes, where are they located? Are all values numeric? Are all values from a specific column in the interval between 0 and 1? Which are the rows where values are larger than a certain threshold? etc.

Even if the dataset is small and fits on the screen, it is still a good practice advice to address questions like the above using written commands, thus refrain from trusting what we usually see on the screen. For example, there are situations when a certain column is encoded as character, even if all values are numeric: "1", "5", "3" etc. When printed on the screen, the quotes are not automatically displayed, therefore users naturally assume the variable is numeric. A better approach is to query R if a certain vector (column) is numeric or not, as it will be shown in Sect. 1.3.

Before delving deeper into these issues, the simplest possible command for a first interaction with R is:

```
1 + 1
```

the result being:

```
[1] 2
```

R reads the command, it inspects it for possible syntax mistakes and if no errors occur it executes it and prints the result on the screen. This is how an interpreted language like R works, spending some time to first understanding what the text command is, then execute it. The big advantage of such an approach is interactivity, and an increased flexibility.

There are many details that, once understood and mastered, help users get passed the steep learning curve. Among those details, perhaps some of the most important are the concepts of (a) working directory, associated with data input and output, and (b) working space, with the associated types of objects.

1.1 Working Directory

One of the most challenging tasks for an absolute beginner is to get data in. Graphical user interfaces from the classical software make it look like a natural activity by offering a way to navigate through different directories and files using a mouse. In R, there are two ways to read a certain file: either by using its name, if it was previously saved in the so called “working directory”, or by providing the complete path with all directories and subdirectories, including the file name. More details will be provided in Sect. 2.4.8.

The simplest is to save it in the working directory, which can be found using this command:

```
getwd()
```

```
[1] "/Users/jsmith/"
```

There are differences between operating systems, for example in Windows one might see something like:

```
[1] "C:/Documents and Settings/jsmith"
```

A data file can be saved into this default directory, or alternatively the user can set the working directory to a new location where the file is found:

```
setwd("D:/data")
```

These two commands are very useful to select files and directories using commands, and as an alternative one can use the computer’s native file browser using the command `file.choose()`. Instead of specifying the complete path to a certain file, the import command (suppose reading from a CSV file) can be combined with the file browser:

```
datafile <- read.csv(file.choose())
```

This command is a combination of two functions, executed in a specific order. First, the file browser is activated by the function `file.choose()`, and once the user selects a specific CSV file by navigating to a specific directory, the path to that file is automatically constructed and served to the function `read.csv()` which performs the actual data reading.

For the moment, the result of this combination of commands is printed on the screen, but the usual method is to assign this result to a certain object, as it will be shown in the next section.

1.2 Workspace

Every object, created or imported, resides in the computer's memory (RAM). In R, almost everything is an object: data files, scalars, vectors, matrices, lists, even functions etc. For instance, creating an object that contains a single value (a scalar) can be done using the same operator “<-” for the assignment:

```
scalar <- 3
```

A new object called `scalar` was created in the workspace, and it contains the value 3. The list of all objects currently in the workspace can be seen using this command:

```
ls()
```

```
[1] "datafile" "scalar"
```

Some objects are read from the hard drive, while others are created directly in the R environment. With sufficient RAM available, it really is not a problem to accommodate (very) many objects, even if some of them are large. It is only troublesome when the workspace gets polluted with many temporary objects, that makes finding the important ones more difficult.

Individual objects can be eliminated with:

```
rm(scalar)
```

Left unsaved, the workspace is lost when closing R. Apart from exporting datasets using `write.table()` (more details in Sect. 2.4.8), individual objects can also be saved in binary format:

```
save(datafile, file = "datafile.Rdata")
```

It is also possible to save the entire workspace in binary format, with all containing objects:

```
save.image(file = "workspace.Rdata")
```

Cleaning up the entire workspace (removing all objects at once) can be done using this command:

```
rm(list = ls())
```

With binary workspace images saved on the hard drive, the opposite command is to bring them back to R when needed. There is a single function to load both types of binary files, a single object or an entire workspace, using:

```
load("workspace.Rdata")
```

1.3 Object Types

A vector is a collection of values of a certain type (called “mode”): numeric, character, logical etc. It is probably the most common type of object in R, making an important difference from other software. In many cases, the structure of the data is restricted to a rectangular shape with cases on rows and variables on columns but R is more flexible, allowing that type of file but also all kinds of other structures.

It is not always necessary to structure values in a rectangular shape. Sometimes we might be interested in the values of a (single) simple vector, for any possible reason including to just play with it and see what happens when applying different transformation commands.

Vectors are simple, but very powerful structures. Sometimes they contain values of their own, and some other times they contain indexes referring to the positions of certain values in other vectors or other objects. The simplest way to create a vector is to use the `c()` function:

```
nvector <- c(1, 3, 5, 7, 9, 11)
```

The name of the vector (just as the name of any other object) is not important, in this case chosen as `nvector` but users are free to experiment with any other. Invoking the name will print its content on the screen:

```
nvector
```

```
[1] 1 3 5 7 9 11
```

The same values can be obtained using some other predefined functions, such as `seq()` which generates a sequence of every two values between 1 and 9:

```
seq(1, 11, by = 2)
```

```
[1] 1 3 5 7 9 11
```

As stated before, the most common type of vectors are numeric, logical, character or factor. The above example is numeric, and creating character vectors is just as simple:

```
svector <- c("C", "C", "B", "A", "A")  
svector
```

```
[1] "C" "C" "B" "A" "A"
```

Factors are a special type of vectors, that define categorical variables. The categories are called “levels” in R, and they can be either unordered (nominal variables) or ordered (ordinal variables).


```
fvector <- factor(svector)
fvector
```

```
[1] C C B A A
Levels: A B C
```

Unless otherwise specified, the levels are printed in alphabetical order but irrespective of their arrangement this kind of factor still defines a nominal variable:

```
fvector <- factor(svector, levels = c("C", "A", "B"))
fvector
```

```
[1] C C B A A
Levels: C A B
```

To qualify as an ordinal variable, there is an argument called `ordered` that accepts a logical value:

```
ofvector <- factor(svector, ordered = TRUE)
ofvector
```

```
[1] C C B A A
Levels: A < B < C
```

The difference between this object and the previous one is given by the “<” sign between levels. If not otherwise specified, their order is still ordered alphabetically by default, but a preferred order can be specified:

```
cofvector <- factor(svector, ordered = TRUE, levels = c("B", "A", "C"))
cofvector
```

```
[1] C C B A A
Levels: B < A < C
```

Matrices are another type of useful objects, having two dimensions: a number of rows, and a number of columns:

```
mobject <- matrix(nvector, nrow = 2)
mobject
```

```
      [,1] [,2] [,3]
[1,]   1   5   9
[2,]   3   7  11
```

One useful thing about matrices is the fact they can have names for columns, as well as for rows, and they can be assigned with:

```
rownames(mobject) <- c("ROW1", "ROW2")
colnames(mobject) <- c("COL1", "COL2", "COL3")
mobject
```

```

      COL1 COL2 COL3
ROW1    1    5    9
ROW2    3    7   11

```

Perhaps the most important object type, used in almost all data analyses, is the data frame. Such an object has the same, familiar rectangular shape with cases on rows and variables on columns. In fact it is a list, with an additional restriction that all its components have the same length (just like all variables in a dataset have the same number of cases).

```

dfobject <- data.frame(A = c("urban", "urban", "rural", "rural"),
                      B = 12:15, C = 5:8, D = c("1", "2", "3", "4"))
rownames(dfobject) <- paste("C", 1:4, sep = "")
dfobject

```

```

      A B C D
C1 urban 12 5 1
C2 urban 13 6 2
C3 rural 14 7 3
C4 rural 15 8 4

```

In other software, only the columns (variables) have names. In R the rows (cases) can also have names and, especially for QCA related analyses, it makes sense because each case is well known and studied, as opposed to large N quantitative analyses where cases are randomly selected from a larger population. On a moderately large number of cases, QCA works with aggregate entities (communities, countries, regions) where the case names are just as important as the columns, or causal conditions.

One very important thing to notice is the way column D is printed on the screen. It looks numeric, just as columns B and C, however it is clear from the command this was written using characters like "1". Returning to the advice in the preamble of this chapter, we should not trust what we “see” on the screen. Instead, we can query the structure of the R objects:

```
str(dfobject$D)
```

```
Factor w/ 4 levels "1","2","3","4": 1 2 3 4
```

It can be seen the column D is not numeric, instead it is a factor due to the automatic coercion of character to factor in the function `data.frame()`. The same kind of query can be performed using the `is.something()` family:

```
is.numeric(dfobject$D)
```

```
[1] FALSE
```

This is also a very good way to verify if a seemingly numeric column is indeed numeric, and similar queries can be performed using `is.character()`, or `is.factor()`, or `is.integer()` etc.

1.4 Mathematical and Logical Operations

R is a vectorized language. This makes it not only very powerful, but also very easy to use. In other programming languages, to add the number 1 to a numerical vector, the commands have to explicitly iterate through each value of the vector and increase it with 1, something like:

```
for (i in 1:6) {
  print(nvector[i] + 1)
}
```

In a vectorized language like R, this kind of iteration is redundant, because R can work with entire vectors at once:

```
nvector + 1
```

```
[1]  2  4  6  8 10 12
```

R already knows the object in this mathematical operation is a vector, and it takes care of the iteration behind the scene. The code is not only easier to write, but it is also a lot easier to read and understand. This is one of the many reasons why R became so popular: it is so easy to use that thousands of non-programmers have been able to quickly use it and contribute with more packages.

In the above iteration, there is a certain sequence operator “:” which can be easily overlooked. It generates the sequence of all numbers between 1 and 6:

```
1:6
```

```
[1] 1 2 3 4 5 6
```

R’s vectorized nature has even more advantages, because mathematical operations can be applied with two vectors at once:

```
nvector + 1:2
```

```
[1]  2  5  6  9 10 13
```

The resulting vector seems strange, but reveals another very powerful feature in R called “recycling”: the values in the smaller vector (of length 2) are recycled until reaching the length of the longer vector. The values 1 and 2 are automatically reused three times and added to the next values of the `nvector`.

Addition is one of the possible mathematical operations in R. There is also subtraction with “-”, multiplication with “*”, division with “/” and so on.

But there are also logical operations, that in combination with indexing (see next section) contribute to the dialogue between the user and the data. With very large datasets that are impossible to print on the screen, users have to interrogate:

Is there a value there? Is that value numeric? If yes, is it bigger than the other number? Is there any value equal to 3? Which is the first one?

These kinds of human interpretable questions can be “translated” into the R language, via logical operations applied to either entire object or to specific values.

```
is.numeric(nvector)
```

```
[1] TRUE
```

The above command asks R if the object `nvector` is numeric, which means that all its values are numbers. The result of a logical operation can be a `TRUE` if the answer is true, and `FALSE` if not true. These kinds of operations can also be applied on each of the containing values:

```
lvector <- nvector > 3
lvector
```

```
[1] FALSE FALSE TRUE TRUE TRUE TRUE
```

There is one logical result for each value compared with 3, which means the result of this logical operation is a logical vector of the same length. The same kind of procedure can be applied for all kinds of human like questions, such as testing equality with 3:

```
nvector == 3
```

```
[1] FALSE TRUE FALSE FALSE FALSE FALSE
```

Note the usage of the double equal “==” sign to test equality. This is necessary because in R, much like in all other programming languages, a single “=” is used to assign value(s) to an object, just like the “<-” operator.

The result is a logical vector containing 6 values, and now it is possible to ask which value is equal to 3:

```
which(nvector == 3)
```

```
[1] 2
```

Combining functions like `which()` with logical operators like “==” makes the communication with R possible and easy. Knowing the result of the command above is a vector, it is now possible to further introduce indexing in the command to ask even more complex questions like which is the first value bigger than 5, in this case being found on the fourth position:

```
which(nvector > 5)[1]
```

```
[1] 4
```

1.5 Indexing and Subsetting

Indexing is a powerful tool to inspect various parts of the data or to perform data modifications based on certain criteria. Understanding how indexing works is fundamental for data interrogation, extracting subsets of data, analysing the structure of the objects and generally it facilitates a communication method between R and the user.

R is a 1-based language, which means the first element of an object is indicated with the number 1. By contrast, there are 0-based languages where the first element is indicated with the number 0, while number 1 refers to the second element. This is a useful information, especially for users previously exposed to other software.

Through indexing, it is possible to refer to specific values in different positions of an object. The second element of the numeric vector is 3, and that can be seen with:

```
nvector[2]
```

```
[1] 3
```

Indexing is performed using the square brackets “[”, and the value(s) inside depend on the number of dimensions an object has. A vector has a single dimension, therefore a single number is needed to refer to a specific position. Matrices and data frames have two dimensions (rows and columns), hence two numbers are needed to indicate a specific cell at the intersection between a row and a column.

```
dfobject[3, 2]
```

```
[1] 14
```

In the data frame object above, the value found in the third row of the second column is 14, and the positions inside the square brackets are separated by a comma. This is an important detail that defines a syntax rule which, if not properly used, can generate an error.

The same type of indexing can be performed on matrices:

```
mobject[2, 3]
```

```
[1] 11
```

An interesting matrix property is the way it is stored in the memory. It looks like a two dimensional object (just like a data frame), but underneath the matrix is just a vector that only acts like it has two dimensions. The same value 11 can be obtained by indexing with the last, sixth value:

```
mobject[6]
```

```
[1] 11
```

For data frames, columns can be selected (and indexed) via the second value in the square brackets. But the variables (on columns) can also be selected using the useful “\$” operator before the variable name:

```
dfobject$A
```

```
[1] urban urban rural rural
Levels: rural urban
```

The command above selects a single column `A` from `dfobject`, and it is printed on the screen as a vector. The `data.frame()` function automatically converts any character variable to a factor (categorical variable) and prints its levels. To prevent coercion, there is a logical argument called `stringsAsFactors`, which needs to be set as `FALSE` when creating the data frame.

The vector can be further indexed via the usual method:

```
dfobject$A[4]
```

```
[1] rural
Levels: rural urban
```

As mentioned, a data frame is a list with an additional constraint that all its components must have the same length. Lists can also be indexed using the double square brackets “[[”:

```
dfobject[[2]]
```

```
[1] 12 13 14 15
```

The result of the above command is a vector which can be further indexed using a sequence of square brackets, the example below selecting the second component (second variable column) and inside that, the third value:

```
dfobject[[2]][3]
```

```
[1] 14
```

One of the other features of the indexing system in R is the negative indexing. The positive one selects values from certain positions, as seen in the previous examples. Negative indexing, on the other hand, presents all values except those specified between the square brackets.

```
dfobject[[2]][-3]
```

```
[1] 12 13 15
```

There are many other things about indexing, outside the scope of this section, but there is one more important feature that is worth presenting: the indexes can be single values, if a single value is to be selected from a certain position, but in other circumstances indexes can be vectors themselves (vectors of indexes).

This is an important observation that prepares other advanced concepts like subsetting, where vectors of certain positions are used to select certain observations from a dataset, or to apply certain calculations on some, or based on, certain observations. Indexing is present in every data manipulation in R, and it is well worth learning how to properly use it.

Subsetting is similar to indexing, but with the specific objective to obtain a “subset”, a certain part of a dataset that need to be analyzed to answer specific research questions. It is sometimes called “filtering”, especially when applied to the rows of a dataset.

Subsetting can be performed using numerical vectors like in the examples above (with certain positions to keep or to eliminate), but it can also use logical vectors where any position that has a TRUE value will be preserved (or the other way round, any position with a value of FALSE is eliminated).

```
nvector
```

```
[1] 1 3 5 7 9 11
```

```
# using logical vectors (all values greater than 3)
nvector[lvector]
```

```
[1] 5 7 9 11
```

```
# using numerical positions
npos <- which(lvector)
npos
```

```
[1] 3 4 5 6
```

```
# now the same result with
nvector[npos]
```

```
[1] 5 7 9 11
```

It is especially useful for subsetting data frames, on both rows and columns. For columns it selects among their names (possibly with a specific pattern), and on rows it creates a subset of cases, in the example below those where the column C is greater than 6:

```
dfobject[dfobject$C > 6, ]
```

```
   A B C D
C3 rural 14 7 3
C4 rural 15 8 4
```

Note how the column C was specified in conjunction with the name of the object and separated by a “\$” sign, but otherwise it is possible to refer directly at the column name using the function `with()`, in this command adding the values from the columns B and C:

```
with(dfobject, B + C)
```

```
[1] 17 19 21 23
```

R has a base function called `subset()`, that can be used on both vectors, and rectangular objects like matrices and data frames. For dataframe, a possible command is:

```
subset(dfobject, B > 13, select = c(A, C, D))
```

```
      A C D
C3 rural 7 3
C4 rural 8 4
```

The command should be self explanatory, obtaining a subset of the `dfobject` keeping all rows where the column B has a value greater than 13, and from the columns selecting A, C and D.

1.6 Data Coercion

The function `is.numeric()` verifies all values in a vector, but unlike most of the logical operations in the previous section, it results into a single value indicating if the entire object is numeric or not.

This is possible because of a particular feature in R, that all values in a certain vector have to be of the same “mode”. If all values in a vector are numbers, such as those from `nvector`, the object is called numeric. But if a single value among all numbers is a character, the entire vector is represented as character.

Many people don’t realize that numbers are characters themselves, and most importantly the character “1” can be something different from the number 1 (especially if coerced to a factor). This conversion may happen if, by any chance, a true character is added to a numeric vector, or it replaces one of the values in the vector. At that moment, the entire vector which previously was numeric, is transformed (coerced) to a character type.

All values in a vector have to be of the same type. One single value of a different type is sufficient to coerce the entire vector to the other type, but the coercion goes only one way: a single number in a character vector does not coerce that into numeric. That happens because all numbers can act like characters, while not all characters can act as numbers.


```
cnvector <- c(1, 2, 3, "a")
cnvector
```

```
[1] "1" "2" "3" "a"
```

The object `cnvector` contains three numbers and one character, but due to the presence of the character, all numbers are now presented as characters. Removing the character leaves the vector with the same character type, even though all values are in fact numbers:

```
cnvector <- cnvector[-4]
cnvector
```

```
[1] "1" "2" "3"
```

Converting from one type to another is done using the “`as.`” family of functions, in this case with:

```
cnvector <- as.numeric(cnvector)
cnvector
```

```
[1] 1 2 3
```

All this demonstration reveals one of the most common situations in data analysis, especially for those users who want to “see” the data. Some objects are not always what we think they are, simply because we see them printed on the screen:

```
dfobject$D <- as.character(dfobject$D)
dfobject
```

```
      A  B C D
C1 urban 12 5 1
C2 urban 13 6 2
C3 rural 14 7 3
C4 rural 15 8 4
```

At a first sight, the fourth variable `D` seems to contain numbers just like `B` and `C`. However they only appears to be numbers, while in reality column `D` contains their character representation (e.g. “1”, “2”, “3” and “4”). This is an important observation for two reasons, with cascading effects over the next chapters:

- if at any data analysis stage, we need to perform mathematical operations on a vector we think it is numeric but in reality it is not, an (unexpected) error will be thrown and most novice users remain puzzled how did that error appeared “out of nowhere”
- as it was shown, assuming that something it is of a certain type just because we “see” it printed on the screen does not necessarily make it of a that type; instead, it is by far a better idea to always check if a variable is indeed what we think it is

A special type of coercion happens between the logical and numerical vectors. In many languages, including R, logical vectors are treated as numerical binary vectors with two values: 0 as a replacement of FALSE, and 1 as a replacement of TRUE.

```
# lvector is the logical vector created in the previous section
sum(lvector)
```

```
[1] 4
```

The reverse is also valid, meaning a numerical vector can also be interpreted as logical, where 0 means FALSE, any other number means TRUE, and the exclamation sign ! negates the entire vector:

```
!c(0, 2, 1)
```

```
[1] TRUE FALSE FALSE
```

1.7 User Interfaces

There are various ways to work with R, and the starting window presented in Fig. 1.1 is deceptively simple.

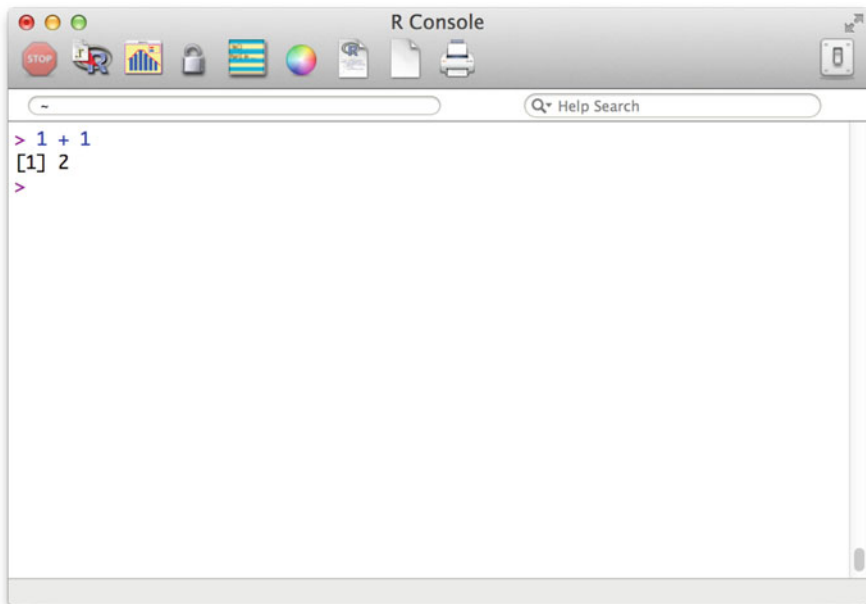


Fig. 1.1 The starting R window under MacOS

In fact, it is so basic that many first time users are puzzled: no fancy menus, and only a handful of buttons that are completely unrelated to data analysis. The starting window looks different in Windows and Linux, but the overall appearance is the same: a console, where users are expected to type commands in order to get things done.

There are very good reasons for which the base R window looks so simple. The best explanation is that a graphical user interface is completely unable to replace a command line. Writing efficient commands is an art, as there are literally thousands of different ways to obtain a certain result, via an infinitely large number of possible combinations of commands. Some are simple, some look complicated but are extremely efficient, it all depends on the user's knowledge.

In contrast, a graphical user interface is necessarily restricting to exactly one possible way (among thousands) of obtaining a result. For simple activities such as inspecting the data, or basic data manipulations (creating or modifying variables etc.) it is relatively easy to provide a graphical user interface. But unless the work is highly standardized with exactly similar input and output, each research problem is unique and requires a custom solution.

Customizing solutions using written commands is a lot easier than using point and click graphical interfaces. In some cases, the interface is simply unable to answer very specific problems, because it has not been designed for those problems in the first place. This is the very reason for which, although R exists for almost three decades, there are very few graphical user interfaces that are capable of a smooth interaction with the base R environment.

On the other hand, especially for those who have never used R before, the simple command line can be a deterrent. The newest interface that is currently most fashionable and widely recognized as user friendly is RStudio, presented in Fig. 1.2. It depends on a valid R installation, and it is built in a local web environment which solves the problem of differences between different operating systems.

Most importantly, RStudio manages to bring into a single interface not only the command console, but also an object browser, as well as a package installer, a plot window etc. It is more or less the closest possible user interface that balances both the expectation of a point and click user with the inherently rich set of possibilities from the R environment.

One of the most useful features of the RStudio is the History tab, where all previous commands in the Console are stored and can be inspected later. It is not shown in the figure, but it also has a modern syntax editor with a bracket matching, different colors to highlight functions, comments and operators etc. This is extremely useful, especially for the possibility to quickly select the commands to be tested and immediately sent to the console. Although having a much better user experience, RStudio is also a command centric interface, for the same reasons stated above.

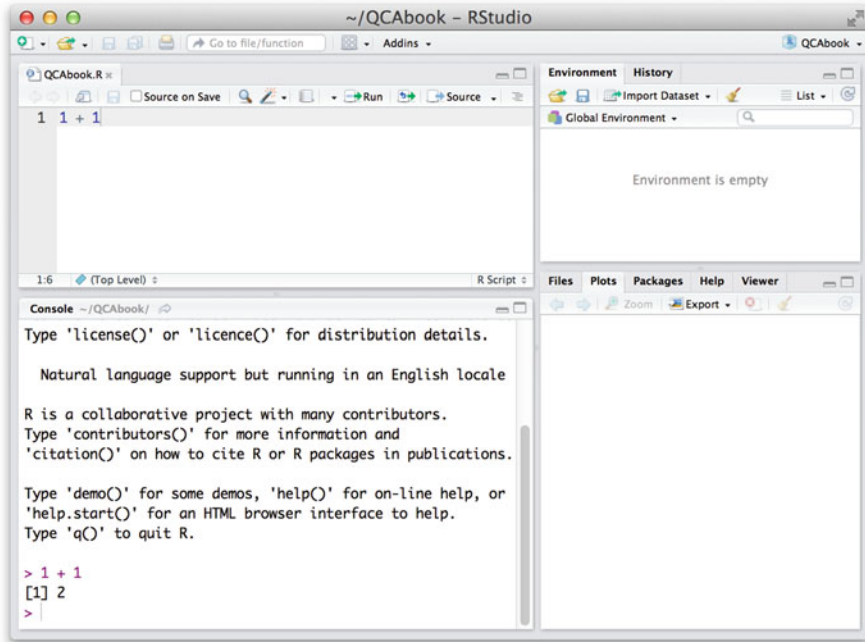


Fig. 1.2 The RStudio user interface

1.8 Working with Script Files

The upper left part of the RStudio window in Fig. 1.2 above is a text file with an extension “.R”, where specific commands can be written and saved for later references. Commands can be selected and run in the R environment from the lower left part, either individually (specific rows) or the entire file with all commands at once.

These kinds of scripts, containing written commands, are the heart of the R environment. Not only because written commands are infinitely more flexible than a graphical user interface with precise boundaries, but also because scripts allow one of the most appreciated features of scientific research: replicability.

It is extremely difficult (if not impossible) to obtain exactly the same results using a graphical user interface, unless the exact sequence of clicks, selections and various other settings are somehow recorded, assuming the same input data is used. Users who click though a graphical dialog rarely think to document the exact sequence of operations, and simply reports results as if all other users would know what the sequence is.

At the other end, commands can be saved in text files, so that anyone else could follow (simply by copy/paste) the commands and obtain exactly the same results. And syntax files have an even more important role for the user: they can also serve as documentation vehicles for particular ideas contained in different commands.

Any line which starts with a hash tag `#` is considered a comment (as readers might have already noticed in the previous examples), and anything on that line is not evaluated therefore it can contain natural language about what the commands do. It is a good practice advice to document as much as possible the script files, especially when beginning to use R. Not only it makes it easier for a different reader to understand the commands, but many times it makes it easier for the same user to understand the code, some times later.

Just like data files (or R binary data files that usually have the extension “.Rdata”), syntax files can reside in a working directory, usually having the extension “.R” but none of these file extensions are mandatory. The commands inside can be run line by line, or in several batches of lines (using either RStudio or by simply copy and paste to the R console), but otherwise it is also possible to run all commands from the entire script file (possibly starting by reading the dataset, manipulating it, and finally obtaining the end results) using one single command in R:

```
source("path/to/script/file.R")
```

Such script files are easy to exchange with peers and colleagues, and they work just the same under any operating system, making R a highly attractive and interoperable working environment for teams of people all over the world.

The new *shinyapps*,² as well as the *flexboards* and probably even more spectacular *R Notebooks*, although requiring a more advanced R knowledge, are even more attractive ways of working with international teams by exchanging not only the script files but also their immediate, visible results.

RStudio has many more functionalities, for instance the package *bookdown* can be used to write articles and even books. This very book was written using this package.

² <https://www.shinyapps.io/>.

Chapter 2

The *QCA* Package



In the first chapter, we have briefly covered some of the basics in using R. This chapter presents the structure of the *QCA* package with its associated functions, complementing their explanation with an overview of the associated graphical user interface.

Many functions will be thoroughly explained in dedicated chapters (for example how to construct a truth table, or how to calculate parameters of fit), while this chapter will only cover the general usage, valid for all functions. Since the user interface in this package is novel, most sections are recommended for all users, beginners and advanced alike.

Some of the sections from this chapter are highly specific and very technical about the package itself, but have very little to do with either R in general or *QCA* in particular. Readers need not insist too much on those sections (for instance Sect. 2.2 about the package's structure or Sect. 2.4.3 about how to create an executable icon to start the graphical user interface), they are provided just in case it might prove to be useful for advanced users.

2.1 Installing the *QCA* Package

The easiest way to install the *QCA* package is to start R and follow the platform specific menus to install packages from CRAN.

- For MacOS, the menu is: **Packages & Data/Package Installer**. In the resulting window, there is a button called 'Get List' to get all the binaries from CRAN, then select the *QCA* package also click on 'Install Dependencies'.
- For Windows, the relevant menu is: **Packages/Install package(s)...** and a list of CRAN mirror servers will appear (choose the one closest to your location), after which the list of packages is presented.

- On Linux, probably one of the easiest ways is to install from the source package(s). Simply download them from CRAN, open a Terminal window and type: `R CMD INSTALL path/to/the/source/package.tar.gz` (this method works on all platforms, provided that all the necessary build tools are installed). For certain Linux distributions like Ubuntu, there are package binaries already produced and maintained on CRAN,¹ and users can install them via the usual package management software.

Apart from the obvious point and click method using R's graphical user interface, there is also a cross-platform method from within R, where users have to type:

```
install.packages("QCA", dependencies = TRUE)
```

There is also a package installer within the RStudio user interface (select the Packages tab, then click the 'Install' button), which is more convenient to use, but from my own experience it is always better to install packages from the base R, and then use them via any other user interface.

Once installed, the package needs to be loaded before the functions inside can be used. For all chapters and sections to follow, this book assumes the user has already run this mandatory command:

```
library(QCA)
```

2.2 Structure

The package *QCA* is organized in a series of directories and files, some of them containing the actual code (functions) while others containing their associated documentation and help files.

A good practice advice is to read the ChangeLog file upon each package install or update, because functions can slightly be changed or patched, or arguments can change their default values and previous code that used to work in some way would give different results on the next run (detailed explanations on how functions work will be given at Sect. 2.3 Command line mode).

One possible way to access the ChangeLog file (the text file mentioning each modification and addition that happens from one version to another) is to read it directly in the *QCA* webpage² on CRAN, where a link to the ChangeLog file is automatically provided by the build system.

¹ <https://cran.r-project.org/bin/linux/ubuntu/README.html>.

² <https://cran.r-project.org/web/packages/QCA/index.html>.

On the local computer, the `ChangeLog` file is located in the root of the package's installation directory, which usually is:

- on Windows, in `Program Files/R/R-3.3.3/library/QCA`
- on MacOS, in `/Users/your_username/Library/R/3.3/library/QCA/`
- on Linux, in `/home/your_username/R/architecture/version`

Each of these paths may change depending of the version of R (for example 3.3.3 at the time of this writing) or depending on the Linux distribution (via the `architecture` string), whether on 32 bit or 64 bit etc.

Users are allowed to define a custom library installation directory, which would help not only to circumvent the version-dependent path, but also to avoid re-installing all packages upon the release of a new version of R. As a bonus, it also helps working with multiple version of R in the same time (this is sometimes important for developers who want to test functionality for different versions).

Defining a custom installation directory is platform dependent (for example Linux users have to export the paths), therefore readers are encouraged to read more specific details on the 'R Installation and Admin manual' or via the `help(Startup)` command in R. Windows users may also want to take a look on the more automatic *batchfiles* package³ written by Gabor Grothendieck.

Once locating the installation directory of the *QCA* package, the actual content of the individual files (R code and its documentation) is not directly readable because the package is installed as a binary. However, the content is available in the source package, downloadable from the CRAN server.

Apart from the `ChangeLog` file, users may also be interested in the alternative help system which is located in the `staticdocs` subdirectory.

One thing that is likely important for all users is the so called "namespace" of the *QCA* package. R is a contributed software, with thousands of developers who constantly upload their work to CRAN. It is very likely that two or more developers to write functions using the same name (sometimes for completely different purposes), therefore R needed a way to avoid function name conflicts. The solution was to contain all function names in each package into a namespace, which protects the usage of some function names to a specific package.

Some other times, competing packages use the very same name for exactly the same purposes, and conflicts cannot be avoided. Users should be aware that precedence has the last loaded namespace, but to avoid such situations the best scenario is to refrain from using competing packages at the same time, or at least not in the same R environment. Starting R can be done in various ways, for example in the dedicated installed R user interface and also in the Terminal window.

³ <https://code.google.com/archive/p/batchfiles/>.

2.3 Command Line Mode

R is essentially a CLI—command line interface, which means it can be accessed via commands typed in the console (either in the R’s basic graphical user interface, or in the Terminal window). The first chapter explained the basics of the R commands, and the purpose of this section is to take the user one step further. Once properly understood, things will prove to be deceptively simple, but otherwise they are seemingly complex, especially for beginners.

Both the command line functions, and the user interface in the *QCA* package are developed with the intention to make things as easy as possible for any user. The design of the functions, with their formal arguments, have a twofold purpose: to perform the operations for Qualitative Comparative Analysis, and to involve an absolute minimal effort to use them.

Care must be taken, however, to make sure the right input is fed for the right arguments, otherwise the functions will stop with descriptive error messages.

The structure of a written command in the *QCA* package is determined, and closely follows the structure of any basic R command: it has arguments, default values, and relations between various values of associated argument. What is different and better, on top of the basic level of functionality, is an utmost care to capture all possible error situations and create clear and helpful error messages.

The R messages can sometimes be cryptical for beginners. They are very clear for advanced users, but for those who are only beginning to learn R they seem incomprehensible. Since the *QCA* package assumes only a minimal R knowledge, it is extremely important to provide highly descriptive error messages that allow users of any level to carry on their analysis. Achieving this level of information involved a huge effort to capture all possible situations where users might have difficulties, and return that kind of message which helps users to quickly understand the nature of the problem.

2.3.1 Getting Help

The arguments are generally self explanatory, and the functions have associated help files to provide more details where needed. To access such a help file, the easiest is to type a question mark “?” sign in front of the function name. For example, to see the help file of the parameters of fit function `pof()`, the command is simply:

```
?pof
```

As previously indicated, this command assumes the package *QCA* has already been loaded, so its namespace and functions are available to the user level.

Another possibility to browse the help files is to access the Index of all functions and datasets in the package (on the bottom of every help page there is a link to that Index), and select the function for which the help is needed. The very same set of help files is also available as a mini-series of static web pages built with the *staticdocs* package. Those pages are found in the `staticdocs` directory from the root installation directory of the package, or (as it will be shown in Sect. 2.4) as a dedicated button in the graphical user interface.

The information from these alternative help files are identical, with the added bonus for the web pages that they also present the result of the commands (especially useful in the case of plots or Venn diagrams).

2.3.2 Function Arguments

The structure of a function consists of one command (the name of the function) and a series of formal arguments that need to be specified by the user in order to obtain a result, or a specific type of result.

Let us examine one of the most simple functions in the package, `findTh()` to find calibration threshold(s) for a numerical causal condition, in the absence of any theory about how that condition should be calibrated (more information about calibration in Chap. 4).

The structure of this function is presented below:

```
findTh(x, n = 1, hclustm = "complete", distm = "euclidean", ...)
```

It has four formal arguments (`x`, `n`, `hclustm` and `distm`), plus the possibility for additional informal arguments via the “...” operator (mainly for historical reasons, to ensure backwards compatibility with the code in the former versions of the package).

As it can be seen, some of the arguments have already defined values (called “default”), as in the case of the argument `n`, the number of thresholds to find, which is pre-set at the value 1. On the other hand, the argument `x`, that contains the numerical input, doesn’t have any default value because it differs from case to case and from user to user.

The main difference between these two kinds of arguments is that `x` needs to be specified (without data, the function does not have any input to find thresholds for), while the argument `n` is optional (if the user doesn’t specify anything, the function will use its default value of 1).

To demonstrate, we first need some data, for example this vector containing hypothetical country GDP values:

```
gdp <- c(460, 500, 900, 2000, 2100, 2400, 15000, 16000, 20000)
```

Having created the object `gdp`, these two commands give identical results:

```
findTh(gdp)
```

```
[1] 8700
```

and

```
findTh(gdp, n = 1)
```

```
[1] 8700
```

This is a fundamental information on how functions generally work in R and it must always be remembered because, many times, these default arguments are “at work” even if users don’t specify them explicitly. Perhaps more important, sometimes arguments are “at work” without the user’s knowledge, a highly important incentive to carefully read the help files and make sure that all arguments have their proper values or settings.

Modifying the number of thresholds (more details in Chap. 4) will of course change the result:

```
findTh(gdp, n = 2)
```

```
[1] 8700 18000
```

Another important aspect regarding the written commands is the logical relationship between various arguments. Some of these arguments depend on the values of other arguments from the same command.

As an example, as it will be further explained in Sect. 4.2.2, let us examine the `calibrate()` function which has these formal arguments:

```
calibrate(x, type = "fuzzy", thresholds = NA, logistic = TRUE,
          idm = 0.95, ecdf = FALSE, below = 1, above = 1, ...)
```

The argument `logistic` depends on the fuzzy type of calibration: it would not make any sense to ask the function for a logistic function of a crisp type calibration. If the argument `type` is modified to “`crisp`”, then argument `logistic` will be ignored irrespective of its value.

The same dependency relationship can be observed between the arguments `logistic` and `idm`, where the later (which stands for the inclusion degree of membership) makes sense only for those cases where the logistical function is used for calibration, and since the logistic function depends on the fuzzy type of calibration it means the argument `idm` is logically associated with a

chain of dependencies starting from argument `type`. If either or both of those arguments are changed from their default values, the argument `idm` will be ignored.

This kind of behavior might be difficult to conceptualize on a written command, but is absolutely evident in the graphical user interface for the calibration menu (details in Chap. 4), where not only that arguments are ignored, but they actually disappear from the dialog when different previous arguments options have been selected. When choosing the crisp type of calibration the dialog presents only the options (arguments) for the crisp version, and when choosing the fuzzy type of calibration the options from the crisp version disappear and those for the fuzzy type appear instead. The same thing happens in the command line, it is just more obvious in the graphical user interface.

Perhaps one final thing that is worth noting, in the command line description for the package *QCA*, is a feature that simplifies users' experience to a maximal extent. Many times, especially in *QCA* applications, users are required to specify which causal conditions are used for a truth table, or what kind of directional expectations are used to further enhance the solution space.

All of those situations, as well as many others, involve specifying a character vector and the causal conditions is an easy example. The normal accepted way to deal with character vectors, in R, is to use quotes for each value, something like: `c("ONE", "TWO", "THREE")`.

But beginners often forget to add the `c()` function to construct the vector, or sometimes they might forget a quote somewhere, especially when the conditions' names are longer. This kind of error can only be captured by the base R, with the associated, seemingly "cryptical" messages. To help prevent these situations, and enhance the user experience, the *QCA* package has automatic ways to understand and interpret a single character string which, many times, can be used instead: `"ONE, TWO, THREE"`.

Such a string is easy to type, and it doesn't involve much attention on behalf of the user. The package automatically recognizes this string as a character vector (because it contains multiple commas inside), and it splits the string into several character substring corresponding to the initial values in the "normal" specification.

This is useful for two main reasons. First, it is now possible to specify variously complex disjunctive expressions (understood as sum of products) to calculate parameters of fit, something like `"natpride + GEOCON => PROTEST"`.

But as a more convincing example, specifying directional expectations (see Sect. 8.7) can sometimes prove to be tricky. These expectations are most of the times specified in the form of `c(1, 0, 0, 1, 1)`, but there are situations where users don't have any expectation at all (a "don't care"), and the usual way to specify this situation is to use the character `"-"`.

An error will appear if typing `c(1, 0, -, 1, 1)`, where the correct way to specify this vector is `c(1, 0, "-", 1, 1)`, and this is a very common situation for many first time users. To circumvent this possibility, the *QCA* package can read the more simple string `"1, 0, -, 1, 1"` which works just as well for both advanced and beginners because it is entirely enclosed between quotes. As an added bonus, it is easier to write and decreases the likelihood to make a mistake, overall enhancing users' experience.

2.4 The Graphical User Interface (GUI)

Working in a command line driven environment is both a blessing and a curse. With most of the other software offering intuitive menus and dialogs, users have a natural expectation for a similar interface when first opening R. It can actually be discouraging, because a command line is completely outside most of the users' experience.

But there are good reasons for which R is designed the way it is, and those programmers who started to build a graphical user interface know how very difficult a task this is, highlighted by the fact that no universally accepted graphical user interface was designed since the first release of R, over 20 years ago. There have been many attempts to design a GUI (in a random order, for exemplification: Deducer, Rcmdr, RStudio, JGR, RKWard, Rattle, Tinn-R), most of them presented by Valero-Mora and Ledesma (2012).

What is common to all these efforts is the attempt to accommodate R specific features (workspace, packages, console, object types etc.) in a graphical user interface which is normally designed for a different kind of functionality. This is especially evident in the data editor, a spreadsheet like feature that all user interfaces present: as any spreadsheet, it has a rectangular shape with cases on rows and variables on columns and that is the most expected shape when users want to view or edit their data.

The trouble is, R has many more types of data. There are scalars, vectors, matrices, arrays and lists to name the most common, and implementing a viewer/editor for all of those objects, as simple as it might seem, it is in fact close to impossible. Restricting the data editor to datasets only is even worse, fueling many users' expectations that data means a dataset when in fact data can have many other shapes.

And that is only a fraction of the entire complexity. R is constructed as a package centered software, where in addition to a core set of packages there are potentially thousands of other contributed packages from all users. There are also multiple environments, sessions, graphical devices, it would be practically impossible to accommodate everything into a single system of menus, adding up to the difficulty of designing a multi-purpose user interface.

2.4.1 Description

The graphical user interface in the *QCA* package is not different from any other attempt. It is strictly dataset oriented, an understandable fact since it is not designed as a general R graphical user interface but specific for QCA purposes, where input data usually has a rectangular shape.

To avoid duplicating efforts and make the users' experience as smooth as possible, it was extremely important from the very beginning to create a single version for all operating systems, without any additional software necessary to be installed except for the base R.

One highly appreciated user interface that accomplishes these goals is the R Commander (Fox 2005), a point and click interface that can be installed as a regular R package and opened from within R. No other software is necessary, and it can be opened (just like R) in any operating system because it was written in the Tcl/Tk language.

This is one of the best user interfaces for R, and the structure of the user interface in the *QCA* package is similar to the one from R Commander (it has menus and separate dialogs), however its internal functionality is closer to the other highly appreciated user interface, RStudio.

Much like RStudio, the *QCA* graphical user interface is constructed in a webpage using the powerful *shiny* package (RStudio, Inc 2013) with additional Javascript custom code on top. Perhaps the best description of the *QCA* user interface is the effort to achieve *Rcmdr* functionality using *shiny* and RStudio technology, combining the best features from both.

A webpage is likely to become the most widely used environment to exchange and visualize data, partial results or even complete reports or presentations. It is obvious that many of the “traditional” software have siblings in cloud based environment (Google Docs for example), and it is actually possible to fully use an entire software in a webpage, through some sort of a virtual machine.

The team from RStudio is actually leading the way to building all sorts of webpage based useful tools, for example Shiny dashboards and html widgets. There is also another collaboration tool with a high potential to produce a significant impact in the academic world, named R Notebooks: not only it makes the code exchange possible, but thanks to a careful design it also exchanges the output of the code, all in a single HTML document that can be literally opened everywhere, irrespective of having R installed.

It seems that cloud and webpages, with flavors of HTML and Javascript inside, are starting to become a new kind of collaboration environment that R is happy to work with, either directly communicating via the package *shiny* or building snippet documents like R Notebooks.

These are precisely the reasons for which the *QCA* graphical user interface has been designed to be opened in a webpage: it is a modern approach, and with so many new tools appearing every day it has good prospects to still be valid many years from now.

The purpose of this chapter is less to offer an exhaustive presentation of the entire user interface, some menus and dialogs will be presented in specialized chapters. At this point, it is probably more important to introduce the most important features, especially how to open it and how to make the best out of the interface. Although it allows most of the base R features through the web based R console (see Sect. 2.4.5), this is not a general purpose R graphical user interface, but a *QCA* specific one.

Once opened, it looks like any regular software installed on the local computer with menus, and each menu opens a dialog, where users click through various options and set various parameters then run the respective command.

The only difference, because the interface is opened in a webpage, is that dialogs are not separate windows but are created inside the webpage environment: if that is closed, all open dialogs will be closed.

Otherwise, there are multiple advantages building the user interface in a webpage. To begin with, the user interface looks exactly the same irrespective of the operating system: since the HTML and Javascript languages are standard and cross-platform, a webpage should look identical everywhere.

But there are even more advantages. Unlike traditional point and click software where dialogs are static, building these dialogs into a webpage makes possible to use all HTML and Javascript features that are normally present into a web environment: reactivity, mouse-over events, click events etc. The dialogs themselves change, function of specific options being clicked.

This will be especially evident in Chap. 4, where the calibration dialog contains a plot region where users can actually see the distribution of raw against the calibrated values, function of specific calibration options. This is helpful because it eliminates an additional step of verifying the result of the calibration after the calibration process: instead, it shows how the calibration looks like before the command will be sent to R.

Some dialogs can be resized to accommodate complex information, and all content will automatically be resized at the new width and height of the dialog. Such examples will be presented for example in Sect. 11.6, where Venn diagrams for more than 4 or 5 sets become too complex for a smaller plot region, or even for an XY plot dialog with very many points to plot.

2.4.2 Starting the Graphical User Interface

The QCA graphical user interface is essentially a shiny app. An “app” is a generic term for any application project done with the package *shiny*, which provides the possibility to build interactive user interfaces that help understand the data structure and clarifies the nature of the problem to be solved. There are thousands of potential applications built with Shiny, either on local machines, or personal servers or even published and shared on the RStudio servers (<https://www.shinyapps.io/>).

While the RStudio family of applications open up an entire world of possibilities, like the interchange format of an R Notebook discussed in the previous section, a Shiny app does even more: it allows the user to interact with the R system using an easy to use, intuitive interface. A Shiny app need not be limited to a specific example, it can be customized to accepts various inputs which are fed to R by the interface, making the user’s life as easy as possible.

Shiny can do this kind of mediation between the user and R, because it has a communication engine that sends the commands to R, which responds to the HTML environment and the answer is displayed on the screen. Users don’t even have to know R to use these kinds of interfaces, they only need to click, drag and generally do whatever the web-based app offers within. This communication was possible because R provides a native web server, initially designed to display help pages for the various functions in various packages. Shiny built on this capability, and uses the R’s internal web server as a communication tool between the webpage and R.

Any such Shiny app can be opened using a function in the *shiny* package called `runApp()`, which takes as its first argument the path to the directory where the app resides, and has different other parameters to set the host and the port where the web server should listen on. Users don’t need to know all this stuff, because the function automatically assigns a local host and a random port, unless specifically stated otherwise.

Which brings the attention over the important aspect of what a web “server” is. Most naive users think that a server is something which runs on the internet, in a public or a private mode. And this is true, but few users are aware of the fact that web servers can be run on the local computer, and the most common web address for a local web server is `127.0.0.1` which is the automatic default for the *shiny* package. The port number follows after a colon, making the web address of a Shiny app look similar to something like `127.0.0.1:6479`.

The *QCA* package offers a convenient wrapper function for all these details, and opening up the graphical user interface can be simply achieved with:

```
runGUI()
```


2.4.3 *Creating an Executable Icon*

While the user interface can be easily started from the R console after loading the *QCA* package, this approach suffers from a single drawback: the user's experience is now restricted to the web user interface for the entire session.

From the very moment the web application has been initiated, R will not do anything else but to listen for the commands generated by the point and click menus. It is therefore impossible to resume working at the R console while the user interface is active, because R accepts no other commands while in the communication state with the web server. It is a choice of either the web user interface, or the normal R console, but not both in the same time.

Fortunately, this can be circumvented thanks to R's internal operation procedures. As a command line interface software, R is primarily programmed to be started from a terminal console, usually from any kind of a terminal window. In the Unix environments (both Linux and MacOS), a terminal window is so natural that it doesn't need further explanation, and Windows users might be familiar with the DOS black window, and something similar can be started by entering the command `cmd` to the **Start/Run** menu.

In any of these terminal windows, provided that R has been installed with administrator privileges and the path to the binary executables have been added to the system paths, R can be opened by simply typing the command `R` (if installed to set its executable paths, or providing the complete path to the R executable file). The built-in R user interface/console is nothing but a shortcut to such a procedure, plus access to more advanced graphical devices.

Users can have two R sessions opened, one in the normal R console, and the other started in the terminal. Both can load the same package *QCA*, and one of them can be used to open the web user interface. While one of the sessions will be busy maintaining the web server communication, the other session can be used for any other R related task.

This kind of dual R usage can be further simplified by creating an executable file (icon) to start the terminal based R session and automatically open the web user interface, through a simple double click procedure. For the Linux users this is rather straightforward, involving the creation of a shell script and modifying its executable properties via the command `chmod`. While this is natural for the Linux environment, the rest of this section will provide detailed instructions for Windows and MacOS users.

In the Windows environment, an executable file can easily be obtained by creating a file having the extension ".bat". This can be achieved by opening any text editor and save an empty file having this extension. The content of this file should be the following:

```
CLS
```

```
TITLE QCA Qualitative Comparative Analysis
```

```
C:/PROGRA~1/R/R-3.3.3/bin/R.exe --slave --no-restore -e ^  
"setwd('D:/'); QCA::runGUI()"
```

The first two lines are harmless, cleaning up everything in the terminal window (if already opened) and setting a title. The third and more complex command requires some explanation because it involves a combination of commands, all related to each other.

It starts by specifying the path to the R executable, in this example referring to R version 3.3.3 but users should modify according to the installed version. The `C:/PROGRA~1` part simply refers to the `C:` drive and the shortcut to the Program Files directory (where usually all programs are installed), and the rest up to the `/bin/R.exe` completes the path to the R executable file, to start R.

The `--slave` part suppresses the prompts, commands, and R's starting information, while the `--no-restore` specifies that none of the objects created through this R session will be saved for later usage.

The `-e` part indicates to R that a series of commands are to be executed immediately after R has been started, and those commands are found between the double quotes. The `^` part signals that a long command is continued on the next line.

While the sequence `C:/PROGRA~1/R/R-3.3.3/bin/R.exe --slave --no-restore -e` are terminal specific commands, the sequence `setwd('D:/'); QCA::runGUI()` is a series of two R commands, separated by a semicolon.

Specifically, `setwd('D:/')` sets the working directory to the `D:/` drive (users might want to change that to a preferred directory of their own, see Sect. 2.4.8), while `QCA::runGUI()` indicates running the `runGUI()` function from the package *QCA* (the `::` part is an R feature, allowing the users to run a function from a specific package without having to load that package). The `runGUI()` function is responsible with the creation of the web page containing the user interface.

And that should be enough: with a bit of care to correctly specify the path to the executable R file, and setting the working directory to a place where the user has read and write permissions, saving this text file with a “.bat” extension will allow opening up the web user interface by a simple double-click on that file.

For MacOS users, the procedure is just as simple, although not immediately obvious because, unlike the Windows environment where a “.bat” file is automatically recognized as an executable file, in the Unix environment a file becomes executable only after specifically changing its properties.

The MacOS executable icon is also a text file, having a slightly longer extension called “.command”, and it contains the following:

```
R --slave --no-restore -e "setwd('/Users/jsmith/'); QCA::runGUI()"
```

It is basically the same command as in the Windows environment, only it doesn't need the complete path to the R executable because under MacOS the R installation process automatically sets these paths in the system.

The /Users/jsmith/ is the home directory of a hypothetical user called jsmith, and this is also something users want to customise.

Having this text file created, and named for instance QCA.command (assuming it was created on the Desktop, which is a directory found in the user's home directory), the only thing left to make it executable is to type this command to the Terminal window:

```
chmod u+x Desktop/QCA.command
```

This activates the executable bit of that file, and a subsequent double click will trigger the above mentioned behavior ending up with a web page containing a fresh instance of the user interface.

In Linux, a similar file can be created:

```
#!/bin/sh
R --slave --no-restore -e 'QCA::runGUI()' >/dev/null 2>&1 &
exit 0;
```

The `exit 0` command ensures the terminal window is closed when the R process stops.

2.4.4 Command Construction

Designing a graphical user interface for a command line driven software like R can be a daunting task. The variety of R commands is huge, and the same result can be obtained through many different ways. This is one of the reasons for which R is such a versatile and flexible data analysis environment.

A menu system, by contrast, is highly inflexible: computer programs don't choose the best or the most efficient set of commands for a given problem, they are designed with a single way to deal with every mouse interaction, making the user interface look like a prison. Users of traditional, point and click data analysis software (for example SPSS) many times believe the particular way that software presents the “solution” is a golden standard.

But that belief is very far from the truth, because any problem can be solved in hundreds of different ways, depending only on users' imagination. Some ways are more efficient, some are less efficient but easier to read by other peers, some are both efficient and easy to understand. By contrast, a point and click menu system offers only one possibility out of a potentially infinite number of ways to solve the same problem. It is the developer's responsibility to design intuitive ways to interact with the user interface, if possible some that are suitable for both advance and novice users.

One purpose of this graphical user interface is to facilitate using the *QCA* package, even for those users who don't necessarily know how to use R, however it should be very clear that a user interface, even an intuitive one, can never replace written commands.

To help users learn both *QCA* and R at the same time, one of the particularly nice features of this user interface is the way it automatically constructs R commands for any given dialog (Fig. 2.1).

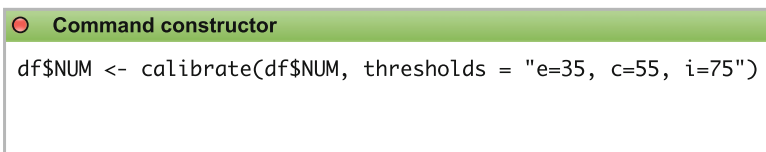


Fig. 2.1 The command constructor dialog

This kind of feature is not new, in fact it was employed for many years by John Fox in his R Commander user interface for R. What is different, and could be seen as an improvement over R Commander, is the way the command is constructed: interactively, upon each and every click on a specific dialog.

For novice users, it is extremely useful to see how a visual dialog to import a dataset (for example), is transformed into written commands, using various clicks to specify natural, but unknown parameters like what kind of column separator it has, decimal separator, which column refers to the case names etc.

The ultimate, long term goal is to avoid trapping users in a specific system of menus and dialogs, by demonstrating how to achieve the same thing with both clicks and written commands. There is a special "Command constructor" dialog which is always present (and cannot be closed), which displays how the command is constructed after every click on dialog options. Every such option (or combination of options) corresponds to a specific set of formal arguments from the *QCA* package functions, and hopefully users will understand how these commands are constructed.

This ultimate goal will have been achieved when users will eventually stop using the user interface completely, in favor of the more flexible written syntax:

it should be a transition phase towards written commands, but it can be used as a standalone application.

The command constructor is not editable: it automatically builds the commands from various clicks in the menu dialogs, but this is not (yet) a fully editable syntax editor. Future plans include extending this functionality, but for the moment it is only meant for display purposes.

2.4.5 The Web R Console

The first versions of the user interface offered only the menu system, with all communication between the webpage and R being rather hidden. While this approach was simple enough to be understood by novice R users, it suffered from a serious drawback: at some point, it becomes a burden to repeat dialog clicks in order to obtain similar truth tables or minimized solutions.

In the regular R session, users can create many objects, including truth tables, and also save the results of a Boolean minimization to a given object. It is not only to avoid repeating commands, but more importantly because these kinds of objects usually contain a lot of additional, useful information. For example, the minimization process involves creating and solving a prime implicants chart, which is not normally printed on the screen but can be inspected at a later stage to completely understand how the minimization was performed.

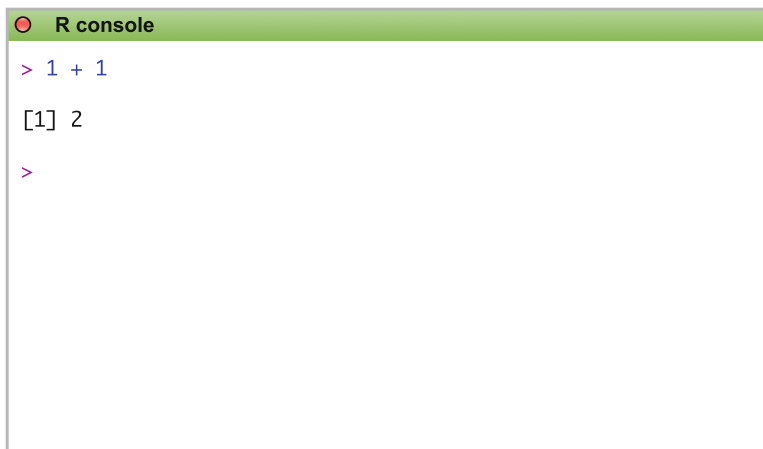
Other users might be interested to inspect the easy and difficult counterfactuals for a specific minimization (see Sect. 8.6), especially when directional expectations have been used (Sect. 8.7). All this information is returned into a single list object, out of which the internal function print only the solutions, eventually their inclusion and coverage scores. Saving that list to an object makes it possible to inspect it later, and this kind of interactivity is difficult to capture in a graphical user interface because there are more types of objects than what a user interface usually can display.

Newer versions of the user interface allow the possibility to create such objects directly from menus, however their inspection can only be performed from the command line R console. Along with the “Command constructor” dialog, the user interface has another mandatory dialog called the “R console”, with an attempt to mimic a regular R console and display the results of the minimization commands, or even to display truth table objects.

In the first version(s), this web based R console was not interactive: it was displaying what a normal R console would display, but users could not interact with those objects. A lot of effort was invested to make this interaction possible, in such a way that it would be as close as possible from a normal R

window. Everything that would normally be displayed in the normal R console is captured by the interface and printed on the webpage.

The newest versions of the user interface feature this interactive web R console, that supports typing commands (exemplified in Fig. 2.2) so that users can use menus and dialogs, as well as written commands, to produce and modify objects.

A screenshot of a web-based R console window. The window has a green title bar with a red close button and the text "R console". The main area is white and contains the following text: a prompt ">" followed by the command "1 + 1", a blank line, the output "[1] 2", another blank line, and a second prompt ">".

```
R console
> 1 + 1
[1] 2
>
```

Fig. 2.2 The R console dialog

Although it looks like the user interacts with R in a very direct way, in fact the information from the user interface is collected (either from dialogs or from the written commands) and sent to R as plain text. The communication is mediated by package *shiny*, which brings back the response and specialized Javascript code displays it either in the web R console or in various other dialogs, including plots.

The experience is dual: dialogs can create or modify objects, while written commands can re-create or modify various options in the dialogs. If the user creates a new variable in the R console, the variable will be immediately reported to the dialogs in the user interface, that get refreshed to display the new set of variables. This kind of reactive interactivity is valid for almost anything in the graphical user interface, from entire columns in the data to individual values that are displayed in various plot dialogs.

The web console behaves almost exactly like a normal R console, just in a distinct and separate evaluation environment. Objects that are created in the web environment do not exist in the original R environment that started the web interface, they are specific and unique to the web one. But otherwise practically all normal R commands are allowed, including `source()`: it reads the code within the sourced file and it interprets it just like the commands from the web R console, and the result is brought to the web environment.

2.4.6 Graphics

R is well known for the quality of its graphical devices, first developed at the Bell Laboratories (Murrell 2006). It has a rather sophisticated system of both low level and high level graphics, capable of drawing anything from simple statistical plots to complex images with pattern recognition.

As complex as it is, this system is fixed: once drawn, a graphic is like an image and to change something it needs to be recreated. It is possible to alter various components of the graphic, but it is still not an interactive system.

On the other hand, the web based graphics could not be more different. They might not be as sophisticated as R's graphical system, but they more than compensate through a lot of interactivity. A simple bar chart can behave differently, if only because the bars can be programmed to react on mouse over, or at click events: they change color, or get thicker borders, or display various information about a specific category etc.

Since R commands are possible through the web R console, it made sense to also capture graphics, in addition to printed output, errors and messages. There are related packages like *evaluate*, *knitr*, *raport* or *pander*, which parse source files to produce complex outputs (like reports, books or articles) and they can include plot results within the text. Inspired by the work in those packages, some effort was invested to bring the R graphics in the webpage environment. Without going into many technical details, it involves saving the plot from the base R to an SVG device, which can be read and displayed in the web environment.

And since an SVG can be resized on the fly, the resulting plot dialog can be resized and the graphic will automatically follow. Figure 2.3 is simple plot example obtained with the following commands which can be typed in the web user interface:

```
plot(1:11)
title("A simple plot")
abline(h = 6, v = 6)
```

Adding subsequent plot commands is possible (in this example adding a title and drawing the cross lines), because the code first verifies if there is a plot dialog open. If the plot dialog would be closed, any of the last two commands would trigger the R error that “plot.new has not been called yet”.

There are many attempts to extend the base R graphics and make them interactive, using either Java add-ins or RGtk2 with the *cairoDevice*, and there is even an implementation of an OpenGL in package *rgl* for 3D interactive plots (heavily used by the *Rcmdr* package). For some reasons, in the past 5 or 10 years, more results have been achieved with graphics in a webpage than for traditional software. Perhaps because of the unprecedented use of the

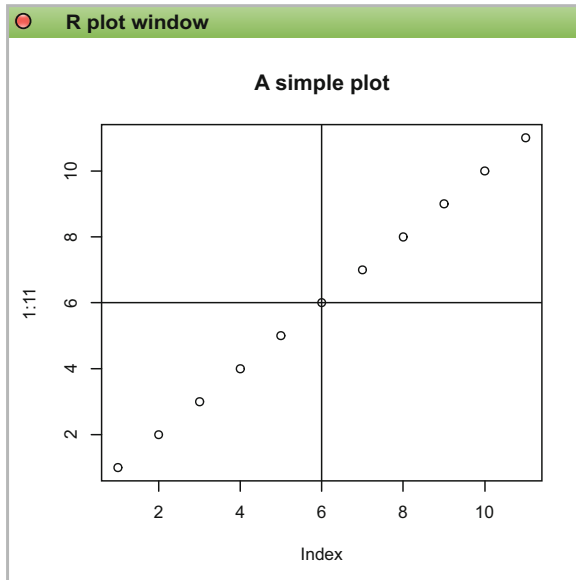


Fig. 2.3 The main plot dialog

web world that attracts a high number of web developers, there are Javascript libraries out there that can already rival traditional software and the trend shows no signs of a slow down.

The SVG—Scalable Vector Graphics standard has been extended and it can accommodate animations, morphing, easing and all sorts of path transformations based on matrix calculations. It is not surprising that libraries like D3.js have appeared, which seems to be ideal for data driven graphic displays.

Improved results have been recently presented at the RStudio conference in 2017, with yet another interactive 3D network visualisation tool of the future that uses a Javascript library called “threejs”, packed in a htmlwidget which can be opened everywhere there is a fairly recent web browser installed. This library can easily draw (and animate) force directed graphs with hundreds of thousands of points.

It is far from the goal of this user interface to completely replace R’s graphical system. Rather, the purpose is to enhance the traditional graphics system with more interactive options, specific to a webpage.

Web graphics are different from the base graphics, and are drawn separately in the web environment, reason for which the user interface presents three different types of plot dialogs:

- one for a normal R plots (captured from the base R),
- one for the XYplots, and
- one for the Venn diagrams (last two started from dedicated menus).

The first has already been discussed, allowing any kind of static graphics including *ggplot2* type of plots. In the XY plots, the points are allocated labels that are displayed at mouse-over events, and the labels can be displayed and rotated around the points. In the Venn diagrams, for every intersection in the truth table that has cases, the mouse-over event displays those cases.

The long term goal is to unify all these plot dialogs in a single environment, that would:

- automatically detect which type of plot is being used, and mediate different interaction possibilities for each
- offer various export options depending on each type of plot: for the interactive Javascript plots, it might necessary to export from SVG, or determine the best command that produces the most similar static plot.

All of the web-based graphics are drawn using an excellent SVG based library called “Raphäel.js”, entirely written in Javascript. In fact, all dialogs have been designed using this Javascript library, with custom code for all radio buttons, check boxes and every bit of interactivity.

The normal R plots can be saved on the hard drive in six most common formats (the base R has even more possibilities), using the menu (Fig. 2.4):

Graphs/Save R Plot Window

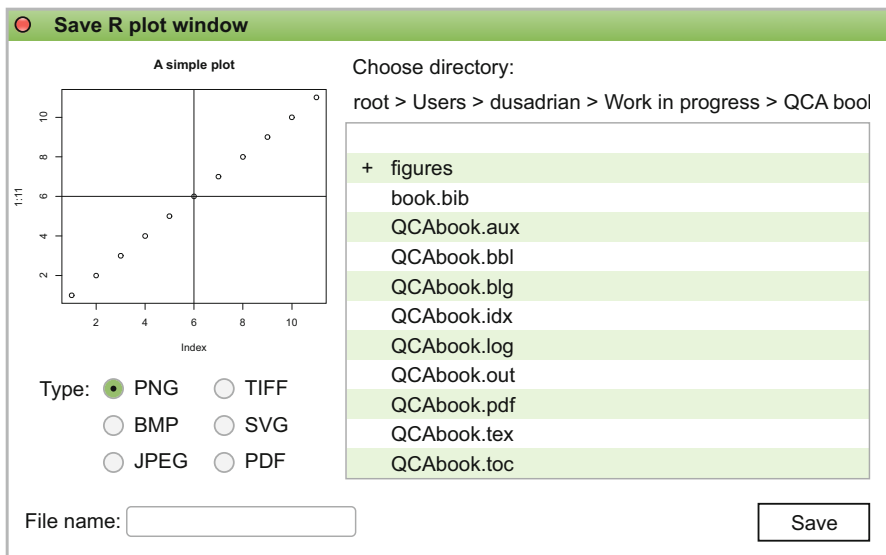


Fig. 2.4 The GUI dialog to export R graphics

2.4.7 *The Data Editor*

One highly appreciated feature of any graphical user interface seems to be a data editor. In principle, this kind of dialog is useful only for small datasets. For large or very large data (thousands of rows and hundreds of columns), it is obvious the data would not fit into the screen. Whatever the users “see” in the data editor, no matter how large the dialog, is only a fraction of the entire dataset, and the usual procedure is to move the visible area up and down, or left and right.

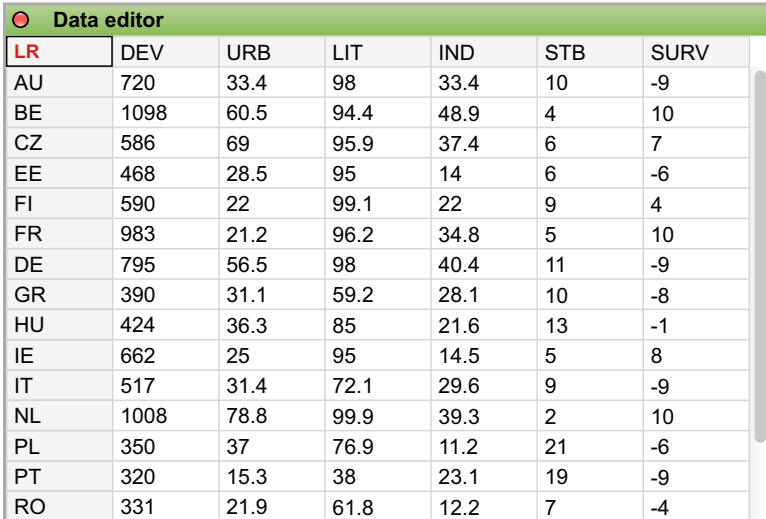
Such a dialog is difficult to develop in a web environment. Usually, HTML tables are split into small chunks, and users click on a “Next”, “Previous”, “First”, “Last” buttons, or any other chunk number to have the respective part displayed on the screen. This solution is understandable, because scrolling through the entire dataset is sometimes impossible because the dataset would have to first be loaded in the webpage before it can be scrolled.

Normal installed software can deal with this “loading” on the fly, working with local compiled code which is extremely fast, but a webpage relies on many other (slow) factors, for example the speed of the internet connection. The Javascript language can be extremely fast too, but it cannot beat a locally compiled code and it is not designed as a replacement. Its nature, that does something in a webpage, serves very different purposes.

Loading time is especially essential, for nobody wants to spend more than a couple of seconds before a webpage could be used. The traditional solution, therefore, is to split the data in smaller chunks that could be quickly delivered in the webpage. Faithful to this procedure, it is exactly the way RStudio displays data in most of their Shiny apps.

This inconvenience has been properly dealt with in the QCA user interface, which does not rely on classical HTML for display purposes. As mentioned in the previous section, it uses an SVG based, Javascript library called Raphaël.js that acts as a workhorse for the entire user interface. Instead of a regular HTML table, the dialog constructs a Raphaël “paper” (a concept similar to an HTML5 canvas), where it is possible to achieve scroll behavior by defining the full length and width of the paper without actually populating the dialog with the entire data.

