Package ‘QCA’

March 24, 2017

Version 2.6
Date 2017-03-24
Title Qualitative Comparative Analysis
Depends R (>= 3.0.0)
Imports lpSolve, methods, shiny, venn (>= 1.2), fastdigest
Description An extensive set of functions to perform Qualitative Comparative Analysis: crisp sets ('csQCA'), temporal ('tQCA'), multivalue sets ('mvQCA') and fuzzy sets ('fsQCA'), using a GUI - graphical user interface. ‘QCA’ is a methodology that bridges the qualitative and quantitative divide in social science research. It uses a Boolean algorithm that results in a minimal causal combination which explains a given phenomenon.
License GPL (>= 2)
NeedsCompilation yes
Author Adrian Dusa [aut, cre, cph],
     jQuery Foundation [cph] (jQuery library and jQuery UI library),
     jQuery contributors [ctb, cph] (jQuery library; authors listed in inst/gui/www/shared/jquery-AUTHORS.txt),
     Vasil Dinkov [ctb, cph] (jquery.smartmenus.js library),
     Dmitry Baranovskiy [ctb, cph] (raphael.js library),
     Emmanuel Quentin [ctb, cph] (raphael.inline_text_editing.js library),
     Jimmy Breck-McKye [ctb, cph] (raphael-paragraph.js library),
     Alrik Thiem [aut] (from version 1.0-0 up to version 1.1-3)
Maintainer Adrian Dusa <dusa.adrian@unibuc.ro>
Repository CRAN
Date/Publication 2017-03-24 19:54:30 UTC

R topics documented:

   QCA-package ....................................................... 2
calibrate ......................................................... 4
deMorgan ............................................................ 8
eqmcc ............................................................... 10
The package \texttt{QCA} contains functions to perform Qualitative Comparative Analysis, complemented with a graphical user interface. It implements the comparative method as first described by Ragin (1987), and extended by Cronqvist and Berg-Schlosser (2009) and Ragin (2000, 2008). QCA is a bridge between the qualitative and quantitative research methodologies, making use of the qualitative procedures in a systematic, algorithmic way (therefore increasing the “confidence” in the results, as understood by quantitative researchers).

The Quine-McCluskey minimization algorithm implemented in this package is mathematically exact, as described by Dusa (2007), Dusa (2010) and Dusa and Thiem (2015). It returns the relevant prime implicants for both \texttt{csQCA} (binary crisp sets QCA) and also for \texttt{mvQCA} (multi-value QCA).

Starting with version 0.4-5 the package \texttt{QCA} gained a new function called \texttt{eqmcc} (from "e"nhanced Quine-McCluskey) that finds exact solutions with a substantially lower memory consumption. The multi-value extension of the \texttt{eqmcc} function was implemented in version 0.6-0.

Between versions 1.0-0 and up to version 1.1-3, the package welcomed a second co-author Alrik Thiem, responsible with the manual testing and documentation of functions, while the main author Adrian Dusa was responsible with all of the software development. The package gained new functionality by employing \texttt{fsQCA} (fuzzy-set QCA) and also other types of QCA like \texttt{tsQCA} (temporal QCA), see Caren and Panofsky (2005), Ragin and Strand (2008). The package is also able to obtain the causal chains similar to those from the package \texttt{cna} (see Ambuehl et all 2015).
The author of this package is grateful for the effort of the former coauthor, Alrik Thiem, given his responsibilities of testing and documenting the functions. Although the current documentation is completely re-written, it owes a great deal of impact from the previous versions.

The results of the QCA package are very consistent with (and sometimes even better than) the results of the other software packages for QCA, most notably fs/QCA by Ragin and Davey (2014) and Tosmana by Cronqvist and Berg-Schlosser (2009). A comparison of several such software is provided by Thiem and Dusa (2013).

From version 2.0, this package uses a new graphical user interface based on the package shiny. In order to avoid developing different interfaces for different operating systems, the current GUI was designed to work into a webpage. It uses a combination of HTML, CSS, jQuery for the user interface, a custom development framework using vector graphics based on the Raphael library, and extensive Javascript code to bind these altogether. A first version of this user interface was presented by Dusa (2007), but users should be aware the current version is much more advanced. Starting with version 2.5, the user interface gained a web-based command console to offer a complete experience of available functionality.

Details

Package: QCA
Type: Package
Version: 2.6
Date: 2017-03-24
License: GPL (>= 2)

Author(s)

Authors:
Adrian Dusa
Department of Sociology
University of Bucharest
<dusa.adrian@unibuc.ro>

Maintainer:
Adrian Dusa

References

URL: https://cran.r-project.org/package=cna.


Trier: University of Trier.
calibrate

Calibrate raw data to crisp or fuzzy sets

Description

This function transforms (calibrates) the raw data to either crisp or fuzzy sets values, using the direct method of calibration.

Usage

\[
\text{calibrate}(x, \text{type} = \text{"fuzzy"}, \text{thresholds} = \text{NA}, \text{logistic} = \text{TRUE}, \\
\text{idm} = 0.95, \text{ecdf} = \text{FALSE}, \text{below} = 1, \text{above} = 1, \ldots)
\]

Arguments

- **x**: A numerical causal condition.
- **type**: Calibration type, either "crisp" or "fuzzy".
- **thresholds**: A vector of (named) thresholds.
- **logistic**: Calibrate to fuzzy sets using the logistic function.
- **idm**: The set inclusion degree of membership for the logistic function.
**calibrate**

**ecdf** Calibrate to fuzzy sets using the empirical cumulative distribution function of the raw data.

**below** Numeric (non-negative), determines the shape below crossover.

**above** Numeric (non-negative), determines the shape above crossover.

... Additional parameters, mainly for backwards compatibility.

**Details**

Calibration is a transformational process from raw numerical data (interval or ratio level of measurement) to set membership scores, based on a certain number of qualitative anchors.

When `type = "crisp"`, the process is similar to recoding the original values to a number of categories defined by the number of thresholds. For one threshold, the calibration produces two categories (intervals): 0 if below, 1 if above. For two thresholds, the calibration produces three categories: 0 if below the first threshold, 1 if in the interval between the thresholds and 2 if above the second threshold etc.

When `type = "fuzzy"`, calibration produces fuzzy set membership scores, using three anchors for the increasing or decreasing *s-shaped* distributions (including the logistic function), and six anchors for the increasing or decreasing *bell-shaped* distributions.

The argument `thresholds` can be specified either as a simple numeric vector, or as a named numeric vector. If used as a named vector, for the first category of *s-shaped* distributions, the names of the thresholds should be:

- "e" for the full set exclusion
- "c" for the set crossover
- "i" for the full set inclusion

For the second category of *bell-shaped* distributions, the names of the thresholds should be:

- "e1" for the first (left) threshold for full set exclusion
- "c1" for the first (left) threshold for set crossover
- "i1" for the first (left) threshold for full set inclusion
- "i2" for the second (right) threshold for full set inclusion
- "c2" for the second (right) threshold for set crossover
- "e2" for the second (right) threshold for full set exclusion

If used as a simple numerical vector, the order of the values matter.

If `e < c < i`, then the membership function is increasing from `e` to `i`. If `i < c < e`, then the membership function is decreasing from `i` to `e`.

Same for the *bell-shaped* distribution, if `e1 < c1 < i1 ≤ i2 < c2 < e2`, then the membership function is first increasing from `e1` to `i1`, then flat between `i1` and `i2`, and then decreasing from `i2` to `e2`. In contrast, if `i1 < c1 < e1 ≤ e2 < c2 < i1`, then the membership function is first decreasing from `i1` to `e1`, then flat between `e1` and `e2`, and finally increasing from `e2` to `i2`.
When \texttt{logistic = TRUE} (the default), the argument \texttt{idm} specifies the inclusion degree of membership for the logistic function. If \texttt{logistic = FALSE}, the function returns linear \textit{s-shaped} or \textit{bell-shaped} distributions (curved using the arguments \texttt{below} and \texttt{above}), unless activating the argument \texttt{ecdf}.

If there is no prior knowledge on the shape of the distribution, the argument \texttt{ecdf} asks the computer to determine the underlying distribution of the empirical, observed points, and the calibrated measures are found along that distribution.

Both \texttt{logistic} and \texttt{ecdf} arguments can be used only for \textit{s-shaped} distributions (using 3 thresholds), and they are mutually exclusive.

The parameters \texttt{below} and \texttt{above} (active only when both \texttt{logistic = TRUE} and \texttt{ecdf} are deactivated, establish the degree of concentration and dilation (convex or concave shape) between the threshold and crossover:

\begin{align*}
\emptyset < \texttt{below} < 1 & \quad \text{dilates in a concave shape below the crossover} \\
\texttt{below} = 1 & \quad \text{produces a linear shape (neither convex, nor concave)} \\
\texttt{below} > 1 & \quad \text{concentrates in a convex shape below the crossover} \\
0 < \texttt{above} < 1 & \quad \text{dilates in a concave shape above the crossover} \\
\texttt{above} = 1 & \quad \text{produces a linear shape (neither convex, nor concave)} \\
\texttt{above} > 1 & \quad \text{concentrates in a convex shape above the crossover}
\end{align*}

Usually, \texttt{below} and \texttt{above} have equal values, unless specific reasons exist to make them different.

**Value**

A numeric vector of set membership scores, either crisp (starting from 0 with increments of 1), or fuzzy numeric values between 0 and 1.

**Author(s)**

Adrian Dusa

**References**


**Examples**

```r
# generate heights for 100 people
# with an average of 175cm and a standard deviation of 10cm
set.seed(12345)
x <- rnorm(n = 100, mean = 175, sd = 10)
```
calibrate

```r
cx <- calibrate(x, type = "crisp", thresholds = 175)
plot(x, cx, main="Binary crisp set using 1 threshold",
     xlab = "Raw data", ylab = "Calibrated data", yaxt="n")
axis(2, at = 0:1)

cx <- calibrate(x, type = "crisp", thresholds = c(170, 180))
plot(x, cx, main="3 value crisp set using 2 thresholds",
     xlab = "Raw data", ylab = "Calibrated data", yaxt="n")
axis(2, at = 0:2)

# calibrate to a increasing, s-shaped fuzzy-set
cx <- calibrate(x, thresholds = "e=165, c=175, i=185")
plot(x, cx, main = "Membership scores in the set of tall people",
     xlab = "Raw data", ylab = "Calibrated data")

# calibrate to a decreasing, s-shaped fuzzy-set
cx <- calibrate(x, thresholds = "i=165, c=175, e=185")
plot(x, cx, main = "Membership scores in the set of short people",
     xlab = "Raw data", ylab = "Calibrated data")

# when not using the logistic function, linear increase
cx <- calibrate(x, thresholds = "e=165, c=175, i=185", logistic = FALSE)
plot(x, cx, main = "Membership scores in the set of tall people",
     xlab = "Raw data", ylab = "Calibrated data")

# tweaking the parameters "below" and "above" the crossover,
# at value 3.5 approximates a logistic distribution, when e=155 and i=195
cx <- calibrate(x, thresholds = "e=155, c=175, i=195", logistic = FALSE,
                below = 3.5, above = 3.5)
plot(x, cx, main = "Membership scores in the set of tall people",
     xlab = "Raw data", ylab = "Calibrated data")

# calibrate to a bell-shaped fuzzy set
cx <- calibrate(x, thresholds = "e1=155, c1=165, i1=175, i2=175, c2=185, e2=195",
                below = 3, above = 3)
plot(x, cx, main = "Membership scores in the set of average height",
     xlab = "Raw data", ylab = "Calibrated data")

# calibrate to an inverse bell-shaped fuzzy set
cx <- calibrate(x, thresholds = "i1=155, c1=165, e1=175, e2=175, c2=185, i2=195",
                below = 3, above = 3)
plot(x, cx, main = "Membership scores in the set of non-average height",
     xlab = "Raw data", ylab = "Calibrated data")
```
# the default values of "below" and "above" will produce a triangular shape
cx <- calibrate(x, thresholds = "e1=155, c1=165, i1=175, i2=175, c2=185, e2=195")
plot(x, cx, main = "Membership scores in the set of average height",
     xlab = "Raw data", ylab = "Calibrated data")

# different thresholds to produce a linear trapezoidal shape
cx <- calibrate(x, thresholds = "e1=155, c1=165, i1=172, i2=179, c2=185, e2=195")
plot(x, cx, main = "Membership scores in the set of average height",
     xlab = "Raw data", ylab = "Calibrated data")

# larger values of above and below will increase membership in or out of the set
cx <- calibrate(x, thresholds = "e1=155, c1=165, i1=175, i2=175, c2=185, e2=195",
                below = 10, above = 10)
plot(x, cx, main = "Membership scores in the set of average height",
     xlab = "Raw data", ylab = "Calibrated data")

# while extremely large values will produce virtually crisp results
cx <- calibrate(x, thresholds = "e1=155, c1=165, i1=175, i2=175, c2=185, e2=195",
                below = 10000, above = 10000)
plot(x, cx, main = "Binary crisp scores in the set of average height",
     xlab = "Raw data", ylab = "Calibrated data", yaxt="n")
axis(2, at = 0:1)
abline(v = c(165, 185), col = "red", lty = 2)

# check if crisp
round(cx, 0)

# using the empirical cumulative distribution function
# require manually setting logistic to FALSE
cx <- calibrate(x, thresholds = "e=155, c=175, i=195", logistic = FALSE,
                ecdf = TRUE)
plot(x, cx, main = "Membership scores in the set of tall people",
     xlab = "Raw data", ylab = "Calibrated data")

deMorgan

**Negate Boolean expressions**

**Description**

This function negates an boolean expression written in Disjunctive Normal Form.

