QCA and R

Thomas Jensen

Contents

1 Installing all of the Components

1.1 Getting R

Go to

```
http://cran.r-project.org
```

there you will find the latest pre-compiled versions of R for Windows, MAC and Linux. To install simply double click the file.

1.2 Getting QCA3

Once you have installed R you need to install the QCA3 package. First open R, then type

```
> install.packages("QCA3", dependencies = TRUE)
```

in the command prompt. Or chose Packages & Date in the menu bar, then Package Installer. Type in QCA3 in the search interface and press Get List. Choose QCA3 from the list and click Install Selected.

To check that QCA3 is installed properly, load the package and see if you get any error messages

```
> library(QCA3)
```

If you get any error messages regarding other packages, then you will have to install them as well.

1.3 Getting a Good Editor for R

There are several good editors for R. The classic editors are EMACS and VIM. These are general editors that support practically all programming languages and have several very advanced features. However both of these editors have a very steep learning curve. A two good general purpose editors with good support for R are Notepad++ and Tinn-R. Both of these editors allow you to write your code with proper syntax highlighting and standard features such as search, replace and undo/redo. Furthermore both editors allows you to execute your code straight from the editor.

R Editors
EMACS: `http://www.gnu.org/software/emacs/`

VIM: `http://www.vim.org/`

Notepad++: `http://notepad-plus-plus.org/`

Tinn-R: `http://www.sciviews.org/Tinn-R/`

Note that in order to get Notepad++ to run your R script, you need to install NppToR (`http://sourceforge.net/projects/npptor/`). After installing you can run NppToR and it will launch in your taskbar. In order to run your script in Notepad++ simply select the commands that should be run and hit F8.

2 Basics of Using R

R is a high-level language and an environment for data analysis and graphics. The first sight that will meet you when starting R is a white screen with a `>`. This is the command prompt, and it is here that we feed instructions to R. Since R is a language there are some conventions that you should be aware of from the start. First R is case sensitive, `QCA3` is interpreted differently than `qca3`. Second all R commands are separated by either a `;` or a new line. If a command is not complete you will see a `+` in the prompt. To break out of this type `CTRL + c` (press the control key and c at the same time). R remembers all the lines you have feed to it during your session, to see the previous or next lines simply use the up and down arrow keys.

The most important aspect of the R language is the it is object based. This means that every piece of information can be stored in an object, that can then be called from the R environment. To store data in an object we use the `<-` operator. This is best illustrated with an example

```
> Integer <- 4
```

Here we have store the integer 4 in an object labelled `integer`. To see the content of an object we simply type the name of the object in the command

prompt

```
> Integer
[1] 4
```

This returns the value 4. Although this is a very simple illustration the basic logic is the same even with very complicated structures such as data sets (as we will se later).

2.1 Getting Help in R

If you get stuck with a particular function in R there is an extensive library of documentation available. simply type `help()` followed by the command that bothers you. For example

```
> help(lm)
```

will prompt R to open the documentation for linear modeling. If you cannot remember the precise name of the function that you want help for type

```
> help.search("data input")
```

and with luck you will see the names of the R functions associated with your query. There are tons of information on R online, and often google is your best friend. However on the CRAN website they also have a host of information and guides.

3 Getting Started with R

3.1 Setting Up R

The first step is to define the working directory. This is the place that holds all the data and files which you will be using during your R session. Unless told otherwise, R will also save all the output you request (graphs, charts etc.) to this directory. To define the working directory we use the command `setwd()`.

```
> setwd("T:\\Thomas_ETH\\workspace\\QCA_R_Guide")
```

Notice that we have written the path inside two " ", this is to tell R that this is a text string, and not an object within the R environment.

3.2 Reading Data Into R

The first step of doing an analysis in R is to read the data into R in the correct format.This is one aspect of R that often frustrates people beginning with R. The basic command for reading data in simple .txt files into R is `read.table()`. To illustrate how this command works consider the following example

```
> data <- read.table("Winzen_raw.txt")
> data
                V1      V2   V3    V4    V5 V6
1           caseid support parl trust gains ID
2          Austria      44 0.67    49    46 46
3          Belgium      58 0.17    44    56 45
4          Denmark      62 0.83    58    72 41
5            Spain      57 0.17    45    60 38
6          Finland      37 0.83    46    39 59
7           France      50 0.33    38    53 34
8          Germany      55  0.5    40    44 40
9           Greece      68 0.17    43    78 58
10         Ireland      81 0.17    47    90 55
11           Italy      64 0.17    31    57 31
12      Luxembourg      81 0.17    59    71 22
13     Netherlands      74  0.5    58    67 45
14        Portugal      63 0.17    45    73 51
15          Sweden      43 0.67    49    31 50
16 United Kingdom      33 0.33    38    36 71
```

Here we have read a data set collected by a colleague. which has values on the country level for EU15. However this does not look correct. The reason is that we have not told R how the .txt file is structured. In every .txt file there is some basic information that R needs in order to read it properly. First R needs to know what character was used to separate the entries in the file, second R needs to know if the first row in the file contains the names of the variables/sets. Third we can tell R to use any column in the .txt file that might contain names of the observations as row identifiers.

```
> data <- read.table("Winzen_raw.txt", sep = "\t", header = TRUE,
     row.names = 1)
```

```
> data
                support parl trust gains ID
Austria              44 0.67    49    46 46
Belgium              58 0.17    44    56 45
Denmark              62 0.83    58    72 41
Spain                57 0.17    45    60 38
Finland              37 0.83    46    39 59
France               50 0.33    38    53 34
Germany              55 0.50    40    44 40
Greece               68 0.17    43    78 58
Ireland              81 0.17    47    90 55
Italy                64 0.17    31    57 31
Luxembourg           81 0.17    59    71 22
Netherlands          74 0.50    58    67 45
Portugal             63 0.17    45    73 51
Sweden               43 0.67    49    31 50
United Kingdom       33 0.33    38    36 71
```

The `sep` option specifies the character that separates the entries, in our case we have tab-delimited file, thus we use the `sep="\t"` to specify that the file is tab-delimited. If the entries were separated by a space we would use `sep=" "`, and if a comma was used to separate the entries we would use `sep=","`. The `header` option tells R if the first row of the file contains the names of the variables/sets, this is a binary option that only takes two arguments, namely `TRUE` or `FALSE`. By default this is set to `FALSE` by R. Finally the `row.names` option tells R if there is a column with row names, and what number that column has, usually it is the first column in which case we write `row.names=1`.