Only the visible part is populated, and the rest is empty but ready to be populated when the user scrolls to another specific location. Figure 2.5 shows a vertical scrollbar on the right side, and the Javascript code behind detects a mouse-scroll event that is binded to the Raphaël paper. Whenever the paper is scrolled in any direction, the content of the cells is erased and the dialog starts a communication process with R to ask about the content from the new location. The behavior is very similar to a chunked HTML table (moving from chunk to chunk), while in this case it is moving from one location to another location (given by the new scroll).



LR	DEV	URB	LIT	IND	STB	SURV
AU	720	33.4	98	33.4	10	-9
BE	1098	60.5	94.4	48.9	4	10
CZ	586	69	95.9	37.4	6	7
EE	468	28.5	95	14	6	-6
FI	590	22	99.1	22	9	4
FR	983	21.2	96.2	34.8	5	10
DE	795	56.5	98	40.4	11	-9
GR	390	31.1	59.2	28.1	10	-8
HU	424	36.3	85	21.6	13	-1
IE	662	25	95	14.5	5	8
IT	517	31.4	72.1	29.6	9	-9
NL	1008	78.8	99.9	39.3	2	10
PL	350	37	76.9	11.2	21	-6
PT	320	15.3	38	23.1	19	-9
RO	331	21.9	61.8	12.2	7	-4

Fig. 2.5 The data editor dialog

This is one feature that brings the usage experience a step closer to a normally installed software. Custom code has been developed to make it look like a normal spreadsheet instead of an HTML table. On the entire content of the data editor dialog, there are click events that simulate the selection of a single cell, or a row name, or a column name (all of them can be considered “cells”).

There is also a double-click event to simulate editing the content of any given cell, by creating a temporary text box on top of the designated cell (at exact coordinates): once the text has been edited, the webpage detects the “Enter” or “Escape” events to move decide whether to commit the change or drop it. When committing a change, a new communication process is started with R to announce the change at that specific cell.

It might seem trivial, but this simple change has ripple effects throughout the entire user interface. If the change modifies the range of values (say, by changing a value from 100 to 1000), the new range is reported to any other editor which uses that variable (many such examples in the calibration chapter).

Finally, since the user interface has been upgraded to deal with multiple objects, it seemed natural to provide a quick way to select various data frames from the workspace. The top left corner of the dialog shows a button with the name “LR” written in red (the name of the dataset currently being displayed). A click on that button opens up a dropbox containing all loaded datasets, and selecting any of them will refresh the dialog with the new data.

When multiple datasets are loaded in the memory, they can be visualized in multiple ways. Either selecting from the dropbox in the top left corner, or if a dataset is small it can simply be printed on the screen, by typing its name

in the R console. The normal R has another way to display a dataset, using the function `View()`.

When typing a command such as `View(LR)` in the base R console, a separate window pops up where users are presented a very basic data browser. Under Windows the data can be scrolled up-down and left-right, while in MacOS it cannot even be scrolled, it looks as though it is a still image constructed in the graphics window. And this is a window designed for browsing data only: it cannot be modified in any way.

This is another reason to improve this functionality in the web interface, and more custom core has been produce to capture the normal R output and divert it to the web interface. Using the same command `View(LR)` in the web R console, the data will be displayed in the interactive web data editor, despite the command being evaluated in the “base” R, global environment.

2.4.8 Import, Export and Load Data

To bring the data file into R there are many alternatives, depending on the file’s type. There are many binary formats from other software, but one of the simplest way to exchange files, especially between different operating systems, is to use a text based file. Usually, these files have a `.csv` extension (comma separated value):

```
datafile <- read.csv("datafile.csv")
```

This command creates a new object called `datafile` in the workspace, combining the function `read.csv()` with the assignment operator “`<-`”.

When looking at this command, a first time R user might be puzzled, as it makes little sense and would likely fail on other computers. Most of the commands in this book can simply be copied and run on the local computer, and they should run just fine. But a command such as this one is bound to fail on a different computer, for two reasons. First, the file on the hard drive has to be named exactly “`datafile.csv`”, otherwise the user should change the name of the CSV file, and second because the local computer has zero knowledge about the location of the file.

As simple as it might seem, this scenario is potentially disruptive for the beginners. There are two solutions to this situation in base R, plus an additional one in the graphical user interface. The first basic solution is to set the working directory (a topic presented in Sect. 1.1) to the place where the CSV file is located:

```
setwd("/path/to/a/certain/directory")
```

The string `"/path/to/a/certain/directory"` should be changed according to the local file system (here similar to Linux and MacOS). It can be verified if the working directory was indeed changed to the desired location by typing the command `getwd()`.

The second solution, and probably one which should be preferred, is to specify the complete path to a specific file, an approach which is independent of the working directory settings above (the example below is specific to a Windows environment):

```
datafile <- read.csv("D:/data/datafile.csv")
```

`read.csv()` has some other formal arguments, compatible with those from the more general function `read.table()`. The user can specify whether the file has a different separator (often tab separated value, for example) via the argument `sep`, or indicate if the data file contains the variable names in the first row, via the argument `header`, and so on.

The extension `“.csv”` is not important, sometimes the text files have a `“.dat”` extension and rarely `“.tsv”`, the latest meaning tab separated values. The separator is the character that differentiates values from two different columns.

More details about all their arguments can be read using this command:

```
?read.table
```

The third solution is the most simple for all users, at any level. It requires no prior knowledge of R commands, and it uses interactive dialogs for navigating to a specific file, in a similar way to any normal graphical user interface. The dialog is opened using the menu:

File/Import

Figure 2.6 shows a minimalistic, yet comprehensive import dialog where users can navigate to a specific location in multiple ways. A first way is to double click the lines on the right side of the dialog. Directories are signaled with a plus `+` sign, and the parent directory is always the line with the two dots `..` at the top of the navigation area.

There are no “Back” (or “Forward”) button to select the previous directory, however above the navigation area there is a structure of directories that compose the current path. By double clicking any of the directories on that path, the navigation area changes with the content of that directory. If the path is very long (as in the figure), and it does not fit on the screen, it can be dragged left and right using the mouse.

Finally, a fourth possible way to navigate to a specific directory is to manually enter the path in the **Directory** textbox. This is very useful when working with USB flash disks, that are mounted to very specific places (or under

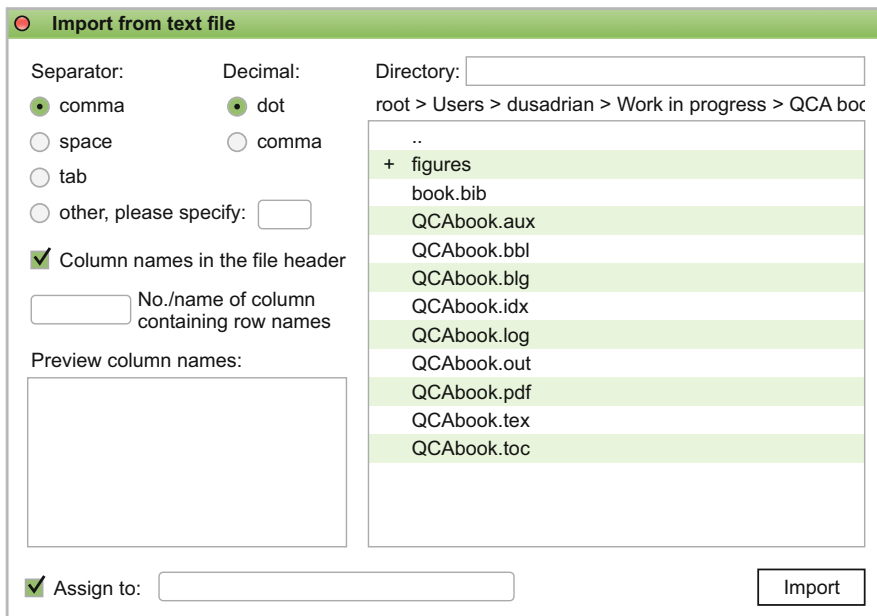


Fig. 2.6 The import dialog in the graphical user interface

Windows in very specific drives) that the web based user interface does not know anything about.

Once a specific CSV file has been selected, there is an area on the left (below the **Preview column names**) which is automatically filled with the names of the columns, if they exist in the header of the file. This is an extra measure to make sure the data will be correctly imported, before it is actually imported. Should all settings be correct according to the structure of the CSV file, the names of the columns are listed in that area.

Much like in the written command, there are various arguments to select one or another column separator. By default, this is a **comma** (hence the name of the CSV—Comma Separated Value file), but there are various other possibilities, the most frequently encountered being a **tab**, or a **space**. Any other separator (for instance a semicolon ; sign) should be manually entered in the **other, please specify:** textbox.

Another setting is the **Decimal** radio box, for files where the decimal values are separated with a dot (the default in the English speaking countries) or with a comma (as encountered for instance in the Eastern Europe and German speaking countries).

There is one final, optional setting to make, by specifying the **No./name of the column containing row names** textbox. If any of the columns from the datafile contains the names of the cases (or rows, as opposed to the names

of the variables, or columns), this option will place these names on the rows of the imported data. It can be either a number (the position of the column containing the row names) or even the actual name of the column containing these row names.

Before actually importing the data, a name should be given in the textbox next to **Assign** checkbox, to save the data into an R object in the workspace. Finally, a click on the **Import** button brings the data into R. This procedure mimics the written command, with various arguments of the function being manipulated via the dedicated options in the visual dialog.

All this procedure is needed to import custom data, but otherwise the examples from this book are exclusively based on the datasets that are already part of the *QCA* package. They can be loaded by simply typing the command `data()` with the name of the dataset, or use another dialog from the graphical user interface:

File/Load

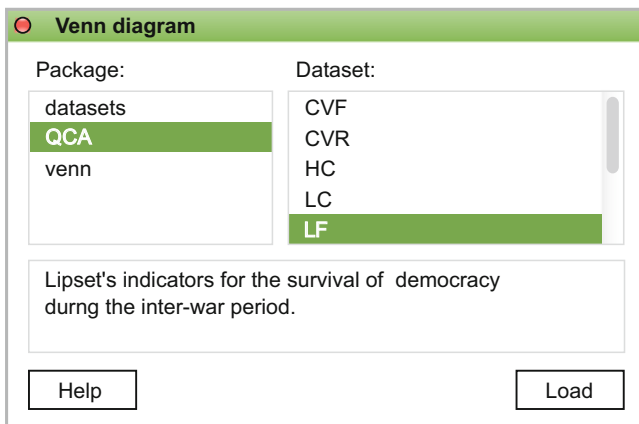


Fig. 2.7 Loading data from attached packages

In order to demonstrate functionality, many packages contain datasets needed to replicate the commands from the functions' help files. This is also very convenient, saving the users from having to import their data themselves. To access the datasets from a specific package, the package has to be loaded first, using the function `library()` and the name of the package.

Once this command is given, the package becomes *attached* and the functions inside (as well as the contained datasets) become available. In Fig. 2.7, there are only three attached packages (*datasets*, *QCA* and *venn*), and package *QCA* was selected. There are multiple datasets contained in this package, among which the selected one in the dialog (LR).

Once a specific dataset is selected, the box immediately below offers some basic information about the data. A complete information about the dataset (for instance what the name LR means, what columns does it contains, how were they calibrated, from which theoretical reference is the dataset taken from etc.) can be obtained by clicking on the **Help** button, which opens up a different webpage.

Data is usually modified during the analysis, and there is no “Save” button to make sure the modifications are preserved after the current session is closed, neither in command line nor in the graphical user interface. That is due to a particularity of R to maintain all objects in the memory (in the *workspace*, see Sect. 1.2), and they need not be saved as long as R is still opened. Only when closing R, all objects can be lost if they are not written to the hard drive.

The usual command to save a dataframe (or to export and save it to the hard drive) is `write.csv()`, which is a particular case of the general function `write.table()`. There are very many arguments to these functions, especially those which are specific to certain operating systems (for instance, the end of line code is different between Windows, Linux and MacOS). Also, there are various options to select a text encoding, because most of the languages have special characters (accents, diacritics etc.) which require a different encoding than the default one.

This book does not cover all of those options, and readers are encouraged to read about them in a general purpose book about R. Assuming that most of the default options are sufficient for the vast majority of users, only some of the arguments from `write.csv()` are going to be introduced.

The graphical user interface dialog, specific to exporting a dataset, is even more simple. It contains only the very basic options, which should be enough for most situations. There is a dedicated function in package *QCA* called `export()`, that is basically a wrapper for the base function `write.table()` with only one difference.

The main reason for this function is the default convention for the function `write.table()` to add a blank column name for the row names. Despite this (standard) convention, not all spreadsheet software can read this column properly, and many times the name of the first column in the dataset is allocated to the column containing the row names.

The function `export()` makes sure that, if row names are present in the dataset (in the sense they are different from the default row numbers) the exported file will contain a proper column name called *cases*. If the dataset already contains a column called *cases*, the row names will be ignored. Users are naturally free to choose their own name for the column containing the row names.

The dialog to export a dataset is opened with the menu (Fig. 2.8):

File/Export

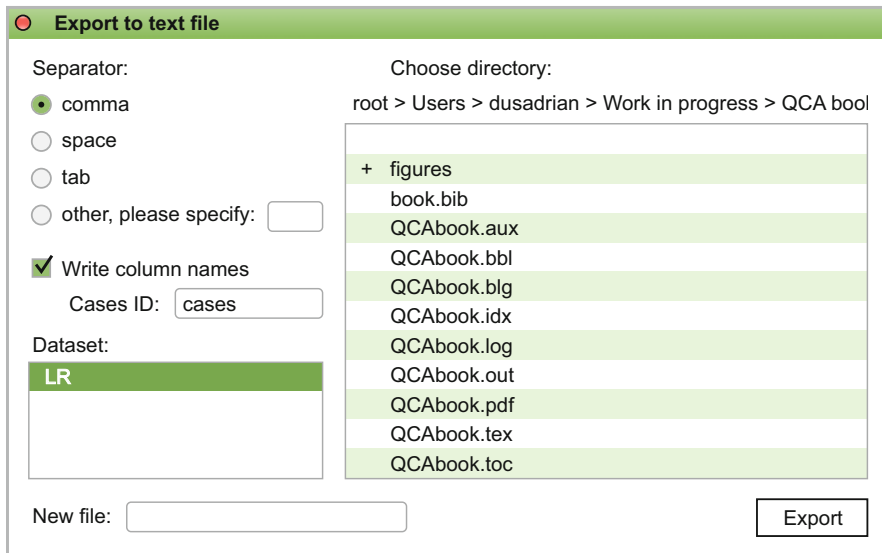


Fig. 2.8 Exporting a dataset from the graphical user interface

Since there is a single dataset in the workspace, it is the only object that appears in the area below **Dataset** in the left part of the dialog. The rest of the options should be self explanatory, from selecting the type of column separator to choosing a directory where to save the dataset to.

A name for the file should be provided in the **New file** textbox. By default, if a file with the same name exists in the selected directory, a checkbox will appear asking to choose if the file will be overwritten.

Chapter 3

Set Theory



QCA is a methodology based on sound theoretical foundations and robust software implementations. In the first version, Ragin (1987) presented a binary system which was dubbed as the “crisp” sets (csQCA), and following various critiques in the next years it was extended to fuzzy sets (fsQCA) with a pit stop through the “multi-value” variant (mvQCA).

Whatever the version, the term “sets” is extensively used throughout the QCA terminology, and its understanding is key to using the entire methodology. Although very common in the social sciences, this term has its roots in mathematics, where there are various types: set of real numbers, set of natural numbers etc.

In the social sciences, a set can be understood as a synonym for a category, and research methodology mentions many categorical variables such as Residence, where the category “Urban” can be understood as the set of people living in urban areas, and the category “Rural” as the set of people living in rural areas.

Fueled by the developments of the Qualitative Comparative Analysis in the recent years (Ragin 2000, 2008b) social research methodology witnesses an interesting theoretical duel. Emerging from the artificial opposition between the qualitative and quantitative research strategies (they approach the social world in very different ways), a new kind of conceptual competition captivates the community’s attention: the debate opposing variables and sets, with deep implications regarding measurement and interpretation.

3.1 The Binary System and the Boolean Algebra

The natural language can be transposed into a mathematical language, combining some fundamental laws of logic and a special type of language having only two values: *true* and *false*.

The history of these values dates back from the Chinese concepts of Yin and Yang, expressing a continuous duality in the nature. A similar line of thought was advanced by the philosopher and mathematician Gotfried Leibniz (from a biography written by Aiton 1985), who believed so much in the power of these symbols that led him to invent the binary mathematics.

Leibniz devoted his entire life to this system, which in his final years became almost religious where 1 represented the good and 0 represented the bad. Similar to the Chinese philosophy, Leibniz's world was a continuous struggle between good and bad, and he truly believed the binary system of mathematics had a divine origin.

His work was neglected for almost 150 years until the middle of nineteenth century, when another great mathematician named George Boole refined the binary system to become useful for logics, as well as mathematics. Both Leibniz and Boole were well ahead of their times, when the scientific community was not prepared for their work and failed to understand its use.

Boole's system was also neglected by his peers, until a few decades later the first real applications at the MIT—Massachusetts Institute of Technology in the United States.

3.2 Types of Sets

Towards the end of the nineteenth century, with an interest of some properties of the infinite series, the German mathematician and philosopher Georg Cantor created a theory of abstract sets which fundamentally changed the foundations of mathematics (Dauben 1979). His initial version (also known as the naive set theory) was later very much extended, and the modern set theory contains more axioms and types of sets than those described by Cantor. In summary, a set can be defined as a collection of objects that share a common property.

The objects inside the set are called elements, and each such element is unique. Mathematics has many set related properties and concepts: finite, infinite, conjunct, disjunct, equal etc. It can be noted that social science methodology, and especially comparative social science, uses a formal terminology with many terms borrowed from set theory, for example employing categories when segmenting human populations: rural/urban, males/females, lower/medium/upper levels of education etc.

Formally, there are two main types of sets:

- Crisp
 - bivalent
 - multivalent
- Fuzzy

The first family of crisp sets is Cantor’s original creation: an element is either inside the set or outside. It contains two subcategories: the bivalent sets with only two values, and the multivalent sets which can contain more than two values (no limit on the number of values, but all are discrete).

Despite the QCA terminology making a distinction between crisp sets (csQCA) and multi-value sets (mvQCA), in fact they are both “crisp” and what has been coined as “crisp sets” are in fact bivalent crisp which, as it will be shown, are a special case of multivalent crisp sets. Any bivalent crisp set can be specified as a multi-value set with only two values.

The fuzzy sets family is different because a fuzzy set can have an infinitely large number of possible values, and elements are not just in or out but more or less included in a given set, starting from the value 0 (completely out) to the value 1 (completely inside).

3.2.1 Bivalent Crisp Sets

The bivalent crisp sets are collections of well defined elements, having or not having a certain property, therefore belonging or not belonging in the set defined by that property. In a formal notation, any such set can be represented by enumerating all its distinct elements:

$$A = \{a_1, a_2, \dots, a_n\}$$

The set of all elements from all set is called the Universe (U), and its elements are $x_{1..n} \in U$. All other sets are subsets of this Universe.

For each element in the Universe it can be said “yes” (true) if the element belongs to a set A, and “no” (false) if it doesn’t. In the classical set theory, elements have only two values: 0 (false) and 1 (true). In formal notation, there is a function to attribute these two values:

$$\mu_A(x) = \begin{cases} 0 & \text{if } x \notin A \\ 1 & \text{if } x \in A \end{cases}$$

In the social sciences, crisp sets are also called mutually exclusive categories. A rose is part of the flowers set, while a horse does not belong to this set. One element can belong to multiple sets: a person can belong to the set of women and to the set of mothers.

For any two sets A and B from universe U, we can say that A is a subset of B if and only if any element that belongs to set A is also an element of the set B:

$$A \subset B : \{x \in A \Rightarrow x \in B\}$$

A set that does not contain any element is called the empty set (the notation is \emptyset), and this is a subset of any other set.

Starting from Aristotle, the origin of logics is bivalent, where any proposition has a single truth value: either true or false. This system has three principles (laws) of logics that constitute bivalent crisp sets:

- Principle of identity: an object is what it is (in other words, it is equal to himself). This principle makes the distinction between an object and all the others, similar to the psychology concept of “self” as compared with “others”
- Principle of non-contradiction: it is impossible for an object to exist and not exist at the same time, or for a phenomenon to both happen and not happen. It is either one, or the other, but not both.
- Principle of excluded middle: a proposition is either true or false, there is no third alternative.

3.2.2 Multivalent Crisp Sets

Traditional, scholastic logic is inherently bivalent, according to the principle of excluded middle. However, bivalency has been questioned by Aristotle himself, who formulated a paradox using propositional logic and a combination of logical expressions, testing bivalency in a temporal order situation.

Truth values can be attributed to past events: once it happened, we cannot say that a phenomenon did not happen because its truth value transcends time. After it happened, it becomes true at any point in time: immediately after, in the present and in the future.

The bivalent logic is compatible with the past (that we already know), but it is difficult to apply for the future, a situation which laid Aristotle’s paradox. Two thinking systems can be applied, with respect to the future:

1. Determinism: if something must happen, it will happen no matter what we do. This is similar to the concept of fatalism, the belief that everything is predetermined and inevitable, and we don’t have any control over our fate.

2. Free will: we decide what is going to happen, admitting the ability to choose what it happens (as well as what it does not happen), in the absence of any external constraint.

Any event has successive previous causal conditions, which have their own causal conditions, to the infinity until the beginning of time. If going to the future, as if it already happened, we will say the same thing about an event which didn't yet happen but is about to happen in the present, and that event will have had its own infinite chain of causes. In a deterministic world, an event is atemporal because, given all causal conditions which happened in the past, are happening in the present and will happen in the future, the event is necessarily bound to happen.

Aristotle made the statement: "Tomorrow, there will be a battle", for which it is impossible to attribute a truth value today, because it did not happen yet. However, applying the deterministic logic to the future (the causal chain inevitably leading to the event) negates the free will which today we know that it's true, thus breaking the principle of non-contradiction which states that something cannot be true and false in the same time.

Approaching the deterministic system, the Polish philosopher Jan Łukasiewicz created at the beginning of the twentieth century a logical system (Borkowski 1970) that goes past the traditional bivalent philosophy and offers a solutions to Aristotle's paradox. This system, noted with L_3 has not two, but three truth values:

$$\mu_A(x) = \begin{cases} 0 & \text{false} \\ 1/2 & \text{undetermined (neither true, nor false), partially true} \\ 1 & \text{true} \end{cases}$$

Dismissed at first, Łukasiewicz's philosophy was eventually accepted and led to the generalisation of his trivalent system to multivalent systems with n values. The truth values are obtained through a uniform division of the interval $[0, 1]$ into n distinct values:

$$\left\{ 0 = \frac{0}{n-1}, \frac{1}{n-1}, \frac{2}{n-1}, \dots, \frac{n-1}{n-1} = 1 \right\}$$

It can be seen that bivalent sets with only two values (0 and 1) are just a particular case of a multivalent crisp set with n values. The attribute "crisp" can be applied to any set where the elements are separated and distinct from one another.

3.2.3 Fuzzy Sets

Crisp sets are generally finite. At least in the social sciences, qualitative variables don't have an infinite number of categories. After the generalisation of Łukasiewicz's theory from 3 (\mathbb{L}_3) to n truth values (\mathbb{L}_n), it was only a question of time to a full extension of the theory towards an infinite number of possible values. In fact, the model \mathbb{L}_n is very close to the fuzzy sets, because n can be a very large number (close to, or equal to infinity).

Instead of dividing the space in n distinct values, another solution was proposed several decades later by the mathematician Lotfi Zadeh, who introduced the concept of "fuzzy sets".

In this original definition (Zadeh 1965, p. 338), these are:

... a class of objects with continuum of grades of membership. Such a set is characterised by a membership (characteristic) function which assigns to each object a grade of membership ranging between zero and one.

Between a minimum (nothing, zero) and a maximum (completely, one) there is a continuum, an infinite number of degrees of membership. An element can more or less belong to a set, not just inside or outside of the set. A population is not simply rich or poor, but somewhere in between. Function of the definition of welfare, a population belongs more in one set and less in the other, but it is difficult to imagine it strictly belongs into only one of them.

At this point, fuzzy set theory separates from the established social science methodology, where "rich" and "poor" are just the two opposite ends of the same continuum. It is a current practice to use bipolar scales (e.g. Likert type response scales) to measure attitudes and opinions on various levels of agreement/disagreement.

From the perspective of fuzzy sets, on the other hand, rich and poor are two separate sets, not just two ends of the same continuum. Each set has its own continuum of degrees of inclusion, so that a person (or a country) can be both rich and poor, in various degrees.

This is somewhat counter-intuitive from a quantitative, correlational point of view (where social reality is defined symmetrically from small to large) but it makes perfect sense from the perspective of set theory. It all relates to something called *simultaneous subset relations* that defines social reality as asymmetric, a topic which will be covered in more depth in Chaps. 5 and 6.

Another difference, and often source of confusion, is the apparent similarity between a fuzzy set and a probability. Although they both take values in the interval $[0, 1]$, they are in fact very different things. If a stove has a 1% probability of being very hot, there is still a chance to get severely burned when touching it, but if the same stove has a 1% inclusion in the set of very hot objects, we can touch and hold for as long as we want and it will still be harmless.

3.3 Set Operations

In the algebra that carries his name, Boole (1854) formulated three basic operations that are still very much in use today, extensively used in computer programming. These three can be applied for both crisp and fuzzy sets, just using different calculation methods.

3.3.1 Set Negation

This is probably the simplest possible operation, and the calculation method is similar to both binary crisp and fuzzy sets. Negation consists in finding the complement of a set A from the universe U , which is another set written as $\sim A$, formed by all other elements from the universe U that are not in A .

Negation is called a “unary” operation because it accepts only a single argument. In \mathbb{R} , there are various ways to negate either a truth value or a binary numerical one.

```
!TRUE
```

```
[1] FALSE
```

The “!” sign is interpreted as “not” and it negates any logical value. It can work on a scalar, and it also works on vectors.

```
lvector <- c(TRUE, TRUE, FALSE, FALSE)
!lvector
```

```
[1] FALSE FALSE TRUE TRUE
```

In this example, all values from the object `lvector` have been inverted. The same kind of operation can be performed subtracting the values from 1:

```
1 - lvector
```

```
[1] 0 0 1 1
```

Due to the automatic coercion of data types (in this case from logical to numeric), the true value become equal to 1 and those false become equal to 0, and the result is as simple as the mathematical subtraction from 1, which works for binary crisp values as well as for fuzzy values:

```
fvector <- c(0.3, 0.4, 0.5)
1 - fvector
```

```
[1] 0.7 0.6 0.5
```


Negation is a bit more complex for multivalent sets, but the essence is the same. For any such set with at least three values, say $\{0, 1, 2\}$, the complement of $\{0\}$ is the set containing the rest of the values $\{1, 2\}$. In the same fashion, the complement of $\{1\}$ is the set containing $\{0, 2\}$.

Negation is usually used with such numerical inputs but it doesn't necessarily be applied to numbers only. In natural language it can be applied just as easily using the expression "not". In terms of categories, an expression such as "not male" consists of all individuals who are not males. Other expressions can have logical implications, such as "not mother" which refers to all women who are not mothers, because the set of mothers is a subset of the set of women.

Neither the "!" operator, nor the "1 -" operation can automatically be applied to a multivalent set, because such a negation needs an additional information regarding the set of all possible values. Negating a certain value from a multivalent set has an unknown result in the absence of the complete information about all possible (other) values. To the limit, this information could be read from the input dataset, but there is no guarantee the input is exhaustive.

3.3.2 Logical AND

This operation is also called a "logical conjunction" (or simply a conjunction), and it takes a true value only when all its elements are true. If any of the elements is false, the whole conjunction will be false.

In natural language, we may say students who have high grades are intelligent and study hard. Non intelligent students don't have high grades, just as intelligent students who don't study hard. Only when both attributes are met, both intelligent and study hard, the grades are high.

This can be exemplified with logics, the result of the logical AND operation is true only when all conditions are true:

0 AND 0 = 0	0 * 0 = 0
0 AND 1 = 0	0 * 1 = 0
1 AND 0 = 0	1 * 0 = 0
1 AND 1 = 1	1 * 1 = 1

There are various combinations of vectors in \mathbb{R} to exemplify, one of the specific functions is `all()`:

```
all(lvector)
```

```
[1] FALSE
```

The result of the function `all()` is false because at least one of the values of `lvector` is false. Subsetting for the first two:

```
all(lvector[1:2])
```

```
[1] TRUE
```

In this case, the result is true because both first two values of `lvector` are true.

Another R operator that refers to conjunctions (intersections) is the ampersand sign “&”, and the example can be further extended with combinations of logical vectors:

```
rvector <- c(TRUE, FALSE, TRUE, FALSE)
lvector & rvector
```

```
[1] TRUE FALSE FALSE FALSE
```

Here, the result of the “&” (logical AND) operation is a vector of length 4, comparing each pair of the values from `lvector` with those of `rvector`, and the resulting values are true only when both values from `lvector` and `rvector` are true, in this case only the first pair.

The R specific recycling rule can also be applied:

```
rvector <- c(FALSE, TRUE)
lvector | rvector
```

```
[1] TRUE TRUE FALSE TRUE
```

In this example, the shorter `rvector` (of length 2) has been recycled to reach the length 4 of the longer `lvector`, and the final result shows a true value on the second position because only the second values from both vectors are true. Recycling didn’t matter in this case, because the third and fourth values from `lvector` are both false.

The logical AND is also useful to make data subsets based on various criteria. Using the data frame `dfobject` from Sect. 1, which has four rows, the following can be applied:

```
subset(dfobject, A == "rural" & B > 13)
```

```
   A  B C D
C3 rural 14 7 3
C4 rural 15 8 4
```

In this example, only two cases (C3 and C4) conform to both conditions that A is rural and B is greater than 13. Both these conditions generate logical vectors that are combined with the “&” (AND) sign, and the resulting logical vector is used to select rows from `dfobject`.

As already mentioned, another interpretation of the logical AND is that of an intersection. When A is the set of intelligent students and B the set of students who study hard, then intelligent students who study hard are found at the intersection between A and B.

The base R has a function `intersect()` that intersects two sets:

```
lvector <- c(1, 3, 5, 6)
rvector <- c(2, 4, 5, 6)
intersect(lvector, rvector)
```

```
[1] 5 6
```

The fuzzy version of the logical AND has a different calculation method, because the set elements have values anywhere in the interval $[0, 1]$, that is each element has a membership value in a specific set.

A person may have a 0.8 membership score in the set of intelligent students, and 0.3 in the set of students who study hard, and the logical operation is obtained by taking the minimum of those membership scores:

$$A \cap B = \min(0.8, 0.3) = 0.3$$

The result is a membership score in the set of “A and B”, equal to the minimum between the two membership scores in the component sets. Zadeh (1965, p. 341) calls it: “the largest fuzzy set which is contained in both A and B”.

The **QCA** package has a function called `fuzzyand()` that is similar to the built-in function `pmin()` that performs parallel minima, with some improvements. It accepts vectors, and also data frames and matrices for which it applies a `min()` function on the rows, thus simulating a parallel minima on a rectangular object.

In addition, the result value of `fuzzyand()` is automatically given an attribute called “name”, which is the formal conjunctive notation of the result. To achieve this, another improvement is the automatic detection of negated inputs using the usual “1 -” notation:

```
A <- c(0.7, 0.2, 0.4)
B <- c(0.2, 0.4, 0.5)
result <- fuzzyand(A, 1 - B)
result
```

```
[1] 0.7 0.2 0.4
```

Detection of negations becomes visible when inspecting the name attribute:

```
attr(result, "name")
```

```
[1] "A*b"
```

3.3.3 Logical OR

Logical OR is also called a “logical disjunction” (or simply a disjunction) and refers to any alternative way to achieve a result. The logical OR is more strict than the natural language word “or”, which has many other interpretations.

In the natural language it may sometimes refer to exclusively to one, or the other, but not to both in the same time: “You can get to your workplace taking a taxi *or* taking a bus”.

Some other times it may also refer to any option (inclusive), even both happening at the same time: “That person lost weight, so must have kept diet *or* made a lot of sports”, where these options don’t exclude each other.

In logics, if any condition is true (even both at the same time), the result of the logical OR operation is true:

0 OR 0 = 0	0 + 0 = 0
0 OR 1 = 1	0 + 1 = 1
1 OR 0 = 1	1 + 0 = 1
1 OR 1 = 1	1 + 1 = 1

These kinds of relations may be exemplified using various combinations of logical vectors in R, having an immense applicability potential, especially for indexing and subsetting purposes.

```
lvector <- c(TRUE, TRUE, FALSE, FALSE)
any(lvector)
```

```
[1] TRUE
```

In the example above, the object `lvector` is a logical with four values, first two being true. The function `any()` returns true if any of those four values is true, and it can be rewritten in natural language as: first is true, or second is true, or third is true, or fourth is true. The final result is `TRUE`, because at least one of those four values is true.

The example above can be further extended with combinations of logical vectors:

```
rvector <- c(TRUE, FALSE, TRUE, FALSE)
lvector | rvector
```

```
[1] TRUE TRUE TRUE FALSE
```

In this example we have two logical vectors, combined with the “|” (logical OR) sign, the result being another logical vector of length four, where only the last value is `FALSE` because it is false in both input vectors.

The R specific recycling rule can also be applied:

```
rvector <- c(FALSE, TRUE)
lvector | rvector
```

```
[1] TRUE TRUE FALSE TRUE
```

Here, the shorter vector `rvector` has been recycled at the length of the longer vector `lvector`, and this time the third value is `FALSE` because when `rvector` has been recycled, its third value becomes false and the third value in `lvector` is also false.

Just as the previous two logical operations, the logical OR can also be useful to make data subsets based on various criteria. Using the same data frame `dfobject`:

```
subset(dfobject, B > 13 | C > 5)
```

```
      A  B C D
C2 urban 13 6 2
C3 rural 14 7 3
C4 rural 15 8 4
```

Here, there are three rows which conform to both expressions. The row “C2” was preserved into the subset even though it was not true for the expression `B > 13`, however it was true for the other expression `C > 5`.

Another interpretation of the logical OR is a union of two or more sets, the result being another set containing all unique elements from the other sets.

In R there is a function `union()` that does precisely that thing:

```
lvector <- c(1, 3, 5, 6)
rvector <- c(2, 4, 5, 6)
union(lvector, rvector)
```

```
[1] 1 3 5 6 2 4
```

The resulting union has all the unique values from both `lvector` and `rvector`, in the order they first appear as unique.

The fuzzy version of the logical OR also implies values in the interval $[0, 1]$. If a person has a 0.8 membership score in the set *A* of intelligent students, and 0.3 in the set *B* of students who study hard, the logical operation is obtained by taking the maximum of those membership scores:

$$A \cup B = \max(0.8, 0.3) = 0.8$$

The result is a membership score in the set of “either *A* or *B*”, equal to the maximum between the two membership scores in the component sets. Zadeh (1965, p. 341) calls it: “the smallest fuzzy set containing both *A* and *B*”.

The *QCA* package has a corresponding function called `fuzzyor()`, similar to the built-in function `pmax()` (from parallel maxima), with an added feature to automatically detect the negated inputs using the “1-” notation:

```
A <- c(0.7, 0.2, 0.4)
B <- c(0.2, 0.4, 0.5)
result <- fuzzyor(1 - A, B)
result
```

```
[1] 0.3 0.8 0.6
```

Similar to the sibling function `fuzzyand()`, there is a `name` attribute which contains the string expression corresponding to the input vectors. By default, the expression is constructed using a lower/upper case notation unless the object names already contain lower case letters, a situation when the negation is signalled with a tilde sign.

Enforcing the tilde sign is possible by activating the argument `use.tilde`:

```
result <- fuzzyor(1 - A, B, use.tilde = TRUE)
attr(result, "name")
```

```
[1] "~A + B"
```

The same result can be obtained by directly negating the object using a tilde:

```
attr(fuzzyor(~A, B), "name")
```

```
[1] "~A + B"
```

3.4 Complex Operations

The examples presented in the previous section are demonstrative only, referring either to single sets or at most two sets. Most commonly however, expressions can involve more sets with various combinations of all three basic operations of set negation, logical union and logical intersection.

The functions `fuzzyand()` and `fuzzyor()` themselves, although capable of handling any kind of complex example, are also mainly used for didactic purposes. The package *QCA* has yet another function which can replace both them, and perform even more complex operations. The final example in the last section can be re-written as:

```
compute("~A + B")
```

```
[1] 0.3 0.8 0.6
```

In these kinds of string based expressions, the tilde sign “ \sim ” can be used to negate a condition, then logical union is signaled with a plus sign “ $+$ ”, while the intersections are most of the times signaled with a star sign “ $*$ ”, excepting the situations when the expression is multi-value (when the star sign is redundant) or when the set names are taken from a dataset.

The function `compute()` is versatile enough to search for the input conditions A and B, either within the list of objects created in the user’s workspace, or within the columns of a dataset specified with the argument `data`:

```
data(LF)
compute("DEV*STB + ~URB*~LIT", data = LF)
```

```
[1] 0.43 0.98 0.58 0.16 0.58 0.95 0.31 0.87 0.12 0.72 0.59 0.98 0.41
[14] 0.98 0.83 0.70 0.91 0.98
```

The same result would be obtained using a combination of two `fuzzyand()` and one `fuzzyor()`, but feeding a sum-of-products (SOP) expression to the function `compute()` is many orders of magnitude more simple.

Complex expressions can be simplified by applying a few simple Boolean rules:

$$\begin{aligned} A \cdot A &= A \\ A \cdot A \cdot B &= A \cdot B \\ A + A \cdot B &= A \\ A + \sim A &= 1 \\ A \cdot \sim A &= \emptyset \\ A \cdot \emptyset &= \emptyset \end{aligned}$$

In particular, negations are very effective applying DeMorgan’s rules:

$$\begin{aligned} \sim(A + B) &= \sim A \cdot \sim B \\ \sim(A \cdot B) &= \sim A + \sim B \end{aligned}$$

Such simplifications are automatically applied by the function `sop()`, for instance on some of Ragin’s examples from his 1987 original book. The first is an intersection between the developmental perspective theory (Lg) and the resulting equation for the outcome E, ethnic political mobilization (page 146):

```
sop("Lg(SG + LW)", snames = "S, L, W, G", use.tilde = TRUE)
```

```
[1] "L*W*~G"
```

A more complex example shows the subnations that exhibit ethnic political mobilization (E) but not hypothesised by any of the three theories (page 147):

```
sop("(SG + LW)(GLs + GLw + GsW + lsw)", snames = "S, L, W, G")
```

```
[1] "S*L*w*G + s*L*W*G"
```

The simpler expression can be used as input for the function `compute()`, with an identical result as the one for the complex expression.

Chapter 4

Calibration



The usual QCA data is numeric, and has specific formats for each flavour: when crisp (either binary or multi-value) the data consists of integers starting from the value of 0, and when fuzzy the values span over a continuous range, anywhere between 0 and 1.

Not all data from scientific research conform to these formats. In fact, one can actually expect the contrary, that most of the times the raw data has a different shape than expected to perform QCA. There are numeric variables of all types and most importantly of all ranges, and there are qualitative variables that separate cases into categories etc.

Calibration is a fundamental operation in Qualitative Comparative Analysis. It is a transformational process from the raw numerical data to set membership scores, based on a certain number of qualitative anchors or thresholds. The process is far from a mechanical transformation, because the choice of the calibration thresholds is a theoretically informed one and dramatically changes the result of the calibration process.

Although this book is more about the practical details on how to perform QCA specific operations with R, it is still important to cover (at least in a brief way) the theoretical concepts underpinning these operations.

Typically, social science research data are separated into four levels of measurement: nominal, ordinal, interval and ratio. Virtually all of them can be calibrated to binary crisp sets, almost but not all can be calibrated to multi-value crisp sets, while in the case of fuzzy sets only the “interval” and “ratio” levels of measurement can be used. However, the concept of “calibration” is something different from the concept of “measurement” (Ragin 2008a,b).

When thinking about “measurement”, the social science researcher arranges the values of a variable in a certain order (where some of the values are smaller/higher than others), or calculate their standardized score by

comparing all of them with the average of the variable: some will have positive scores (with values above the average), and some will have negative scores (values below the average).

While this approach makes perfect sense from a mathematical and statistical point of view, it tells almost nothing in terms of set theory. In the classical example of temperature, values can be arranged in ascending order, and any computer will tell which of the values are higher than others. But no computer in the world would be able to tell which of the temperature values are “hot”, and which are “cold”, only by analysing those values.

Mathematics and statistics usually work with a sample, and the derived measurement values are sample specific. The interpretation of the values, however, need more information which is not found in the sample, but in the body of theory: “cold” means when the temperature approaches 0°C (the point where the water transforms into ice) and “hot” means when the temperature approaches 100°C (the point where the water transforms into vapours). This kind of information is not found in the sample, but outside.

The same can be said about any other numerical variable. Considering people’s heights, for instance, computers can tell which person has a higher height the other(s), but it could not tell what it means to be “tall” and what it means to be “short”. These are human interpreted, abstract concepts which are qualitative in nature, not quantitative, and most importantly they are culturally dependent (a tall person in China means a very different thing than a tall person in Sweden).

In the absence of these humanly assessed qualitative anchors, it is impossible for a computer (or a researcher with no other prior information) to derive qualitative conclusions about the values in the sample.

The first section in this chapter is dedicated to crisp calibrations, showing that it is actually improper to use the term “calibration” for crisp sets, since they involve a simple recoding of the raw data instead of a seemingly complicated calibration process.

The second section is devoted to the *proper* fuzzy calibration, including various types of “direct” calibration and also the “indirect” calibration. Multiple methods are available for the direct method of calibration, depending of the definition of the concept to be calibrated, and package *QCA* is well equipped with a complete set of tools to deal with each such situation. The indirect method of calibration is very often misinterpreted with a direct assignment of fuzzy scores, but it is much more than that.

The final section is dedicated to calibrating categorical data, as an attempt to clarify some aspects regarding the meaning of the word calibration and how the final outcome of the calibration process depend heavily on the type of input data.

4.1 Calibrating to Crisp Sets

When using only one threshold, the procedure produces the so called “binary” crisp sets dividing the cases into two groups, and if using two or more thresholds the procedure produces the “multi-value” crisp sets. In all situations, the number of groups being obtained is equal to the number of thresholds plus 1.

All previously given examples (temperature, height) imply numerical variables, which means the concept of calibration is most often associated with fuzzy sets. That is a correct observation, because the process of calibration to crisp sets is essentially a process of data recoding. As the final result is “crisp”, for binary crisp sets the goal is to recode all values below a certain threshold to 0, and all values above that threshold to 1 (for multi-value crisp sets, it is possible to add new values for ranges between two consecutive thresholds).

As in all other sections and chapters, wherever there are options for both command line and graphical user interface, the demonstration starts with the command line and the user interface will follow in close synchronization.

To exemplify this type of calibration, a well known data from the Lipset (1959) study is going to be loaded into the working space via this command:

```
data(LR)
```

There are four versions of the Lipset dataset included in the *QCA* package: LR (the raw data), LC (calibrated to binary crisp sets), LM (calibrated to multi-value crisp sets) and LF (calibrated to fuzzy sets). The description of all columns, including the outcome, can be found via the command `?LR`.

This example concentrates in the column `DEV`, referring to the level of development as GNP per capita, measured in US dollars (values for 1930), which spans from a minimum value of 320 and a maximum value or 1098:

```
sort(LR$DEV)
```

```
[1] 320 331 350 367 390 424 468 517 586 590 662 720 795
[14] 897 983 1008 1038 1098
```

Before determining what an appropriate threshold is, to separate these values into two groups (0 as not developed, and 1 as developed to create a binary crisp set), it is always a good idea to graphically inspect the distribution of points on a horizontal axis.

The graphical user interface has an embedded threshold setter in the dialog for the calibration menu, and there are various ways to create a similar plot via command line, using for example the function `plot()`, but the simplest would be to use the dedicated function `Xplot()` that inspects just one variable, with a similar looking result as the threshold setter area in the user interface (Fig. 4.1):

```
Xplot(LR$DEV, at = pretty(LR$DEV), cex = 0.8)
```

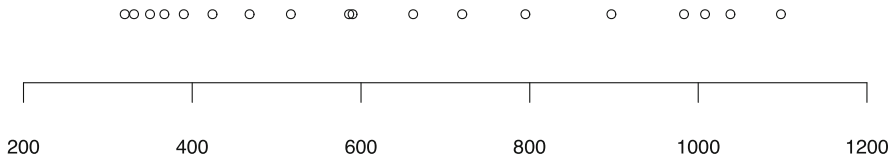


Fig. 4.1 The distribution of DEV values

This particular plot has sufficiently few points that don't overlap much, but if there are many overlapping points the function has an argument called `jitter` which can be activated via `jitter = TRUE`.

In the absence of any theoretical information about the what “development” means (more exactly, what determines “high” or “low” development), one approach is to inspect the plot and determine if the points are grouped in natural occurring clusters. It is not the case with this distribution, therefore users can either resort to finding a threshold using a statistical clustering technique, or search a relevant theory.

In the *QCA* package there is a function called `findTh()` which employs a cluster analysis to establish which threshold values best separates the points into a certain number of groups. To separate into two groups, as explained in Sect. 2.3.2, no other additional parameters are needed because the number of thresholds (argument `n`) is by default set equal to 1. The command is:

```
findTh(LR$DEV)
```

```
[1] 626
```

The value of 626 was found by a complete hierarchical clustering, using the Euclidean distance (see the default values of arguments `hclustm` and `dism`). However, Rihoux and De Meur (2009) have decided to use a close but different threshold value of 550 USD for their binary crisp calibration. Initially, they have used a threshold of 600 USD but upon closer inspection during their analysis, they have found that a value of 550 USD accounts for a natural ‘gap’ in the distribution of values and better “differentiates between Finland (590 USD) and Estonia (468 USD)”.

If not for the threshold value of 550 USD, Finland and Estonia would fall in the same category of development, and there is clearly a difference between the two neighbouring countries. This is a fine example of how theory and practical experience are used as a guide to establish the best separating line(s) which define the crisp groupings of cases. This is a qualitative assessment originating from outside the dataset: it is not something derived from the raw data values, but from external sources of knowledge, and makes a perfect example of the difference between “calibration” and raw data “measurement”.

The final, calibrated values can be obtained with two different methods. The first is to use the `calibrate()` function and chose the `type = "crisp"` argument (the default is `"fuzzy"`):

```
calibrate(LR$DEV, type = "crisp", thresholds = 550)
```

```
[1] 1 1 1 0 1 1 1 0 0 1 0 1 0 0 0 0 1 1
```

There are other arguments in the function `calibrate()`, but all of those refer to the fuzzy type calibration. In the crisp version, these two arguments are the only necessary ones to obtain a calibrated condition.

The crisp version of calibration is essentially equivalent to recoding the original raw data to a finite (and usually very low) number of crisp scores. Therefore a second method, which will give exactly the same results, is to use the `recode()` function from package *QCA*, using the following command:

```
recode(LR$DEV, rules = "lo:550 = 0; else = 1")
```

```
[1] 1 1 1 0 1 1 1 0 0 1 0 1 0 0 0 0 1 1
```

The syntax of the `recode()` function is very simple, having only two formal arguments: `x` and `rules`, where the first is the initial raw vector of data to be recoded, while the second is a string determining the recoding rules to be used. In this example, it can be translated as: all values between the lowest `lo` and 550 (inclusive) should be recoded to 0, and everything else should be recoded to 1.

Calibrating to multi-value crisp sets is just as simple, the only difference being the number of thresholds `n` that divide the cases into `n + 1` groups:

```
findTh(LR$DEV, n = 2)
```

```
[1] 626 940
```

The clustering method finds 626 and 940 as the two thresholds, while Cronqvist and Berg-Schlösser (2009) used the values of 550 and 850 USD to derive a multi-value causal conditions with three values: 0, 1 and 2:

```
calibrate(LR$DEV, type = "crisp", thresholds = "550, 850")
```

```
[1] 1 2 1 0 1 2 1 0 0 1 0 2 0 0 0 0 2 2
```

The argument `thresholds` can in fact be specified as a numerical vector such as `c(550, 850)`, but as it will be shown in the next section, when calibrating to fuzzy sets this argument is best specified as a named vector and its most simple form is written between two double quotes. This is an improvement over the former specification of this argument, but both are accepted for backwards compatibility.

Using the `recode()` function gives the same results:

```
recode(LR$DEV, rules = "lo:550 = 0; 551:850 = 1; else = 2")
```

```
[1] 1 2 1 0 1 2 1 0 0 1 0 2 0 0 0 0 2 2
```

This specification of the argument `rules` assumes the raw data are discrete integers, but in fact the `recode()` function has another specification inspired by the function `cut()`, which works with both discrete and continuous data. This other method uses a different argument named `cuts` (similar to the argument `breaks` from function `cut()` and also similar to the function `thresholds` from function `calibrate()`), to define the cut points where the original values will be recoded) and a related argument named `values` to specify the output values:

```
recode(LR$DEV, cuts = "550, 850", values = 0:2)
```

```
[1] 1 2 1 0 1 2 1 0 0 1 0 2 0 0 0 0 2 2
```

As mentioned, there are various dialogs in the graphical user interface to match these commands. The calibration dialog is one of the most complex from the entire user interface, and Fig. 4.2 shows the one appearing after selecting the menu:

Data/Calibrate

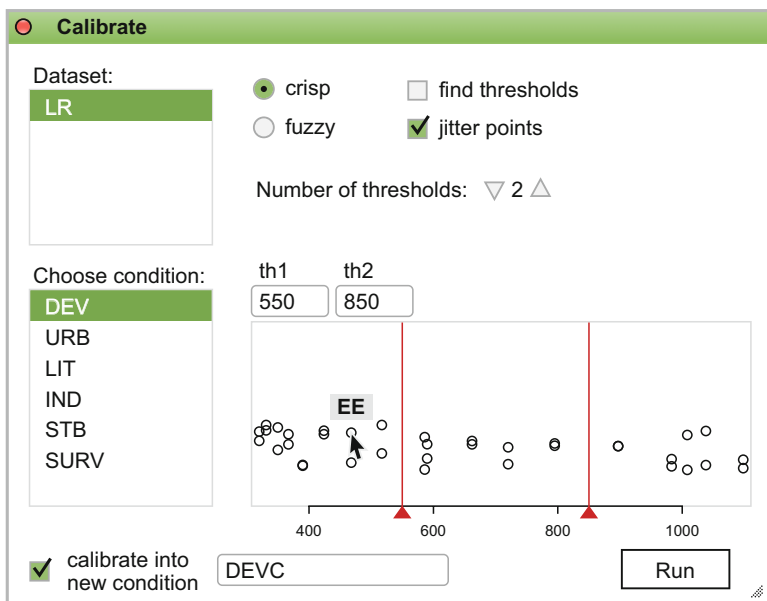


Fig. 4.2 The crisp “Calibrate” dialog

The procedure to use this dialog is very straightforward and involves a number of intuitive steps:

1. select the dataset from the list in the **Dataset** area: in this example a single dataset is loaded, but R can work with any number of datasets in the same time (one thing to notice, if there is a single dataset loaded in R, it will automatically be selected by the interface)
2. select the condition from the list under the **Choose condition** area, with the immediate effect of the distribution of values appearing in the threshold setter area
3. choose **crisp** from the radio button pairing with fuzzy (this is the equivalent of the argument **type** in the written command)
4. if threshold values are to be suggested by the computer, check the **find thresholds** checkbox; it has no direct equivalent with the arguments of the `calibrate()` function, but it is using the `findTh()` function behind the scenes
5. if points are too densely clustered, check on the **jitter points** checkbox to scatter the points vertically with small random values
6. adjust the number of thresholds via the down or up buttons
7. whether or not asking the computer for thresholds, their values can be manually (over)written in the text boxes right above the plot area
8. independently of manual or automatic specification of the thresholds values, their correspondent vertical red bars in the plot area can be manually dragged left or right, and the text boxes from step 7 will change accordingly
9. if the calibrated values should be saved as a different variable in the same dataset, check the **calibrate into new condition** and specify the new (calibrated) condition name (otherwise it will overwrite the same condition with the new values)
10. click the **Run** button and the new column should appear in the dataset, visible either in the console or in the data editor.

Figure 4.2 presents a situation where the condition DEV from the dataset LR is calibrated to multi-value crisp sets using two thresholds (550 and 850) which are manually set (the “find thresholds” option is unchecked), with points jittered vertically to avoid overlapping.

Since the user interface is developed into a webpage, it makes sense to use all the advantages of this environment. The points have a “mouse-over” property, and respond with the label of the point (the row name of that particular case), in this example displaying EE (Estonia), a country from the dataset LR.

The dialog allows up to six thresholds for the crisp type, dividing a causal condition in at most seven groups. This is a limitation due to the lack of space in the dialog, but otherwise the command line can specify any number of thresholds. Cronqvist and Berg-Schlusser (2009) have given a fair number of practical advice in order to decide for how many thresholds should be set. Apart from the already mentioned guides (to look for naturally occurring clusters of points, and employing theoretically based decisions), one other

very good advice is to avoid creating large unbalances in the group sizes, otherwise solutions will possibly be too case specific (finding solutions that explain exactly 1 or 2 cases, whereas a scientifically acceptable result should allow more general solutions, at least to some degree).

This particular distribution of data points are rather clearly scattered, but other datasets can have hundreds of overlapping points, a situation when the area dedicated for the thresholds setter will prove to be too small even if points are jittered. However, this is not a problem for an interface designed into a webpage: unlike traditional user interfaces where dialogs are fixed, this particular interface is designed to be responsive, reactive and above all interactive.

Notice the small handling sign in the bottom right corner of the dialog (all resizable dialogs have it), which can be used to enlarge the dialog's width to virtually any dimension until the points will become more clearly scattered for the human eye. The result of such dialog enlargement can be seen in Fig. 4.3.

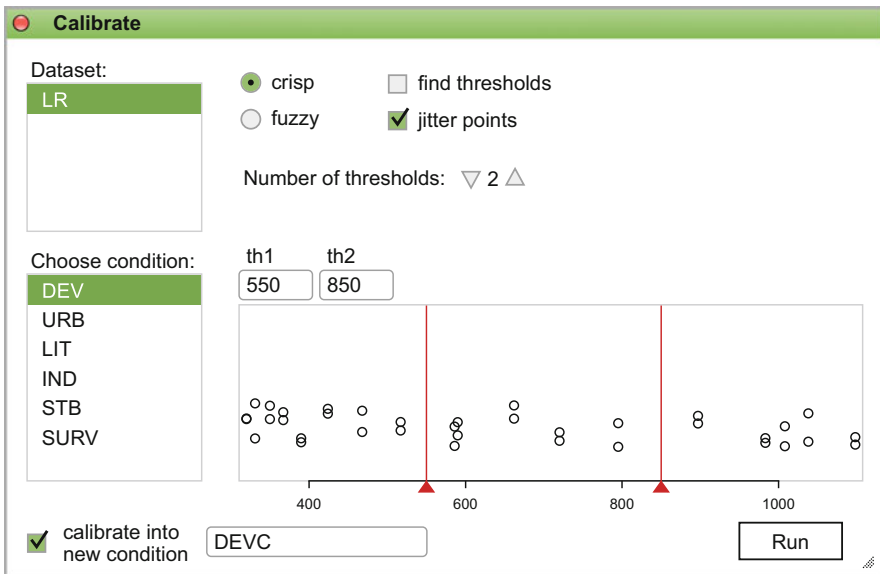


Fig. 4.3 The resized “Calibrate” dialog

Many dialogs are allowed to be resized (for example the plot window), and the content inside is automatically recalculated to the new dialog dimensions. In this particular example, only the threshold setter area was redrawn and the bottom controls (including the **Run** button) have been repositioned. All other buttons and controls have been left to their original position.

As shown in Sect. 2.4, each click and every action like dragging thresholds left or right, triggers a modification of the “Command constructor” dialog. From the second step where the condition is selected, the command constructor

starts to display the equivalent written command which, upon the click of the **Run** button, will be sent to the R console. There are so many features in the graphical user interface that a thorough description of every single one of them would require too much book space and distort the user’s attention from the important topics. To mention just one such “hidden” feature, when multiple thresholds are present in the thresholds setter area, they are automatically sorted and displayed in increasing order, with a limited bounded drag range, between the minimum and maximum of the values found in the data.

The “Command constructor” dialog is refreshed on every click, with the aim to help the user construct the command itself, rather than clicking through the dialogs. This will pay off since the written commands are always better than a point-and-click approach. While users can easily forget what did they click to obtain a particular result, commands saved in dedicated script files are going to be available at anytime and this is helpful for replication purposes: it is more complicated to replicate clicks than to run a script file.

Since there are two functions that accomplish the same goal of calibrating to crisp sets, there are also two separate dialogs. The next to be presented refers to a menu which is not so much related to QCA per se but it accomplishes a more general data transformation process which is present in almost any other software:

Data/Recode

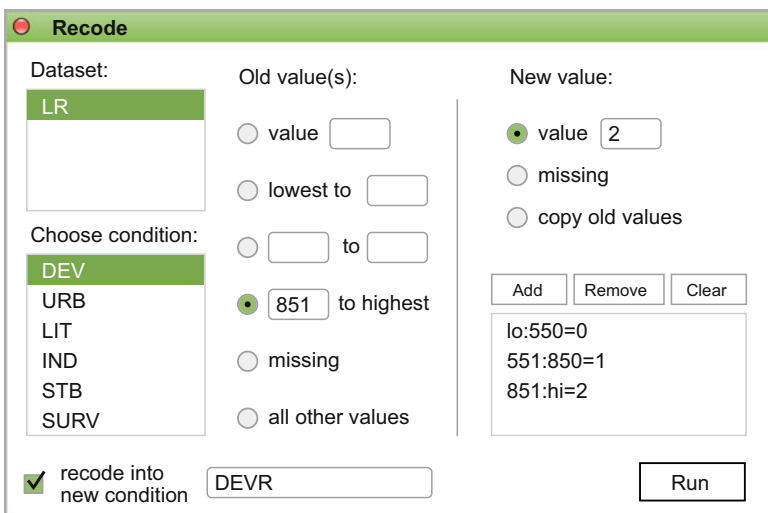


Fig. 4.4 The “Recode” dialog

The design of this dialog is largely inspired from the recoding menu of the SPSS software, to which many of the social science users are very accustomed

with. Figure 4.4 presents the same areas to select for **Dataset** and **Choose condition**: as in the calibration dialog, but otherwise it has the familiar “Old values” and “New values” sections which are found in the other software.

This dialog also has a rather straightforward procedure to use, with the same first two steps as in the calibrate dialog (to select the dataset and the condition), therefore will continue from the third step:

3. select the radio button or the relevant text box(es) and insert the threshold(s), in the **Old value(s)** part of the dialog
4. insert the new value or select from the other options in the **New value** side of the dialog
5. press the **Add** button to construct the rule
6. repeat steps 3–5 for each recoding rule
7. if the recoded values should be saved as a different variable in the same dataset, check the **recode into new condition** and specify the new (recoded) condition name (otherwise it will overwrite the same condition with the new values)
8. click the **Run** button and the new column should appear in the dataset, visible either in the console or in the data editor.

There are two additional buttons on the right side of the dialog: **Remove** erases any selected rule(s) from the recoding rules area, and **Clear** erases all rules at once. As any rule of the rules is selected, their correspondent radios and text boxes are completed with the rule values, both in the old and new parts. This allows modifications to any of the rules, and a second press on the **Add** button brings those modifications in the rules area.

As long as the recoding rules do not overlap (an ‘old’ value should be covered by only one of the recodings), the order of the rules doesn’t matter. But if many recoding rules cover the same old values, then precedence has the last specified rule (which overwrites recodings made by the first specified rules). As always, a few toy examples in the command line with only a handful of values will show the user how the command works for every scenario.

This section ends with the conclusion that calibration to crisp sets is essentially equivalent to recoding the initial raw causal condition with a set of discrete new values. The next section demonstrates what the “real” calibration is all about, applied to fuzzy sets. It is the main reason why the default value of the **type** argument has been changed to “fuzzy”, despite its long lasting traditional default value of “crisp” in all previous versions of the *QCA* package.

4.2 Calibrating to Fuzzy Sets

Social science concepts are inherently difficult to measure. Unlike physical sciences where things can be directly observed and measured, in the social sciences things are not directly observable, hence their measurement is always challenging and often problematic. Some of the concepts from the social world are more easily observable: sex, age, race etc., but the bulk of the social science concepts are highly abstract and need substantially more effort to have them measured, or at least to create an attempt of measurement model. These are complex, multi-dimensional concepts which require the use of yet another (set of) concepts just to obtain a definition, and those concepts need a definition of their on etc.

Neuman (2003) presents an in-depth discussion about the role of concepts in social research, for both quantitative (positivist) and qualitative (interpretive) approaches. The measurement process is different: quantitative research define concepts before data collection and produces numerical, empirical information about the concepts, while in the qualitative research concepts can be produced during the data collection process itself.

Both approaches use conceptualization and operationalization in the measurement process, in a tight connection with the concept definition, although Goertz (2006b) has an ontological perspective arguing there is more about concepts than a mere definition, because researchers need to first establish what is “important” about the entity in order to arrive at a proper definition.

Concepts have a tight interconnection with theory: sometimes the concept formation process leads to new theories, while established theories always use accepted definitions of their concepts. These definitions can change depending on the theoretical perspective employed in the research process. Although not directly concerning QCA, it is a nevertheless important discussion for calibration purposes.

From yet another point of view, concepts have a cultural background just as much they have a theoretical one. This cultural dependence can happen in at least two different ways:

- (a) concepts have different meanings in different cultures: altruism in Hungary is probably something different from the altruism in Korea (to compare very different countries), or the well known continuum for left and right political positioning doesn't have the same meaning in Japan, where political participation resembles very little if nothing at all with the Western concept of “participation”
- (b) even if concepts have the same meaning, their level of concentration can dramatically differ in different cultural and/or historical contexts, for example citizenship or public participation which has very high levels in a country like Belgium, and very low levels in a post-communist country like Romania.

4.2.1 Direct Assignment

The method of direct assignment is the simplest possible way to obtain a (seemingly) fuzzy calibrated condition from some raw numerical data. The term “direct assignment” has been introduced by Verkuilen (2005), while something similar was briefly mentioned by Ragin (2000).

It is likely a method that is tributary to Verkuilen’s formal training in experimental psychology, where expert knowledge is both studied and employed in conjunction with various scales. In the direct assignment, the fuzzy scores are allocated by experts the way they seem fit, according to their expertise. There can be some form of theoretical justification for the various thresholds separating the fuzzy scores, but in the end this is a highly subjective method and it is likely that no two experts will reach exactly the same values.

To avoid the point of maximum ambiguity 0.5, the experts typically choose four, and sometimes even six fuzzy scores to transform the raw data into a fuzzy set. This procedure is extremely similar to the recoding operation when calibrating to crisp sets, with the only exception that the final values are not crisp, but fuzzy between 0 and 1.

To exemplify, we can recode the same condition DEV from the raw version of the Lipset data:

```
recode(LR$DEV, cuts = "350, 550, 850", values = "0, 0.33, 0.66, 1")
```

```
[1] 0.66 1.00 0.66 0.33 0.66 1.00 0.66 0.33 0.33 0.66 0.33 1.00 0.00
[14] 0.00 0.00 0.33 1.00 1.00
```

All values between 0 and 350 are recoded to 0, the ones between 351 and 550 to 0.33, the ones between 551 and 850 to 0.66 and the rest are recoded to 1. Supposing the thresholds (in this case, the cuts) have some theoretical meaning, this is a very simple and rudimentary way to obtain a seemingly fuzzy calibrated condition.

Arguably, the end result is by no means different from a calibration to crisp sets, obtaining a new condition with four levels:

```
recode(LR$DEV, cuts = "350, 550, 850", values = "0, 1, 2, 3")
```

```
[1] 2 3 2 1 2 3 2 1 1 2 1 3 0 0 0 1 3 3
```

Naturally, more levels and generally more multi-value conditions in a dataset expand the analysis with even more possible causal configurations, and from this point of view a fuzzy set (even one having four fuzzy categories) is preferable because it is at least confined between 0 and 1. But while fuzzy sets are notoriously averse against the middle point 0.5, the crisp sets are more than willing to accommodate it in a middle level, for instance creating a multi-value crisp set with three levels:

```
recode(LR$DEV, cuts = "500, 850", values = "0, 1, 2")
```

```
[1] 1 2 1 0 1 2 1 0 0 1 1 2 0 0 0 0 2 2
```

Here, the crisp value of 1 would correspond to a fuzzy membership value of 0.5 and the crisp value of 2 would correspond to a full fuzzy membership value of 1. As it will later be shown, especially in Chap. 7, such a thing is forbidden with fuzzy sets.

Due to its sensitivity to individual expertise, it is difficult to conceptualize the result of a direct assessment as a proper fuzzy set. One year later, Smithson and Verkuilen (2006) don't mention this method at all in their book, and another 2 years after that Ragin (2008b) doesn't even discuss anything related to the direct assignment and presents exclusively what he calls the "direct method" and the "indirect method" of calibration, which are going to be presented in the next sections.

Very likely, many users might confuse the direct assignment with the direct method, and believe that manually assigning (seemingly) fuzzy scores to some properties is not only an acceptable method, but it is recommended by Ragin. This is undoubtedly far from the actual situation, and the two should not be mistaken.

Despite being presented in this book in a dedicated section, the main recommendation is *not* to use it, if at all possible. There are far better alternatives, for any possible scenario. Some argue that such a method is the only one possible when transforming Likert type response scales (therefore ordinal measured data) to fuzzy values, and indeed the direct and the indirect methods would not be appropriate in such situations because they both need numerical data from at least an interval level of measurement. But the final Sect. 4.3 shows one possible way to deal with such situations.

When presenting the direct assignment, Verkuilen himself (2005, p. 471) mentions no less than five main problems associated with this method, and only for this method (for all the other, transformational assignments, no such problems are mentioned):

1. The interpretation of the generated set membership values is very difficult
2. The direct assignment is mostly unreliable, especially for very abstract concepts
3. It contains bias (the expert's own expertise)
4. There is no error (uncertainty) associated with the generated set membership values
5. Combining the expertise from multiple judges is also difficult

Having presented all this information, the main advice is to refrain from using the direct assignment. The next sections will introduce the standard calibration techniques as they are used in the current QCA practice.

4.2.2 Direct Method, the “S-Shape” Functions

This above argument holds for complex multidimensional concepts, but interestingly it also holds for very simple concepts, like age or height. In a Western culture, suppose we have an average height of 1.75 m (roughly 5 ft 9 in), with a range of heights between a minimum of about 1.5 m (an inch below 5 ft) and a maximum of about 2 m (about 6 ft 7 in).

We can simulate a sample of 100 such height values (in centimeters) via these commands:

```
set.seed(12345)
height <- rnorm(n = 100, mean = 175, sd = 10)
range(height)
```

```
[1] 151.1964 199.7711
```

This is a normally distributed, random sample of heights where the “tallest” person has 1.99 m and the “shortest” one has 1.51 m. It doesn’t display any large separation for clusters of points, as Fig. 4.5 shows, perhaps with the exception of the two values on the left side (but since this is randomly generated data it has no theoretical interpretation). In this example, there are 100 values, and most of them overlap in the middle part of the plot. To have a more accurate idea of what this distribution looks like, the argument `jitter` was activated to add a bit of “noise” on the vertical axis.

```
Xplot(height, jitter = TRUE, cex = 0.8)
```

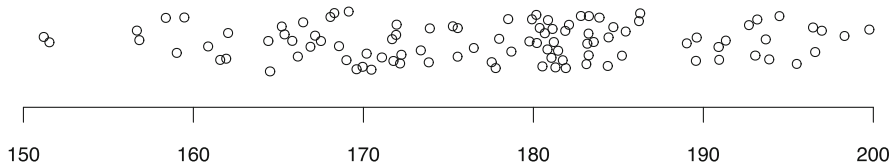


Fig. 4.5 The distribution of generated heights

Now suppose an accepted definition of a “tall” person, for the Western cultures, is anyone having at least 1.85 m (at least 6 ft) so anyone above this threshold would be considered “tall”. Conversely, anyone below 1.65 m (5 ft 5 in) would be considered “short”. If these anchors hold, it is now possible to calibrate the concept of height and transform its raw values into membership scores for the set of “tall people” using this command:

```
# increasing calibration of height
tich <- calibrate(height, thresholds = "e=165, c=175, i=185")
```

It is the same function `calibrate()`, with two important differences from the previous section:

1. the argument `type` needs no special specification (its default value is already "fuzzy"), to signal we are going to obtain a continuous set of membership values between 0 and 1
2. the argument `thresholds` has a different structure, specifying three letters ("e" stands for complete exclusion, "c" is the crossover point, and "i" is the threshold for complete inclusion in this set), and three associated numbers which are the actual threshold values.

