Package 'admisc'

September 1, 2021

Version 0.18

Date 2021-09-01

Title Adrian Dusa's Miscellaneous

Depends R (>= 3.5.0)

Imports methods

Suggests QCA (>= 3.7)

Description Contains functions used across packages 'declared', 'DDIwR', 'mixed', 'QCA' and 'venn'. Interprets and translates, factorizes and negates SOP - Sum of Products expressions, for both binary and multi-value crisp sets, and extracts information (set names, set values) from those expressions. Other functions perform various other checks if possibly numeric (even if all numbers reside in a character vector) and coerce to numeric, or check if the numbers are whole. It also offers, among many others, a highly flexible recoding routine.

License GPL (>= 3)

URL https://github.com/dusadrian/admisc

BugReports https://github.com/dusadrian/admisc/issues

NeedsCompilation yes

Author Adrian Dusa [aut, cre, cph] (<https://orcid.org/0000-0002-3525-9253>)

Maintainer Adrian Dusa <dusa.adrian@unibuc.ro>

Repository CRAN

Date/Publication 2021-09-01 08:20:05 UTC

R topics documented:

`admisc-package` *Adrian Dusa's Miscellaneous*

Description

Contains functions used across packages 'QCA', 'DDIwR', and 'venn'. Interprets and translates DNF - Disjunctive Normal Form and SOP - Sum of Products expressions, for both binary and multi-value crisp sets, and extracts information (set names, set values) from those expressions. Other functions perform various other checks if possibly numeric (even if all numbers reside in a character vector) and coerce to numeric, or check if the numbers are whole. It also offers, among others, a highly flexible recoding function. Some of the functions in this package use related functions from package **QCA**, and users are encouraged to install that package despite not being listed in the Imports field (due to circular dependency issues).

Details

Package:	admisc
Type:	Package
Version:	0.18
Date:	2021-09-01
License:	GPL (>= 2)

Author(s)

Authors:
Adrian Dusa
Department of Sociology

University of Bucharest
<dusa.adrian@unibuc.ro>

Maintainer:
Adrian Dusa

`.rda functions: list.rda, obj.rda`
Load and list objects from an .rda file

Description

Utility functions to read the names and load the objects from an .rda file, into an R list.

Usage

```
list.rda(.filename)

obj.rda(.filename)
```

Arguments

`.filename`	The path to the file where the R object is saved.

Details

Files with the extension .rda are routinely created using the base function `save()`.

The function `list.rda()` loads the object(s) from the .rda file into a list, preserving the object names in the list components.

The .rda file can naturally be loaded with the base `load()` function, but in doing so the containing objects will overwrite any existing objects with the same names.

The function `obj.rda()` returns the names of the objects from the .rda file.

Value

A list, containing the objects from the loaded .rda file.

Author(s)

Adrian Dusa

`Brackets` *Extract information from a multi-value SOP/DNF expression*

Description

Functions to extract information from an expression written in SOP - sum of products form, (or from the canonical DNF - disjunctive normal form) for multi-value causal conditions. It extracts either the values within brackets, or the causal conditions' names outside the brackets.

Usage

```
insideBrackets(x, type = "[", invert = FALSE)
outsideBrackets(x, type = "[")
curlyBrackets(x, outside = FALSE)
squareBrackets(x, outside = FALSE)
roundBrackets(x, outside = FALSE)
```

Arguments

`x`	A DNF/SOP expression.
`type`	Brackets type: curly, round or square
`invert`	Logical, if activated returns whatever is not within the brackets
`outside`	Logical, if activated returns the conditions' names outside the brackets

Details

Expressions written in SOP - sum of products are used in Boolean logic, signaling a disjunction of conjunctions.

These expressions are useful in Qualitative Comparative Analysis, a social science methodology that is employed in the context of searching for causal configurations that are associated with a certain outcome.

They are also used to draw Venn diagrams with the package `venn`, which draws any kind of set intersection (conjunction) based on a custom SOP expression.

The functions `curlyBrackets`, `squareBrackets` and `roundBrackets` are just special cases of the functions `insideBrackets` and `outsideBrackets`, using the argument `type` as either `"{"`, `"["` or `"("`.

The function `outsideBrackets` itself can be considered a special case of the function `insideBrackets`, when it uses the argument `invert = TRUE`.

SOP expressions are usually written using curly brackets for multi-value conditions but to allow the evaluation of unquoted expressions, they first needs to get past R's internal parsing system. For this reason, multi-value conditions in unquoted expresions should use the square brackets notation, and conjunctions should always use the product `*` sign.