If a data set has many observations it is not very helpful to simple list the data as it is, rather we would like to get some summary statistics on each variable in the data set. This is accomplished with the `summary` command.

```
> summary(data)
    support        parl           trust          gains
 Min.   :33   Min.   :0.170   Min.   :31.0   Min.   :31.0
 1st Qu.:47   1st Qu.:0.170   1st Qu.:41.5   1st Qu.:45.0
 Median :58   Median :0.330   Median :45.0   Median :57.0
```

```
Mean   :58   Mean   :0.390   Mean   :46.0   Mean   :58.2
3rd Qu.:66   3rd Qu.:0.585   3rd Qu.:49.0   3rd Qu.:71.5
Max.   :81   Max.   :0.830   Max.   :59.0   Max.   :90.0
      ID
Min.   :22.00
1st Qu.:39.00
Median :45.00
Mean   :45.73
3rd Qu.:53.00
Max.   :71.00
```

as you can see the `summary` command gives us the maximum, minimum, median, mean and the 25th and 75th quartiles of a variable.

To read in data from other statistical packages such as SPSS or STATA, we need some additional help. The package `foreign` has been written for R to allow us to do this. This package comes with separate commands for each data type. To read in a STATA file we use the command `read.dta()`, and to read in an SPSS file we use the commands `read.spss()`.

```
> library(foreign)
> data <- read.spss("<your_data.sav>")
> data <- read.dta("<your_data.dta>")
```

4 Manipulating Data in R

Most of the time you will be dealing with data sets where you are only interested in one or two variables. Since most data sets are huge it is often useful to extract the variables you are interested in. To view just a single variable from a data set we write:

```
> data$trust
 [1] 49 44 58 45 46 38 40 43 47 31 59 58 45 49 38
```

To get a quick summary of a variable we write:

```
> summary(data$trust)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   31.0    41.5    45.0    46.0    49.0    59.0
```

To extract several variables from a data set we proceed as follows. Remember that R is an object oriented language, thus we can simply create a new object that contains the data we are interested in. Using the Winzen data as an example we can create a new data frame with a subset of the variables as follows:

```
> new_data <- data.frame(data$trust, data$parl, data$ID)
> colnames(new_data) <- c("trust", "parl", "ID")
> new_data
   trust parl ID
1     49 0.67 46
2     44 0.17 45
3     58 0.83 41
4     45 0.17 38
5     46 0.83 59
6     38 0.33 34
7     40 0.50 40
8     43 0.17 58
9     47 0.17 55
10    31 0.17 31
11    59 0.17 22
12    58 0.50 45
13    45 0.17 51
14    49 0.67 50
15    38 0.33 71
```

The `data.frame()` command tells R to create a new data frame with the chosen variables, this is then stored in the `new_data` object. The `colnames()` command lets us name the variables we extract. Notice that we use `c()` to create a list of variable names. The list must name the variables in the correct order in which they appear in the new data frame.

Often when creating subsets of data we end up with missing values for a series of observations. Missing values are not allowed when doing QCA thus we need a way to deal with this. The most straightforward and brute approach is

to simple exclude any row with missing values, this is done with the `na.omit()` command.

```
> new_data <- na.omit(new_data)
```

However often it is sensible to see if we cannot somehow find information on the missing values, or impute them somehow. For large data sets this is often very difficult from a QCA perspective, but for small data set with perhaps only 20 - 30 cases this might be feasible.

4.1 Combining Data From Two Datasets

Lets read in a new data set. The `CPI_2005` data set contains the perceived corruption level index for 2005. In order to be able to merge the two data sets we read in the Winzen data again, but this time we leave the `caseid` in the data set. This is because we want to use this as an identifier variable, as the `CPI_2005` data also has a similar country label variable.

```
> data <- read.table("Winzen_raw.txt", sep = "\t", header = TRUE)
> data.2 <- read.table("CPI_2005.txt", sep = "\t", header = TRUE)
> summary(data.2)
         caseid         cpi
 Afghanistan:  1   Min.   :1.700
 Albania    :  1   1st Qu.:2.500
 Algeria    :  1   Median :3.200
 Angola     :  1   Mean   :4.078
 Argentina  :  1   3rd Qu.:5.000
 Armenia    :  1   Max.   :9.700
 (Other)    :153
```

Notice that the `CPI_2005` has many more observations than the Winzen data set. To merge the two data sets we use the `merge()` command

```
> merged.data <- merge(data, data.2, by = "caseid", all.x = TRUE)
> merged.data
           caseid support parl trust gains ID cpi
1         Austria      44 0.67    49    46 46 8.7
2         Belgium      58 0.17    44    56 45 7.4
```

```
3          Denmark      62 0.83    58    72 41 9.5
4          Finland      37 0.83    46    39 59 9.6
5           France      50 0.33    38    53 34 7.5
6          Germany      55 0.50    40    44 40 8.2
7           Greece      68 0.17    43    78 58 4.3
8          Ireland      81 0.17    47    90 55 7.4
9            Italy      64 0.17    31    57 31 5.0
10      Luxembourg      81 0.17    59    71 22 8.5
11     Netherlands      74 0.50    58    67 45 8.6
12        Portugal      63 0.17    45    73 51 6.5
13           Spain      57 0.17    45    60 38 7.0
14          Sweden      43 0.67    49    31 50 9.2
15 United Kingdom      33 0.33    38    36 71 8.6
```

Here we have specified the `all.x=TRUE` option,as this tell R to use the first named data set as the basis for the merging. If we had written `all.y=TRUE` R would have used the second data set as the basis, resulting in many `NA` values (go ahead and try it out).

4.2 Writing Your Date Frame to a .txt File

Once you have created the data that you want to work with it is good to save it as a .txt file. In order to do so we use the `write.table()` command.

```
> write.table(merged.data, file = "merged.data.txt", sep = "\t",
     col.names = TRUE, row.names = FALSE)
```

Notice that we specify the data to be written, the name of the file, including the extension, the separator to be used and whether there are labels for the rows and columns. This will make R write a .txt file with the new data and place it in the working directory.

5 Doing a Crisp Set QCA in R

5.1 Building a Dichotomous Data Table

First we need to read to package(s) into R that we are going to use.

```
> library(QCA3)
```

Here we will be using the Winzen data that we used above. The first step is to dichotomize the Winzen data. Now the question is: what criteria can we use to dichotimize these variables into set memberships? The standard answer is that the dichotomizing should be justified theoretically and match the researchers knowledge about the cases. However sometimes this is not possible, and in these instances we turn to technical criteria. One approach is to do a cluster analysis. Luckily there is a clustering command, `thresholdssetter()`, in the `QCA3` which makes this very easy for us.

```
> support <- thresholdssetter(data$support, 1, value = TRUE,
     method = "complete")
ans
 0  1
 5 10
> parl <- thresholdssetter(data$parl, 1, value = TRUE,
     method = "complete")
ans
 0  1
11  4
> trust <- thresholdssetter(data$trust, 1, value = TRUE,
     method = "complete")
ans
 0  1
 4 11
> gains <- thresholdssetter(data$gains, 1, value = TRUE,
     method = "complete")
ans
0 1
9 6
> ID <- thresholdssetter(data$ID, 1, value = TRUE, method = "complete")
ans
0 1
6 9
```

Notice that when we use the `thresholdsetter` function we first specify the variable that is to be dichotomized, then we specify in how many was we would like to "slice" the data, in our case we want dichotomize the data so we specify one slicing of the data. The `value` option tells the function whether to return just the threshold used to dichotomize the data (when set to `FALSE`) or return the full vector of dichotomized values (when set to `TRUE`). The method used is the so-called `complete linkage clustering`. The option accepts several types of clustering methods, for an overview type

```
> help(hclust)
```

in the R terminal, and you will get a help page that explains the different clustering methods available.