As previously mentioned, although this command looks deceptively simple, there are other arguments "at work" with their default values associated with the fuzzy calibration. Generating set membership values has to use a mathematical function, in this case the default being the logistic distribution due to the argument `logistic = TRUE`. Since the logistic distribution has a well known shape which resembles the letter "S", package *QCA* names the resulting values an "s-shape" (to differentiate from "bell-shape" which will be introduced later in this section).

There is more to tell about these inter-related arguments, because not all of them are active all the time. They get activated in a logical sequence, once the first link of the sequence has been activated. In this example, the first link is the argument `type` which has been left to the default `fuzzy`. Only when this happens, the argument `logistic` starts to work (for the crisp version it didn't have any effect whatsoever), and a third link in the sequence is the argument `idm` which depends on both `type` and `logistic`, and becomes active only when the `type` is `fuzzy` and `logistic` is `TRUE`. If any of the first two has a different value, the argument `idm` will stop producing any effect (see Sect. 4.2.3 for details of what this argument is and how it works).

The function is smart enough to detect all these logical sequences of arguments, but on the other hand the user needs to properly understand how they work in order to effectively make use of the function.

And this is not the end of the story, for there is even more to tell about the specification of the `thresholds` argument:

- (a) Whenever the function starts with exclusion and ends with inclusion thresholds, the function will increase from left to right. That means it is logically possible to specify a decreasing function, if the specification is reversed: when it starts with inclusion (for the low raw values on the left) and ends with exclusion (for the large raw values on the right), the function will be decreasing from left to right.
- (b) If `thresholds` contains exactly three values, it signals the use of the "s-shaped" functions, and if it uses exactly six values, it signals the use of the "bell-shaped" function (see Sect. 4.2.4). For fuzzy calibration, any other number of thresholds will generate an error.

Therefore we can create the set membership in the set of “short” people by simply changing the order of the thresholds from left to right, noting the thresholds’ values are always increasing from left to right, as we move from the smaller raw values on the left of the distribution towards the larger raw values on the right side of the distribution:

```
# decreasing calibration of height
dch <- calibrate(height, thresholds = "i=165, c=175, e=185")
```

Figure 4.6 presents two plots of raw versus their correspondent calibrated values, in the distinctive, increasing s-shape fuzzy set on plot (a) and decreasing s-shape fuzzy set on plot (b).

The horizontal axes correspond to the raw data (ranging from 1.5 to 2.0 m in height), and the vertical axes have membership scores in the set of “tall” and “short” people, ranging from 0 to 1. Each initial value has been transformed into a membership degree, but unlike the crisp version(s) where the raw data got recoded to small number of new values, in this case each value has its own degree of membership, along the logistic function starting from the small raw values on the left and progressively going towards the large raw values on the right.

```
par(mfrow = c(1, 2))
plot(height, ich, main = "a. Set of tall people", xlab = "Raw data",
      ylab = "Calibrated data")
plot(height, dch, main = "b. Set of short people", xlab = "Raw data",
      ylab = "")
```

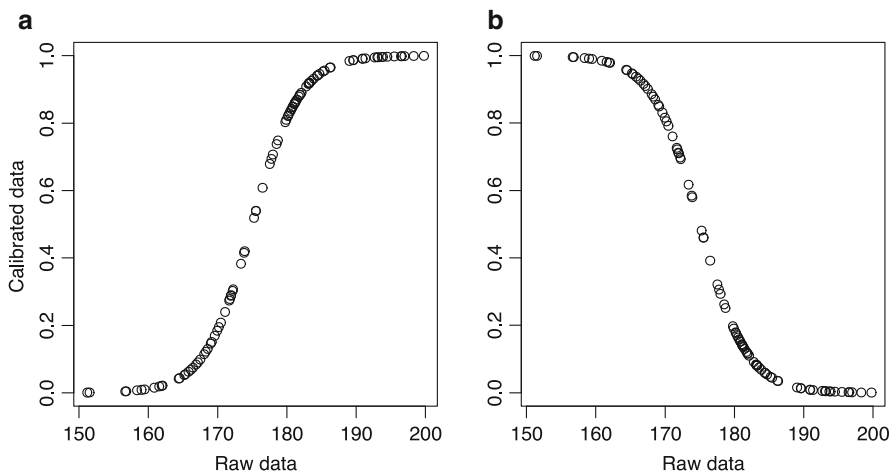


Fig. 4.6 Calibration using the logistic function. (a) Set of tall people. (b) Set of short people

In Fig. 4.6a, tall people should have a higher inclusion in the set than short people, therefore the function increases from 0 in the lower left side to 1 in the upper right side, such that a person standing 1.5 m in height will have a 0 inclusion score in (or completely out of) the set of “tall” people.

The opposite situation can be seen in Fig. 4.6b, where it seems natural for tall people to have low inclusion scores in the set of “short” people, therefore the shape of the function decreases from 1 in the upper left side to 0 in the lower right side (an inverted s-shape function), such that a person standing 2 m tall would have a 0 inclusion score in (or completely out of) the set of “short” people.

Before going too deep into the specificities of how the logistic function is used to obtain these membership scores, let us return to the original discussion about the differences between cultures. The exclusion, crossover and inclusion thresholds of 1.65, 1.75 and 1.85 are valid for the Western context, but would not produce the entire range of values between 0 and 1 for an Eastern context.

Suppose there is a hypothesis relating height to labor market outcomes (Perrico et al. 2004), or to the general wellness and quality of life (Bolton-Smith et al. 2000). It would not make any sense to use the same threshold values for all countries in a comparative study. The average height in a country like Korea is much lower than the average in a Western country like France (not to mention Sweden), therefore using the same thresholds would cluster most of the calibrated membership scores to the lower left part of the plot.

The goal of the calibration process is to obtain membership values for the full range from 0 to 1, for each and every compared country, according to the meaning of the concept for each particular country of culture. “Tall” has a very different meaning for tribes in Guatemala (Bogin 1998), where the average height in the 1970s was reported for the Mayan men to about 1.575 m (5 ft 2 in). In order to obtain the full range between 0 and 1, is it mandatory to use a different set of thresholds values for exclusion, crossover and inclusion in the set.

As each country has a different set of thresholds, establishing the exact values for each set doesn’t have anything to do with mathematics or statistics. Rather, it has everything to do with the researcher’s familiarity with each studied country, which is a qualitative, rather than quantitative approach to research. The researcher has to know and to understand the “meaning” of each concept for each particular country, and this cannot be mathematically derived from separate random samplings from every studied country.

In this example the individual cases represent people’s heights, but the same line of thought can be applied in the situation when cases represent whole

Figure 4.7 shows the corresponding dialog for the fuzzy calibration in the graphical user interface, using the raw values from the Lipset dataset LR. It is the same dialog, but displaying only the controls specific to fuzzy calibration. This demonstrates the logical sequence relations between various arguments of the `calibrate()` function, those specific to the crisp version having no effect when calibrating to fuzzy sets, therefore taken out of the picture. Previously, the thresholds setter was specific to the crisp calibration only, but starting with version 2.5 this reactive area is displayed for fuzzy calibration as well.

In this respect, the graphical user interface has an advantage over the command line interface, because certain clicks already trigger the logical sequence of argument relations, saving the user from constantly checking those logical relations. To add more to the advantages, clicks and their logical implications are immediately translated to the command constructor (the user interface is a reactive shiny app, after all), and users learn about these implications by studying how the perfect logical command looks like, with each click.

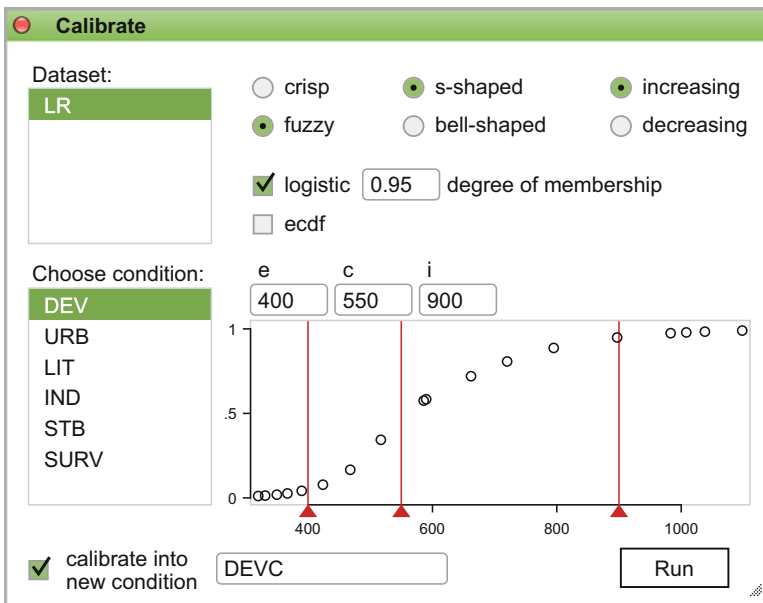


Fig. 4.7 The fuzzy “Calibrate” dialog

In the fuzzy version of the calibrate dialog, it can already be seen that the **s-shaped** type of function is predefined (default), increasing from left to right. Choosing the **decreasing** in the radio button, will automatically change the labels of the thresholds and their corresponding command, when all values are provided. Also, the logistic function is selected by default (a second link in the logical sequence of fuzzy calibration), and with it the text box specifying the degree of membership (which depends on both **fuzzy** and **logistic** being checked, the third link in the logical sequence of argument relations).

countries, and the dataset is a collection of values for each individual country. Ragin's calibrated values (rounded to two decimals) can be obtained easily via:

```
inc <- c(40110, 34400, 25200, 24920, 20060, 17090, 15320, 13680, 11720,
        11290, 10940, 9800, 7470, 4670, 4100, 4070, 3740, 3690, 3590,
        2980, 1000, 650, 450, 110)
incal <- round(calibrate(inc, thresholds = c(2500, 5000, 20000)), 2)
```

The calibrated values have been rounded to two decimals for an easier comparison with Ragin (2008b, p. 89) table and also with the *fs/QCA* software which also rounds to two decimals by default. But a good recommendation, for the purposes of QCA minimization, is to leave all decimals of the calibrated values intact.

Table 4.1 displays the raw values and their calibrated counterparts, side by side for each individual country. There are very minor differences from Ragin's values (for example Israel), explained in Sect. 4.2.3.

Table 4.1: Per capita income (INC) calibrated (INCAL)

	INC	INCAL
Switzerland	40,110	1.00
United States	34,400	1.00
Netherlands	25,200	0.98
Finland	24,920	0.98
Australia	20,060	0.95
Israel	17,090	0.91
Spain	15,320	0.88
New Zealand	13,680	0.85
Cyprus	11,720	0.79
Greece	11,290	0.77
Portugal	10,940	0.76
Korea, Rep.	9800	0.72
Argentina	7470	0.62
Hungary	4670	0.40
Venezuela	4100	0.26
Estonia	4070	0.25
Panama	3740	0.18
Mauritius	3690	0.18
Brazil	3590	0.16
Turkey	2980	0.08
Bolivia	1000	0.01
Cote d'Ivoire	650	0.01
Senegal	450	0.00
Burundi	110	0.00

The selector for the number of thresholds was removed from the dialog for the fuzzy calibration, because unlike the crisp version where the user can potentially specify any number of thresholds, in the fuzzy calibration this number is restricted to either 3 (for the s-shaped functions) or 6 (for the bell-shaped function) and the user is not given any possibility to make a mistake.

Only when all thresholds values have been specified (they can be copied from the crisp version, if first using the thresholds setter to determine their values), the command constructor will introduce the `thresholds` argument in the written command, with the associated labels “e” for exclusion, “c” or crossover and “i” for inclusion.

The last checkbox in the list of dialog options is `ecdf`, which stands for the empirical cumulative distribution function. This is different possibility to generate degrees of membership in particular sets, whenever the researcher does not have a clear indication that the logistic function has the best possible fit on the original raw values.

There are many other types of functions that can be employed besides the logistic function (Thiem and Duşa 2013; Thiem 2014), which are all special types of CDFs (cumulative distribution functions) that will be discussed later, but sometimes researchers don’t have a clear preference for a certain type of fit function and prefer to let the raw data speak for themselves.

This is the case with the ECDF (empirical CDF) which uses the data itself to incrementally derive a cumulative function in a stepwise procedure. In the written command, although arguments `logistic` and `ecdf` are mutually exclusive, it is important to explicitly set `logistic = FALSE` before making use of `ecdf = TRUE` (otherwise the default activated logistic function will have precedence).

```
ech <- calibrate(height, thresholds = "e=155, c=175, i=195",
  logistic = FALSE, ecdf = TRUE)
```

Another reason why this setup is important, as it will be shown, is the case where both `logistic` and `ecdf` are deactivated, when a different set of calibrating functions will enter into operation. In order to minimize the number of arguments and reduce complexity, every last drop of logical relations between arguments has been employed in this function, but it becomes all too important to understand these relations.

The object `ech` contains the calibrated values using the ECDF—empirical cumulative distribution function, which are bounded in the interval between 0 and 1 just like all the other CDFs, with the difference that all values below the exclusion threshold are automatically allocated to a set membership score of 0, and all values above the inclusion threshold are allocated to a set membership score of 1 (unlike the logistic function, where by default “full membership” is considered to be any membership score above 0.95).

The following command produces Fig. 4.8.

```
plot(height, ech, xlab = "Raw data", ylab = "Calibrated data", cex = 0.8)
```

The points don't follow a very clear mathematical function line, as is the case with the perfect shape of the logistical function in Fig. 4.6, but it is remarkably close if taking into account that the "shape" of the distribution (if it can be called like that) has been determined from the distribution of the raw, observed data.

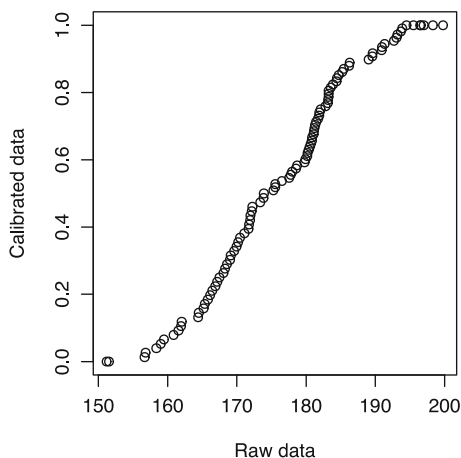


Fig. 4.8 Calibration in the set of tall people, using the ECDF

In the graphical user interface, this seemingly complex relation between arguments is relaxed by the automatic activation or deactivation of the related arguments, once the starting arguments have been activated. In Fig. 4.7, if the user clicks on the **ecdf** checkbox, the **logistic** counterpart is automatically deactivated and the command constructor displays the written command above. Even more, since the **degree of membership** text box only makes sense in relation with the **logistic** checkbox, when the **ecdf** is activated the text box will disappear from the dialog. This way, the user can visually understand which argument is related to which other, in a logical sequence.

Where both **logistic** and **ecdf** are set to **FALSE**, neither one of these two CDFs are used and the **calibrate** function employs the following mathematical transformation (adapted after Thiem and Duşa 2013, p. 55):

$$dm_x = \begin{cases} 0 & \text{if } x \leq e, \\ \frac{1}{2} \left(\frac{e-x}{e-c} \right)^b & \text{if } e < x \leq c, \\ 1 - \frac{1}{2} \left(\frac{i-x}{i-c} \right)^a & \text{if } c < x \leq i, \\ 1 & \text{if } x > i. \end{cases} \quad (4.1)$$

where:

- e is the threshold for full exclusion
- c is the crossover
- i is the threshold for full inclusion
- x is the raw value to be calibrated
- b determines the shape below the crossover (equivalent to argument `below`)
- a determines the shape above the crossover (equivalent to argument `above`)

If a raw value is smaller than the threshold for full exclusion, it will be assigned a degree of membership equal to 0, and the same happens at the other end being attributed a value of 1 if greater than the threshold for full inclusion. The interesting part happens in the two areas between the thresholds, and the shape of the function is determined by the values of a and b :

- if left to their default values equal to 1, the function will be a perfect increasing line
- when positive but smaller than 1, it dilates the distribution into a concave shape (a above and b below the crossover)
- when greater than 1, it concentrates the distribution into a convex shape (same, a above and b below the crossover)

The need for such alternative functions appeared for methodological reasons (CDFs cannot produce, for example, bell-shaped curves), and it must be said that researchers commonly misunderstand how the logistic function operates (the default, and the only calibration function in the *fs/QCA* software). When performing calibration to crisp sets, one expects and indeed it is happening that everything to the left of the threshold to be allocated one value and everything to the right of the threshold, a different value.

The “threshold setter” acts as a first visual, mind conditioning tool that in my opinion affects the expectations on how the fuzzy calibration should work. One such natural expectation is, when establishing a threshold for full set exclusion, everything below that threshold should be fully excluded from the set (thus being allocated a value of 0) and when establishing a threshold for full set inclusion, everything above that would be fully included in the set (thus being allocated a value of 1).

Contrary to this expectation, Ragin’s procedure considers “fully out” everything with a membership value up to 0.05 and “fully in” everything with a membership value of at least 0.95. Thus, by using the logistic function, it is common to have calibrated values that never reach the maximum inclusion of 1, despite the existence of raw values above the inclusion threshold. That is a bit counter-intuitive, but it makes sense when following Ragin’s logic.

The family of functions from Eq. (4.1) (with respect to the choice of values for parameters a and b) makes sure that everything outside the two full exclusion and inclusion thresholds are going to be calibrated accordingly. This sort

of reasoning holds true if the researcher expects a linear shape, that is all calibrated values should be lined up against a straight line, with inflexions at the outer thresholds.

Figure 4.9 shows three possible calibration functions, all using the same set of exclusion threshold $e = 155$, a crossover $c = 175$ and a full inclusion threshold $i = 195$.

```
c1h <- calibrate(height, thresholds = "e=155, c=175, i=195",
  logistic = FALSE) # by default below = 1 and above = 1
c2h <- calibrate(height, thresholds = "e=155, c=175, i=195",
  logistic = FALSE, below = 2, above = 2)
c3h <- calibrate(height, thresholds = "e=155, c=175, i=195",
  logistic = FALSE, below = 3.5, above = 3.5)

plot(height, c3h, cex = 0.6, col = "gray80", main = "",
  xlab = "Raw data", ylab = "Calibrated data")
points(height, c2h, cex = 0.6, col = "gray50")
points(height, c1h, cex = 0.6)
```

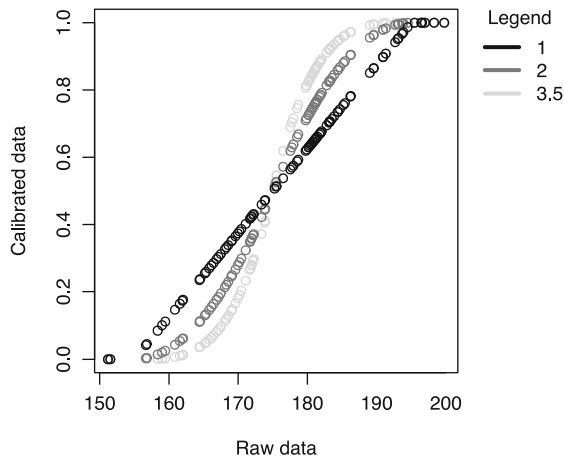


Fig. 4.9 Three alternative calibration functions

The distribution with the black colour has both arguments `below` and `above` at their default values of 1 to produce a linear shape. Changing the values of the two arguments, first at the value of 2 (gray colour in the plot) and next to a value of 3.5 (the light gray colour in the plot) curves the distribution progressively, and at the value of 3.5 it is remarkably similar with the logistic shape (the only difference being that raw values outside the exterior thresholds have been allocated membership scores of exactly 0 and respectively 1).

For the decreasing type of functions (fully including the low values in the raw data, and excluding the large ones), the order of the calculations simply reverses, as shown in Eq. (4.2):

$$dm_x = \begin{cases} 1 & \text{if } x \leq i, \\ 1 - \frac{1}{2} \left(\frac{i-x}{i-c} \right)^a & \text{if } i < x \leq c, \\ \frac{1}{2} \left(\frac{e-x}{e-c} \right)^b & \text{if } c < x \leq e, \\ 0 & \text{if } x > e. \end{cases} \quad (4.2)$$

When both parameters a and b are equal, a “decreasing” type of function is the same thing as the negation of an increasing function, and the scores from Eq. (4.2) can also be obtained by negating (subtracting from 1) the scored from Eq. (4.1).

Obtaining a linear calibrated condition using the graphical user interface is a matter of deactivating both the **logistic** and **ecdf** checkboxes. As the argument **idm** and its text box depend on the activated **logistic** checkbox, it has been removed from the dialog as shown in Fig. 4.10. Two new controls have appeared instead, that control the shape of the calibration function: **above** and **below** (the equivalents of the a and b parameters from (4.2)).

They control whether the shape between the thresholds is linear (when they are equal to 1) or gain a certain degree of curvature, either above of below the crossover threshold. When **below** has a value between 0 and 1, the curve dilates in a concave shape below the crossover, and when it has values above 1 it concentrates in a convex shape below the crossover. The same happens with the other control, only **above** the crossover.

Mathematically, both can have negative values, but the results are unexpected and the shape is meaningless, therefore in package **QCA** they have been restricted to positive values.

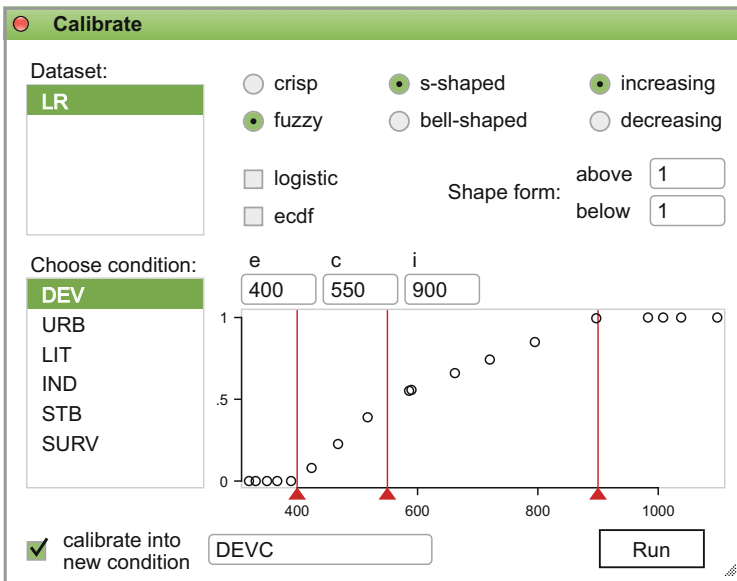


Fig. 4.10 Linear fuzzy calibration dialog

In theory, the choice of calibration functions should not be a matter of personal taste or aesthetics, and Thiem (2014) speculates that it might affect the coverage of the calibrated causal condition on the outcome. However, in practice there is no empirical evidence that calibration functions dramatically alter the final minimization results in QCA, using the same set of calibration thresholds. If the final solutions are identical, irrespective of the calibration functions, the default logistic one should be the simplest to use for most QCA applications.

4.2.3 How Does It Works: The Logistic Function

Ragin’s idea to use the logistic function for calculating the degree of membership is a very clever one. He adapted a mathematical transformation to the set theory, starting with the observation that both probability (specific to mathematics and statistics) and the degree of membership (specific to set theory) are ranging in the bounded interval from 0 to 1.

Probability and set membership are very different things, as Ragin (2008b, p. 88) rightly points out. To demonstrate how this adaptation works, I am going to start from the definition of a simple mathematical transformation of the probability, called “odds”. In the binomial distributions there are two simple concepts which contribute to the calculation of the odds, namely the probability of success (p) and the probability of failure (q), where the following equation happens: $p + q = 1$, and its counterpart is also intuitive: $p = 1 - q$.

In plain language, the probability of success p is the complement of (1 minus) the probability of failure q , and vice-versa. The odds are simply the ratio between the two probabilities:

$$\frac{p}{q} = \frac{p}{1-p}$$

One particularity of the odds is the fact they are non-negative, ranging from a minimum value of 0, which happens when $p = 0$, to a maximum value of plus infinity which happens when $p = 1$. To allow the full range from minus to plus infinity, statistics uses the so called “log odds”, which is the natural logarithm of the odds¹:

$$\ln\left(\frac{p}{1-p}\right)$$

¹ This term should not be confused with the log odds in the logistic regression (aka “logit”), that is the natural logarithm of the “odds ratio”.

When $p = 0$ the log odds is equal to $-\infty$; when $p = 1$ the log odds is equal to $+\infty$, and when $p = q = 0.5$ the log odds will be equal to zero. This observation will play an important role in the process of calibration to set membership scores.

Ragin adapted the log odds to set theory by replacing the probability of success with the degree of membership (argument `idm` in function `calibrate`), which is also ranging from 0 to 1 so that the log odds is now equal to:

$$\ln\left(\frac{dm}{1-dm}\right)$$

Similar to the probability model, there is a mathematical “direct” relation between any degree of membership in a set and its associated log odds (likely the reason why Ragin called this transformation the “direct method”). Whenever the degree of membership `idm` is known, their associated log odds can be calculated by a direct transformation, but most importantly knowing the value of the log odds allows calculating the degree of membership, which is the purpose of the calibration process.

It all boils down to calculate the equivalent of the log odds for any particular raw value, and the degree of membership can be mathematically calculated. This is precisely what the fuzzy calibration does, using the logistic function. In his example with the per capita income, Ragin employs the following procedure:

1. establish the thresholds: 20,000 for full inclusion, 5000 for crossover and 2500 for the full exclusion from the set of developed countries
2. calculate the deviation of each raw value from the crossover, where values above that threshold will be positive, and values below will be negative.
3. calculate the ratio between the log odds associated with full membership (3) and the distance between the full inclusion and crossover thresholds (15,000), for the positive deviations from the crossover: $3/15,000 = 0.0002$
4. calculate the ratio between the log odds associated with full exclusion (-3) and the distance between the full exclusion and crossover thresholds (-2500), for the negative deviations from the crossover: $-3/-2500 = 0.0012$
5. calculate the equivalent log odds for each raw value of per capita income, multiplying each deviation from the crossover with the scalar resulting from step 3 or step 4, depending if the deviation is positive or negative
6. mathematically derive the degree of membership out of the calculated log odds.

For example the case of Israel, which has a per capita income of 17,090 USD and a deviation from the crossover equal to 12,090 USD. This deviation should be multiplied with 0.0002 to obtain the associated log odds of 2.42, and the degree of membership can be mathematically derived, cancelling out the effect of the logarithm by using the exponential function:

$$\ln\left(\frac{dm}{1-dm}\right) = 2.42$$

that is equivalent to:

$$\frac{dm}{1-dm} = e^{2.42}$$

and finally the degree of membership is trivially extracted:

$$dm = \frac{e^{2.42}}{1 + e^{2.42}} \approx 0.92$$

The value of 0.92 (in fact 0.9183397 rounded to two decimals) is what the calculation arrives at, and is exactly what the *fs/QCA* software outputs for Israel. Using some mathematical tricks specific to the logistic function combining the natural logarithm and the exponential function, the same value can be obtained in a single step calculation:

$$dm = \frac{1}{1 + e^{-\frac{12090 \times 3}{15000}}} = \frac{1}{1 + e^{-2.42}} \approx 0.92$$

However, there is a very slight approximation introduced by Ragin (2008b, p. 90) in his calculations, when referring to:

... cases with per capita income of \$20,000 or greater (i.e. deviation scores of \$15,000 or greater) are considered fully in the target set, with set membership scores ≥ 0.95 and log odds of membership ≥ 3.0

Actually, for a set membership score of 0.95 the exact value of the log odds is 2.944439, or the other way round for a log odds value of 3 the exact set membership score is 0.9525741. However there is no direct, mathematical relation between the set membership score of 0.95 and the log odds value of 3. Keeping 0.95 as the fixed set membership score for full set inclusion, the calculation becomes:

$$dm = \frac{1}{1 + e^{-\frac{12090 \times 2.944439}{15000}}} = \frac{1}{1 + e^{-2.373218}} \approx 0.91$$

The function `calibrate()` in package *QCA* offers the argument `idm` (inclusion degree of membership) which has a default value of 0.95, thus deriving the exact value of the log odds for each particular raw value. In the vast majority of situations, the results between *fs/QCA* and the R package *QCA* are identical but in very few situations, due to the difference between approximate versus exact values, there are very slight differences, for example Israel where the exact calculated value is 0.9147621, and rounded to two decimals becomes 0.91 as opposed to the value of 0.92 presented by Ragin. These kinds of differences, however, are insignificant with respect to the overall, calibrated set membership scores.

4.2.4 Direct Method, the “Bell-Shape” Functions

A cumulative distribution function is not the universal answer for all calibration questions. There are situations when the problem is less about which calibration function to choose, but more with the fact that a CDF like the logistic function is simply unable to perform calibration for certain types of concepts.

All previous examples refer to calibration functions which always end in the opposite diagonal corner of where they started. If starting from the lower left part of the plot, it is an increasing function (therefore excluding the low values and including the large ones from the raw data distribution), and if starting from the upper left corner it is a decreasing function (including low values and excluding the large ones from the set). These types of functions are called monotonic, always increasing or always decreasing to a certain point.

But there are situations where the concept to be calibrated does not refer to the extreme points of the raw data distribution. Taking Ragin’s example of the “developed” countries, another possible concept could be for example “moderately developed” countries. In this definition, Burundi (110 USD) is clearly out of the set, but interestingly Switzerland (40,110 USD) should also be out of this set because it is a highly developed country, not a “moderately developed” one.

Both extremes of very poor and very rich countries should be excluded from the set of moderately developed ones. So the calibration function should only include the countries in the middle of the data distribution (the mid points), excluding the extremes (the end points). Clearly, this sort of calibration cannot be performed by a monotonic function which is built to include points only at the extremes. Rather, it should be a non-monotonic function which can change signs in the middle of the distributions and decrease if previously was increasing, or increase if previously was decreasing.

This is the very description of a “bell-shaped” function curve, as depicted in Fig. 4.11, where part a. displays two types of linear calibrations (trapezoidal with gray colour and triangular with black) that both resemble a “bell”-like shape, while part b. displays a their “curved” counterparts using the same sets of thresholds but with different values for the parameters `above` and `below`.

Plotting the objects produced with the code below, against the initial height which contains the initial raw values, will produce Fig. 4.11, using a plot matrix of 1 row and 2 columns, as in the similar looking Fig. 4.6 (only the objects are created with these commands, to actually produce the plots readers should use the `plot()` and `points()` functions).

```

triang <- calibrate(height, thresholds = "e1=155, c1=165, i1=175, i2=175,
                                         c2=185, e2=195")
trapez <- calibrate(height, thresholds = "e1=155, c1=164, i1=173, i2=177,
                                         c2=186, e2=195")
bellsh <- calibrate(height, thresholds = "e1=155, c1=165, i1=175, i2=175,
                                         c2=185, e2=195", below = 3, above = 3)
trabel <- calibrate(height, thresholds = "e1=155, c1=164, i1=173, i2=177,
                                         c2=186, e2=195", below = 3, above = 3)

```

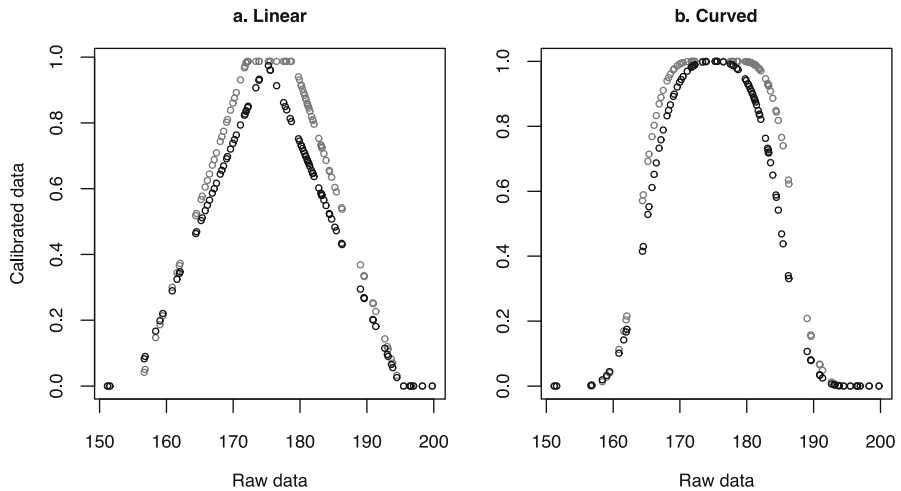


Fig. 4.11 Fuzzy calibrations in the set of “average height”. (a) Linear. (b) Curved

Unlike the s-shaped functions which need three thresholds to define the specific shape, the bell-shaped functions need six. Since the inclusion happens at the middle of distribution, there are two areas that are excluded, hence two exclusion points and two associated crossovers. Finally, the two inclusion points define the exact range which is going to be included. In the code above, it can be noticed the inclusion points for the triangular shape are equal, which explains why the shape has a single point in the top middle area.

Based on the (non)equality of the inclusion thresholds, there are only two main types of bell-shape functions: triangular, with a single point at the top, and trapezoidal, with a range of points at the top. The other shapes are all part of this family, with different degrees of curvature based on the values of `above` and `below` arguments.

What is shared by both s-shaped and bell-shaped families of functions, and it makes the task of defining and identifying the set of thresholds easier, is the fact their thresholds are specified in ascending order, from left to right. Whether increasing or decreasing, it does not matter: the thresholds are always specified in ascending order, in the same logic as a plot of points which is always drawn from left to right on the x axis, in ascending order.

The key to identify the difference between an increasing and decreasing function lies in the names of the thresholds: if beginning with an exclusion it means an increasing function (first excluding the smaller values on the left, then gradually increasing towards the larger values on the right), and if beginning with an inclusion it means a decreasing function.

This is valid for both s-shapes and bell-shapes alike. In the examples above, thresholds are all starting from the left with an *e* to exclude the points on the left, then increasing towards, and include the middle points, then decreasing to again exclude the larger values on the right.

The mathematical transformations that define these families of curves are similar to the s-shape functions, combining increasing and decreasing equations.

$$dm_x = \begin{cases} 0 & \text{if } x \leq e_1, \\ \frac{1}{2} \left(\frac{e_1 - x}{e_1 - c_1} \right)^b & \text{if } e_1 < x \leq c_1, \\ 1 - \frac{1}{2} \left(\frac{i_1 - x}{i_1 - c_1} \right)^a & \text{if } c_1 < x \leq i_1, \\ 1 & \text{if } i_1 < x \leq i_2, \\ 1 - \frac{1}{2} \left(\frac{i_2 - x}{i_2 - c_2} \right)^a & \text{if } i_2 < x \leq c_2, \\ \frac{1}{2} \left(\frac{e_2 - x}{e_2 - c_2} \right)^b & \text{if } c_2 < x \leq e_2, \\ 0 & \text{if } x > e_2. \end{cases} \quad (4.3)$$

It is easy to see Eq. (4.3), specific to the increasing bell-shape function, is not much different from Eqs. (4.1) and (4.2).

The equation for the decreasing bell-shape simply reverses the order of transformations for the increasing bell-shape function, but as already mentioned in the s-shape section, when **above** is equal to **below**, a decreasing function is equivalent to the negation of the increasing one, both in terms of logics (“short” really means “not tall”), and also mathematically because a decreasing calibration is the same as negating the degrees of membership for the increasing function, by subtracting the membership scores from 1. The decreasing bell-shape function, in this example, means “not average height” which includes both short and tall people and excludes the middle heights.

If parameters *a* and *b* are not equal, then a simple negation is not possible because the shape of the curve below and above the crossover differs. In this situation, and generally valid for all situations, the best approach is to apply the mathematical transformation from Eq. (4.4).

$$dm_x = \begin{cases} 0 & \text{if } x \leq i_1, \\ 1 - \frac{1}{2} \left(\frac{i_1 - x}{i_1 - c_1} \right)^a & \text{if } i_1 < x \leq c_1, \\ \frac{1}{2} \left(\frac{e_1 - x}{e_1 - c_1} \right)^b & \text{if } c_1 < x \leq e_1, \\ 1 & \text{if } e_1 < x \leq e_2, \\ \frac{1}{2} \left(\frac{e_2 - x}{e_2 - c_2} \right)^b & \text{if } e_2 < x \leq c_2, \\ 1 - \frac{1}{2} \left(\frac{i_2 - x}{i_2 - c_2} \right)^a & \text{if } c_2 < x \leq i_2, \\ 0 & \text{if } x > i_2. \end{cases} \quad (4.4)$$

The corresponding graphical user interface dialog, as seen in Fig. 4.12, is similar to the previous one but the radio button is now switched to **bell-shaped** instead of **s-shaped**. The user interface automatically creates the six text boxes where the values of the thresholds are inserted (in ascending order from left to right).

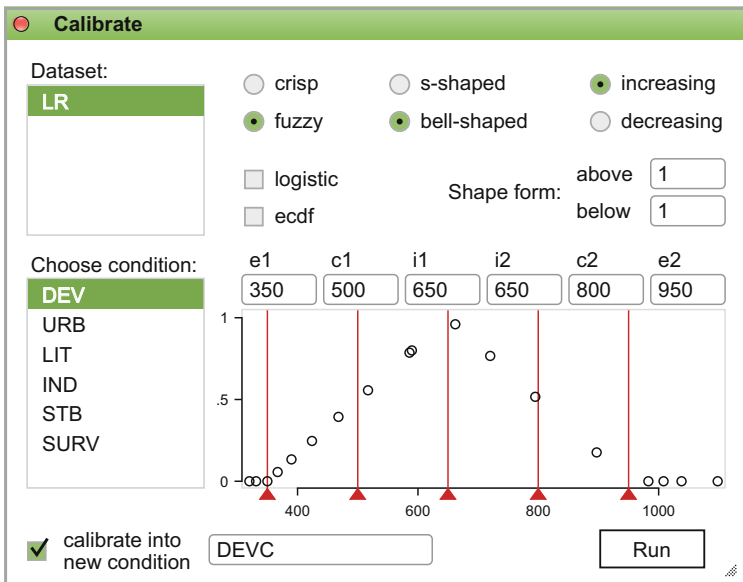


Fig. 4.12 Calibration dialog for the fuzzy set of “average development”

For each value, one threshold is created in the thresholds setter area, and function of their positions the points display the familiar triangular shape, given the two inclusion thresholds in the middle are equal. As in the case of crisp calibration, thresholds can be dragged left and right (or modified directly in the text boxes), and the vertical position of the points (their associated degree of membership) is recalculated every time the action is performed.

Once generated, the thresholds are not allowed to move past other thresholds, in order to maintain a strict order of their associated values, in ascending order. It should also be mentioned that the thresholds setter area has an informative role only, the dataset doesn't get modified until the user presses the "Run" button.

This makes the thresholds setter highly interactive, and more intuitive for users who, up until now, were calibrating "blind": the result was visible only in the database, but now it is possible to visualize the end result before creating a new calibrated condition.

4.2.5 The Indirect Method

The indirect method of calibration, although attributing fuzzy set scores in the target condition, has very little to do with the fuzzy calibration from the previous sections. It has no anchors, and it doesn't need to specify a calibration function to calculate the scores along with.

It implies a rather simple procedure involving two steps: a first one of data recoding, which is trivially done using the function `recode()`, and a second one of a numerical transformation involving both the original, interval level condition and the recoding from the first step.

In the data recoding step, the task is to first establish the number of categories to recode to. Ragin (2008b) mentions six such qualitative categories, as seen in Table 4.2, but this number can be flexible allowing more or sometimes even less categories.

Table 4.2: Indirect calibration categories

	Score
Completely in the set	1.0
Mostly but not completely in the set	0.8
More in than out of the set	0.6
More out than in the set	0.4
Mostly but not completely out of the set	0.2
Completely out of the set	0.0

Using Ragin's per capita income `inc` object, the actual recoding is performed using:

```
incr <- recode(inc, cuts = "1000, 4000, 5000, 10000, 20000",
              values = seq(0, 1, by = 0.2))
```

The choice of recoding intervals are arbitrary and used here only to match Ragin’s qualitative codings, with the same end result.

The next step is to have the computer predict the qualitative codings, based on the researcher’s own codings. One of the statistical techniques which can be used for prediction purposes is the regression analysis: having calculated the intercept and the regression coefficients, for any given value of X (the independent variable) it is possible to predict the value of Y (the dependent variable). In this example `inc` is the independent variable, while the recoded object `incr` will play the role of the dependent variable.

There are many types of regression analyses, depending mostly on the structure of the dependent variable Y , but also on the particular distribution of points in the scatterplot. For example, if Y is a numerical (interval level) variable and the scatterplot is linear, then a linear regression would suffice.

But in our example the dependent variable is hardly a genuine interval-level variable, at least because it only has six unique values (not enough variation) but also, and perhaps most importantly, because it ranges from 0 to 1, which would suggest something closer to a logistic regression. That is also not possible because the dependent variable has to have only two binary values 0 and 1 (round numbers), therefore the most appropriate method to use is a so called fractional polynomial function.

There is a series of papers discussing these kinds of models (Royston and Altman 1994; Sauerbrei and Royston 1999), demonstrating very good fitting capabilities, which is especially useful for this example. For a single independent variable (covariate), the fractional polynomial of degree $m > 0$ for an argument $X > 0$ with powers $p_1 < \dots < p_m$ is defined as the function:

$$\phi_m(X; \beta, p) = \beta_0 + \sum_{j=1}^m \beta_j X^{(p_j)} \quad (4.5)$$

In a regular polynomial function, X is additively raised to the vector of powers p until reaching the degree m . In the case of fractional polynomials, there are two differences:

1. the vector of powers is not restricted to whole numbers, but it is rather taken from a small predefined vector $\{-2, -1, -0.5, 0, 0.5, 1, 2, 3\}$.
2. the round brackets notation (p_j) signals a Box-Tidwell transformation when $p = 0$, using the natural logarithm $\ln(X)$ instead of X^0 .

In R, there are many ways to calculate fractional polynomials, most notably using the package `mfp` designed for multiple covariates. For a single covariate there is a simpler alternative in the `glm()` function that can be used to access the binomial logit family.

Given that all raw values of income per capita are positive, we are only interested in the subset of the special vector $\{0, 0.5, 1, 2, 3\}$. But Royston and

Altman also demonstrated that a fractional polynomial of degree 2 is more than enough to fit the vast majority of data, therefore we are in fact interested in the subset of the vector of powers $\{0, 0.5, 1, 2\}$.

The β_j are the regression parameters, so that a fractional polynomial of second degree with powers $p = \{0, 0.5, 1, 2\}$ has the very similar, regression like form:

$$\beta_0 + \beta_1 \ln(X) + \beta_2 X^{0.5} + \beta_3 X + \beta_4 X^2$$

This equation is specified in the `fracpol` model:

```
fracpol <- glm(incr ~ log(inc) + I(inc^(1/2)) + I(inc^1)+I(inc^2),
              family = quasibinomial(logit))
```

This command executes a (quasi)binomial logistic regression with a fractional polynomial equation, calculating the intercept and the four regression coefficients, which can be inspected using `summary(fracpol)`.

The rest is a simple calculation of the predicted values, based on the `glm` model, using the command `predict(fracpol, type = "response")`. All this procedure is already encapsulated in the function `calibrate()`:

```
cinc <- calibrate(inc, method = "indirect",
                 thresholds = "999, 4000, 5000, 10000, 20000")
round(cinc, 3)
```

```
[1] 1.000 1.000 0.992 0.991 0.957 0.909 0.866 0.817 0.746 0.729 0.714
[12] 0.665 0.548 0.371 0.328 0.326 0.300 0.296 0.287 0.236 0.061 0.035
[23] 0.022 0.004
```

Here, the first threshold is set to 999 instead of 1000, to make a decision about Bolivia which has a GDP of exactly 1000. The resulting values are not an exact match with Ragin's indirect calibrated ones, but that is understandable since Ragin fitted his model using all 136 countries while here we only used a subset of 24, and the implementation of the `fracpoly` function in Stata might be different from this attempt.

However these values, even for a small subset of 24 countries, are remarkably similar to the direct calibrated values and seem to solve some problems, for example faithfully implementing \$20,000 as the threshold for full membership in the set of developed countries (0.95), correctly separating the fifth country Australia (0.963) from the sixth Israel (0.886).

4.3 Calibrating Categorical Data

When introducing the concept of calibration in QCA, Ragin (2008a,b) writes solely about "transforming interval-scale variables into fuzzy sets", and the different methods to obtain the equivalent fuzzy set scores.

While Ragin offers plenty of examples but not a formal definition of the calibration process, Schneider and Wagemann (2012, p. 23) are more general and define calibration as the process of how:

... set membership scores are derived from empirical and conceptual knowledge.

This definition is so large, that it can incorporate basically anything, because empirical and conceptual knowledge is not limited strictly to interval level data, there are all sorts of knowledge being accumulated from both quantitative, and especially from the qualitative research strategies (after all, the “Q” from QCA comes from qualitative, not quantitative comparative analysis).

As we have seen, social sciences present two main classes of data: categorical (composed of nominal and ordinal variables), and numeric (interval and ratio level of measurement). Calibrating numerical data was covered extensively in the previous sections, but there is another trend to transform categorical data into fuzzy sets, and this is often called calibration as well.

The final outcome of the calibration process is a vector of fuzzy, numeric scores. It is important to underline the focus the attention on the word “numeric”, because fuzzy scores are proper numbers between 0 and 1.

With respect to nominal variables (pure, unordered categorical data), they can only be calibrated to crisp sets. It would be impossible to derive continuous, fuzzy numeric scores for categories such as “urban” and “rural”. These types of data can be transformed (“calibrated”) into either binary crisp sets, if there are only two categories, or multi-value crisp sets if there are multiple categories belonging to a certain causal condition.

Binary crisp sets are the most general type of calibrated data, and virtually any kind of raw data (qualitative and quantitative alike) can be transformed to this type of crisp sets. Whether nominal, ordinal, interval or ratio, all of them can be coerced to 1 and 0 values, characteristic for binary crisp sets.

Interval and ratio type of data have already been discussed: with proper care when choosing the inclusion thresholds, numeric raw data can be transformed into either crisp, or fuzzy data using the direct or the indirect methods of calibration.

The other level of measurement which is still open to discussion, is the ordinal level of measurement for the raw data. Ordinal variables are categorical, with a further restriction that categories must be arranged in a certain order.

Some types of ordinal variables, especially those with a very limited number of values, are also very easy to calibrate. A variable called “traffic lights” having three categories “red”, “yellow” and “green” (in that particular order, or reversed, but yellow is always in the middle) is still a very clear categorical variable which can only be calibrated to a multi-value crisp set having three values: 0, 1 and 2, where 0 could mean a complete stop of vehicle movement (“red”), 1 could mean to prepare to stop and slow down speed (“yellow”) and 2 could mean move ahead freely (“green”).

It would not make any sense, and would not serve any practical purpose to transform this type of raw ordinal variable into a pure fuzzy set having values 0, 0.5 and 1 (for many reasons, including the fact that calibration should always avoid the value of 0.5, with more details in the next chapter).

Values 0, 1 and 2 are just as good, and one can find all sorts of necessity and sufficiency claims with respect to one of these values and a given outcome, for example producing and accident or not.

The only type of ordinal data which can potentially be confusing are the Likert-type response scales. These are also categorical, but many researchers seem to have little reservations to calculate central tendency measures such as the mean or standard deviation, typically used for numeric data only.

Although categorical, response values from the Likert type scales are often treated as if they were interval numbers. While I believe this is a mistake, it is a completely different topic than calibration. Especially for a low number of values (there are Likert type response scales with as little as four categories), treating the values as numbers is difficult in general, and it represents an even bigger difficulty for calibration purposes.

But there are two other, major reasons for which calibrating Likert scales is difficult. The first one is related to the bipolar nature of the Likert response scales. While fuzzy sets are unipolar, for example satisfaction, in the case of Likert response scales they are usually constructed from a negative end (1. Very unsatisfied) to a positive end (5. Very satisfied).

A mechanical transformation of a bipolar scale into a uni-dimensional set is likely to introduce serious conceptual questions, as there is no logical reason for which the end “very unsatisfied” should be treated as the zero point in the set of “satisfaction”. Of course, a very unsatisfied person is certainly outside the set of satisfied people, but so can be argued about the mid point (3. Neither, nor) on a five values Likert scales, where only the last two values refer to satisfaction: 4. Satisfied and 5. Very satisfied.

In set theory, satisfaction and dissatisfaction can be treated as two separate sets, rather than two ends of the same set. A person can be both satisfied and unsatisfied in the same time, despite the fact that a single response is given for a Likert type scale.

Exploring the robustness of QCA result applied to large-N data, Emmenegger et al. (2014) used the European Social Survey data wave 2002/2003 to study the attitudes towards immigration in Germany. They have proposed a method to transform a five points Likert type response scale into fuzzy set scores, using exactly this technique to consider values 1, 2 and 3 more out of the set, and values 4 and 5 more in the set (in between defining a region of indifference, as a qualitative anchor point), thus exposing the analysis to the arguments above.

A second, perhaps even more important reason for which Likert type data is difficult to be calibrated as if they were interval, is the fact that many times responses are skewed towards one of the ends. Either in situations where respondents avoid extremes (thus concentrating responses around the mid point), or in situations where people avoid the negative end (thus concentrating responses towards to positive half of the scale), it is possible to find that most responses are clustered around a certain area of the scale. Rarely, if ever, are responses uniformly distributed across all response values.

When responses are mostly clustered (skewed) around three values instead of five, they can easily be calibrated to a multi-value crisp set, but even with five evenly distributed values, it is still possible to construct a such a multi-value crisp set. More challenging are Likert type response scales with more than five values, most often from seven values and up, because at this number it gets increasingly difficult for respondents to think in terms of categories, the distance between values becoming apparently equal.

Skewness is a serious issue that needs to be addressed, even for a seven values Likert response scale. Theoretical knowledge does not help very much, first because such response scales are highly dependent on the specification of the extremes ends (for example, something like “Extremely satisfied” is surely different from “Very satisfied”), and second because there no real guarantee that a particular choice of wording has the same meaning in different cultures.

In any situation, transforming ordinal data into numeric fuzzy scores is highly problematic, and enters a challenging territory even for the quantitative research. To claim that qualitative research can do a better job than its quantitative counterpart, in creating numeric scores is questionable to the very least.

That being said, in a situation where the Likert response scale is large enough (at least seven points), and the responses are more or less evenly distributed across all values, there might be a straightforward method to obtain fuzzy scores from these categorical response values, combining from the quantitative research strategy but insufficiently explored for QCA purposes.

In a study addressing the fuzzy and relative poverty measures, Cheli and Lemmi (1995) seek to analyze poverty in a multidimensional perspective. Their work is relevant because poverty studies have to categorize respondents into poor and non-poor, which is a very similar approach to the fuzzy calibration. For this objective, they propose a method called TFR (totally fuzzy and relative) based on rank orders, thus applicable to both ordinal and interval levels of measurement.

The TFR technique uses an empirical cumulative distribution function on the observed data, and it is best suited to interval level data (a situation already covered by the function `calibrate()`, activating the argument `ecdf = TRUE`). However, when data is categorical (even skewed), they propose a normalized

version by applying a simple transformation to create a membership function that outputs scores between 0 and 1.

The formula below is an adaptation of their function, restricted to values equal to 0 or above, to make sure it can never output negative values (a safety measure also employed by Verkuilen 2005):

$$TFR = \max\left(0, \frac{E(x) - E(1)}{1 - E(1)}\right)$$

$E()$ is the empirical cumulative distribution function of the observed data, and the formula basically calculates the distance from each CDF value to the CDF of the first value 1 in the Likert response scale, and divide that to the distance between 1 (the maximum possible fuzzy score) to the same CDF of the first value 1 in the same Likert response scale.

To demonstrate this with the R code, I will first generate an artificial sample of 100 responses on a seven points response scale, then calibrate that to fuzzy sets using the method just described.

```
# generate artificial data
set.seed(12345)
values <- sample(1:7, 100, replace = TRUE)

# calculate the ECDF
E <- ecdf(values)

# calculate the fuzzy scores
TFR <- pmax(0, (E(values) - E(1)) / (1 - E(1)))

# the same values can be obtained via the embedded method:
TFR <- calibrate(values, method = "TFR")
```

The object TFR contains the fuzzy values obtained via this transformation:

```
table(round(TFR, 3))
```

0	0.151	0.314	0.477	0.605	0.814	1
14	13	14	14	11	18	16

The fuzzy values resulted from this transformation are not mechanically spaced equally between 0 and 1, because they depend on the particular distribution of the observed data. This is very helpful, giving guaranteed suitable fuzzy scores even for highly skewed data coming from ordinal scales.

Chapter 5

Analysis of Necessity



Social phenomena have complex causal configurations. While researchers can advance various hypotheses about how these phenomena are produced, their complete causal mix will probably never be completely understood.

While all causes have an impact in the causal structure, some causes are more important than others. Some are so important, that the outcome doesn't happen in their absence. They might not be sufficient to trigger the outcome on their own, but they are important enough to be a necessary part of the causal mix: whatever the causal combination it contains, the mix will always contain those necessary conditions. From Skocpol's (1979) book on social revolutions, we could derive the following statement: a social revolution (Y) is produced only in the context of a state break down (X). If the state break down would not exist, the social revolution could not be produced; it means the state break down is a necessary (although not always sufficient) cause of a social revolution:

$$X \Leftarrow Y$$

5.1 Conceptual Description

This line of reasoning is valid for all types of sets: binary crisp, multi-value crisp and fuzzy. They can all be graphically represented by a subset/superset relation, as shown in Fig. 5.1.

A necessary set X is a superset of an outcome set Y, which means X is present in all instances where Y happens. Because Y is completely included in X, there is no single instance of Y to be present and X to be absent. There are situations where X is present and Y is not, but the situations where Y is present all happen within the situations where X is present, therefore X is a necessary condition for Y.

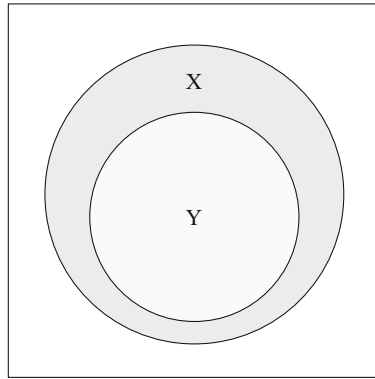


Fig. 5.1 $X \Leftarrow Y$: causal condition X necessary for the outcome Y

The sign “ \Leftarrow ” does not imply any causal direction. Although it looks like an arrow, the sign implies nothing about Y causally leading to X, but it is translated to a logical implication: whenever Y happens, X is present as well.

Braumoeller and Goertz (2000, p. 846) present two complementary definitions about necessity:

Definition 5.1. X is a necessary condition for Y if X is always present when Y occurs.

Definition 5.2. X is a necessary condition for Y if Y does not occur in the absence of X.

In terms of set theory, and in line with the Euler/Venn diagram above, Goertz (2006a, p. 90) supplements with a third, equivalent definition:

Definition 5.3. X is a necessary condition for Y if Y is a subset of X.

There are other types of graphical or tabular representations of necessity, that tell the same story. The simplest one, that involves binary crisp sets, is a 2×2 crosstable to demonstrate the difference between correlation and set necessity relation.

Table 5.1 Correlation (left) and necessity (right)

	0	37
Y		
0	35	0
	0	1
	X	

	0	37
Y		
0	35	23
	0	1
	X	

In Table 5.1, the left side speaks the language of statistics, where the most important numbers are located on the main diagonal. It shows a perfect correlation, because there are no cases on the other diagonal. On the right side, the correlation is broken because of the 23 cases on the other diagonal, but a different kind of (set) language is spoken: given there are no cases where Y is present and X is absent, the set corresponding to Y is completely included within the set corresponding to X, which leads to the conclusion that X is a necessary condition for Y.

The set of X is bigger than the set of Y, containing 60 cases compared with only 37 in Y. Out of those 60 cases, 23 are the ones where X is present but outside of Y (where Y is absent), making the crosstable tell the same story as the one from Fig. 5.1.

Unlike correlation, sets are asymmetric. If a condition X is necessary for Y, the same relation does not automatically hold for their absence. It is not mandatory for the absence of X to be a necessary condition for the absence of Y ($\sim X \Leftarrow \sim Y$), and most often there is another causal condition Z, which is necessary for the absence of Y:

$$Z \Leftarrow \sim Y$$

For this reason, the analysis of necessity should be performed separately for the presence of the outcome Y and for its absence.

To have some words surrounding the numbers in the 2×2 table, we can employ Goertz's (2003, p. 49) very good example for the relation between economic development as condition X and democracy as outcome Y, with the following insightful distinction between a hypothesis based on correlation and another hypothesis based on set necessity:

- the higher the level of economic development, the more likely a country is a democracy
- a minimum level of economic development is necessary for democracy

These are two hypotheses combining the same concepts, but in very different settings. The first one says democracies are bound to happen where the level of democracy is large enough. In practice, however, there are countries where the level of development is large but they did not develop democracies. This kind of contradiction is difficult to explain using correlations, but it makes perfect sense when thinking about set necessity relations: it is true there are nondemocratic countries where there is a large enough level of economic development, but on the other hand there is absolutely no instance of democracy where the level of economic development is very weak.

This empirically observed, asymmetric nature of set relations, can further be extended to demonstrate the necessity relation does not hold for the absence of democracy: there are indeed nondemocratic countries that do not have a

minimal level of economic development (which would seem to support the hypothesis $\sim X \Leftarrow \sim Y$), but there are also nondemocratic countries that are doing very well from an economic point of view.

The lack of economic development is not a necessary condition for the lack of democracy, which is explained by different other factors (an alien conclusion from a statistical, correlational point of view). Statistical methods try to explain the outcome (dependent variable) using a single model for both high and low values of the dependent variable, while QCA finds multiple causal combinations that lead to the same outcome (equifinality).

Necessity relations can be extended from binary to multi-value crisp sets, using the same overall method. In fact, since a binary crisp set is a particular case of a multi-value crisp set with two values, the necessity relation $X \Leftarrow Y$ can also be written (using standard multi-value notation) as:

$$X\{1\} \Leftarrow Y$$

meaning that X is a necessary condition for Y when it takes the value of 1 (i.e., for binary crisp sets, where it is present), if and only if there is no instance of Y being present where $X = 0$ (X is absent).

Multi-value sets can have more than two values, but the same explanation holds for any one. Assuming a set X has three values (0, 1 and 2), then $X\{2\}$ is a necessary condition for Y (Fig. 5.2) iff:

- all cases of Y being present are included in the set of $X = 2$, and
- there is no instance where Y to be present outside the area where $X = 2$ (outcome Y is present only where $X = 2$, and nowhere else)

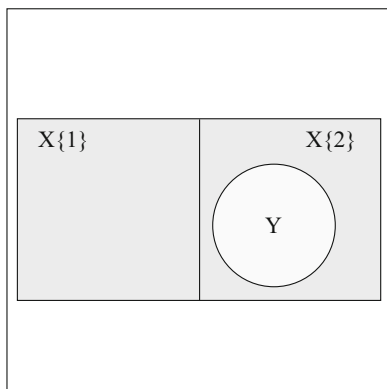


Fig. 5.2 $X\{2\} \Leftarrow Y$: causal condition X is necessary for Y when equal to 2

Such a perfect set inclusion (of Y into X , all democracies have a minimum level of economic development) is also very rare to observe. In reality, there are some countries which develop democracies, despite having a very weak level of economic development.

This is a situation where fuzzy sets prove their usefulness, for it is less important to have absolutely all cases in Y included in the set of X , but more important to have a high percentage of cases of Y inside X (higher than a certain threshold). In classical crisp sets, a single case of Y happening outside of X can undermine the entire necessity claim. But in a situation where this single case would be compared with another 100 cases of Y happening within X , wouldn't this be an overwhelming (albeit not complete) evidence that X is necessary?

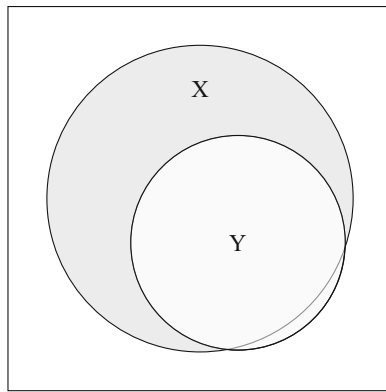


Fig. 5.3 Almost but not complete inclusion of Y into X

Figure 5.3 shows such a situation, where the vast majority of Y cases are included in X , and a few are outside. Y is still included in X , although not completely, but the inclusion is high enough to conclude that X is a necessary condition for Y .

The small part of Y outside of X is the upper left cell of the 2×2 crosstable (Table 5.1). If equal to zero, Y would be completely included in X , and the more cases appear in that cell, the more Y goes outside.

When crisp sets are involved, it is easy to think in terms of number of cases for each cell of the crosstable. But Fig. 5.3 can also be specific to fuzzy sets, since the inclusion itself is not only “in” or “out”, but more or less in (or more or less out).

The inclusion score (its calculation will be introduced in the next section), is any number between 0 (completely outside) and 1 (completely inside), and this is the very essence of a fuzzy set, where the fuzzy scores themselves are numbers between 0 and 1.

When dealing with an infinite number of potential values between 0 and 1, a crosstable between X and Y becomes impossible. For fuzzy sets, necessity relations are not a simple matter of 0 cases in the upper left cell, but a matter of having the fuzzy scores of X higher than the fuzzy scores of Y.

Crosstables are useful for categorical variables (any crisp value can be considered a category), and cases can be counted for each cell of the table. Fuzzy sets are continuous numeric variables, and that kind of data is best represented using a scatterplot, which in QCA jargon is called an XY plot.

When fuzzy scores of X are larger than the fuzzy scores of Y, the points are located in the lower right part of the XY plot. In a way, this is similar to a crosstable, where necessity means there are zero cases in the upper left cell, while in the XY plot necessity means zero points in the upper left part, above the main diagonal.

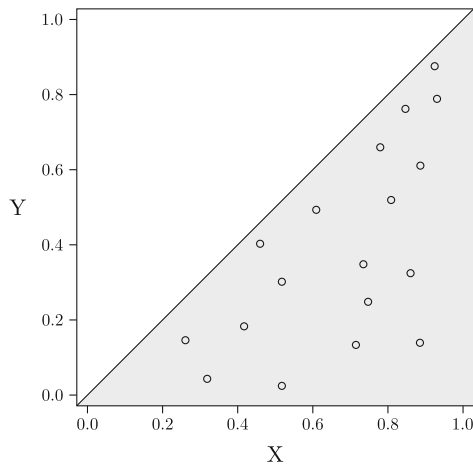


Fig. 5.4 Fuzzy necessity

Figure 5.4 is a classical example of a fuzzy necessity relation, and relate the same story of Y being completely included within X, given that all Y scores are lower than X scores (i.e. all points are located below the main diagonal, in the greyed area), and it corresponds to the Euler/Venn diagram in Fig. 5.1.

If some of the Y values would be greater than corresponding values in X, they would be found outside the greyed area, above the main diagonal, and would correspond to the Venn diagram in Fig. 5.3. To maintain the necessity relation, it is important to have not more than a few cases above the diagonal, or in other words, the proportion of cases below the diagonal has to be very large, or at least larger than a certain threshold.

5.2 Inclusion/Consistency

The term “inclusion” means exactly what the Fig. 5.3 displays: the proportion of the set Y that is included in the set X . It has quite a natural interpretation in terms of fuzzy sets (different from either included or not, but rather more or less included), although for fuzzy sets there are better graphical representations in terms of XY plots.

For the binary crisp case, a 2×2 table can be represented using a general case (Table 5.2).

Table 5.2 General 2×2 table for necessity

		0	1
Y	1	a	c
	0	b	d
		0	1
		X	

The focus, for the analysis of necessity, is on the cells **a** and **c** and the inclusion score is simply calculated with the formula:

$$inclN_{X \leftarrow Y} = \frac{X \cap Y}{Y} = \frac{c}{a + c}$$

This should be read as the intersection between X and Y (both happening, equal to 1 in cell **c**), out of the total set Y .

There are multiple ways to calculate this inclusion score in **R**. Given two binary crisp objects X and Y , the simplest form to calculate is using this command:

```
sum(X & Y)/sum(Y)
```

To demonstrate, we will load the crisp version of the Lipset data, which has the following conditions: DEV, URB, LIT, IND, STB and SURV.

Before going into more details, a complete information about this dataset (and other datasets shipped with the package **QCA**), as well as about individual conditions and outcome can be obtained by typing `?LC` or `heLp(LC)` on the **R** command prompt.

Alternatively, in the graphical the help page can be reached by clicking the “Help” button in the dialog **Load data from attached packages**, as shown in Sect. 2.4.8.

Suppose we want to test the necessity of causal condition URB for the outcome SURV, with the following cross table:

```
data(LC)
with(LC, table(SURV, DEV))
```

```
      DEV
SURV 0 1
0    8 2
1    0 8
```

There are eight cases of SURV happening, and for all of them DEV is happening as well (the R table prints the data from top to bottom, so the relevant line is the one below). The inclusion of the outcome SURV in the causal condition DEV is then:

```
with(LC, sum(DEV & SURV) / sum(SURV))
```

```
[1] 1
```

The logical operator “&” can be employed to calculate the intersection, because binary values are interpreted by R as logical vectors. It assumes of course that both the condition and especially the outcome, are binary crisp sets.

But package *QCA* has a dedicated function called `pof()`, with a default value of the argument `relation = "necessity"`, which calculates the same thing but with more information which will become relevant later:

```
pof("DEV", "SURV", data = LC)
```

```
      inclN  RoN   covN
-----
1 DEV  1.000  0.800  0.800
-----
```

This is a situation where a single condition is necessary for the outcome, with a complete inclusion of the set SURV (survival of democracy) into the causal condition DEV (level of development). The function `pof()` accepts another straightforward way to ask for this necessity relation, using the left arrow “<=” relation:

```
pof("DEV <= SURV", data = LC)
```

```
      inclN  RoN   covN
-----
1 DEV  1.000  0.800  0.800
-----
```

For multi-value conditions, the procedure is quite similar. Since the conditions are also crisp, they can form crosstables with as many columns as the number

of values in the causal condition, while the region of interest remains the same: the row where outcome Y is present with the value of 1 (Table 5.3).

Table 5.3 General crosstable for multi-value necessity

		0	1	2
Y	1	a	c	e
	0	b	d	f
		X		

The inclusion score for each of the values in X (seen as quasi-separate sets, as it was presented in Fig. 5.2), has a very similar calculation method as the one for binary crisp sets:

$$inclN_{X\{v\} \leftarrow Y} = \frac{X\{v\} \cap Y}{Y}$$

It is the intersection between the outcome set Y with the (sub)set formed by a specific value v of the condition X, out of the total number of cases where Y it is present. For example, to calculate the necessity of value 2 of X for the outcome Y, we can calculate the necessity inclusion by dividing the cell e and divide it by the sum of all cells where Y is present:

$$inclN_{X\{2\} \leftarrow Y} = \frac{e}{a + c + e}$$

As a practical example, the Lipset data has a multi-value version:

```
data(LM)
with(LM, table(SURV, DEV))
```

```
      DEV
SURV 0 1 2
  0  8 2 0
  1  0 3 5
```

Calculating the necessity inclusion for the set corresponding to condition DEV equal to 2, can be done either with:

```
with(LM, sum(DEV == 2 & SURV) / sum(SURV))
```

```
[1] 0.625
```

or with:

```
pof("DEV{2} <= SURV", data = LM)
```

		incLN	RoN	covN
1	DEV{2}	0.625	1.000	1.000

There are five cases for DEV equal to 2 where SURV is present, but they do not account for all cases where SURV is present. There are three other cases when DEV is equal to 1, and for this reason none of the multiple individual values of DEV are necessary for SURV.

For fuzzy sets, the calculation method differs (there are no cross-tables to get the number of cases from), but the spirit of the method remains the same. In fact, as we will see, the formula for fuzzy sets can be successfully employed for crisp sets, with the same result.

The Euler/Venn diagrams from Fig. 5.1 or 5.3 are tale telling for all variants, crisp and fuzzy. The necessity inclusion is the proportion of Y in the intersection between X and Y.