Sufficiency is recognized as `"=>"` in quoted expressions but this does not pass over R's parsing system in unquoted expressions. To overcome this problem, it is best to use the single arrow `"->"` notation. Necessity is recognized as either `"<="` or `"<-"`, both being valid in quoted and unquoted expressions.

Author(s)

Adrian Dusa

Examples

```
sop <- "A[1] + B[2]*C[0]"

insideBrackets(sop) # 1, 2, 0

insideBrackets(sop, invert = TRUE) # A, B, C

# unquoted (valid) SOP expressions are allowed, same result
insideBrackets(A[1] + B[2]*C[0]) # the default type is "["

# curly brackets are also valid in quoted expressions
insideBrackets("A{1} + B{2}*C{0}", type = "{")

# or
curlyBrackets("A{1} + B{2}*C{0}")

# and the condition names
curlyBrackets("A{1} + B{2}*C{0}", outside = TRUE)

squareBrackets(A[1] + B[2]*C[0]) # 1, 2, 0

squareBrackets(A[1] + B[2]*C[0], outside = TRUE) # A, B, C
```

---

combnk *Generate all combinations of n numbers, taken k at a time*

Description

A fast function to generate all possible combinations of n numbers, taken k at a time, starting from the first k numbers or starting from a combination that contain a certain number.

Usage

```
combnk(n, k, ogte = 0, zerobased = FALSE)
```

Arguments

n	Vector of any kind, or a numerical scalar.
k	Numeric scalar.
ogte	At least one value greater than or equal to this number.
zerobased	Logical, zero or one based.

Details

When a scalar, argument n should be numeric, otherwise when a vector its length should not be less than k.

When the argument ogte is specified, the combinations will sequentially be incremented from those which contain a certain number, or a certain position from n when specified as a vector.

Value

A matrix with k rows and choose(n,k) columns.

Author(s)

Adrian Dusa

Examples

```
combnk(5, 2)

combnk(5, 2, ogte = 3)

combnk(letters[1:5], 2)
```

---

export *Export a dataframe to a file or a connection*

Description

This function is a wrapper to write.table(), to overcome possible issues with the row names.

Usage

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

Arguments

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

Details

The default convention for write.table() is to add a blank column name for the row names, but (despite it is a standard used for CSV files) that doesn't 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 always set to FALSE, a new column being added anyways (if possible).

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

Author(s)

Adrian Dusa

See Also

The "R Data Import/Export" manual.

`write.table`

`factorize` *Factorize Boolean expressions*

Description

This function finds all combinations of common factors in a Boolean expression written in SOP - sum of products. It makes use of the function `simplify()`, which uses the function `minimize()` from package **QCA**). Users are highly encouraged to install and load that package, despite not being present in the Imports field (due to circular dependency issues).

Usage

```
factorize(input, snames = "", noflevels = NULL, pos = FALSE, ...)
```

Arguments

`input`	A string representing a SOP expression, or a minimization 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.
`pos`	Logical, if possible factorize using product(s) of sums.
`...`	Other arguments (mainly for backwards compatibility).

Details

Factorization is a process of finding common factors in a Boolean expression, written in SOP - sum of products. Whenever possible, the factorization can also be performed in a POS - product of sums form.

Conjunctions should preferably be indicated with a star `*` sign, but this is not necessary when conditions have single letters or when the expression is expressed in multi-value notation.

The argument `snames` is only needed when conjunctions are not indicated by any sign, 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 `input` is an object of class `"qca"` (the result of the function `minimize()` from package **QCA**), a factorization is performed for each of the minimized solutions.

Value

A named list, each component containing all possible factorizations of the input expression(s), found in the name(s).

Author(s)

Adrian Dusa

References

Ragin, C.C. (1987) *The Comparative Method. Moving beyond qualitative and quantitative strategies*, Berkeley: University of California Press

See Also

`translate`

Examples

```
## Not run:
  # make sure the package QCA is loaded
  library(QCA)

## End(Not run)

# typical example with redundant conditions
factorize(a~b~cd + a~bc~d + a~bcd + abc~d)

# 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 = c(one, two, three, four))
```

```
# using pos - products of sums
factorize(~a~c + ~ad + ~b~c + ~bd, pos = TRUE)

## Not run:
# using an object of class "qca" produced with function minimize()
# in package QCA

pCVF <- minimize(CVF, outcome = "PROTEST", incl.cut = 0.8,
                 include = "?", use.letters = TRUE)

factorize(pCVF)

# using an object of class "deMorgan" produced with negate()
factorize(negate(pCVF))

## End(Not run)
```

`finvert` *Inverts the values of a factor*

Description

Useful function to invert the values from a categorical variable, for instance a Likert response scale.