The numbers printed after each variable is the number of cases coded as either zero or one. Here one represents cases that score higher than cases coded with a zero.

The next step is to combine our new dichotomized variables into a new data set. Here we use the `data.frame()` command, where we specify the name of each variable and the values (in our case stored in the objects created above) associated with the variable.

```
> data_dich <- data.frame(support = support, parl = parl,
     trust = trust, gains = gains, ID = ID)
> row.names(data_dich) <- data$caseid
> data_dich
               support parl trust gains ID
Austria              0    1     1     0  1
Belgium              1    0     1     0  1
Denmark              1    1     1     1  0
Spain                1    0     1     0  0
Finland              0    1     1     0  1
France               0    0     0     0  0
Germany              1    0     0     0  0
Greece               1    0     1     1  1
Ireland              1    0     1     1  1
Italy                1    0     0     0  0
Luxembourg           1    0     1     1  0
```

```
Netherlands          1    0     1     1  1
Portugal             1    0     1     1  1
Sweden               0    1     1     0  1
United Kingdom       0    0     0     0  1
```

We now have a nice dichotomized data set!

5.2 Constructing a Truth Table

Producing a truth table is a very important step in the analysis, as all the results are derived from our truth table. Creating a truth table is at first a very mechanical process where we simply list every possible configuration of our causal factors, and then see which combinations we have in our data, and which outcome these combinations exhibit. First we must choose which variable is our outcome, here we will try to explain which factors lead to support for the EU.

```
> truthtable <- cs_truthTable(data_dich, outcome = "support", conditions = c("parl",
     "trust", "gains", "ID"))
> truthtable
   parl trust gains ID NCase freq1 freq0 OUT                                Cases
41    0     0     0  0     3     2     1   C               [France], Germany, Italy
44    0     1     0  0     1     1     0   1                                 Spain
53    0     1     1  0     1     1     0   1                            Luxembourg
54    1     1     1  0     1     1     0   1                               Denmark
68    0     0     0  1     1     0     1   0                        United Kingdom
71    0     1     0  1     1     1     0   1                               Belgium
72    1     1     0  1     3     0     3   0               Austria, Finland, Sweden
80    0     1     1  1     4     4     0   1 Greece, Ireland, Netherlands, Portugal
```

Notice that we have specified explicitly in the `cs_truthTable` command which variables are the explanatory causal factors (the `conditions` option in the truthtable command), and which one is the outcome. Furthermore we have specified the `method` option as being deterministic, i.e. we do not want to use probabilistic criteria to asses whether a configuration of causal factors score 1 or 0. Finally the `complete` option tells the function to return the whole truth table, including configurations that are not covered by empirical observations (the so-called logical remainders).

In the output the `NCase` column shows how many cases are covered by on combination, the `freq1` and `freq0` show how many cases in one combination exhibit either a presence of support for the EU (1), or the absence of support for the EU (0). Finally the `OUT` column show what value a given combination has on the outcome set. Here we see that we have one contradiction in the first row, hence the outcome is here labelled with a `C`, and the contradictory case France has been bracketed. Combinations where we have no information on the outcome are labelled with a `?`. From our point of view this truth table is not optimal, it has a lot of combinations where we have no data (these are called logical remainders), and we have a combination with contradictory outcomes. Thus there is still a lot of work to be done before we can proceed with the analysis.

5.3 Resolving Contradictory Configurations

In essence there are 8 strategies we could follow (or some combination). namely:

1. Add more causal factors to the data
2. Replace one or more causal factors
3. Re-examine the operationalization
4. Reconsider the outcome variable
5. Reconsider whether all cases belong to the same population
6. Recode all contradictory configurations as 0
7. Use frequency criteria to decide how the outcome should be coded

Here we will only cover the last two "technical" solutions. The first possibility is to recode the contradictory configuration as a 0, to do this we specify the option `contradictions` in the `reduce` function as negative. The second option is the treat the contradiction as positive based on a frequency criteria, in our case that would be two out of three of a ratio of positive to negative case equal to .66. Lets see what happens when we treat the contradictions as negative.

```
> solution_1 <- reduce(truthtable, explain = "positive",
     remainders = "exclude", contradictions = "negative")
> solution_1
```

```
Call:
reduce(x = truthtable, explain = "positive", remainders = "exclude",     contradictions

truthTable with 8 configuration(s)

   parl trust gains ID NCase freq1 freq0 OUT
41    0     0     0  0     3     2     1   C
44    0     1     0  0     1     1     0   1
53    0     1     1  0     1     1     0   1
54    1     1     1  0     1     1     0   1
68    0     0     0  1     1     0     1   0
71    0     1     0  1     1     1     0   1
72    1     1     0  1     3     0     3   0
80    0     1     1  1     4     4     0   1
                                     Cases
41                [France], Germany, Italy
44                                   Spain
53                               Luxembourg
54                                  Denmark
68                            United Kingdom
71                                  Belgium
72                Austria, Finland, Sweden
80 Greece, Ireland, Netherlands, Portugal

----------------
Prime implicant No. 1 with 2 implicant(s)

parl*TRUST + TRUST*GAINS*id

Common configuration: TRUST
```

Notice that we have specified the `remainders` option as exclude, that is we do not want to make any assumptions about possible values for the logical remainders. We have also specified the `keepTruthTable` as false, this simply tells the function not to show the truth table after we call it (we already know from above how thee truth table looks like).

Here we get a nice solution with two paths towards support for the EU. One path has the absence of a strong parliament AND a trust in the political system in the population as leading to support for the EU. The other path has trust in the political system AND the feeling that the population has on average gained from EU membership AND the absence of a strong national identity as leading to support for the EU.

If we conduct the same analysis, only this time we code the contradiction as a positive case based on the fact that two out of three countries in the configuration exhibits positive scores we get the following result.

```
> solution_2 <- reduce(truthtable, explain = "positive",
     remainders = "exclude", contradictions = "positive",
     keepTruthTable = FALSE)
> solution_2
Call:
reduce(x = truthtable, explain = "positive", remainders = "exclude",     contradictions


----------------
Prime implicant No. 1 with 3 implicant(s)

parl*TRUST + parl*gains*id + TRUST*GAINS*id

Common configuration: None
```

This is a nice albeit a surprising and not very common result, as this means that the contradictory configuration to not have an impact on our final solution.