Section 3.3.2 introduced the fuzzy intersection and the dedicated function `fuzzyand()`. For exemplification, the fuzzy version of the same Lipset data will be used:

```
data(LF)
with(LF, sum(fuzzyand(DEV, SURV)) / sum(SURV))
```

```
[1] 0.8309859
```

This is a command very similar to those from the crisp sets, and the result is confirmed by the general use `pof()` function:

```
data(LF)
pof("DEV <= SURV", data = LF)
```

		incLN	RoN	covN
1	DEV	0.831	0.811	0.775

The function `fuzzyand()` is universal and it can successfully replace the “&” operator from the binary crisp version:

```
with(LC, sum(fuzzyand(DEV, SURV)) / sum(SURV))
```

```
[1] 1
```

as well as from the multi-value crisp version, when DEV is equal to 2:

```
with(LM, sum(fuzzyand(DEV == 2, SURV)) / sum(SURV))
```

[1] 0.625

This demonstrates that a general fuzzy version for the necessity inclusion equation can be used for both crisp and fuzzy sets:

$$inclN_{X \leftarrow Y} = \frac{\sum \min(X, Y)}{\sum Y}$$

As intuitive as it seems, this formula works counter-intuitive to a Venn diagram. If the set Y has a 0.7 inclusion in a condition X, we would expect the set Y to be 0.3 outside (excluded from) the set X.

For crisp sets it works, but contrary to this expectation fuzzy sets can have surprisingly high “inclusions” in both X and ~X.

Table 5.4 Fuzzy intersections

X	Y	X*Y	~X*Y
0.30	0.20	0.20	0.20
0.50	0.40	0.40	0.40
0.55	0.45	0.45	0.45
0.60	0.50	0.50	0.40
0.70	0.60	0.60	0.30
		2.15	1.75

Table 5.4 presents hypothetical values for a condition X and an outcome Y, with the intersections X*Y and ~X*Y. Using the fuzzy version of the formula, the necessity inclusion for X is 2.15/2.15 = 1, and the necessity inclusion for ~X is 1.75/2.15 = 0.814.

Normally, we do not expect a set to be “included” this much in both X and ~X, thus questioning the term “inclusion” which is more accurately referred to as “consistency”. Using this alternative term, it makes sense to think of both X and its negation ~X as being consistent with Y in terms of necessity.

This situation is called a simultaneous subset relation and is a feature of fuzzy sets, which allow an element to be partially consistent with the presence of a set and in the same time to be consistent with the negation of that set. This is also true in terms of set relations, meaning that a causal condition set can be simultaneously necessary for the presence of an outcome set, as well as for its absence.

In fuzzy sets, it helps to think about a set and its negation as two completely different sets: they are complementary, but unlike crisp sets where an element either in or out of a set (rule of excluded middle), in fuzzy sets an element is

allowed to be part of both a set and its negation, and in some situations an element can have high inclusions in both.

In terms of consistency, a set X is necessary for a set Y when the fuzzy values of Y are consistently lower than the fuzzy scores of X across all cases (when the fuzzy values of Y consistently display a subset relation with X). In such a situation, the set Y is included in the set X because most of their intersection belong to Y (or cover Y), therefore we can say the necessity consistency is high.

5.3 Coverage/Relevance

Coverage is a measure of how trivial, or relevant is a necessary condition X for an outcome Y . The classical example of the relationship between fire and the air, says that air (oxygen) is necessary to start a fire. But this is an irrelevant necessary condition, for a fire is not started from the mere presence of the oxygen. That is necessary to maintain it, but does not start it as there are many other situations when air is present without a fire.

In terms of Euler/Venn diagrams, trivialness can be detected by measuring the proportion of the area within condition X that is covered by the set Y , as seen in Fig. 5.5.

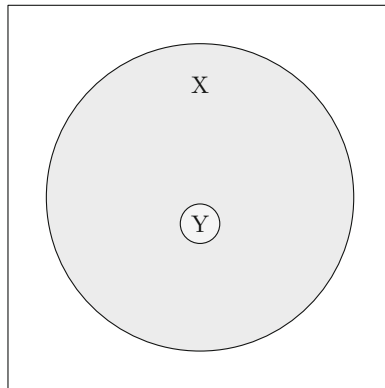


Fig. 5.5 X as an irrelevant necessary condition for Y

The outcome Y is a very small set, compared to the necessary condition X : there are very many cases (practically most) where X is present but Y does not happen. This is a typical description of an irrelevant necessary condition: although Y is completely included in the set X , its coverage is very small.

For crisp sets, this is the equivalent of a 2×2 table where the number of cases in cell *c* (where both *X* and *Y* are present) is very small compared to the number of cases in cell *d* where *X* is present and *Y* is not.

Table 5.5 Testing trivialness and relevance in a 2×2 table

	0	5
Y	25	120
	0	1
	X	

	a	c
Y	b	d
	0	1
	X	

The formula to calculate necessity coverage, for binary crisp sets, is:

$$covN_{X \leftarrow Y} = \frac{X \cap Y}{X} = \frac{c}{c + d}$$

This is interpreted as the proportion of *X* covered by its intersection with *Y*. For necessity coverage, the attention is focused on the cells located on the right column, instead of the first row as in the necessity inclusion.

In the hypothetical crosstable (Table 5.5), *X* is a perfect necessary condition for *Y* (it is completely included within *X*, there are no cases in cell *a*), but is an irrelevant one because the proportion covered by their intersection $5/125 = 0.04$ is extremely small. Only 4% of *X* is covered by *Y*, given there are many cases in cell *d* where *Y* is not present.

For most people, air is a trivial necessary condition for fire. If Table 5.5 would refer to the relation between air (*X*) and fire (*Y*), the small coverage supports our empirical knowledge that fire does not start from thin air. However, there are many cases in the cell *b* (let's say 25 attempts to start a fire in the absence of oxygen), but that number can be close to plus infinity if we think about the extra-terrestrial, outer solar system conditions in the rest of the Universe, where we don't observe oxygen and also don't observe fire.

Analyzed from this perspective, air does not appear to be a trivial condition for fire, as fire and air are highly associated with one another. If we could observe this relationship from outside the Universe, fire is always observed in the presence of oxygen, while at the same time it can be observed the vast majority of (the rest of) the space where there is no oxygen and no fire.

Contrary to the common sense perception, air is actually a non-trivial necessary condition for fire. Empirically, we do know that air is irrelevant because it cannot cause fire, as there are also very many cases of air without fire, but their relation is far from trivial.

Goertz (2006a) has a welcome contribution to the analysis of necessity, making a difference between the concepts of trivialness and relevance. His work is fundamental for the modern QCA, following Ragin and in the same time pivotal for Ragin's later developments.

He relates trivialness to cell **b** of the table and relevance to cell **d**, both of them being found in the bottom row where *Y* is absent. The earlier work of Braumoeller and Goertz (2000) make this point very clear, designing a series of tests to evaluate trivialness of (proven) necessary conditions using a χ^2 test of homogeneity applied to a contingency (2×2) table.

While Braumoeller and Goertz employed the χ^2 test of homogeneity, I believe they have in fact employed the χ^2 test of independence (the difference is subtle, and the calculation method is exactly the same) for which the null hypothesis is that two categorical variables are independent of one another.

Their analysis boils down to stating that a necessary condition *X* is trivial if *X* and *Y* are independent (if the null hypothesis cannot be rejected).

Given that cell **a** is equal to zero (because *X* is necessary), this conclusion is true only if cell **b** is also equal to zero (that is, we have no empirical evidence of $\sim X$ and $\sim Y$ occurring together). Put differently, a necessary condition *X* is trivial if and only if there is no empirical evidence of $\sim X$, as in the Venn diagram from Fig. 5.6.



Fig. 5.6 Trivial necessary condition *X* for the outcome *Y*

It looks as though condition *X* is missing, but in fact the necessary condition is so large that it covers the entire universe defined by the outer square, while the outcome *Y* is still a perfect subset of the condition *X*.

Returning to the Lipset data, we may explore the relation between LIT (level of literacy) and SURV (survival of democracy) for the crisp version:

```
tbl <- with(LC, table(SURV, LIT))
tbl
```

```
      LIT
SURV 0 1
     0 5 5
     1 0 8
```

The condition LIT is a necessary condition, with a necessity inclusion of 1 for the outcome SURV (0 cases occurring when democracy survives in the absence of literacy). It is now possible to test for trivialness, by testing their independence.

Two events are said to be independent, if the occurrence of one does not affect the odds of the other occurring, which is the spirit of the Fisher's exact test of independence between two categorical variables.

```
fisher.test(tbl)
```

Fisher's Exact Test for Count Data

```
data: tbl
p-value = 0.03595
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.9305029      Inf
sample estimates:
odds ratio
      Inf
```

With a significant p -value of 0.03595, we can reject the null hypothesis of independence between rows and columns.¹ There are other functions, for example `fisher.exact()` in package `exact2x2` that do a better job in calculating the confidence interval, but the overall decision to reject the null hypothesis is the same and decide with 95% confidence that survival of democracy is not independent of literacy (making it non-trivial).

It is perhaps important to stress, one more time, that a test of trivialness can only be performed if the condition is necessary, that is to have a very small proportion of cases (towards 0) in the cell **a** where Y is present and X is absent.

Relevance, on the other hand, is directly tested by the coverage of X by Y: a higher proportion of cases in cell **d** (X occurring but not Y) relative to cell **c** (X and Y occurring together), lowers the relevance of the necessary condition. A necessary condition becomes more and more relevant as the proportion of cell **c** increases, except for the situation when both X and Y become irrelevant

¹ The null hypothesis that true odds ratio is equal to 1 is rejected despite the fact the confidence interval contains the value of 1.

if they are so large that they both fill the entire universe (both being constant and omni-present).

In the Lipset case, although LIT is definitely a necessary condition it is not the most relevant one, given the raw coverage of 0.615:

```
pof("LIT <= SURV", data = LC)
```

	inclN	RoN	covN
1 LIT	1.000	0.500	0.615

The function `pof()` returns, for necessity relations, another parameter of fit called `RoN` (Relevance of Necessity) that will be discussed below, when calculating the fuzzy version of the coverage. For the moment, it is sufficient to notice that literacy has a relevance score of 0.5 (even lower than its coverage).

The lower the `RoN` score, the more trivial a condition is, and the higher the `RoN` score the higher the relevance, the bigger the relative importance of that condition as a necessary condition. This conclusion is valid for both binary and multi-value crisp.

Extending the test of trivialness for multi-value data is just as simple, because multi-value data can be simplified to a binary situation by considering each individual data versus everything else (versus its complement). Let us examine again the relation between `DEV` and `SURV` in the multi-value version of the Lipset data:

```
with(LM, table(SURV, DEV))
```

	DEV		
SURV	0	1	2
0	8	2	0
1	0	3	5

There are exactly zero cases in cell `a` (remember that R table prints the rows in the reversed order), where `SURV` is present and `DEV` is completely absent. But that does not make any of the other two values of `DEV` necessary for the survival of democracy, their inclusion being rather modest:

```
pof("DEV{1} + DEV{2} <= SURV", data = LM)
```

	inclN	RoN	covN
1 DEV{1}	0.375	0.867	0.600
2 DEV{2}	0.625	1.000	1.000
3 expression	1.000	0.800	0.800

That happens because $DEV\{1\}$ and $DEV\{2\}$ behave as self-contained sets, despite the fact they are two categories of the same causal condition. The necessity inclusion of $DEV\{1\}$ for SURV is calculated by taking the cases from $DEV\{1\}$ in opposition to all other categories $DEV\{0\}$ and $DEV\{2\}$.

We can arrive at exactly the same result by collapsing all other categories to create a familiar, binary crisp 2×2 table:

```
DEV1 <- recode(LM$DEV, "1 = 1; else = 0")
table(LM$SURV, DEV1)
```

```
DEV1
  0 1
0 8 2
1 5 3
```

We can use this separate object DEV1 to calculate its necessity inclusion for the outcome SURV from the data LF (notice how the structure of the command is changed), with the same end result:

```
pof(DEV1, LM$SURV)
```

		inclN	RoN	covN
1	DEV1	0.375	0.867	0.600

Since no single value of condition DEV is individually necessary, it doesn't make sense to test for trivialness. The RoN score of 0.867 is high, but it does not matter since the value 1 of condition DEV is not individually necessary for the survival of democracy.

The fuzzy situation is more challenging, similar to the necessity inclusion: since both conditions and outcome have fuzzy scores, there is no contingency table around to apply these specific calculations.

Much like in the case of inclusion, fuzzy coverage is about the proportion of X that is covered by Y, or better said covered by the intersection between X and Y, given that Y already is a (perfect) subset of X, a very easy task using the same fuzzy intersection, but this time dividing over the sum of X:

$$covN_{X \leftarrow Y} = \frac{\sum \min(X, Y)}{\sum X}$$

Using the fuzzy version of the Lipset data, the necessity coverage for literacy LIT and the outcome SURV is:

```
with(LF, sum(fuzzyand(LIT, SURV)) / sum(LIT))
```

```
[1] 0.6428027
```

This function holds for all versions, crisp and fuzzy. Remembering that coverage for the crisp version of LIT was 0.615, we can test this with:

```
with(LC, sum(fuzzyand(LIT, SURV)) / sum(LIT))
```

```
[1] 0.6153846
```

Naturally, one need not go through all these individual calculations, as the function `pof()` already provides both inclusion and coverage in a single output:

```
pof("LIT <= SURV", data = LF)
```

		inclN	RoN	covN
1	LIT	0.991	0.509	0.643

Once we know that LIT is a necessary condition, trivialness was detected in the crisp version if there were (close to) zero empirically observed cases in the $\sim X$ column. But this kind of finding is also valid the other way round: a necessary condition X is trivial if all observed cases are located in the presence column where $X = 1$.

For fuzzy sets necessity, all (most) cases are located below the main diagonal, and trivialness is detected just like in the crisp version, when all cases are located vertically on the right side of the XY plot, where $X = 1$. If all cases would be constantly equal to 1 in the condition X , then no matter what fuzzy values Y would take, X would still be a superset of Y (hence a necessary condition), but it would be a trivial one because X is constant and omnipresent (Fig. 5.7).

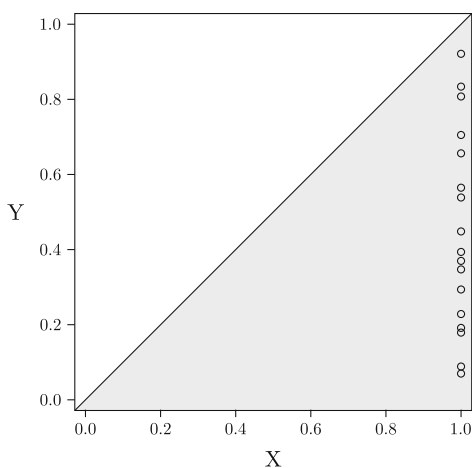


Fig. 5.7 Fuzzy trivialness of the necessary condition X

The more values come closer to the main diagonal, and away from the right side of the plot, the more relevant the condition X becomes, as a necessary condition for Y . To the limit, when points become perfectly aligned with the main diagonal, X becomes fully relevant not only as a necessary, but also as a sufficient condition for Y . In the crisp 2×2 table, this is the equivalent of all cases from cell **d** being moved in cell **b**, signalling a perfect correlation.

Goertz (2006a, p. 95) was the first to propose a measure of trivialness, by measuring the distance between the fuzzy value and 1:

$$T_{nec} = \frac{1}{N} \sum \frac{1-X}{1-Y}$$

It is a measure of trivialness, but at the same time a measure of relevance (what is not trivial, is relevant). The more this measure moves away from 0, the more relevant the necessary condition is.

Later, Schneider and Wagemann (2012) observed that in some cases, this formula can produce values above 1 (which do not make a lot of sense for a fuzzy interpretation), hence they have proposed a modified version which is the default `RoN` parameter of fit from the output of the `pof()` function:

$$RoN = \frac{\sum(1-X)}{\sum(1-\min(X,Y))}$$

The distance between 1 to X is divided by the distance between 1 to the intersection of X and Y , and since the intersection is a smaller value (taking the minimum between X and Y), the denominator is always greater or equal to the numerator, hence this parameter will never exceed the value of 1.

To demonstrate this with the necessity relation between `LIT` and `SURV`:

```
with(LF, sum(1 - LIT) / sum(1 - fuzzyand(LIT, SURV)))
```

```
[1] 0.5094142
```

The function `XYplot()` in package `QCA` creates an XY plot between a condition and an outcome, and at the same time presenting all parameters of fit. Just like the `pof()` function, it has an argument called `relation` which is by default set to `sufficiency`.

```
XYplot(LIT, SURV, data = LF, jitter = TRUE, relation = "necessity")
```

The code above creates an XY plot with slightly jittered points using the argument `jitter = TRUE` (because some of them are so close that become overlapped). The condition `LIT` is clearly a necessary condition for `SURV`, with an inclusion score of 0.991, but its relevance of a bit more than 0.5 is rather modest. This happens because many of the points below the main diagonal have very high values on `LIT` (close to or equal to 1), therefore they are constantly located close to the right margin of the plot.

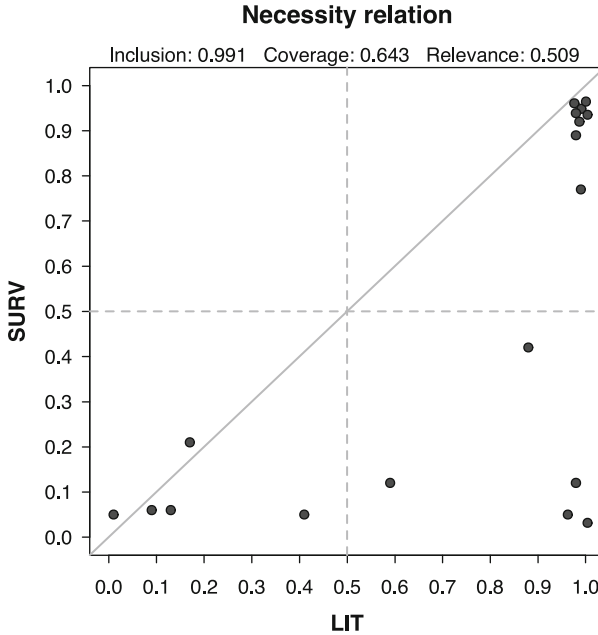


Fig. 5.8 Literacy as a necessary condition for the survival of democracy

The average distance of the points from 1 gets smaller with each point close to the right margin, which explains the semi-trivialness of the literacy as a necessary condition for the survival of democracy.

Schneider and Wagemann (2012) point that a low relevance score should be accompanied by studying the deviant cases for necessity. There are two middle, dotted lines on the XY plot (a horizontal and a vertical one) that split the fuzzy area into something which resembles a 2×2 crosstable.

A perfect relevance of necessity is met when cases are found in cells b (lower left) and c (upper right). For necessity, cases found in cell a (upper left) are called deviant cases consistency in kind, having a too large value on the outcome Y, and a too low value on the condition X.

In the example from Fig. 5.8, there are no deviant cases consistency *in kind*, in the upper left part of the plot. Schneider and Rohlfling (2013) introduce yet another type of deviant cases consistency, namely *in degree*: for necessity statements, those where both values (for the condition and for the outcome) are greater than 0.5, but the outcome value is greater than the condition's value (thus invalidating the subset relation of the outcome in the condition). The cases would be located in cell c (upper right), but above the diagonal.

5.4 Necessity for Conjunctions and Disjunctions

For complex outcome phenomena, single conditions are rarely necessary. Many times, a condition becomes necessary in combination with another condition, either in conjunction, which is essentially an intersection of two or more sets, or in disjunction which is a union of two or more sets.

Conjunctions are rather easy to interpret: if a conjunction of two sets A and B is necessary for the outcome, it means that both A and B are necessary on their own, since the intersection is part of both A and B , as in Fig. 5.9.

Since the outcome Y is located (more or less completely) inside the intersection AB , it is also located within A and within B separately, which means that specifying the atomic expressions A and B is redundant, since their conjunction logically implies both of them individually.

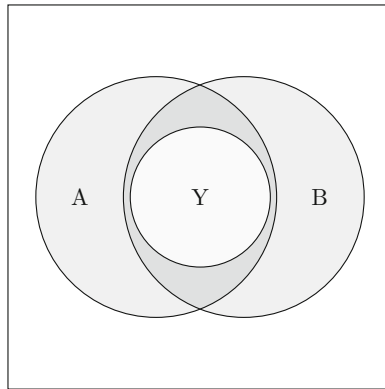


Fig. 5.9 Conjunction AB necessary for Y

A more interesting situation is when conditions A and B are both individually necessary, but their conjunction is not necessary (i.e. the outcome Y is not sufficiently included in their intersection). Figure 5.10 is somewhat similar to Fig. 5.3, but extended to conjunctions: the outcome Y is almost (but not completely) included in A , and with the same amount (again not completely) included in B .

For both A and B , the inclusion of outcome Y is high enough to decide that both A and B are individually necessary. But their intersection is small enough, and Y large enough to get outside the intersection, such as the proportion of Y within AB is not high enough to conclude the conjunction is necessary.

For both of these reasons (incomplete inclusion, or if completely included the conjunction is redundant) it is actually very rare, if ever, to observe a non-redundant necessary conjunction. But in both these situations, it is obvious

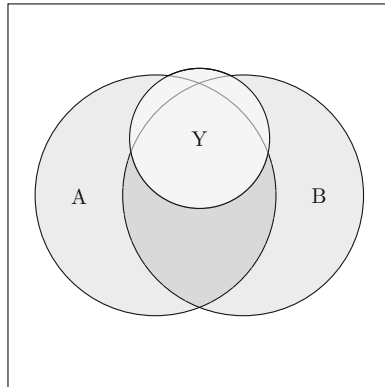


Fig. 5.10 A and B individually necessary, but not their conjunction AB

that Y is completely included in the disjunction (union) of the sets $A + B$, although for the same arguments this particular disjunction is also redundant because both A and B are individually necessary and the simpler expressions are always preferred.

There are many situations (actually, most frequently) when no sets are individually necessary, and implicitly neither their conjunction(s). But there will always be a disjunction of conditions which is higher than the outcome set Y, because unions form larger and larger areas with every causal condition added to the disjunction. At some point, the union (the disjunction) will be so large that a quasi complete inclusion of outcome Y in that disjunction is bound to happen.

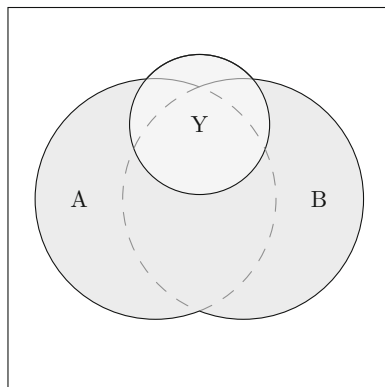


Fig. 5.11 $A + B \Leftarrow Y$: the union of A and B is necessary for Y

Figure 5.11 presents such a situation, where the outcome Y is sufficiently outside A, and sufficiently outside B, to render both these conditions as individually not necessary.

But the outcome is sufficiently included in their disjunction $A + B$, since the union of these sets is a much larger area than each of individual sets, so large that help each other covering Y to a much larger proportion.

Disjunctions are easy to construct, through the logical OR operation presented at Sect. 3.3.3 (taking the maximum of each pair of values), and in R using the function `fuzzyor()`. But this easiness must be taken with a grain of salt, however, because interpreting these disjunctions is not such a trivial matter as it first might seem.

Theoretical meaning of conjunctions is straightforward: they are intersections of known concepts, and will be discussed in depth in the next chapter at the analysis of sufficiency. A possible hypothesis involving conjunctions is: democratic countries (D) which have strong economic relations (E) don't go to war against each other ($\sim W$). This isn't a necessity statement (conjunctions are redundant under necessity), but it shows clearly at which conjunction (subset) it refers: countries which are both democratic, AND having strong economic relations with other democratic countries: $D \cdot E$.

Disjunctions, on the other hand, need a more careful interpretation, for they usually result in unions referring to a higher order concept that is different from the simple union of the constituent concepts. A disjunction $D+E$ such as: "democracies OR having strong economic relations" is something different from the mere juxtaposition of D and E.

Disjunctions are higher order concepts, sometimes called super-concepts, formed by the union of two or more concepts and their analytic meaning depends on theory and researchers' expertise in the field. There is no mechanical or "one size fits all" interpretation for all cases. Some disjunctions might be theoretically meaningless, and in various contexts they might not satisfy the criteria for the relevance of necessity.

Care must be taken to avoid automatic super-concepts. The next section shows how to use the function `superSubset()` to find all possible necessary expressions from a certain data, but those are expressions derived from a data specific environment, and not all of them have theoretical meaning.

Computers are very good at finding everything that meets certain criteria, but it is the job of the researcher to sift through computer's results and select those which do have some meaning. Failing to do so is similar to fishing for data: one can derive many conclusions based on a certain dataset, while some other data might give more or less different conclusions.

It is not the data which should be leading the research process, the correct approach is to test whether certain theoretical expectations are met in a particular dataset.

5.5 Exploring Possible Necessity Relations

The normal analysis of necessity involves specific tests for each causal condition of interest, as well as each possible larger (but theoretically meaningful) superset disjunction. This can be time consuming, and sometimes we might want to simply explore the data for possible necessity relations.

The package *QCA* offers a useful function called `superSubset()`, which does all this works automatically. It explores every possible necessity relation, for individual conditions, or conjunctions (even though conjunctions are redundant), as well as all possible disjunctions of conditions that are necessary for a given outcome.

When the number of causal conditions is large, it can be very helpful to get an overview of all possible necessity relations, eliminating useless tests on expressions that are not necessary for the outcome.

Using the same fuzzy version of the Lipset data, the command can be used as simply as:

```
superSubset(LF, outcome = "SURV", incl.cut = 0.9)
```

		inclN	RoN	covN
1	LIT	0.991	0.509	0.643
2	STB	0.920	0.680	0.707
3	LIT*STB	0.915	0.800	0.793
4	DEV+urb	0.964	0.183	0.506
5	DEV+ind	0.964	0.221	0.518
6	DEV+stb	0.912	0.447	0.579
7	urb+IND	0.989	0.157	0.511
8	DEV+URB+lit	0.924	0.414	0.570
9	DEV+URB+IND	0.903	0.704	0.716
10	DEV+lit+IND	0.919	0.417	0.569

The analysis is performed for the default value of the argument `relation = "necessity"` (which does not need to be formally specified, since it is default), and a similar analysis can be performed for the sufficiently relation, changing it to `"sufficiency"`, or even for relations which are both necessary and sufficient, via `"necsuf"`, or even `"sufnec"` if an analysis of sufficiency is used first.

It is extremely important to make it clear the list of necessary expressions is automatically generated in a mechanical way, the underlying algorithm merely crunching numbers in a process of indiscriminate search for all possible necessity relations. It is the researcher who should make sense out of all these disjunctions, and select only those which have a theoretical meaning.

For the moment, it is sufficient to notice that two atomic conditions were found as individually necessary for the outcome, as well as their conjunction, and

the disjunctions presented in the output are non-redundant (there are more necessary disjunctions, but redundant as subsets of the ones found above).

Another important aspect to observe is that coverage is relatively low for all expressions, and for some expressions their relevance is even closer to zero. For such situations, one possibility is to specify a cut-off for the relevance of necessity, for example list only those expressions with a decent relevance threshold of at least 0.6:

```
superSubset(LF, outcome = "SURV", incl.cut = 0.9, ron.cut = 0.6)
```

		inclN	RoN	covN
1	STB	0.920	0.680	0.707
2	LIT*STB	0.915	0.800	0.793
3	DEV+URB+IND	0.903	0.704	0.716

This is an interesting example that is worth discussing, for several things are happening and all are important. To begin with, it can be noticed that the conjunction LIT·STB is necessary, but the atomic condition LIT is not part of this list of relevant necessary expressions.

Although a bit counter-intuitive, it does makes logical sense. If a conjunction is necessary, this logically implies the atomic conditions are also necessary. And that is absolutely true, since the inclusion score for the atomic condition LIT is 0.991, which is a very high inclusion score that without any doubt renders LIT as a necessary condition.

But necessary conditions are not always relevant. Air is a necessary condition for a battle, but it is also completely irrelevant for the set of conditions that lead to a battle. This is a similar situation, where LIT (literacy) is a necessary but irrelevant condition for the outcome SURV (survival of democracy) in the inter-war period.

This means the condition LIT is a high enough set, and the outcome SURV is small enough to fit inside the intersection with STB (government stability), and the intersection itself is small enough to cover SURV to such an extent that the intersection is relevant and the atomic condition LIT is not.

Depending on how large the outcome set is, and where it is located within a necessary causal condition, sometimes the atomic condition is necessary but a conjunction with another condition is not necessary. Some other times, as in this example, the conjunction is necessary and relevant and the atomic condition is also necessary but not relevant.

This is why, in the results from the function `superSubset()`, there might be both atomic conditions and their conjunctions listed as necessary conditions but granted, when the conjunction itself is both necessary and relevant, all superset conditions are redundant. In such a situation both LIT and STB are

implied by the conjunction LIT·STB, and they could as well be removed from the list of necessary expressions.

The opposite situation happens for the sufficiency statements (to be discussed in the next chapter), where if an atomic condition is sufficient for an outcome, any of its subset conjunctions are redundant because they are logically sufficient as part of the larger, atomic condition. In necessity statements, the reverse happens that necessary conjunctions make the atomic conditions redundant.

A third and perhaps most important thing that happened is that most of the disjunctions disappeared from the resulting list of relevant necessary expressions. Although necessary, some of them are highly irrelevant, for example the disjunctive expression `urb+IND` with a very high inclusion score of 0.989 but a very low relevance score of 0.157.

This is a perfect example of illogical disjunction, for it is very difficult to make any logical and theoretical sense for the union between low urbanisation (`urb`) and high industrialisation (`IND`), and their relationship as a necessary expression for the survival of democracy. Similar illogical unions are those between the high level of development (`DEV`) and low level of urbanisation (`urb`) or with a low level of industrialisation (`ind`).

These are all textbook examples of disjunctive necessary expressions that are meaningless constructs for the outcome of interest. Superset constructs should not be automatically considered relevant, just because they have a high inclusion score under a necessity statement.

Such a misjudgement was made by Thiem (2016), using precisely this Lipset dataset in the attempt to demonstrate that necessary expressions should not be used when identifying incoherent counterfactuals for the ESA—Enhanced Standard Analysis (see Sects. 8.6 and 8.7), because they lead to an alleged effect named *CONSOL* which states that removing such counterfactuals would generate the conservative solution instead of the enhanced intermediate one.

In order to generate the conservative solution, most if not all remainders should be blocked from the minimization process, therefore he employed the entire list of necessary expressions resulting from the function `superSubset()` irrespective of their relevance or their theoretical and even logical meaning.

By doing so, Thiem demonstrates a lack of understanding of how disjunctive necessary expressions should be interpreted and employed in further analyses. Ignoring the already existing standards of good practice, he abused the list of mechanically generated necessary expressions and failed to consider that a methodology which does not serve a theoretical purpose is just a meaningless display of skills.

Chapter 6

Analysis of Sufficiency



The analysis of sufficiency is the main purpose of the QCA methodology, to find the minimal configurations of conditions that are sufficient for a given outcome. All main algorithms, to create a truth table, or to produce a logical minimization (to name just the more important ones) are designed around finding sufficiency relations.

Initially, this chapter contained a lot of information from all of these topics and grew disproportionately larger than other chapters, so much so that it almost had a contents of its own. Although topics like truth tables and minimizations are integral part of the analysis of sufficiency, they ended up as separate chapters to treat them in-depth and make a more clear distinction between these important topics.

The current chapter introduces the basic concepts of the sufficiency that are structurally very similar to the ones presented for the analysis of necessity. As it will be unfolded in the next sections, especially in the conceptual description, necessity and sufficiency are two complementary and actually mirrored concepts. For this reason, much of the common details are left out to concentrate on the important differences.

Just like necessity statements, the literature abounds with claims or hypotheses involving the sufficiency of a cause or a configuration of causes over an outcome of interest. When studying an outcome of interest, necessity statements are important but sufficiency hypotheses resemble the most what we usually conceptualize in terms of causes and effects.

It is therefore not a coincidence that sufficiency statements are denoted by the forward arrow “ \Rightarrow ” sign. Given a cause X and an outcome Y , the statement “ X is sufficient for Y ” is written as:

$$X \Rightarrow Y$$

6.1 Conceptual Description

Just like in the case of necessity, the sufficiency claim can be graphically represented by a subset/superset relation, as in the Venn diagram from Fig. 6.1. If X is a sufficient condition for Y , then whenever Y is present, X is present as well.

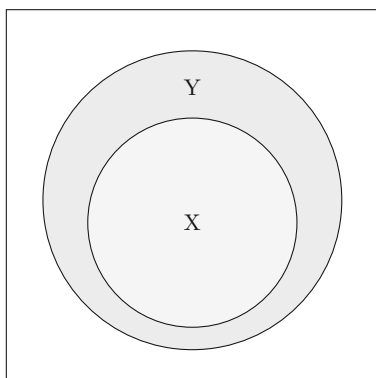


Fig. 6.1 $X \Rightarrow Y$: causal condition X sufficient for the outcome Y

In terms of set relations, in the sufficiency claim Y is a superset of X , which means that X is never present in the absence (outside) of Y . When this relation is met, whenever X is present, Y is also present: there are situations where Y is present in the absence of X , but in every situation where X happens, Y happens as well. In other words, if we know that X is present, it is guaranteed that Y will be present as well (as a sufficient condition, X guarantees Y to happen).

The fact that Y is a bigger set than X (and it usually is the case) is an indication that no single condition explains all of Y . There might be other conditions, or other combinations of conditions which can explain the rest of the set Y , and the only possible way that X would cover all of Y is the situation when X is both necessary and sufficient for Y , a situation when both sets are equally large.

The complementarity between necessity and sufficiency relations becomes evident from yet another perspective: if X is a *sufficient* condition for Y , then the absence of the condition X is a *necessary* condition for the absence of Y . In terms of set representations, if X is a subset of Y then $\sim X$ covers a much larger region than $\sim Y$, therefore $\sim X$ is a superset of (necessary for) $\sim Y$. For binary crisp sets, the same thing can be represented using 2×2 crosstables (Table 6.1).

It should now be clear that, if X is sufficient for Y and $\sim X$ necessary for $\sim Y$, due to the asymmetrical nature of set relations this is not a proof that

Table 6.1 $X \Rightarrow Y$ (left) and $\sim X \Leftarrow \sim Y$ (right)

		49	58
Y	1		
	0	23	0
		0	1
		X	

		49	58
Y	1		
	0	23	0
		0	1
		X	

$\sim X$ should be sufficient for $\sim Y$, nor that X should be necessary for Y . Two separate analyses of sufficiency must be performed for the presence and for the absence of the output, leading to different sufficiency statements (expressions).

Mirroring the definitions from necessity, the following might be used for sufficiency:

Definition 6.1. X is a sufficient condition for Y when every time X is present, Y is also present (Y is always present when X occurs).

Definition 6.2. X is a sufficient condition for Y if X does not occur in the absence of Y .

And in terms of set theory, the following equivalent definition:

Definition 6.3. X is a sufficient condition for Y if X is a subset of Y .

Knowing that X occurs is enough (sufficient) evidence to know that Y occurs. As a concrete example, following Weber’s (1930) well known theory about the relation between the Protestant ethic and the spirit of capitalism, we could imagine at aggregate community level, having a Protestant ethic (causal condition X) is sufficient to accumulate capital (outcome Y). Naturally, capital can be accumulated in many other ways, therefore a Protestant ethic is not a necessary condition to accumulate capital, but knowing that a group lives through this kind of ethic and abides certain Protestant traits, is sufficient to understand that group must be accumulating capital.

If necessity for multi-value conditions are a bit difficult to understand (Y having to be a subset of a specific value of X , not a subset of X as a whole), in the case of sufficiency things are a lot more straightforward. Assuming a multi-value condition X has three values (0, 1 and 2), then a sufficiency for value 2 is written as:

$$X\{2\} \Rightarrow Y$$

As it turns out, $X\{2\}$ is sufficient for Y iff:

- all cases where X is equal to 2 are included in the set of Y , and
- there is no instance where of X equal to 2 outside the set of Y (X takes the value 2 only inside Y , and nowhere else)

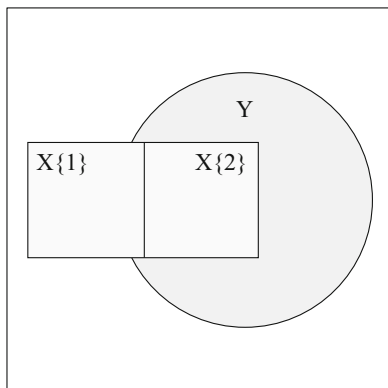


Fig. 6.2 $X\{2\} \Rightarrow Y$: causal condition X is sufficient for Y when equal to 2

Only $X\{2\}$ is a subset of the outcome Y , while both $X\{1\}$ and $X\{0\}$ can happen in the absence of Y , therefore only $X\{2\}$ is a sufficient information to conclude that Y occurs in its presence (Fig. 6.2).

Admittedly, perfect subset relations are rare to happen in the real world. Instead, most of the times a set is more or less included into another set, a topic that will be extensively covered in the next section. For the moment, partial inclusions can be used as decision criteria to conclude that one “set” is sufficient for another, as displayed in Fig. 6.3 below.

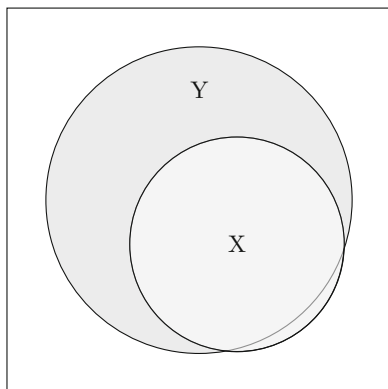


Fig. 6.3 Almost but not complete inclusion of X into Y

This would be a good moment to pause and reflect upon the sufficiency relation $X \Rightarrow Y$: most readers would understand that “a” (single) causal condition X is sufficient for the outcome Y , and most of the times this is true. But “ X ” is only a notation, and the meaning of the word “set” should not be confused with an automatic equivalence with a “single” causal condition.

A set may refer to many things, including entire conjunctions and/or disjunctions of multiple causal conditions. Imagine a situation with two causal conditions A and B, conjunctively sufficient for an outcome Y. Replacing X with A·B, then $X \Rightarrow Y$ in fact means $A \cdot B \Rightarrow Y$, with the “set” X being represented by the conjunction A·B. The same thing applies for an entire expression like $A \cdot B + C$, interpreted as a “set” in its entirety.

Although Venn diagrams such as the one from Fig. 6.3 can tell a story about partial membership of a set X into Y, such diagrams are designed for crisp sets only. To be more precise, they are designed for *bivalent* crisp sets, as there are no Venn diagrams for multivalent sets.

In case of fuzzy sets, a similar representation for sufficiency can be graphically represented through an XY plot. Figure 6.4 is another proof that necessity and sufficiency are mirrored: in the necessity case the points should be found below the main diagonal, while in the sufficiency case the points should be located in the grey area *above* the diagonal.

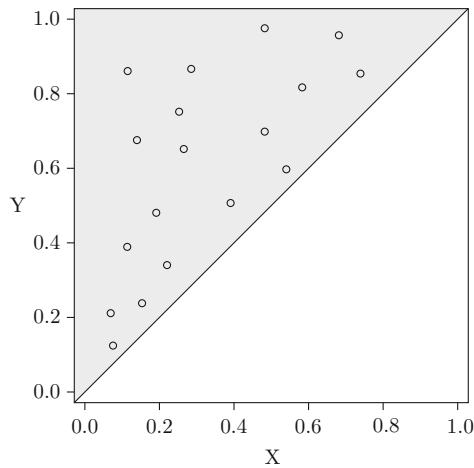


Fig. 6.4 Fuzzy sufficiency

This is also a superset/subset relation, since all values in X are lower than the corresponding values in Y, therefore Y is a fuzzy superset of X. Figure 6.4 is about a perfect fuzzy superset relation, given that all points are located above the diagonal, but there are situations where a small proportion of the points are located below the diagonal, lowering the inclusion score.

6.2 Inclusion/Consistency

Inclusion and consistency are two words that refer to the same thing. If the inclusion of X (into Y) is high, then we say that X is highly consistent, or X has a high consistency score. The term “consistency” is a synonym for the word “inclusion”, and they are often used interchangeably.

For binary crisp sets, the same kind of general 2×2 can represent a sufficiency relation (Table 6.2).

Table 6.2 General 2×2 table for sufficiency

Y	1	a	c
	0	b	d
		0	1
		X	

The focus, here, is on the cells c and d, with the sufficiency inclusion score calculated with the formula:

$$inclS_{X \Rightarrow Y} = \frac{X \cap Y}{X} = \frac{c}{c+d} \quad (6.1)$$

The inclusion is the proportion of the cases where both X and Y happen (the intersection between X and Y, cell c), out of all cases where X happen (cells c and d together). In a most trivial command, this intersection can be calculated as:

```
sum(X & Y)/sum(X)
```

Using the same crisp version of the Lipset data, the inclusion of the condition "DEV" in the outcome SURV is:

```
with(LC, sum(DEV & SURV) / sum(DEV))
```

```
[1] 0.8
```

There are eight cases where both DEV and SURV are present, out of a total of ten cases where DEV is present. Although not the highest possible, a sufficiency inclusion score of 0.8 is high enough to conclude the level of development DEV is sufficient, or at least an important part of an expression that is sufficient for the outcome SURV, survival of democracy.

Chapter 7 will introduce the concept of an inclusion cut-off, to compare this score against.

The same result is obtained by using either version of the `pof()` command, specifying relation "sufficiency" but "suf" will be accepted as well:

```
pof("DEV", "SURV", data = LC, relation = "sufficiency")
```

or

```
pof("DEV => SURV", data = LC)
```

	inclS	PRI	covS	covU
1 DEV	0.800	0.800	1.000	-

In the second version, the sufficiency relation is signalled with the help of the right pointing arrow => sign, making the specification of the argument relation = "sufficiency" redundant.

Calculations for multi-value data are just as simple, and extend a 2 × 2 table to one with multiple columns:

Table 6.3 General crosstable for multivalued sufficiency

		0	1	2
Y	1	a	c	e
	0	b	d	f
		X		

Here, it is actually more clear that each value of X has its own inclusion in the outcome Y, as if each value was a separate set from all other values in X. This is actually very close to what is happening, each value behaving indeed as a separate set, with an additional constraint that “sets” corresponding to each value are mutually exclusive.

$$inclS_{X\{v\} \Rightarrow Y} = \frac{X\{v\} \cap Y}{X} \tag{6.2}$$

Replacing *v* with the value 2, as highlighted in the crosstable (Table 6.3), the focus is on the last column where X takes the value 2, specifically on cell e (where the intersection occurs) and cell f. The equation becomes:

$$inclS_{X\{2\} \Rightarrow Y} = \frac{e}{e + f}$$

As a simple practical example, to mimic this command in R involves first loading the multi-value version of the Lipset data (if not already loaded), and the actual inclusion score is given by the second line of code:

```
data(LM)
with(LM, sum(DEV == 2 & SURV) / sum(DEV == 2))
```

```
[1] 1
```

There is a perfect consistency score of 1 between the value 2 of the development condition, that is sufficient for the survival of democracy, which makes it very likely to find this expression in the final minimization solution. This consistency can further be inspected by subsetting the dataset with:

```
LM[LM$DEV == 2, c(1, 6)]
```

	DEV	SURV
BE	2	1
FR	2	1
NL	2	1
SE	2	1
UK	2	1

The consistency is perfect, with all countries having a surviving democracy where the level of development is equal to 2. The same conclusion is drawn using the parameters of fit function:

```
pof("DEV{2} => SURV", data = LM)
```

		inclS	PRI	covS	covU
1	DEV{2}	1.000	1.000	0.625	-

To compare with a different value, the situation is not similar in the cases where the level of development is equal to 1:

```
LM[LM$DEV == 1, c(1, 6)]
```

	DEV	SURV
AU	1	0
CZ	1	1
FI	1	1
DE	1	0
IE	1	1

Out of the five countries where that happens, democracy survives only in three, which means that DEV{1} is not sufficient for the outcome because its inclusion score of 0.6 is too low:

```
pof("DEV{1} => SURV", data = LM)
```

	inclS	PRI	covS	covU
1 DEV{1}	0.600	0.600	0.375	-

The equation for fuzzy sets is only a tiny bit more challenging to calculate. Much like in the case of necessity, there are no crosstables to calculate simple cell frequencies but the formula for calculating fuzzy inclusion remains the same, a straightforward matter of summing their fuzzy intersection (the minimum of each pair of values between X and Y), and divide over the sum of all values in X. This gives a score that reflects how much of X is included in Y (or how consistent X is with the outcome Y).

$$inclS_{X \Rightarrow Y} = \frac{\sum \min(X, Y)}{\sum X} \tag{6.3}$$

Using the fuzzy version of the Lipset data:

```
# load the data if not previously loaded
data(LF)
# then
with(LF, sum(fuzzyand(DEV, SURV)) / sum(DEV))
```

[1] 0.7746171

The same inclusion score (rounded to three decimals) is obtained using the parameters of fit function:

```
pof("DEV => SURV", data = LF)
```

	inclS	PRI	covS	covU
1 DEV	0.775	0.743	0.831	-

Much like the necessity relation, simultaneous subset relations appear for sufficiency relations as well. The inclusion in the negation of the outcome set is not simply the complement of the inclusion in the set itself. With an inclusion of 0.775 for the level of development into survival of democracy outcome, one would expect exactly 0.225 to be outside (included in the negation of) the outcome set, but the actual value is 0.334:

```
pof("DEV => ~SURV", data = LF)
```

	inclS	PRI	covS	covU
1 DEV	0.334	0.241	0.322	-

6.3 The PRI Score

To further analyze the issue of simultaneous subset relations, Table 5.4 will be reexamined with the subset relations reversed, the values in set X being switched with the values in set Y:

```
X <- c(0.2, 0.4, 0.45, 0.5, 0.6)
Y <- c(0.3, 0.5, 0.55, 0.6, 0.7)
```

Since all the elements of X are now lower than the corresponding elements in Y, we expect there is a full consistency of 1 (full inclusion) for the sufficiency of X for Y:

```
pof(X, Y, relation = "sufficiency") # "suf" is also accepted
```

	inclS	PRI	covS	covU
1 X	1.000	1.000	0.811	-

As expected, the inclusion score reaches a maximum, and we should expect the inclusion in the negation of Y to be very low or even equal to zero. As it turns out, the inclusion of X into the negation of Y is also very high:

```
pof(X, 1 - Y, relation = "sufficiency")
```

	inclS	PRI	covS	covU
1 X	0.814	0.000	0.745	-

This is a typical situation of simultaneous subset relations, with X appearing to be sufficient for both Y and $\sim Y$, which is a logical contradiction. A decision has to be made, and declare X as sufficient for either Y or $\sim Y$, as logically it cannot be sufficient for both.

This is where the PRI measure comes into play, which is found in the output for the parameters of fit function, in the sufficiency relation. PRI stands for Proportional Reduction in Inconsistency, and was introduced in the fs/QCA software by Ragin (2006) and further explained by Schneider and Wagemann (2012), and has the following formula that is faithfully implemented in the *QCA* package:

$$\text{PRI} = \frac{\sum \min(X, Y) - \sum \min(X, Y, \sim Y)}{\sum X - \sum \min(X, Y, \sim Y)} \quad (6.4)$$

Equation (6.4) is an extension of Eq. (6.3). The first part is exactly the same, both numerator and denominator subtracting the intersection between X , Y and $\sim Y$, to take into account the negation of the outcome as well as its presence, in the same formula.

When simultaneous subset relations occur, a decision has to be made, and declare X as sufficient for either Y or $\sim Y$. This decision is based on the PRI score, whichever has the highest product between the consistency score and the PRI.

In our example, at the sufficiency relation for the presence of the output, the PRI score is equal to the maximum 1. For the negation of the outcome, the PRI score drops to 0, therefore it can safely be concluded that X is sufficient for Y , and reject that X could be sufficient for $\sim Y$.

Schneider and Wagemann (2012) explain that simultaneous subset relations appear when there is at least one logically contradictory case, for at least one of the relations with Y and/or $\sim Y$. In the example above, X is a perfect subset of Y , hence logically contradictory cases can only appear in the sufficiency relation with the negation of the outcome (Fig. 6.5):

```
XYplot(X, 1 - Y, relation = "sufficiency")
```

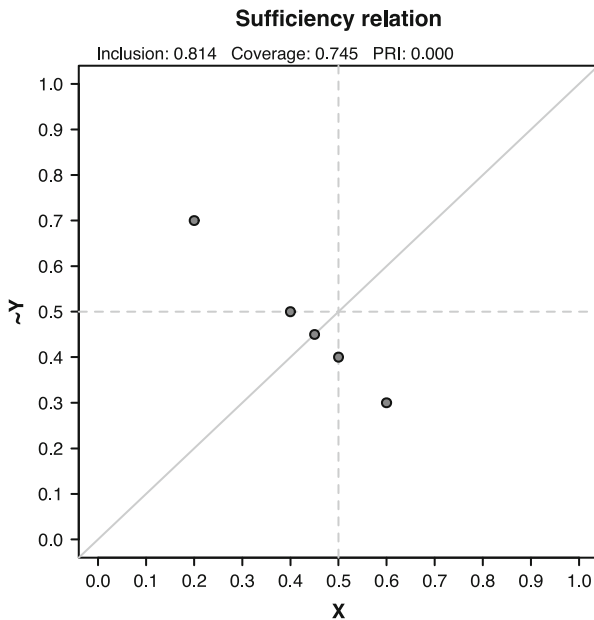


Fig. 6.5 Sufficiency relation between X and the negation of Y

Indeed, there is one case in the lower right part of the XY plot, the region where deviant cases in kind are found. Such careful attention to sufficiency scores (for both presence of the outcome and its negation), combined with an analysis of XY plots to identify deviant cases, possibly with a more in-depth analysis of the deviant cases themselves, define an important preliminary step before deciding that a certain condition or combination of conditions could enter in the minimization process, as it will be shown in chapter 8.

6.4 Coverage: Raw and Unique

Unlike necessity, where coverage is a measure of how trivial a condition is for an outcome, in the sufficiency relation the coverage is used as a measure to calculate how much of the entire outcome Y is “explained” by a causal condition X.

Readers who are familiar with standard statistical measures might also be familiar with a regression based measure called R^2 , which reports how much of the variation in the dependent variable is explained by a certain regression model.

Obviously, QCA has nothing to do with statistics and variation, but the overall interpretation is very similar. For sufficiency relations, set X is more important for Y, the more it covers. To the limit, when a set X covers exactly 100% from Y, it becomes not only sufficient but also necessary for the outcome.

An extension of X beyond 100% of Y would still render it as necessary, but it will not sufficient anymore given that X needs to be a subset of Y for a valid sufficiency relation.

A high coverage does not necessarily means that X needs to be large. When Y is a small set, even a small X can cover a lot of Y. If Y is a large set, then X needs to be large in order to cover a significant area of Y. More important is the relative difference between X and Y: if the relative difference is small, then X covers a lot, and if the relative difference is high, it means that X covers very little from Y.

This kind of coverage is called a *raw* coverage (denoted with `covS` in the *QCA* package, from coverage for sufficiency), to differentiate from the so-called *unique* coverage (denoted with `covU`), specific to each set that covers Y. Again, this is also very similar to the regression model, where two independent variables can have their own R^2 explaining the dependent variable Y, but if the independent variables are collinear then much of their independent explanation overlap and the overall R^2 for the entire regression model is not equal to the sum of the individual explanations.

It is more or less similar in the sufficiency relation, raw coverage showing how much of the outcome Y is explained by a set, and unique coverage showing how much of that explanation can be uniquely attributed to that set, and to no other.

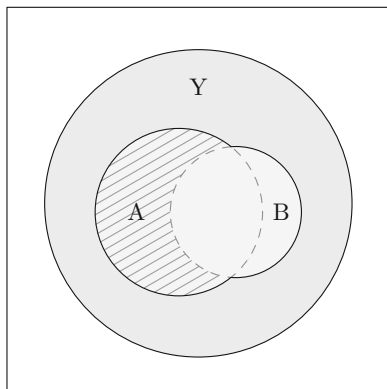


Fig. 6.6 Unique coverage for X

Figure 6.6 displays two sufficient expressions A and B , together covering a certain part of Y . That is the raw coverage of the union between the sets A and B (the disjunctive expression $A + B$). Out of that total coverage, a good part is covered by both sets, and whatever is covered uniquely by A is shown in the striped area.

That is called the *unique* coverage of A , and it can be calculated by subtracting the coverage of the intersection $A \cdot B$, from the entire (raw) coverage of A :

$$covU_{A \Rightarrow Y} = covS_{A \Rightarrow Y} - covS_{A \cdot B \Rightarrow Y}$$

As coverage for sufficiency has the same formula as the inclusion for necessity, this is translated as:

$$covU_{A \Rightarrow Y} = \frac{\sum \min(Y, A)}{\sum Y} - \frac{\sum \min(Y, A, B)}{\sum Y}$$

Here, the set B can be interpreted by a combination of individual sets, for example if we had any number of sets A, B, C and so on, the general formula for the unique coverage of A is the raw coverage of A minus the coverage of the intersection between Y, A , and the union of all other sets B, C and so on:

$$covU_{A \Rightarrow Y} = \frac{\sum \min(Y, A)}{\sum Y} - \frac{\sum \min(Y, A, \max(B, C, \dots))}{\sum Y} \tag{6.5}$$

It is perhaps less obvious that raw and unique coverage apply not only for individual (atomic) sets but also for any type of sufficient expression. Any kind of SOP—sum of products expression can be interpreted as an individual set, which has a certain coverage within the outcome set Y .

To put it differently, a “set” is not always atomic (formed by the elements of a single condition). A conjunction of conditions is also a set, or a disjunction and also a more complex expression of multiple conjunctions and disjunctions is also a “set”. No matter how complex the expression is, the resulting set has a coverage within the outcome set, if it is sufficient.

Chapter 7

The Truth Table



7.1 General Considerations

The truth table is the main analytical tool needed to perform the minimization process that was invented in the engineering, and which Charles Ragin adapted for the social sciences. It is well known the electric engineering procedure was pioneered by Shannon (1940) at MIT, after having been earlier exposed to the work of George Boole, during his studies at the University of Michigan. However it is less known that something extremely similar was developed even earlier by the American sociologist Lazarsfeld (1937) who introduced the concept of “attribute space”, later to become what is known today as the concept of “property space”.

Lazarsfeld advanced a social research method based on typologies, arranging the observed empirical information on various combinations of properties or attributes. On page 10 from his 1937 paper, he presents a table containing eight combinations of three binary attributes to study discrimination: “To have (+) or not to have (-) a college degree, to be of white (+) or colored (-) race, and to be native (+) or foreign born (-) in America”:

Table 7.1 Lazarsfeld’s attribute space

Combination number	College degree	White	Native born
1	+	+	+
2	+	+	-
3	+	-	+
4	+	-	-
5	-	+	+
6	-	+	-
7	-	-	+
8	-	-	-

This is almost exactly the description of a modern truth table, with reversed rows (Table 7.1). What Lazarsfeld advanced, and what is probably less obvious to the untrained eye, is that a truth table is a generalization of a regular crosstable for multiple attributes (causal conditions in QCA).

For two conditions, keeping things simple and restrict each to only two values, a crosstable and its truth table counterpart is easy to construct. Figure 7.1 shows such an example, with the familiar 2×2 crosstable on the left side and the truth table on the right side, plus the corresponding cells.

		X		
		0	1	
	1	a	c	
	0	b	d	
Y				

X	Y	
0	0	b
0	1	a
1	0	d
1	1	c

Fig. 7.1 Crosstable and truth table for two conditions

The only difference is the arrangement of the individual combinations of values, in the crosstable along the familiar cross pattern, while in the truth table the combinations are arranged row-wise, below the others.

A three-way crosstable, although not very difficult to imagine, is less obvious and clearly more complicated. On the other hand, as seen in Fig. 7.2, the truth table looks just as simple as the one from the previous example.

			X				
			0		1		
			0	1	0	1	
	1	a	c	e	g		
	0	b	d	f	h		
Z							

X	Y	Z	
0	0	0	b
0	0	1	a
0	1	0	d
0	1	1	c
1	0	0	f
1	0	1	e
1	1	0	h
1	1	1	g

Fig. 7.2 Crosstable and truth table for three conditions

Crosstables can get even more complicated for four conditions, even more for five until they reach a practical limit. They are useful when kept simple, otherwise losing their practical applicability when they get very complicated. Truth tables, on the other hand, look similar for any number of conditions, just doubling their number of rows with each new condition.

The structure of a truth table is therefore a matrix with k columns, where k is the number of causal conditions included in the analysis. The number of rows is often presented as 2^k , in the example above for three causal conditions there are $2^3 = 8$ rows. This is partially correct, because it applies to binary crisp sets only.

The universal equation that applies for both binary and multi-value crisp sets, is the product of the number of levels l (valents, categories) for each causal condition from 1 to k :

$$\prod_{i=1}^k l_i = l_1 \times l_2 \times \dots \times l_k \quad (7.1)$$

Incidentally, $2^3 = 2 \times 2 \times 2 = 8$, but if one of the causal conditions had three values (levels) instead of two, the structure of the truth table changes to:

```
createMatrix(noflevels = c(3, 2, 2))
```

```

      [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    1
[3,]    0    1    0
[4,]    0    1    1
[5,]    1    0    0
[6,]    1    0    1
[7,]    1    1    0
[8,]    1    1    1
[9,]    2    0    0
[10,]   2    0    1
[11,]   2    1    0
[12,]   2    1    1

```

In the above command, the argument `noflevels` refers to the number of levels for three causal conditions (and the output shows three columns as well), however the number of rows is not 8, but $3 \times 2 \times 2 = 12$. Equation (7.1), as well as the above function `createMatrix()`, give all possible combinations of properties for any given number of causal conditions k , with any number of levels for each, describing the entire property space mentioned by Lazarsfeld.

The next step in the truth table procedure is to allocate individual cases to the corresponding truth table rows. This is something similar to constructing a table of frequencies, as the calibrated values in the input data have the same structure as the combinations from the truth table.

A good example here is the binary crisp version of the Lipset data, examining the first six rows:

```
data(LC) # if not already loaded
head(LC)
```


	DEV	URB	LIT	IND	STB	SURV
AU	1	0	1	1	0	0
BE	1	1	1	1	1	1
CZ	1	1	1	1	1	1
EE	0	0	1	0	1	0
FI	1	0	1	0	1	1
FR	1	0	1	1	1	1

Very often, multiple cases display the same configuration, and in the Lipset data BE and the CZ have positive values for all conditions from DEV to STB, and even the same value for the outcome SURV. These two cases are allocated to the same truth table combination 11111, which has at minimum two cases:

```
truthTable(LC, outcome = "SURV", complete = TRUE, show.cases = TRUE)
```

OUT: output value
 n: number of cases in configuration
 incl: sufficiency inclusion score
 PRI: proportional reduction in inconsistency

	DEV	URB	LIT	IND	STB	OUT	n	incl	PRI	cases
1	0	0	0	0	0	0	3	0.000	0.000	GR,PT,ES
2	0	0	0	0	1	0	2	0.000	0.000	IT,RO
3	0	0	0	1	0	?	0	-	-	
4	0	0	0	1	1	?	0	-	-	
5	0	0	1	0	0	0	2	0.000	0.000	HU,PL
6	0	0	1	0	1	0	1	0.000	0.000	EE
7	0	0	1	1	0	?	0	-	-	
8	0	0	1	1	1	?	0	-	-	
9	0	1	0	0	0	?	0	-	-	
10	0	1	0	0	1	?	0	-	-	
11	0	1	0	1	0	?	0	-	-	
12	0	1	0	1	1	?	0	-	-	
13	0	1	1	0	0	?	0	-	-	
14	0	1	1	0	1	?	0	-	-	
15	0	1	1	1	0	?	0	-	-	
16	0	1	1	1	1	?	0	-	-	
17	1	0	0	0	0	?	0	-	-	
18	1	0	0	0	1	?	0	-	-	
19	1	0	0	1	0	?	0	-	-	
20	1	0	0	1	1	?	0	-	-	
21	1	0	1	0	0	?	0	-	-	
22	1	0	1	0	1	1	2	1.000	1.000	FI,IE
23	1	0	1	1	0	0	1	0.000	0.000	AU
24	1	0	1	1	1	1	2	1.000	1.000	FR,SE
25	1	1	0	0	0	?	0	-	-	
26	1	1	0	0	1	?	0	-	-	
27	1	1	0	1	0	?	0	-	-	
28	1	1	0	1	1	?	0	-	-	
29	1	1	1	0	0	?	0	-	-	
30	1	1	1	0	1	?	0	-	-	
31	1	1	1	1	0	0	1	0.000	0.000	DE
32	1	1	1	1	1	1	4	1.000	1.000	BE,CZ,NL,UK

There are actually four cases with that particular combination (last row in the truth table), namely BE, CZ, NL and UK. The same number of four cases is displayed in the truth table for that configuration, under the column `n`.

The other two columns `incl` and `PRI` should be self-explanatory. Inclusion, for crisp sets, show how consistent are the cases from a given causal configuration, to display the same value in the outcome `SURV`. In this truth table, all observed configurations are perfectly consistent, and the column `OUT` is allocated a value of 1 where the inclusion is 1, and 0 otherwise.

In the past, when fuzzy sets were not introduced yet, a single case with a different value for the outcome would render the causal combination as a contradiction. The modern approach is to allocate a value of 1 in the `OUT` column if the inclusion score is at least equal to a certain inclusion cut-off, something which will become more evident when dealing with fuzzy sets. The same thing applies for the `PRI` score, which is always equal to the inclusion score for crisp sets, and will play a greater role for fuzzy sets.

7.2 Command Line and GUI Dialog

At this point, some more explanations are needed about the written command and its arguments, as well as the output. A first thing to notice is the missing argument `conditions`, a situation where all columns in the dataset except the specified outcome are automatically considered causal conditions. The same output can be produced with the following, equivalent command:

```
truthTable(LC, outcome = "SURV", conditions = "DEV, URB, LIT, IND, STB",
           complete = TRUE, show.cases = TRUE)
```

The complete structure of the `truthTable()` function contains some other arguments, like: `incl.cut`, `n.cut`, `sort.by`, `use.letters` and `inf.test`. They will be described during this chapter, but for the time being it is important to remember that not specifying an argument doesn't mean they have no influence over the output. Quite the contrary, they have default values that are automatically employed, if not otherwise specified.

Before going through the rest of the arguments, it is also important to remember that package *QCA* has a graphical user interface for the most important functions, including the creation of a truth table. Having the web user interface started (see Sect. 2.4.2), the dialog is opened by choosing the menu:

Analyse/Truth Table

Figure 7.3 is an accurate match of the command used to produce the earlier truth table:

Fig. 7.3 The “Truth table” dialog

- it has the LC, the binary crisp version of the Lipset data selected
- in the **Outcome** space the column SURV is selected
- no columns are selected in the **Conditions** space (all others are automatically considered causal conditions)
- the checkboxes **complete** and **show cases** are selected

In addition, the truth table is attributed to an object called `ttLC`, otherwise it will just be printed on the screen and then disappear from memory. Assigning truth tables to objects, in the graphical user interface, has an additional role to be presented later.

The checkbox **complete** (an argument) is not mandatory, here used for demonstrative purposes to create the entire truth table containing all possible configurations. It can be seen that most of them are empty, with zero empirical information in the column `n`, and the column `OUT` coded as a question mark. These configurations are the so called *remainders* in QCA, and they become counterfactuals when included in the minimization process.

In the middle part of the options space in the dialog, there are three possible ways to sort the truth table: by the output score in the column `OUT`, by the inclusion score, and by the number of cases (frequency) in column `n`. All of these belong to the argument `sort.by` of the written function.

By default, there is no sorting and the truth table is presented in the natural order of the configurations, from the absence of all (`00000` on line 1) to the presence of all (`11111` on line 32). Activating any of those (by clicking in the interface), opens up another possibility to sort: in decreasing (default) or increasing order.



Fig. 7.4 Different possibilities to sort the truth table

Figure 7.4 shows two different possibilities to sort the truth table, in the interface. The left side activates the sort by *inclusion* (default in decreasing order) and *frequency* (increasing order, having unchecked the corresponding checkbox under **Decr.**easing).

In the right side, should all of these options be activated, the figure demonstrates the action of dragging the frequency option on the top of the list, to establish the sorting priority.

Unchecking the **complete** checkbox results in this command which is visible in the command constructor dialog:

```
truthTable(LC, outcome = "SURV", show.cases = TRUE, sort.by = "incl, n+")
```

- OUT: output value
- n: number of cases in configuration
- incl: sufficiency inclusion score
- PRI: proportional reduction in inconsistency

	DEV	URB	LIT	IND	STB	OUT	n	incl	PRI	cases
22	1	0	1	0	1	1	2	1.000	1.000	FI,IE
24	1	0	1	1	1	1	2	1.000	1.000	FR,SE
32	1	1	1	1	1	1	4	1.000	1.000	BE,CZ,NL,UK
6	0	0	1	0	1	0	1	0.000	0.000	EE
23	1	0	1	1	0	0	1	0.000	0.000	AU
31	1	1	1	1	0	0	1	0.000	0.000	DE
2	0	0	0	0	1	0	2	0.000	0.000	IT,RO
5	0	0	1	0	0	0	2	0.000	0.000	HU,PL
1	0	0	0	0	0	0	3	0.000	0.000	GR,PT,ES

The truth table is now sorted according to the above criteria, and the row numbers reflect this change: first row is number 22, which is also the first where the inclusion score has the value or 1, and also the first there the column **OUT** was allocated a value of 1 for a positive output.

The graphical user interface presents only three sorting options, but the written command is more flexible and allows for any truth table column (including the causal conditions) to be used for sorting. The `sort.by` argument accepts a string with the columns of interest separated by a comma, and signal an increasing order sorting by adding a “+” sign after a specific column.

In the past, there was an additional logical argument called **decreasing**, and applied to all sorting criteria. That argument has become obsolete with the new structure of the `sort.by` argument, in the example above "`incl, n+`" indicating to first sort by inclusion (by default in decreasing order) then after the frequency in increasing order.

7.3 From Fuzzy Sets to Crisp Truth Tables

To better explain the cut-off values for inclusion and frequency, it is the right time to talk about constructing a truth table from a fuzzy dataset. Crisp datasets are straightforward, cases being allocated to their corresponding truth table rows. With fuzzy sets this is not that easy because fuzzy scores are not simply equal to 0 or 1, but anywhere in between, which means that a case has partial memberships in all truth table configurations.

Ragin (2000) defines the fuzzy sets as a multidimensional vector space with a number of corners equal to 2^k , where each corner of the vector space is a unique configuration of (crisp) causal conditions. The number of corners 2^k is accurate here, because fuzzy sets don't have multiple "levels" (or categories) like crisp sets but only two extreme limits represented by the numbers 0 (full exclusion) and 1 (full inclusion).

The next section will introduce some notions of Boolean minimization, that requires crisp scores. Since obviously fuzzy scores cannot be used for this algorithm, the fuzzy scores need to be transformed into crisp scores to allocate cases into one or another configuration from the truth table.

Basically, there is no "pure" fuzzy minimization procedure (given an infinite number of possible combinations of fuzzy scores, for multiple dimensions), therefore fuzzy sets need to be first transformed into crisp truth table before the minimization.

This process requires a further detailed explanation of the relation between the different contingency tables (crosstabulations) and the process of constructing typologies. The simplest example is a 2×2 contingency table with four corners. As fuzzy sets are continuous, the squared vector space is also continuous and a case has fuzzy coordinates anywhere in this space.

The vector space is continuous, but the corners are crisp and they define all $2^2 = 4$ possible configurations of presence/absence for two causal conditions. The fuzzy coordinates determine how close a certain point (a case) is to one of the four corners. In Fig. 7.5 things are very clear, the point being closest to the corner (01), but when coordinates are closer to the center of the vector space it becomes more difficult to decide where to allocate the case.

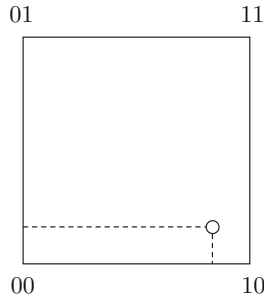


Fig. 7.5 Bidimensional vector space

Each causal condition acts like a geometrical dimension, defining the shape of the vector space. For one causal condition it is a line with two ends, for two causal conditions the space is a cartesian square and four corners, for three causal conditions the vector space is three dimensional having eight corners etc. All dimensions are assumed to be orthogonal to each other, measuring separate and distinct properties (Fig. 7.6).