**Usage**

```r
demorgan(expression, snames = ",", nolevels, use.tilde = FALSE)
```
Arguments

expression  A string representing an expression written in SOP, or an object of class "qca".
snames  A string containing the sets’ names, separated by commas.
noflevels  Numerical vector containing the number of levels for each set.
use.tilde  Logical, use tilde for negation with bivalent variables.

Details

In Boolean algebra, there are two transformation rules named after the British mathematician Augustus De Morgan. These rules state that:

1. The complement of the union of two sets is the intersection of their complements.
2. The complement of the intersection of two sets is the union of their complements.

In "normal" language, these would be written as:

1. "not (A and B)" = "(not A) or (not B)"
2. "not (A or B)" = "(not A) and (not B)"

Based on these two laws, any Boolean expression written in disjunctive normal form can be transformed into its negation.

It is also possible to negate all models and solutions from the result of a Boolean minimization from function eqmcc(). The resulting object, of class "qca", is automatically recognised by this function (provided the minimization is Boolean).

The products should normally be split by using a star * sign, otherwise the sets’ names will be considered the individual letters in alphabetical order, unless they are specified via snames.

To negate multilevel expressions, the argument noflevels is required.

Value

A list with the following two components:

- **initial**  The initial expression.
- **negated**  The negation of the initial expression.

If \( x \) is an object of type "qca", the result is a list of solutions.

Author(s)

Adrian Dusa

References

See Also
eqmcc

Examples

# example from Ragin (1987, p.99)
demorgan("AC + B*C")

# with different intersection operators
demorgan("AB*EF + ~CD*EF")

# using an object of class "qca" produced with eqmcc()
data(LC)
cLC <- eqmcc(LC, "SURV", include = "?")
demorgan(cLC)

# parsimonious solution
pLC <- eqmcc(LC, "SURV", include = "?")
demorgan(pLC)

eqmcc QCA minimization using the enhanced Quine-McCluskey algorithm

Description

This function performs the QCA minimization of some causal conditions, with respect to an outcome. It is called “eqmcc” from the (e)nhanced Quine-McCluskey, a different algorithm which returns the same, exact solutions.

Usage

eqmcc(data, outcome = "", conditions = "", relation = "suf", n.cut = 1,
   incl.cut = 1, explain = "1", include = "", row.dom = FALSE, all.sol = FALSE,
   omit = NULL, dir.exp = "", details = FALSE, show.cases = FALSE,
   inf.test = "", use.tilde = FALSE, use.letters = FALSE, ...)

Arguments

data A truth table object or a data frame containing calibrated causal conditions and an outcome.
outcome A string containing the outcome(s) name(s), separated by commas.
conditions: A single string containing the conditions' (columns) names separated by commas, or a character vector of conditions' names.

relation: The set relation to outcome, either "suf" or "sufnec".

n.cut: The minimum number of cases under which a truth table row is declared as a remainder.

incl.cut: The inclusion cutoff(s): either a single value for the presence of the output, or a vector of length 2, the second for the absence of the output.

explain: A vector of output values to explain.

include: A vector of other output values to include in the minimization process.

row.dom: Logical, perform row dominance in the prime implicants’ chart to eliminate redundant prime implicants.

all.sol: Derive all possible solutions, irrespective of the number of prime implicants.

omit: A vector of row numbers from the truth table, or a matrix of causal combinations to omit from the minimization process.

dir.exp: A vector of directional expectations to derive intermediate solutions.

details: Logical, print more details about the solution.

show.cases: Logical, print case names.

inf.test: Specifies the statistical inference test to be performed (currently only "binom") and the critical significance level. It can be either a vector of length 2, or a single string containing both, separated by a comma.

use.tilde: Logical, use tilde for negation with bivalent variables.

use.letters: Logical, use letters instead of causal conditions’ names.

...: Other arguments (mainly for backwards compatibility).

Details

The argument data can be either a truth table object (created with the function truthTable()) or a data frame containing calibrated columns.

Calibration can be either crisp, with 2 or more values starting from 0, or fuzzy with continuous scores from 0 to 1. Raw data containing relative frequencies can also be continuous between 0 and 1, but these are not calibrated, fuzzy data.

Some columns can contain the placeholder "~", indicating a “don’t care”, which is used to indicate the temporal order between other columns in tQCA. These special columns are not causal conditions, hence no parameters of fit will be calculated for them.

The argument outcome specifies the column name to be explained. If the outcome is a multivalue column, it can be specified in curly bracket notation, indicating the value to be explained (the others being automatically converted to zero).

The outcome can be negated using a tilde operator ~X. The logical argument neg.out is now deprecated, but still backwards compatible. Replaced by the tilde in front of the outcome name, it controls whether outcome is to be explained or its negation.

If the outcome column is multi-value, the argument outcome should use the standard curly-bracket notation X{value}. Multiple values are allowed, separated by a comma (for example X{1,2}).
Negation of the outcome can also be performed using the tilde ~ operator, for example \(~x\{1,2\}\), which is interpreted as: "all values in X except 1 and 2" and it becomes the new outcome to be explained.

Using both `neg.out = TRUE` and a tilde ~ in the outcome name don’t cancel each other out, either one (or even both) signaling if the outcome should be negated.

This function supports multiple outcomes, in which case all of them should also be specified in the argument `conditions`.

The argument `conditions` specifies the causal conditions’ names among the other columns in the data. For backwards compatibility, this argument also accepts a character vector of condition variables’ names. When this argument is not specified, all other columns except for the outcome are taken as causal conditions (and in case there are multiple outcomes, all columns are considered causal conditions).

A good practice advice is to specify both outcome and conditions as upper case letters. It is possible, in a next version, to negate outcomes using lower case letters, situation in which it really does matter how the outcome and/or conditions are specified.

The argument `relation` is used to identify solutions which are sufficient for the outcome. When using `relation = "suf"`, the function will return all solutions which are sufficient for the outcome (whether necessary or not). If using `relation = "sufnec"`, only those solutions which are both sufficient and necessary will be returned.

The argument `n.cut` specifies the frequency threshold under which a truth table row is coded as a remainder, irrespective of its inclusion score.

The argument `incl.cut` replaces the (deprecated, but still backwards compatible) former arguments `incl.cut1` and `incl.cut0`. Most of the analyses use the inclusion cutoff for the presence of the output (code "1"). When users need both inclusion cutoffs (see below), `incl.cut` can be specified as a vector of length 2, in the form: `c(ic1, ic0)` where:

- `ic1` is the inclusion cutoff for the presence of the output, a minimum sufficiency inclusion score above which the output value is coded with "1".
- `ic0` is the inclusion cutoff for the absence of the output, a maximum sufficiency inclusion score below which the output value is coded with "0".

If not specifically declared, the argument `ic0` is automatically set equal to `ic1`, but otherwise `ic0` should always be lower than `ic1`.

Using these two cutoffs, the observed combinations are coded with:

- "1" if they have an inclusion score above `ic1`
- "C" if they have an inclusion score below `ic1` and above `ic0` (contradiction)
- "0" if they have an inclusion score below `ic0`

The argument `explain` specifies the output values corresponding to the truth table rows which enter
in the minimization process. Such values can be "$1", "$C", "$\emptyset", "$1, C" and "$\emptyset, C" but not "$1, \emptyset" and "$1, \emptyset, C". Note that for "$\emptyset", "$C" and "$\emptyset, C" configurations will be reduced but no solution details printed.

The argument *include* specifies which other truth table rows are included in the minimization process. Most often, the remainders are included but any value accepted in the argument *explain* is also accepted in the argument *include*.

The argument *row.dom* is used to further eliminate redundant prime implicants when solving the PI chart, applying the principle of row dominance: if a prime implicant $X$ covers the same configurations as another prime implicant $Y$ and in the same time covers other configurations which $Y$ does not cover, then $Y$ is redundant and eliminated.

When solving the PI chart, the algorithm finds the minimal number of prime implicants needed ($k$) to cover all configurations, then finds all possible pairs of $k$ prime implicants which do cover those configurations. The argument *all.sol* presents all possible combinations of $n$ prime implicants which solves the PI chart, where $n \geq k$.

*all.sol* deactivates the argument *row.dom*, thus inflating the number of possible solutions. Depending on the complexity of the PI chart, sometimes it is not even possible to get all possible solutions.

The argument *omit* is used to exclude truth table rows from the minimization process, from the positive configurations and/or from the remainders. It can be specified as a vector of truth table line numbers, or as a matrix of causal combinations.

The argument *dir.exp* is used to specify directional expectations, as described by Ragin (2003). They can be specified as a single string, with values separated by commas. For multi-value directional expectations, they are specified together, separated by semicolons. The total length of the directional expectations must match the number of causal conditions specified in the analysis, using a dash "$-$" if there are no particular expectations for a specific causal condition.

Activating the *details* argument has the effect of printing parameters of fit for each prime implicant and each overall solution, the essential prime implicants being listed in the top part of the table. It also prints the truth table, in case the argument *data* has been provided as a data frame instead of a truth table object.

When argument *show.cases* is set to TRUE, the case names will be printed at their corresponding row in the truth table, and also at their corresponding prime implicants in the table containing the parameters of fit. Cases separated by commas belong to the same truth table row, while groups separated by semicolons belong to different truth table rows.

The argument *inf.test* combines the inclusion score with a statistical inference test, in order to assign values in the output column from the truth table (assuming the argument *data* is not already a truth table object). For the moment, it is only the binomial test, which needs crisp data (it doesn’t work with fuzzy sets). For a given (specified) critical significance level, the output for a truth table row will be coded as:

- "$1$" if the true inclusion score is significantly higher than $ic1$,
- "$C$" contradiction, if the true inclusion score is not significantly higher than $ic1$ but significantly higher than $ic0$,
- "$\emptyset$" if the true inclusion score is not significantly higher than $ic0$. 
It should be noted that statistical tests perform well only when the number of cases is large, otherwise they are usually not significant. For a low number of cases, depending on the inclusion cutoff value(s), it will be harder to code a value of "1" in the output, and also harder to obtain contradictions if the true inclusion is not significantly higher than icθ.

The argument use.letters controls using the original names of the causal conditions, or replace them by single letters in alphabetical order. If the causal conditions are already named with single letters, the original letters will be used.

Starting with version 2.6, the package employs a different search algorithm based on consistency cubes, considering all possible combinations of causal conditions and all possible combinations of their respective levels. The structure of the input dataset (number of causal conditions, number of levels, number of unique rows in the truth table) has a direct implication on the search time, as all of those characteristics become entry parameters when calculating all possible combinations.