5.4 What to do With the Logical Remainders?

However, what about the `remainders` option in the `reduce` function. This option tells the algorithm what to do with configurations that are not covered by empirical instances (all the configurations marked with a `?` in the truth table). Right now we have excluded them completely from the analysis, but they could potentially help us to create a more parsimonious solution, if we are willing to make some simplifying assumptions. The reason why this is the case is that if we decide, for instance, to code all logical remainders as positive, then

we will have more combinations to chose from when conducting the Boolean minimization. Thus the logical remainders represent a pool of potential useful cases. Lets see what happens if we *assume* that all the logical remainders would be positive if we would observe these configurations in the real world.

```
> solution_3 <- reduce(truthtable, explain = "positive",
     remainders = "include", contradictions = "positive",
     keepTruthTable = FALSE)
> solution_3
Call:
reduce(x = truthtable, explain = "positive", remainders = "include",     contradictions



----------------
Prime implicant No. 1 with 2 implicant(s)

parl*TRUST + id

Common configuration: None
```

Here we get two parsimonious solutions. One solution has one path to support for the EU being simply the absence of a strong national identity, and another path has the absence of a strong parliament AND trust in the political system (the same path as above). The second solution is essentially the same except for the gains path has been replaced by a path that has the absence of a strong national identity as sufficient. We get two equally parsimonious solutions because depending on how the program uses the remainders, it can get these two equally parsimonious solutions. This illustrates what effect it can have to make an assumption about how we should treat the logical remainders. To see what assumptions was made in getting the solutions we use the `SA` function (SA stand for simplifying assumptions):

```
> sa_1 <- SA(solution_3)
> sa_1
Call:
reduce(x = truthtable, explain = "positive", remainders = "include",     contradictions
```

```
----------------
Prime implicant No. 1 with 4 implicant(s)

PARL*trust*gains*id + PARL*TRUST*gains*id +
parl*trust*GAINS*id + PARL*trust*GAINS*id

Common configuration: id
```

Here we can see the configurations with no empirical data that has been assumed to be positive in order to get the different solutions. The big question now is: are those assumptions justified? This is where the researcher has to rely on his theoretical and case knowledge in order to decide whether any of these configurations should be used when conducting the analysis. Inspecting the assumptions made in the second solution, we see that it is assumed that the absence of all causal factors is assumed to lead to support for the EU. It seems very unrealistic that the absence of trust in the political system, absence of a strong parliament, on average not having gained from the EU and the absence of a strong national identity should lead to support for the EU. It could be argued that this puts to much weight on the causal leverage of the absence of a strong national identity. If we buy this argument then the second solution is based on unjustified assumptions and should be considered unrealistic. Thus we can conclude based on this argument that in the absence of a strong parliament the population has on average to trust the political system, OR feel that there has been some gains from the EU in order to support the EU.

It also possible to manually set the configurations that you think are realistically plausible, leaving the rest out. This is referred to as a intermediate solution, as we do not let the algorithm search for the most parsimonious solution, and neither do we choose the conservative solution. In order to do this we first create a data frame with the configurations we think realistically should be a 1 if we would observe them in real life. The we feed this data frame along with our original analysis to the `constrReduce` function.

```
> x <- as.data.frame(matrix(c(1, 0, 0, 0, 1, 1, 0, 0, 1,
     0, 1, 0), nrow = 3, byrow = T))
```

```
> colnames(x) <- c("parl", "trust", "gains", "ID")
> x
  parl trust gains ID
1    1     0     0  0
2    1     1     0  0
3    1     0     1  0
```

Notice that we specify the configurations as one long list of 0's and 1's. The list is to be read as blocks of fours, where each block specify whether a causal condition is present or not. Since we have four blocks of fours in the list, we tell R to create four rows (the `nrow=4` option), and we tell R to structure the matrix by rows, not columns (the `byrow=T` option. Finally we use the `colnames` function to name each column according to what causal factor it represents. We do all of this because we want to tell the `constrReduce` function that it is allowed to treat the logical remainders: PARL*trust*Gains*id, PARL*TRUST*gains*id, parl*trust*GAINS*id and parl*TRUST*gains*id as configurations that could lead to a positive outcome where we to observe them in real life. Notice that now the algorithm will try to come up with the most parsimonious solution involving all or just a few of the plausible configurations specified

```
> solution_4 <- constrReduce(solution_1, include = x)
> solution_4
Call:
constrReduce(object = solution_1, include = x)

truthTable with 8 configuration(s)

   parl trust gains ID NCase freq1 freq0 OUT
41    0     0     0  0     3     2     1   C
44    0     1     0  0     1     1     0   1
53    0     1     1  0     1     1     0   1
54    1     1     1  0     1     1     0   1
68    0     0     0  1     1     0     1   0
71    0     1     0  1     1     1     0   1
72    1     1     0  1     3     0     3   0
80    0     1     1  1     4     4     0   1
```

```
                                    Cases
41               [France], Germany, Italy
44                                    Spain
53                               Luxembourg
54                                  Denmark
68                           United Kingdom
71                                  Belgium
72               Austria, Finland, Sweden
80 Greece, Ireland, Netherlands, Portugal

----------------
Prime implicant No. 1 with 2 implicant(s)

parl*TRUST + PARL*id

Common configuration: None

----------------
Prime implicant No. 2 with 2 implicant(s)

parl*TRUST + TRUST*id

Common configuration: TRUST
```

Each of the prime implicants represent a possible solution. The function returns three solutions as these are all equally parsimonious, to check which assumptions went into which solution we again use the `SA` function:

```
> sa_2 <- SA(solution_4)
> sa_2
Call:
constrReduce(object = solution_1, include = x)


----------------
Prime implicant No. 1 with 3 implicant(s)
```

```
PARL*trust*gains*id + PARL*TRUST*gains*id +
PARL*trust*GAINS*id

Common configuration: PARL*id

----------------
Prime implicant No. 2 with 1 implicant(s)

PARL*TRUST*gains*id

Common configuration: PARL*TRUST*gains*id
```

If we look the second prime implicant we can see that this solution only relies on one simplifying assumption, namely that PARL*TRUST*gains*id would be positive in the real world. Since we only need to make one simplifying assumption for this solution this is the better solution for out purposes. Notice also that the solution is more complex than the solution we found above where we allowed the algorithm to use all logical remainders.

If we are careful in which simplifying assumptions we make, then the solution

parl*TRUST + TRUST*id

is the most parsimonious we can get. This solution tells us that when there is no strong parliament present in a country, the population has to on average to trust the political system OR if the population trust the political system and has a weak national identity then we see support for the EU.

6 Coverage and Consistency for Crisp Set QCA

Once we have found the solutions that we want to use for the analysis we can begin calculating the consistency of our solutions. To calculate the consistency we must first find out which cases that are part of our analysis, i.e. if we analyze positive or negative outcomes, and from these cases we must find the ones that are covered by our solution(s).