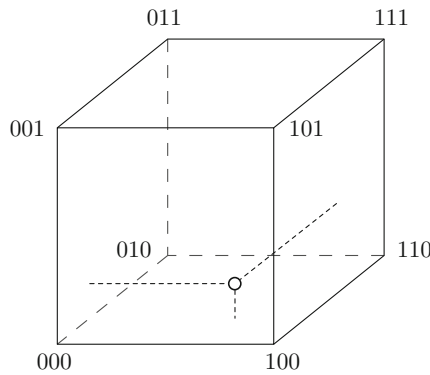


Fig. 7.6 Three dimensional vector space

Fuzzy sets almost never reach a corner with perfect scores, instead the coordinates from the vector space tend to get closer to one corner or another. The perfect scores from the corner are crisp, and they are similar to the Weberian concept of “ideal-type”: as no scientific theory would ever account for the full diversity of all possible particular manifestations of some phenomena in the real world, it needs to be abstracted to some kind of a tool that is not a perfect match for every possible real situation, but a close enough match to decide that observed manifestations are similar to certain theoretical models.

The ideal-type is such an abstract model that cannot be measured directly, but its manifestations in the real world can be observed through various behavioral indicators. The complex social reality is nothing but the manifestation of an unobserved ideal-type, an important feature that allows categorizing the

infinite combinations of manifestations to a finite collection of ideal-types, corners of the multidimensional vector space (Lazarsfeld’s property space). This way, a very complex reality can be located near a single such corner from the vector space.

As logical as it may seem, this operation is not trivial because cases are similar to multiple corners, especially when they are positioned close to the center of the vector space. It is again Ragin (2005, 2008b) who developed the transformation technique from fuzzy sets to the crisp corners from the truth table, based on the fundamental property that:

...each case can have only a single membership score greater than 0.5 in the logically possible combinations formed from a given set of causal conditions.

Ragin demonstrates this process using this simple dataset, which can be found in the package *QCA*, along with a dedicated help file to describe each column. The outcome is “W” weak class voting, and the causal conditions are “A” affluent countries, “I” substantial levels of income inequality, “M” high percentage of workers employed in manufacturing, and “U” countries with strong unions.

```
data(NF)
NF
```

	A	I	M	U	W
AU	0.9	0.7	0.3	0.7	0.7
BE	0.7	0.1	0.1	0.9	0.7
DK	0.7	0.3	0.1	0.9	0.1
FR	0.7	0.9	0.1	0.1	0.9
DE	0.7	0.9	0.3	0.3	0.6
IE	0.1	0.7	0.9	0.7	0.9
IT	0.3	0.9	0.1	0.7	0.6
NL	0.7	0.3	0.1	0.3	0.9
NO	0.7	0.3	0.7	0.9	0.1
SE	0.9	0.3	0.9	1.0	0.0
UK	0.7	0.7	0.9	0.7	0.3
US	1.0	0.9	0.3	0.1	1.0

Particular fuzzy scores make the coordinates of each case close to multiple corners, but only one corner is the closest. Cases have membership scores for each corner, and Ragin observed that only one corner has a membership score greater than 0.5, provided that none of the individual fuzzy scores are equal to exactly 0.5 (maximum ambiguity). It is important, especially in the calibration phase, to make sure that no score is equal to 0.5 otherwise the transformation to crisp truth tables becomes impossible.

The first step of the procedure is to create a matrix of all observed cases on the rows, and all possible combinations from the truth table on the columns. The next step is to calculate inclusion scores for each case on each truth table configuration, to decide which of the truth table rows has an inclusion score above 0.5. There are $2^4 = 16$ possible configurations in the truth table, with as many columns in the matrix.

For reasons of space, the columns are labeled by their corresponding row numbers from the truth table, but otherwise number 1 means **0000**, number 2 means **0001**, number 3 means **0010** etc.

Table 7.2 Ragin’s matrix

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
AU	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.3	0.3	0.3	0.3	0.3	0.7	0.3	0.3
BE	0.1	0.3	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.7	0.1	0.1	0.1	0.1	0.1	0.1
DK	0.1	0.3	0.1	0.1	0.1	0.3	0.1	0.1	0.1	0.7	0.1	0.1	0.1	0.3	0.1	0.1
FR	0.1	0.1	0.1	0.1	0.3	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.7	0.1	0.1	0.1
DE	0.1	0.1	0.1	0.1	0.3	0.3	0.3	0.3	0.1	0.1	0.1	0.1	0.7	0.3	0.3	0.3
IE	0.1	0.1	0.3	0.3	0.1	0.1	0.3	0.7	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
IT	0.1	0.1	0.1	0.1	0.3	0.7	0.1	0.1	0.1	0.1	0.1	0.1	0.3	0.3	0.1	0.1
NL	0.3	0.3	0.1	0.1	0.3	0.3	0.1	0.1	0.7	0.3	0.1	0.1	0.3	0.3	0.1	0.1
NO	0.1	0.3	0.1	0.3	0.1	0.3	0.1	0.3	0.1	0.3	0.1	0.7	0.1	0.3	0.1	0.3
SE	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.7	0.0	0.1	0.0	0.3
UK	0.1	0.1	0.3	0.3	0.1	0.1	0.3	0.3	0.1	0.1	0.3	0.3	0.1	0.1	0.3	0.7
US	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.1	0.7	0.1	0.3	0.1

Calculating membership scores in truth table configurations is rather straightforward. As an example, the first configuration in the truth table (first column in Table 7.2) is **0000**, or in product notation it is $\sim A \cdot \sim I \cdot \sim M \cdot \sim U$.

The fuzzy scores for the first country (Austria) are:

```
NF[1, 1:4] # the same can be produced with NF["AU", 1:4]
```

```
      A   I   M   U
AU 0.9 0.7 0.3 0.7
```

Calculating its inclusion score in the first truth table configuration (which should be equal to 0.1) involves a simple negation of each score then taking the minima, using the familiar `fuzzyand()` intersection function, or perhaps even more simple with:

```
compute("~A~I~M~U", data = NF[1, ])
```

```
[1] 0.1
```

The only truth table configuration where Austria has a membership score above 0.5 is number 14, which is **1101**. When calculating the fuzzy intersection for this configuration, only the third causal condition M is negated:

```
compute("AI~MU", data = NF[1, ])
```

```
[1] 0.7
```


Ragin's procedure, as simple and intuitive as it may seem, involves calculating all possible membership scores for all cases into all truth table configurations. When the number of causal conditions is small, this is really easy for any decent computer, but the more causal conditions are added into the model, the more difficult it becomes. That happens because the number of rows in the truth table increases exponentially to the powers of 2, quickly reaching a physical limit even for strong computers.

The required memory grows more than twice for each new condition (the truth table grows not only on the rows, but also on the columns), and the calculation time becomes prohibitively slow and very soon grinds to a halt. But looking at Table 7.2, it is clear that not all columns have membership scores above 0.5. Especially for large(r) truth tables, most configurations lack any empirical information due to the issue of limited diversity.

For as little as 10 causal conditions, a truth table has $2^{10} = 1024$ rows, and more than 1000 of them would be empty (no empirical information from the observed cases). For 20 causal conditions, more than one million rows are empty and they require not only a lot of memory but also pointless calculations that consume a lot of time.

Ragin's procedure can be greatly improved by further taking note that a case can have only one membership score above 0.5. To find the configurations of interest for each case, a simple dichotomization does the job:

```
ttrows <- apply(NF[, 1:4], 2, function(x) as.numeric(x > 0.5))
rownames(ttrows) <- rownames(NF)
ttrows
```

```
  A I M U
AU 1 1 0 1
BE 1 0 0 1
DK 1 0 0 1
FR 1 1 0 0
DE 1 1 0 0
IE 0 1 1 1
IT 0 1 0 1
NL 1 0 0 0
NO 1 0 1 1
SE 1 0 1 1
UK 1 1 1 1
US 1 1 0 0
```

These are exactly the truth table configurations where the cases have membership scores above 0.5, the code below transforming their binary representation into decimal notation using the matrix multiplication operator `%%`:

```
drop(ttrows %% c(8, 4, 2, 1)) + 1
```

```
AU BE DK FR DE IE IT NL NO SE UK US
14 10 10 13 13 8 6 9 12 12 16 13
```

Ragin (2008b, p. 138) warns against a mechanical dichotomization of fuzzy sets, and with good reasons. While dichotomization should not be used to convert fuzzy data to crisp sets, it can however be used to quickly identify the truth table configurations where the cases belong to, with a dramatic improvement on performance.

By not verifying all possible configurations from the truth table, Ragin's procedure becomes much faster and allows the possibility to generate truth tables with virtually any number of causal conditions, including the parameters of fit (something impossible in his initial, original version).

7.4 Calculating Consistency Scores

Each configuration in the truth table has a consistency score with the outcome (an inclusion in the outcome set). Table 7.2 present 16 columns for all configurations, and it is easy to be seen that each such column (configuration) has as many values as the number of cases in the dataset, and at the same time as many as the number of values in the outcome.

The inclusion score of a certain configuration in the outcome is calculated using the usual formula from Eq. (6.3), where X is replaced by the inclusion scores of the cases in that particular configuration (the columns from Table 7.2). The truth table for the NF data is presented below:

```
ttNF <- truthTable(NF, outcome = "W", incl.cut = 0.8, show.cases = TRUE)
ttNF
```

```
OUT: output value
n: number of cases in configuration
incl: sufficiency inclusion score
PRI: proportional reduction in inconsistency
```

	A	I	M	U	OUT	n	incl	PRI	cases
6	0	1	0	1	0	1	0.760	0.400	IT
8	0	1	1	1	1	1	0.870	0.667	IE
9	1	0	0	0	1	1	1.000	1.000	NL
10	1	0	0	1	0	2	0.700	0.438	BE,DK
12	1	0	1	1	0	2	0.536	0.071	NO,SE
13	1	1	0	0	1	3	0.971	0.944	FR,DE,US
14	1	1	0	1	1	1	0.821	0.583	AU
16	1	1	1	1	0	1	0.654	0.100	UK

What is printed on the screen is only a fraction of the total information contained in the object produced by the function `truthTable()`. This is essentially a list object with several other components, among which the useful `minmat`.

ttNF\$minmat

	6	8	9	10	12	13	14	16
AU	0.1	0.1	0.3	0.3	0.3	0.3	0.7	0.3
BE	0.1	0.1	0.1	0.7	0.1	0.1	0.1	0.1
DK	0.3	0.1	0.1	0.7	0.1	0.1	0.3	0.1
FR	0.1	0.1	0.1	0.1	0.1	0.7	0.1	0.1
DE	0.3	0.3	0.1	0.1	0.1	0.7	0.3	0.3
IE	0.1	0.7	0.1	0.1	0.1	0.1	0.1	0.1
IT	0.7	0.1	0.1	0.1	0.1	0.3	0.3	0.1
NL	0.3	0.1	0.7	0.3	0.1	0.3	0.3	0.1
NO	0.3	0.3	0.1	0.3	0.7	0.1	0.3	0.3
SE	0.1	0.1	0.0	0.1	0.7	0.0	0.1	0.3
UK	0.1	0.3	0.1	0.1	0.3	0.1	0.1	0.7
US	0.0	0.0	0.1	0.1	0.1	0.7	0.1	0.1

This is precisely the relevant part from Table 7.2, where the column numbers correspond to the row numbers from the empirically observed truth table configurations above. As shown in the previous section, the matrix represents the inclusion of all cases in each configuration, and the scores are used to calculate the overall consistency of all these configurations.

To demonstrate, consider the first column number 6, that is **0101** first row in the truth table. In product notation it is written as “ $\sim A \cdot I \cdot \sim M \cdot U$ ”, but since the conditions are simple letters it can also be written more simple as “aImU”.

In Table 7.3 below, the sum of the scores for the intersection with the outcome W is 1.9, while the sum of the scores in column “aImU” is 2.5, making the ratio between them equal to 0.760 which is the inclusion score from the first row in the truth table object **ttNF** above.

Table 7.3 Calculating the inclusion of configuration aImU in outcome W

	aImU	W	<i>min</i> (aImU, W)
AU	0.1	0.7	0.1
BE	0.1	0.7	0.1
DK	0.3	0.1	0.1
FR	0.1	0.9	0.1
DE	0.3	0.6	0.3
IE	0.1	0.9	0.1
IT	0.7	0.6	0.6
NL	0.3	0.9	0.3
NO	0.3	0.1	0.1
SE	0.1	0.0	0.0
UK	0.1	0.3	0.1
US	0.0	1.0	0.0
	2.5		1.9

The same value can be obtained with the familiar parameters of `fit pof()` command:

```
pof("aImU => W", data = NF)

      inclS  PRI    covS  covU
-----
1 aImU 0.760 0.400 0.279  -
```

And even with the `fuzzyand()` function:

```
aImU <- ttNF$minmat[, 1] # [, "6"] would also work
with(NF, sum(fuzzyand(aImU, W)) / sum(aImU))
```

```
[1] 0.76
```

The `minmat` component is thus more than just informative. It can be used for various purposes, including to identify the so called *deviant cases consistency*: that have an inclusion in the configuration higher than 0.5, but a lower (than 0.5) inclusion in the outcome.

```
truthTable(NF, outcome = "W", incl.cut = 0.8, show.cases = TRUE,
            dcc = TRUE)
```

```
OUT: output value
n: number of cases in configuration
incl: sufficiency inclusion score
PRI: proportional reduction in inconsistency
DCC: deviant cases consistency
```

	A	I	M	U	OUT	n	incl	PRI	DCC
6	0	1	0	1	0	1	0.760	0.400	
8	0	1	1	1	1	1	0.870	0.667	
9	1	0	0	0	1	1	1.000	1.000	
10	1	0	0	1	0	2	0.700	0.438	DK
12	1	0	1	1	0	2	0.536	0.071	NO,SE
13	1	1	0	0	1	3	0.971	0.944	
14	1	1	0	1	1	1	0.821	0.583	
16	1	1	1	1	0	1	0.654	0.100	UK

The `dcc` argument is a new one, and complements the argument `show.cases`. It prints the cases that represent the true logical remainders, and that happens if and only if the argument `show.cases` is activated. If this happens, the printing method asks which cases to print, the normal or the deviant cases.

The graphical user interface has a correspondent checkbox called **deviant cases**, that is activated only when the checkbox **show cases** is checked.

7.5 The OUTput Value

In a situation where all cases from a given configuration are perfectly consistent with the outcome, the column `OUT` will be assigned the value from the data.

More difficult are the situations where the consistency is not perfect, having values below 1. Back in the `csQCA` days, a single case from the same configuration, having a different output was problematic, since the requirement was to have a perfect inclusion (no cases with the presence of X, outside Y).

When the evidence is small (one contradictory case out of, say 5), that single case can be meaningful, but often the evidence is large enough to justify a natural question: what is the value of a single contradictory case out of 100? Is that case enough to dismiss a close to perfect (but not full) consistency?

This kind of question, concomitantly with the appearance of fuzzy sets `QCA`, marked the beginning of what today is the useful inclusion cut-off. The actual consistency score is compared against a certain threshold (specified by the researcher), and the column `OUT` is attributed a value of 1 only if the consistency score is higher. Even if not perfectly consistent, an inclusion score of 0.91 is still high enough, in any case higher than a plausible cut-off value of 0.9. Cases with a consistency score below this value are attributed a value of 0 in the `OUT` column.

In the fuzzy version of the Lipset data, no single truth table configuration has a perfect inclusion score, therefore in order to have at least one or some configurations assigned with a positive output, the argument `incl.cut` needs to be lowered down:

```
data(LF) # if not already loaded
truthTable(LF, outcome = "SURV", incl.cut = 0.8)
```

```
OUT: output value
     n: number of cases in configuration
incl: sufficiency inclusion score
PRI:  proportional reduction in inconsistency
```

	DEV	URB	LIT	IND	STB	OUT	n	incl	PRI
1	0	0	0	0	0	0	3	0.216	0.000
2	0	0	0	0	1	0	2	0.278	0.000
5	0	0	1	0	0	0	2	0.521	0.113
6	0	0	1	0	1	0	1	0.529	0.228
22	1	0	1	0	1	1	2	0.804	0.719
23	1	0	1	1	0	0	1	0.378	0.040
24	1	0	1	1	1	0	2	0.709	0.634
31	1	1	1	1	0	0	1	0.445	0.050
32	1	1	1	1	1	1	4	0.904	0.886

It should be clear that the output value from the truth table is not the same thing as the outcome column from the original data, even for binary crisp data. Not only because the outcome might have fuzzy values and the output should be binary, but most importantly because the outcome is given while the truth table output is assigned, sometimes against evidence from the data.

So far, the inclusion cut-off argument `incl.cut` was successfully used to produce the required two values of the output, 0 and 1. But in the former crisp sets procedure there was another possibility for the output value, namely the contradiction. That happens when the evidence is not enough for a positive output, but neither it is for a negative one.

To allow for such situations, the argument `incl.cut` accepts two values (two thresholds), the first above which the output is assigned a value of 1, and the second below which the output is assigned a value of 0. Such a length 2 numerical vector has the form `c(ic1, ic0)` where `ic1` is the inclusion cut-off for a positive output, and `ic0` the inclusion cut-off for a negative output. If not otherwise specified, the value of `ic0` is set equal to the value of `ic1`.

As an example, let us have a look at the truth table for the negation of the outcome (note the tilde in front of the outcome's name `~SURV`). In the graphical user interface, the checkbox **negate outcome** does the same thing:

```
truthTable(LF, outcome = "~SURV", incl.cut = 0.8)
```

```
OUT: output value
n: number of cases in configuration
incl: sufficiency inclusion score
PRI: proportional reduction in inconsistency
```

	DEV	URB	LIT	IND	STB	OUT	n	incl	PRI
1	0	0	0	0	0	1	3	1.000	1.000
2	0	0	0	0	1	1	2	0.982	0.975
5	0	0	1	0	0	1	2	0.855	0.732
6	0	0	1	0	1	1	1	0.861	0.772
22	1	0	1	0	1	0	2	0.498	0.281
23	1	0	1	1	0	1	1	0.974	0.960
24	1	0	1	1	1	0	2	0.495	0.366
31	1	1	1	1	0	1	1	0.971	0.950
32	1	1	1	1	1	0	4	0.250	0.106

As expected, the configurations which previously were coded with a positive output 1, are now coded with a negative output 0, with one exception. The configuration number 24 (**10111**) has a value of 0, and previously (for the normal outcome) it was also coded with 0, due to its low consistency of 0.709 (below the inclusion cut-off of 0.8). For the negation of the outcome, its inclusion score of 0.495 is also very low.

Something happens with this configuration, since there is not enough evidence for neither a positive output, nor for a negative one.

It looks like a contradiction, but using a single inclusion cut-off does not solve the puzzle. This is the purpose for the second value in the `incl.cut` argument, with the following decision process:

- if the inclusion score is greater than `ic1`, the output value is coded to 1
- if the inclusion score is lower than `ic0`, the output value is coded to 0
- when the inclusion score is between `ic0` and `ic1`, (greater than `ic0` but lower than `ic1`) the output value is coded as a contradiction, with the letter C:

```
truthTable(LF, outcome = "SURV", incl.cut = c(0.8, 0.6))
```

OUT: output value

n: number of cases in configuration

incl: sufficiency inclusion score

PRI: proportional reduction in inconsistency

	DEV	URB	LIT	IND	STB	OUT	n	incl	PRI
1	0	0	0	0	0	0	3	0.216	0.000
2	0	0	0	0	1	0	2	0.278	0.000
5	0	0	1	0	0	0	2	0.521	0.113
6	0	0	1	0	1	0	1	0.529	0.228
22	1	0	1	0	1	1	2	0.804	0.719
23	1	0	1	1	0	0	1	0.378	0.040
24	1	0	1	1	1	C	2	0.709	0.634
31	1	1	1	1	0	0	1	0.445	0.050
32	1	1	1	1	1	1	4	0.904	0.886

All of these arguments seem to provide a complete experience to create a truth table. There is however one more situation that is not covered by any of those, stemming from the steady expansion of the QCA methodology from the small-N world to the large-N situations, with hundreds of cases in the dataset.

In the quantitative methodology, a certain value is not *exactly* smaller or greater than another, because of the inherent uncertainty generated by the random sampling. Each dataset is unique, drawn from a very large population, and some situations can be sample specific. The same value can be smaller in a different random sample. The usual strategy is to bring sufficient statistical evidence to declare one value *significantly* greater than another, entering the realm of inferential statistics.

To dismiss possible suspicions that inclusion scores (greater than a certain cut-off) are sample specific, the truth table function has another argument called `inf.test`, which assigns values in the `OUT` column based on an inferential test. It is a binomial test, and it only works with binary crisp data (no fuzzy, no multi-values).

The argument is specified as a vector of length 2 or a single string containing both, with the type of test (currently only "binom") and the critical significance level.

```
truthTable(LC, outcome = "SURV", incl.cut = 0.8, inf.test = "binom, 0.05")
```

```

OUT: output value
n: number of cases in configuration
incl: sufficiency inclusion score
PRI: proportional reduction in inconsistency
pval1: p-value for alternative hypothesis inclusion > 0.8

```

	DEV	URB	LIT	IND	STB	OUT	n	incl	PRI	pval1
1	0	0	0	0	0	0	3	0.000	0.000	1.000
2	0	0	0	0	1	0	2	0.000	0.000	1.000
5	0	0	1	0	0	0	2	0.000	0.000	1.000
6	0	0	1	0	1	0	1	0.000	0.000	1.000
22	1	0	1	0	1	0	2	1.000	1.000	0.640
23	1	0	1	1	0	0	1	0.000	0.000	1.000
24	1	0	1	1	1	0	2	1.000	1.000	0.640
31	1	1	1	1	0	0	1	0.000	0.000	1.000
32	1	1	1	1	1	0	4	1.000	1.000	0.410

It seems that all outcome values have been coded to zero. Suggestion: lower the inclusion score for the presence of the outcome, the relevant argument is "incl.cut" which now has a value of 0.8.

This command performs a binomial statistical test, using a 5% significance level. Only the configurations with a *significantly* greater than 0.8 inclusion score would be coded with a positive output 1.

The example is provided for demonstration purposes only. Due to the very small number of cases for each configuration (from 1 to at most 4), binomial testing is obviously not producing any significant results. The probability of error `pval1` is too high (100% in many cases), therefore the null hypothesis that the true (population) inclusion score is less than 0.8, cannot be rejected. This kind of statistical test requires large(*r*) samples to draw meaningful conclusions, but if such data would be available, specifying the `inf.test` should be done as in the example above.

A more important argument, albeit rarely used, is the frequency cut-off `n.cut`, the minimum number of cases under which a configuration is declared as a remainder (the output value is assigned a "?" sign), even if it has an consistency score above the inclusion cut-off. As QCA data belongs to the so called "small-N" world, `n.cut` has a default value of 1 but there are situations with very large data available, when it might make sense to raise the frequency cut-off to more than 1 case per configuration.

The final argument that needs introducing is a logical one called `use.letters`. When the names of the causal conditions are too long, they can be replaced by alphabetical letters. Switching this argument to `TRUE` will automatically replace each causal condition's name with an upper case letter, starting with `A` for the first column, `B` for the second etc (naturally, there should not be more than 26 causal conditions in the dataset).

7.6 Other Details

Theory makes a distinction between crisp and fuzzy sets, and yet another distinction between (binary) crisp and multi-value (crisp) sets. For some unknown reason, probably dating back when some *QCA* packages created separate truth tables for crisp, multi-value and fuzzy sets, many researchers remain under the impression that analyses themselves must be carried separately for each of these types.

In package *QCA*, there are no separate truth table functions for different types of sets. The same function `truthTable()` can be used for crisp, multi-value and fuzzy data. However, this information does not seem to be obvious enough since there are still situations when researchers believe the *whole* dataset (with all its conditions) should be calibrated binary, multi-value or fuzzy.

This misconception should be clarified, as the function `truthTable()` can easily accommodate a dataset containing all types of calibrations at once.

Another detail that is probably less known, that is especially useful for replication purposes, is that a truth table object contains all the information necessary to trace its creation. It contains a component named `call` that displays the exact command that was used to produce it, and also the original dataset that was used as an input for the function `truthTable()`, in the component `initial.data`.

Researchers can thus either exchange the initial data and the exact set of commands used to produce the truth table, or perhaps more simple they can exchange the truth table itself, since it already contains all relevant information:

```
ttLF <- truthTable(LF, outcome = "SURV", incl.cut = c(0.8, 0.6))
names(ttLF)
```

```
[1] "tt"           "indexes"      "noflevels"    "initial.data"
[5] "recoded.data" "cases"        "DCC"          "minmat"
[9] "options"      "fs"           "call"         "origin"
```

Due to many considerations, it is always a good idea to name all columns from the dataset with upper case letters. This advice is not only a good practise, but it can save users from many hours of trying to understand what does not work, should anything happen. Generally, the functions in package *QCA* are robust enough to deal with such situations, and this is one reason for which there is another component named `recoded.data` in the output list object.

Chapter 8

The Logical Minimization



The logical, or Boolean minimization process is the core of the QCA methodology, which seeks to find the simplest possible expression that is associated with the explained value of an output. The term *expression*, here, is a synonym for sums of products, or unions of intersections, or disjunctions of conjunctions (of causal conditions). It will also be used as a synonym for a causal configuration, since that is a conjunction (a product) of causal conditions.

McCluskey (1956) was the first to introduce this procedure, building on the previous work by Quine (1952, 1955), leading to an algorithm which is today known as the “Quine-McCluskey” (from here on, QMC). Their work was dedicated to electrical engineering switching circuits (opened and closed gates, Boolean state), and for each combination of such gates in the circuit design, an output is either present or absent.

But instead of a circuit designed for all possible combinations where an output occurs, McCluskey observed that it is much cheaper to specify the simplest possible expression, with a superior overall performance of the circuit. The idea is quite simple, and it is based on the following Boolean algebra theorem:

$$A \cdot B + A \cdot \sim B = A(B + \sim B) = A \tag{8.1}$$

Given any two expressions, if they differ by exactly one literal, that literal can be minimized. Table 8.1 presents a comparison of two expressions with three causal conditions each. Since the condition B is the only different one, it can be eliminated and the initial two expressions can be replaced by A·C.

Table 8.1 Boolean minimization

A	B	C	
1	1	1	A·B·C
1	0	1	A·~B·C
1	-	1	A·C

It is the simplest possible example of Boolean minimization, and the QMC algorithm is an extension of this operation for all possible pairs of cases. The first part of the original algorithm consists of successive iterations of these operations:

- compile a list of all possible pairs of two expressions
- check which pairs differ by only one literal, and minimize them
- the minimized pairs, plus the surviving unminimized expressions enter in the next iteration
- repeat operations until nothing can further be minimized

The final surviving, minimal expressions are called prime implicants (PI). The second part of the QMC algorithm creates the so called prime implicants chart, a matrix with the prime implicants on the rows, and the initial expressions on the columns. The PI chart from the original book by Ragin (1987, p. 97) is a very good example (Fig. 8.1).

	A·B·C	A·b·C	A·B·c	a·B·c
A·C	×	×		
A·B	×		×	
B·c			×	×

Fig. 8.1 Ragin's prime implicants chart

To solve the PI chart, the task is to find all possible combinations of rows (PIs) that cover all columns (initial expressions). In this example, there is a single combination of first and third PI that together, cover all initial expressions. The final solution is: $A \cdot C + B \cdot c$.

This is obviously a very simple PI chart, and the solution is straightforward. More complicated examples have more rows, and it is possible to have multiple solutions from multiple combinations of prime implicants that cover all columns. This is a quite technical jargon, and sometimes the columns are called *primitive expressions*, but it actually means that a solution is valid when the combination of prime implicants account for all observed empirical evidence.

Each prime implicant is a sufficient expression itself, but it doesn't always account for all primitive expressions. In such a situation, it needs (an)other sufficient prime implicant(s) that account for different other columns, and disjunctively (union with logical OR) create a larger sufficient expression accounting for all observed, empirical evidence.

The QMC minimization procedure is a two-level algorithm, the first phase identifying the prime implicants and the second solving the PI chart. Both are quite technical and could be described in a separate annex.

8.1 Command Line and GUI Dialog

Version 3.0 of package *QCA* brought important changes in the structure of the main `minimize()` function. Like all previous updates of the package, a lot of effort is being invested to make these changes backwards compatible, so that old code will still work in the new version.

The new structure is more simple, more clear and most of all more logical. The previous one was perpetuated from version to version, for historical reasons since the first version(s) of the package. In the beginning, the `minimize()` function had a dataset as an input, for which an `outcome` and some `conditions` were specified, and it followed the normal procedure to construct a truth table and perform the minimization. Later, a separate truth table was created, that function `minimize()` was calling behind the scenes.

But the normal input for the minimization procedure is a truth table, not a dataset. The function was designed to detect when the input is a dataset or a truth table, and if a dataset it would call the `truthTable()` function to create the actual input for the minimization. It amounts to the same thing, just less visible for the user.

Most of the arguments in the `minimize()` functions had nothing to do with the minimization process, they were just served to the `truthTable()` function. The only benefit of directly using a dataset instead of a truth table, is the specification of multiple outcomes to mimic CNA (details in Sect. 10.2).

Since they did not belong to the minimization function per se, arguments such as `outcome`, `conditions`, `incl.cut`, `n.cut`, `show.cases` are now removed as formal arguments for the function `minimize()`. The change is backwards compatible and it is still possible to specify these arguments, but they are passed to function `truthTable()`. The function now has the following form:

```
minimize(input, include = "", exclude = NULL, dir.exp = "", pi.cons = 0,
         pi.depth = 0, sol.cons = 0, sol.cov = 1, sol.depth = 0,
         min.pin = FALSE, row.dom = FALSE, all.sol = FALSE,
         details = FALSE, use.tilde = FALSE, method = "CCubes", ...)
```

The current formal arguments are specific to the minimization process, as it should logically be the case. The other previous formal arguments are now detected via the three dots “...” argument, and dealt with according to the situation. But truth table construction and minimization are two separate activities, and the new structure makes this more obvious.

Among others, the first argument is now called `input` instead of `data`, driven by the same idea that a normal input is not a dataset, but a truth table. The same change is also reflected in the graphical user interface dialog, that can be opened with the following menu:

Fig. 8.2 The starting minimization dialog

Analyse/Minimization

This is one of the most complex dialogs in the entire graphical user interface, and it makes full use of the newly added feature to display and use multiple objects at the same time. Many of the options will be presented in the next sections, for the moment we will cover the absolute basics.

First of all, in the upper left part of the dialog there is a radio button with two options: **Dataset** and **TT**, with a default on the **TT** meaning the truth table. Figure 8.2 displays nothing in the space underneath that radio button, because no truth tables have been assigned to particular objects. Truth tables can be created for visualization purposes only, but once the user is certain a truth table is ready for minimization, it should be assigned to an object.

Before creating one to see the difference, since all versions of the Lipset datasets (binary crisp, multi-value and fuzzy) are already loaded from the previous sections, a click on the **Dataset** option of the radio button displays those datasets, along with the columns from the dataset being selected, as in Fig. 8.3.

Only when the **Dataset** radio option is selected, will the options above the horizontal separator become available. As the input is not a truth table but a regular dataset, these options will no longer be greyed out and can be modified, just like their exact counterparts in the truth table dialog.

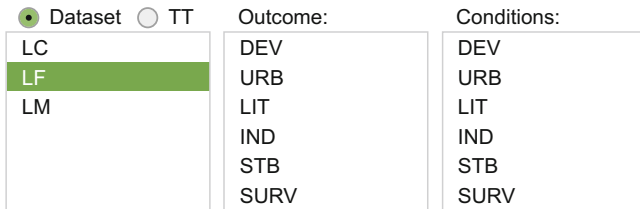


Fig. 8.3 Dataset option in the minimization dialog

Back to the main dialog in Fig. 8.2, another important change in the user interface is a slight re-arrangement of the check-boxes and textbox inputs. This rearrangement reflects the change in the written command arguments, but more importantly separates the options specific to the truth table from the options specific to the minimization process.

The horizontal separator between the two groups of options is new, and another important and visible modification is that options specific to the truth table (above the separator) are now greyed out. It means that, when the upper left radio button is set for the **TT** option, the user cannot modify the options specific to the truth table (more details in Sect. 8.2).

The space below the **Include** and **Exclude** options is empty for now, but will be populated with even more options when using directional expectations, with examples at Sect. 8.7.

8.2 Conservative (Complex) Solutions

A conservative solution is obtained with the classical QMC algorithm presented at the beginning of this chapter. It takes the cases where the outcome is present, and performs the necessary minimizations, in successive iterations, until the simplest prime implicants are generated.

Much like in the electrical engineering, the conservative solution is focused on the expressions associated with the presence of the outcome. Back in the days of McCluskey, the purpose of the minimization procedure was (and still is) to create a cheaper circuit that *does* something. It would be pointless to create an elaborate algorithm for something that *does not* produce any result.

Rather, the purpose of the classical QMC algorithm is to find the shortest, most simple expression that is equivalent with the initial positive configurations. This is the reason why, despite empirical evidence for configurations where the output is *not* present, the classical QMC algorithm ignores that evidence and focuses strictly on the configurations with a positive output.

Actually, the QMC algorithm is agnostic not only about observed empirical evidence for the negative output, but also about configurations with unknown output (where there is too little or no empirical evidence to assess the value of the output) and also about contradictions. All of those are ignored as if they are unknown, and the only empirical evidence that is used as input for the classical QMC function is the set of positive output configurations.

Unless a configuration has its output value changed to positive one, it does not play any role whatsoever in the minimization process, therefore it does not contribute to the final result.

It is the very reason why the solution is called “conservative”, because it does not make any assumptions about any other configurations (most notably the remainders) but those with a positive outcome. It is conservative when compared with the parsimonious solution (presented in the next section) which includes the remainders, hence making counterfactual assumptions for which little or no empirical evidence exists.

It is also called “complex”, because although the solution is a more simple expression than the initial configurations, the parsimonious solution is even more simple (contains fewer literals). The conservative solution is complex compared to the parsimonious solution, and the other way round, but both are equivalent to (and more simple than) the initial configurations.

To have an example of conservative minimization, a truth table has to exist in the workspace (assigned to an object) for function `minimize()` to find it. Using the crisp version of the Lipset data, it is assigned to object `ttLC`:

```
ttLC <- truthTable(LC, outcome = "SURV", incl.cut = 0.9, show.cases = TRUE)
```

Once this command is run, or its equivalent clicks in the truth table dialog from the graphical user interface, the options in the minimization dialog are automatically changed, as shown in Fig. 8.4.

The options above the separator, as well as the output and the conditions (all specific to the selected truth table), are still greyed out and cannot be modified. That happens because the truth table already exists, thus it does not make sense to modify things from an already created object. Modifying a truth table, or creating a new one with different options, should be done from the truth table dialog itself, or by modifying the equivalent written command (the new object will immediately become visible in the dialog).

However, the truth table options displayed in the minimization dialog are informative, especially if there are multiple truth tables to choose from. Selecting any existing truth table object will automatically refresh the options used at the time they were created. In this example, we used an inclusion cut-off equal to 0.9 and clicked on the option **show cases**.

Assigning the result of the minimization to the object `consLC` is left optional to the user. If not assigned, the result is simply printed on the screen (for the

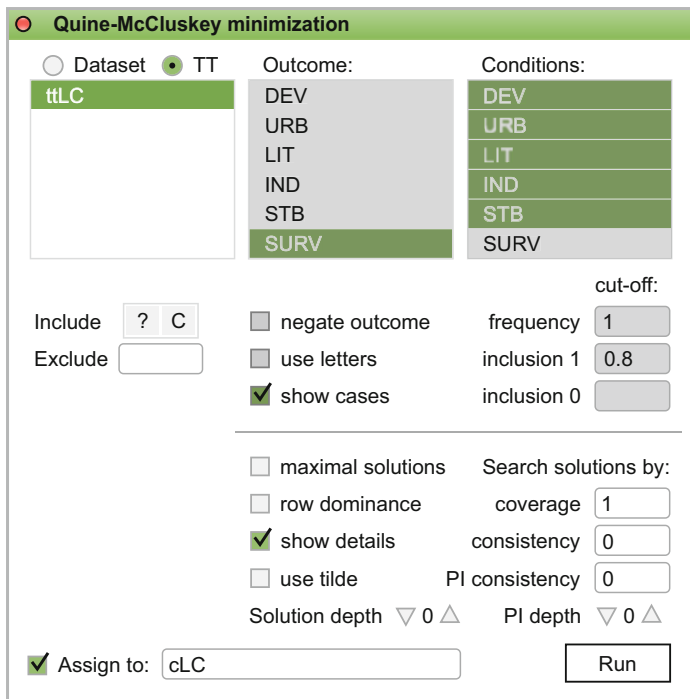


Fig. 8.4 The minimization dialog using an existing truth table

graphical user interface, in the web R console), but otherwise it is always a good idea to assign the results to an object for the simplest reason that it contains a lot more information than what is printed on the screen.

The minimization dialog, as presented on the screen, amounts to the following automatically generated command, in the command constructor dialog (Fig. 8.5).

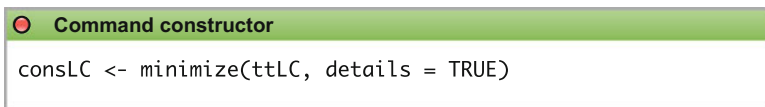


Fig. 8.5 The minimization command, auto-constructed from the dialog

There are three positive configurations in the object `ttLC` (numbers 22, 24 and 32, they can be found in the equivalent truth table from Sect. 7.1):
 $DEV \cdot urb \cdot LIT \cdot ind \cdot STB + DEV \cdot urb \cdot LIT \cdot INT \cdot STB + DEV \cdot URB \cdot LIT \cdot INT \cdot STB$

The minimization procedure produces two prime implicants in a disjunctive expression that is not only simpler and equivalent to the initial three configurations, but most of all it is both necessary and sufficient for the outcome:
 $DEV \cdot urb \cdot LIT \cdot STB + DEV \cdot LIT \cdot IND \cdot STB$


```
consLC <- minimize(ttLC, details = TRUE)
consLC
```

```
n OUT = 1/0/C: 8/10/0
  Total      : 18
```

Number of multiple-covered cases: 2

M1: DEV*urb*LIT*STB + DEV*LIT*IND*STB <=> SURV

	inclS	PRI	covS	covU	cases	
1	DEV*urb*LIT*STB	1.000	1.000	0.500	0.250	FI,IE; FR,SE
2	DEV*LIT*IND*STB	1.000	1.000	0.750	0.500	FR,SE; BE,CZ,NL,UK

M1		1.000	1.000	1.000		

It may not seem much, but with only three positive configurations the minimized solution cannot be expected to be more parsimonious than that. As a rule of thumb, the more positive configurations enter the minimization process, the more parsimonious the final solution(s) will get.

Notice that cases were printed in the parameters of fit table, even though not specifically required by the function `minimize()`. This is possible because that option is inherited from the input truth table `ttLC`.

Also note the relation `<=>` signals both necessity and sufficiency. Since this is a sufficiency solution, all prime implicants are sufficient; the necessity relation appears when the coverage for the solution (in this case `1.000`) is at least as high as the inclusion cut-off.

The object `consLC` is also structured as an R list with many other components, and some of them are going to play a pivotal role in the next sections:

```
names(consLC)

[1] "tt"           "options"      "negatives"    "initials"     "PIchart"
[6] "primes"      "solution"    "essential"    "inputcases"   "pims"
[11] "IC"          "numbers"     "SA"           "call"
```

The interesting one for this section is the prime implicants chart:

```
consLC$PIchart

          22 24 32
DEV*urb*LIT*STB  x  x  -
DEV*LIT*IND*STB  -  x  x
```

As expected, the chart shows the three initial primitive expressions, with both prime implicants needed to cover them.

8.3 What Is Explained

Before approaching the parsimonious solution, it is the right moment to concentrate on the defunct argument `explain` from function `minimize()`. The structure of the function used to have the explained value as "1" and it seemed redundant, for what could it possibly be explained other than the positive output? Remainders cannot be explained, as they are unobserved, perhaps contradictions but they are very few and seldom encountered.

More plausible is the negative output, and that generates a huge confusion. The negative output is many times mistaken with the negation of the outcome, but they are very different things. Consider the binary crisp Lipset data:

```
truthTable(LC, outcome = "SURV")
```

```
OUT: output value
     n: number of cases in configuration
incl: sufficiency inclusion score
PRI: proportional reduction in inconsistency
```

	DEV	URB	LIT	IND	STB	OUT	n	incl	PRI
1	0	0	0	0	0	0	3	0.000	0.000
2	0	0	0	0	1	0	2	0.000	0.000
5	0	0	1	0	0	0	2	0.000	0.000
6	0	0	1	0	1	0	1	0.000	0.000
22	1	0	1	0	1	1	2	1.000	1.000
23	1	0	1	1	0	0	1	0.000	0.000
24	1	0	1	1	1	1	2	1.000	1.000
31	1	1	1	1	0	0	1	0.000	0.000
32	1	1	1	1	1	1	4	1.000	1.000

When negating the outcome, notice how the output values are simply inverted:

```
truthTable(LC, outcome = "~SURV")
```

```
OUT: output value
     n: number of cases in configuration
incl: sufficiency inclusion score
PRI: proportional reduction in inconsistency
```

	DEV	URB	LIT	IND	STB	OUT	n	incl	PRI
1	0	0	0	0	0	1	3	1.000	1.000
2	0	0	0	0	1	1	2	1.000	1.000
5	0	0	1	0	0	1	2	1.000	1.000
6	0	0	1	0	1	1	1	1.000	1.000
22	1	0	1	0	1	0	2	0.000	0.000
23	1	0	1	1	0	1	1	1.000	1.000
24	1	0	1	1	1	0	2	0.000	0.000
31	1	1	1	1	0	1	1	1.000	1.000
32	1	1	1	1	1	0	4	0.000	0.000

This is a situation with perfect consistency scores (either 1 or 0), and indeed explaining the negative output or negating the outcome lead both to exactly the same solutions, hence it is understandable how matters can be confused. But truth tables do not always display perfect consistencies, as in the following two examples:

```
truthTable(LC, outcome = "SURV", conditions = "DEV, URB, LIT, IND")
```

```
OUT: output value
n: number of cases in configuration
incl: sufficiency inclusion score
PRI: proportional reduction in inconsistency
```

	DEV	URB	LIT	IND	OUT	n	incl	PRI
1	0	0	0	0	0	5	0.000	0.000
3	0	0	1	0	0	3	0.000	0.000
11	1	0	1	0	1	2	1.000	1.000
12	1	0	1	1	0	3	0.667	0.667
16	1	1	1	1	0	5	0.800	0.800

While the upper three configurations have perfect consistencies, notice the bottom two configurations that have less perfect scores and their output values remain negative after negating the outcome:

```
truthTable(LC, outcome = "~SURV", conditions = "DEV, URB, LIT, IND")
```

```
OUT: output value
n: number of cases in configuration
incl: sufficiency inclusion score
PRI: proportional reduction in inconsistency
```

	DEV	URB	LIT	IND	OUT	n	incl	PRI
1	0	0	0	0	1	5	1.000	1.000
3	0	0	1	0	1	3	1.000	1.000
11	1	0	1	0	0	2	0.000	0.000
12	1	0	1	1	0	3	0.333	0.333
16	1	1	1	1	0	5	0.200	0.200

Explaining the negative output for the presence of the outcome (rows "1", "3", "11" and "16" in the first) would definitely not give the same solutions as explaining the positive output for the negation of the outcome (rows "1" and "3" in the second), the configurations entering the minimization process being different from their complementary set of configurations in the first truth table.

Hopefully, this digression makes it clear that explaining the negative output doesn't make any sense: it is not the same thing as negating the outcome, and it is meaningless to explain configurations having consistencies below the inclusion cut-off. All of this becomes much clear if refraining from directly minimizing the data and produce truth tables beforehand.

Conversely, as it most likely is the case, users who are not aware of the difference simply trust the software knows what it's doing. That is always a danger.

For this reason, starting with version 3.0 the package *QCA* outputs an error when trying to explain or include negative outputs. Taking remainders away, since by definition they cannot be explained, the only other value of the output that remains to discuss is represented by the contradictions.

The difference between explaining and including contradictions is found in the construction of PI charts, where the primitive expressions in the columns are represented by the explained configurations:

- explaining contradictions (in addition to positive outputs) results in adding more columns to the prime implicants chart;
- including the contradictions, they are treated similar to remainders, contributing in the minimization process but not affecting the PI chart;
- if neither explaining, nor including the contradictions, they are by default treated as negative output configurations.

As an example, we will again reproduce the truth table containing for the fuzzy version of the Lipset data tweaking the inclusion cut-offs to produce more contradictions:

```
ttLF1 <- truthTable(LF, outcome = "SURV", incl.cut = c(0.8, 0.5))
ttLF1
```

```
OUT: output value
n: number of cases in configuration
incl: sufficiency inclusion score
PRI: proportional reduction in inconsistency
```

	DEV	URB	LIT	IND	STB	OUT	n	incl	PRI
1	0	0	0	0	0	0	3	0.216	0.000
2	0	0	0	0	1	0	2	0.278	0.000
5	0	0	1	0	0	C	2	0.521	0.113
6	0	0	1	0	1	C	1	0.529	0.228
22	1	0	1	0	1	1	2	0.804	0.719
23	1	0	1	1	0	0	1	0.378	0.040
24	1	0	1	1	1	C	2	0.709	0.634
31	1	1	1	1	0	0	1	0.445	0.050
32	1	1	1	1	1	1	4	0.904	0.886

First inspect the complex solution and the associated PI chart when explaining the contradictions:

```
minimize(ttLF1, explain = "1, C")
```

```
M1: dev*urb*LIT*ind + DEV*LIT*IND*STB + (DEV*urb*LIT*STB) => SURV
M2: dev*urb*LIT*ind + DEV*LIT*IND*STB + (urb*LIT*ind*STB) => SURV
```

```
minimize(ttLF1, explain = "1, C")$PIchart
```

	5	6	22	24	32
dev*urb*LIT*ind	x	x	-	-	-
DEV*urb*LIT*STB	-	-	x	x	-
DEV*LIT*IND*STB	-	-	-	x	x
urb*LIT*ind*STB	-	x	x	-	-

The same solution, with the same prime implicants and the same PI chart can be produced by lowering the inclusion cut-off until all contradictions become explained:

```
ttLF2 <- truthTable(LF, outcome = "SURV", incl.cut = 0.5)
minimize(ttLF2)
```

M1: dev*urb*LIT*ind + DEV*LIT*IND*STB + (DEV*urb*LIT*STB) => SURV
 M2: dev*urb*LIT*ind + DEV*LIT*IND*STB + (urb*LIT*ind*STB) => SURV

```
minimize(ttLF2)$PIchart
```

	5	6	22	24	32
dev*urb*LIT*ind	x	x	-	-	-
DEV*urb*LIT*STB	-	-	x	x	-
DEV*LIT*IND*STB	-	-	-	x	x
urb*LIT*ind*STB	-	x	x	-	-

As it can be seen, choosing to add the contradictions in the `explain` argument (along with the positive output) or lowering the inclusion cut-off until the contradictions become explained (they would be allocated a positive output themselves), lead to the same complex solutions.

A similar phenomenon happens when including the remainders:

```
minimize(ttLF1, explain = "1, C", include = "?")
```

M1: dev*LIT + DEV*STB => SURV
 M2: dev*LIT + LIT*STB => SURV
 M3: DEV*STB + LIT*ind => SURV
 M4: LIT*ind + LIT*STB => SURV
 M5: LIT*ind + IND*STB => SURV

```
minimize(ttLF2, include = "?")
```

M1: dev*LIT + DEV*STB => SURV
 M2: dev*LIT + LIT*STB => SURV
 M3: DEV*STB + LIT*ind => SURV
 M4: LIT*ind + LIT*STB => SURV
 M5: LIT*ind + IND*STB => SURV

The solutions are again identical, which is to be expected since the PI charts are also identical:

```
minimize(ttLF1, explain = "1, C", include = "?")$PIchart
```

	5	6	22	24	32
dev*LIT	x	x	-	-	-
DEV*ind	-	-	x	-	-
DEV*STB	-	-	x	x	x
URB*STB	-	-	-	-	x
LIT*ind	x	x	x	-	-
LIT*STB	-	x	x	x	x
IND*STB	-	-	-	x	x

```
minimize(ttLF2, include = "?")$PIchart
```

	5	6	22	24	32
dev*LIT	x	x	-	-	-
DEV*ind	-	-	x	-	-
DEV*STB	-	-	x	x	x
URB*STB	-	-	-	-	x
LIT*ind	x	x	x	-	-
LIT*STB	-	x	x	x	x
IND*STB	-	-	-	x	x

This proves that explaining contradictions is meaningless because it leads to exactly the same solutions as lowering the inclusion cutoff. It is an important observation, leaving the argument `explain` with only one logically possible value (positive output configurations) and that makes it redundant.

It has survived through different software, from one version to another, since the beginning of crisp sets QCA when fuzzy sets and consistency scores have not been introduced yet. But once consistency scores made their way into the truth table, this argument should have been abandoned, for as it turns out no other output except the positive one makes any logical sense to explain. And since "1" is the default value, it is perhaps the reason why most of the times this argument is not even explicitly mentioned in users' commands.

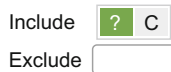


Fig. 8.6 The “Include” and “Exclude” tuple in the minimization dialog

In the graphical user interface, the argument `include` is located in the left part of the minimization dialog (Fig. 8.6), above another argument `exclude` that will be introduced later. Although specific to function `minimize()`, they are located above the separating line at the same level with the truth table options, which seems logical since they refer to the output values in the truth table.

8.4 Parsimonious Solutions

A parsimonious solution is a more simplified but equivalent expression, compared to the complex solution. It is obtained by employing a less conservative approach over the empirical evidence and include remainders in the minimization process. Before delving into the technical details of this solution, some preliminary considerations need to be made about the problem of limited diversity and the different strategies to deal with this problem in social research methodology.

Conventional statistical analysis manages to produce rather accurate predictions for the values in the dependent variable, even in the absence of empirical information. It does that by inferring the information drawn from existing evidence can be extrapolated to empty areas where no such evidence exists. The simplest example is the linear regression model, where all predicted values are deterministically located on the regression line.

Things are rather clear for a scatterplot with the relation between the independent and the dependent variable, and even for a 3D scatterplot with two independent variables versus the dependent, the prediction being made on the regression (hyper)plane. It is easy to visualize where the points are (the empirical data), and where there are no points to make predictions.

Once the number of independent variables grows, visualizing empty areas becomes much more difficult and the inference is usually accepted as a natural expansion from the simple 2D or 3D examples to a k -dimensional cube (vector space), with some strong assumptions like the multivariate normality.

But it is a fact, as Ragin and Sonnett (2008) eloquently point out, that social reality is very limited in its diversity. This should not be confused with a limited number of cases (the small-N or medium-N research situation), instead it actually refers to a limited number of empirically observed configurations in the truth table. It does not matter how large the data is (how many cases it has), if all of them cluster in a few such configurations.

Thompson (2011) analyzed the 1970 Birth Cohort Study (BCS70) in UK, a large study with no less than 17,000 cases and still ran into the problem of limited diversity. Whatever we empirically observe seems to cover a very small area of the entire vector space, and for the biggest part of this space researchers either make strong assumptions to extrapolate the model through statistical regularities, or engage in counterfactual analysis.

This aspect of limited diversity is well addressed in the QCA methodology, due to the very structure of the truth table that not only guides researchers to think in terms of configurations but more importantly shows exactly how much (or how small) empirical evidence exists in the property space. Unlike the quantitative methodology where areas empty of empirical evidence are covered by an estimated underlying distribution, QCA uses exclusively the

existing evidence to map the property space and specifically expose all unknown configurations.

The normal scientific way to deal with lack of evidence (non-existence or incomplete data) is to perform experiments and produce the data needed to formulate theories. Science is driven by curiosity, and where such experiments are possible researchers engage into an active process of discovery by tweaking various input parameters and observe if (and how much) the outcome changes.

But social sciences are not experimental, researchers being unable to tweak the input parameters. Being left with the observed empirical information at hand, they conduct thought experiments involving counterfactual analysis.

In the absence of direct evidence for critical cases to compare against, researchers often resort to the question “*What if...?*” and begin wondering what the reality would be like, if things in the past would have been different. Such counterfactual thinking can be traced back to Hume (1999, p. 146), where in his definition of causation the following statement is found:

... if the first object had not been, the second never had existed.

This is a clear counterfactual statement, and social sciences (especially in the qualitative research tradition) abounds in such statements particularly if the events being studied are rare, such as state revolutions, as the complex reality is reduced to an abstract concept (the ideal type) that does not exist in the reality, for which there is no empirical evidence to claim its existence.

It seems that counterfactual thinking is not only desirable, but actually indispensable to advance social research. In QCA, all remainders are potential counterfactual statements that can be used to further minimize the observed configurations. The decision to include remainders in the minimization makes the implicit assumption that, should it be possible to empirically observe these configurations, they would have a positive output.

But this is a very serious assumption(!) that can be subjected to immediate question marks, mainly to how impossible configurations contribute to a meaningful minimal solution, or how remainders that contradict our theory contribute to a solution that confirms the theory. Such issues will be dealt with, in the next sections.

Existing software make it extremely easy to add remainders in the minimization process, and most likely (depending on how large the truth table is and how many empirical configurations do exist) the solutions will be less complex.

There is a high temptation to rush and find the simplest possible (most parsimonious) solution, indiscriminately throwing anything into the minimization routine, something similar to grabbing all available ingredients from a kitchen and lump everything into the oven to make food. But not all ingredients go well together, and not all of them can be cooked in the oven.

Fortunately the “food” can be inspected post-hoc, after the minimization, but for the moment it is good to at least be alert of the situation. Either before the minimization, or post (or even better before and post) the minimization, the structure of the remainders should constantly be monitored.

For the pure parsimonious solution, the argument of interest in function `minimize()` was already introduced. It is `include`, specifying which of the other output values (“?” and “C”) are *included* in the minimization process. The Quine-McCluskey algorithm treats remainders as having the same output value as the explained configurations, but they will not be used to construct the PI chart as remainders are not primitive expressions.

Having a truth table object available, the simplest form of the minimization command is:

```
ttLF <- truthTable(LF, outcome = "SURV", incl.cut = 0.8, show.cases =TRUE)
minimize(ttLF, include = "?")
```

M1: DEV*ind + URB*STB => SURV

This command outputs just the most parsimonious solution, without any other details about the parameters of fit. It also uses by default upper case letters for their presence and lower case letters for their absence. Choosing to print the details, and to using a tilde to signal a negated (absent) condition, the command and the output changes to:

```
minimize(ttLF, include = "?", details = TRUE)
```

```
n OUT = 1/0/C: 6/12/0
  Total      : 18
```

Number of multiple-covered cases: 0

M1: DEV*ind + URB*STB => SURV

	inclS	PRI	covS	covU	cases
1 DEV*ind	0.815	0.721	0.284	0.194	FI,IE
2 URB*STB	0.874	0.845	0.520	0.430	BE,CZ,NL,UK
M1	0.850	0.819	0.714		

In the table containing the parameters of fit, there are two sufficient conjunctions, each covering part of the empirically observed positive configurations. They are displayed with a logical OR relation, either one being sufficient for some of the positive configurations, but both are needed to cover all of them.

Both are highly consistent sufficient expressions but their cumulated unique coverage is lower, suggesting there is some more space in the outcome set that remains unexplained (Fig. 8.7).

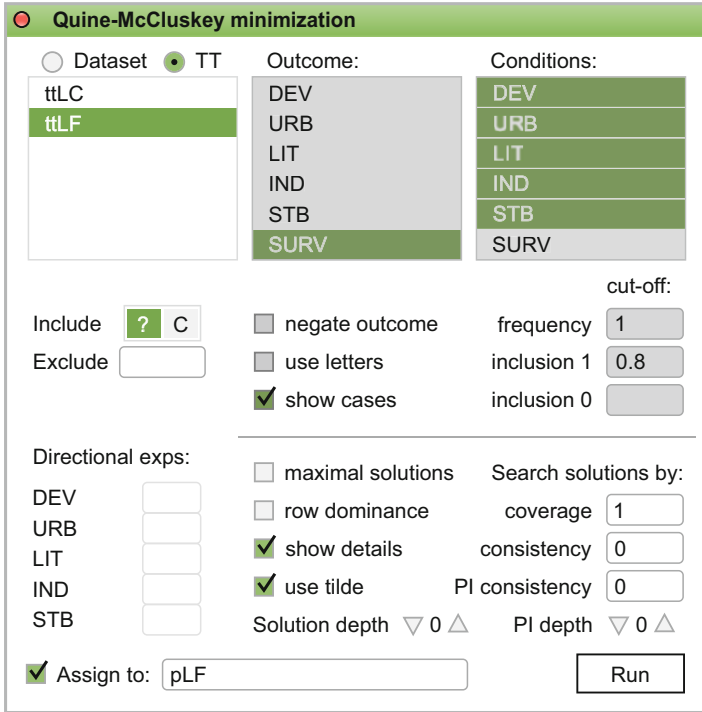


Fig. 8.7 Including remainders using the minimization GUI dialog

In the dialog from the graphical user interface, the options being selected accurately reflect the written minimization command: the remainders "?" are selected, and the checkboxes **show details** and **use tilde** are checked. A notable aspect is the display of the cases in the parameters of fit table, this option being inherited from the truth table used as input for the minimization.

Below the separating line there are some other options, where **depth** has two sets of counters, one for a maximum number of conditions in conjunctive prime implicants, and the other for a maximum number of prime implicants in disjunctive solutions. At the default value equal to 0, it signals an exhaustive search for all possible PIs and all possible solutions. The solution depth is useful only when **maximal solutions** is checked, and/or the consistency threshold for the solution is lower than 1.

The other two options in the left side are specific to solving the prime implicants chart. The **row dominance** option eliminates irrelevant PIs from the truth table before finding the minimal solutions, if they are covered by other prime implicants. Finally, **maximal solutions** finds all possible non-redundant disjunctions of prime implicants that cover all initial, truth table positive configurations, even if not minimal. It is mainly used to mimic CNA, discussed in Sect. 10.2.

8.5 A Note on Complexity

Describing the Standard Analysis procedure, Schneider and Wagemann (2012, p. 161) introduce three dimensions to classify the types of solutions obtained using different logical remainders:

1. set relation
2. complexity
3. type of counterfactuals

Having analyzed all possible combinations of remainders from their truth table to include in the minimization, they meticulously show that the parsimonious solution is the least complex one, and it is a superset of not only the complex solution but also of all other solutions using various remainders.

Defining the complexity of a solution "... by the number of conditions and the logical operators AND and OR that it involves ...", they advise to refrain from using the alternative name of "complex" solution, for the conservative one, because some of the solutions involving remainders are even more complex than the complex one (which does not make any assumption on, and does not include any remainders).

While certainly valid from a logical point of view, Schneider and Wagemann's conclusions are not quite right from an objective point of view. Their truth table can be replicated with the following commands:

```
SW <- data.frame(A = c(0, 0, 0, 1, 1), B = c(0, 1, 1, 0, 1),
                C = c(1, 0, 1, 1, 0), Y = c(1, 0, 0, 1, 1))
ttSW <- truthTable(SW, outcome = "Y", complete = TRUE)
ttSW
```

```
OUT: output value
n: number of cases in configuration
incl: sufficiency inclusion score
PRI: proportional reduction in inconsistency
```

	A	B	C	OUT	n	incl	PRI
1	0	0	0	?	0	-	-
2	0	0	1	1	1	1.000	1.000
3	0	1	0	0	1	0.000	0.000
4	0	1	1	0	1	0.000	0.000
5	1	0	0	?	0	-	-
6	1	0	1	1	1	1.000	1.000
7	1	1	0	1	1	1.000	1.000
8	1	1	1	?	0	-	-

The row numbers are not the same, their ordering being a bit different from those in package *QCA* where they represent the base 10 conversion from base 2, but all configurations are identical for the three positive ones (001, 101, 110),

for the two negative ones (010, 011) and finally for the three remainders left (000, 100, 111).

For three remainders, there are eight possible way to use them in the minimization process, from none (the equivalent of the conservative solution) to all (leading to the parsimonious solution). Schneider and Wagemann then show their solution (e) that includes only the remainder 000 leads to the most complex solution: $\sim A \cdot \sim B + \sim B \cdot C + A \cdot B \cdot \sim C$.

However, the actual solution using that remainder does not seem to entirely coincide:

```
pSW <- minimize(ttSW, include = "?", exclude = c(5, 8), use.tilde = TRUE)
pSW
```

M1: $\sim B * C + A * B * \sim C <=> Y$

This command can be translated as “include all remainders except those from row numbers 5 and 8”, and of course since there are only three remainders it means that only one is actually included (row number 1, that is 000).

But it is obvious this solution is not the same with Schneider and Wagemann’s. It most likely happened because they tried to explain the remainder (which doesn’t make sense), instead of just including it in the minimization. That affected the PI chart, which is supposed to have only as many columns as the initial number of primitive expressions (in this situation, three).

Consequently, their PI chart contains an additional column:

	1	2	6	7
$\sim A * \sim B$	x	x	-	-
$\sim B * C$	-	x	x	-
$A * B * \sim C$	-	-	-	x

while the correct PI chart renders $\sim A \cdot \sim B$ as irrelevant (it is *dominated*):

```
pSW$PIchart
```

	2	6	7
$\sim A * \sim B$	x	-	-
$\sim B * C$	x	x	-
$A * B * \sim C$	-	-	x

The purpose of this demonstration is less to point the finger to a reasonable and easy to make error, but more to restate that as it turns out, no other solution can be more complex than the conservative one. At best, other solutions can be equally complex therefore the word “complex” doesn’t always identify with the conservative solution. I would also advocate using the word “conservative”, for the same stated, and more important reason that it does not make any assumption on the remainders.

8.6 Types of Counterfactuals

This is one of the most informationally intensive sections from the entire analysis of sufficiency, and at the same time the pinnacle of all the information presented so far. Everything that has been introduced up to this point, plus many others, is extensively used in what is about the follow.

If things like constructing a truth table or logical minimization are relatively clear, this section contains an entire host of additional concepts where many overlap in both understanding and dimensionality. Users who possess at least a minimal understanding of what QCA is used for, undoubtedly heard about remainders, directional expectations, different types of counterfactuals (easy and difficult, impossible, implausible, incoherent), simplifying assumptions, contradictory simplifying assumptions, tenable and untenable assumptions, all within the context of Standard Analysis, Enhanced Standard Analysis, Theory-Guided Enhanced Standard Analysis etc.

Without a deep understanding of each such concept, they tend to gravitate around in a spinning motion that can provoke a serious headache. It is therefore mandatory to have a thorough introduction for these concepts, before using the R code in package *QCA*. Most of them are already discussed and properly introduced in multiple other places, see Ragin and Sonnett (2005), Ragin (2008b), Schneider and Wagemann (2012, 2013) to name the most important ones. Some of these concepts are going to be deepened in Chap. 10, therefore the beginning of this section is not meant to replace all of those important lectures but mainly to facilitate their understanding.

It should be clear, however, that all of these concepts are connected to the remainders and how are they included, filtered or even excluded from the minimization process to obtain a certain solution.

In this book, the terms “counterfactuals” and “remainders” can be used interchangeably. They are synonyms and refer to one and the same thing: causal configurations that, due to the issue of limited diversity in social phenomena, have no empirical evidence. They are unobserved configurations which, if by any chance would be observed, could contribute in the minimization process to obtain a more parsimonious solution.

In Sect. 8.5, I showed that the parsimonious solution is the least complex one, and no other solution is more complex than the conservative one, leading to the following **Complexity rule**:

Any remainder included in the minimization process can only make the solution simpler. Never more complex, always more and more parsimonious.

To the limit, when all remainders are included in the minimization, the final solution is “the” most parsimonious solution.

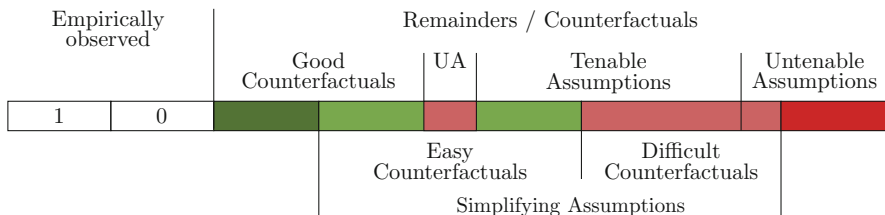


Fig. 8.8 The truth table composition

Figure 8.8 is yet another attempt to classify the remainders based on the current theory, using a linear approach and various vertical separators. The first delimitation is made between the empirically observed configurations and the remainders (counterfactuals). Among those empirically observed, some configurations have a positive output, while some have a negative output (for simplicity, this figure assumes there are no contradictions).

Among the remainders on the right hand side of the figure:

- some are simplifying assumptions and some are not (simplifying meaning they contribute to the logical minimization)
- the simplifying assumptions is a set composed from the easy counterfactuals and the difficult counterfactuals
- remainders which are not simplifying assumptions can be simply good counterfactuals (that don't have any logical difficulties, but don't contribute to the logical minimization), or untenable assumptions
- easy counterfactuals are also good counterfactuals, but some are also part of the tenable assumptions
- some of the difficult counterfactuals are also tenable, while some are untenable etc.

The meaning for all of these is going to be gradually unfolded. They are all connected with the moment when QCA theoreticians realized that not all remainders can be used as counterfactuals.

A very simple example is the relation between the presence of the air (as a trivial necessary condition) and the existence of a big fire in a city. Undoubtedly, the air is not a cause for a fire, but on the other hand a fire without air is impossible in normal situations.

There are numerous non-trivial causes for a fire, and a researcher might select 5 or 6 most commonly observed conditions (among which the necessary air). Due to the limited diversity, many of the 64 logically possible configurations are not empirically observed, and they could be used as counterfactuals for the parsimonious solution.

The problem is that many of the remainders also contain the absence of air.

According to the **Complexity rule**, any remainder has the potential to make the solution less and less complex, including those where the air is absent. But such remainders are difficult counterfactuals, despite making the solution simpler, since they contradict all our knowledge about how a fire is maintained.

For any given phenomenon (outcome) of interest, there is some established theoretical corpus that guides the research and offers a number of hints about how causal conditions should contribute to the presence of that outcome. These can be called *directional expectations*.

We *expect* air to be present when a fire is produced, therefore all remainders that do not conform to this expectation are difficult. To put it differently, it would be very difficult to explain how did we arrive at a minimal explanation (solution) about the presence of fire, involving remainders associated with the absence of air in the minimization.

For this reason, out of all possible remainders in the truth table, a sensible decision would be to filter out those which contradict our theory. By contrast, those remainders which are accepted in the minimization process are called *easy counterfactuals*, because they are in line with our current knowledge.

With a good part of the remainders out of the minimization, it is clear the final solution will not be the most parsimonious one. It will be less complex compared with the conservative solution, but more complex than the parsimonious one. This is a key idea presented by Ragin and Sonnett (2005), who introduced the concepts of easy and difficult counterfactuals in QCA.

Later, Ragin (2008b) wrapped up the whole procedure which is today known as the Standard Analysis, that produces three types of solutions: conservative, intermediate and parsimonious.

To reach the parsimonious solution, all remainders are included in the minimization process. But not all of them really contribute to the minimization, some of the pairs being compared differ by more than one literal, hence they don't produce implicants for the next iterations. Consequently, part of the remainders are never used in the minimization. Those remainders that do help producing prime implicants are called Simplifying Assumptions and, as shown in Fig. 8.8 they are composed from the Easy Counterfactuals plus the Difficult Counterfactuals:

$$SA = EC + DC$$

This is as far as the Standard Analysis goes, separating the simplifying assumptions from all remainders, and differentiating between easy and difficult counterfactuals. This model would be extremely effective if not for the red area located right in the middle of the easy counterfactuals segment. It is called UA, part of a category identified by Schneider and Wagemann (2013), and dubbed the Untenable Assumptions.

Many things can be Untenable Assumptions:

- logical impossibilities (the famous example of the pregnant man);
- *contradictory simplifying assumptions* that end up being sufficient for both the outcome and its negation (Yamasaki and Rihoux 2009);
- a most interesting category of that combines the analysis of sufficiency with the analysis of necessity.

The later category is part of the *incoherent counterfactuals*, and makes a full use of the mirror effect between necessity and sufficiency: when a condition is identified as necessary for an outcome Y, it is a superset of the outcome. The negation of the condition, therefore, cannot be assumed to be sufficient for the outcome for it would need to be a subset of the outcome (to be sufficient).

But since the condition itself is bigger than the outcome set, what is outside the condition ($\sim X$) is by definition outside the outcome set: it has no inclusion in the outcome (it would be illogical to think otherwise), as it can be seen in Fig. 8.9.