Usage

```
finvert(x, levels = FALSE)
```

Arguments

`x`	A categorical variable (a factor)
`levels`	Logical, invert the levels as well

Value

A factor of the same length as the original one.

Author(s)

Adrian Dusa

Examples

```
words <- c("ini", "mini", "miny", "moe")
variable <- factor(words, levels = words)

# inverts the value, preserving the levels
finvert(variable)
```

```
# inverts both values and levels
finvert(variable, levels = TRUE)
```

`frelevel` *Modified* `relevel()` *function*

Description

The base function `relevel()` accepts a single argument "ref", which can only be a scalar and not a vector of values. `frelevel()` accepts more (even all) levels and reorders them.

Usage

```
frelevel(variable, levels)
```

Arguments

`variable`	The categorical variable of interest
`levels`	One or more levels of the factor, in the desired order

Value

A factor of the same length as the initial one.

Author(s)

Adrian Dusa

See Also

`relevel`

Examples

```
words <- c("ini", "mini", "miny", "moe")
variable <- factor(words, levels = words)

# modify the order of the levels, keeping the order of the values
frelevel(variable, c("moe", "ini", "miny", "mini"))
```

getName *Get the name of the object being used in a function call*

Description

This is a utility to be used inside a function.

Usage

```
getName(x)
```

Arguments

x	A function argument

Details

Within a function, the argument x can be anything and it is usually evaluated as an object.

This function should be used in conjunction with the base `match.call()`, to obtain the original name of the object being served as an input, regardless of how it is being served.

A particular use case of this function relates to the cases when a variable within a data.frame is used. The overall name of the object (the data frame) is irrelevant, as the real object of interest is the variable.

Value

A character vector of length 1.

Author(s)

Adrian Dusa

Examples

```
foo <- function(x) {
    funargs <- unlist(lapply(match.call(), deparse)[-1])
    return(getName(funargs[1]))
}

dd <- data.frame(X = 1:5)

foo(dd)
# dd

foo(dd$X)
# X

foo(dd[["X"]])
```

```
# X

foo(dd[, 1])
# X
```

`Interpret DNF/SOP expressions: compute, simplify, expand, translate`

Functions to interpret and manupulate a SOP/DNF expression

Description

These functions interpret an expression written in sum of products (SOP) or in canonical disjunctive normal form (DNF), for both crisp and multivalue notations. The function `compute()` calculates set membership scores based on a SOP expression applied to a calibrated data set (see function `calibrate()` from package **QCA**), while the function `translate()` translates a SOP expression into a matrix form.

A function similar to `compute()` was initially written by 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 `simplify()` transforms a SOP expression into a simpler equivalent, through a process of Boolean minimization. The package uses the function `minimize()` from package **QCA**), so users are highly encouraged to install and load that package, despite not being present in the Imports field (due to circular dependency issues).

Function `expand()` performs a Quine expansion to the complete DNF, or a partial expansion to a SOP expression with equally complex terms.

Usage

```
compute(expression = "", data = NULL, separate = FALSE)

simplify(expression = "", snames = "", noflevels = NULL, ...)

translate(expression = "", snames = "", noflevels = NULL, data = NULL, ...)

expand(expression = "", snames = "", noflevels = NULL, partial = FALSE,
      implicants = FALSE, ...)
```

Arguments

`expression`	String: a SOP - sum of products expression.
`data`	A dataset with binary cs, mv and fs data.
`separate`	Logical, perform computations on individual, separate paths.
`snames`	A string containing the sets' names, separated by commas.
`noflevels`	Numerical vector containing the number of levels for each set.
`partial`	Logical, perform a partial Quine expansion.
`implicants`	Logical, return an expanded matrix in the implicants space.
`...`	Other arguments, mainly for backwards compatibility.

Details

An expression written in sum of products (SOP), is a "union of intersections", for example `A*B + B*~C`. The disjunctive normal form (DNF) is also a sum of products, with the restriction that each product has to contain all literals. The equivalent DNF expression is: `A*B*~C + A*B*C + ~A*B*~C`

The same expression can be written in multivalue notation: `A[1]*B[1] + B[1]*C[0]`.