In the first solution found with the Winzen data we got the following result:

```
> summary(solution_1)
Call:
reduce(x = truthtable, explain = "positive", remainders = "exclude",     contradictions

Total number of cases: 15
Number of cases [1]: 10
Number of cases [0]: 5

----------------
Prime implicant No. 1 with 2 implicant(s)

parl*TRUST + TRUST*GAINS*id
Number of case: 7 + 2 = 9
Percentage: 0.467 + 0.133 = 0.6
No. of cases by Multiple PIs: 1 (0.067)
Cases: (1)Spain (2)Luxembourg (1)Belgium (1)Greece, Ireland,
Netherlands, Portugal + (2)Luxembourg (1)Denmark
```

To find all of the cases that are part of the two paths we use the `subset()` function to first find all positive cases, and then find all the cases that matches the two parts of the solution. The `subset()` function allows us to use logical operators to select subsets of our data. The logical operators mostly used in this function are:

Operator	Sign
AND	&
OR	|
equal to	==

Table 1: LOGICAL OPERATORS IN THE `SUBSET()` FUNCTION: These operators are the most used in the `subset()` function, and allows us to do quite sophisticated operations on our data

Say we want to find all rows that have a score on the `gains` variable that is equal to 1, then we write:

```
> subset(data_dich, gains == 1)
```

```
             support parl trust gains ID
Denmark            1    1     1     1  0
Greece             1    0     1     1  1
Ireland            1    0     1     1  1
Luxembourg         1    0     1     1  0
Netherlands        1    0     1     1  1
Portugal           1    0     1     1  1
```

If we want to find the rows with a score on `gains` AND `ID` equal to 1 then we write:

```
> subset(data_dich, gains == 1 & ID == 1)
             support parl trust gains ID
Greece             1    0     1     1  1
Ireland            1    0     1     1  1
Netherlands        1    0     1     1  1
Portugal           1    0     1     1  1
```

If we want to find the rows that has `gains` OR `ID` equal to 1 we write:

```
> subset(data_dich, gains == 1 | ID == 1)
                support parl trust gains ID
Austria               0    1     1     0  1
Belgium               1    0     1     0  1
Denmark               1    1     1     1  0
Finland               0    1     1     0  1
Greece                1    0     1     1  1
Ireland               1    0     1     1  1
Luxembourg            1    0     1     1  0
Netherlands           1    0     1     1  1
Portugal              1    0     1     1  1
Sweden                0    1     1     0  1
United Kingdom        0    0     0     0  1
```

As you can see it is possible to write quite complex statements using these simple rules in order to select the rows that we are interested in. This is something that we will exploit a lot when calculating the consistency and coverage

scores. To start with we select all the positive cases, and the cases that are covered be each part of the solution.

```
> all <- subset(data_dich, support == 1)
> part_1 <- subset(data_dich, parl == 0 & trust == 1)
> part_2 <- subset(data_dich, trust == 1 & gains == 1 &
     ID == 0)
> all
            support parl trust gains ID
Belgium           1    0     1     0  1
Denmark           1    1     1     1  0
Spain             1    0     1     0  0
Germany           1    0     0     0  0
Greece            1    0     1     1  1
Ireland           1    0     1     1  1
Italy             1    0     0     0  0
Luxembourg        1    0     1     1  0
Netherlands       1    0     1     1  1
Portugal          1    0     1     1  1
> part_1
            support parl trust gains ID
Belgium           1    0     1     0  1
Spain             1    0     1     0  0
Greece            1    0     1     1  1
Ireland           1    0     1     1  1
Luxembourg        1    0     1     1  0
Netherlands       1    0     1     1  1
Portugal          1    0     1     1  1
> part_2
           support parl trust gains ID
Denmark          1    1     1     1  0
Luxembourg       1    0     1     1  0
```

Here we see that for both parts of the solution there are no cases that score a zero, thus we have perfect consistency.

To see how large the coverage of this part of the solution is, we simply take the ratio of the cases found to fit with one part of the solution to all positive cases. In `R` we can do this by using the `nrow()` command which counts the number of rows in a data frame or matrix. Remember that the number of rows corresponds to the number of observations we have in a data set. For the two parts of the solution above we get the following coverage scores:

```
> nrow(part_1)/nrow(all)
[1] 0.7
> nrow(part_2)/nrow(all)
[1] 0.2
```

The unique coverage of each solution is found by first finding the cases only covered by this solution and then dividing this number by the total number of positive cases. To do this we use the `setdiff()` operator. This takes the set theoretical difference between the first and the second vector that we feed it, thus for the first part of the solution we do it like this:

```
> one <- setdiff(rownames(part_1), rownames(part_2))
> one
[1] "Belgium"     "Spain"       "Greece"      "Ireland"
[5] "Netherlands" "Portugal"
> length(one)/nrow(all)
[1] 0.6
```

We have used the rownames of the vectors in order to also be able to extract the names of the cases that are covered by each part of the solution. The `length()` function simply gives us a count of all the elements in a vector, and this we divide by the number of rows in the subset with all positive cases.

For the second part of the solution we get this:

```
> two <- setdiff(rownames(part_2), rownames(part_1))
> two
[1] "Denmark"
> length(two)/nrow(all)
[1] 0.1
```

Notice that when finding the unique element of the second part of the solution, we simply switch the place of our vectors in the function.

To find the coverage of the full solution we again use the subset function, and proceed in the same way as above.

```
> full <- subset(data_dich, parl == 0 & trust == 1 | trust ==
     1 & gains == 1 & ID == 0)
> full
            support parl trust gains ID
Belgium           1    0     1     0  1
Denmark           1    1     1     1  0
Spain             1    0     1     0  0
Greece            1    0     1     1  1
Ireland           1    0     1     1  1
Luxembourg        1    0     1     1  0
Netherlands       1    0     1     1  1
Portugal          1    0     1     1  1
> nrow(full)/nrow(all)
[1] 0.8
```

Notice how we have written out the full solution in the `subset()` function. This is a good example of the kind of logical statements that we can use to extract subsets from our data. The reason why our solution do not cover all positive cases, is that when doing the first analysis we treated the contradiction as negative. There were two positive cases in the contradictory row, namely Italy and Germany. These cases are not covered by our solution, and thus we cannot get a perfect coverage, although the solution is perfectly consistent with the cases used in the analysis.

To summarize the information about solutions, coverage and consistency we put all the informatio in a table like the table below.