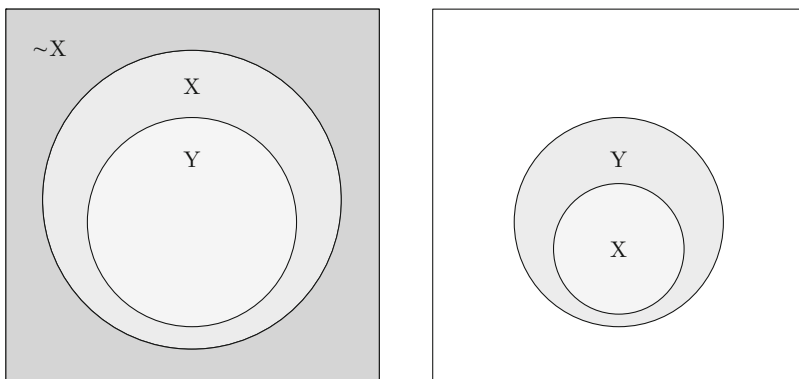


Fig. 8.9 $\sim X \not\Rightarrow Y$ (left, $\sim X$ not a subset of Y), while $X \Rightarrow Y$ (right, X is a subset of Y)

This is one of the key aspects put forward by Schneider and Wagemann (2013) and poorly understood by some QCA users (Thiem 2016), as it was shown in Sect. 5.5.

Indeed, when a condition (more generally a phenomenon) is necessary for an outcome, its negation cannot be at the same time sufficient for the same outcome, therefore any counterfactual containing the negation of a necessary condition should be eliminated from the minimization process.

This is the starting line for the Enhanced Standard Analysis, an extension of the standard analysis by further eliminating the untenable assumptions from the entire minimization process. Not only to create intermediate solutions, but more importantly remove the untenable assumptions from the most parsimonious solution(s) as well.

But Schneider and Wagemann (2013) didn't stop here and went even further than ESA, formulating what is now known as TESA—Theory-Driven Enhanced Standard Analysis, that expands the inclusion of remainders by formulating the so-called *conjunctural directional expectations*.

The difference from the simple directional expectations is the formulation of these expectations not only for atomic conditions, but for entire conjunctions of conditions. Something like: we expect a fire to appear when there is both air AND a sparkle AND inflammable materials around: $(A \cdot S \cdot I \Rightarrow F)$

There is still a need to prove the usefulness of using conjunctural directional expectations, since a similar result can be obtained by formulating atomic directional expectations, not necessarily in conjunction:

- we expect a fire when there is air ($A \Rightarrow F$)
- we expect a fire when there is a sparkle ($S \Rightarrow F$)
- we expect a fire when there are inflammable materials around ($I \Rightarrow F$)

The example provided by Schneider and Wagemann, that is going to be replicated in the next section, suffers from the same benign (but still an) error to explain these remainders and change the PI chart in the process. Contrary to their findings the actual solution is equal to the conservative one, therefore in the absence of a more concrete example it is unclear what TESA brings more to the table.

On the other hand, ESA is a welcomed addition to the methodological QCA field and I fully agree with its creators that implausible counterfactuals (especially those which contain the negation of a necessary condition) should be removed from the minimization process.

Finally, this section should not be finished without discussing yet another type of truth table rows that can be excluded from the minimization. It is not about remainders, but about the empirically observed configurations and do not belong to the counterfactual analysis. The property on which we might eliminate observed rows is called *simultaneous subset relations*.

Some of these observed configurations might pass the sufficiency threshold for both the presence and for the absence of the outcome, which renders them incoherent. Although empirically observed, one possible decision is to remove them too from the minimization. This will of course change the final solution, but at least it would be based on firmly and coherently consistent observed configurations.

There are also the *true logical contradictory cases* (Schneider and Wagemann 2012, p. 127), that have an inclusion in the configuration higher than 0.5, but a lower (than 0.5) inclusion in the outcome. More recently, Schneider and Rohlfing (2013, p. 585) prefer to call these *deviant cases consistency in kind*, so the QCA terminology is not only expanding but is also overlapping. A bit confusing, but hopefully this section will have made some helpful clarifications.

8.7 Intermediate Solutions: SA and ESA

No less than three arguments are available when referring to the remainders: `include`, `exclude` and `dir.exp`. They should be self-explanatory, first specifying what to include in the minimization, the second what to exclude from the minimization, and the third specifying the directional expectations.

Assuming the reader didn't go through all previous code examples where multiple truth tables have been created, we will again use the fuzzy version of the Lipset data. The very first step is to create and inspect the truth table:

```
data(LF)
ttLF <- truthTable(LF, "SURV", incl.cut = 0.8, show.cases = TRUE,
                  sort.by = "OUT, n")
```

The function `minimize()` allows, and there is a strong tendency to initiate a direct minimization process using the dataset, bypassing the truth table, but I would warmly recommend producing a truth table first.

The truth table looks like this:

```
ttLF
```

```
OUT: output value
n: number of cases in configuration
incl: sufficiency inclusion score
PRI: proportional reduction in inconsistency
```

	DEV	URB	LIT	IND	STB	OUT	n	incl	PRI	cases
32	1	1	1	1	1	1	4	0.904	0.886	BE,CZ,NL,UK
22	1	0	1	0	1	1	2	0.804	0.719	FI,IE
1	0	0	0	0	0	0	3	0.216	0.000	GR,PT,ES
2	0	0	0	0	1	0	2	0.278	0.000	IT,RO
5	0	0	1	0	0	0	2	0.521	0.113	HU,PL
24	1	0	1	1	1	0	2	0.709	0.634	FR,SE
6	0	0	1	0	1	0	1	0.529	0.228	EE
23	1	0	1	1	0	0	1	0.378	0.040	AU
31	1	1	1	1	0	0	1	0.445	0.050	DE

With 5 causal conditions, there are 32 rows in the truth table, out of which 2 positive output, 7 negative output configurations, and 23 remainders. The truth table is sorted first by the values of the `OUT` column (in descending order), then by the values of the frequency column `n`. The consistency scores are unsorted, but this choice of structure emulates Schneider and Wagemann (2013).

Note the row numbers are also changed, due to the sorting choices. But each configuration, remainders included, has a unique row number and that is going to prove very useful when deciding which to include and which to remove from the minimization process.

Before proceeding to the minimization, it is a good idea to check if there are deviant cases consistency in kind, in the truth table, using the argument `dcc`:

```
truthTable(LF, "SURV", incl.cut = 0.8, show.cases = TRUE, dcc = TRUE,
           sort.by = "OUT, n")
```

OUT: output value
 n: number of cases in configuration
 incl: sufficiency inclusion score
 PRI: proportional reduction in inconsistency
 DCC: deviant cases consistency

	DEV	URB	LIT	IND	STB	OUT	n	incl	PRI	DCC
32	1	1	1	1	1	1	4	0.904	0.886	
22	1	0	1	0	1	1	2	0.804	0.719	
1	0	0	0	0	0	0	3	0.216	0.000	GR,PT,ES
2	0	0	0	0	1	0	2	0.278	0.000	IT,RO
5	0	0	1	0	0	0	2	0.521	0.113	HU,PL
24	1	0	1	1	1	0	2	0.709	0.634	
6	0	0	1	0	1	0	1	0.529	0.228	EE
23	1	0	1	1	0	0	1	0.378	0.040	AU
31	1	1	1	1	0	0	1	0.445	0.050	DE

The DCC cases are all associated with the negative output configurations, which is good. As long as they are not associated with the positive output configurations, everything seems to be alright.

The “pure” parsimonious solution is produced, as always, by specifying the question mark (“?”) in the include argument:

```
pLF <- minimize(ttLF, include = "?", details = TRUE, show.cases = TRUE)
pLF
```

n OUT = 1/0/C: 6/12/0
 Total : 18

Number of multiple-covered cases: 0

M1: DEV*ind + URB*STB => SURV

	inclS	PRI	covS	covU	cases
1 DEV*ind	0.815	0.721	0.284	0.194	FI,IE
2 URB*STB	0.874	0.845	0.520	0.430	BE,CZ,NL,UK
M1	0.850	0.819	0.714		

For this solution, the assumption is that all remainders contribute equally to parsimony, despite the fact that not all of them have been used by the minimization algorithm. The simplifying assumptions (the set of remainders actually contributing to the minimization process), can be seen by inspected the component `SA` from the newly created object:

```
pLF$SA
```

```
$M1
  DEV URB LIT IND STB
10  0  1  0  0  1
12  0  1  0  1  1
14  0  1  1  0  1
16  0  1  1  1  1
17  1  0  0  0  0
18  1  0  0  0  1
21  1  0  1  0  0
25  1  1  0  0  0
26  1  1  0  0  1
28  1  1  0  1  1
29  1  1  1  0  0
30  1  1  1  0  1
```

Out of the 23 remainders, only 12 have been used and the rest did not contribute at all to the minimization. These SAs might contain both easy and difficult counterfactuals, as it will be seen later. For the moment, the currently recommended procedure is to continue preparing for the intermediate solution, by verifying the existence of contradictory simplifying assumptions.

This can be done in two ways, the first and most intuitive being to produce another minimization on a truth table for the negation of the output (using the same inclusion cut-off) assigned to an object pLFn, and check its simplifying assumptions component:

```
ttLFn <- truthTable(LF, outcome = "~SURV", incl.cut = 0.8)
pLFn <- minimize(ttLFn, include = "?")
pLFn$SA
```

```
$M1
  DEV URB LIT IND STB
 3  0  0  0  1  0
 4  0  0  0  1  1
 7  0  0  1  1  0
 8  0  0  1  1  1
 9  0  1  0  0  0
10  0  1  0  0  1
11  0  1  0  1  0
12  0  1  0  1  1
13  0  1  1  0  0
14  0  1  1  0  1
15  0  1  1  1  0
16  0  1  1  1  1
17  1  0  0  0  0
19  1  0  0  1  0
21  1  0  1  0  0
25  1  1  0  0  0
27  1  1  0  1  0
29  1  1  1  0  0
```

Since some of the rows are present in both matrices, it is a proof there are indeed contradictory simplifying assumptions. To identify exactly which ones are found in both:

```
intersect(rownames(pLF$SA$M1), rownames(pLFn$SA$M1))
```

```
[1] "10" "12" "14" "16" "17" "21" "25" "29"
```

According to Enhanced Standard Analysis, these CSAs should be avoided and not included in the minimization process. This identification method works when there is only one solution for both the presence and the absence of the outcome. If multiple solutions exist, the component SA will contain the simplifying assumptions for each solution, some obviously being duplicated since they contribute to more than one solution.

A second and more straightforward way to check for contradictory simplifying assumptions is to use the built-in function `findRows()` which takes care of all these details. The input for this function is either one of the two truth tables, usually the one for the presence of the outcome:

```
findRows(obj = ttLF, type = 2)
```

```
[1] 10 12 14 16 17 21 25 29
```

The conclusion is the same, this function being more integrated and saving users from having to type additional commands. The purpose of finding these CSAs is going to be introduced later, for the time being a next step in the Standard Analysis is to specify directional expectations, for the purpose of constructing intermediate solutions.

For the sake of simplicity, we will be assuming it is the presence of the causal conditions that lead to the survival of democracy (the presence of the outcome). These expectations are specified using the argument `dir.exp`:

```
iLF <- minimize(ttLF, include = "?", dir.exp = "1,1,1,1")
```

This argument can be specified either as a normal vector like `dir.exp = c(1,1,1,1)` or between the double quotes, values being separated by commas. The latter is recommended, due to the structure of directional expectations: they can take one of the values in the causal conditions, or a dash “-” if there are no particular expectations for a specific causal condition.

Using a regular vector, the temptation is many times to write `c(1,1,-,1,1)` which returns an error, the correct form being `c(1,1,"-",1,1)`. Specifying an equivalent vector within double quotes, such as `"1,1,-,1,1"`, safeguards against this possibility.

For multi-value sets, directional expectations can take multiple values. The Lipset data has a multi-value version, where the first causal condition has three values. For an assumption that both values 1 and 2 lead to the presence of the outcome, the directional expectations would be specified as

"1;2,1,1,1,1" (note the separation of the multiple values using a semicolon) (Fig. 8.10).

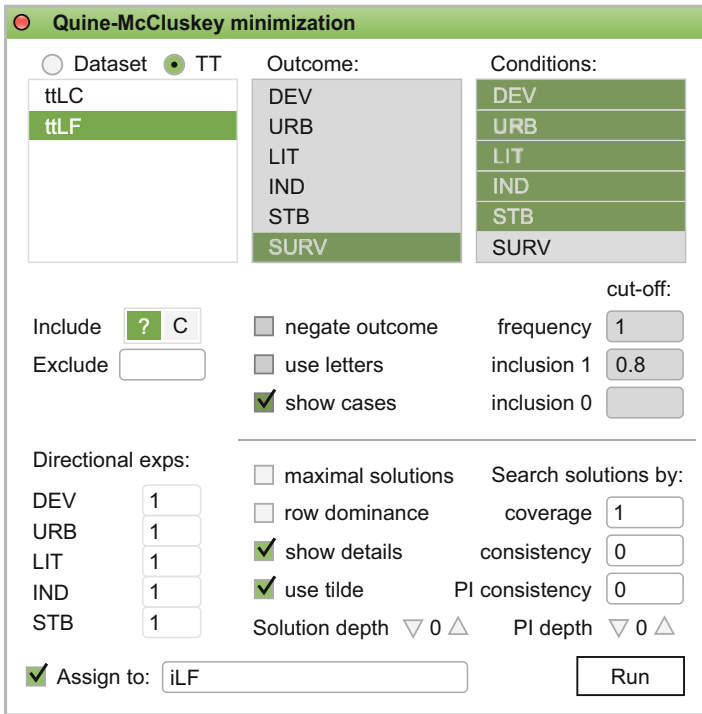


Fig. 8.10 Specifying directional expectations (lower left) in the minimization dialog

In the graphical user interface, the directional expectations become available as soon as the button ? is activated, for each causal condition selected when the truth table was produced. Back to the intermediate solution, it is:

```
iLF
```

From C1P1:

$$M1: \text{DEV} \cdot \text{URB} \cdot \text{LIT} \cdot \text{STB} + \text{DEV} \cdot \text{LIT} \cdot \text{ind} \cdot \text{STB} \Rightarrow \text{SURV}$$

This is a subset of the most parsimonious solution, and a superset of the conservative solution. The term $\text{DEV} \cdot \text{URB} \cdot \text{LIT} \cdot \text{STB}$ is a subset of $\text{URB} \cdot \text{STB}$, and $\text{DEV} \cdot \text{LIT} \cdot \text{ind} \cdot \text{STB}$ is a subset of $\text{DEV} \cdot \text{ind}$. And they are both supersets of the conservative solution:

```
minimize(ttLF)
```

$$M1: \text{DEV} \cdot \text{URB} \cdot \text{LIT} \cdot \text{IND} \cdot \text{STB} + \text{DEV} \cdot \text{urb} \cdot \text{LIT} \cdot \text{ind} \cdot \text{STB} \Rightarrow \text{SURV}$$

To print the parameters of fit the argument `details = TRUE` should be specified, or since the object has a dedicated printing function a command such as `print(iLF, details = TRUE)` is also possible.

The intermediate solution is always in the middle between the conservative and the parsimonious solutions, both in terms of complexity (it is less complex than the conservative solution, but more complex compared to the parsimonious one) and also in terms of set relations.

Ragin and Sonnett (2005) explain in detail how to derive these intermediate solutions, based on the comparison between the complex and the parsimonious solutions. Their procedure is implemented in the package *QCA* using the prime implicants' matrices from both solutions, combining them according to the directional expectations to filter those which are ultimately responsible with the intermediate solutions.

The reason for which the intermediate solution is halfway between the conservative and the parsimonious ones is due to the fact that less remainders end up being used in the minimization process, as a result of filtering out by directional expectations. Those which did contribute to producing the intermediate solution are the easy counterfactuals, the others are the difficult counterfactuals.

The resulting object from the minimization function is a list with many components. The one referring to the intermediate solutions is called `i.sol`:

```
names(iLF)
```

```
[1] "tt"           "options"      "negatives"    "initials"     "PIchart"
[6] "primes"      "solution"     "essential"    "inputcases"   "pims"
[11] "IC"          "numbers"     "SA"           "i.sol"        "call"
```

The `i.sol` component is also a list with multiple possible subcomponents:

```
names(iLF$i.sol)
```

```
[1] "C1P1"
```

In this example, it contains only one subcomponent, the pair of the (first) conservative and the (first) parsimonious solutions. Sometimes it happens to produce multiple conservative and also multiple parsimonious solutions, a situation when there will be more subcomponents: `C1P2`, `C2P1`, `C2P2` etc.

For each pair of conservative and parsimonious solutions, one or more intermediate solutions are produced within the corresponding subcomponent, which is yet another object itself containing the following sub-sub-components:

```
names(iLF$i.sol$C1P1)
```

```
[1] "EC"          "DC"           "NSEC"         "PIchart"     "c.sol"
[6] "p.sol"      "solution"     "essential"    "primes"      "IC"
[11] "pims"
```

This is where all the relevant information is located: the conservative solution `c.sol`, the parsimonious solution `p.sol`, a dedicated PI chart in the subcomponent `PIchart` as well as what we are interested most, to see which of the simplifying assumptions are easy (EC) and which are difficult (DC).

```
iLF$i.sol$C1P1$EC
```

	DEV	URB	LIT	IND	STB
30	1	1	1	0	1

```
iLF$i.sol$C1P1$DC
```

	DEV	URB	LIT	IND	STB
10	0	1	0	0	1
12	0	1	0	1	1
14	0	1	1	0	1
16	0	1	1	1	1
17	1	0	0	0	0
18	1	0	0	0	1
21	1	0	1	0	0
25	1	1	0	0	0
26	1	1	0	0	1
28	1	1	0	1	1
29	1	1	1	0	0

Out of the 12 simplifying assumptions, only one was consistent with the specified directional expectations (the easy counterfactual on row number 30), the rest of them have been filtered out, being identified as difficult.

This is as far as the Standard Analysis goes, with respect to the intermediate solutions, it is only a matter of specifying directional expectations. From here on, it is all about the Enhanced Standard Analysis which states that many of the remainders (sometimes including some of the easy counterfactuals from the intermediate solution) are incoherent, implausible, some even impossible, hence any assumption using such remainders is untenable.

Applying directional expectations performs a similar operation of filtering out some of the remainders (the difficult counterfactuals) so the process is somewhat similar. But filtering based on directional expectations is a mechanical procedure that cannot possibly identify incoherent counterfactuals.

Whatever is called “incoherent” makes sense only from a human interpreted perspective, for example it is impossible for an algorithm to detect an impossible counterfactual such as the “pregnant man”, unless it is an artificial intelligence (but we are not there just yet).

Identifying untenable assumptions is therefore a human activity, and it is a task for the researcher to tell the minimization algorithm not to use those remainders identified as untenable. Excluding this kind of remainders from the minimization process ensures a set of solutions that are not only minimally complex, but also logically and theoretically coherent.

What is about to be described next is highly important: given that some of the remainders (the untenable ones) have been removed from the minimization, it is obvious the minimal solution will not be as parsimonious as the “most parsimonious” solution including all possible remainders. It will be a bit more complex, but still less complex than the conservative one.

It is called the *enhanced parsimonious solution* (EPS) to differentiate it from the “most” parsimonious solution which is almost always useless, being derived using not only difficult counterfactuals but also untenable assumptions.

Sometimes, the enhanced parsimonious solution can be identical with the intermediate solution, a reason for which many users confuse the two. But this is a special situation, when directional expectations filter out untenable remainders as well as the difficult counterfactuals. However this is not always guaranteed, therefore the two should not be confused.

The result of the minimization process excluding the untenable assumptions is the enhanced parsimonious solution, which can subsequently be used to derive yet another intermediate solution (this time based on ESA), called the *enhanced intermediate solution* (EIS).

It goes without saying that all other remainders that are tenable are included in the minimization to obtain the EPS, otherwise it really does not matter how many remainders are excluded, the result will always be a superset of the conservative solution.

Some examples are going to be illustrative. For example, some contradictory simplifying assumptions have been identified earlier, using the function `findRows()`. Since these are incoherent counterfactuals, they should be excluded from the minimization process.

```
CSA <- findRows(obj = ttLF, type = 2)
minimize(ttLF, include = "?", exclude = CSA)
```

```
M1: DEV*URB*STB + DEV*ind*STB => SURV
```

This is the EPS, the “most” parsimonious solution that could be obtained given the available remainders after excluding the CSAs. And the argument which makes it possible is `exclude`, the third one referring to the remainders, after having introduced `include` and `dir.exp`.

This solution is also a superset of the conservative solution. In fact, all other solutions are supersets of the conservative one, or the other way round the conservative solution is a subset of all other possible solutions. But at the same time, this EPS it is a superset of all possible EISs enhanced intermediate solutions based on the remaining easy counterfactuals, after excluding the CSAs. The complexity rule is also valid, starting from the conservative solution all the way down to the EPS.

Applying directional expectations leads to the EIS, which is identical to the normal intermediate solution:

```
eiLF <- minimize(ttLF, include = "?", exclude = CSA, dir.exp = "1,1,1,1,1")
eiLF
```

From C1P1:

```
M1:    DEV*URB*LIT*STB + DEV*LIT*ind*STB => SURV
```

It is the same intermediate solution because it uses the same easy counterfactual:

```
eiLF$i.sol$C1P1$EC
```

```
      DEV URB LIT IND STB
30    1   1   1   0   1
```

The procedure is the same, regardless of the identified untenable assumptions. They are simply served to the `minimize()` function via the argument `exclude`, and the result is one EPS or another, depending which untenable assumptions have been excluded.

For example, some other types of untenable assumptions are related to the negated necessary conditions. At the analysis of necessity in Chap. 5, the following necessary conditions have been identified:

```
superSubset(LF, outcome = "SURV", incl.cut = 0.9, ron.cut = 0.6)
```

		inclN	RoN	covN
1	STB	0.920	0.680	0.707
2	LIT*STB	0.915	0.800	0.793
3	DEV+URB+IND	0.903	0.704	0.716

This replicates the example presented by Schneider and Wagemann (2013), just using different column names. They have identified LIT (literacy) and STB (government stability) as necessary conditions. Here, only STB is necessary because LIT has a lower relevance, but it does not matter because the conjunction LIT·STB is necessary and that implies by default the atomic conditions are necessary: if the outcome is a subset of an intersection, it surely is a subset of the entire set.

What Schneider and Wagemann (2013) argue has been covered in the previous section, graphically illustrated in Fig. 8.9: when a condition is identified as necessary for the outcome, it is logically impossible for its negation to be sufficient for the outcome. In this case, the conjunction LIT·STB is necessary and its negation is \sim LIT + \sim STB, which means that any remainder containing the negation of LIT or the negation of STB is incoherent.

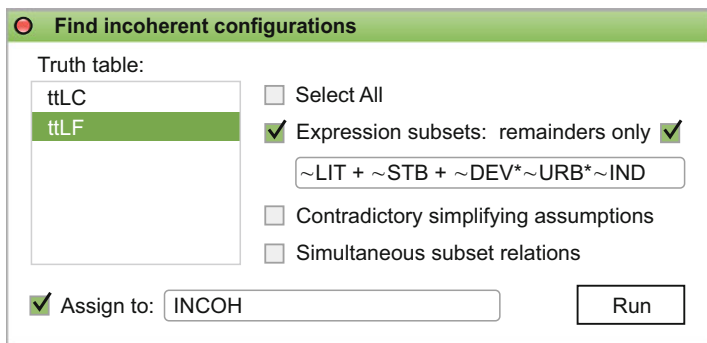


Fig. 8.11 The GUI dialog to find incoherent configurations

All these incoherent counterfactuals can be identified using the function `findRows()`, which has a dedicated dialog in the graphical user interface (in Fig. 8.11 above), that is opened by selecting the menu:

Analyse/Incoherent Configurations

It uses the default `type = 1` to identify the expression subset remainders:

```
INCOH <- findRows("~LIT + ~STB + ~DEV*~URB*~IND", ttLF)
minimize(ttLF, include = "?", exclude = INCOH)
```

M1: $URB * LIT * STB + DEV * LIT * ind * STB \Rightarrow SURV$

This is the same solution presented by Schneider and Wagemann, and irrespective of what further directional expectations would be defined, the enhanced intermediate solution is the same because the incoherent counterfactuals have already been excluded from the minimization.

These two examples demonstrate how to use the argument `exclude` in order to perform the ESA—Enhanced Standard Analysis. How exactly this is made possible, will be presented in the next section but it is hopefully clear by now that ESA is possible by excluding any number of untenable assumptions via the argument `exclude`: all truth table rows that do not seem logical, coherent or tenable can be excluded to obtain a parsimonious or an intermediate solution free of such assumptions.

In the graphical user interface, specifying the exclusion vector is trivially done by typing the name of the object containing the row numbers (Fig. 8.12).

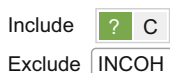


Fig. 8.12 Specifying the excluded configurations in the minimization dialog

One final type of incoherent configurations that is worth presenting refers to the simultaneous subset relations. Just like in the case of remainders, the function `findRows()` can be used to identify this category of rows in the truth table. This time, however, it is not about the remainders but about the empirically observed configurations.

Sometimes, it may happen that an such an observed configuration to have a consistency score above the threshold for both the presence and the absence of the outcome. Such a configuration is simultaneously a subset of both the presence and of the absence of the outcome, this is quite possible in fuzzy sets.

```
findRows(obj = ttLF, type = 3)
```

```
numeric(0)
```

For this particular dataset, there are no such simultaneous subset relations (of `type = 3` in the command above). Similar to the untenable assumptions, where some remainders are excluded from the minimization, the same is possible about observed configurations running the minimization process excluding the simultaneous subset relations. The function `minimize()` doesn't care if it is an observed configuration or a remainder, everything supplied via the argument `exclude` is equally excluded.

Before ending this section, the argument `type` has an additional possible value equal to `0`, that means finding all types of incoherent or untenable configurations, provided an expression for type 1:

```
ALL <- findRows("~LIT + ~STB", ttLF, type = 0)
minimize(ttLF, include = "?", exclude = ALL)
```

```
M1: DEV*URB*LIT*STB + DEV*LIT*ind*STB => SURV
```

The solution above is the enhanced *parsimonious* solution excluding all those untenable configurations. Interestingly, it is the same as the enhanced *intermediate* solution after excluding the contradictory simplifying assumptions and using directional expectations, so one way or another the solutions seem to converge.

Any other untenable assumptions that researchers might find via other methods not part of the function `findRows()` can be added to the numeric vector supplied via the argument `exclude`, using the usual function `c()`. The final enhanced parsimonious and intermediate solutions are then going to be derived using what is left from the remainders.

8.8 Theory Evaluation

Theory evaluation is vaguely similar to what the quantitative research calls *hypothesis testing*: having empirical data available, it is possible to test a certain hypothesized relation between (say) an independent and a dependent variable, or more often between a control and an experimental group.

QCA is also focused on the empirical part, although much less on the actual (raw) data but rather on the solutions obtained via the minimization process. Using Boolean logic, Ragin (1987, p. 118) showed how to create an intersection between a minimization solution and a theoretical statement (expectation) about how an outcome is produced, and Schneider and Wagemann (2012) extended this approach using consistency and coverage scores.

In the *QCA* package, there are two main functions which can be used to create such intersections, both being able to natively involve various negations: the first function is called `intersection()`, that works exclusively on the sum of products type of expressions (including those from the minimization objects) and the second one is called `modelFit()` which relies on a heavier use of the minimization objects to also calculate parameters of fit.

It is perhaps important to mark the difference and not confuse the function `intersection()` with the base R function called `intersect()` which performs set intersection but has nothing to do with QCA.

As a first example, it is possible to re-use some of the objects created in the previous section, but to make these examples self-contained they are going to be re-created here:

```
data(LF)
ttLF <- truthTable(LF, outcome = "SURV", incl.cut = 0.8)
iLF <- minimize(ttLF, include = "?", dir.exp = "1,1,1,1")
iLF
```

From C1P1:

```
M1:    DEV*URB*LIT*STB + DEV*LIT*ind*STB => SURV
```

Now, suppose that we have a strong theoretical expectation that democracy survives where a country is both developed and has a stable government. Such a hypothesis can be written in the expression: DEV·STB.

Using the function `intersection()`, we can see how is this expectation covered by the empirical minimization solution:

```
intersection(iLF, "DEV*STB")
```

```
E1-C1P1-1: (DEV*URB*LIT*STB + DEV*LIT*ind*STB)*DEV*STB
  I1-C1P1-1: DEV*URB*LIT*STB + DEV*LIT*ind*STB
```

In this intersection, it could be argued the intermediate solution perfectly overlaps our theory, since the conjunction DEV·STB is found in both solution terms from the model.

In a similar fashion, and also very interesting is to see how the empirical solutions overlap with the negation of the theory, or even how both negations intersect:

```
intersection(negate(iLF), negate("DEV*STB"))
```

```
E1: (dev + lit + stb + urb*IND)(dev + stb)
I1: dev + stb
```

Note how both arguments that form the input for the function `intersection()` can be natively and intuitively negated, the function `negate()` detecting automatically when the input is a character expression or a minimization object.

The intersection of both negated inputs seems like an even more perfect match with the (negated) theoretical expectation. And things can be made even better, not only by automatically calculating all possible intersections between expressions and/or their negations, but also including parameters of fit:

```
modelFit(model = iLF, theory = "DEV*STB")
```

```
M-C1P1
MODEL:      DEV*URB*LIT*STB + DEV*LIT*ind*STB
THEORY:     DEV*STB
MODEL*THEORY: DEV*URB*LIT*STB + DEV*LIT*ind*STB
MODEL*theory: -
model*THEORY: DEV*lit*STB + DEV*urb*IND*STB
model*theory: dev + stb
```

	incLS	PRI	covS
1 DEV*URB*LIT*STB	0.901	0.879	0.468
2 DEV*LIT*ind*STB	0.814	0.721	0.282
3 MODEL	0.866	0.839	0.660
4 THEORY	0.869	0.848	0.824
5 MODEL*THEORY	0.866	0.839	0.660
6 MODEL*theory	-	-	-
7 model*THEORY	0.713	0.634	0.242
8 model*theory	0.253	0.091	0.295

Just as in the case of QCA solutions, where multiple sufficient paths can be found (each having potentially multiple disjunctive solution terms), there can be alternative theories about a certain outcome. All of these theories can be formulated disjunctively using the + operator, for example if another theory about democracy survival states that industrialization alone ensures a survival of democracy, these two theories can be formulated as DEV·STB + IND.

```
modelFit(iLF, "DEV*STB + IND")
```

```
M-C1P1
```

```
MODEL:      DEV*URB*LIT*STB + DEV*LIT*ind*STB
```

```
THEORY:     DEV*STB + IND
```

```
MODEL*THEORY: DEV*URB*LIT*STB + DEV*LIT*ind*STB
```

```
MODEL*theory: -
```

```
model*THEORY: dev*IND + lit*IND + IND*stb + urb*IND + DEV*lit*STB
```

```
model*theory: dev*ind + ind*stb
```

		inclS	PRI	covS
1	DEV*URB*LIT*STB	0.901	0.879	0.468
2	DEV*LIT*ind*STB	0.814	0.721	0.282
3	MODEL	0.866	0.839	0.660
4	THEORY	0.733	0.698	0.871
5	MODEL*THEORY	0.866	0.839	0.660
6	MODEL*theory	-	-	-
7	model*THEORY	0.533	0.438	0.302
8	model*theory	0.272	0.070	0.251

The output of the `modelFit()` function is a list, having as many components as the number of models in the minimization object. An artificial example to generate two models could be:

```
iLF2 <- minimize(ttLF, include = "?", dir.exp = "1,0,0,0,0")
iLF2
```

```
From C1P1:
```

```
M1:      DEV*URB*STB + (DEV*urb*ind) => SURV
```

```
M2:      DEV*URB*STB + (DEV*ind*STB) => SURV
```

For each minimization model (solution), a model fit is generated:

```
mflF2 <- modelFit(iLF2, "DEV*STB")
length(mflF2)
```

```
[1] 2
```

Any of the contained solutions, for instance the second, can be accessed very easily using the regular `[]` operator to index lists, such as:

```
mflF2[[2]]
```

Chapter 9

Pseudo-Counterfactual Analysis



The word “assumption” has been used and overused, across all these sections referring to the conservative, parsimonious and intermediate solutions of all kinds. The time is ripe to discuss yet another assumption, perhaps the biggest myth of all in terms of scope.

The information presented insofar, especially in the previous section to filter remainders and produce intermediate solutions, is based on the implicit assumption that once the remainders are established, then classical Quine-McCluskey algorithm steps in to find the prime implicants.

Let us discuss again the how the simplifying assumptions are understood: a subset of the remainders which are actually used in the minimization process, those that have an active role in producing the prime implicants. It is in line with the philosophy of the QMC algorithm, that compares all possible pairs of configurations (observed with a positive outcome plus the included remainders), to identify those which differ through a single literal, and iteratively minimize them until nothing can be minimized further.

Except for a very small number of developers, the vast majority of the QCA users don't have a very clear idea of how the programming of the QMC algorithm really works, it is just a black box that outputs the expected results.

On the other hand, researchers do expect that remainders are actually *used* in the minimization process. Much to everyone's surprise, this is not how prime implicants are produced in the current version of package *QCA*. The final solutions are exactly the same, but they have almost nothing to do with the remainders, be them good, tenable, untenable, easy, difficult, incoherent, implausible and anything else.

It is easy to imagine this is an almost perplexing statement, that raises a potential mistrust in the package (after all, it does not follow the classical, exact algorithm put forward by Quine and McCluskey), or at the very least raises two immediate questions:

1. How does the package **QCA** find the correct solutions, if no remainders are actually used?
2. If the solutions are correct (and they are) what is then left of the entire theoretical corpus around the remainders? Are all of these publications sideways?

The short answer is: no. All publications around the remainders are valid, it is only the procedure to derive all categories of remainders that is different. While the current methodology focuses on what remainders are *included* in the minimization (following the classical QMC algorithm), quite the opposite the package **QCA** concentrates on the configurations that are *excluded* from the minimization process.

The end result is exactly the same, much like a simple addition equation like $P + N + R = S$, where parameters P and N are always known (the positive and the negative output configurations). If $P = 4$ and $N = 5$, the equation becomes $4 + 5 + R = S$. By filtering the remainders, suppose three of them are identified so $R = 3$, then S can be calculated as 12. But the reverse is also true: knowing the end result $S = 12$, we can derive R to be equal to 3.

This highly simplified example is only a conceptualization, of course, but it is a very accurate description of what is happening in package **QCA**: it can calculate the end result S based on the first two parameters P and N , and reverse engineer the calculation of parameter R by using the result S .

From an algebraic point of view this seems like a magician activity pulling a rabbit out of an empty hat, for the end result involving an unknown quantity should be unknown. But using some very precise regularities of the minimization process, this is already proven to be possible. Contrary to the common perception, the remainders are never actually *used* in the minimization, they are only derived after the solutions have been generated.

9.1 eQMC

How exactly this is possible was first described by Duşa (2007), and later formalized by Duşa and Thiem (2015). The idea is very simple and has been implemented in package **QCA** since version 0.6-5 back in 2007, after having made these two absolute basic observations about the prime implicants:

1. they are always supersets of the initial positive output configurations
2. they are never supersets of the negative output configurations

The quest was to find all supersets of the positive output configurations, that at the same time were not supersets of the negative output configurations, and it was a simple matter of applying three functions:

- `findPrimes()`, later renamed to `findSupersets()`
- a base function `setdiff()` to obtain the set difference with all elements in the first set that were not present in the second, and
- a function to remove the redundant implicants, supersets that had supersets of their own (meaning they were implicants but not *prime* implicants).

Finding all supersets of a given set of positive output configurations builds on the previous findings from Duşa (2010), where two other helpful ideas have been introduced:

- using the implicant matrix, which for binary crisp sets it is the same as the so-called 3^k matrix where k is the number of causal conditions)
- instead of searching through all columns from the implicant matrix, it is more simple to just use its row numbers so the information from an entire matrix can be compressed in a single vector.

All these information might seem too abstract and unrelated to the Boolean minimization process, but a simple example will reveal the underlying idea.

A hypothetical dataset (HD) will be used for both the classical QMC algorithm and this new procedure. First, suppose we have empirical evidence for all possible configurations of causal conditions (no limited diversity), amounting to a (c)ompletely specified truth table:

```
HD <- cbind(createMatrix(rep(2, 4)), rep(c(1, 0), c(12, 4)))
colnames(HD) <- c(LETTERS[1:4], "Y")
ttHdc <- truthTable(HD, outcome = "Y")
ttHdc
```

OUT: output value
 n: number of cases in configuration
 incl: sufficiency inclusion score
 PRI: proportional reduction in inconsistency

	A	B	C	D	OUT	n	incl	PRI
1	0	0	0	0	1	1	1.000	1.000
2	0	0	0	1	1	1	1.000	1.000
3	0	0	1	0	1	1	1.000	1.000
4	0	0	1	1	1	1	1.000	1.000
5	0	1	0	0	1	1	1.000	1.000
6	0	1	0	1	1	1	1.000	1.000
7	0	1	1	0	1	1	1.000	1.000
8	0	1	1	1	1	1	1.000	1.000
9	1	0	0	0	1	1	1.000	1.000
10	1	0	0	1	1	1	1.000	1.000
11	1	0	1	0	1	1	1.000	1.000
12	1	0	1	1	1	1	1.000	1.000
13	1	1	0	0	0	1	0.000	0.000
14	1	1	0	1	0	1	0.000	0.000
15	1	1	1	0	0	1	0.000	0.000
16	1	1	1	1	0	1	0.000	0.000

This is not only a hypothetical truth table but also an unrealistic one with 12 out of the 16 possible configurations having a positive output and the rest of 4 have a negative output. It is presented here for demonstration purposes only but it nevertheless has a lot of potential to show how the QMC algorithm really works.

The classical QMC minimization generates two prime implicants $\sim A$ and $\sim B$:

```
pHD <- minimize(ttHDc, use.tilde = TRUE, method = "QMC")
rownames(pHD$PIchart)
```

```
[1] "~A" "~B"
```

In a real situation, it would be close to impossible to find empirical evidence for all 16 possible configurations, and the truth table would display the observed positive and negative configurations as well as the remainders which have a question mark “?” in the OUTput column. Completely specifying the truth table just simulates the process of including the remainders.

But suppose only the first four configurations have a positive output, leading to an (i)ncompletely specified and more realistic truth table:

```
ttHdi <- truthTable(HD[-c(5:12), ], outcome = "Y")
print(ttHdi, complete = TRUE)
```

```
OUT: output value
n: number of cases in configuration
incl: sufficiency inclusion score
PRI: proportional reduction in inconsistency
```

	A	B	C	D	OUT	n	incl	PRI
1	0	0	0	0	1	1	1.000	1.000
2	0	0	0	1	1	1	1.000	1.000
3	0	0	1	0	1	1	1.000	1.000
4	0	0	1	1	1	1	1.000	1.000
5	0	1	0	0	?	0	-	-
6	0	1	0	1	?	0	-	-
7	0	1	1	0	?	0	-	-
8	0	1	1	1	?	0	-	-
9	1	0	0	0	?	0	-	-
10	1	0	0	1	?	0	-	-
11	1	0	1	0	?	0	-	-
12	1	0	1	1	?	0	-	-
13	1	1	0	0	0	1	0.000	0.000
14	1	1	0	1	0	1	0.000	0.000
15	1	1	1	0	0	1	0.000	0.000
16	1	1	1	1	0	1	0.000	0.000

Minimizing this truth table (this time including the remainders) leads to the same two prime implicants $\sim A$ and $\sim B$:

```
pHDI <- minimize(ttHDI, include = "?", use.tilde = TRUE, method = "QMC")
rownames(pHDI$PIchart)
```

```
[1] "~A" "~B"
```

The same prime implicants are obtained with the eQMC method, using the `findSupersets()` function applied to the first four rows and the first four columns from the HD data (same as the first four configurations from the truth table) to yield the following superset row numbers from the 3^k matrix:

```
posimp <- findSupersets(HD[1:4, 1:4] + 1, noflevels = rep(3, 4))
posimp
```

```
[1] 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 28 29 30 31 32
[23] 33 34 35 36 37 38 39 40 41 42 43 44 45
```

There are of course more than 16 rows from the truth table, because in this example the implicants matrix has $3^4 = 81$ rows. The 3^k matrix is signaled by using the vector `rep(3, 4)`, where the number 3 literally implies 3^4 , and the configurations from `ttHDI` are raised in the implicants matrix by adding 1 to their values and use the value 2 for the presence and the value 1 for the absence of a causal condition.

The function `findSupersets()` is next applied to the last four rows of the HD matrix, to find all supersets of the negative output configurations:

```
negimp <- findSupersets(HD[13:16, 1:4] + 1, noflevels = rep(3, 4))
negimp
```

```
[1] 2 3 4 5 6 7 8 9 19 20 21 22 23 24 25 26 27 55 56 57 58 59
[23] 60 61 62 63 73 74 75 76 77 78 79 80 81
```

The next step is to find all supersets from the positive output configurations, that are *not* supersets of the negative output configurations:

```
posimp <- setdiff(posimp, negimp)
posimp
```

```
[1] 10 11 12 13 14 15 16 17 18 28 29 30 31 32 33 34 35 36 37 38 39 40
[23] 41 42 43 44 45
```

These are all supersets of the positive output configurations, but most of them are redundant because there are bigger supersets in this vector. The final step of this procedure is to remove all redundant supersets, to obtain the minimal non-redundant prime implicants. It involves two functions called `removeRedundants()` and `getRow()` which are internal, not designed for the user level (therefore undocumented) but it is easy to see what they do:

```
primeimp <- removeRedundants(posimp, noflevels = rep(3, 4))
primeimp
```

```
[1] 10 28
```

These are the row numbers in the 3^k matrix that are the equivalent of the $\sim B$ and $\sim A$ prime implicants, recalling that a value of 0 signals a minimized condition, a value of 1 signals the absence of a condition and a value of 2 signals the presence of a condition:

```
getRow(primeimp, noflevels = rep(3, 4))
```

```
      [,1] [,2] [,3] [,4]
[1,]    0    1    0    0
[2,]    1    0    0    0
```

The entire procedure is encapsulated in the eQMC minimization method:

```
epHDi <- minimize(ttHDi, include = "?", use.tilde = TRUE, method = "eQMC")
rownames(epHDi$PIchart)
```

```
[1] "~A" "~B"
```

The resulting prime implicants are exactly the same, despite using a very different procedure which brought a great deal of speed compared with the previous one, consuming a lot less memory (for 15 causal conditions, about 15 MB compared to 1.4 GB for the classical QMC). It also made it possible to explore up to 18 conditions at once, therefore additional three conditions meaning $3^3 = 27$ times more complex situations than before.

Past that threshold, the eQMC algorithm reaches its maximum possibilities. The culprit is less represented by memory consumption, although at an extremely high number of causal conditions the generated vectors of row numbers for the supersets can potentially grow larger than available memory. Possibly problematic, after 19 causal conditions the row numbers cannot even be represented in 32 bit computers, which have an upper representation limit of 2^{31} , that is still higher than 3^{19} but not enough compared to 3^{20} for the implicant matrix with 20 causal conditions.

The more serious issue is the calculation time, especially at the final step to remove the redundant implicants: for millions or tens of millions of implicants, even this “quick” process can take a lot more time than expected.

As shown, the input of the classical QMC procedure consists of 12 positive output configurations, either having a completely specified truth table or including the 8 remainders that are treated the same as the empirically observed positive output configurations. In the eQMC procedure, sifting through the supersets of the 4 positive and 4 negative output configurations generates the very same prime implicants, without even touching the “included” remainders.

One might argue this is not entirely accurate, because the remainders are still somehow involved in this alternative procedure, among the redundant superset implicants. This would be a valid counter argument, if not for a third minimization method called “CCubes”, that finds the prime implicants not by searching through supersets but using consistency cubes.

9.2 Consistency Cubes

The hypothetical dataset in the previous section is highly straightforward, both $\sim A$ and $\sim B$ showing perfect consistency with the presence of the outcome. For the next procedure, suppose we have the following simulated truth table with three positive and three negative output configurations:

```
tt <- matrix(c(1,0,0,0,1,1, 0,0,1,0,0,1,
              1,0,0,1,1,1, 0,1,1,0,1,1,
              1,1,1,0,0,0), nrow = 6)
dimnames(tt) <- list(c(11, 2, 6, 3, 12, 16), c("A", "B", "C", "D", "OUT"))
tt
```

	A	B	C	D	OUT
11	1	0	1	0	1
2	0	0	0	1	1
6	0	1	0	1	1
3	0	0	1	0	0
12	1	0	1	1	0
16	1	1	1	1	0

Similar to the eQMC method, the input for this procedure does not involve any remainders but only the positive and the negative output configurations. The following definition is necessary to properly describe how consistency cubes work:

Definition 9.1. A prime implicant is the simplest possible, non-redundant, fully consistent superset of any positive output configuration.

In this definition, fully consistent means it is not a superset of an observed negative output configuration. The simplest possible superset is a single condition (either its presence, or its absence). Figure 9.1 below is a graphical representation of the truth table tt , from which it can be observed that none of the conditions is fully consistent in their presence. There is only one condition (C) that is fully consistent in its *absence* but it does not cover all observed positive configurations and search is continued at the next level of complexity.

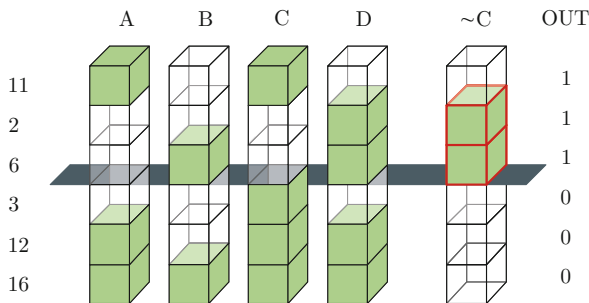


Fig. 9.1 Individual presence inconsistencies, absence of C consistent

In the next level of complexity, superset expressions are searched in all combinations of 2 conditions out of 4, and within each combination in all possible pairs of their respective levels. In none are found at this level, search is continued in all combinations of 3 conditions out of 4, until everything is exhausted.

Since binary crisp conditions are just a subset of the general multi-value conditions, a complete search space S containing all these possibilities can be calculated as the sum of all combinations of c selected conditions out of k , times the product of their respective number of levels l_s for each such selected combination:

$$S_{MV} = \sum_{c=1}^k \binom{k}{c} \prod_{s=1}^c l_s \quad (9.1)$$

The search space for the binary crisp sets (CS) involve just two levels (0 and 1) for each condition, and the equation above results in exactly $3^k - 1$ possible expressions, describing what Ragin (2000, pp. 132–136) calls “groupings”. These are all possible configurations from the 3^k implicant matrix specific for binary crisp sets:

$$S_{CS} = \sum_{c=1}^k \binom{k}{c} \prod_{s=1}^c 2 = \sum_{c=1}^k \binom{k}{c} 2^c = 3^k - 1 \quad (9.2)$$

Using a similar technique, Ragin lists all possible expressions combining presence and absence of causal conditions in Table 5.4 (p. 134). His search technique is very similar to this bottom-up approach (starting from the simplest expressions), using what he calls the “containment rule” to eliminate more complex, redundant expressions.

The efficiency of the bottom-up approach (from simplest to the most complex expressions) can be measured by the performance of the CCubes algorithm, that combines the properties of Eq. (9.1) with the requirements of Definition 9.1. These are all key ingredients that make the CCubes algorithm very fast.

At the next level of complexity $c = 2$, Fig. 9.2 displays all possible combinations of presence and absence for the conditions A and D, out of which two pairs are consistent: $\sim A \cdot D$ and $A \cdot \sim D$ are simultaneously associated with an observed positive output (above the separator), while at the same time not associated with an observed negative output (below the separator).

Both sides of the cube have to be lit below the separator in order to prove inconsistency (such as those from fourth implicant $A \cdot D$), otherwise in the absence of complete negative cubes they are consistent with the positive ones.

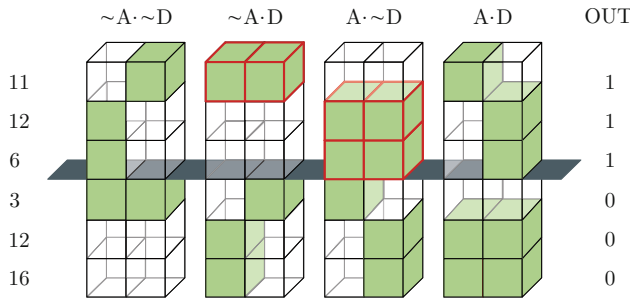


Fig. 9.2 Conjunctive prime implicants

9.2.1 Search Space

Exploring the entire search space is a polynomial, time consuming operation. To make sure the results are exhaustive, all possible combinations of conditions must be verified, along with all their combinations of levels. There are 4 pairs of such combinations of levels in Fig. 9.2, but there are 6 combinations of 4 conditions with 4 pairs of levels each, consuming 48 computer cycles (six times four times two comparisons).

To solve this exponential computational problem, the search space is partitioned into layers of complexity, starting with the least complex ($c = 1$) to a maximum value of k causal conditions. A second strategy to eliminate redundant search becomes evident by noticing that not all combinations of levels should be checked, as the relevant ones are already found in the truth table configurations, thus solving yet another usual problem (memory consumption) in such polynomial search strategies.

In Fig. 9.2 there are four combinations for the levels (presence and absence) of the two conditions A and D, but only two of them are relevant and they are already found in the observed configurations (the other two combinations of levels will certainly prove to be void, as they are not observed).

```
tt[, c("A", "D", "OUT")]
```

	A	D	OUT
11	1	0	1
2	0	1	1
6	0	1	1
3	0	0	0
12	1	1	0
16	1	1	0

By eliminating certain void search, the search space is shortened to only 6 rows times two columns (12 computer cycles instead of 48). The identified prime implicants are the conjunctions $A \cdot \sim D$ is also the same as the binary

combination “10” on the first line, truth table row number 11), and the conjunction $\sim A \cdot D$, same as the binary combination “01” on the next two lines, truth table row numbers 2 and 6), as they are found in the observed positive output configurations and not in the negative output ones.

A third strategy to shorten the search space is made possible by using the mathematical regularities from the previous two implementations, in this example transforming the binary representations of the truth table configurations in their decimal equivalent:

```
data.frame(AD = tt[, c("A", "D")] %**% 2:1, OUT = tt[, "OUT"])
```

	AD	OUT
11	2	1
2	1	1
6	1	1
3	0	0
12	3	0
16	3	0

While the exhaustive approach consumed 48 cycles, the same results can be obtained by searching through only these 6 decimal numbers: 2, 1, 1, 0, 3, 3 (the first three corresponding to the observed positive configurations where $OUT = 1$). Since they are not found in the negative truth table configurations (numbers 0, 3, 3), both are identified as prime implicants.

This bottom up approach is superior to the previous QMC and eQMC methods in terms of efficiency, and generates the same prime implicants:

```
ppt <- minimize(truthTable(tt, outcome = "OUT"),
               include = "?", use.tilde = TRUE, method = "CCubes")
ppt$PIchart
```

	2	6	11
$\sim C$	x	x	-
$\sim A \cdot B$	-	x	-
$\sim A \cdot D$	x	x	-
$A \cdot \sim D$	-	-	x

All prime implicants from Figs. 9.1 and 9.2 are found: $\sim C$, $\sim A \cdot D$ and $A \cdot \sim D$. The fourth one $\sim A \cdot B$ is actually redundant because it does not contribute to solving the prime implicants chart (explaining all positive output configurations).

This is the search strategy corresponding to the parsimonious solution, and contrary to the common (mis)perception, the pseudo-counterfactual analysis does not actually *use* any remainders to generate prime implicants and yet the final solutions are the same as those from the classical QMC algorithm. It may not seem highly relevant at this point, but it will prove important in the next chapter.

9.3 Include vs. Exclude

Both Standard Analysis (SA) and the Enhanced Standard Analysis (ESA) insist on which counterfactuals to be *included* in the minimization process, together with the observed positive output configurations. By default, all remainders are included, unless specifically filtering out some of them: incoherent counterfactuals, untenable assumptions, contradictory simplifying assumptions etc.

The usual conceptualization of the minimization process follows the classical QMC procedure with a set of configurations composed from the positive output ones plus what is left from the remainders after filtering out. In the classical procedure, what is left out (the negative output configurations and the filtered counterfactuals) do not play any role whatsoever in the minimization process. Only what is preserved contribute to a minimal solution, where the word “minimal” should be interpreted function of how many remainders are left after filtering out some of them.

While QMC ignores what is left out, the current default algorithm in package *QCA* does exactly the opposite: it ignores the “included” remainders and instead concentrates on the contrast between the positive output configurations (on one hand) and the set composed from the negative output configurations plus what is filtered out from the remainders (on the other).

In the classical QMC algorithm, the remainders which are not filtered out and survived this process are treated as if they had a positive output value, based on the assumption that should these remainders ever be observed, they would contribute to producing the phenomenon of interest. And this is a very intuitive conceptual image of the minimization process, taking out of this picture the remainders which are filtered out.

But the same results are obtained by ignoring the surviving remainders, and instead concentrating on those which are filtered out. In this alternative process a different assumption is made, that should these (filtered out) remainders ever be observed, they would not lead to producing the phenomenon of interest.

The counterfactual analysis assumes the surviving remainders would produce the output, whereas the pseudo-counterfactual analysis assumes the filtered out remainders would *not* produce the output, should they be empirically observed. These are, of course, the two sides of the same coin: QMC focuses on the *included* remainders, while eQMC and the faster CCubes concentrate on the *excluded* remainders, with the same end result.

The reason for this alternative approach is speed and memory. The classical QMC approach reaches its limits at about 11–12 input causal conditions (consuming a lot of memory in the process), while CCubes can easily deal with up to 30 causal conditions with no additional memory needs.

Building on the previous example with the simulated truth table, there are 6 observed configurations having the row numbers: 11, 2, 6, 3, 12, and 16. Since there are four binary crisp conditions, the remainders are found on row numbers:

```
setdiff(1:16, rownames(tt))
```

```
[1] 1 4 5 7 8 9 10 13 14 15
```

Suppose the remainders on rows numbers 7 and below should be excluded from the minimization process, which means the classical QMC algorithm would include the rest of the surviving remainders 8, 9, 10, 13, 14 and 15, which are assumed to take a positive output value of 1.

```
toinclude <- c(8, 9, 10, 13, 14, 15)
tt2 <- rbind(tt, cbind(getRow(toinclude, noflevels = rep(2, 4)), 1))
rownames(tt2) <- c(rownames(tt), toinclude)
```

Using a cascading combination of functions `rbind()`, `cbind()` and `getRow()`, the code above simulates the data becoming the input for the classical QMC algorithm. It is very important to understand this is a simulated truth table, because in a normal one the column OUT would be coded with a question mark sign "?" for the included remainders, not a value of 1 as here.

```
minimize(truthTable(tt2, outcome = "OUT"),
         use.tilde = TRUE, method = "QMC")$PIchart
```

	2	6	8	9	10	11	13	14	15
A*~C	-	-	-	x	x	-	x	x	-
A*~D	-	-	-	x	-	x	x	-	x
~C*D	x	x	-	-	x	-	-	x	-
~A*B*D	-	x	x	-	-	-	-	-	-

The net effect of hardcoding the output value with 1 (instead of "?") for the remainders is the generation of a larger PI chart, where the correct one would only display columns "2", "6" and "11" (the observed positive configurations). It can be seen that only the prime implicants $A \cdot \sim D$ and $\sim C \cdot D$ cover these three columns, and the other two PI lines do not count in the solution.

Manual hardcoding of the output column is not a very good idea, this is the morale that captures the essence of this example.

The correct PI chart is obtained by using the initial truth table `tt`, specifically excluding the hypothetical untenable remainders 1, 4, 5, 7:

```
minimize(truthTable(tt, outcome = "OUT"), exclude = c(1, 4, 5, 7),
         include = "?", use.tilde = TRUE, method = "CCubes")$PIchart
```

	2	6	11
A*~D	-	-	x
~C*D	x	x	-
~A*B*D	-	x	-

Chapter 10

QCA Extensions



10.1 Temporal QCA

Ever since the QCA method has been developed, it has been a promising methodological candidate to study complex phenomena. The empirical data is subjected to a theoretical model which consists of a series of causal conditions and one outcome, which could partially explain why it is sometimes confused with a regression model.

While regressions share a similar, simple diagram (independent variables having a direct effect over the dependent variable), in reality the phenomena being studied have a much more complex structure. Some causal factors naturally happen before others, thus determining (directly or indirectly) their structure and their possible effects on the final outcome.

The temporality aspect of QCA has been extensively discussed before (De Meur et al. 2009; Schneider and Wagemann 2012; Hak et al. 2013; Marx et al. 2014), but few of the identified solutions have been converted into concrete research instruments to be combined with the current QCA software. While many of these solutions are tempting to discuss, this section will cover only a smaller subset which can be used in conjunction with R and wherever possible with the package *QCA*.

Caren and Panofsky (2005) wanted to tackle this important aspect of the social research and propose a new methodology called TQCA (an extension of the main QCA with a temporal dimension). As they eloquently put it:

Variables and cases, as Ragin deploys them in QCA, are frozen in time—they are treated neither as containing sequences of events nor as forces that cause changes to occur over time.

Much like in the clarifying response from Ragin and Strand (2008), I have the same opinion that such an attention to the dimension of time is highly

welcomed in the methodological field. Since these two papers have appeared, the concept of temporality has been a recurrent theme in the literature, in close connection with causality which is going to be treated in more depth in the next section.

Although Caren and Panofsky proposed an extension of the QCA methodology, actually modifying the minimization algorithm to incorporate this dimension, Ragin and Strand showed that it needs no special modification because it was always possible to do that with the classical QCA. The solution was pointed right back to the original book by Ragin (1987, p. 162):

Note that it is possible to include causal variables relevant to historical process in a truth table (such as “class mobilization preceded ethnic mobilization,” true or false?) and to analyze combinations of such dichotomies. This strategy would enhance the usefulness of Boolean techniques as aids to comparative historical interpretation.

The solution is therefore as simple as using just one additional column in the data (called EBA) to account for the cases where the causal condition E (support for elite allies) happens or not before the condition A (national union affiliation). In this version of the data, conditions P and S are placed first and last but don’t carry a temporal information.

```
data(RS)
RS
```

	P	E	A	EBA	S	REC
1	1	1	1	1	1	1
2	1	1	1	1	1	1
3	1	1	1	0	1	1
4	1	1	1	1	0	1
5	1	1	1	1	0	1
6	1	1	0	-	1	1
7	1	0	1	-	1	0
8	1	0	1	-	1	0
9	1	0	0	-	1	0
10	1	0	0	-	1	0
11	1	0	0	-	1	0
12	1	0	0	-	0	0
13	0	1	1	0	1	1
14	0	1	0	-	0	0
15	0	0	0	-	0	0
16	0	0	0	-	0	0
17	0	0	0	-	0	0

The dash is a special sign, departing from the regular binary values 0 and 1, but it is recognized as such by package *QCA*. It is there to account for the cases where it is impossible to assess if the condition E really happens before the condition A, because one of them or even both are not happening.

For instance, the first case with a dash at condition EBA is on line 6, where condition E is present (value 1) but condition A is absent (value 0, it does not happen). It would be logically impossible to say that E happens before A happens, given that A does not happen at all.

Even more obvious, on the last case where both E and A are absent, it would not make sense to say that E happens before A, just as it would be illogical to say that E doesn't happen before A doesn't happen. To gain any logical interpretation, the expression E before A needs both E and A to actually happen.

With this kind of data, it is possible to perform a regular Boolean minimization and obtain the same results as those initially advanced by Caren and Panofsky, and further simplified by Ragin and Strand:

```
minimize(truthTable(RS, outcome = "REC"))
```

M1: P*E*S + P*E*A*EBA + E*A*eba*S <=> REC

This result faithfully replicates those presented by Ragin and Strand, and expression such as P*E*A*EBA indicating a causal order, with both E and A as part of this prime implicant and the simultaneous information EBA, that is E happens before A happens. The third expression E*A*eba*S is also sufficient for the outcome REC (union recognition), but in this case the condition E does *not* happen before condition A.

As simple as it may seem, this approach suffers from the main disadvantage that it only works with a conservative solution type. It would be impossible to select any meaningful counterfactuals, not only configurations of conditions for which there is no empirical information, but also with an added uncertainty related to the causal order of the counterfactuals.

Given this rather severe restriction, it is not surprising that an initially promising extension to incorporate the temporal dimension, was not further developed. There are now over 10 years since the article by Caren and Panofsky appeared, and nothing else was further mentioned.

Another, under appreciated and almost forgotten paper was contributed by Hino (2009). Building on the work of Caren and Panofsky, as well as Ragin and Strand, he proposed a number of different other techniques to introduce the temporal dimension, using various forms of time-series: Pooled QCA, Fixed Effects QCA, and Time Differencing QCA. While in the previous TQCA technique, it was the *sequence* of events that leads to a specific outcome, Hino's approach (called TS/QCA) considers the *changes* of the events as causal factors that produce a certain outcome.

One major advantage of the TS/QCA is related to the number of causal conditions, that remains equal to the one from the classical QCA (in the former TQCA, additional columns are created for each pair of causal order between two conditions). In TS/QCA, it is the number of dataset rows that is multiplying, as a function of the numbers of cases and the number of temporal measurements: the same case can appear multiple times in the dataset, function of how many measurement points it has.

In the Time Difference method, only two measurements are of interest (the start point, in Hino's case 1980, and the end point, 1990) for each case. The difference between them is then trivially transformed into binary crisp sets using a value of 1 for a positive difference (the end point value is higher than the starting point), and a value of 0 otherwise.

```
data(HC)
minimize(truthTable(HC, outcome = "VOTE"))
```

M1: FOREIGN*PRES80 + FOREIGN*UNEMP*CONV + UNEMP*CONV*PRES80 <=> VOTE

Including the counterfactuals, this becomes:

```
minimize(truthTable(HC, outcome = "VOTE"), include = "?")
```

M1: CONV + PRES80 <=> VOTE

Arguably, Hino's method could perhaps be improved by employing fuzzy sets when calibrating the difference between the start time and end time. Instead of 1 for positive difference and 0 otherwise, one could imagine fuzzy scores to take into account *large* differences, negative towards zero and positive towards 1, while very small differences would be close to the 0.5 crossover point. By examining the full range of differences, dedicated calibrated thresholds could be employed and allow the final calibrated conditions to have fuzzy, rather than crisp values.

The Time Difference method is quite simple, and can be used without resorting to any other specially designed algorithms, it works with a normal QCA minimization process. And due to the recent theoretical developments, it can also benefit the counterfactual analysis in the (Enhanced) Standard Analysis, using directional expectations etc.

However, Berg-Schlusser (2012, p. 210) points that Hino's method can be applied with metric variables only, and in addition it does not detect particular sequences of events, as the previous TQCA method did. I would personally not worry too much about the metric issue as almost all causal conditions (especially in fuzzy sets QCA) are measured on a metric scale, one way or another.

The second issue is indeed something to be concerned about, and Berg-Schlusser mentions a possible solution of defining the condition itself as a causal sequence, and something very similar was mentioned by De Meur et al. (2009, p. 162) to integrate the temporal dimension into the conditions themselves and create "dynamic" conditions.

For Ragin and Strand's example, this would probably imply having the condition EBA indicating with a value of 1 if E happens before A happens, and 0 otherwise (including the situations when E does not happen, or A does not happen or both not happening).

However this approach still implies an additional information about the causal order between two *existing* conditions, whereas a dynamic temporal condition should contain the information itself. Obviously, a different condition ABE would be needed to signal if A happens before E happens.

This approach has the big advantage of eliminating the “don’t care” code, and the data becomes more standard, more simple to minimize, and the ice on the cake it opens up the possibility to include remainders for temporal QCA. With the two (binary crisp) conditions EBA and ABE, the conditions A and E are now redundant, and the solution changes to:

```
RS2 <- RS
RS2$ABE <- recode(RS$EBA, "0 = 1; else = 0")
RS2$EBA <- recode(RS$EBA, "- = 0; else = copy")
minimize(RS2[, c(1, 4, 7, 5, 6)], outcome = "REC")
```

M1: P*EBA*abe + eba*ABE*S => REC

This result is perfectly equivalent to the one minimizing the original dataset RS, without the conditions A and E:

```
minimize(RS[, -c(2, 3)], outcome = "REC")
```

M1: P*EBA + eba*S => REC

This solution is a bit different from the original one presented by Ragin and Strand, but much of the essential is preserved. For instance, the expression P*EBA is logically equivalent with the former P*E*A*EBA, and eba*S is also part of the original solution. The expression P*E*S is no longer found, but it neither entailed anything about the temporal order between E and A.

Temporal order of events is almost always associated with causal analysis, therefore a full fledged temporal QCA methodology has to be intrinsically related to causation. The paper by Mahoney et al. (2009) is a must read for anyone wanting to understand both concepts, in terms of set relations. They described a method called *sequence elaboration*, introducing all sorts of relations between a cause and an effect: necessary, sufficient, both necessary and sufficient, INUS and SUIN, combined with yet another battery of such relations with an intervening cause (either antecedent or subsequent to the main cause, etc.).

These are all very interesting, especially since they employ set theory which is the bread and butter of the QCA analysis. The next section deals with a methodology specifically tailored for causal analysis, similar yet somehow different from QCA.

10.2 Coincidence Analysis: CNA

About two decades after the introduction of QCA by Ragin (1987), another interesting methodological approach has been contributed by Baumgartner (2009), based on his PhD thesis defended a couple or years earlier, and named CNA—coincidence analysis.

While QCA analyses one outcome using various configurations of causal conditions, CNA inspects all possible causal configurations not only with respect to the outcome, but also among the causal conditions themselves. It is more than a temporal QCA, although a causal context presumes that the cause must happen before the effect.

The idea of analyzing possible relations between conditions is not new, having been used for decades in SEM—Structural Equation Models. In fact, the newer terminology in CNA (Baumgartner and Thiem 2017, [online first](#)) speaks of exogenous and endogenous factors that are custom for SEM models. It is the adaptation of causal models to QCA which is certainly interesting and constitutes a possible advancement of the QCA methodology.

Although claiming to be different and even superior to QCA, as it will be shown in this section, CNA is in fact a sub-variant of QCA as acknowledged by its creator (Baumgartner 2013, p. 14):

... instead of CNA one might just as well speak of case analysis—or even of causal-chain- QCA (ccQCA)...

The terminology of CNA is a bit different: *factors* (and sometimes *residuals*) instead of *conditions*, or *effects* instead of *outcomes*, or *coincidence lists* instead of *truth tables* (although not identical), but the similarities are obvious.

At the time of its appearance, CNA was (and I believe it still is) a promising QCA extension. More recently, Baumgartner and Thiem (2017, [online first](#)) claim that QCA is actually incorrect when employing the conservative and the intermediate solutions, a good enough reason for more thorough analysis of the CNA methodology, both on its own and also comparing its capabilities to those of its larger QCA sibling. As it will be shown, CNA has its merits but for truth table type of analyses it is essentially a sub-variant of QCA.

There are all sorts of myths being vehiculated, that initiate a methodological need for clarifications. For instance, the following initial affirmation from Baumgartner (2009, p. 72) is interesting:

... QCA is designed to analyze causal structures featuring exactly one effect and a possibly complex configuration of mutually independent direct causes of that effect.

In addition, on page 78:

... Contrary to QCA, however, the data fed into CNA are not required to mark one factor as the effect or outcome.

This is essentially stating that QCA as a technique cannot observe but a single outcome at a time, while CNA is free to observe all possible relations between all conditions, each one being considered an outcome. This is partially true, the QCA algorithm being indeed restricted to a single outcome, but it induces an image of a more limited QCA technique because of this apparent limitation. In reality, there is nothing to prevent researchers running several QCA analyses and vary the outcome among the conditions, to see which of the conditions form causal structures.

The only difference is the possibility of performing the entire CNA procedure in an automated fashion, but this is merely a software feature not a methodological limitation. In fact, as it will be shown later, the version 3 of *QCA* package offers a similar automated procedure that can reproduce the results from CNA. But it should be noticed that such an automated procedure is mechanical and data driven, an important aspect that will be treated later in more depth.

Another myth that is vehiculated, probably more important in scope because it is supposed to show the alleged superiority of CNA, refers to the Quine-McCluskey algorithm and the use of remainders. Baumgartner (2013, p. 14) states that:

... contrary to QCA, CNA does not minimize relationships of sufficiency and necessity by means of Quine-McCluskey optimization but based on its own custom-built minimization procedure.

Also, from the abstract of Baumgartner (2015, p. 839):

... in order to maximize parsimony, QCA—due to its reliance on Quine-McCluskey optimization (Q-M)—is often forced to introduce untenable simplifying assumptions. The paper ends by demonstrating that there is an alternative Boolean method for causal data analysis, viz. Coincidence Analysis (CNA), that replaces Q-M by a different optimization algorithm and, thereby, succeeds in consistently maximizing parsimony without reliance on untenable assumptions.

All of these quotes imply that different from QCA, the “custom built” algorithm behind CNA does not rely on any remainders, and thus emerges as a reliable alternative to QMC, that is not “forced” to introduce untenable simplifying assumptions.