Expressions can contain multiple values for the same condition, separated by a comma. If B was a multivalue causal condition, an expression could be: `A[1] + B[1,2]*C[0]`.

Whether crisp or multivalue, expressions are treated as Boolean. In this last 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.

Negating a multivalue condition requires a known number of levels (see examples below). Improvements from 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]*A[1,2]` is equivalent with `A[1]`, while `A[0]*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 `data`. When specified, arguments `snames` and `noflevels` have precedence over `data`.

The product operator `*` should always be used, but it can be omitted when the data is multivalue (where product terms are separated by curly brackets), and/or when the set names are single letters (for example `AD + B~C`), and/or when the set names are provided via the argument `snames`.

When expressions are simplified, their simplest equivalent can result in the empty set, if the conditions cancel each other out.

Value

For the function `compute()`, a vector of set membership values.

For function `simplify()`, a character expression.

For the function `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 when signaling a minimized condition. The mode of this matrix is character, to allow printing multiple levels in the same cell, such as "1,2".

For function `expand()`, a character expression or a matrix of implicants.

For function `generate()`, a data frame.

Author(s)

Adrian Dusa

References

Ragin, C.C. (1987) *The Comparative Method: Moving beyond Qualitative and Quantitative Strategies*. Berkeley: University of California Press.

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

Examples

```
# -----
# for compute()
## Not run:
# make sure the package QCA is loaded
library(QCA)
compute(DEV*~IND + URB*STB, data = LF)

# calculating individual paths
compute(DEV*~IND + URB*STB, data = LF, separate = TRUE)

## End(Not run)


# -----
# for simplify(), also make sure the package QCA is loaded
simplify("(A + B)(A + ~B)") # result is "A"

# works even without the quotes
simplify((A + B)(A + ~B)) # result is "A"

# but to avoid confusion POS expressions are more clear when quoted
# to force a certain order of the set names
simplify("(URB + LIT*~DEV)(~LIT + ~DEV)", snames = c(DEV, URB, LIT))

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


# Ragin's (1987) book presents the equation E = SG + LW as the result
# of the Boolean minimization for the ethnic political mobilization.

# intersecting the reactive ethnicity perspective (R = ~L~W)
# with the equation E (page 144)

simplify("~L~W(SG + LW)", snames = c(S, L, W, G))

# [1] "S~L~WG"


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

# [1] "SWG + SLW"
```

```
# and factorized
factorize(simplify("SW(SG + LW)", snames = c(S, L, W, G)))

# F1: SW(G + L)


# developmental perspective (D = Lg) and E (page 146)
simplify("L~G(SG + LW)", snames = c(S, L, W, G))

# [1] "LW~G"

# subnations that exhibit ethnic political mobilization (E) but were
# not hypothesized by any of the three theories (page 147)
# ~H = ~(~L~W + SW + L~G) = GL~S + GL~W + G~SW + ~L~SW

simplify("(GL~S + GL~W + G~SW + ~L~SW)(SG + LW)", snames = c(S, L, W, G))


# -----
# for translate()
translate(A + B*C)

# same thing in multivalue notation
translate(A[1] + B[1]*C[1])

# tilde as a standard negation (note the condition "b"!)
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], snames = c(A, B, C), 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(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(AaBb + ~CcDd, snames = c(Aa, Bb, Cc, Dd))

# to print _all_ codes from the standard output matrix
(obj <- translate(A + ~B*C))
print(obj, original = TRUE) # also prints the -1 code


# -----
# for expand()
expand(~AB + B~C)
```

```
# S1: ~AB~C + ~ABC + AB~C

expand(~AB + B~C, snames = c(A, B, C, D))

# S1: ~AB~C~D + ~AB~CD + ~ABC~D + ~ABCD + AB~C~D + AB~CD

# In implicants form:
expand(~AB + B~C, snames = c(A, B, C, D), implicants = TRUE)

#      A B C D
# [1,] 1 2 1 1    ~AB~C~D
# [2,] 1 2 1 2    ~AB~CD
# [3,] 1 2 2 1    ~ABC~D
# [4,] 1 2 2 2    ~ABCD
# [5,] 2 2 1 1    AB~C~D
# [6,] 2 2 1 2    AB~CD
```

`intersection` *Intersect expressions*

Description

This function takes two or more SOP expressions (combinations of conjunctions and disjunctions) or even entire minimization objects, and finds their intersection.