Value

An object of class "qca" when using a single outcomes, or class "mqca" when using multiple outcomes. These objects are lists having the following components:

- **tt** The truth table object.
- **excluded** The line number(s) of the negative configuration(s).
- **initials** The initial positive configuration(s).
- **PIs** The prime implicant(s).
- **PIchart** A list containing the PI chart(s).
- **solution** A list of solution(s).
- **essential** A list of essential PI(s).
- **pims** A list of PI membership scores.
- **SA** A list of simplifying assumptions.
- **i.sol** A list of components specific to intermediate solution(s), each having a prime implicants chart, prime implicant membership scores, (non-simplifying) easy counterfactuals and difficult counterfactuals.

Author(s)

Adrian Dusa

References


See Also  

truthTable, factorize

Examples

```r
# ------
# Lipset binary crisp data
data(LC)

# the associated truth table
ttlLC <- truthTable(LC, "SURV", sort.by = "incl, n")
ttlLC

# conservative solution (Rihoux & De Meur 2009, p.57)
cLC <- eqmcc(ttlLC)
cLC

# view the Venn diagram for the associated truth table
library(venn)
venn(cLC)

# add details and case names
eqmcc(ttlLC, details = TRUE, show.cases = TRUE)

# negating the outcome
ttlLCn <- truthTable(LC, "~SURV", sort.by = "incl, n")
eqmcc(ttlLCn)

# using a tilde instead of upper/lower case names
eqmcc(ttlLCn, use.tilde = TRUE)

# parsimonious solution, positive output
pLC <- eqmcc(ttlLC, include = "?", details = TRUE, show.cases = TRUE)
pLC

# the associated simplifying assumptions
```
plC$SA

# parsimonious solution, negative output
plCn <- eqmcc(ttlCn, include = "?", details = TRUE, show.cases = TRUE)
plCn

# -----
# Lipset multi-value crisp data (Cronqvist & Berg-Schlosser 2009, p.80)
data(LH)

# truth table
ttLM <- truthTable(LH, "SURV", conditions = "DEV, URB, LIT, IND",
                   sort.by = "incl", show.cases = TRUE)

# conservative solution, positive output
eqmcc(ttlLM, details = TRUE, show.cases = TRUE)

# parsimonious solution, positive output
eqmcc(ttlLM, include = "?", details = TRUE, show.cases = TRUE)

# negate the outcome
ttLMn <- truthTable(LH, "~SURV", conditions = "DEV, URB, LIT, IND",
                    sort.by = "incl", show.cases = TRUE)

# conservative solution, negative output
eqmcc(ttlLMn, details = TRUE, show.cases = TRUE)

# parsimonious solution, positive output
eqmcc(ttlLMn, include = "?", details = TRUE, show.cases = TRUE)

# -----
# Lipset fuzzy sets data (Ragin 2009, p.112)
data(LF)

# truth table using a very low inclusion cutoff
ttLF <- truthTable(LF, "SURV", incl.cut = 0.7,
                   show.cases = TRUE, sort.by="incl")

# conservative solution
eqmcc(ttlLF, details = TRUE, show.cases = TRUE)

# parsimonious solution
eqmcc(ttlLF, include = "?", details = TRUE, show.cases = TRUE)

# intermediate solution using directional expectations
ilF <- eqmcc(ttlLF, include = "?", details = TRUE, show.cases = TRUE,
             dir.exp = "1,1,1,1,1")
# -----
# Cebotari & Vink (2013, 2015)
data(CVF)

`ttCVF <- truthTable(CVF, outcome = "PROTEST", incl.cut = 0.8, show.cases = TRUE, sort.by = "incl, n")`

`pCVF <- eqmcc(ttCVF, include = "?", details = TRUE, show.cases = TRUE)`

`pCVF`

# inspect the PI chart
`pCVF$PIchart`

# DEMOC*ETHFRACT*poldis is dominated by DEMOC*ETHFRACT*GEOCON
# using row dominance to solve the PI chart
`pCVFrd <- eqmcc(ttCVF, include = "?", row.dom = TRUE, details = TRUE, show.cases = TRUE)`

# plot the prime implicants on the outcome
`pims <- pCVFrd$pims`

`par(mfrow = c(2, 2))`
`for(i in 1:4) {
  XYplot(pims[, i], CVF$PROTEST, cex.axis = 0.6)
}`

# -----
# temporal QCA (Ragin & Strand 2008)
data(RS)
eqmcc(RS, "REC", details = TRUE, show.cases = TRUE)

---

**export**

*Export a dataframe to a file or a connection*

---

**Description**

This function is a wrapper to `write.table()`, see the details section.

**Usage**

`export(x, file = "", ...)`

**Arguments**

- `x` The object to be written
- `file` A character string containing the path to the file to be created
- `...` Same arguments that are used in `write.table()`
factorize

Details

The default convention for `write.table()` is to add a blank column name for the row names, but (despite it is the convention used for CSV files to be read by spreadsheets) that doesn’t always work with all spreadsheets or other programs that attempt to import the result of `write.table()`.

This function acts as if `write.table()` was called, with only one difference: if row names are present in the dataframe (i.e. any of them should be different from the default row numbers), the final result will display a new column called `cases` in the first position, except the situation that another column called cases already exists in the data, when the row names will be completely ignored.

If not otherwise specified, an argument `sep = ",",` is added by default.

The argument `row.names` is set to `FALSE` by default, since a new column is added anyways if possible.

Since this function pipes everything to `write.table()` anyway, the argument `file` can also be a connection open for writing. "" indicates output to the console.

Author(s)

Adrian Dusa

See Also

The “R Data Import/Export” manual.

`write.table`

---

factorize  
\textit{Factorize Boolean expressions}

Description

This function finds all combinations of common factors in a Boolean expression written in SOP - Sum Of Products form.

Usage

```r
factorize(expression, snames = "", nolevels, pos = FALSE, tilde, ...)
```

Arguments

- `expression`: A string containing the SOP expression, or an object of class "qca".
- `snames`: A string containing the sets’ names, separated by commas.
- `nolevels`: Numerical vector containing the number of levels for each set.
- `pos`: Logical, if possible factorize using product(s) of sums.
- `tilde`: Logical, use a tilde to negate the expression.
- `...`: Other arguments (mainly for backwards compatibility).
Details

Factorization is a process of finding common factors in a Boolean expression, written in DNF - disjunctive normal form, or in a sum of products (SOP) form. The factorization is also possible using products of sums (POS) form, whenever possible.

Starting with version 2.1 of the package, a number of arguments have been deprecated. For example, prod.split, because intersections (products) are indicated by the standard * sign, and sometimes products are not indicated by anything (for example when the set names are single letters).

The argument snames is only needed when products are not indicated by any sign at all, and the set names have more than one letter each (see function translate() for more details).

The number of levels in noflevels is needed only when negating multivalue conditions, and it should complement the snames argument.

If expression is an object of class “qca” (the result of the eqmcc() minimization function), this function performs a factorization for each of the minimized solutions.

Value

A list with two components:

- initial The input expression.
- factored All possible factorizations of the input expression.

Author(s)

Adrian Dusa

References


See Also

translate

Examples

# results presented in alphabetical order
factorize("one*two*four + one*three + three*four")

# to preserve a certain order of the set names
factorize("one*two*four + one*three + three*four", snames = "one, two, three, four")

factorize("~one*two*four + ~one*three + three*~four", snames = "one, two, three, four")

# using pos - products of sums
factorize("a*c + a*D + b*c + b*D", pos = TRUE)

# using an object of class "qca" produced with eqmcc()
data(CVF)
pCVF <- eqmcc(CVF, outcome = "PROTEST", incl.cut = 0.8,
             include = "?", use.letters = TRUE)
factorize(pCVF)

## Not run:
# using an object of class "demorgan" produced with demorgan()
factorize(demorgan(pCVF))

## End(Not run)

---

findTh

**Find calibration thresholds**

**Description**

The purpose of this function is to automatically find calibration thresholds for a numerical causal condition, to be split into separate groups.

**Usage**

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

**Arguments**

- **x** A numerical causal condition.
- **n** The number of thresholds to find.
- **hclustm** The agglomeration (clustering) method to be used.
- **distm** The distance measure to be used.
- **...** Other arguments (mainly for backwards compatibility).

**Details**

The process of calibration into crisp sets assumes expert knowledge about the best threshold(s) which separates the raw data into the most meaningful groups.

In the absence of such knowledge, an automatic procedure might help grouping the raw data according to statistical clustering techniques.

The number of groups to split depends on the number of thresholds: one thresholds splits into two groups, two thresholds splits into three groups etc.

Previous versions of this function had an argument named `groups` instead of argument `n`, but they are backwards compatible.
For more details about how many groups can be formed with how many thresholds, see ?cutree. More details about the clustering techniques used in this function are found using ?hclust, and also more details about different distance measures can be found with ?dist. This function uses their default values.

Value
A numeric vector of length n.

Author(s)
Adrian Dusa

See Also
cutree, hclust, dist

Examples

# hypothetical list of country GDPS
gdp <- c(460, 500, 900, 2000, 2100, 2400, 15000, 16000, 20000)

# find one threshold to separate into two groups
findTh(gdp)
# 8700

# find two thresholds to separate into two groups
findTh(gdp, n = 2)
# 8700 18000

# using different clustering methods
findTh(gdp, n = 2, hclustm = "ward.D2", distm = "canberra")
# 1450 8700

Description
These functions perform logical "and" and "or", either for binary crisp or fuzzy set membership scores.
Usage

fuzzyand(...)
fuzzyor(...)

Arguments

Two or more numerical (calibrated) membership scores, or a matrix / data frame of calibrated columns.

Value

A numerical vector of resulting values

Author(s)

Adrian Dusa

Examples

# -----
data(CVF)

# DEMOC+GEOCON+NATPRIDE
fuzzyand(CVF$DEMOC, CVF$GEOCON, CVF$NATPRIDE)

# same thing with
fuzzyand(CVF[, c(1,3,5)])

# DEMOC+geocon+NATPRIDE
fa <- fuzzyand(CVF$DEMOC, 1 - CVF$GEOCON, CVF$NATPRIDE)
fa

attr(fa, "name")

# ETHFRAC + POLDIS
fuzzyor(CVF$ETHFRACT, CVF$POLDIS)

# same thing with
fuzzyor(CVF[, c(2,4)])

# ETHFRAC + poldis
fo <- fuzzyor(CVF$ETHFRACT, 1 - CVF$POLDIS)
fo

attr(fo, "name")
Implicant matrix functions: allExpressions, createMatrix, getRow

Functions Related to the Implicant Matrix

Description

This is a set of functions dedicated to the implicant matrix, a space where all causal configurations and their minimized solutions are found.

They can produce all possible implicants and prime implicants, or all possible combinations for a specific number of causal conditions and their number of values (either binary or multi-value).

Usage

allExpressions(noflevels, arrange = FALSE, depth = NULL, raw = FALSE, ...)
createMatrix(noflevels, ...)
getRow(noflevels, row.no, zerobased = FALSE)

Arguments

- noflevels: The number of levels (values) for each causal condition.
- arrange: Logical, if TRUE the result matrix is arranged for visual inspection.
- depth: Integer, an upper number of causal conditions to form expressions with.
- raw: Logical, if TRUE it returns the matrix indicating which conditions have been minimized, using MQ.
- row.no: A vector, the desired row numbers.
- zerobased: Logical, if TRUE the first row number is zero.
- ...: Other arguments.

Details

A truth table for binary crisp conditions is a matrix with $2^k$ rows, where $k$ is the number of causal conditions.

For multi-value causal conditions, the same equation can be generalised to:

$v_1 \ast v_2 \ast \ldots \ast v_k$

where $v$ is the number of values (levels) for every causal condition from 1 to $k$.

Implicant matrices contain all rows from the truth table, plus all of their supersets, (all implicants and prime implicants), including the empty set (Dusa 2007, 2010).

For a binary crisp set procedure, there are $3^k - 1$ possible expressions (groupings), see Ragin (2010). Including the empty set (the situation when all causal conditions have been minimized), the implicant matrix consists of exactly $3^k$ rows, including the truth table configurations.

In fact, $3^k$ is also obtained by the product:
Implicant matrix functions: allExpressions, createMatrix, getRow

\[(2 + 1) \times (2 + 1) \times \ldots \times (2 + 1)\]

For multi-value causal conditions, the same equation can be generalised to:

\[(v_1 + 1) \times (v_2 + 1) \times \ldots \times (v_k + 1)\]

where every number of levels in each causal conditions is incremented with 1, to allow coding the minimization of literals in each (prime) implicant (see examples).