7 Fuzzy Sets QCA

7.1 Calibration

There are no method (yet) for calibrating sets in `QCA3`, however it is not difficult to write a small function to do this in R. The following code is a simple

	Coverage			
Solution	Raw	Unique	Consistency	Cases
parl*TRUST	0.7	0.6	1	Belgium, Spain, Greece, Ireland, Netherlands, Portugal, Luxembourg
TRUST*GAINS*id	0.2	0.1	1	Luxembourg, Denmark
Solution Coverage	0.8			
Solution Consistency	1			

Table 2: SUMMARY OF THE SOLUTION, COVERAGE and CONSISTENCY: The table show all the information necessary to be able to evaluate whether a solution fits the cases it is supposed to explain.

implementation of the direct method described in Ragin 2008 chapter 5.

```
> direct.method <- function(data, high, low, cross) {
      var <- data
      var.dev <- var - cross
      scalar.high <- 3/(high - cross)
      scalar.low <- -3/(low - cross)
      out <- vector()
      for (i in 1:length(var.dev)) {
          if (var.dev[i] > 0) {
              out[i] <- var.dev[i] * scalar.high
          }
          else {
              out[i] <- var.dev[i] * scalar.low
          }
      }
      final <- vector()
      for (i in 1:length(out)) {
          final[i] <- exp(out[i])/(1 + exp(out[i]))
      }
      final <- round(final, digits = 4)
```

```
    return(final)
}
```

The function takes four argument, which must all be supplied. First you need to supply a vector with values (the data argument), the function only works on one set at a time, so no data frames. We also need to supply the function with the ceiling (high) crossover (cross) point and floor (low). Notice that the function has been hardcoded to use the membership scores of .95 for high and .05 for low. The function thus assumes that you will use the logodds of 3 for completely in and -3 for completely out, when calibrating the sets. To see the how the function works we here use the undichotomized Winzen data to create a data frame with fuzzy sets. However keep in mind that the upper bound, crossover point and lower bound of a set should always be determined through theory, in the example below the values have been chosen somewhat arbitrarily.

```
> support <- direct.method(data$support, 75, 40, 58)
> parl <- direct.method(data$parl, 0.75, 0.2, 0.39)
> trust <- direct.method(data$trust, 55, 35, 46)
> gains <- direct.method(data$gains, 85, 35, 58)
> ID <- direct.method(data$ID, 65, 35, 30)
> fuzzy <- data.frame(support = support, parl = parl, trust = trust,
    gains = gains, ID = ID)
> rownames(fuzzy) <- data$caseid
> fuzzy
              support   parl  trust  gains     ID
Austria        0.0884 0.9116 0.7311 0.1729 0.7976
Belgium        0.5000 0.0301 0.3669 0.4351 0.7834
Denmark        0.6695 0.9751 0.9820 0.8257 0.7197
Spain          0.4584 0.0301 0.4322 0.5553 0.6650
Finland        0.0293 0.9751 0.5000 0.0774 0.9231
France         0.2086 0.2794 0.1014 0.3425 0.5849
Germany        0.3775 0.7144 0.1630 0.1387 0.7021
Greece         0.8538 0.0301 0.3061 0.9022 0.9168
Ireland        0.9830 0.0301 0.5826 0.9722 0.8950
Italy          0.7425 0.0301 0.0164 0.4674 0.5214
```

```
Luxembourg      0.9830 0.0301 0.9870 0.8091 0.9918
Netherlands     0.9439 0.7144 0.9820 0.7311 0.7834
Portugal        0.7073 0.0301 0.4322 0.8411 0.8581
Sweden          0.0759 0.9116 0.7311 0.0287 0.8474
United Kingdom  0.0153 0.2794 0.1014 0.0537 0.9711
```

7.2 Plotting Fuzzy Sets

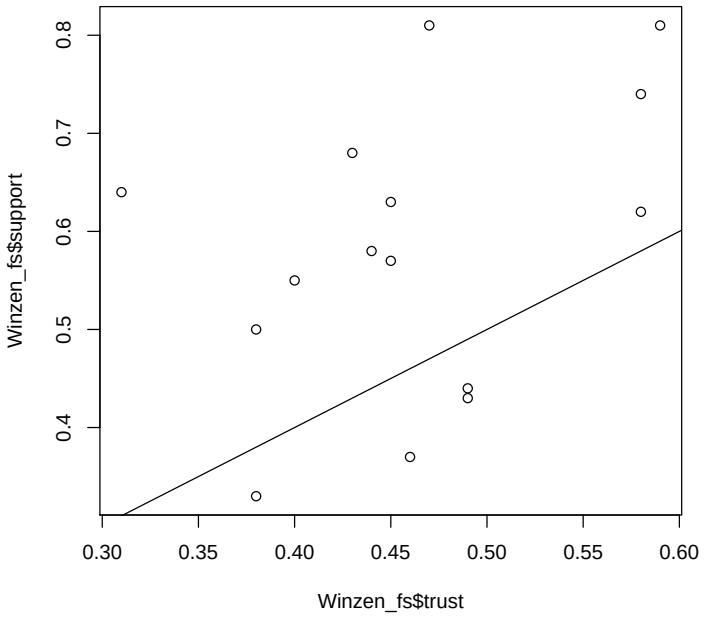
One of the greatest strengths of R is the ability to create publication quality graphics that are extremely customizable. In fsQCA it is often of interest to get a visual feel for how the causal sets behave compared to the outcome set. To make a simple two dimensional plot of one causal factor and the outcome set we use the `plot(x,y)` command. Where x refers to the x-axis and y refers to the y-axis. Let us first read in the fuzzy set version of the Winzen data:

```
> Winzen_fs <- read.table("Winzen_fs1.txt", sep = "\t",
     header = TRUE)
> Winzen_fs
            caseid parl trust   id gains support
1          Austria  0.8  0.49 0.46  0.46    0.44
2          Belgium  0.2  0.44 0.45  0.56    0.58
3          Denmark  1.0  0.58 0.41  0.72    0.62
4            Spain  0.2  0.45 0.38  0.60    0.57
5          Finland  1.0  0.46 0.59  0.39    0.37
6           France  0.4  0.38 0.34  0.53    0.50
7          Germany  0.6  0.40 0.40  0.44    0.55
8           Greece  0.2  0.43 0.58  0.78    0.68
9          Ireland  0.2  0.47 0.55  0.90    0.81
10           Italy  0.2  0.31 0.31  0.57    0.64
11     Luxembourg  0.2  0.59 0.22  0.71    0.81
12    Netherlands  0.6  0.58 0.45  0.67    0.74
13       Portugal  0.2  0.45 0.51  0.73    0.63
14         Sweden  0.8  0.49 0.50  0.31    0.43
15 United Kingdom  0.4  0.38 0.71  0.36    0.33
```

The next step is to plot some of the fuzzy sets against the outcome, again we will be using the support for the EU set as the outcome. Notice that below