Usage

```
intersection(..., snames = "", noflevels)
```

Arguments

`...`	One or more expressions, combined with / or minimization objects of class `"QCA_min"`.
`snames`	A string containing the sets' names, separated by commas.
`noflevels`	Numerical vector containing the number of levels for each set.

Details

The initial aim of this function was to provide a software implementation of the intersection examples presented by Ragin (1987: 144-147). That type of example can also be performed with the function `simplify()`, while this function is now mainly used in conjunction with the `modelFit()` function from package **QCA**, to assess the intersection between theory and a QCA model.

Irrespective of the input type (character expressions and / or minimiation objects), this function is now a wrapper to the main `simplify()` function (which only accepts character expressions).

It can deal with any kind of expressions, but multivalent crisp conditions need additional information about their number of levels, via the argument `noflevels`.

The 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 all 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.

For minimzation objects of class `"QCA_min"`, the number of levels, and the set names are automatically detected.

Author(s)

Adrian Dusa

References

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

Examples

```
# using minimization objects
## Not run:
library(QCA) # if not already loaded
ttLF <- truthTable(LF, outcome = "SURV", incl.cut = 0.8)
pLF <- minimize(ttLF, include = "?")


# for example the intersection between the parsimonious model and
# a theoretical expectation
intersection(pLF, DEV*STB)


# negating the model
intersection(negate(pLF), DEV*STB)

## End(Not run)


# -----
# 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(~L~W, SG + LW, snames = c(S, L, W, G))


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

```
# and factorized
factorize(intersection(SW, SG + LW, snames = c(S, L, W, G)))


# developmental perspective (D = L~G) and E (page 146)
intersection(L~G, SG + LW, snames = c(S, L, W, G))


# subnations that exhibit ethic political mobilization (E) but were
# not hypothesized by any of the three theories (page 147)
# ~H = ~(~L~W + SW + L~G)
intersection(negate(~L~W + SW + L~G), SG + LW, snames = c(S, L, W, G))
```

`Negate DNF/SOP expressions`

Negate Boolean expressions

Description

Functions to negate a DNF/SOP expression, or to invert SOP to POS or POS to SOP.

Usage

```
negate(input, snames = "", noflevels, simplify = TRUE, ...)

invert(input, snames = "", noflevels)
```

Arguments

`input`	A string representing a SOP expression, or a minimization object of class `"QCA_min"`.
`snames`	A string containing the sets' names, separated by commas.
`noflevels`	Numerical vector containing the number of levels for each set.
`simplify`	Logical, allow users to choose between the raw negation or its simplest form.
`...`	Other arguments (mainly for backwards compatibility).

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 `minimize()` in package `QCA`. The resulting object, of class `"qca"`, is automatically recognised by this function.

In a SOP expression, 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.

It is entirely possible to obtain multiple negations of a single expression, since the result of the negation is passed to function `simplify()`.

Function `invert()` simply transforms an expression from a sum of products (SOP) to a product of sums (POS), and the other way round.

Value

A character vector when the input is a SOP expresison, or a named list for minimization input objects, each component containing all possible negations of the model(s).

Author(s)

Adrian Dusa

References

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

See Also

`minimize`, `simplify`

Examples

```
# example from Ragin (1987, p.99)
negate(AC + B~C, simplify = FALSE)

# the simplified, logically equivalent negation
negate(AC + B~C)

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

# invert to POS
invert(a*b + ~c*d)

## Not run:
# using an object of class "qca" produced with minimize()
```

```
# from package QCA
library(QCA)
cLC <- minimize(LC, outcome = SURV)

negate(cLC)


# parsimonious solution
pLC <- minimize(LC, outcome = SURV, include = "?")

negate(pLC)

## End(Not run)
```

`Number equality` *Check difference and / or (in)equality of numbers*

Description

Check if one number is greater/lower than or equal to another.

Usage

```
agtb(a, b)
altb(a, b)
agteb(a, b)
alteb(a, b)
aeqb(a, b)
aneqb(a, b)
```

Arguments

a	Numerical vector
b	Numerical vector

Details

Not all numbers (especially the decimal ones) can be represented exactly in floating point arithmetic, and their arithmetic may not give the normal expected result.

This set of functions check for the in(equality) between two numerical vectors a and b, with the following name convention:

`gt` means "greater than"

`lt` means a "lower than" b

`gte` means a "greater than or equal to" b

`lte` means a "lower than or equal to" b

`eq` means a "equal to" b

`neq` means a "not equal to" b

Author(s)

Adrian Dusa

References

Goldberg, David (1991) "What Every Computer Scientist Should Know About Floating-point Arithmetic", ACM Computing Surveys vol.23, no.1, pp.5-48, doi: 10.1145/103162.103163

Numeric testing and coercion

Numeric vectors

Description

Coerces objects to class "numeric", and checks if an object is numeric.

Usage