The function allExpressions() creates a matrix which contains all possible implicants and prime implicants, displayed in the original values form using the code ~1 to point the minimized literals, while the other functions use the code 0, all other values being incremented with 1.

The function createMatrix() creates a base matrix for truth tables and implicant matrices.

When the argument arrange is activated, the output is arranged in the increasing order of the number of conditions which form conjunctions, up to the maximum number specified by the argument depth (which if NULL it is considered equal to the number of columns in the matrix).

The function getRow() takes the number of a row in the truth table or implicant matrix (in its decimal form), and transforms it into its binary (or multi-base) representation, as a configuration of binary or multi-values for each causal condition.

Note that R is a 1-based language (all numbers start from 1), and similarly positions in vectors and matrices start with 1. For this reason, although (mathematically) the binary representation of the decimal number 0 (for example, at three causal conditions) is 0 0 0, in R that would be the “first” line in the implicant matrix, therefore 0 0 0 is translated into the number 1.

Value

A matrix with \(k\) columns and:

- \(v_1 \times v_2 \times \ldots \times v_k\) rows if a truth table;
- \((v_1 + 1) \times (v_2 + 1) \times \ldots \times (v_k + 1)\) rows if an implicant matrix;
- \(x\) rows, equal to the length of row.no.

Author(s)

Adrian Dusa

References


See Also

expand.grid
Examples

```r
# three binary causal conditions, having two levels each: 0 and 1
noflevels <- c(2, 2, 2)

# for three binary causal conditions
allExpressions(noflevels)

# the same matrix, this time arranged better
# (last rows represent the truth table)
allExpressions(noflevels, arrange = TRUE)

# show only the implicants (excluding the truth table)
allExpressions(noflevels, arrange = TRUE, depth = 2)

# using the raw form
allExpressions(noflevels, raw = TRUE)

# create a truth table based on 3 conditions
createMatrix(noflevels)

# its implicant matrix
createMatrix(noflevels + 1)

# create a truth table based on 3 conditions where the second has three levels
createMatrix(c(2, 3, 2))

# deriving rows
rows <- c(2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17)
mat <- getRow(noflevels + 1, rows)  # note the +1
rownames(mat) <- rows
colnames(mat) <- c("A", "B", "C")
mat

# implicant matrix  normal values

# A B C | A B C
# 2 0 0 1 | 2 - - 0  c
# 4 0 1 0 | 4 - 0  -  b
# 5 0 1 1 | 5 - 0  bc
# 7 0 2 0 | 7 - 1  -  B
# 8 0 2 1 | 8 - 1  Bc
# 10 1 0 0 | 10 0  -  a
# 11 1 0 1 | 11 0  -  ac
# 13 1 1 0 | 13 0  -  ab
# 14 1 1 1 | 14 0 0  abc
# 16 1 2 0 | 16 1  -  aB
# 17 1 2 1 | 17 0 1  aBc
```
Interpret SOP expressions: translate, compute, findRows, sop

Functions to interpret or create a SOP expression

Description

These functions interpret an expression written in a SOP (sum of products) form, for both crisp and multivalue QCA. The function `translate()` translates the expression into a standard (canonical) SOP form using a matrix of implicants, while `compute()` uses the first to compute the scores based on a particular dataset.

The function `findRows()` takes a QCA expression written in SOP form, and applies it on a truth table to find all rows that match the pattern in the expression.

For crisp sets notation, upper case letters are considered the presence of that causal condition, and lower case letters are considered the absence of the respective causal condition. Tilde is recognized as a negation, even in combination with upper/lower letters.

Functions similar to `translate()` and `compute()` have initially been written by Jirka Lewandowski (2015) but the actual code in these functions has been completely re-written and expanded with more extensive functionality (see details and examples below).

The function `sop()` transforms any expression (most notably a POS product of sums) into a sum of products, minimizing it to the simplest equivalent logical expression.

Usage

- `translate(expression = "", snames = "", noflevels, data, ...)`
- `compute(expression = "", data, separate = FALSE)`
- `findRows(expression = "", ttobj, remainders = FALSE)`
- `sop(expression = "", snames = "", noflevels, data)`

Arguments

- `expression` String: a QCA expression written in sum of products form.
- `snames` A string containing the sets’ names, separated by commas.
- `noflevels` Numerical vector containing the number of levels for each set.
- `data` A dataset with binary cs, mv and fs data.
- `separate` Logical, perform computations on individual, separate paths.
- `ttobj` A truth table, an object of class "tt".
- `remainders` Logical, find remainders only.
- `...` Other arguments, mainly for backwards compatibility.
Details

A SOP ("sum of products") is also known as a DNF ("disjunctive normal form"), or in other words a "union of intersections", for example \( A \times D + B \times C \).

The same expression can be written in multivalue notation: \( A\{1\} \times \{D\{1\} \ + \ B\{1\} \times C\{\emptyset\}. \) Both types of expressions are valid, and yield the same result on the same dataset.

For multivalue notation, causal conditions are expected as upper case letters, and they will be converted to upper case by default. Expressions can contain multiple values to translate, separated by a comma. If \( B \) was a multivalue causal condition, an expression could be: \( A\{1\} + B\{1, 2\} \times C\{\emptyset\}. \)

In this example, all values in \( B \) equal to either 1 or 2 will be converted to 1, and the rest of the (multi)values will be converted to 0.

The function automatically detects the use of tilde "~" as a negation for a particular causal condition. \(~A\) does two things: it identifies the presence of causal condition \( A \) (because it was specified as upper case) and it recognizes that it must be negated, because of the tilde. It works even combined with lower case names: \(~a\), which is interpreted as \( a \).

To negate a multivalue condition using a tilde, the number of levels should be supplied (see examples below). Improvements in version 2.5 allow for intersections between multiple levels of the same condition. For a causal condition with 3 levels (0, 1 and 2) the following expression \(~A\{0, 2\} \times A\{1, 2\} \) is equivalent with \( A\{1\} \), while \( A\{0\} \times A\{1\} \) results in the empty set.

The number of levels, as well as the set names can be automatically detected from a dataset via the argument \( \text{data} \). Arguments \( \text{snames} \) and \( \text{noflevels} \) have precedence over \( \text{data} \), when specified.

The use of the product operator \( \times \) is redundant when causal names’ names are single letters (for example \( AD + Bc \)), and is also redundant for multivalue data, where product terms can be separated by using the curly brackets notation.

When causal names are binary and their names have multiple letters (for example \( AA + CC \times bb \)), the use of the product operator \( \times \) is preferable but the function manages to translate an expression even without it (\( AA + CC \times bb \)) by searching deep in the space of the conditions’ names, at the cost of slowing down for a high number of causal conditions. For this reason, an arbitrary limit of 7 causal names is imposed, to write an expression with.

Value

For the function \( \text{translate}() \), a matrix containing the implicants on the rows and the set names on the columns, with the following codes:

- 0 absence of a causal condition
- 1 presence of a causal condition
- -1 causal condition was eliminated

The matrix was also assigned a class "translate", to avoid printing the -1 codes. The mode of this matrix is character, to allow printing multiple levels in the same cell, such as "1,2".

Author(s)

Adrian Dusa
References

Jirka Lewandowski (2015) QCAtools: Helper functions for QCA in R. R package version 0.1

Examples

```r
translate("A + B*C")
# same thing in multivalue notation
translate("A(1) + B(1)*C(1))")
# using upper/lower letters
translate("A + b*C")
# the negation with tilde is recognised
translate("~A + b*C")
# even in combination of upper/lower letters
translate("~A + ~b*C")
# and even for multivalue variables
# in multivalue notation, the product sign * is redundant
translate("C(1) + T(2) + T(1)V(0) + C(0)")
# negation of multivalue sets requires the number of levels
translate("~A(1) + ~B(0)*C(1)", noflevels = c(2, 2, 2))
# multiple values can be specified
translate("C(1) + T(1,2) + T(1)V(0) + C(0)")
# or even negated
translate("C(1) + ~T(1,2) + T(1)V(0) + C(0)", snames = "C, T, V", noflevels = c(2,3,2))
# if the expression does not contain the product sign *
# snames are required to complete the translation
translate("AB + cD", snames = "A, B, C, D")
# otherwise snames are not required
translate("PER*ECT + string")
# snames are required
translate("PERECT + string", snames = "PER, FECT, STR, ING")
# it works even with overlapping columns
# SU overlaps with SUB in SUBER, but the result is still correct
translate("SUBER + subset", "SU, BER, SUB, SET")

## Not run:
# error because combinations of condition names clash
translate("SUPER + subset", "SUP, ER, SU, PER, SUB, SET")

## End(Not run)
```
# to print _all_ codes from the standard output matrix (obj <- translate("A + b*C")) print(obj, original = TRUE) # also prints the -1 code

# for compute()
data(CVF)
compute("natpride + GEOCON", data = CVF)

# calculating individual paths
compute("natpride + GEOCON", data = CVF, separate = TRUE)

# for findRows()
data(LC)
 ttlC <- truthTable(LC, "SURV", show.cases = TRUE)
 findRows("DEV*ind*STB", ttlC)
 findRows("DEV*ind*STB", ttlC, remainders = TRUE)

# for sop()
sop("(A + B)(A + ~B)") # result is "A"

# to force a certain order of the set names
sop("(URB + LIT*~DEV)(~LIT + ~DEV)", snames = "DEV, URB, LIT")

# multilevel conditions can also be specified (and negated)
sop("(A(1) + ~B(0))(B(1) + C(0))", snames = "A, B, C", noflevels = c(2, 3, 2))

---

**intersection**

**Intersect two expressions**

**Description**

This function takes two QCA expressions (combinations of conjunctions and disjunctions) and finds their intersection.

**Usage**

intersection(e1 = "", e2 = "", snames = "")
**Arguments**

- `e1` String, the first expression to intersect.
- `e2` String, the first expression to intersect.
- `snames` A string containing the set names, separated by commas.

**Details**

The aim of this function is to provide a software implementation of the intersection examples presented by Ragin (1987: 144-147).

It can only deal with expressions using bivalent crisp-set conditions only, it doesn’t deal with multivalent crisp-set conditions because it requires additional information about the number of levels in each causal condition.

The two expressions can be formulated in terms of either lower case - upper case notation for the absence and the presence of the causal condition, or use the tilde notation (see examples below). Usage of either of these is automatically detected, as long as both expressions use the same notation.

If the `snames` argument is provided, the result is sorted according to the order of the causal conditions (set names) in the original dataset, otherwise it sorts the causal conditions in alphabetical order.

**Author(s)**

Adrian Dusa

**References**


**Examples**

```r
# in Ragin's (1987) book, the equation E = SG + LW is the result
# of the Boolean minimization for the ethnic political mobilization.

# intersecting the reactive ethnicity perspective (R = lw)
# with the equation E (page 144)
intersection("lw", "SG + LW", snames = "S, L, W, G")

# [1] "SlwG"

# resources for size and wealth (C = SW) with E (page 145)
intersection("SW", "SG + LW", snames = "S, L, W, G")

# [1] "SWG + SLW"

# and factorized
```
PI chart functions: `makeChart`, `solveChart`

Create and solve a prime implicants chart

Description

These functions help creating a demo for a prime implicant chart, and also show how to solve it using a minimum number of prime implicants.

Usage

```r
makeChart(primes = "", configs = "", snames = "")

solveChart(chart, row.dom = FALSE, all.sol = FALSE, ...)
```

Arguments

- `primes`: A string containing prime implicants, separated by commas, or a matrix of implicants.
- `configs`: A string containing causal configurations, separated by commas, or a matrix of causal configurations in the implicants space.
- `snames`: A string containing the sets’ names, separated by commas.
- `chart`: A logical matrix (as the PI chart).
- `row.dom`: Logical, apply row dominance to eliminate redundant prime implicants.
- `all.sol`: Derive all possible solutions, irrespective of the number of PIs.
- `...`: Other arguments (mainly for backwards compatibility).
Details

A PI chart, in this package, is a logical matrix (with TRUE/FALSE values), containing the prime implicants on the rows and the starting configurations of causal conditions, on the columns, like the one produced by `makeChart()`. It is useful to determine visually which prime implicant (if any) is essential.