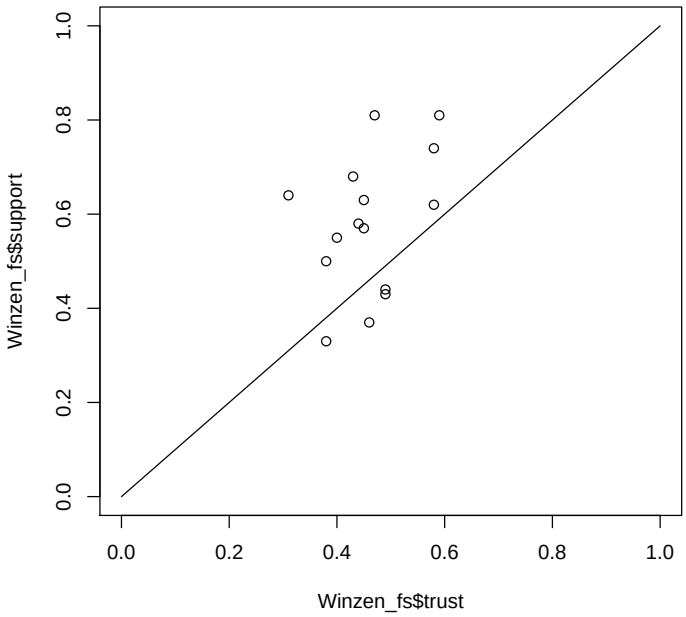
we first use the plot command to generate the `plot()`, and then we use the `segments(`x_1`,`y_1`,`x_2`,`y_2`)` command to add the 45 degree line to the plot.

```
> plot(Winzen_fs$trust, Winzen_fs$support)
> segments(0, 0, 1, 1)
```



As you can see the plot does not look quite right, this is because R automatically scales the x an y axis to fit with the data, and since there are no observations below .3 on the y-axis and .3 on the x-axis R uses these as the start of the scales. To change this we need to tell R where to begin and end the scales. Since we are dealing with fuzzy sets that are bound by 0 below and 1 above, it is natural that the x and y scales should also be between 0 and 1. To do this we specify the `xlim` and `ylim` options in the `plot()` command.

```
> plot(Winzen_fs$trust, Winzen_fs$support, xlim = c(0,
      1), ylim = c(0, 1))
> segments(0, 0, 1, 1)
```

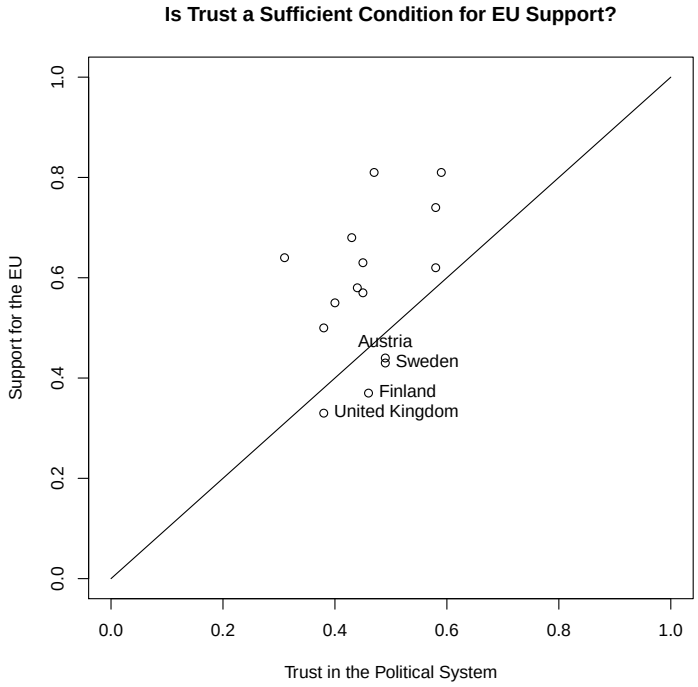


Now this looks much better!

Sometime we want to identify some or all of the cases that we have plotted. If you have a variable in your data set that records the name of every case (i.e a name for every row), then we can use the `identify()` function to add names to the plot. The identify function is somewhat unique as you have to click with your mouse on the points that you want to be labelled . To end the identification process simply click the right mouse key. Often it is interesting to see which cases that are not consistent with an argument of necessity or sufficency. In order to make the plot a little more readable we can also specify names for the axes, as well as main title for the plot, this is done through the `xlab`, `ylab` and `main` options in the `plot()` function. Below the cases are marked that are not consistent with trust being sufficient for support for the EU. Furthermore the axes and plot has been renamed.

```
> plot(Winzen_fs$trust, Winzen_fs$support, xlim = c(0,
      1), ylim = c(0, 1), xlab = "Trust in the Political System",
      ylab = "Support for the EU", main = "Is Trust a Sufficient Condition for EU Support
> segments(0, 0, 1, 1)
```

```
> identify(Winzen_fs$trust, Winzen_fs$support, labels = Winzen_fs$caseid)
```



7.3 Calculating Consistency and Coverage

To calculate the consistency of trust in the political system as a sufficient condition of support for the EU we can use the `suffnec()` function in the `QCA3` package. Remember that the formula for consistency for a necessary condition is:

$$\text{Necessary Condition } = \frac{\sum min(x_i, y_i)}{\sum y_i} \tag{1}$$

And the formuly for the consistency of a sufficient condition is:

$$\text{Sufficient Condition } = \frac{\sum min(x_i, y_i)}{\sum x_i} \tag{2}$$

The `suffnec()` function will take a data frame and give a matrix with consistency scores for necessity and sufficiency for every causal factor in the data set. As such it gives us an overview of consistency *and* coverage for all

causal factors in relation to each other. However the function does not work with a data set that contains variables with text strings, we therefore have to remove the `caseid` variable from the data set. This we do with the `subset()` function.

```
> Winzen_new <- subset(Winzen_fs, select = c("parl", "support",
    "trust", "id", "gains"))
```

Then we feed this new data set into the `suffnec()` function.

```
> suffnec(Winzen_new)
Necessity Scores Matrix:
'X is necessary condition of Y'
         Y
X          parl support trust    id gains
  parl    1.000   0.591 0.748 0.722 0.582
  support 0.734   1.000 0.964 0.899 0.940
  trust   0.737   0.764 1.000 0.888 0.756
  id      0.707   0.709 0.883 1.000 0.701
  gains   0.726   0.944 0.957 0.892 1.000

Sufficiency Scores Matrix:
'X is sufficient condition of Y'
         Y
X          parl support trust    id gains
  parl    1.000   0.734 0.737 0.707 0.726
  support 0.591   1.000 0.764 0.709 0.944
  trust   0.748   0.964 1.000 0.883 0.957
  id      0.722   0.899 0.888 1.000 0.892
  gains   0.582   0.940 0.756 0.701 1.000
```

To find out what the consistency score for trust as a sufficient condition for support is, we go to the second matrix in the output, the sufficiency matrix, go down the X column and find the trust row. Then we go along that row until we reach the support column. This tells us that the consistency score is 0.964 (try and calculate it by hand to verify the result). To assess how large a proportion of cases this quite consistent set relation covers we look in the necessity matrix

and find that the coverage of trust as a sufficient condition for support is 0.764 (try and verify this by hand).