The apparent reliance on the classical QMC optimization was already disproven in Chap. 9: the modern QCA algorithms do not necessarily rely explicitly on remainders and more importantly do not necessarily rely on the classical QMC algorithm. Instead, package *QCA* employs (since 2007) a pseudo-counterfactual analysis that produces exactly the same results as those from the classical QMC approach.

This myth should not be left without a further clarification relating to the second part, that CNA does not rely on any remainders in the process of maximizing parsimony. It is rather easy to compare the new QCA algorithms to the one described by CNA. Baumgartner (2009, p. 86) provides the following

definition to find a *coincidence* (another terminological departure from the established concept of a *prime implicant*):

SUF: A coincidence X_k of residuals is sufficient for Z_i iff the input list C contains at least one row featuring $X_k Z_i$ and no row featuring $X_k \bar{Z}_i$.

This is, unsurprisingly, very similar to Definition 9.1 of a prime implicant (i.e. a sufficient expression X_k for Z_i) introduced when describing the Consistency Cubes procedure:

A prime implicant is the simplest possible, non-redundant, fully consistent superset of any positive positive output configuration.

To reiterate the explanation, fully consistent also means it can never be a superset of an observed negative configuration (“no row featuring $X_k \bar{Z}_i$ ”). It has a full inclusion score equal to 1, in the set of supersets of the positive output configurations, and conversely an inclusion score of 0 in the set of supersets of the negative output configurations.

This is of course valid for *exact* Boolean minimization, for truth table based analyses, but it is not out of the question, and also in the spirit of the fuzzy sets, to sometimes lower this inclusion score such that if an expression has a 0.95 consistency for sufficiency (therefore not full but almost), it could still be accepted as a prime implicant.

Chapter 9 demonstrates unequivocally that a pseudo-counterfactual analysis produces exactly the same parsimonious solutions as the one produced via the classical Quine-McCluskey procedure, as if it included the remainders. Whether or not some methodologists agree, for truth table type of analyses CNA behaves like a pseudo-counterfactual analysis therefore it is in fact implicitly using the remainders.

If the CNA and QCA algorithms are indeed similar and equivalent, they should reach exactly the same solutions. As indicated before, version 3 of package *QCA* offers a function called `causalChain()`, that is going to be compared with the function `cna()` from package *cna* version 2.0 (Ambuehl and Baumgartner 2017).

The specification of the function `causalChain()` is very basic:

```
causalChain(data, ordering = NULL, strict = FALSE, ...)
```

The arguments `ordering` and `strict` are there to ensure similar functionality with function `cna()`, while the three dots `...` argument can be used to pass various parameters to function `minimize()`, which is used in the background by the wrapper function `causalChain()`.

Two things are necessary to be explained. First, package *cna* is only focused on parsimonious solution types. If not otherwise specified, function `causalChain()` silently sets the argument `include = "?"` to arrive at this type

of solution. Second and similar, there is another argument called `all.sol` in function `minimize()` that allows (when activated) to find all possible solutions for a given data, irrespective of how many conjunctions it contains.

To be compliant with the QCA standard, the default is to leave this argument inactive but since package *cna* explores the full solution space, function `causalChain()` silently activates it (if not otherwise specified) to obtain the same solutions.

Having introduced all this background information, the stage is set to test the results produced by both packages. To demonstrate that package *cna* arrives at exactly the same solutions as those produced by the classical QMC procedure, the argument `method` is going to be provocatively set to the value "QMC" before being passed to the main minimization function. Identical results would count as evidence that CNA is a pseudo-counterfactual method.

As the examples from package *cna* are geared for the coincidence analysis, some of the examples from the help file of function `cna()` are going to be used:

```
library(cna)
data(d.educate)
cna(d.educate, what = "a")
```

```
--- Coincidence Analysis (CNA) ---
```

```
Factors: U, D, L, G, E
```

```
Atomic solution formulas:
```

```
Outcome E:
```

condition	consistency	coverage	complexity
L + G <-> E	1	1	2
U + D + G <-> E	1	1	3

```
Outcome L:
```

condition	consistency	coverage	complexity
U + D <-> L	1	1	2

The purpose of developing the function `causalChain()` is less to be a complete replacement of the function `cna()`, but rather to show they produce equivalent results. For this reason, it will only be compared with the *atomic solution formulas* (asf, printed using the argument `what = "a"`) that are part of the output produced by function `cna()`.

```
cc <- causalChain(d.educate, method = "QMC")
cc
```

```
M1: U + D <=> L
```

```
M1: L + G <=> E
```

```
M2: U + D + G <=> E
```

As it can be seen, the classical QMC algorithm produces exactly the same *atomic* (parsimonious) solutions as those produced by the package *cna*. The output is not completely similar, but the results are identical: there are two models produced by for the outcome E and one produced for the outcome L. As already mentioned, the two models for outcome E are produced by activating the argument `all.sol`, to find the disjunction $G + U + D$ which is otherwise not minimal. Just like package *cna*, the package *QCA* is also able to identify the full model space of a solution.

Although not printed on the screen, the resulting object contains the minimization results for all columns in the data `d.educate`.

```
names(cc)
```

```
[1] "U" "D" "L" "G" "E"
```

```
cc$U
```

```
M1: d*L => U
```

The minimization for the outcome U, despite being sufficient, is not revealed as necessary because it's coverage for sufficiency (which is the same as the consistency for necessity) is less than the coverage cut-off specified by the argument `sol.cov` from function `minimize()` defaulted to 1. This is revealed by the right arrow sign `=>` only, whereas to be part of a causal chain, the atomic solutions have to be both sufficient and necessary, as displayed in the output for outcomes E and L with the double arrow sign `<=>`.

To inspect the actual inclusion and coverage scores, each solution is an object of class "qca" which has a component called IC:

```
cc$E$IC
```

	inclS	PRI	covS	covU	(M1)	(M2)
1 G	1.000	1.000	0.571	0.143	0.143	0.143
2 U	1.000	1.000	0.571	0.000		0.143
3 D	1.000	1.000	0.571	0.000		0.143
4 L	1.000	1.000	0.857	0.000	0.429	
M1	1.000	1.000	1.000			
M2	1.000	1.000	1.000			

As both models for the outcome E have inclusion and coverage equal to 1, they are identified as necessary and sufficient, hence part of the causal chain model. The second example from the function `cna()` uses a well known Krook (2010) data on representation of women in western-democratic parliaments:

```
data(d.women)
cna(d.women, what = "a")
```

```
--- Coincidence Analysis (CNA) ---
```

```
Factors: ES, QU, WS, WM, LP, WNP
```

```
Atomic solution formulas:
```

```
-----
*none*
```

I have deliberately modified the original example command in the function, to emphasize a certain aspect that I am going to refer immediately. For the moment, let us inspect the result from function `causalChain()`:

```
causalChain(d.women, method = "QMC")
```

```
M1: WS + es*LP + ES*WM + QU*LP <=> WNP
```

```
M2: WS + ES*WM + QU*LP + WM*LP <=> WNP
```

While the default settings of the function `cna()` do not reveal any atomic solution formulas, the function `causalChain()` accurately reflects the results published by Krook. The original example command from package *cna* contains an additional argument called `maxstep`, that has three values:

```
data(d.women)
cna(d.women, maxstep = c(3, 4, 9), what = "a")
```

```
--- Coincidence Analysis (CNA) ---
```

```
Factors: ES, QU, WS, WM, LP, WNP
```

```
Atomic solution formulas:
```

```
-----
Outcome WNP:
```

	condition	consistency	coverage	complexity
WS + ES*WM + es*LP + QU*LP <-> WNP		1	1	7
WS + ES*WM + QU*LP + WM*LP <-> WNP		1	1	7

Using these settings, the function `cna()` is also able to reveal Krook's necessary and sufficient conditions. From the help file of the function, we learn more about the argument `maxstep`, which has a form of `c(i, j, k)`, that indicates:

... the generated asf have maximally *j* disjuncts with maximally *i* conjuncts each and a total of maximally *k* factors (*k* is the maximal complexity).

This argument raises a question about how the CNA algorithm works. Naturally, users can experiment with various combinations of *i*, *j* and *k*, but the bottom line is that nobody knows for sure to which is the set of minimal values that produces a complete causal chain in a reasonable time, as increasing any of those values will increase the time spent to search for a possible solution.

This is even more important as some of the values might yield a necessary and sufficient model for one particular outcome, but not for another outcome. In order to make sure that such necessary and sufficient models are found (if they exist) for all outcomes, the values from the argument `maxstep` would have to as large as needed, and this increases the search time. The explanation from the help file is revealing:

As the combinatorial search space for `asf` is potentially too large to be exhaustively scanned in reasonable time, the argument `maxstep` allows for setting an upper bound for the complexity of the generated `asf`.

By setting a default (low) upper bound for the complexity, neither minimality nor exhaustiveness are guaranteed. The search space is indeed very large, sometimes, and it should be reduced to find at least some solutions. But some other times it may happen that a minimal solution exists above the default bound, and many more complex disjunctions even above that.

In contrast, the function `causalChain()` need not to be told of such bounds as it performs an exhaustive search to find: (a) all possible prime implicants, at all levels of complexity, and (b) all possible solutions from a given prime implicants chart. Since the `CCubes` algorithm is exact and exhaustive, this is perfectly possible for classical QCA minimizations on truth tables.

It is true however, that using a lower threshold for the solution consistency sidesteps a finite PI chart and the search space potentially increases to infinity. In such situations, some upper bound is needed and if not otherwise specified it is silently set to a default of five prime implicants per solution, an upper bound that can be modified via the argument `sol.depth` in function `minimize()`.

Continuing the comparison with package `cna`, it has other interesting features for causal analysis. By default, all factors from a certain dataset are rotated as outcomes, having all other columns as causal conditions. But this need not always be the case, in some situations theory might stipulate for sure which causal condition happens before which other effect. In a way, this looks very similar to declaring a SEM diagram in a path analysis.

To achieve this, the function `cna()` provides the argument called `ordering`, which has to be specified as a list where each component might have one or several factors on the same level. The net effect is to make sure that temporally antecedent factors may never be used as outcomes for the temporally subsequent factors. The complementary argument called `strict` dictates whether factors from the same temporal level may be outcomes for each other.

```
data(d.pban)
mvcna(d.pban, ordering = list(c("C", "F", "T", "V"), "PB"),
      cov = 0.95, maxstep = c(6, 6, 10), what = "a")
```

--- Coincidence Analysis (CNA) ---

Causal ordering:
C, F, T, V < PB

Atomic solution formulas:

Outcome PB=1:

	condition consistency
C=1 + F=2 + C=0*F=1 + C=2*V=0 <-> PB=1	1
C=1 + F=2 + C=0*T=2 + C=2*V=0 <-> PB=1	1
C=1 + F=2 + C=2*F=0 + C=0*F=1 + F=1*V=0 <-> PB=1	1
C=1 + F=2 + C=2*F=0 + C=0*T=2 + F=1*V=0 <-> PB=1	1
C=1 + F=2 + C=0*F=1 + C=2*T=1 + T=2*V=0 <-> PB=1	1
coverage complexity	
0.952	6
0.952	6
0.952	8
0.952	8
0.952	8
... (total no. of formulas: 14)	

One thing to notice, at this command, is that it uses a different function called `mvca()`, which is a shortcut for using the main function `cna()` with the argument `type = "mv"`. The function needs to be told what kind of data is used, while package *QCA* automatically detects it. That is only a matter of convenience, but more importantly it leads to a drawback for package *cna*, because it apparently cannot use a mix of different types of sets in the same data, as indicated in the help file of the function `cna()`:

... data comprising both multi-value and fuzzy-set factors cannot be meaningfully modelled causally.

This is an absolute major drawback that is already solved in package *QCA* since many years ago (for multi-value sufficiency, see Eq. (6.2)), accepting any kind of data for *QCA* minimizations, and now for causal modeling as well.

Returning to the actual command, a coverage threshold of 0.95 is used, to relax the very strict default value of 1, generating 14 different models for the outcome PB. While the similar argument `ordering` can also be fed as a list for the function `causalChain()`, I took the liberty of improving the specification of the temporal order in a single string, as customary for many other functions in package *QCA*:

```
causalChain(d.pban, ordering = "C, F, T, V < PB", sol.cov = 0.95,
            method = "QMC")
```

- M01: C{1} + F{2} + C{0}*F{1} + C{2}*V{0} <=> PB{1}
- M02: C{1} + F{2} + C{0}*T{2} + C{2}*V{0} <=> PB{1}
- M03: C{1} + F{2} + C{0}*F{1} + C{2}*F{0} + F{1}*V{0} <=> PB{1}
- M04: C{1} + F{2} + C{0}*F{1} + C{2}*T{1} + T{2}*V{0} <=> PB{1}
- M05: C{1} + F{2} + C{0}*F{1} + T{1}*V{0} + T{2}*V{0} <=> PB{1}
- M06: C{1} + F{2} + C{0}*T{2} + C{2}*F{0} + F{1}*V{0} <=> PB{1}
- M07: C{1} + F{2} + C{0}*T{2} + C{2}*T{1} + T{2}*V{0} <=> PB{1}


```

M08: C{1} + F{2} + C{0}*T{2} + T{1}*V{0} + T{2}*V{0} <=> PB{1}
M09: C{1} + F{2} + C{0}*F{1} + C{2}*F{0} + F{1}*T{1} + T{2}*V{0} <=> PB{1}
M10: C{1} + F{2} + C{0}*F{1} + C{2}*T{1} + F{0}*T{2} + F{1}*V{0} <=> PB{1}
M11: C{1} + F{2} + C{0}*F{1} + F{0}*T{2} + F{1}*V{0} + T{1}*V{0} <=> PB{1}
M12: C{1} + F{2} + C{0}*T{2} + C{2}*F{0} + F{1}*T{1} + T{2}*V{0} <=> PB{1}
M13: C{1} + F{2} + C{0}*T{2} + C{2}*T{1} + F{0}*T{2} + F{1}*V{0} <=> PB{1}
M14: C{1} + F{2} + C{0}*T{2} + F{0}*T{2} + F{1}*V{0} + T{1}*V{0} <=> PB{1}

```

Hopefully, it is by now evident that CNA produces the very same solutions as the classical Quine-McCluskey algorithm, even though it is not explicitly using any remainders. This fact leads to the conclusion that CNA is implicitly using remainders, and therefore qualifies as a pseudo-counterfactual method.

There is however a certain feature of the CNA algorithm that makes it different from the classical Boolean minimization: it can search for solutions with lower levels of consistency (note the argument `con` instead of `cov`):

```

mvcna(d.pban, ordering = list(c("C", "F", "T", "V"), "PB"),
      con = .93, maxstep = c(6, 6, 10), what = "a")

```

--- Coincidence Analysis (CNA) ---

Causal ordering:

C, F, T, V < PB

Atomic solution formulas:

Outcome PB=1:

	condition	consistency	coverage
C=1 + F=2 + T=2 + C=2*T=1 <-> PB=1		0.955	1
C=1 + F=2 + T=2 + C=2*F=0 + F=1*T=1 <-> PB=1		0.955	1
complexity			
5			
7			

The major drawback with this command is related to the combination of arguments `con = 0.93` (which automatically sets the related argument `con.msc` to the same value) and `maxstep = c(6, 6, 10)`. It is highly unclear how a regular user could ever find this particular combination of numbers to produce these solutions, other than trial and error, a fact which makes the usage experience cumbersome and prone to potential imprecisions.

The function `minimize()`, and of course the wrapper function `causalChain()` from package *QCA* also has some arguments to allow modifying these consistency thresholds. For instance, the argument `pi.cons` is equivalent to the argument `con.msc` from function `cna()`, and the argument `sol.cons` is equivalent to the argument `con` from the same function. The solutions are again the same:

```
causalChain(d.pban, ordering = "C, F, T, V < PB", pi.cons = 0.93,
            sol.cons = 0.95)
```

```
M1: C{1} + F{2} + T{2} + C{2}*T{1} <=> PB{1}
M2: C{1} + F{2} + T{2} + C{2}*F{0} + F{1}*T{1} <=> PB{1}
```

In this command, it is important to note the removal of the argument `method = "QMC"` to allow employing the default method "CCubes", the tweaking of the prime implicant consistency threshold being a feature that is foreign to the classical minimization algorithm.

Also, readers might notice the argument `sol.depth` was not used, despite a less perfect consistency threshold for the solution. When left unspecified, an automatic upper bound of 5 prime implicants is used, which is enough to produce the same solutions found by package *cna* (the second solution is a disjunction of exactly 5 prime implicants).

Introducing a coverage threshold of 0.95 for the solution finds six other, different solutions, some even more parsimonious. Needless to say they are identical with those found by function `cna()`:

```
causalChain(d.pban, ordering = "C, F, T, V < PB", pi.cons = 0.93,
            sol.cons = 0.95, sol.cov = 0.95)
```

```
M1: C{1} + F{2} + T{2} <=> PB{1}
M2: C{1} + T{2} + C{2}*T{1} <=> PB{1}
M3: F{2} + T{2} + F{1}*T{1} <=> PB{1}
M4: C{1} + F{2} + V{0} + C{0}*F{1} <=> PB{1}
M5: C{1} + T{2} + C{2}*F{0} + F{1}*T{1} <=> PB{1}
M6: F{2} + V{0} + C{0}*F{1} + F{1}*T{1} <=> PB{1}
```

Since it is now possible to deviate from the *exact* minimization and plunge into the fuzzy sets consistencies, the results can become even more spectacular while still highly consistent (albeit not perfect). The downside is that users are now over-equipped with multiple arguments, many referring to various consistency or coverage thresholds. They should know for certain what each argument does, before changing the defaults.

It is perhaps a good moment to take stock of the overall landscape. While CNA does not use a traditional truth table (but a coincidence list), QCA is still a truth table based procedure, and I am arguing this is a feature, not a problem, for QCA. Besides being an useful intermediate object to inspect before a minimization, truth tables provide a very synthetic image of the complex data researchers are analyzing, as described in Chap. 7.

And this is nowhere more relevant than in the examples using calibrated fuzzy data, such as the one from the next example from package *cna*:

```
data(d.autonomy)
dat2 <- d.autonomy[15:30, c("AU", "RE", "CN", "DE")]
fscna(dat2, ordering = list("AU"), con = .9, con.msc = .85, cov = .85,
      what = "a")
```

--- Coincidence Analysis (CNA) ---

Causal ordering:
RE, CN, DE < AU

Atomic solution formulas:

```
-----
Outcome AU:
           condition consistency coverage complexity
RE*cn + re*CN <-> AU           0.92    0.851           4
re*DE + cn*DE <-> AU           0.90    0.862           4
```

Again, the same solutions are obtained by function `causalChain()`, using the corresponding arguments `sol.cons`, `pi.cons` and `sol.cov`, with an additional information. When using a certain threshold for the solution consistency, it is mandatory to use a similar (better an equal) threshold for the argument `incl.cut` from function `truthTable()`, otherwise the generated truth table will transform the fuzzy sets to crisp truth tables using a full inclusion score of 1. This is of course valid for both crisp and fuzzy sets.

```
causalChain(dat2, ordering = "AU", sol.cons = 0.9, pi.cons = 0.85,
            sol.cov = 0.85)
```

```
M1: re*CN + RE*cn <=> AU
M2: re*DE + cn*DE <=> AU
```

Unless otherwise specified, when either `pi.cons` or `sol.cons` are set below the default value of 1, the function `causalChain()` silently sets the value for the argument `incl.cut` to 0.5 to accommodate all possible truth tables for all possible outcomes. The argument `incl.cut` is then passed to function `minimize()`, which in turn passes it to function `truthTable()` which uses it to construct the truth table.

The fact that both algorithms found exactly the same solutions in all presented examples is yet another evidence of the similarity of the algorithms behind the *cna* and *QCA* packages. For full consistencies, one should expect a 100% exact overlap between the results of QCA and CNA, but these are not identical packages and there is no formal proof that their outcome are identical every single time when lowering the consistency thresholds. Rarely, discrepancies might appear and further investigation would be needed to uncover the reasons for their appearance.

But as it can be seen, the arguments from the function `causalChain()` are perfectly compatible with those from the function `cna()`, although it was not built to completely replace all the features in the sibling package (additional

features are rather easy to implement). It is a proof of concept that, given the similar minimization procedures, *QCA* can be successfully used to perform causal analysis and achieve the same results.

One crucial difference between *CNA* (version 2) and *QCA* is that *CNA* does *not* use a truth table (but a coincidence list), while *QCA* remains a truth table based method. Apart from the obvious advantages of creating a truth table, I argue that *QCA* preserves a net superiority over *CNA* because it allows an exhaustive search, therefore a guaranteed complete list of solutions.

CNA is not guaranteed to be exhaustive, even when a coincidence list is identical with a truth table. By contrast, there is only a single situation where *QCA* might use an upper bound, when searching for solutions with a lower consistency threshold (using the argument `sol.depth`). In all other situations the *CCubes* algorithm is exact and exhaustive.

As shown, package *QCA* shares none of the drawbacks identified in package *cna*, and benefits from all theoretical achievements related to the Standard Analysis and Enhanced Standard Analysis, working with various types of counterfactuals, dealing with contradictions, having a single integrated function to deal with all types of sets (crisp, multi-value, fuzzy), negating multi-value sets etc.

And best of all, one could only imagine the potential, fruitful interplay between causal chains and intermediate solutions based on the Enhanced Standard Analysis, and package *QCA* is able to provide such possibilities.

10.3 Panel/Clustered Data

For the information presented so far, the functions provided in package *QCA* are more than sufficient. There are however recent, cutting-edge theoretical developments that have not been (yet) implemented in this package. It is actually impossible for a single software package to keep track of every possible development, a good reason for other authors to write their own extensions.

R is very well suited for such situations, because the functions from the main package are public and can be subsequently used in another package, as long as the new package declares package *QCA* in the dependency chain. The new functionality becomes available by simply loading the extension package.

For the remaining sections from this chapter, several other packages will be introduced, along with their functionality. Since they are outside the scope of the main package, I will only make a brief introduction and invite users to further consult the papers and manuals written by their authors. It would actually be difficult for me to properly describe the functionality of the functions developed by other programmers, other than what it can be read from their help files.

A first package that is deeply integrated and provides the largest extension is called *SetMethods* (Medzihorsky et al. 2017) arrived at version 2.3 at the time of this writing. Apart from some usual QCA functions that seem to duplicate part of the existing functionality (XY plots, parameters of fit etc.), this package has an original set of functions to deal with panel data, and some others designed for what is called set-theoretic MMR (multi method research).

Adapting QCA to panel data is a natural extension involving temporal events, presented in Sect. 10.1. Panel data also contain temporal events, but unlike the previous attempts where the data is cross-sectional, panels are longitudinal and the data for the same cases are gathered over a larger number of cross-sectional units over time.

Garcia-Castro and Ariño (2016) paved the way for such an adaptation, and proposed a number of modified consistency and coverage measures for both across cases and over time. By facilitating this kind of analysis:

... we can infer sufficient and necessary causal conditions that are likely to remain hidden when researchers look at the data in a purely cross-sectional fashion. Further, across-time consistencies can be used as robustness checks in empirical studies.

The data being spread over cases and years, consistency and coverage measures can be calculated:

- between cases, in the same year (one measure per year)
- within cases, over the years (one measure per case)
- pooled, combining the measurements for all cases and all years

The *between consistency* is the most common, with the same equation as for any other consistency, measuring how consistent are the cases with respect to the outcome inside each year. It may be seen as a sequence of regular cross-sectional data consistencies, for each time t :

$$inclB_{X_t \Rightarrow Y_t} = \frac{\sum_{i=1}^N \min(X_{it}, Y_{it})}{\sum X_{it}} \quad (10.1)$$

In this equation, t is held constant to indicate a certain year, and all cases from 1 to N contribute to the calculation of the between consistency.

The *within consistency* does the opposite, holding each case i as constant (one measure per case) and the calculations are performed using the different measurements of the same case over the years. It essentially measures how the changes of the same case in condition X are related to the outcome Y of the same case, across time:

$$inclW_{X_i \Rightarrow Y_i} = \frac{\sum_{t=1}^T \min(X_{it}, Y_{it})}{\sum X_{it}} \quad (10.2)$$

It can be seen the two equations are very similar, measuring how consistent is X as a subset of Y, either between all cases of the same year or within each case over time.

Finally, the *pooled consistency* brings together all cases over all years, to calculate a composite (pooled) measure:

$$\text{incl}P_{X \Rightarrow Y} = \frac{\sum_{t=1}^T \sum_{i=1}^N \min(X_{it}, Y_{it})}{\sum_{t=1}^T \sum_{i=1}^N X_{it}} = \frac{\sum \min(X, Y)}{\sum X} \quad (10.3)$$

Equation (10.3) can be expressed either in the most simple form (not specifying what to sum meaning that it sums everything, all cases from all years), or separating the double sums to make it more obvious the summation happens for both years and cases.

The coverage measures are calculated exactly the same, just replacing the X in the numerator with Y. In principle, the consistency and coverage measures work even for two or three unit observations, but the more time measurements in the panel the more precise the calculations of these measures will become.

There are some other measures specific to panel data, and they would be better exemplified using a real dataset. The package *SetMethods* should be loaded first, to access its datasets and functions:

```
library(SetMethods)
```

This package provides a dataset called `SchneiderLong`, used by Schneider et al. (2010) to explore the institutional capital of high-tech firms and their export performance, using on data from 19 OECD countries between 1990 and 2003.

```
data(SCHLF)
head(SCHLF)
```

	EMP	BARGAIN	UNI	OCCUP	STOCK	MA	EXPORT	COUNTRY	YEAR
Australia_90	0.07	0.90	1.00	0.68	0.45	0.33	0.19	Australia	1990
Austria_90	0.70	0.98	0.01	0.91	0.01	0.05	0.25	Austria	1990
Belgium_90	0.94	0.95	0.14	0.37	0.26	0.14	0.14	Belgium	1990
Canada_90	0.04	0.21	0.99	0.11	0.62	0.31	0.28	Canada	1990
Denmark_90	0.59	0.78	0.10	0.55	0.53	0.10	0.34	Denmark	1990
Finland_90	0.70	0.97	0.20	0.95	0.02	0.13	0.17	Finland	1990

Apart from the usual conditions and outcome columns specific to QCA, the package *SetMethods* requires two more columns for a panel dataset analysis: one containing an identifier for the years when the data was collected for each case, and the other containing an identifier for the cases.

A regular dataframe in R has the option of providing the case identifiers using the row names, and the row names have to be unique. Since the same cases can be repeated for multiple measurements in time, they cannot be assigned to the row names and instead specified into a different column.

These columns are used by the function `cluster()`, with argument `unit` referring to the case names and the argument `cluster` referring to the year. The overall structure of the function, including all arguments, is presented below:

```
cluster(data, results, outcome, unit_id, cluster_id, sol = 1,
        necessity = FALSE)
```

The arguments `data` and `results` can also be vectors, but their most natural usage refers to the original dataset. The argument `sol` specifies the number of the solution to be selected, in case of multiple models. The argument `results` can be used to feed the function with a minimization object of class "qca", in which case the argument `sol` is specified as a string of type "c1p1i1" to specify the exact combination of conservative, parsimonious and intermediate solution to be selected.

For instance, the following object is created when minimizing the Schneider data for intermediate solutions (for demonstrative purposes, only the first four conditions were used, but the package documentation contains the complete example):

```
ttSL <- truthTable(SCHLF, conditions = "EMP, BARGAIN, UNI, OCCUP",
                  outcome = "EXPORT", incl.cut = .9, show.cases = TRUE)
sol_yi <- minimize(ttSL, include = "?", dir.exp = "0, 0, 0, 0")
cluster(results = sol_yi, data = SCHLF, outcome = "EXPORT",
        unit_id = "COUNTRY", cluster_id = "YEAR")
```

Consistencies:

	emp*bargain*OCCUP	EMP*bargain*occup	emp*BARGAIN*occup
Pooled	0.909	0.960	0.924
Between 1990	0.839	0.991	0.838
Between 1995	0.903	0.991	0.912
Between 1999	0.928	1.000	1.000
Between 2003	0.951	0.878	0.954
Within Australia	1.000	1.000	0.791
Within Austria	1.000	1.000	1.000
Within Belgium	1.000	1.000	1.000
Within Canada	1.000	1.000	1.000
Within Denmark	1.000	1.000	0.774
Within Finland	1.000	1.000	1.000
Within France	1.000	1.000	1.000
Within Germany	1.000	1.000	1.000
Within Ireland	1.000	1.000	1.000
Within Italy	1.000	1.000	1.000
Within Japan	1.000	1.000	1.000
Within Netherlands	1.000	1.000	1.000
Within NewZealand	0.414	0.868	0.437
Within Norway	0.965	0.958	0.948
Within Spain	1.000	0.706	1.000
Within Sweden	1.000	1.000	1.000
Within Switzerland	0.880	1.000	1.000
Within UK	1.000	1.000	1.000
Within USA	1.000	1.000	1.000

Distances:

```

-----
                                emp*bargain*OCCUP  EMP*bargain*occup  emp*BARGAIN
                                                                *occup
From Between to Pooled                0.023                0.026                0.032
From Within to Pooled                  0.031                0.017                0.033
    
```

Coverages:

```

-----
                                emp*bargain*OCCUP  EMP*bargain*occup  emp*BARGAIN*occup
Pooled                                0.194                0.229                0.334
Between 1990                          0.231                0.289                0.399
Between 1995                          0.206                0.249                0.469
Between 1999                          0.174                0.206                0.261
Between 2003                          0.184                0.203                0.274
Within Australia                       0.415                0.333                0.951
Within Austria                         0.075                0.075                0.442
Within Belgium                        0.138                0.138                0.372
Within Canada                         0.328                0.299                0.545
Within Denmark                        0.273                0.273                0.604
Within Finland                        0.059                0.059                0.332
Within France                         0.070                0.070                0.173
Within Germany                        0.236                0.251                0.308
Within Ireland                        0.113                0.103                0.580
Within Italy                           0.173                0.327                0.276
Within Japan                          0.161                0.656                0.064
Within Netherlands                    0.150                0.169                0.355
Within NewZealand                    1.000                0.917                0.861
Within Norway                         0.598                0.739                0.598
Within Spain                          0.204                0.828                0.204
Within Sweden                         0.061                0.075                0.189
Within Switzerland                    0.738                0.315                0.337
Within UK                             0.075                0.080                0.282
Within USA                            0.037                0.052                0.045
    
```

Since this particular example contains exactly one intermediate solution, the default arguments to select it need not be respecified in the command. Following Garcia-Castro and Ariño’s paper, the function returns a complete set of pooled, between and within consistencies and coverages of the solution terms, for each unit and each cluster. There are also distances from between to pooled and from within to pooled, offering a complete snapshot of the complex panel relations that exist in this dataset.

All of these parameters of fit are calculated for the sufficiency relation, but otherwise the function `cluster()` can calculate the same parameters for the necessity relation, by activating the argument `necessity = TRUE`.

Package *SetMethods* has a large variety of functions that greatly extend the main package *QCA*, not only for panel data but also for what is probably the hottest QCA extension, called Set Theoretic Multi-Method Research. Since both the method and the associated functions are still under a heavy development, they are not presented here but readers are warmly encouraged to follow the literature and the associated functions from this package.

10.4 Robustness Tests

As a middle ground between the qualitative and the quantitative approaches, QCA has inevitably attracted a lot of attention but also a lot of critique, especially from the quantitative researchers. One of the particular contested features of QCA is the ability to work with a small number of cases (fitting in the *small and medium-N* world), as opposed to statistical analyses which require a high number of cases to perform inference testing.

Arguably, it is easier to collect a small sized data, and it requires a lot of effort to collect data from large scale studies involving thousands of individuals. This is only apparent, because the data collection is very much different: while the quantitative studies focus on a standardized approach over thousands of individuals, the qualitative approach is much more intensive for an in-depth study of each and every case.

In order to establish, for instance, at which particular values to set the calibration thresholds, the QCA researcher should have a deep understanding of not only each separate case but also of each case in comparison with all the others. In the same situation, the quantitative researcher would simply calculate the mean and standard deviation, and Chap. 4 shows clearly why such an approach is not suitable.

Nevertheless, quantitative researchers continue to raise doubts about inferences drawn from a small number of cases (small “sample”) and their ability to generalize conclusions to a larger “population”. But this is very far from the QCA approach, which does not use the Central Limit Theorem, it does not use the normal curve, and it does not use confidence intervals or p -values. Instead, QCA works by observing patterns of consistency between various configurations of causal conditions, and how they are associated with the presence of the phenomenon of interest.

A well known, landmark qualitative study is Skocpol’s (1979) book on states and social revolutions, comparing no more than three cases: France, Russia and China. These are very rare events, and they all involve crucial moments in the history of humanity. Obviously, these cases cannot be analyzed using the standard statistical techniques and the quantitative researchers are powerless when asked to draw meaningful insights.

On the other hand to abstain from analyzing these cases, just because they don’t comply with the normal curve, would be an even bigger mistake. There is a genuine possibility to analyze them using the comparative method, since they all display the same outcome and there are some obvious similarities, as well as differences between them.

And this is precisely what QCA does, searching for patterns of consistency among the causal conditions with the outcome. Three cases are clearly not enough to apply QCA, even for a moderately complex theoretical model, in

terms of number of causal conditions. Marx and Duşa (2011) have paved the way for a possible optimal ratio between the number of cases and the number of causal conditions, and that work should be extended to with more tests using fuzzy sets. With an optimal such ratio, the Boolean minimization method can be applied to identify consistency patterns, but the quantitative researchers continue to compare the results of this algorithmic method with the standard statistical techniques.

One such critique, emerging from the work of King et al. (1994, notorious for their interpretation of qualitative research from the quantitative perspective), refers to the selection of cases being analyzed. Generally in the qualitative research, and more specifically in the QCA, cases being studied are not selected using random sampling techniques (as it is usually the situation in quantitative research) but quite the reverse, they are selected precisely because they display something which is theoretically meaningful.

In the language of “sampling”, this is similar to what is called *purposive sampling* (a non-probability method) where cases are selected because they are typical for some research scenario, or inversely because they are deviant compared to the other cases: something potentially interesting happens in those cases and we want to find out what.

What is probably most misunderstood is that QCA is a *configurational* analysis, and the focus on the (number of) cases is of a secondary importance. The Boolean minimization process is not aware of the number of cases within each configuration, since the input consists of the configurations themselves. QCA is less concerned with the number of cases, since the limited diversity phenomenon states that only a few causal configurations are actually observable.

There could be thousands of cases and only a handful of observed configurations, and that is something different from the quantitative focus on the number of cases. The only situation when the number of cases does count, is their effect on output value of a particular truth table configuration, either through the inclusion threshold, or via the frequency cut-off.

This is a rather long introduction to the topic of this section, related to the robustness of QCA results, but it was necessary to understand that robustness is a very often ill interpreted topic. It all relates to a natural expectation that QCA results should be consistent with previous findings. In this respect, it overlaps with an overarching scientific requirement that a finding is valid (only) if it can be generalized to future similar research situations, or even more strict if they can be generalized to a “population”.

But this is a very quantitative assessment of robustness, as if lab results from a micro research situation are expected to work in the real world scenarios, or inferences drawn from a sample are expected to be generalized to the entire population. It is an inferential way of looking at the QCA models, and expects consistent results on relatively similar input data.

Robustness is indeed an important issue, and it can be approached in a systematic way. There are so many publications dealing with this issue (some of the most recent including Skaaning 2011; Hug 2013; Lucas and Szatrowski 2014; Seawright 2014; Ragin 2014; Lucas 2014; Krogslund et al. 2015) many of which are similar in nature, that is difficult to address all of their arguments.

Arguably, however, many critiques misunderstand and sometimes misinterpret what has been written before. For instance, Lucas and Szatrowski (2014) argue that Ragin and Rihoux (2004):

... reject simulations, claiming that QCA is for researchers intrinsically interested in the cases they study.

Upon a careful reading of that paper, there is little evidence that Ragin and Rihoux explicitly reject simulations. It is true that QCA researchers are intrinsically interested in the cases they study, but from here to declaring this general approach as equivalent to rejecting simulations, is a long stretch.

In a subsequent rejoinder to clarify matters, Lucas (2014) manages to bring even further confusion over some of them. For instance, Ragin's example with the difference between a set membership and a probability is very clear, but Lucas includes a particular Bayesian interpretation of the probability as a degree of belief and declares the probability as a mathematical function of the set membership. This is (at best) an opinion, since no such mathematical function exists.

The same type of deliberate (since Lucas seems to understand what QCA is all about but still prefers the) misunderstanding is related to equating the configurations with the interaction effects. These are strong indicators that, although he rejects looking at QCA through a quantitative perspective, it is precisely what he is doing.

Whether or not QCA is able to find the "correct" answer depends on the quality of the raw data, and to follow Skaaning (2011) on the choice of calibration thresholds, as well as on the choice of frequency and consistency thresholds when constructing the truth table. Measurement error is important in both micro-quantitative and macro-qualitative comparative analyses. On a macro level, measurement error has a lower influence on the results. Societal indicators fluctuate a whole lot less than individual scores, and if we agree on this point then it is obvious that "error" has a different meaning in QCA.

In particular, Hug (2013) attempted a more systematic way of looking at the effect of errors (deleting a case from the analysis, or changing the value of the outcome) to calculate the extent to which the final solution is preserved: the results are "robust" if the solution is preserved.

His attempt (exhaustive enumeration) has already been outperformed by Thiem et al. (2016), with a proper combinatorial method to calculate the retention rate. The simulations include two types of assumptions: DPA—Dependent Perturbation Assumption when perturbations depend on each

other and are tied ex-ante to a fixed number of cases, and IPA—Independent Perturbation Assumption when perturbations are assumed to occur independently of each other.

The entire procedure is enclosed in the function called `retention()` in package *QCA*, with the following structure:

```
retention(data, outcome = "", conditions = "", incl.cut = 1, n.cut = 1,
          type = "corruption", dependent = TRUE, p.pert = 0.5, n.pert = 1)
```

Similar to the main `truthTable()` function, the arguments `incl.cut` and `n.cut` decide which configurations are coded positive, negative or left as logical remainders. The argument `p.pert` specifies the probability of perturbation under the independent perturbations assumption, while argument `n.pert` specifies the number of perturbations under the dependent perturbations assumption with at least one perturbation needed to possibly change a csQCA solution, otherwise the solution remains the same (retention rate equal to 100% if zero perturbations occur).

The argument `type` is the one to choose between "corruption" when various values in the conditions are changed (either from 0 to 1 or from 1 to 0), or "deletion" when the result is the probability of retaining the same solution if a number of cases are deleted from the original data.

The effect of corrupting a case is to allocate it to a different truth table configuration. This may or may not have an influence on the final result, as the input in the minimization procedure is not the cases themselves but the truth table. A case leaving a certain configuration might influence the consistency of that configuration, which in turn might affect its output value. The same with a new case arriving into another truth table configuration.

The effect of deleting a case can have an effect over the same truth table configuration, either by modifying its consistency, or possibly by transforming it into a remainder if it does not pass the frequency cut-off.

It is also possible for a corruption or a deletion to not have any effect at all, depending on the number of cases and the consistency of each configuration. And even in the case the configurations are changed, it can still lead to the same solution if those configurations contain redundant (minimizable) conditions. In all these situation, the solution is retained. The solution is changed in the inverse situation: when the number of cases per configuration is small, the consistency score the configuration being changed is not very high, and the configuration being changed plays an important role in the minimization.

However this scenario is perfectly logical in a macro-qualitative comparative setting, changing the value of a case (a country) has the same effect as replacing one country with another. Under these circumstances the solution being modified should not be surprising at all, especially when the group of countries being analyzed is small. What is truly surprising is Hug's (and all the others') expectation the solution should be retained.

Using his own version of the data, the exact retention probability can be calculated with:

```
Hug <- data.frame(matrix(c(
  rep(1,25), rep(0,20), rep(c(0,0,1,0,0),3),
  0,0,0,1,0,0,1,0,0,0,0, rep(1,7),0,1),
  nrow = 16, byrow = TRUE, dimnames = list(
  c("AT", "DK", "FI", "NO", "SE", "AU", "CA", "FR",
    "US", "DE", "NL", "CH", "JP", "NZ", "IE", "BE"),
  c("P", "U", "C", "S", "W")))
))

retention(Hug, outcome = "W", type = "corruption", dependent = FALSE,
  p.pert = 0.025, incl.cut = 1)
```

```
[1] 0.7962228
```

In the independent perturbation assumption, using a perturbation probability of 0.025 when corrupting cases, the exact retention rate is 0.796, which means the probability that the solution will change is 0.204. This is neither good, nor bad: it is what it is.

A different type of simulation was performed by Seawright (2014), who generated a truth table based on a certain Boolean function, then randomly sampled (with replacement) from those configurations in various sizes between 20 and 100. Apart from the five conditions in the initial Boolean function, Seawright introduced three other conditions that are irrelevant, and tested to what extent the limited diversity problem (including the remainders) affects the final solutions.

In the absence of a replication file, it is unclear how this simulation study was performed. His study raises more questions than answers:

- How many positive configurations did the generated truth table had?
- How many negative configurations?
- How did Seawright made absolutely sure the three additional conditions are causally irrelevant?

It is quite possible, even through random sampling, that causally relevant structures were introduced in the initial truth table for the additional three conditions. This should be tested, verified and ultimately validated, assuming of course the that Seawright was careful enough and used a mandatory starting point for the sampling procedures via the base function `set.seed()`, otherwise none of his results can ever be replicated.

It is interesting to note that, although QCA is inherently applicable to a small and medium sized number of cases, Seawright treats the input as if it was drawn from a known large population. Leaving aside his justified intention to test the results against something known, in reality he performed a very dubious exercise, as it is also unclear if he sampled from positive configurations (in which case the underlying model should be preserved to a greater extent) or from both positive and negative output configurations.

Both positive and negative output configurations are important: the positive ones influence the underlying model, while the negative ones contribute to eliminating remainders that would otherwise incorrectly contribute to parsimony (thus artificially altering the final solutions).

A newer, but similar to Seawright's simulation involving a data generating structure, was performed by Baumgartner and Thiem (2017, [online first](#)), who also make a comprehensive evaluation of the inappropriate use or inadequate tests by the previous critiques, especially Lucas and Szatrowski. Their approach is not to sample from the saturated truth table, but instead to delete all possible combinations of rows (from 1 to 14) and perform an exhaustive enumeration of the model preserving solutions. Opposite to Seawright, they found a retention rate of 100% for the parsimonious solution in all their simulation setups, with lower retention rates for the conservative and intermediate solutions.

Baumgartner and Thiem's paper serve a dual purpose: demonstrate why all the other previous attempts to measure the QCA robustness are incorrect (thus defending QCA), while at the same time they claim the parsimonious solution is the only correct solution and the conservative and intermediate solutions are also incorrect (thus indirectly becoming QCA critics themselves).

Both Seawright (2014) and Baumgartner and Thiem (2017, [online first](#)) start with the same setup of data generating structure using an underlying Boolean function, and they reach different conclusions using different methods to test robustness, yet it can be shown that both studies are incorrect.

In his attempt to verify the role of the limited diversity, Seawright most likely included *all* remainders in the analysis to obtain the parsimonious solution, and this is definitely a mistake. The decision to produce only the parsimonious solution ignores years of efforts by an entire community of QCA theoreticians and practitioners, around the intermediate solutions. Directional expectations should be used, and untenable, difficult or even impossible counterfactuals should be avoided from the minimization process.

A similar mistake can be identified in Baumgartner and Thiem, although to their credit a proper effort was invested to calculate intermediate solutions. In their case, the method is partially right and the conclusion is plain wrong, based on an ad-hoc definition of what a "correct" solution is.

The community agreed standard is that both the conservative (CS) and the intermediate solutions (IS) are supersets of the parsimonious solutions (PS). This means that every time a PS is "correct", the other two solutions are also correct because they contain the elements from the PS. Baumgartner and Thiem's definition, inspired from Seawright, is that a solution is correct only: "... iff it does not commit causal fallacies..." (i.e. if it does not contain causally irrelevant factors).

In a real limited diversity scenario (the norm in QCA research) the number of empirically observed configurations is low and the number of remainders is high. In such a situation, it is absolutely certain that CS will not be as parsimonious as PS (i.e. CS will almost always contain causally irrelevant factors), therefore in their definition CS is bound to be “incorrect”.

It is the very reason why the parsimonious solution is called “parsimonious”, having a more simple structure, opposed to the conservative solution which is highly complex when the diversity is very limited. To change the nature of what is “correct”, Baumgartner and Thiem should have provided some kind (any kind) of philosophical, formal logic or mathematical demonstration of the incorrectness. Instead, they resorted to a change in the definition, and that is not embraced by the larger community which issued a Statement on rejecting article submissions because of QCA solution type.¹

Defining a solution as “correct” if it does not contain any causally irrelevant factors, automatically increases the likelihood that a conservative solution is “incorrect”, the more rows are eliminated from the saturated truth table.

Despite the article not referring to a specific minimization algorithm, their proof rely on a series of computer simulations performed with the package *QCApro* version 1.1-1 (a fork of an older version of the main package *QCA*), using the eQMC algorithm developed by myself in 2007 and described in Sect. 9.1. Since the eQMC algorithm is pseudo-counterfactual, it is implicitly using all possible remainders, including all those untenable, difficult or even impossible counterfactuals. Such an approach has been proven as incorrect for many years, leading to the ESA—Enhanced Standard Analysis.

This fact alone should be sufficient to dismiss Baumgartner and Thiem’s conclusions, but there is even more to reveal. In their simulations, all possible combinations of rows are iteratively eliminated from the saturated truth table, and test if any components from the parsimonious solution are retained. According to this procedure, the expected percentage of “correct” conservative solutions should approach zero when the number of deleted rows approaches the total number of rows in the saturated truth table (when the diversity gets more and more limited).

Figure 4 in their paper shows the correctness preserving curve for the QCA-CS rapidly approach zero after 5 deleted rows, and springs back to life after 11 deleted rows, with an implausible 100% correct ratio for all 16 rows being deleted. In this extreme situation with all rows from the truth table deleted, a solution is impossible since there is no data remaining at all.

By mechanically applying their definition, an impossible situation (no solution) is counted as “correct” because it does not contain causally irrelevant factors. This is a logical fallacy, because something unknown (no solution)

¹ <http://www.compass.org/files/compass-rejection-policy-statement-20170814.html>.

cannot be considered correct or incorrect, it is just unknown. Consider this simple example containing seven values:

```
correct <- c(1, 0, 0, 0, NA, 0, 1)
mean(correct)
```

```
[1] NA
```

The result of averaging this vector is unknown, because the fifth value is unknown (not available). The average value can only be calculated by removing the unknown value, in which case it would be 2 out of 6, not 3 out of 7:

```
mean(correct, na.rm = TRUE)
```

```
[1] 0.3333333
```

In the Baumgartner and Thiem's case, it would imply calculating the percentage of "correct" conservative solutions only for those situations when a minimization is possible. After 11 rows being deleted from the truth table, what they report as the percentage of identified "correct" solutions is in fact the proportion of situations when no solutions are possible. For instance, there are $\text{choose}(16, 14) = 120$ possible situations of removing 14 rows out of the total of 16 from the truth table. In 10 of these situations, no solution is possible with a percentage of $10/120 = 0.083$, which is exactly the percentage reported as "correct" in the article, while the proper percentage is $0/110 = 0$.

It is not only the definition of "correctness" which is artificially bended towards their hypothesis, but also the operationalization of this definition. To consider an inexistent solution as correct counts as a programming error, and their entire simulation is flawed. Combined with the fact that a parsimonious solution is not guaranteed to be correct in the limited diversity scenario (because it involves difficult, untenable or impossible counterfactuals), their ambition to reset the de-facto QCA standards get suddenly deflated.

The same phenomenon can be identified for the intermediate solutions, when the correctness preserving curves display a very quick recovery after 11 eliminated rows, whereas they should approach zero the more rows are eliminated from the saturated truth table. By removing this logical fallacy, all curves except for the parsimonious solution should approach zero, the more limited diversity is accounted for.

The constant rate of 100% correctness for the parsimonious solutions is also highly implausible, and on a closer inspection it appears to be the result of another programming artifact. For instance, when removing the third row from the saturated truth table (using the same underlying model $aB+Bc+D$), there are not one but two parsimonious solutions: $aC+Bc+D$ and $aB+Bc+D$.

Clearly, the first one does not conform to their own definition, since the component aC is not part of the expression that generated the data, this particular simulation should be counted as a half success.

There are multiple other similar situations for all possible combinations of deleted rows, for instance deleting the rows 3 and 9 produce three parsimonious solutions: $aC+Bc+D$, $aB+Bc+D$, and $aB+Ac+D$, and in the most extreme situations there are no less than 12 possible parsimonious solutions, out of which only one conforms to their own definition.

Trying to solve this problem, Baumgartner and Thiem resort to yet another ad-hoc definition of causal correctness, by considering the entire set of models (i.e. solutions) as correct if *at least* one of the models is correct. In justifying their decision, they quote Spirtes et al. (2000, p. 81), but that page is not even remotely related to this particular choice. The only correctness property theorem found there is:

If the input to any of the algorithms is data faithful to G , the output of each of the algorithms is a pattern that represents the faithful indistinguishability class of G .

Leaving aside the fact that Spirtes et al. (2000) present statistical (not Boolean) algorithms to identify causal structures in acyclic directed graphs, it is highly unclear how did their theorem of correctness justifies the claim that all models are correct if at least one is correct.

Baumgartner and Thiem's paper relies not only on a correctness definition that is not embraced by the community, but as it seems that definition is even further artificially bended to fit the computer generated results. In a normal scientific enquiry, data should be used to test a hypothesis or a definition, but in this case the definition itself is constructed to 100% fit the output produced by the Boolean minimization algorithm.

For each simulation, the causal structure is known but it must not be forgotten that in the real life researchers have absolutely no idea which of the concurrent models is the true model representing the underlying causal structure, therefore all models should be given the same weight. In such situations displaying model ambiguity, the success rate should be calculated as 1 over the total number of models, given that only one model faithfully represents the true causal structure.

Counting such situations as 100% correct, when only one model is correct, could only be described as a programming abuse if it was intentional, or otherwise as a programming error. Either way, it is clear they published results that are misleading, as a direct consequence of an artificially constructed initial definition.

From a Boolean minimization perspective, the fact that parsimonious solutions contain less causally irrelevant factors is perfectly logical given that eliminated rows are either irrelevant in the minimization process, or they are included back through the remainders, which means the simulations are arranged in such a way that parsimonious solutions will always contain components from the data generating expression.

Contrary to their claim, the parsimonious solution does not have a magical property to uncover causal structure, it is merely a direct consequence of the Boolean minimization procedure: the more remainders are *included*, the more parsimonious the solutions will become (but never more parsimonious than the data generating causal structure).

To the other extreme the more rows are *eliminated* from the saturated truth table, the less observed configurations are served as input for the Quine-McCluskey procedure, the more severe the limited diversity problem and the more complex the conservative solutions will become.

Their article describe the built in properties of the Quine-McClukey (QMC) Boolean minimization algorithm, and they sell it as a groundbreaking finding to claim for instance that QCA-CS (conservative solution of QCA) is incorrect by employing the QMC algorithm. But they fail to mention that QCA-CS is in fact QMC proper, therefore produce a logical impossibility because what they truly claim is that QMC demonstrates that QMC is incorrect.

As Baumgartner and Thiem do not present a formal demonstration of their claims (they resort to simulated results as a form of inductive pseudo-demonstration), their paper can be subjected to at least two logical fallacies. The first is called a “definist fallacy” (Bennett 2012, p. 96):

A has definition X. X is harmful to my argument. Therefore, A has definition Y.

Similarly, it is also a case of a circular definition fallacy: the simulated results are determined as “correct” using an ad-hoc definition, and the definition is “demonstrated” as correct by the simulated results.

Given this series of logical impossibilities, logical fallacies and (ab)use of programming to change the true nature of my own algorithm eQMC, it is difficult to attribute very much credibility to the conclusions presented in this paper.

Going back to Seawright (2014, p. 121), he presents one other interesting conclusion that QCA is “fairly prone to false-positive results.” That is, for some of the solutions, at least one additional condition is found in the solutions despite not being causally related to the outcome, according to the initial Boolean function.

According to Baumgartner and Thiem, this effect never happens for the parsimonious solution if the causally irrelevant conditions are truly random. As mentioned, this finding can also be interpreted as highly subjective as they consider a model as correct if at least one solution preserves the causal structure, thus disregarding model ambiguity. It would have been interesting to validate Seawright’s own conclusions, but in the absence of a replication file this is not possible.

However on the official R repository (CRAN), there are some other packages dealing with QCA, among which one being called *QCAfalsePositive*, created by Braumoeller (2015a) as an applied programming tool to demonstrate

the findings from Braumoeller (2015b). This package has a number of interesting functions to calculate the probability of producing a type I error when including remainders. Applied to QCA, a type I error occurs when a solution contains components that appear by chance, rather than configurational merits.

Further experimentation should be performed to assess the usefulness of these inferential tests. At a first sight, and acknowledged by Braumoeller:

... If a solution set is only found in a single case and the outcome of interest occurs in 90% of the observations, the probability of a false positive result is dangerously high...

This is very true but on the other hand it might also reflect the inherent uncertainty that is specific to inferential tests when the number of observations is very small. It might be the probability of a false positive is high, or it might be that an inferential test is highly imprecise with only one case. However, such initiatives to involve statistical testing, wherever possible, are laudable.

Although QCA is inherently a method that has absolutely nothing to do with probability and statistics, there are some possible intersections, especially at the simplest 2×2 tables and XYplots where Bayesian analysis on (conditional) proportions might prove to be interesting, and possibly very useful (see for instance Barrenechea and Mahoney 2017, [online first](#)).

Dealing with robustness is another associated package from CRAN, named *braQCA* (Gibson and Burrell 2017) which is built to assess how sensitive is a QCA solution to randomness, using bootstrapping. Their main function returns a probability that a given solution is reached by chance, as it would be the result of a random experiment.

What is particularly interesting about this package is its ability to offer possible threshold values for the inclusion and frequency cut-offs, which can served as data specific guidelines to reduce the probability that a given QCA solution is spurious. This is also an interesting combination of classical statistics with QCA, and more work is needed to further demonstrate its practical utility for the QCA research.

Chapter 11

Less Known Features



This final chapter is a more practical and applied one, describing in detail the rest of the graphical user interface as well as the most used functions in package *QCA*. Although widely used, there are still some not so obvious features that remain unemployed in most of the cases. In a way, this is a chapter about all the little secrets that each function has, that in many situations are capable of making the usage experience as straightforward as possible.

Both the written functions and the graphical user interface were designed with this very purpose, to give the impression of extreme simplicity and allow users to concentrate less on the R code and focus more to the substantive theoretical part of the analysis.

A lot of effort has been spent to write the R code in order to allow all possible styles of specifying causal expressions. Negation, for example, can be specified in four different ways:

- by subtracting from 1, using the $1 - A$ type of expression when the condition *A* is a fuzzy numeric or even logical vector
- by using the universal negation exclamation sign $!A$, for logical vectors
- by using a tilde in front of the condition's name, like: $\sim A$
- by simply using lower case letters, especially in a multiple sets expression such as $a*B$

The exclamation sign is already part of the base R, and it is implemented in most programming languages. Using lower and upper case letters was the original style from the beginnings of *QCA* and it is still very much used, while using a tilde seems to be the new standard, a reason for which most of the functions have an argument called `use.tilde` to format the output and it recognizes it when signaling a negation.

All these possibilities are not native to a software like R, they had to be custom developed. While there are specific tips and tricks for each function,

there are also some overall features that are valid for multiple functions across the package.

For example, in the past some functions had an argument called `incl.cut`, while others had a similar argument called `incl.cut1`, although both were in fact referring to one and the same thing, specifying an inclusion cut-off. The former argument `incl.cut1` is still possible (all functions are backwards compatible with older arguments), however the modern versions of the package have all these arguments uniform across functions, in this case using the more universal `incl.cut`.

Specifying multiple inclusion cut-offs is made possible using R's vectorized nature. Many function arguments accept one value as an input, and sometime more values using a vector. In fact, one simple but often forgotten R feature is that single values are in fact vectors of length 1 (scalars). But there is nothing wrong with providing multiple values with one argument, as in the case of the directional expectations argument `dir.exp`, making the two former separate arguments `incl.cut1` and `incl.cut0` redundant in favor of a single argument `incl.cut`. This is just one example of small, but very effective ways to simplify the user experience as much as possible.

11.1 Boolean Expressions

There are multiple functions that deal with arbitrary sum of products (SOP) type of expressions, specified as text: `pof()`, `fuzzyor()`, `fuzzyand()`, `compute()`, `sop()` to name a few, but basically these types of strings are recognized throughout the package, for instance the function `truthTable()` recognizes whether the outcome is negated or not, using a tilde sign.

The standard way of specifying strings in R is to enclose them within double quotes, and this is recommended as a good practice advice. In addition, some of the functions recognize a negation of an object using a tilde sign, even without the quotes.

Most of these functions rely heavily on a less known function called `translate()`, that transforms any SOP expression into a corresponding matrix with the names of the conditions on the columns, and the values from the expression in the cells. This function is the workhorse for almost all other functions that deal with such expressions, and it is capable of detecting negations and even double negations such as:

```
translate("A + ~b*C")
```

	A	B	C
A	1		
~b*C		1	1

In this example, the condition B is negated twice, using both the lower case letter notation and the tilde. This is actually important, especially when such expressions contain object names from the user's workspace, it is also a good practice advice to name all objects using upper case letters. Naturally, this advice extends to the columns names from a calibrated dataset, to allow the detection of negated conditions using lower case letters.

Most of the following examples are going to use this function `translate()`, even though it has nothing to do with the QCA methodology. It is just a substitute for any of the QCA specific functions that deal with string expressions. What works for this function, works for all of them, for instance being able to recognize multi-value sets, and even negated them if the number of levels is known for each set name:

```
translate("A{1} + ~B{1}*C{1}", snames = "A, B, C", noflevels = c(2, 3, 2))
```

	A	B	C
A{1}	1		
~B{1}*C{1}		0, 2	1

The output is relatively straightforward, with one row for each of the disjunctive terms in the expression, the cells representing the values for each set, after translating the expression. All of those values are subsequently used in the recoding step: fuzzy sets are inverted (if negated), while binary and multi-value crisp sets are transformed (recoded) to a binary crisp set. In this example, values 0 and 2 for the set B are going to be recoded to 1, and the former value 1 is going to be recoded to 0.

The expression `~B{1}` is translated as “all values in set B except 1”, and since the set B has three values, all others cannot be anything else but 0 and 2, assuming the set is properly calibrated and values always start from 0.

Using a star sign “*” to separate conjunctions is recommended, but not needed for multi-value expressions, the set names being already separated by using the curly brackets notation for the values. The same is true when providing the set names:

```
translate("AB + cD", snames = "A, B, C, D")
```

	A	B	C	D
AB	1	1		
cD			0	1

Specifying the set names has another useful side effect, to order the terms in the expression according to the order of their names in the `snames` argument. In the absence of the set names, they are by default sorted alphabetically as in the following example:

```
sop("(URB + LIT)(~LIT + ~DEV)")
```

```
[1] "~LIT*URB + ~DEV*URB + ~DEV*LIT"
```

By contrast, when the set names are provided they are properly sorted in the output. Here, the condition URB precedes the condition LIT and the first term of the expression changes:

```
sop("(URB + LIT)(~LIT + ~DEV)", snames = "DEV, URB, LIT")
```

```
[1] "URB*~LIT + ~DEV*URB + ~DEV*LIT"
```

If a dataset is provided, neither the number of levels nor the set names are needed because they are taken directly from the dataset:

```
data(LM)
compute("~DEV{0} + URB{1}*IND{1}", data = LM)
```

```
[1] 1 1 1 0 1 1 1 0 0 1 0 1 0 0 0 0 1 1
```

The example above uses the multi-value version of the Lipset data, where the causal condition DEV has three values. Negating the value 0 implies all other values, 1 and 2. The same effect is obtained by directly specifying them between the curly brackets:

```
compute("DEV{1,2} + URB{1}*IND{1}", data = LM)
```

```
[1] 1 1 1 0 1 1 1 0 0 1 0 1 0 0 0 0 1 1
```

There are many ways to specify the causal conditions into an expression, and the functions from package *QCA* are compatible with those provided by the base R:

```
with(LM, compute("~DEV{0} + URB{1}*IND{1}"))
```

```
[1] 1 1 1 0 1 1 1 0 0 1 0 1 0 0 0 0 1 1
```

Out of all functions in package *QCA* that deal with Boolean expressions, the function `pof()` to calculate parameters of fit is by far the most versatile. Just like `compute()` it evaluates a Boolean expression with either objects from the workspace or column names from a specific dataset, and it does that relative to a certain outcome for sufficiency or necessity relation. This function is the Swiss army knife, many input and many purpose function of the entire package, and it is employed by all other functions that require parameters of fit (most notably the output from the main minimization function).

The most basic type of input, when thinking about inclusion, coverage and all other parameters of fit, involves two calibrated vectors. The first will play the role of the causal condition and the second the role of the outcome. In fact, the first two arguments of the function `pof()` are called `setms` and `outcome`.

The name `setms` denotes “set membership scores”, to emphasize that almost any object can contain such scores: a data frame, a matrix of implicants, and simple vectors, and this argument can even be a string containing a Boolean expression. It can even be a numeric vector containing the row numbers from the implicants matrix, in which situation the `setms` argument is automatically transformed into their corresponding set membership scores. Since it can carry so many things, the function is specifically programmed to recognize what kind of operations should be performed on which kind of input.

The argument `outcome` can also be a string containing the name of the outcome (and it will be searched for in the list of objects or in a specified dataset), or it can be a proper vector containing the membership scores for the outcome set. To begin an example, the fuzzy version of the CVF data will be used (Cebotari and Vink 2013):

```
data(CVF)
conditions <- CVF[, 1:5]
PROTEST <- CVF$PROTEST
```

Here, the conditions are the first five columns of the CVF dataset, and the outcome is the column called PROTEST. Both are now separate objects in the workspace, and can be used as such and negated using the `1-` notation:

```
pof(1 - conditions, PROTEST, relation = "sufficiency")
```

		inclS	PRI	covS	covU
1	~DEMOC	0.601	0.354	0.564	0.042
2	~ETHFRACT	0.614	0.337	0.661	0.036
3	~GEOCON	0.601	0.246	0.317	0.000
4	~POLDIS	0.493	0.250	0.631	0.035
5	~NATPRIDE	0.899	0.807	0.597	0.025

A number of things have happened, even with a simple command like this one. First of all, the function `pof()` automatically detected the input for the `setms` argument is a dataset. Consequently, it determined that each column contains set membership scores and calculated parameters of fit for each. Last but not least, it determined that each such column should be negated (because all conditions in the dataset were subtracted from 1), and their corresponding names in the output have a tilde in front to signal their negation.

This is a simple dataset containing single columns, but there are examples of datasets containing set membership scores for more complex expressions, such as the solutions resulting from function `minimize()` or those resulting from the function `superSubset()`. As in the case of most functions, the output of these two functions are lists containing all sorts of components, the relevant ones for this section being the component `pims` (prime implicants membership scores) from function `minimize()` and component `coms` (combinations membership scores) from function `superSubset()`.

For more complex expressions, it applies the `negate()` function to present their negated counterpart in the output.

```
ttCVF <- truthTable(CVF, outcome = "PROTEST", incl.cut = 0.8)
cCVF <- minimize(ttCVF, details = TRUE)
colnames(cCVF$pims)
```

```
[1] "DEMOC*ETHFRACT*GEOCON"
[2] "ETHFRACT*GEOCON*POLDIS"
[3] "DEMOC*ETHFRACT*POLDIS*natpride"
[4] "DEMOC*GEOCON*POLDIS*NATPRIDE"
[5] "democ*ethfract*GEOCON*poldis*natpride"
```

The component `coms` from function `superSubset()` has similar complex column names from the resulting (necessary) expressions:

```
sus <- superSubset(LF, outcome = "SURV", incl.cut = 0.9, ron.cut = 0.6)
colnames(sus$coms)
```

```
[1] "STB"          "LIT*STB"      "DEV+URB+IND"
```

In these examples, the component `pims` contains the set membership scores for the resulting prime implicants in the conservative solution. One possible use case is to check if their negation is also sufficient for the outcome:

```
pof(1 - cCVF$pims, PROTEST, relation = "sufficiency")
```

		inclS	PRI	covS	covU
1	democ+ethfract+geocon	0.575	0.362	0.841	0.000
2	ethfract+geocon+poldis	0.508	0.288	0.790	0.000
3	democ+ethfract+poldis+NATPRIDE	0.526	0.334	0.892	0.000
4	democ+geocon+poldis+natpride	0.542	0.351	0.893	0.011
5	DEMOC+ETHFRACT+geocon+POLDIS+NATPRIDE	0.567	0.388	0.945	0.044

The resulting rows in the output are equivalent to the negated expressions from the component `pims`. Naturally, the same negated expression can be supplied manually, or directly using the function `negate()`:

```
pof(negate("DEMOC*ETHFRACT*GEOCON"), PROTEST, data = CVF, relation = "suf")
```

		inclS	PRI	covS	covU
1	democ	0.601	0.354	0.564	0.137
2	ethfract	0.614	0.337	0.661	0.210
3	geocon	0.601	0.246	0.317	0.029
4	expression	0.575	0.362	0.841	-

The command from the previous example is unnecessarily complicated, however. As shown in Chaps. 5 and 6, the function `pof()` accepts fully contained

string expressions, using the left arrow notation “<=” for necessity and right arrow “=>” for the sufficiency relation:

```
pof("~DEMOC + ~ETHFRACT + ~GEOCON => PROTEST", data = CVF)
```

	inclS	PRI	covS	covU
1 ~DEMOC	0.601	0.354	0.564	0.137
2 ~ETHFRACT	0.614	0.337	0.661	0.210
3 ~GEOCON	0.601	0.246	0.317	0.029
4 expression	0.575	0.362	0.841	-

All of these different examples show just how versatile this function is, accepting almost any type of input that carries or can generate set membership scores. And there is even more to reveal, its output is an object of class "pof" that has a dedicated print method.

For instance, the output of the minimization process usually prints the parameters of fit for the solution model(s), which usually resides in the component named IC (inclusion and coverage):

```
data(LF) # if not already loaded
ttLF <- truthTable(LF, "SURV", incl.cut = 0.7)
cLF <- minimize(ttLF, details = TRUE)
cLF$IC
```

	inclS	PRI	covS	covU
1 DEV*urb*LIT*STB	0.809	0.761	0.433	0.196
2 DEV*LIT*IND*STB	0.843	0.821	0.622	0.385
M1	0.871	0.851	0.818	

Since none of the functions truthTable() and minimize() did not specify showing the cases for each solution term, this is not printed. In this situation, either rerun the commands with the argument show.cases = TRUE or, perhaps even more simple, simply ask the printing function itself to show the cases:

```
print(cLF$IC, show.cases = TRUE)
```

	inclS	PRI	covS	covU	cases
1 DEV*urb*LIT*STB	0.809	0.761	0.433	0.196	FI, IE; FR, SE
2 DEV*LIT*IND*STB	0.843	0.821	0.622	0.385	FR, SE; BE, CZ, NL, UK
M1	0.871	0.851	0.818		

11.2 Negate Expressions

The discussion about Boolean expressions could not have ended without a more in-depth discussion about the different ways to negate such expressions. Previous chapters have already used the function `negate()` in several places, but it was always in the context of another topic being introduced, therefore it never got properly explained.

The structure of the function is rather simple and self-explanatory:

```
negate(expression, snames = "", noflevels, use.tilde = FALSE)
```

The first argument is a Boolean expression, and as previously shown it needs the set names if the causal conditions in the expression are not separated by formal conjunctive “*” or disjunctive “+” signs. To negate multi-value expressions, it needs to know the number of levels, and all of these arguments are passed to the underlying function `translate()` that does the heavy lifting.

Finally, the output expression contains negated expression in either lower/upper case notation (default) or it can be switched to negating using a tilde sign “~” in front of the set names.

The negation of Boolean expressions is possible due to Augustus De Morgan, a British mathematician who lived in the nineteenth century. De Morgan formulated two laws that have passed the test of time, and are highly used in formal logics and computer programming:

$$\begin{aligned}\sim(A + B) &= \sim A \cdot \sim B \\ \sim(A \cdot B) &= \sim A + \sim B\end{aligned}$$

In plain language, these two laws are translated quite simply as:

- the negation of a disjunction is a conjunction of negations
- the negation of a conjunction is a disjunction of negations

An example of the first law could be: “*not*(smartphone *or* tablet)”, which after negation is easily understood as: “*not* smartphone *and not* tablet”.