When primes and configs are character, the individual sets are identified using the function `translate()`, using the SOP - Sum Of Products form, which needs the set names in the absence of any other information. If products are formed using the standard * operator, specifying the set names is not mandatory.

When primes and configs are matrices, they have to be specified at implicants level, where the value 0 is interpreted as a minimized literal.

The chart is subsequently processed algorithmically by `solveChart()` to further reduce the redundant prime implicants. It applies a linear programming function from package `lpSolve`, to find the absolute minimal number $M$ of rows (prime implicants) necessary to cover all columns, then searches through all possible combinations of $M$ rows, to find those which actually cover the columns.

Since all possible combinations grow exponentially with the number of prime implicants resulted from the Quine-McCluskey minimization procedure, this function tries hard to reduce this number by first eliminating the redundant prime implicants.

When set to TRUE, the argument `rowNdom` does something like this by eliminating the dominated rows (those which cover a smaller number of columns than another, dominant prime implicant).

Value

For `makeChart`: a logical matrix.

For `solveChart`: a matrix containing all possible combinations of PI chart rows necessary to cover all its columns.

Author(s)

Adrian Dusa

References


Examples

```r
# non-standard products, it needs the set names
chart <- makeChart("A, B, c", "ABC, Abc, AbC, aBc", snames = "A,B,c")
prettyTable(chart)
#  ABC Abc AbC aBc
# A x x x -
# B x - - x
```
```r
# c - x - x

solveChart(chart)
# first and second rows (A + B)
# and first and third rows (A + c)
# A is an essential prime implicant
# A + B  A + c
# [,1] [,2]
# [1,]  1  1
# [2,]  2  3

# Quine's example, page 528
rows <- c("AB, BC, Ac, aC, abd, bcd")
cols <- c("ABCd, AbCd, ABcd, Abcd, aBCD, aBcd, aBcd, abCd, abCd, abCd")

prettyTable(chart <- makeChart(rows, cols, "A,B,C,D"))
# ABCD AbCd AbC D AbCd AbcD AbCD aBcd aBCd abCd abCd abCd abCd abCd abCd abCd
# AB x x x x - - - - - - - -
# BC x x - - - - x x - - - -
# Ac - - x x x x - - - - - -
# aC - - - - - - x x x x - -
# abd - - - - - - - - x x - -
# bcd - - - - - - x - - - -

solveChart(chart)
# [1,]  1  1  2  2
# [2,]  3  3  3  3
# [3,]  4  4  4  4
# [4,]  5  6  5  6

# using SOP standard product sign
rows <- c("EF, ~GH, IJ")
cols <- c("~EF*GH*IJ, EF*GH~IJ, ~EF*GH*IJ, EF*~GH~IJ")
prettyTable(chart <- makeChart(rows, cols))
# ~EF*GH*IJ EF*GH~IJ ~EF*GH*IJ EF*~GH~IJ
# EF - x - x
# ~GH x - - x
# IJ x - - x

solveChart(chart)
# ~GH is redundant
# EF + IJ
# [,1]
# [1,]  1
# [2,]  3

# using implicant matrices
primes <- matrix(c(2,2,1,0,2,2,0,2,2,2), nrow = 2)
configs <- matrix(c(2,2,2,1,1,2,2,2,1,2,2,2,2), nrow = 3)
```
colnames(primes) <- colnames(configs) <- LETTERS[1:5]

# the prime implicants: AbCE and ACDE
primes
#  A  B  C  D  E
# [1,] 2 1 2 0 2
# [2,] 2 0 2 2 2

# the initial causal combinations: AbCdE, AbCDE and ABCDE
configs
#  A  B  C  D  E
# [1,] 2 1 2 1 2
# [2,] 2 1 2 2 2
# [3,] 2 2 2 2 2

chartLC <- makeChart(primes, configs)
prettyTable(chartLC)
#     AbCdE  AbCDE  ABCDE
# AbCE  x     x      -
# ACDE  -     x      x

---

**pof**

*Calculate parameters of fit*

**Description**

This function returns inclusion (consistency), coverage, PRI and optionally the relevance of necessity scores.

**Usage**

```r
pof(setms, outcome, data, relation = "nec", inf.test = "", incl.cut = c(0.75, 0.5), ...)
```

**Arguments**

- `setms` A data frame of (calibrated) set memberships, or a matrix of implicants, or a vector of row numbers from the implicant matrix, or a string expression
- `outcome` The name of the outcome column from a calibrated data frame, or the actual numerical column from the data frame, representing the outcome.
- `data` The calibrated data frame, in case the outcome is a name.
- `relation` The set relation to outcome, either necessity ("nec") or sufficiency ("suf")
- `inf.test` Specifies the statistical inference test to be performed (currently only "binom") and the critical significance level. It can be either a vector of length 2, or a single string containing both, separated by a comma.
- `incl.cut` The inclusion cutoff(s): either a single value for the presence of the output, or a vector of length 2, the second for the absence of the output.
- `...` Other arguments (mainly for backward compatibility).
Details

This is one of the most flexible functions in the QCA package. Depending on particular situations, its arguments can be provided in various formats which are automatically recognized and treated accordingly.

When specified as a data frame, the argument `setms` contains any kind of set membership scores:
- calibrated causal conditions from the original data,
- membership scores from the resulting combinations (component `coms`) of function `superSubset()`,
- prime implicant membership scores (component `pims`) from function `eqmcc()`,
- any other, custom created combinations of set memberships.

When specified as a matrix, `setms` contains the crisp causal combinations similar to those found in the truth table. If some of the causal conditions have been minimized, they can be replaced by the numerical value -1 (see examples section). The number of columns in the matrix should be equal to the number of causal conditions in the original data.

More generally, `setms` can be a numerical vector of line numbers from the implicant matrix (see function `createMatrix()`) which are automatically transformed into their corresponding set membership scores.

Starting with version 2.1, `setms` can also be a string expression, written in sum of products (SOP) form.

For all situations when `setms` is something else than a data frame, it requires the original data to generate the set memberships.

If a string, the argument `outcome` is the name of the column from the original data, to be explained (it is a good practice advice to provide using upper case letters, although it will nevertheless be converted to upper case letters, by default).

If the outcome column is multi-value, the argument `outcome` should use the standard curly-bracket notation `X{value}`. Multiple values are allowed, separated by a comma (for example `X{1,2}`). Negation of the outcome can also be performed using the tilde ~ operator, for example `~X{1,2}`, which is interpreted as: “all values in X except 1 and 2” and it becomes the new outcome to be explained.

The argument `outcome` can also be a numerical vector of set membership values, either directly from the original data frame, or a recoded version (if originally multi-value).

The argument `infNtest` provides the possibility to perform statistical inference tests, comparing the calculated inclusion score with a pair of thresholds (ic1 and ic0) specified in the argument `inclNcut`. Currently, it can only perform binomial tests ("binom"), which means that data should only be provided as binary crisp (not multivalue, not fuzzy).

If the critical significance level is not provided, the default level of 0.05 is taken.

The resulting object will contain the calculated p-values (pval1 and pval0) from two separate, one-tailed tests with the alternative hypothesis that the true inclusion score is:
- greater than ic1 (the inclusion cutoff for an output value of 1)
- greater than ic0 (the inclusion cutoff for an output value of 0)

It should be noted that statistical tests are performing well only when the number of cases is large, otherwise they are usually not significant.

For the necessity relation, the standard measures of inclusion and coverage are supplemented with the RoN (Relevance of Necessity) measure, as suggested by Schneider & Wagemann’s (2012).
Starting with version 2.0, this function also accepts and recognize negation of both setms and outcome using the Boolean subtraction from 1. If the names of the conditions are provided via an optional (undocumented) argument conditions, the colnames of the setms object are negated using demorgan().

Starting with version 2.1, the logical argument neg.out is deprecated, but backwards compatible. neg.out = TRUE and a tilde ~ in the outcome name don’t cancel each other out, either one (or even both) signaling if the outcome should be negated.

When argument setms is a SOP expression, it is the only place where the everything (including the outcome) can be negated using lower case letters, with or without a tilde. Lower case letters and a tilde does cancel each other out, for example ~x is interpreted as x, while ~x is interpreted as X.

Author(s)

Adrian Dusa

References


See Also

eqmcc, superSubset, translate

Examples

# -----
# Cebotari & Vink (2013) fuzzy data
data(CVF)

conds <- CVF[, 1:5]
PROTEST <- CVF$PROTEST

# parameters of fit (default is necessity)
pof(conds, PROTEST)

# parameters of fit negating the conditions
pof(1 - conds, PROTEST)

# negating the outcome
pof(conds, 1 - PROTEST)

# parameters of fit for sufficiency
pof(conds, PROTEST, relation = “suf”)

# also negating the outcome
pof(conds, 1 - PROTEST, relation = “suf”)
recode

Recode a variable

Description

Recodes a vector (numeric, character or factor) according to a set of rules. Similar to the \texttt{recode()} function in package \texttt{car}, but more flexible.

Usage

\texttt{recode(x, rules, ...)}

```r
# ----- # standard analysis of necessity # using the "coms" component from superSubset()
ncVF <- superSubset(CVF, outcome = "PROTEST", incl.cut = 0.90, cov.cut = 0.6)

# also checking their necessity inclusion score in the negated outcome
pof(ncVF$coms, 1 - PROTEST)

# ----- # standard analysis of sufficiency # using the "pims" component from eqmcc()

cCVF <- eqmcc(CVF, "PROTEST", incl.cut = 0.8, details = TRUE)

# verify if their negations are also sufficient for the outcome
pof(1 - cCVF$pims, PROTEST)

# ----- # using a POS expression, translated using the function translate()

# notice that lower case letters means absence a causal condition
pof("natpride + GEOCON => PROTEST", data = CVF)

# same for the negation of the outcome
pof("natpride + GEOCON => ~PROTEST", data = CVF)

# same using lower letters for the negation
pof("natpride + GEOCON => protest", data = CVF)

# necessity is indicated by the reverse arrow
pof("natpride + GEOCON <= PROTEST", data = CVF)
```
Arguments

- **x**: Any kind of vector, numeric, character or factor.
- **rules**: Character string or a vector of character strings for recoding specifications
- **. . .**: Other parameters, for compatibility with other functions such as `recode()` in package `car` but also `factor()` in package `base`

Details

Similar to the `recode` function in package `car`, the recoding rules are separated by semicolons, of the form `input=output`, and allow for:

- **a single value**: `1=0`
- **a range of values**: `2:5=1`
- **a set of values**: `c(6,7,10)=2`
- **else**: everything that is not covered by the previously specified rules

Contrary to the `recode()` function in package `car`, this function allows the `:` sequence operator (even for factors), so that a rule such as `c(1,3,5:7)`, or `c(a,d,f:h)` would be valid.

Actually, since all rules are specified in a string, it really doesn’t matter if the `c()` function is used or not. For compatibility reasons it accepts it, but a more simple way to specify a set of rules is `"1,3,5:7=A; else=B"`

Special values `lo` and `hi` may also appear in the range of values, while `else` can be used with `else=copy` to copy all values which were not specified in the recoding rules.

In the package `car`, a character output would have to be quoted, like `"1:2='A'"` but that is not mandatory is this function, `"1:2=A"` would do just as well. Output values such as `"NA"` or `"missing"` are converted to `NA`.

Another difference from the `car` package: the output is **not** automatically converted to a factor even if the original variable is a factor. That option is left to the user’s decision to specify `as.factor.result`, defaulted to `FALSE`.

A capital difference is the treatment of the values not present in the recoding rules. By default, package `car` copies all those values in the new object, whereas in this package the default values are `NA` and new values are added only if they are found in the rules. Users can choose to copy all other values not present in the recoding rules, by specifically adding `else=copy` in the rules.

Since the two functions have the same name, it is possible that users loading both packages to use one instead of the other (depending which package is loaded first). In order to preserve functionality, special efforts have been made to ensure this function performs exactly as the other one (plus more).

In order to minimize possible namespace collisions with package `car`, special efforts have been done to ensure perfect compatibility with the other `recode()` function. The argument . . . allows for more arguments specific to the `car` package, such as `as.factor.result, as.numeric.result` and `levels`. In addition, it also accepts `labels` and `ordered` specific to function `factor` in package `base`.