7.4 Creating a Truth Table From Fuzzy Set Data

When we want to use the truth table algorithm detailed in **?**, it is necessary to put our fuzzy sets into a truth table. To do this we use the `fs_truthTable()` function. This function has two key options that we must specify, namely the threshold for cases and the consistency threshold.

```
> fuzzy <- fs_truthTable(Winzen_fs, outcome = "support", conditions = c("parl",
     "trust", "gains", "id"), ncases_cutoff = 1, consistency_cutoff = 0.9,
     complete = FALSE, cases = "caseid")
> fuzzy
   parl trust gains id OUT freq1 freq0 NCase Consistency                      Cases
77    0     0     1  1   1     3     0     3   0.9938398    Greece,Ireland,Portugal
69    1     0     0  1   1     1     0     1   0.9274725                   Finland
68    0     0     0  1   1     1     0     1   0.9326683            United Kingdom
54    1     1     1  0   1     2     0     2   0.9915966       Denmark,Netherlands
53    0     1     1  0   1     1     0     1   1.0000000                Luxembourg
50    0     0     1  0   1     4     0     4   0.9945055 Belgium,Spain,France,Italy
42    1     0     0  0   1     2     0     2   0.9585153            Austria,Germany
```

Notice that we have specified the outcome and conditions exactly as with the crisp set procedure, the only new options are the above mentioned options. Notice furthermore that Sweden is missing from the truth table. The explanation for this is quite simple: Sweden scores a .5 on the `id` factor, thus making it impossible for the algorithm to assign the case to corner in the vector space. For now we will not go further into this issue, just remember that it is worth avoiding coding cases as .5 on a causal factor as this in effect means that the case gets excluded from the analysis.

Often it is worth experimenting with the `consistency_cutoff` value as it has large impact on the further analysis, so we want to be sure that chose the most appropriate cutoff value. Changing the cutoff value from .90 to .97 has the following effect:

```
> fuzzy <- fs_truthTable(Winzen_fs, outcome = "support", conditions = c("parl",
     "trust", "gains", "id"), ncases_cutoff = 1, consistency_cutoff = 0.97,
     complete = FALSE, cases = "caseid")
> fuzzy
   parl trust gains id OUT freq1 freq0 NCase Consistency                      Cases
77    0     0     1  1   1     3     0     3   0.9938398    Greece,Ireland,Portugal
69    1     0     0  1   0     0     1     1   0.9274725                   Finland
68    0     0     0  1   0     0     1     1   0.9326683            United Kingdom
54    1     1     1  0   1     2     0     2   0.9915966        Denmark,Netherlands
53    0     1     1  0   1     1     0     1   1.0000000                Luxembourg
50    0     0     1  0   1     4     0     4   0.9945055 Belgium,Spain,France,Italy
42    1     0     0  0   0     0     2     2   0.9585153            Austria,Germany
```

Notice that in the previous truth table we only had positive cases, changing the cutoff value transform three cases into zero cases, so we now have four cases with the outcome present and three cases with the outcome absent. For now we will assume that this is an appropriate truth table.

7.5 Minimizing the Truth Table

To minimize our truth table we use the same procedure as we used with crisp set QCA.

```
> solution_1 <- reduce(fuzzy, explain = "positive", remainders = "exclude",
     contradictions = "negative", keepTruthTable = TRUE)
> summary(solution_1)
Call:
reduce(x = fuzzy, explain = "positive", remainders = "exclude",     contradictions = "ne

Total number of cases: 14
Number of cases [1]: 10
Number of cases [0]: 4

----------------
Prime implicant No. 1 with 2 implicant(s)

parl*trust*GAINS + TRUST*GAINS*id
```

```
Number of case: 7 + 3 = 10
Percentage: 0.5 + 0.214 = 0.714
No. of cases by Multiple PIs: 0 (0)
Cases: (1)Greece,Ireland,Portugal (1)Belgium,Spain,France,Italy +
(1)Denmark,Netherlands (1)Luxembourg
```

The procedures for using remainders and dealing with contradictions are exactly the same as for a crisp set analysis, thus you are referred to the crisp set QCA above for more details.

7.6 Coverage of the Solution

To find the coverage of our solution we proceed much as above, however now we have an added step of finding the membership values in the solution for the cases.

Looking at the first part of the solution from the above analysis (parl*trus*GAINS) we have to find out how each case scores on the different sets in the part. To do this we have to find the negation of `parl` and `trust`. To do this we simply subtract each of the vectors from 1.

```
> parl <- 1 - Winzen_fs$parl
> trust <- 1 - Winzen_fs$trust
> GAINS <- Winzen_fs$gains
```

Then we bind the new sets into a matrix:

```
> part_1 <- cbind(parl, trust, GAINS)
```

To find the minimum membership in the three sets for each case we use the `apply()` function. This function allows us to apply a specified procedure to every row or column. First we have to tell the function what data to use, then we tell i if the procedure should be carried out on rows (1) or columns (2). Finally we tell the function what procedure to carry out, in this case we want to take the minimum of every row. Then we find the sum of that vector

```
> part_1_min <- apply(part_1, 1, min)
> part1 <- sum(part_1_min)
> part1
```

```
[1] 5.83
```

Now we have the membership score in this part of the solution for all cases, to find the overlap between this and the outcome we take the minimum between the outcome and the above vector

```
> overlap1 <- cbind(part_1_min, Winzen_fs$support)
> overlap1 <- apply(overlap1, 1, min)
> overlap1 <- sum(overlap1)
> overlap1
[1] 5.77
```

We do the same for the second part of the solution:

```
> id <- 1 - Winzen_fs$id
> TRUST <- Winzen_fs$trust
> GAINS <- Winzen_fs$gains
> part_2 <- cbind(id, TRUST, GAINS)
> part_2_min <- apply(part_2, 1, min)
> part2 <- sum(part_2_min)
> part2
[1] 6.47
> overlap2 <- cbind(part_2_min, Winzen_fs$support)
> overlap2 <- apply(overlap2, 1, min)
> overlap2 <- sum(overlap2)
> overlap2
[1] 6.43
```

For the entire solution we take the maximum membership for the cases in either parts of the solution, but otherwise we proceed the same way

```
> full <- cbind(part_1_min, part_2_min)
> full_max <- apply(full, 1, max)
> full <- sum(full_max)
> full
[1] 7.5
```

Finally we find the sum of the outcome set:

```
> out <- sum(Winzen_fs$support)
> out
[1] 8.7
```

Now we can create a matrix that puts it all together:

Causal Conditions	Sum of Consistent Scores	Sum of Outcome Scores	Coverage
parl*trust*GAINS	5.77	8.7	.66
id*TRUST*GAINS	6.43	8.7	.74
parl*trust*GAINS + id*TRUST*GAINS	7.5	8.7	.86

From this table we can partition the coverage into unique and shared coverage:

	Total Coverage	Without Term	Unique
parl*trust*GAINS	.86	.74	.12
id*TRUST*GAINS	.86	.66	.2