The second law is a bit more tricky but just as simple, for instance a Boolean expression such as “*not* young male” is also understood as “*not*(young *and* male)”, which after negation is finally understood as: “*not* young *or not* male”. It can be older, or female, just not both young and male at the same time.

```
negate("~(A*B)")
```

```
S1: ~(A*B)
N1: ~A + ~B
```

Boolean expressions are formulated in many ways. Apart from the different ways to signal a negated causal condition (lower case letters, using a tilde etc.),

conjunctions are also often expressed with a star sign “*” but there are situations, especially when the set names have a single letter, that conjunctions are expressed by a simple juxtaposition of the letters:

```
negate("AC + B~C")
```

```
S1: AC + B~C
N1: ~C + ~AB
```

In this example, it is clear for a human there are three sets A, B and C, but this is far from trivial for a computer program. All functions dealing with Boolean expressions in package *QCA* can detect such situations, most of the times, but it would be a good practice advice to always use a star sign for the conjunctions, even when it is very clear.

The function `negate()` does more than interpreting a simple Boolean expression, it can also detect an object resulting from the minimization process (that has a class `qca`). When the input is such an object, it searches through all its solution components and negates all of them:

```
data(LC)
pLC <- minimize(truthTable(LC, outcome = "SURV"), include = "?")
negate(pLC)
```

```
S1: DEV*STB
N1: dev + stb
```

It can even detect and negate intermediate solutions, as shown in Sect. 8.8:

```
data(LF)
ttLF <- truthTable(LF, outcome = "SURV", incl.cut = 0.8)
iLF <- minimize(ttLF, include = "?", dir.exp = "1,1,1,1,1")
negate(iLF)
```

```
S1-C1P1-1: DEV*URB*LIT*STB + DEV*LIT*ind*STB
N1-C1P1-1: dev + lit + stb + urb*IND
```

This output has a specific codification C1P1-1, which means that at the combination between the first (and the only) conservative solution C1 and the first (and the only) parsimonious solution there is only a single intermediate solution -1 being generated.

But there are situations with many intermediate solutions, in which case there will be multiple numbers where -1 is, and all of those for all possible combinations of conservative and parsimonious solutions, which themselves can be generated in multiple numbers depending on the level of ambiguity found in the data.

11.3 Factorize Expressions

Factorizing a SOP (sum of products) expression leads to what might be seen as an opposite, POS (product of sums) expression: it finds all possible combinations of common factors in a given expression.

The common factors in a SOP expression, especially when the expression is about a sufficiency relation, are all INUS conditions. Therefore the need to factorize an expression is a quest to find those INUS conditions which are found in as many solution terms as possible, revealing their relative importance when the outcome is present.

Similar to the `negate()` function, the conditions in the expression are treated in alphabetical order, unless otherwise specified using the same argument called `snames`:

```
factorize("one*TWO*four + one*THREE + THREE*four",
         snames = "ONE, TWO, THREE, FOUR")
```

M1: one*TWO*four + one*THREE + THREE*four

F1: one*(THREE + TWO*four) + THREE*four
 F2: one*TWO*four + THREE*(one + four)
 F3: four*(THREE + one*TWO) + one*THREE

The treatment of SOP expressions is universal across functions, the negation of individual conditions being signaled either with a lower case letter or using a tilde:

```
factorize("~ONE*TWO*~FOUR + ~ONE*THREE + THREE*~FOUR",
         snames = "ONE, TWO, THREE, FOUR")
```

M1: ~ONE*TWO*~FOUR + ~ONE*THREE + THREE*~FOUR

F1: ~ONE*(THREE + TWO*~FOUR) + THREE*~FOUR
 F2: ~ONE*TWO*~FOUR + THREE*(~ONE + ~FOUR)
 F3: ~FOUR*(THREE + ~ONE*TWO) + ~ONE*THREE

Wherever possible, the input can be transformed into a complete POS expression, where all common factors are combined with exactly the same other INUS conditions. It is not always guaranteed that such a POS expression is possible, but it can be searched by activating the argument `pos`:

```
factorize("ac + aD + bc + bD", pos = TRUE)
```

M1: ac + aD + bc + bD

F1: (a + b)(c + D)

Naturally, such a factorization is possible using a minimization object directly, in which case it is applied on every model in the output:

```
data(CVF)
pCVF <- minimize(CVF, outcome = "PROTEST", incl.cut = 0.8,
                 include = "?", use.letters = TRUE)
factorize(pCVF)
```

M1: $e + a*B*D + A*B*C + A*C*D$

F1: $A*C*(B + D) + e + a*B*D$
 F2: $B*(a*D + A*C) + e + A*C*D$
 F3: $D*(a*B + A*C) + e + A*B*C$

M2: $e + a*B*D + A*B*d + A*C*D$

F1: $A*(B*d + C*D) + e + a*B*D$
 F2: $B*(a*D + A*d) + e + A*C*D$
 F3: $D*(a*B + A*C) + e + A*B*d$

M3: $e + A*B*C + A*C*D + B*C*D$

F1: $A*C*(B + D) + e + B*C*D$
 F2: $B*C*(A + D) + e + A*C*D$
 F3: $C*(A*B + A*D + B*D) + e$
 F4: $C*D*(A + B) + e + A*B*C$

M4: $e + A*B*d + A*C*D + B*C*D$

F1: $A*(B*d + C*D) + e + B*C*D$
 F2: $B*(A*d + C*D) + e + A*C*D$
 F3: $C*D*(A + B) + e + A*B*d$

Finally, the negation of a certain model might be interesting to factorize:

```
factorize(negate("e + a*B*D + A*B*C + A*C*D"))
```

M1: $A*c*E + a*b*E + b*c*E + b*d*E + a*d*E + c*d*E$

F01: $E*(A*c + a*b + b*c + b*d + a*d + c*d)$
 F02: $a*E*(b + d) + E*(A*c + b*c + b*d + c*d)$
 F03: $a*E*(b + d) + b*E*(c + d) + c*E*(A + b + d)$
 F04: $a*E*(b + d) + b*d*E + c*E*(A + b + d)$
 F05: $a*E*(b + d) + c*E*(A + b) + d*E*(b + c)$
 F06: $b*E*(a + c + d) + E*(A*c + a*d + c*d)$
 F07: $a*d*E + b*E*(a + c + d) + c*E*(A + d)$
 F08: $A*c*E + b*E*(a + c + d) + d*E*(a + c)$
 F09: $c*E*(A + b + d) + E*(a*b + b*d + a*d)$
 F10: $a*d*E + b*E*(a + d) + c*E*(A + b + d)$
 F11: $a*b*E + c*E*(A + b + d) + d*E*(b + a)$
 F12: $d*E*(b + a + c) + E*(A*c + a*b + b*c)$
 F13: $A*c*E + b*E*(a + c) + d*E*(b + a + c)$
 F14: $a*b*E + c*E*(A + b) + d*E*(b + a + c)$

11.4 More Parameters of Fit

The parameters of fit are traditionally revolving around inclusion and coverage (raw and unique), plus PRI for sufficiency and RoN for necessity. These are the standard parameters, as far as the current best practices indicate.

But standards are always subject to possible changes. Through sheer testing from the community, special situations can be found where the standard measures don't seem to provide all the answers. Haesebrouck (2015), for instance, believes that QCA's most heavily used parameter of fit—consistency measure—is significantly flawed, because:

...inconsistent cases with small membership scores exert greater bearing on the consistency score than inconsistent cases with large membership scores. In consequence, the measure does not accurately express the degree to which empirical evidence supports statements of sufficiency and necessity.

Starting from the classical consistency formula from Eq. (6.3):

$$inclS_{X \Rightarrow Y} = \frac{\sum \min(X, Y)}{\sum X} \quad (11.1)$$

Haesebrouck observes that cases with a large membership score in X have a greater impact on the consistency measure than cases with a low membership scores in X, even if their inconsistent part is equal: X = 1 and Y = 0.75 versus X = 0.25 and Y = 0. The impact of the inconsistent part (0.25) in the first situation is lower than the impact of an equal inconsistent part (0.25) in the second situation. He also observes that membership scores in X can be redefined by the sum of the consistent plus inconsistent parts:

$$inclS_{X \Rightarrow Y} = \frac{\sum \min(X, Y)}{\sum (\min(X, Y) + \max(X - Y, 0))} \quad (11.2)$$

The consistent part of X is equal to the numerator of the fraction, while the inconsistent part is either equal to 0 (if X is completely consistent) or equal to the difference between X and Y, when X is larger than Y. As a consequence, he proposes to increase the impact of the inconsistent part for higher membership in X by first multiplying it with the value of X, than taking its square root:

$$inclH_{X \Rightarrow Y} = \frac{\sum \min(X, Y)}{\sum (\min(X, Y) + \sqrt{\max(X - Y, 0) \cdot X})} \quad (11.3)$$

This is a possible improvement over the standard formula, a candidate for a new standard if there will be enough recognition and usage from the academic community. However, it is difficult for any new candidate to replace the state of the art standard, since most software offer only the standard measures.

For any such situations, the function `pof()` offers the possibility to add alternative measures via the argument `add`. It accepts a function, simplistically defined with two parameters `x` and `y`, that returns any sort of inner calculation based on these two parameters.

The example below defines such a function and assigns it to an object called `inclH`. The object is served to the function `pof()`, which augments the output with the new function name:

```
inclH <- function(x, y) {
  sum(fuzzyand(x, y)) /
  sum(fuzzyand(x, y) + sqrt(fuzzyor(x - y, 0)*x))
}

pof("DEV => SURV", data = LF, add = inclH)
```

	inclS	PRI	covS	covU	inclH
1 DEV	0.775	0.743	0.831	-	0.720

If more such functions need to be provided, the argument `add` also accepts a list object, containing a function on each component. The names of the components will become the names of the new parameters of fit in the augmented output, trimmed to the first five characters.

While still at the parameters of fit section, for most situations a SOP (sum of products) expression should be sufficient, when referring to specific column names from a dataset. This is a rather new improvement of this function, previously such expressions could also be provided in their matrix equivalent.

For instance, an expression such as `DEV·ind + URB·STB` could also be written in the matrix form:

```
DS <- matrix(c(1, -1, -1, 0, -1,
              -1, 1, -1, -1, 1), ncol = 5, byrow = TRUE)
colnames(DS) <- colnames(LF)[1:5]
DS
```

	DEV	URB	LIT	IND	STB
[1,]	1	-1	-1	0	-1
[2,]	-1	1	-1	-1	1

This matrix uses a standard value of 1 for the presence of a condition, a value of 0 for the absence of the condition and a value or -1 if the condition is minimized. In this example, since `DEV·ind + URB·STB` is the only parsimonious solution for the outcome `SURV` (at a 0.75 inclusion cut-off), their parameters of fit can be obtained through this matrix:


```
pof(DS, "SURV", data = LF, relation = "sufficiency")
```

	inclS	PRI	covS	covU
1 DEV*ind	0.815	0.721	0.284	0.194
2 URB*STB	0.874	0.845	0.520	0.430

This is of course equivalent to the direct SOP expression:

```
pof("DEV*ind + URB*STB => SURV", data = LF)
```

	inclS	PRI	covS	covU
1 DEV*ind	0.815	0.721	0.284	0.194
2 URB*STB	0.874	0.845	0.520	0.430
3 expression	0.850	0.819	0.714	-

As mentioned in Sect. 11.1, functions `minimize()` and `superSubset()` generate in their output components named `pims` (prime implicants membership scores) and `coms` (component membership scores), that can be used as input for the parameters of fit.

```
psLF <- minimize(LF, outcome = "SURV", include = "?", incl.cut = 0.75)
head(psLF$pims)
```

	DEV*ind	URB*STB
AU	0.27	0.12
BE	0.00	0.89
CZ	0.10	0.91
EE	0.16	0.07
FI	0.58	0.03
FR	0.19	0.03

These are the membership scores in the sets defined by the prime implicants DEV·ind and URB·STB, which can be verified using the function `compute()`:

```
compute("DEV*ind", data = LF)
```

```
[1] 0.27 0.00 0.10 0.16 0.58 0.19 0.04 0.04 0.07 0.72 0.34 0.06 0.02
[14] 0.01 0.01 0.03 0.33 0.00
```

The component `pims` from the output of the function `minimize()` can be used to calculate parameters of fit directly, with the same results:

```
pof(psLF$pims, LF$SURV, relation = "sufficiency")
```

	inclS	PRI	covS	covU
1 DEV*ind	0.815	0.721	0.284	0.194
2 URB*STB	0.874	0.845	0.520	0.430

11.5 XY Plots

An XY plot is a scatterplot between two objects measured in fuzzy sets. It is a visualization tool to inspect to what extent one set is a subset of the other, in order to assess the sufficiency and/or necessity of one set to the other.

There are multiple ways of obtaining such a plot, but the most straightforward way is to use the function `XYplot()` from package *QCA*. It has the simplest possible structure of arguments, but most importantly it offers all the flexibility and the wealthy choice of parameters from the native function `plot()`, including all graphical parameters that are available via `?par`.

The function has the following structure:

```
XYplot(x, y, data, relation = "sufficiency", mguides = TRUE,
       jitter = FALSE, clabels = NULL, enhance = FALSE, model = FALSE, ...)
```

Similar to all other functions in this package, it has a set of default values but the only one that has a visible effect is the relational argument, calculating the parameters of fit for the sufficiency relation. The total number of parameters is kept to a minimum, and most of them are deactivated by default (not jittering the points with the argument `jitter`, not enhancing XY plot via the argument `enhanced`, etc.).

The argument `mguides` adds two lines (a horizontal and a vertical one) through the middle of the XY plot, allowing to divide the plot region into four areas corresponding to a 2×2 cross-table. This is a very useful way to visualize where the fuzzy coordinates of the points would be located into the binary crisp cells of a table.

The two main arguments for this function are `x` and `y`, and they accept a variety of things as possible inputs. The most straightforward interpretation of these arguments are two numerical vectors containing fuzzy values between 0 and 1. The first (`x`) is going to be used for the horizontal axis, and the second (`y`) for the vertical axis.

For a first exemplification, we are going to use the same dataset *CVF* that was already loaded in Sect. 11.1. The outcome of this data is called *PROTEST* (more exactly ethnopolitical protest), and we might be interested which of the causal conditions might be sufficient. A first to draw the attention is called *NATPRIDE* (national pride), with the following plausible hypothesis: the absence of the national pride is a sufficient condition for ethnic protests.

This hypothesis could be visualized with the following command:

```
XYplot(1 - CVF$NATPRIDE, CVF$PROTEST)
```

As it can be seen in Fig. 11.1, most of the points are located in the upper left part of the plot, indicating sufficiency. This is confirmed by a relatively high

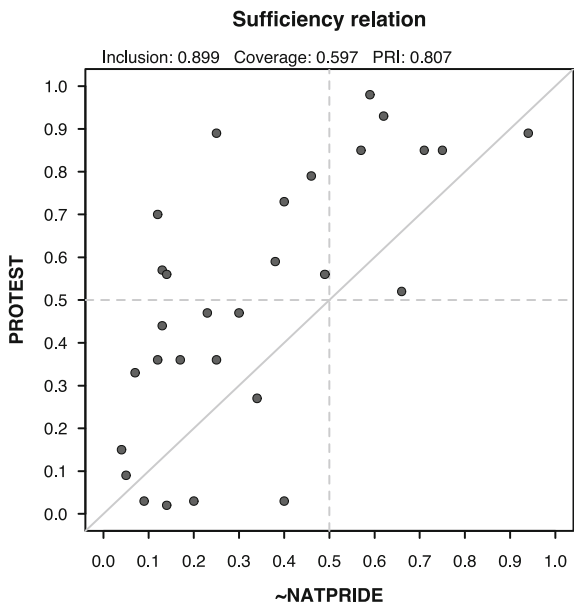


Fig. 11.1 Fuzzy relation between the absence of national pride and ethnic protest

inclusion score of 0.899 reported right below the “Sufficiency relation” title, with a similar high PRI score of 0.807, both confirming that an absence of national pride is indeed associated with ethnopolitical protests.

However, it cannot be concluded this is a causal relationship because the coverage is rather small (0.597) which means that although clearly sufficient, the absence of national pride is not a necessary condition for the appearance of ethnopolitical protests.

Returning to the command, the condition NATPRIDE was negated by subtraction from 1 (as usual in fuzzy sets), and this always works when the input is already represented by numerical vectors. But the same result would have been obtained with the following (unevaluated) command:

```
XYplot(1 - NATPRIDE, PROTEST, data = CVF)
```

In this example, the input for the argument *x* is not a numerical vector, but the name of the condition (NATPRIDE), which is to be found in the data called *CVF* (hence the need to specify the additional argument *data*).

Similar to many other functions in package *QCA*, especially function `pof()`, these column names can be provided with or without the double quotes, and

they can be negated using the 1- notation (subtracting the set membership scores from 1) or using a preceding tilde.

```
XYplot(~NATPRIDE, PROTEST, data = CVF, jitter = TRUE, cex = 0.7,
       clabels = rownames(CVF))
```

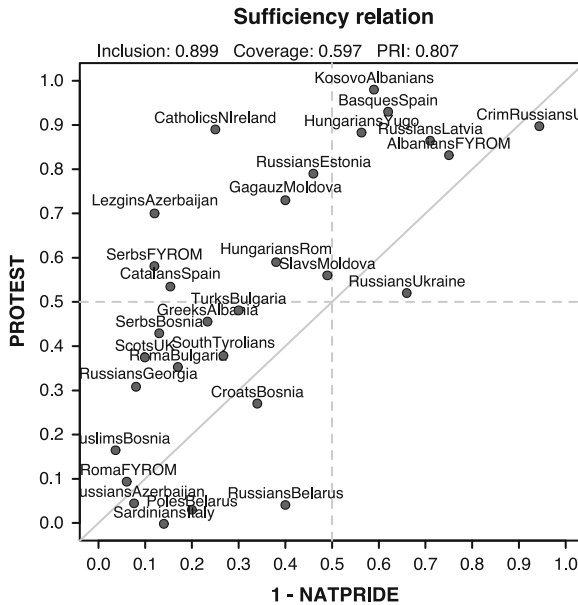


Fig. 11.2 The same XY plot, containing the jittered case labels

Figure 11.2 is an interesting plot for the usage of row names for the case labels added to the plot. A bit off-topic, but this is an example of how *not* to use the row names. Where possible (and most of the QCA data refer to countries as cases) the best approach is to use a two letters abbreviation of each case and explain somewhere in the help file what each abbreviation stands for.

For countries, this is rather easy because there are already established standards for the two letter names of the countries. But even here that would have been possible, for instance using CB instead of CroatsBosnia. The net effect of using very long names is their likely overlap, especially in datasets with a large number of cases. The argument `jitter` adds some random noise to the position of each point, to avoid a complete overlap (especially when the values are very close), but it doesn't completely solve the problem.

As the case labels in this plot are very long and very dense, perhaps a better approach is to use the row numbers as case identifiers. This is the purpose of the argument `clabels`, which can be a vector of case labels with the same length as the numeric vectors specified via argument `x` and `y`. Alternatively, it

can be a logical vector of the same length as the number of rows in the data, to add only those labels where the vector has a true value (Fig. 11.3).

```
XYplot(~NATPRIDE, PROTEST, data = CVF, clabels = seq(nrow(CVF)))
```

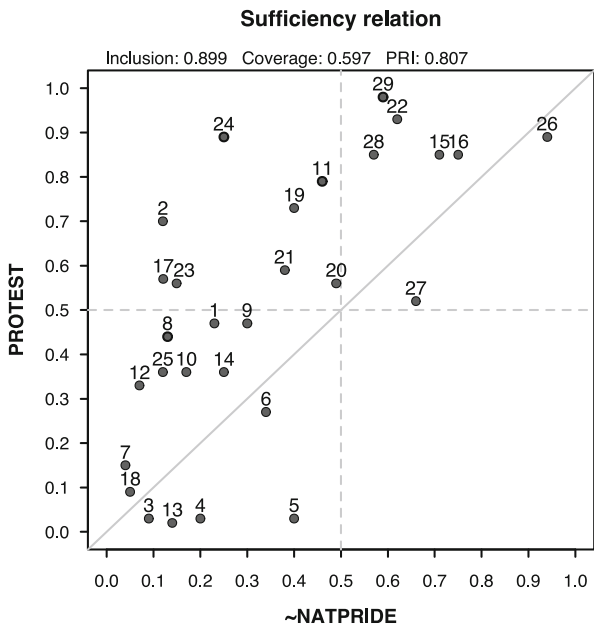


Fig. 11.3 The XY plot using numbers as case identifiers

For this plot, the argument `clabels` has been supplied with a vector of row numbers instead of country names, to avoid overlapping labels especially for very close points. R has a native, base function called `jitter()` which in turn has two additional parameters called `factor` and `amount` to control how much the points are randomly jittered. These arguments are not part of the formal arguments in the function `XYplot()` but they can still be used because they are automatically captured and interpreted via the three dots `...` argument.

This argument with three dots `...` is a special one, among others it can be used to access all graphical parameters from the native plot function. The most useful, to give just a few examples, are: `col` to add specific colors to the points, `bg` to fill the points with specific colors, `pch` (point character) to plot the points using different other signs like squares, triangles or diamonds, `cex` (character expansion, defaulted to 0.8) to control the size of the points and their associated labels etc.

Figure 11.4 is such an example, that mimics the enhanced plot presented by Schneider and Rohlfing (2016), to aid with process tracing by using different

character points for the various quadrants of the plot. The input, to demonstrate a complete range of possibilities for this function, is a SOP expression:

```
XYplot("~NATPRIDE => PROTEST", data = CVF, enhance = TRUE, jitter = TRUE)
```

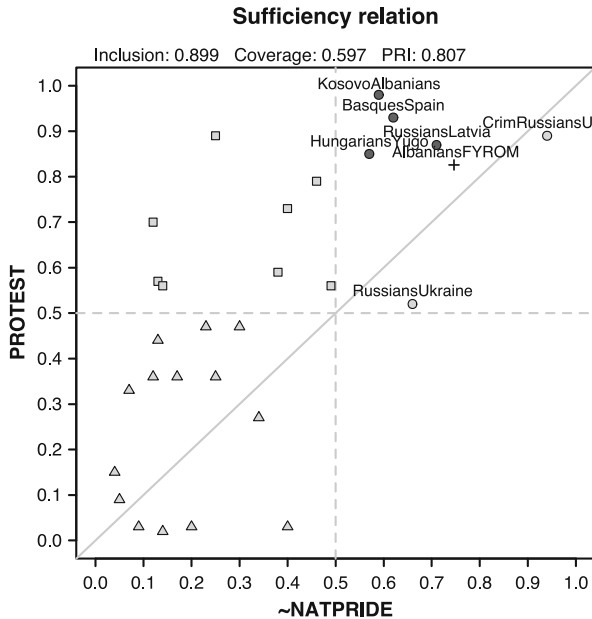


Fig. 11.4 The enhanced XY plot

The choice of point characters is very close to the description from the original paper by Schneider and Rohlfing:

- the dark filled points in zone 1 (upper right quadrant, above the diagonal) indicate the typical cases
- the cross, also in zone 1 indicate the most typical case (closest to the main diagonal)
- the light filled points in zone 2 (upper right quadrant, below the diagonal) indicate the deviant cases consistency in degree
- there are no deviant cases consistency in kind to show in zone 3 (lower right quadrant), otherwise they would have a diamond shape
- the triangles in zone 4 (entire lower left quadrant) indicate the individually irrelevant cases
- the squares in zone 5 (upper left quadrant) indicate the deviant cases coverage

By default, the enhanced plot only displays the labels for the typical and for the deviant cases consistency in degree (zones 1 and 2), a specific choice for the sufficiency relation. When the expression is the result of a minimization

process, however, the advice is add the labels for the points in zones 4 and 5, where the expression has a set membership score below 0.5 (Fig. 11.5).

```
sol <- "natpride + DEMOC*GEOCON*POLDIS + DEMOC*ETHFRACT*GEOCON"
XYplot(sol, "PROTEST", data = CVF, enhance = TRUE, model = TRUE)
```

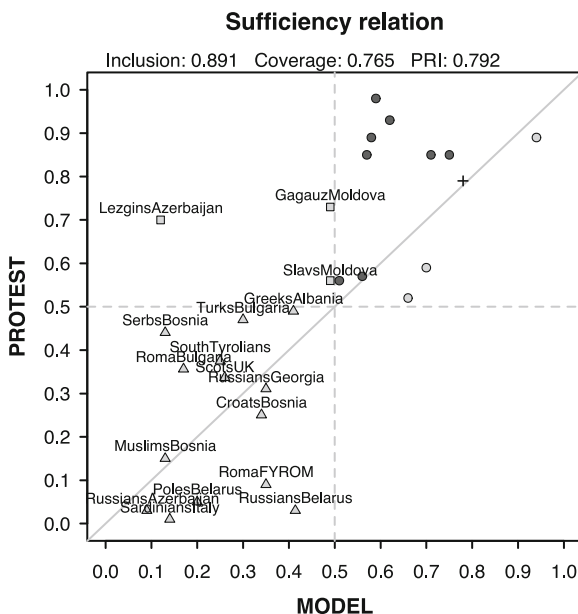


Fig. 11.5 An enhanced solution model

For this purpose, the function `XYplot()` has another logical argument called `model`, which produces effects only when the `enhance` argument is activated. The XY plot above corresponds to one of the parsimonious solutions in the CVF data, using an inclusion cut-off value of 0.85 to construct the truth table.

Activating the argument `model` is only necessary when the input is an expression, but otherwise the function `XYplot()` accepts entire objects containing the result of a minimization. For instance, since there are two parsimonious solutions, the same output could have been obtained by using this equivalent command:

```
ttCVF <- truthTable(CVF, "PROTEST", incl.cut = 0.85)
pCVF <- minimize(ttCVF, include = "?")
XYplot(pCVF$solution[1], "PROTEST", data = CVF, enhance = TRUE)
```

In this command, the function `XYplot()` automatically detects the input is a solution model (from the containing object `pCVF`), so there is no need to specifically switch `model = TRUE` to produce the same figure as above. For the second solution, only the number between the square brackets must be changed: [2]. For the intermediate solutions, the minimization object has a

component called `i.sol` which contains all combinations of conservative and parsimonious solutions (ex. `C1P1`), each containing the solutions.

All these figures have been produced using predefined settings, but otherwise users are free to create their own choice of points, or which cases from which zones to be labeled. In this direction, it is perhaps worth reminding that argument `clabels` also accepts a logical vector of the same length as the number of rows in the data. In this scenario, the XY plot will print the case labels only for those row names where the argument `clabels` has a true value, thus having a direct command over which points should be labeled.

When the argument `clabels` is logical, the case labels for the points will be taken from the row names of the data (if provided), or otherwise an automatic sequence of case numbers will be produced. This is clearly a manual operation (different from the above automatic settings), but on the other hand it gives full flexibility over all possible choices with respect to the plot. Below is a possible final example of an XY plot, using all features (Fig. 11.6):

```
sol <- compute(sol, data = CVF) # turn expression to set membership scores
col <- rep("darkgreen", nrow(CVF)) # define a vector of colors
col[sol < 0.5 & CVF$PROTEST < 0.5] <- "red" # zone 4
col[sol < 0.5 & CVF$PROTEST >= 0.5] <- "blue" # zone 5
clabels <- logical(nrow(CVF))
clabels[c(2, 5, 11)] <- TRUE # only these three labels to print
XYplot(sol, "PROTEST", data = CVF, enhance = TRUE, clabels = clabels,
        col = col, bg = col, xlab = "Solution model")
```

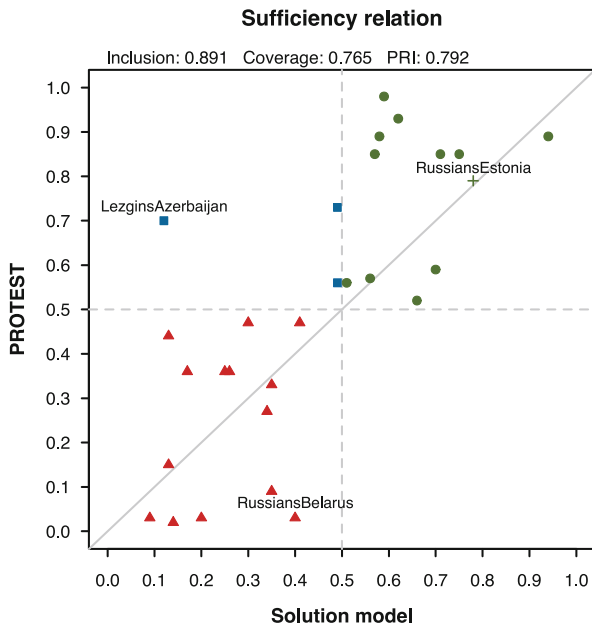


Fig. 11.6 A highly customized XY plot

Naturally, the graphical user interface has a dedicated menu to construct such diagrams:

Graphs/XY Plots

The dialog is less sophisticated, compared to the written command (especially with respect to the very latest enhancements), but on the other hand it compensates with much more interactivity. Figure 11.7 shows all possibilities, from negating conditions and outcome with a single click, changing the relationship using the dedicated radio button below the outcome selection box, to jittering points and rotating their labels to improve visibility.

In the default set up, the points are not labeled but respond with a label upon a mouse over (the point) event. The entire dialog is designed for a quick look over the possible set relations between one condition and one outcome (from the same dataset), with their associated parameters of fit.

Future versions of the dialog should include selecting minimization objects (not just dataframes), and of course enhance the plot to enable process tracing. It is also a possible idea to provide an export button to either SVG, or a bitmap version (PNG, BMP among others) or even the most portable, a PDF version of the plot.

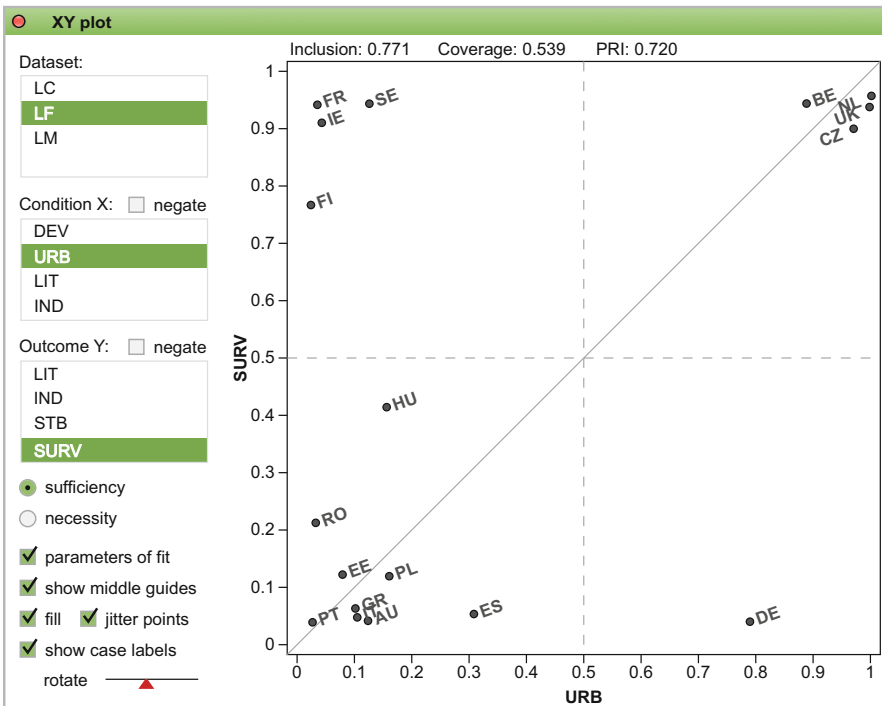


Fig. 11.7 The XY plot dialog in the graphical user interface

11.6 Venn Diagrams

The Venn diagrams are another type of useful visualization tools, particularly useful for the truth table analysis. As it turns out, there are exactly as many intersections in a Venn diagram as the number of rows in a truth table.

This is by no means a coincidence, since both truth table and Venn diagrams gravitate around the same number 2^k , where k is the number of causal conditions. Figure 11.8 is the most simple example, with two sets and $2^2 = 4$ intersections.

It should be stated, however, that truth tables can have more than 2^k rows when at least one of the causal conditions has more than two levels, therefore truth tables can be visualized with a Venn diagram only for binary crisp sets or fuzzy sets (in the latter case, the fuzzy values are transformed to binary crisp values anyways).

There are many possibilities to draw Venn diagrams in R, using various functions from packages *gplots*, *eVenn*, *Vennerable* and probably one of the best, package *VennDiagram*. An overview of such possibilities was once written by Murdoch (2004), but things have changed a lot in the mean time. A more recent presentation was written by Wilkinson (2011).

This section is going to introduce a different package with the same name, called *venn* version 1.5 (Duša 2017). Perhaps confusingly, it has exactly the same name as the one written and described by Murdoch, but his version was either not submitted to CRAN (the official R package repository), or it was abandoned and removed many years before this new package appeared.

Either way, the initial attempt by Murdoch could only draw up to three sets, and more recent packages can deal with up to five sets, while the current package *venn* can draw up to seven sets. For up to three sets, the shapes can be circular, but using circles for more than three sets is not possible. For four and five sets they shapes can be ellipsoidal, but the default shapes are even better, while for more than five sets the shapes cannot be continuous (they might be monotone, but not continuous). The seven sets diagram is called “Adelaide” (Ruskey and Weston 2005).

To complete the presentation of available R packages, it is worth mentioning *venneuler* and *eulerr* that facilitate drawing not only Venn but also Euler diagrams. While the Venn diagrams always have 2^k intersections, Euler diagrams are not bounded to this restriction and can draw sets completely outside of each other (if there is no intersection between them). Euler diagrams are not suitable for QCA research, but one of their interesting features is to allow for proportional area drawing (larger sets or larger intersections appearing larger in the plot).

```
library(venn)
venn(2, ilabels = TRUE)
```

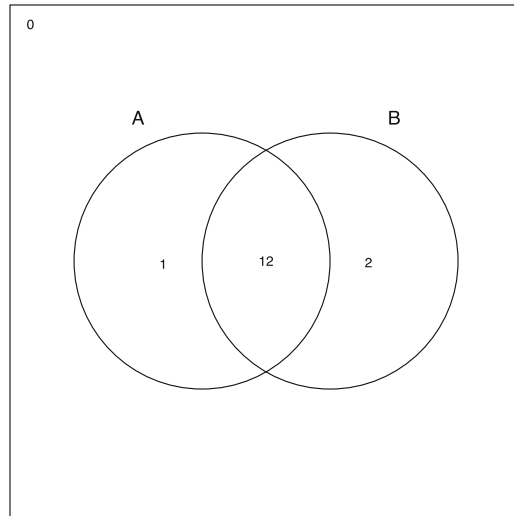


Fig. 11.8 A simple Venn diagram with two sets

The complete syntax for the function `venn()` is presented below:

```
venn(x, snames = "", ilabels = FALSE, counts = FALSE, ellipse = FALSE,
     zcolor = "bw", opacity = 0.3, size = 15, cexil = 0.6, cexsn=0.85, ...)
```

There are a number of arguments to describe, one of them being already used above. `ilabels` stands for intersection labels, and adds a unique number to each intersection, for instance 12 means the intersection between the first and the second sets, number 1 means what is unique to the first set and number 0 means what is outside all sets. When a four set diagram is going to be drawn, the intersection of all four sets will have the number 1234.

Unlike the previous approaches to draw Venn diagrams using complicated mathematical approximations to draw (proportional) intersections between sets, the package *venn* employs a static approach. All intersections between all combinations of up to seven sets are pre calculated and available with the package as a dataset containing x and y coordinates for each point that define a particular shape. This makes the drawing extremely fast, with the added advantage the predefined shapes maximize the area of each intersection to the largest extent possible.

The first argument `x` can be many things, but most importantly (and similar to the sibling package *QCA*) it can be a SOP—sum of products—expression, where intersections (conjunctions) are indicated with an asterisk “*” sign,

while unions (disjunctions) are indicated with a plus “+” sign. For this reason, it should be combined with the argument `snames` to determine the total number of sets (and their order) to be drawn in the diagram.

Like all Venn diagrams functions, `venn()` can also use colors to highlight certain intersections, and even use transparencies to show when two areas overlap (the default value of the argument `opacity` is 0.3, where more opacity means less transparency). An example could be the union of two sets A and B, from a universe containing four sets (Fig. 11.9):

```
venn("A + B", snames = "A, B, C, D", ilabels = TRUE)
```

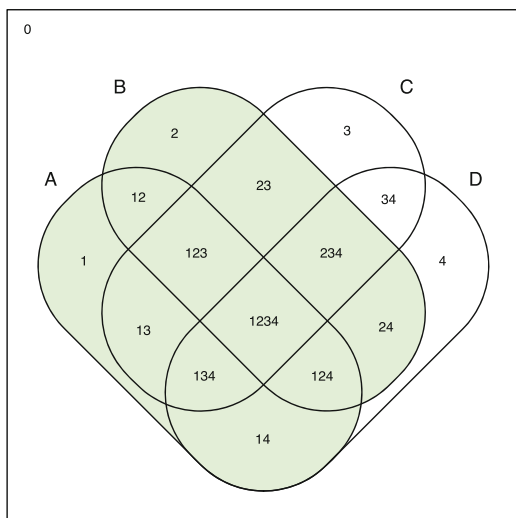


Fig. 11.9 Intersection of A and B out of four sets

The predefined color of the union can be changed using the argument `zcolor` (zone color). It is less about an intersection color, but a “zone” because in the example above the union of A and B contains multiple possible unique intersections, and the entire zone is the union of all these unique shapes.

The predefined value of the `zcolor` argument is “bw” (black and white), when no expression is given and only the shapes of the sets are drawn, as in Fig. 11.8. While `color` accepts a vector of color with the same length as the total number of sets, it has yet another predefined value called “style”, which uses a set of unique and distinct colors generated by the excellent function `colorRampPalette()` from package *grDevices*.

For four and five sets, there are alternative ellipsoidal shapes that are produced by activating the argument `ellipse`, but from six set onwards the shapes are monotone, but not continuous.

Although sets are usually understood as causal conditions in QCA (each column from a dataframe representing a set), the function `venn()` also accepts lists with unequal number of elements. Each set is represented by a list component, and the task is to calculate all possible intersections where the sets have common elements, a quite normal scenario in bioinformatics. A possible example is (Fig. 11.10):

```
set.seed(12345)
x <- list(First = 1:20, Second = 10:30, Third = sort(sample(25:50, 15)))
x

$First
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

$Second
[1] 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

$Third
[1] 25 27 28 29 31 34 35 38 40 41 43 45 46 49 50

venn(x) # argument counts is automatically activated
```

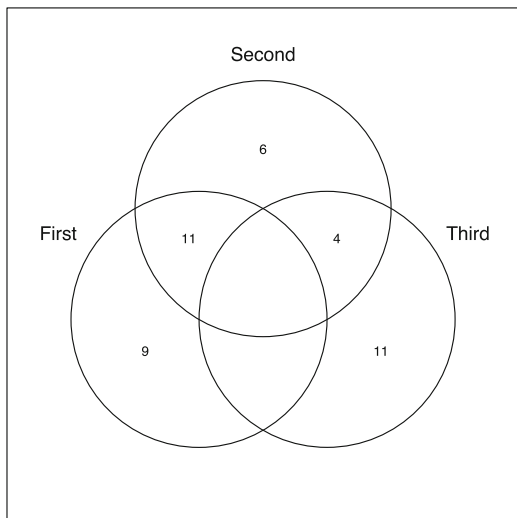


Fig. 11.10 Number of common elements in each intersection

When the input is a list, the function invisibly returns (not printed on the screen but could be assigned to an object) the truth table and the counts for each intersection, with an attribute called “intersections” containing their elements. There are 11 common elements between the First and the Second set, and 4 elements between the Second and the Third set. No common elements exist between all three sets and the number 0 is not printed in such situations

to make a difference from the value 0 from the argument `ilabels` (which means elements outside all sets).

The counts for each intersection (similar to the intersection labels) have a default, small font size, despite the large space available. That is necessary to have a uniform font size for all possible intersections, large and small. The more sets are added to the diagram, the smaller the intersections become. But this is not a restriction, since users can manually change the size of these labels using the argument `cexil`, a combination from the default `cex` argument from the base package, with `il` standing for intersection labels. There is also another related argument called `cexsn`, to adjust the font size for the set names. The base `cex` argument is used to control the width of the set border lines.

Similar to function `XYplot()` presented in the previous section, the function `venn()` has a final three dots `...` parameter to allow the usage of all other graphical parameters from the base plot functions. For instance, one could customize a plot by changing the line type via the parameter `lty`, as well as its color via the parameter `col` (Fig. 11.11):

```
venn(5, lty = 5, col = "navyblue", ilabels = TRUE)
```

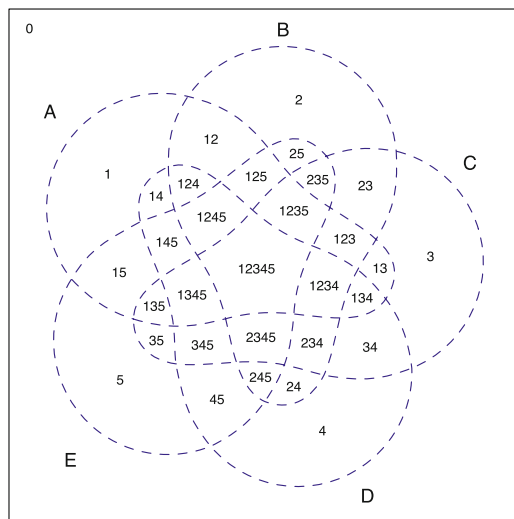


Fig. 11.11 Custom Venn diagram with five sets

This is one of the best shapes that accommodates five sets into a single diagram. It is also possible to use an ellipse as a base shape, obtaining some of the intersections very small in size, while some are very large. The light bulb shape is unique and currently offered only by package `venn`, enlarging the size of all intersections to the maximum extent possible.

All of these diagrams look very nice, but the most important feature of the function `venn()` is its ability to recognize a truth table object and plot all of its intersections according to the output value. Using the object `ttCVF` produced in the previous section (a truth table on the CVF data, using a 0.85 inclusion cut-off), the command to plot this object is as simple as:

```
venn(ttCVF, counts = TRUE, opacity = ttCVF$tt$incl)
```

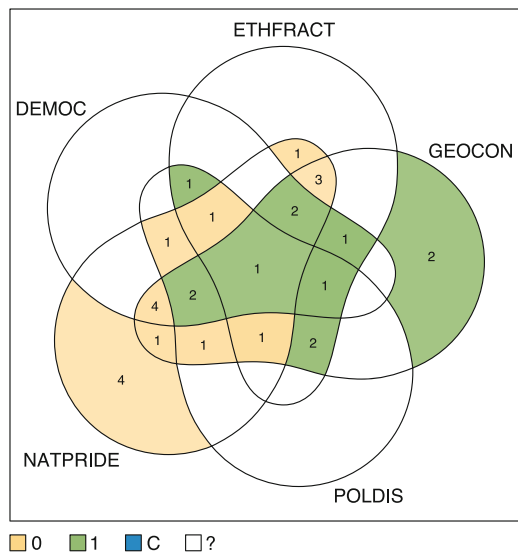


Fig. 11.12 Venn diagram for the CVF truth table

When plotting a truth table, the choice of colors is fixed, to have an easily recognizable standard of the colors associated with the positive output configurations (green), negative output configurations (orange), contradictions (blue) and the rest of the intersections with no empirical evidence (remainders, white). The legend explaining these colors is drawn at the bottom.

The argument `opacity` is optional, but it can use the inclusion scores for each configuration to draw the colors, which means a high inclusion turns the color for a particular intersection more opaque (less transparent). This means of course that positive output configurations will always be more opaque, and more significant differences are noticeable only between the yellow intersections for the negative output configurations, where the range of inclusion scores is larger.

In Fig. 11.12, many of the intersections from set NATPRIDE are associated with the negative output, while many of the intersections from set GEOCON are associated with the positive output. In the middle part, there are intersections (configurations) belonging to multiple sets that are associated with either one of the outputs.

The counts from each intersection represent the number of cases associated with each intersection, but they are not important when deriving the final solution. It is the configuration itself, not the number of cases associated with it, that has a contribution on the final solution(s), as long as the configuration passes the frequency cut-off. However, this is a useful information to visualize how the empirical information from the database is distributed in the truth table configurations. The minimized solutions are:

```
pCVF <- minimize(ttCVF, include = "?")
pCVF
```

```
M1: natpride + DEMOC*GEOCON*POLDIS + (DEMO*ETHFRACT*GEOCON)
    => PROTEST
M2: natpride + DEMOC*GEOCON*POLDIS + (DEMO*ETHFRACT*poldis)
    => PROTEST
```

In a similar vein with the function `XYplot()`, and consistent with a general approach in package *QCA*, the function `venn()` is built to automatically recognize a solution if it comes from a minimization object (such as `pCVF`). As a consequence, it is not needed to specify the set names because they are taken directly from the minimization object.

```
venn(pCVF$solution[1], zcol = "#ffdd77, #bb2020, #1188cc")
```

In Fig. 11.13, the first term of the solution (`natpride`, absence of national pride, the area outside of the set `NATPRIDE`) is drawn using a pale green color, the intersection `DEMO*GEOCON*POLDIS` in red and `DEMO*ETHFRACT*GEOCON` in blue. In principle, the information from this diagram confirms the previous one, where `NATPRIDE` was mainly associated with a negative output (here,

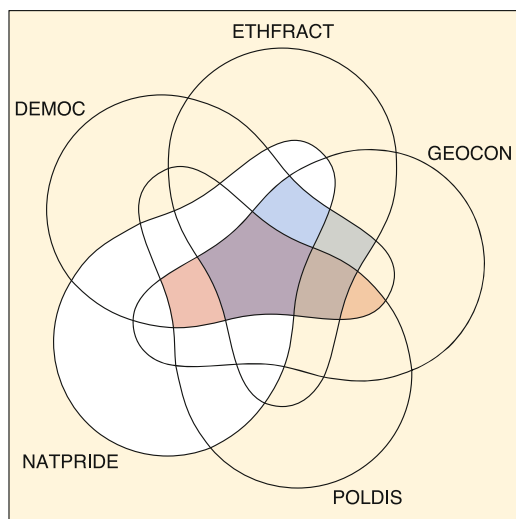


Fig. 11.13 Venn diagram for the first solution

the absence of NATPRIDE is sufficient for the presence of the outcome) and GEOCON is present in both the other conjunctions from the solution. There is one particular area where all colors overlap, at the intersection between the first four sets, and outside the fifth.

In the graphical user interface, a Venn diagram can be produced via the following menu:

Graphs/Venn Diagram

In the absence of a truth table object, this menu will open an empty dialog. It first starts by looking for such objects in the workspace, and creates a diagram only if they exist (from the truth table dialog, the option **Assign** has to be checked, with a suitable name). If multiple such objects exist, the Venn diagram will be drawn for the latest one. It is also worth mentioning that truth tables are also produced automatically by the minimization function (objects of class "qca" contain truth table objects of class "tt"), therefore they all count for the Venn diagram dialog.

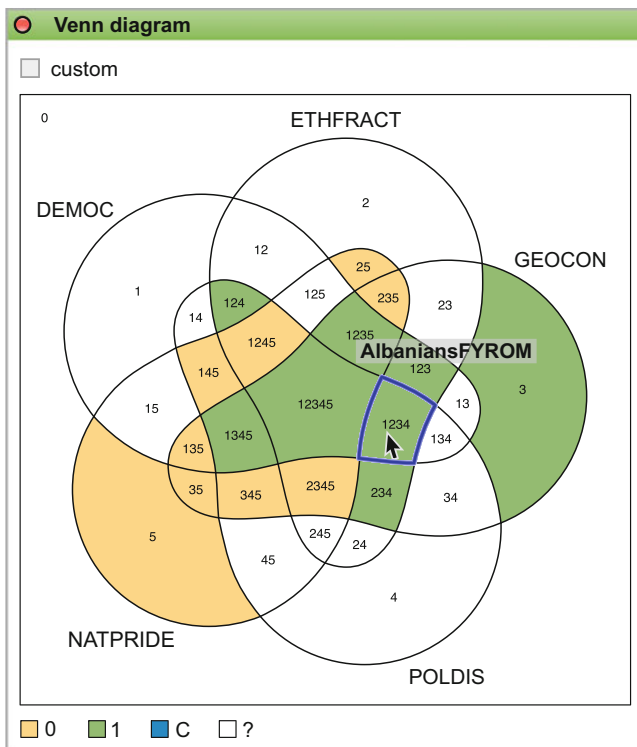


Fig. 11.14 The Venn diagram dialog in the graphical user interface

Figure 11.14 above is very similar to the previous Fig. 11.12, just produced in the graphical user interface. It has the intersection labels instead of the counts, but the most important difference is the ability to interactively explore what cases are associated with each intersection. On hovering the mouse over the intersections, an event is triggered to show a message containing the names of each associated cases. Such an event is naturally impossible in the normal R graphical window, and demonstrates the complementary features that HTML based, Javascript events can bring to enhance the experience of normal R graphics.

The particular region that is hovered, at the intersection between the first four sets, is also the intersection where all three terms from the first solution overlap in Fig. 11.13. Perhaps interesting is the case associated with this intersection (AlbaniansFYROM), which is the same case identified as the most typical case in the enhanced XY plot from Fig. 11.4. Further research is needed to confirm if this is a fact or just a coincidence, but the information seems to converge.

11.7 Custom Labels

The function `venn()` has a predefined set of labels for the intersections, either as numbers representing the sets to which an intersection belongs to, or counting how many cases belong to a certain intersection (configuration).

There are however situations when users need to add custom labels to various intersections, either with the names of the cases belonging to a specific configuration, or any other text description. In order to do that, it is important to understand the conceptual difference between an intersection, a “zone”, and an “area”.

A *zone* is a union of set intersections. If a zone contains all intersections from a particular set, the zone would be equivalent to the set itself. For instance in Fig. 11.8, the set A has two intersections: **Ab** (what is inside of A but not in B), and **AB** (the intersection between A and B). We could also say the set A is the union between **Ab** and **AB** (in other words, $\text{Ab} + \text{AB} = \text{A}$).

An *area* can have one or multiple zones, depending on the complexity of the Venn diagram, with more sets increasing the complexity. For instance there are four sets in Fig. 11.9, and the area **Bc** (what is inside B but not in C), has two zones: the first consists from the intersections 2 and 12 ($\text{aBcd} + \text{ABcd}$), and the second zone with the intersections 24 and 124 ($\text{aBcD} + \text{ABcD}$). This happens because the set B is transversally sectioned by the set C. An area can also be specified by an entire solution, which by definition has multiple zones, for each solution term.

Figure 11.13 could be improved by adding a label containing the cases for the solution term `DEMOC*ETHFRACT*GEOCON`, which is the second term from the first solution of the object `pCVF`. Such a label should be located in the diagram using a set of coordinates for the X and Y axes, therefore we need to calculate these coordinates:

```
coords <- unlist(getCentroid(getZones(pCVF$solution[[1]][2])))
```

This is facilitated by the function `getCentroid()` from package `venn`, which returns a list of coordinates for each zone determined by the function `getZones()`. In this example, there is a single zone for this particular solution term, and the result is unlisted to obtain a vector of two numbers for the coordinates.

From the inclusion and coverage scores table, we can see the cases associated with this term: "HungariansRom", "CatholicsNIreland", "AlbaniansFYROM" and "RussiansEstonia". Having the coordinates of the centroid, adding the label is now a simple matter of:

```
venn(pCVF$solution[1], zcol = "#ffdd77, #bb2020, #1188cc")
cases <- paste(c("HungariansRom", "CatholicsNIreland", "AlbaniansFYROM",
                "RussiansEstonia"), collapse = "\n")
text(coords[1], coords[2], labels = cases, cex = 0.85)
```

The specification of `collapse = "\n"` in function `paste()` above makes sure that all four cases are printed below each other (the string `"\n"` is interpreted similarly to pressing the Enter key after each case name). The argument `cex` adjusts the font size of the text in the label, according to circumstances (Fig. 11.15).

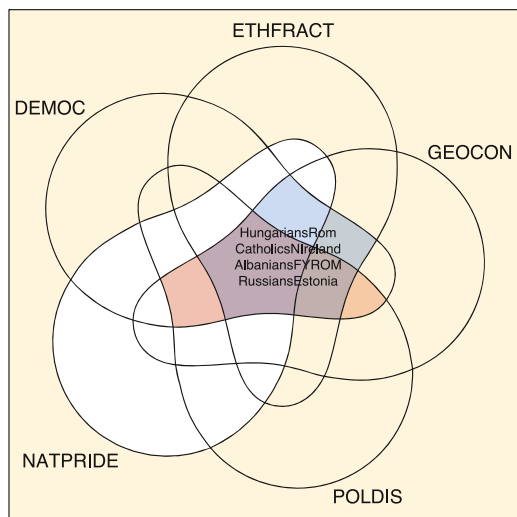


Fig. 11.15 Custom labels for a solution term

References

- Aiton EJ (1985) Leibniz: a biography. Hilger, London
- Ambuehl M, Baumgartner M (2017) CNA: causal modeling with coincidence analysis. R package version 2.0.0. <https://CRAN.R-project.org/package=cna>
- Barrenechea R, Mahoney J (2017) A set-theoretic approach to Bayesian process tracing. *Sociol Methods Res.* <https://doi.org/10.1177/0049124117701489>
- Baumgartner M (2009) Inferring causal complexity. *Sociol Methods Res* 38(1):71–101. <https://doi.org/10.1177/0049124109339369>
- Baumgartner M (2013) Detecting causal chains in small-n data. *Field Methods* 25(1):3–24. <https://doi.org/10.1177/1525822X12462527>
- Baumgartner M (2015) Parsimony and causality. *Qual Quant* 49:839–856. <https://doi.org/10.1007/s11135-014-0026-7>
- Baumgartner M, Thiem A (2017) Often trusted but never (properly) tested: evaluating qualitative comparative analysis. *Sociol Methods Res.* <https://doi.org/10.1007/s11229-008-9348-0>
- Bennett B (2012) Logically fallacious. The ultimate collection of over 300 logical fallacies. eBookit.com, Sudbury. <https://www.logicallyfallacious.com>
- Berg-Schlosser D (2012) Mixed methods in comparative politics: principles and applications. Palgrave Macmillan, Basingstoke
- Bogin B (1998) The tall and the short of it. *Discover* 19(2):40–44
- Bolton-Smith C, Woodward M, Tunstall-Pedo H, Morrison C (2000) Accuracy of the estimated prevalence of obesity from self reported height and weight in an adult Scottish population. *J Epidemiol Community Health* 54:143–148
- Boole G (1854) An investigation of the laws of thought. Walton & Maberly, London. Available online, April 2017: <http://www.gutenberg.org/files/15114/15114-pdf.pdf>
- Borkowski L (ed) (1970) Jan Łukasiewicz: selected works. North Holland, Amsterdam

- Braumoeller B (2015a) QCAfalsePositive: tests for type I error in qualitative comparative analysis (QCA). R package version 1.1.1. <https://CRAN.R-project.org/package=QCAfalsePositive>
- Braumoeller BF (2015b) Guarding against false positives in qualitative comparative analysis. *Polit Anal* 23(4):471–487. <https://doi.org/10.1093/pan/mpv017>
- Braumoeller B, Goertz G (2000) The methodology of necessary conditions. *Am J Polit Sci* 44(4):844–858. <https://doi.org/10.2307/2669285>
- Caren N, Panofsky A (2005) TQCA. A technique for adding temporality to qualitative comparative analysis. *Sociol Methods Res* 34(2):147–172. <https://doi.org/10.1177/0049124105277197>
- Cebotari V, Vink MP (2013) A configurational analysis of ethnic protest in Europe. *Int J Comp Sociol* 54(4):298–324
- Cheli B, Lemmi A (1995) A “totally” fuzzy and relative approach to the multidimensional analysis of poverty. *Econ Notes* 1:115–134
- Cronqvist L, Berg-Schlusser D (2009) Multi-value QCA (mvQCA). In: Rihoux B, Ragin C (eds) *Configurational comparative methods: qualitative comparative analysis (QCA) and related techniques*. Sage, London, pp 69–86
- Dalgaard P (2008) *Introductory statistics with R*, 2nd edn. Springer, New York
- Dauben JW (1979) *Georg cantor. His mathematics and philosophy of the infinite*. Princeton University Press, Princeton
- De Meur G, Rihoux B, Yamasaki S (2009) Addressing the critiques of QCA. In: Rihoux B, Ragin C (eds) *Configurational comparative methods: qualitative comparative analysis (QCA) and related techniques*. Sage, London, pp 147–165
- Duša A (2007) Enhancing Quine-McCluskey. COMPASSS Working Paper. <http://www.compass.org/wpseries/Dusa2007b.pdf>
- Duša A (2010) A mathematical approach to the Boolean minimization problem. *Qual Quant* 44:99–113. <https://doi.org/10.1007/s11135-008-9183-x>
- Duša A (2017) Venn: draw venn diagrams. R package version 1.5. <https://CRAN.R-project.org/package=venn>
- Duša A, Thiem A (2015) Enhancing the minimization of Boolean and multi-value output functions with eQMC. *J Math Sociol* 39:92–108. <https://doi.org/10.1080/0022250X.2014.897949>
- Emmenegger P, Schraff D, Walter A (2014) QCA, the truth table analysis and large-N survey data: the benefits of calibration and the importance of robustness tests. COMPASSS Working Paper. <http://www.compass.org/wpseries/EmmeneggerSchraffWalter2014.pdf>
- Fox J (2005) The R commander: a basic statistics graphical user interface to R. *J Stat Softw* 14(9):1–42. <http://www.jstatsoft.org/v14/i09>
- Garcia-Castro R, Ariño MA (2016) A general approach to panel data set-theoretic research. *J Adv Manag Sci Inf Syst* 2:63–76
- Gibson BC, Burrell V (2017) braQCA: bootstrapped robustness assessment for qualitative comparative analysis. R package version 0.9.9.6. <https://CRAN.R-project.org/package=braQCA>

- Goertz G (2003) Cause, correlation and necessary conditions. In: Goertz G, Starr H (eds) *Necessary conditions: theory, methodology, and applications*. Rowman & Littlefield, Lanham, pp 47–64
- Goertz G (2006a) Assessing the trivialness, relevance, and relative importance of necessary or sufficient conditions in social science. *Stud Comp Int Dev* 41(2):88–109. <https://doi.org/10.1007/BF02686312>
- Goertz G (2006b) *Social science concepts. A user's guide*. Princeton University Press, Princeton
- Haesebrouck T (2015) Pitfalls in QCA's consistency measure. *J Comp Polit* 2:65–80. Handle: 1854/LU-6834276
- Hak T, Jaspers F, Dul J (2013) The analysis of temporarily ordered configurations: challenges and solutions. In: Fiss PC, Cambré B, Marx A (eds) *Configurational theory and methods in organizational research*. Emerald Group Publishing, Bingley, pp 107–127
- Hino A (2009) Time-series QCA: studying temporal change through Boolean analysis. *Sociol Theory Methods* 24(2):247–265
- Hug S (2013) Qualitative comparative analysis: how inductive use and measurement error lead to problematic inference. *Polit Anal* 21(2):252–265. <https://doi.org/10.1093/pan/mps061>
- Hume D (1999) An enquiry concerning human understanding. In: Beauchamp T (ed) *Oxford philosophical texts*. Oxford University Press, Oxford
- King G, Keohane RO, Verba S (1994) *Designing social inquiry. Scientific inference in qualitative research*. Princeton University Press, Princeton
- Krogslund C, Choi DD, Poertner M (2015) Fuzzy sets on shaky ground: parameter sensitivity and confirmation bias in fsQCA. *Polit Anal* 23(1):21–41. <https://doi.org/10.1093/pan/mpu016>
- Krook ML (2010) Women's representation in parliament: a qualitative comparative analysis. *Polit Stud* 58(5):886–908
- Lazarsfeld P (1937) Some remarks on typological procedures in social research. *Z Sozial* 6:1–24. Available on July 2017: <http://dspace.gipe.ac.in/xmlui/handle/10973/33569>
- Lipset MS (1959) Some social requisites of democracy: economic development and political legitimacy. *Am Polit Sci Rev* 53(1):69–105
- Lucas SR (2014) Rejoinder: taking heat and giving light—reflections on the early reception of “qualitative comparative analysis in critical perspective”. *Sociol Methodol* 44(1):127–158. <https://doi.org/10.1177/0081175014544105>
- Lucas SR, Sztatowski A (2014) Qualitative comparative analysis in critical perspective. *Sociol Methodol* 44(1):1–79. <https://doi.org/10.1177/0081175014532763>
- Mahoney J, Kimball E, Koivu KL (2009) The logic of historical explanation in the social sciences. *Comp Polit Stud* 42(1):114–146. <https://doi.org/10.1177/0010414008325433>
- Marx A, Duşa A (2011) Crisp-set qualitative comparative analysis (csQCA), contradictions and consistency benchmarks for model specification. *Methodol Innov Online* 6(2):103–148. <https://doi.org/10.4256/mio.2010.0037>

- Marx A, Rihoux B, Ragin C (2014) The origins, development, and application of qualitative comparative analysis: the first 25 years. *Eur Polit Sci Rev* 6(1):115–142. <https://doi.org/10.1017/S1755773912000318>
- McCluskey EJ (1956) Minimization of Boolean functions. *Bell Syst Tech J* 5:1417–1444
- Medzihorsky J, Oana IE, Quaranta M, Schneider CQ (2017) SetMethods: functions for set-theoretic multi-method research and advanced QCA. R package version 2.1. <https://CRAN.R-project.org/package=SetMethods>
- Murdoch DJ (2004) Venn diagrams in R. *J Stat Softw* 11(1):1–3. <https://doi.org/10.18637/jss.v011.c01>, <https://www.jstatsoft.org/v011/c01>
- Murrell P (2006) R graphics. Chapman & Hall, Boca Raton
- Neuman LW (2003) Social research methods: qualitative and quantitative approaches, 5th edn. Allyn Bacon, Boston
- Persico N, Postlewaite A, Silverman D (2004) The effect of adolescent experience on labor market outcomes: the case of height. *J Polit Econ* 112(5): 1019–1053
- Quine WVO (1952) The problem of simplifying truth functions. *Am Math Mon* 59(8):521–531
- Quine WVO (1955) A way to simplify truth functions. *Am Math Mon* 62(9):627–631
- Ragin C (1987) The comparative method. Moving beyond qualitative and quantitative strategies. University of California Press, Berkeley
- Ragin C (2000) Fuzzy set social science. University of Chicago Press, Chicago
- Ragin C (2005) From fuzzy sets to crisp truth tables. COMPASS Working Paper. <http://www.compass.org/wpseries/Ragin2004.pdf>
- Ragin C (2006) User's guide to fuzzy-set/qualitative comparative analysis 2.0. Tucson, Arizona
- Ragin C (2008a) Measurement versus calibration: a set theoretic approach. In: Box-Steffensmeier J, Brady HE, Collier D (eds) *The Oxford handbook of political methodology*. Oxford University Press, Oxford, pp 174–198
- Ragin C (2008b) Redesigning social inquiry. Fuzzy sets and beyond. University of Chicago Press, Chicago
- Ragin C (2014) Lucas and Szatrowski in critical perspective. *Sociol Methodol* 44(1):80–94. <https://doi.org/10.1177/0081175014542081>
- Ragin C, Rihoux B (2004) Qualitative comparative analysis (QCA): state of the art and prospects. *Qual Methods* 2:3–13
- Ragin C, Sonnett J (2005) Between complexity and parsimony: limited diversity, counterfactual cases, and comparative analysis. In: Kropp S, Minkenberg M (eds) *Vergleichen in der Politikwissenschaft*. VS Verlag für Sozialwissenschaften, Wiesbaden, pp 180–197. <https://doi.org/10.1007/978-3-322-80441-9>
- Ragin C, Sonnett J (2008) Limited diversity and counterfactual cases. In: Ragin C (ed) *Redesigning social inquiry. Fuzzy sets and beyond*. University of Chicago Press, Chicago, pp 147–159

- Ragin C, Strand SI (2008) Using qualitative comparative analysis to study causal order. Comment on Caren and Panofsky. *Sociol Methods Res* 36(4):431–441. <https://doi.org/10.1177/0049124107313903>
- Rihoux B, De Meur G (2009) Crisp-set qualitative comparative analysis (csQCA). In: Rihoux B, Ragin C (eds) *Configurational comparative methods: qualitative comparative analysis (QCA) and related techniques*. Sage, London, pp 33–68
- Royston P, Altman DG (1994) Regression using fractional polynomials of continuous covariates: parsimonious parametric modelling. *J R Stat Soc Ser C* 43(3):429–467
- RStudio, Inc. (2013) Easy web applications in R. <http://www.rstudio.com/shiny/>
- Ruskey F, Weston M (2005) Venn diagrams. *Electron J Comb Dyn Surv DS5*. <http://www.combinatorics.org/ojs/index.php/eljc/article/view/DS5/html>
- Sauerbrei W, Royston P (1999) Building multivariable prognostic and diagnostic models: transformation of the predictors by using fractional polynomials. *J R Stat Soc Ser A* 162(1):71–94
- Schneider CQ, Rohlfing I (2013) Combining QCA and process tracing in set-theoretic multi-method research. *Sociol Methods Res* 42(4):559–597. <https://doi.org/10.1177/0049124113481341>
- Schneider CQ, Rohlfing I (2016) Case studies nested in fuzzy-set QCA on sufficiency: formalizing case selection and causal inference. *Sociol Methods Res* 45(3):536–568. <https://doi.org/10.1177/0049124114532446>
- Schneider C, Wagemann C (2012) *Set-theoretic methods for the social sciences. A guide to qualitative comparative analysis*. Cambridge University Press, Cambridge
- Schneider CQ, Wagemann C (2013) Doing justice to logical remainders in QCA: moving beyond the standard analysis. *Polit Res Q* 66(1):211–220. <https://doi.org/10.1177/1065912912468269h>
- Schneider MR, Schulze-Bentrop C, Paunescu M (2010) Mapping the institutional capital of high-tech firms: a fuzzy-set analysis of capitalist variety and export performance. *J Int Bus Stud* 41:246–266
- Seawright J (2014) Comment: limited diversity and the unreliability of QCA. *Sociol Methodol* 44(1):118–121. <https://doi.org/10.1177/0081175014542082>
- Shannon CE (1940) A symbolic analysis of relay and switching circuits. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering. Available on, July 2017: <https://dspace.mit.edu/handle/1721.1/11173>
- Skaaning SE (2011) Assessing the robustness of crisp-set and fuzzy-set QCA results. *Sociol Methods Res* 40(2):391–408. <https://doi.org/10.1177/0049124111404818>
- Skocpol T (1979) *States and social revolutions. A comparative analysis of France, Russia, and China*. Cambridge University Press, Cambridge
- Smithson M, Verkuilen J (2006) *Fuzzy set theory. Applications in the social sciences*. Sage, Thousand Oaks

- Spirtes P, Glymour C, Scheines R (2000) Causation, prediction, and search, 2nd edn. MIT Press, Cambridge
- Thiem A (2014) Membership function sensitivity of descriptive statistics in fuzzy-set relations. *Int J Soc Res Methodol* 17(6):625–642
- Thiem A (2016) Standards of good practice and the methodology of necessary conditions in qualitative comparative analysis. *Polit Anal* 24(4):478–484
- Thiem A, Duşa A (2013) Qualitative comparative analysis with R. A user's guide. Springer, New York
- Thiem A, Spöhel R, Duşa A (2016) Enhancing sensitivity diagnostics for qualitative comparative analysis: a combinatorial approach. *Polit Anal* 24:104–120. <https://doi.org/10.1093/pan/mpv028>
- Thompson SL (2011) The problem of limited diversity in qualitative comparative analysis: a discussion of two proposed solutions. *Int J Mult Res Approaches* 5(2):254–268. <https://doi.org/10.5172/mra.2011.5.2.254>
- Valero-Mora PM, Ledesma RD (2012) Graphical user interfaces for R. *J Stat Softw* 49(1):1–8
- Verkuilen J (2005) Assigning membership in a fuzzy set analysis. *Social Methods Res* 33(4):462–496. <https://doi.org/10.1177/0049124105274498>
- Verzani J (2005) Using R for introductory statistics, 2nd edn. Chapman & Hall, Boca Raton
- Weber M (1930) The protestant ethic and the spirit of capitalism. Routledge, London
- Wilkinson L (2011) Exact and approximate area-proportional circular venn and Euler diagrams. *IEEE Trans Vis Comput Graph* 18(2):321–331. <https://doi.org/10.1109/TVCG.2011.56>
- Yamasaki S, Rihoux B (2009) A commented review of applications. In: Rihoux B, Ragin C (eds) *Configurational comparative methods: qualitative comparative analysis (QCA) and related techniques*. Sage, London, pp 123–145
- Zadeh LA (1965) Fuzzy sets. *Inf Control* 8:338–353
- Zuur A, Ieno E, Meesters E (2009) A beginner's guide to R. Springer, New York