Blank spaces outside category labels are ignored, see the last example.

Author(s)

Adrian Dusa
Examples

```r
x <- rep(1:3, 3)
x
# [1] 1 2 3 1 2 3 1 2 3
recode(x, "1:2 = A; else = B")
# [1] "A" "A" "B" "A" "A" "B" "A" "A" "B"
recode(x, "1:2 = 0; else = copy")
# [1] 0 0 0 0 0 0 0 0 0

set.seed(1234)
x <- factor(sample(letters[1:10], 20, replace = TRUE),
  levels = letters[1:10])
x
# [1] b g g g i g a c g f g f c j c i c c b c
# Levels: a b c d e f g h i j
recode(x, "b:d = 1; g:hi = 2; else = NA") # note the "hi" special value
# [1] 1 2 2 2 2 2 NA 1 2 NA 2 NA 1 2 1 2 1 1 1 1
recode(x, "a, c:f = A; g:hi = B; else = C", as.factor.result = TRUE)
# [1] C B B B B A A A B A A A A C A
# Levels: A B C
recode(x, "a, c:f = 1; g:hi = 2; else = 3", as.factor.result = TRUE,
  labels = c("one", "two", "three"), ordered = TRUE)
# [1] three two two two two one one two one
# [11] two one one two one two one one three one
# Levels: one < two < three

set.seed(1234)
categories <- c("An", "example", "that has", "spaces")
x <- factor(sample(categories, 20, replace = TRUE),
  levels = categories)
sort(x)
# [1] An  An  An  An  An  An  example
# [7] example example example that has that has that has
# [13] that has that has that has that has that has spaces
# [19] spaces spaces spaces
# Levels: An example that has spaces
recode(sort(x), "An : 'that has' = 1; spaces = 2")
# [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2

# same thing with
recode(sort(x), "An : that has = 1; spaces = 2")
# [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2
```
# treatment of "else" values
x <- 10:20

# recoding rules don't overlap all existing values, the rest are empty
recode(x, "8:15=1")

# all other values are copied
recode(x, "8:15=1; else=copy")

---

**retention**

*Compute the retention probability of a csQCA solution*

**Description**

This function computes the retention probability for a csQCA solution, under various perturbation scenarios. It only works with bivalent crisp-set data, containing the binary values 0 or 1.

**Usage**

```r
retention(data, outcome = "", conditions = "", type = "corruption",
dependent = TRUE, n.cut = 1, incl.cut = 1, p.pert = 0.5,
n.pert = 1)
```

**Arguments**

- **data** A dataset of bivalent crisp-set factors.
- **outcome** The name of the outcome.
- **conditions** A string containing the condition variables’ names, separated by commas.
- **type** Simulate corruptions of values in the conditions ("corruption"), or cases deleted entirely ("deletion").
- **dependent** Logical, if TRUE indicating DPA - Dependent Perturbations Assumption and if FALSE indicating IPA - Independent Perturbations Assumption.
- **n.cut** The minimum number of cases for a causal combination with a set membership score above 0.5, for an output function value of "0" or "1".
- **incl.cut** The minimum sufficiency inclusion score for an output function value of "1".
- **p.pert** Probability of perturbation under independent (IPA) assumption.
- **n.pert** Number of perturbations under dependent (DPA) assumption.
Details

The argument data requires a suitable data set, in the form of a data frame. with the following structure: values of 0 and 1 for bivalent crisp-set variables.

The argument outcome specifies the outcome to be explained, in upper-case notation (e.g. X).

The argument conditions specifies the names of the condition variables. If omitted, all variables in data are used except outcome.

The argument type controls which type of perturbations should be simulated to calculate the retention probability. When type = "corruption", it simulates changes of values in the conditions (values of 0 become 1, and values of 1 become 0). When type = "deletion", it calculates the probability of retaining the same solution if a number of cases are deleted from the original data.

The argument dependent is a logical which choses between two categories of assumptions. If dependent = TRUE (the default) it indicates DPA - Dependent Perturbations Assumption, when perturbations depend on each other and are tied to a fixed number of cases, ex-ante (see Thiem, Spohel and Dusa, 2016). If dependent = FALSE, it indicates IPA - Independent Perturbations Assumption, when perturbations are assumed to occur independently of each other.

The argument n.cut is one of the factors that decide which configurations are coded as logical remainders or not, in conjunction with argument incl.cut. Those configurations that contain fewer than n.cut cases with membership scores above 0.5 are coded as logical remainders (OUT = "?"). If the number of such cases is at least n.cut, configurations with an inclusion score of at least incl.cut are coded positive (OUT = "1"), while configurations with an inclusion score below incl.cut are coded negative (OUT = "0").

The argument p.pert specifies the probability of perturbation under the IPA - independent perturbations assumption (when dependent = FALSE).

The argument n.pert specifies the number of perturbations under the DPA - dependent perturbations assumption (when dependent = TRUE). At least one perturbation is needed to possibly change a csQCA solution, otherwise the solution will remain the same (retention equal to 100%) if zero perturbations occur under this argument.

Author(s)

Adrian Dusa

References


Examples

# the replication data, see Thiem, Spohel and Dusa (2015)

dat <- data.frame(matrix(c(
# calculate the retention probability, for 2.5% probability of data corruption
# under the IPA - independent perturbation assumption
retention(dat, outcome = "W", type = "corruption", dependent = FALSE,  
p.pert = 0.025, incl.cut = 1)

# the probability that a csQCA solution will change
1 - retention(dat, outcome = "W", type = "corruption", dependent = FALSE,  
p.pert = 0.025, incl.cut = 1)

---

**runGUI**

*run the GUI shiny app for the QCA package*

---

**Description**

Runs the graphical user interface app based on the **shiny** package.

**Usage**

```r
runGUI(x)
```

**Arguments**

- `x` Path to the shiny app.

**Details**

This function is nothing but a wrapper to the `runApp()` function in package **shiny**. It locates the `gui` directory in the path where the QCA package has been installed, and runs it.

Since version 2.2, the user interface gained an interactive R console in the webpage. Commands are parsed and evaluated into a dedicated environment, with efforts to capture errors and warnings. Preliminary tests are encouraging, but users should be aware this is not a real console and bugs are likely to appear in this stage of development.

**Author(s)**

Adrian Dusa
**superSubset, findSubsets, findSupersets**

*Functions to find subsets or supersets*

**Description**

Functions to find a list of implicants that satisfy some restrictions (see details), or to find the corresponding row numbers in the implicant matrix, for all subsets, or supersets, of a (prime) implicant or an initial causal configuration.

**Usage**

- `superSubset(data, outcome = "", conditions = "", relation = "nec", incl.cut = 1, cov.cut = 0, ron.cut = 0, pri.cut = 0, use.tilde = FALSE, use.letters = FALSE, depth = NULL, ...)`
- `findSubsets(noflevels, input, stop, ...)`
- `findSupersets(noflevels, input, ...)`

**Arguments**

- **data**: A data frame with crisp (binary and multi-value) or fuzzy causal conditions.
- **outcome**: The name of the outcome.
- **conditions**: A string containing the condition variables’ names, separated by commas.
- **relation**: The set relation to outcome, either "nec", "suf", "necsuf" or "sufnec".
- **incl.cut**: The minimal inclusion score of the set relation.
- **cov.cut**: The minimal coverage score of the set relation.
- **ron.cut**: The minimal score for the RoN - relevance of necessity.
- **pri.cut**: The minimal score for the PRI - proportional reduction in inconsistency.
- **use.tilde**: Logical, use tilde for negation with bivalent variables.
- **use.letters**: Logical, use simple letters instead of original variable names.
- **noflevels**: A vector containing the number of levels for each causal condition plus 1 (all subsets are located in the higher dimension, implicant matrix).
- **input**: A vector of row numbers where the (prime) implicants are located, or a matrix of configurations (only for supersets).
- **stop**: The maximum line number (subset) to stop at, and return.
- **depth**: Integer, an upper number of causal conditions to form expressions with.
- **...**: Other arguments, mainly for backward compatibility.
Details

The function `superSubset()` finds a list of implicants that satisfy some restrictions referring to the inclusion and coverage with respect to the outcome, under given assumptions of necessity and/or sufficiency.

Ragin (2000) posits that under the necessity relation, instances of the outcome constitute a subset of the instances of the cause(s). Conversely, under the sufficiency relation, instances of the outcome constitute a superset of the instances of the cause(s).

When `relation = "nec"` the function finds all implicants which are supersets of the outcome, then eliminates the redundant ones and returns the surviving (minimal) supersets, provided they pass the inclusion and coverage thresholds. If none of the surviving supersets pass these thresholds, the function will find unions of causal conditions, instead of set intersections.

When `relation = "suf"` finds all implicants which are subsets of the outcome, and similarly eliminates the redundant ones and return the surviving (minimal) subsets.

When `relation = "necsuf"`, the relation is interpreted as necessity, and `cov.cut` is automatically set equal to the inclusion cutoff `incl.cut`. The same automatic equality is made for `relation = "sufnec"`, when relation is interpreted as sufficiency.

The argument `outcome` specifies the name of the outcome, and if multi-value the argument can also specify the level to explain, using curly brackets notation.

Outcomes can be negated using a tilde operator `~`. The logical argument `neg.out` is now deprecated, but still backwards compatible. Replaced by the tilde in front of the outcome name, it controls whether outcome is to be explained or its negation. If outcome is from a multivalent variable, it has the effect that the disjunction of all remaining values becomes the new outcome to be explained.

If the argument `conditions` is not specified, all other columns in `data` are used.

Along with the standard measures of inclusion and coverage, the function also returns `PRI` for sufficiency and `RonN` (relevance of necessity, see Schneider & Wagemann, 2012) for the necessity relation.

A subset is a combination (an intersection) of causal conditions, with respect to a larger (super)set, which is another (but more parsimonious) combination of causal conditions.

All subsets of a given set can be found in the so called “implicant matrix”, which is a \( n^k \) space, understood as all possible combinations of values in any combination of bases \( n \), each causal condition having three or more levels (Dusa, 2007, 2010).

For binary causal conditions (values 0 and 1), there are three levels in the implicant matrix:

0 to mark a minimized literal
1 to replace the value of 0 in the original binary condition
2 to replace the value of 1 in the original binary condition

A prime implicant is a superset of an initial combination of causal conditions, and the reverse is also true: the initial combination is a subset of a prime implicant.

Any normal implicant (not prime) is a subset of a prime implicant, and in the same time a superset of some initial causal combinations.

Functions `findSubsets()` and `findSupersets()` find:
- all possible such subsets for a given (prime) implicant, or
- all possible supersets of an implicant or initial causal combination
in the implicant matrix.

The argument depth can be used to impose an upper number of causal conditions to form expressions with. By default, it is set to a maximum, but otherwise the computation time is higher when this number is larger.

**Value**

The result of the `super Subset()` function is an object of class "ss", which is a list with the following components:

- **incl.cov**: A data frame with the parameters of fit.
- **coms**: A data frame with the (m)embersip (s)cores of the resulting (co)mbinations.

For `findSubsets()` and `findSupersets()`, a vector with the row numbers corresponding to all possible subsets, or supersets, of a (prime) implicant.

**Author(s)**

Adrian Dusa

**References**


**See Also**

- `createMatrix`, `getRow`

**Examples**

```r
# Lipset binary crisp sets
data(LC)
```
ssLC <- superSubset(LC, "SURV")

require(venn)
x = list("SURV" = which(LC$SURV == 1),
         "STB" = which(ssLC$coms[, 1] == 1),
         "LIU" = which(ssLC$coms[, 2] == 1))
venn(x, cexil = 0.7)

# Lipset multi-value sets
data(LM)
superSubset(LM, "SURV")

# Cebotari & Vink (2013) fuzzy data
data(CVF)

# all necessary combinations with at least 0.9 inclusion and 0.6 coverage cut-offs
ssCVF <- superSubset(CVF, outcome = "PROTEST", incl.cut = 0.9, cov.cut = 0.6)
ssCVF

# the membership scores for the first minimal combination (GEOCON)
ssCVF$coms$GEOCON

# same restrictions, for the negation of the outcome
superSubset(CVF, outcome = "~PROTEST", incl.cut = 0.9, cov.cut = 0.6)

# to find supersets or supersets, a hypothetical example using
# three binary causal conditions, having two levels each: 0 and 1
noflevels <- c(2, 2, 2)

# second row of the implicant matrix: 0 0 1
# which in the "normal" base is: - - 0
# the prime implicant being: ~C
(sub <- findSubsets(noflevels + 1, input = 2))
# 5 8 11 14 17 20 23 26

getRow(noflevels + 1, sub)

# implicant matrix  normal values
# A B C  |  A B C
# 5 0 1 1  |  5 0 0  bc
# 8 0 2 1  |  8 1 0  Bc
# 11 1 0 1  | 11 0 0  ac
# 14 1 1 1  | 14 0 0  abc
# 17 1 2 1  | 17 0 1  aBc
# 20 2 0 1  | 20 1 0  Ac
# 23 2 1 1  | 23 1 0  Abc
# 26 2 2 1  | 26 1 1  Abc
truthTable

Create a truth table

Description

This function creates a truth table from all types of calibrated data (binary crisp, multi-value crisp and fuzzy). For fuzzy data, Ragin's (2008) procedure is applied to assign cases to the vector space corners (the truth table rows, combinations of causal conditions).
### Usage

```r
thruthTable(data, outcome = "", conditions = "", n.cut = 1, incl.cut = 1, complete = FALSE, show.cases = FALSE, sort.by = ", use.letters = FALSE, inf.test = ", ...")
```

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>A data frame containing calibrated causal conditions and an outcome</td>
</tr>
<tr>
<td>outcome</td>
<td>String, the name of the outcome.</td>
</tr>
<tr>
<td>conditions</td>
<td>A single string containing the conditions’ (columns) names separated by com-</td>
</tr>
<tr>
<td></td>
<td>mas, or a character vector of conditions’ names.</td>
</tr>
<tr>
<td>n.cut</td>
<td>The minimum number of cases under which a truth table row is declared as a</td>
</tr>
<tr>
<td></td>
<td>remainder.</td>
</tr>
<tr>
<td>incl.cut</td>
<td>The inclusion cutoff(s): either a single value for the presence of the out-</td>
</tr>
<tr>
<td></td>
<td>put, or a vector of length 2, the second for the absence of the output.</td>
</tr>
<tr>
<td>complete</td>
<td>Logical, print complete truth table.</td>
</tr>
<tr>
<td>show.cases</td>
<td>Logical, print case names.</td>
</tr>
<tr>
<td>sort.by</td>
<td>Sort the truth table according to various columns.</td>
</tr>
<tr>
<td>use.letters</td>
<td>Logical, use letters instead of causal conditions’ names.</td>
</tr>
<tr>
<td>inf.test</td>
<td>Specifies the statistical inference test to be performed (currently only &quot;binom&quot;) and the critical significance level. It can be either a vector of length 2, or a single string containing both, separated by a comma.</td>
</tr>
<tr>
<td>...</td>
<td>Other arguments (mainly for backward compatibility).</td>
</tr>
</tbody>
</table>

### Details

The data should always be provided as a data frame, with calibrated columns. Calibration can be either crisp, with 2 or more values starting from 0, or fuzzy with continuous scores from 0 to 1. Raw data containing relative frequencies can also be continuous between 0 and 1, but these are not calibrated, fuzzy data.

Some columns can contain the placeholder "~" indicating a “don’t care”, which is used to indicate the temporal order between other columns in tQCA. These special columns are not causal conditions, hence no parameters of fit will be calculated for them.

The argument outcome specifies the column name to be explained. If the outcome is a multivalue column, it can be specified in curly bracket notation, indicating the value to be explained (the others being automatically converted to zero).

The outcome can be negated using a tilde operator ~X. The logical argument neg.out is now deprecated, but still backwards compatible. Replaced by the tilde in front of the outcome name, it controls whether outcome is to be explained or its negation.

If the outcome column is multi-value, the argument outcome should use the standard curly-bracket notation X(value). Multiple values are allowed, separated by a comma (for example X(1,2)). Negation of the outcome can also be performed using the tilde ~ operator, for example ~X[1,2], which is interpreted as: “all values in X except 1 and 2” and it becomes the new outcome to be explained.
Using both neg.out = TRUE and a tilde ~ in the outcome name don’t cancel each other out, either one (or even both) signaling if the outcome should be negated.

The argument conditions specifies the causal conditions’ names among the other columns in the data. When this argument is not specified, all other columns except for the outcome are taken as causal conditions.

A good practice advice is to specify both outcome and conditions as upper case letters. It is possible, in a next version, to negate outcomes using lower case letters, situation in which it really does matter how the outcome and/or conditions are specified.

The argument n.cut specifies the frequency threshold under which a truth table row is coded as a remainder, irrespective of its inclusion score.

The argument incl.cut replaces the (deprecated, but still backwards compatible) former arguments incl.cut1 and incl.cut0. Most of the analyses use the inclusion cutoff for the presence of the output (code "1"). When users need both inclusion cutoffs (see below), incl.cut can be specified as a vector of length 2, in the form: c(ic1, ic0) where:

- ic1 is the inclusion cutoff for the presence of the output, a minimum sufficiency inclusion score above which the output value is coded with "1".
- ic0 is the inclusion cutoff for the absence of the output, a maximum sufficiency inclusion score below which the output value is coded with "0".

If not specifically declared, the argument ic0 is automatically set equal to ic1, but otherwise ic0 should always be lower than ic1.

Using these two cutoffs, the observed combinations are coded with:

- "1" if they have an inclusion score above ic1
- "C" if they have an inclusion score below ic1 and above ic0 (contradiction)
- "0" if they have an inclusion score below ic0

When argument show.cases is set to TRUE, the case names will be printed at their corresponding row in the truth table. The resulting object always contains the cases for each causal combination, even if not printed on the screen (the print function can later be used to print them).

The sort.by argument orders all configurations by any of the columns present in the truth table. Typically, sorting occurs by their outcome value, and/or by their inclusion score, and/or by their frequency, in any order.

Sorting decreasingly (the default) or increasingly can be specified adding the signs - or +, next after the column name in argument sort.by (see examples). Note that - is redundant, because it is the default anyways.

The order specified in this vector is the order in which the configurations will be sorted. When sorting based on the OUTput column, remainders will always be sorted last.

The argument use.letters controls using the original names of the causal conditions, or replace them by single letters in alphabetical order. If the causal conditions are already named with single letters, the original letters will be used.
The argument inf.test combines the inclusion score with a statistical inference test, in order to assign values in the output column OUT. For the moment, it is only the binomial test, which needs crisp data (it doesn’t work with fuzzy sets). Following a similar logic as above, for a given (specified) critical significance level, the output for a truth table row will be coded as:

"1" if the true inclusion score is significantly higher than ic1,
"C" contradiction, if the true inclusion score is not significantly higher than ic1 but significantly higher than ic0,
"0" if the true inclusion score is not significantly higher than ic0.

It should be noted that statistical tests perform well only when the number of cases is large, otherwise they are usually not significant. For a low number of cases, depending on the inclusion cutoff value(s), it will be harder to code a value of "1" in the output, and also harder to obtain contradictions if the true inclusion is not significantly higher than ic0.

The argument complete controls how to print the table on the screen, either complete (when set to TRUE), or just the observed combinations (default). For up to 7 causal conditions, the resulting object will always contain the complete truth table, even if it’s not printed on the screen. This is useful for multiple reasons: researchers like to manually change output values in the truth table (sometimes including in this way a remainder, for example), and it is also useful to plot Venn diagrams, each truth table row having a correspondent intersection in the diagram.

Value

An object of class “tt”, a list containing the following components:

- tt The truth table itself.
- indexes The line numbers for the observed causal configurations.
- nollevels A vector with the number of values for each causal condition.
- initial.data The initial data.
- recoded.data The crisp version of the initial.data, if fuzzy.
- cases The cases for each observed causal configuration.
- options The command options used.
- rowsorder The order of the rows after sorting, if using sort.by.
- minmat A membership scores matrix of cases in the observed truth table combinations.

For reasons related to compatibility with other packages, the options component with temporarily contain both forms of notation for the inclusion cuttofs: in addition to incl.cut it will also contain incl.cut1 and incl.cut0 (but will soon be deprecated).

Author(s)

Adrian Dusa

References

Related Techniques, SAGE.


See Also
eqmcc

Examples

```
# -----  
# Lipset binary crisp data  
data(LC)  
ttLC  <-  truthTable(LC, "SURV")  

# inspect the truth table  
ttLC  

# print the cases too, even if not specifically asked for  
print(ttLC, show.cases = TRUE)  

# the printing function also supports the complete version  
print(ttLC, show.cases = TRUE, complete = TRUE)  

# formally asking the complete version  
truthTable(LC, "SURV", complete = TRUE)  

# sorting by multiple columns, decreasing by default  
truthTable(LC, "SURV", complete = TRUE, sort.by = "incl, n")  

# sort the truth table decreasing for inclusion, and increasing for n  
# note that "-" is redundant, sorting is decreasing by default  
truthTable(LC, "SURV", complete = TRUE, sort.by = "incl-, n+")  

# -----  
```
# Lipset multi-value crisp data (Cronqvist & Berg-Schlosser 2009, p.80)
data(LM)
truthTable(LM, "SURV", sort.by = "incl")

# using a frequency cutoff equal to 2 cases
ttLM <- truthTable(LM, "SURV", n.cut = 2, sort.by = "incl")
ttLM

# the observed combinations coded as remainders
ttLM$excluded

# ----- # Cebotari & Vink fuzzy data
data(CVF)
ttCVF <- truthTable(CVF, "PROTEST", incl.cut = 0.8, sort.by = "incl")

# view the Venn diagram for this truth table
library(venn)
venn(ttCVF)

# each intersection transparent by its inclusion score
venn(ttCVF, transparency = ttCVF$tt$incl)

# the truth table negating the outcome
truthTable(CVF, "~PROTEST", incl.cut = 0.8, sort.by = "incl")

# allow contradictions
truthTable(CVF, "PROTEST", incl.cut = c(0.8, 0.75), sort.by = "incl")

# ----- # Ragin and Strand data with temporal QCA
data(RS)

# truth table containing the "~" placeholder as a "don't care"
truthTable(RS, "REC")

---

**Xplot**

*Display the distribution of points for a single condition*

**Description**

This function creates a plot for a single vector of numerical values, arranging them horizontally (on the X axis) from minimum to maximum.
Usage

Xplot(x, jitter = FALSE, at = NULL, ...)

Arguments

x
A numeric vector.

jitter
Logical, vertically jitter the points.

at
The points at which tick-marks are to be drawn. Non-finite (infinite, NaN or NA) values are omitted. By default (when NULL) tickmark locations are automatically computed, see the help file for ?axis.

...
Other graphical parameters from par

Details

This is a special type of (scatter)plot, with points being arranged only on the horizontal axis (it has no vertical axis). Useful when inspecting if points are grouped into naturally occurring clusters, mainly for crisp calibration purposes.

The argument ... is used to pass various graphical parameters for the various plotting functions used.

The points have a default cex (character expansion) value of 1, and a default pch value of 1 (empty points), which can be modified accordingly (for example with value 21 for filled points). When pch = 21, the color for the margins of the points can be specified via the argument col, while the argument bg will determine the fill color of the points.

The axis labels have a default cex.axis value of 0.8, which affects both the tickmarks labels and the axis labels.

When jittering the points, default values of 0.5 are used for the parameters factor and amount, on the horizontal axis.

Author(s)

Adrian Dusa

See Also

par, text, jitter

Examples

# Lipset's raw data
data(LR)

# plot the DEV (level of developent) causal condition
Xplot(LR$DEV)

# jitter the points vertically
Xplot(LR$DEV, jitter = TRUE)
# make the x axis large enough to accommodate all values
Xplot(LR$DEV, jitter = TRUE, at = pretty(LR$DEV))

# clip plotting between the range of min and max
Xplot(LR$DEV, jitter = TRUE, at = range(LR$DEV))

---

**XYplot**

**Create an XY plot**

### Description

This function creates an XY plot from the first two columns of a dataframe/matrix, or from two separate vectors of numeric values.

### Usage

```
XYplot(x, y, data, relation = "nec", mguides = TRUE,
       jitter = FALSE, clabels = NULL, ...)
```

### Arguments

- **x**
  - Character, the name of the column from the data for the X axis, or the coordinates of points in the plot (either a matrix/dataframe with at least two columns, or a vector of values for the X axis).

- **y**
  - Character, the name of the column from the data for the Y axis, or the Y coordinates of points in the plot, optional if `x` is a matrix/dataframe.

- **data**
  - A calibrated dataset, only if `x` and `y` are names.

- **relation**
  - The set relation to Y, either "nec" (default) or "suf".

- **mguides**
  - Logical, print the middle guides.

- **jitter**
  - Logical, jitter the points.

- **clabels**
  - A vector of case labels with the same length as `x` and `y`.

- **...**
  - Other graphical parameters from `par`

### Details

If `x` is a dataframe or a matrix, the X and Y labels will be taken from the column names of `x`, otherwise they will be inferred from the names of the `x` and `y` objects that are passed to this function.

`x` can also be a string containing either the name of the column for the X axis, or two column names separated by a comma, referring to the X and Y axis respectively. When `x` contains both X and Y column names, the next argument will be considered as the data.

If `data` is provided, and the names of the X and Y columns are valid R statements, quoting them is not even necessary and they can be negated using either a tilde `~` or `1 -`.

The numeric values should be restricted between 0 and 1, otherwise an error is generated.

The XY plot will also produce inclusion and coverage scores for a sufficiency or a necessity relation, along with PRI for a sufficiency relation and RoN (relevance of necessity) for a necessity relation.
The argument ... is used to pass various graphical parameters for the various plotting functions used.

The points have a default cex (character expansion) value of 0.8, and a default pch value of 21 (filled points), which can be modified accordingly (for example with value 1 of empty points). When pch = 21, the color for the margins of the points can be specified via the argument col, while the argument bg will determine the fill color of the points.

The axis labels have a default cex.axis value of 0.8, which affects both the tickmarks labels and the axis labels.

When jittering the points, default values of 0.01 are used for the parameters factor and amount, on both horizontal and vertical axes.

**Value**

A list of x and y values, especially useful when the points are jittered.

**Author(s)**

Adrian Dusa

**References**


**See Also**

par, text, jitter

**Examples**

```r
# Cebotari & Vink (2013)
data(CVF)

# necessity relation between NATPRIDE and PROTEST
XYplot(CVF[, 5:6])

# same using two numeric vectors
XYplot(CVF$NATPRIDE, CVF$PROTEST)

# same using two column names
XYplot("NATPRIDE", "PROTEST", data = CVF)

# or using one string containing both
XYplot("NATPRIDE, PROTEST", data = CVF)

# since they are valid R statements, it works even without quotes
# (this only works in normal R console, not in the GUI version)
```
This data set was used by Cebotari and Vink (2013), and it was taken here from the associated replication file Cebotari and Vink (2015).

Usage

data(CVR)
data(CVF)
Format

A data frame containing 29 cases (ethnic minorities) and the following 6 columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEMOC</td>
<td>Level of democracy</td>
</tr>
<tr>
<td>ETHFRAC</td>
<td>Degree of ethnic fractionalization</td>
</tr>
<tr>
<td>GEOCON</td>
<td>Territorial concentration</td>
</tr>
<tr>
<td>POLDIS</td>
<td>Political discrimination</td>
</tr>
<tr>
<td>NATPRIDE</td>
<td>National pride</td>
</tr>
<tr>
<td>PROTEST</td>
<td>Ethnopolitical protest</td>
</tr>
</tbody>
</table>

Details

There are two different versions of the Cebotari and Vink data: CVR contains the raw data, and CVF contains the data calibrated to fuzzy-sets.

The causal conditions are:

DEMOC (contextual factor), based on a democracy index ranking countries on a scale from strong autocracies (0) to strong democracies (10). The fuzzy scores were calibrated using an exclusion threshold of 2, a crossover of 7 and an inclusion threshold of 9.5.

ETHFRAC (contextual factor), with raw scores ranging from a perfectly homogenous society (0) to a highly fragmented country (1). The fuzzy scores were calibrated using an exclusion threshold of 0, a crossover of 0.495 and an inclusion threshold of 0.8.

GEOCON (group-related factor) with raw data coded as: widely dispersed (0) and primarily urban minorities (1) considered territorially dispersed minorities, and ethnic communities majority in a region (2) and entirely concentrated in one region (3) considered as territorially concentrated minorities. The fuzzy scores were calibrated using an exclusion threshold of 0, a crossover of 1.25 and an inclusion threshold of 3.

POLDIS (group-related factor) captures discrimination practices toward minority groups that vary from no discrimination (0) to exclusive and repressive policies toward a minority group (4). The fuzzy scores were calibrated using an exclusion threshold of 0, a crossover of 0.75 and an inclusion threshold of 3.

NATPRIDE (group-related factor) with raw scores ranging from ‘not at all proud’ (0) to ‘very proud’ (3). The fuzzy scores were calibrated using an exclusion threshold of 0.5, a crossover of 1.5 and an inclusion threshold of 2.5.

The outcome is the column PROTEST, measured on a range from 0 to 5 with higher values indicating more intense protest actions. The fuzzy scores were calibrated using an exclusion threshold of 0.5, a crossover of 1.5 and an inclusion threshold of 3.

References


Legacy datasets

Description

The following datasets are no longer part of this package in the formal documentation, but have been added to ensure backwards compatibility with prior publications.

Usage

data(d.AS)
data(d.Bas)
data(d.biodiversity)
data(d.BWB)
data(d.CS)
data(d.CZH)
data(d.education)
data(d.Emm)
data(d.graduate)
data(d.health)
data(d.HK)
data(d.HMN)
data(d.homeless)
data(d.jobsecurity)
data(d.Kil)
data(d.Kro)
data(d.napoleon)
data(d.partybans)
data(d.represent)
data(d.RS)
data(d.SA)
data(d.socialsecurity)
data(d.SS)
data(d.stakeholder)
data(d.transport)
data(d.urban)
data(Emme)
data(HarKem)
data(Krook)
data(RagStr)
data(Rokkan)
Description

This dataset is taken from Lipset (1959), as used by Rihoux and De Meur (2009), Cronqvist and Berg-Schlosser (2009) and Ragin (2009).

Usage

data(LR)
data(LC)
data(LM)
data(LF)

Format

A data frame containing 18 rows and the following 6 columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEV</td>
<td>Level of development</td>
</tr>
<tr>
<td>URB</td>
<td>Level of urbanization</td>
</tr>
<tr>
<td>LIT</td>
<td>Level of literacy</td>
</tr>
<tr>
<td>IND</td>
<td>Level of industrialization</td>
</tr>
<tr>
<td>STB</td>
<td>Government stability</td>
</tr>
<tr>
<td>SURV</td>
<td>Survival of democracy during the inter-war period</td>
</tr>
</tbody>
</table>

Details

There are four different versions of the Lipset data:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>contains the raw data</td>
</tr>
<tr>
<td>LC</td>
<td>is the same data calibrated to binary crisp sets</td>
</tr>
<tr>
<td>LM</td>
<td>is calibrated to multi-value sets</td>
</tr>
<tr>
<td>LF</td>
<td>is calibrated to fuzzy-sets</td>
</tr>
</tbody>
</table>

The causal conditions are:

DEV referring to the level of development: in the raw data is the GDP per capita (USD), calibrated in the binary crisp version to 0 if below 550 USD and 1 otherwise. For the multi-value crisp version, two thresholds were used: 550 and 850 USD.

URB is the percent of the population in towns with 20,000 or more inhabitants, calibrated in the crisp versions to 0 if below 50% and 1 if above.

LIT is the percent of the literate population, calibrated in the crisp versions to 0 if below 75% and 1 if above.
INd is the percent of the industrial labor force, calibrated in the crisp versions to 0 if below 30% and 1 if above.

STB is a “political-institutional” condition added to the previous four “socioeconomic” ones. The raw data has the number of cabinets which governed in the period under study, calibrated in the crisp versions to 0 if 10 or above and 1 if below 10.

The outcome is the column SURV, calibrated to 0 if negative, and 1 if positive raw data.

References


---

_Ragin and Strand_  
*University recognition of a graduate student union*

**Description**

Original data used by Caren and Panofsky (2005), and reanalysed by Ragin and Strand (2008).

**Usage**

`data(RS)`

**Format**

A data frame containing 17 cases and the following 6 columns:

- **P** Public university
- **E** Support of elite allies
- **A** National union affiliation
- **S** Strike or a strike threat
- **EBA** E happens before A
- **REC** Union recognition
Details

The causal conditions are P, E, A and S. All of them are binary crisp with two values: \( \theta = \text{No} \) and \( \lambda = \text{Yes} \).

The column EBA is not a causal condition, specifying in which case the causal condition E happens before the causal condition A. It has two values (\( \theta = \text{No} \) and \( \lambda = \text{Yes} \)) plus the placeholder "-" to signal a “don’t care”.

The outcome is the union recognition EBA, also binary crisp with two values: \( \theta = \text{No} \) and \( \lambda = \text{Yes} \).

Source


Index

*Topic **datasets**
  _Cebotari and Vink_, 56
  _Legacy datasets_, 58
  _Lipset_, 59
  _Ragin and Strand_, 60

*Topic **functions**
  calibrate, 4
  deMorgan, 8
  eqmcc, 10
  export, 17
  Factorize, 18
  findTh, 20
  fuzzyand, fuzzyor, 21
  Implicant matrix functions:
    allExpressions, createMatrix, getRow, 23
  Interpret SOP expressions:
    translate, compute, findRows, sop, 26
  intersection, 29
  PI chart functions: makeChart, solveChart, 31
  pof, 34
  recode, 37
  retention, 40
  runGUI, 42
  superSubset, findSubsets, findSupersets, 43
  truthTable, 47
  Xplot, 52
  XYplot, 54

*Topic **package**
  QCA-package, 2
  _Cebotari and Vink_, 56
  _Legacy datasets_, 58
  _Lipset_, 59
  _Ragin and Strand_, 60

  allExpressions (Implicant matrix functions: allExpressions,
  createMatrix, getRow), 23
  calibrate, 4
  compute (Interpret SOP expressions: translate, compute, findRows, sop), 26
  createMatrix, 45
  createMatrix (Implicant matrix functions: allExpressions, createMatrix, getRow), 23
  cutree, 21
  CVF (_Cebotari and Vink_), 56
  CVR (_Cebotari and Vink_), 56
  deMorgan, 8
  dist, 21
  eqmcc, 2, 10, 10, 36, 51
  expand.grid, 24
  export, 17
  factorize, 15, 18
  findRows (Interpret SOP expressions: translate, compute, findRows, sop), 26
  findSubsets (superSubset, findSubsets, findSupersets), 43
  findSupersets (superSubset, findSubsets, findSupersets), 43
  findTh, 20
  fuzzyand (fuzzyand, fuzzyor), 21
  fuzzyand, fuzzyor, 21
  fuzzyor (fuzzyand, fuzzyor), 21
  getRow, 45
  getRow (Implicant matrix functions: allExpressions, createMatrix, getRow), 23
  hclust, 21
Implicant matrix functions:
  allExpressions, createMatrix, getRow, 23
Interpret SOP expressions: translate, compute, findRows, sop, 26
intersection, 29
jitter, 53, 55
LC(_lipset), 59
LF(_lipset), 59
LM(_lipset), 59
LR(_lipset), 59
makeChart (PI chart functions: makeChart, solveChart), 31
par, 53, 55
PI chart functions: makeChart, solveChart, 31
pof, 34
QCA (QCA-package), 2
QCA-package, 2
recode, 37
retention, 40
RS(_Ragin and Strand), 60
runGUI, 42
solveChart (PI chart functions: makeChart, solveChart), 31
sop (Interpret SOP expressions: translate, compute, findRows, sop), 26
superSubset, 36
superSubset (superSubset, findSubsets, findSupersets), 43
superSubset, findSubsets, findSupersets, 43
text, 53, 55
translate, 19, 36
translate (Interpret SOP expressions: translate, compute, findRows, sop), 26
truthTable, 13, 47
write.table, 18
Xplot, 52
XYplot, 54