```
asNumeric(x)
possibleNumeric(x)
wholeNumeric(x)
```

Arguments

x	A vector of values

Details

Unlike the function `as.numeric()` from the **base** package, the function `asNumeric()` coerces to numeric without a warning if any values are not numeric. All such values are considered NA missing.

The function `possibleNumeric()` tests if the values in a vector are possibly numeric, irrespective of their storing as character or numbers.

Function `wholeNumeric()` tests if numbers in a vector are whole (round) numbers. Whole numbers are different from "integer" numbers (which have special memory representation), and consequently the function `is.integer()` tests something different, how numbers are stored in memory (see the description of function `double()` for more details).

Author(s)

Adrian Dusa

See Also

`numeric`, `integer`, `double`

Examples

```
x <- c("-.1", " 2.7 ", "B")
asNumeric(x) # (-0.1, 2.7, NA)  and no warning

possibleNumeric(x) # FALSE

possibleNumeric(c("1", 2, 3)) # TRUE

is.integer(1) # FALSE

# Signaling an integer in R
is.integer(1L) # TRUE

wholeNumeric(1) # TRUE
```

`permutations` *Calculates the permutations of a vector*

Description

Generates all possible permutations of elements from a vector.

Usage

```
permutations(x)
```

Arguments

x	Any kind of vector.

Author(s)

Adrian Dusa

Examples

```
permutations(1:3)
```

recode *Recode a variable*

Description

Recodes a vector (numeric, character or factor) according to a set of rules. It is similar to the function recode() from package **car**, but more flexible. It also has similarities with the function findInterval() from package **base**.

Usage

```
recode(x, rules, cut, values, ...)
```

Arguments

x	A vector of mode numeric, character or factor.
rules	Character string or a vector of character strings for recoding specifications.
cut	A vector of one or more unique cut points.
values	A vector of output values.
...	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 in 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 and minimize possible namespace collisions with package **car**, special efforts have been invested to ensure perfect compatibility with the other `recode()` function (plus more).

The argument `...` allows for more arguments specific to the **car** package, such as `as.factor.result`, `as.numeric.result`. In addition, it also accepts `levels`, `labels` and `ordered` specific to function `factor()` in package **base**. When using the arguments `levels` and / or `labels`, the output will automatically be coerced to a factor.

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

It is possible to use `recode()` in a similar way to function `cut()`, by specifying a vector of cut values which work for both numeric and character/factor objects. For any number of `c` cut values, there should be `c + 1` values, and if not otherwise specified the argument `values` is automatically constructed as a sequence of numbers from 1 to `c + 1`.

Unlike the function `cut()`, arguments such as `include.lowest` or `right` are not necessary because the final outcome can be changed by tweaking the cut values.

Author(s)

Adrian Dusa

Examples

```
x <- rep(1:3, 3)
#  [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 3 0 0 3 0 0 3


set.seed(1234)
x <- sample(18:90, 20, replace = TRUE)
#  [1] 45 39 26 22 55 33 21 87 31 73 79 21 21 38 57 73 84 22 83 64

recode(x, cut = "35, 55")
#  [1] 2 2 1 1 2 1 1 3 1 3 3 1 1 2 3 3 3 1 3 3

set.seed(1234)
x <- factor(sample(letters[1:10], 20, replace = TRUE),
          levels = letters[1:10])
#  [1] j f e i e f d b g f j f d h d d e h d h
# 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]  2 NA NA  2 NA NA  1  1  2 NA  2 NA  1  2  1  1 NA  2  1  2

recode(x, "a, c:f = A; g:hi = B; else = C", labels = "A, B, C")
#  [1] B A A B A A A C B A B A A B A A A B A B
# Levels: A B C

recode(x, "a, c:f = 1; g:hi = 2; else = 3",
       labels = c("one", "two", "three"), ordered = TRUE)
#  [1] two   one   one   two   one   one   one   three two   one
# [11] two   one   one   two   one   one   one   two   one   two
# Levels: one < two < three

set.seed(1234)
categories <- c("An", "example", "that has", "spaces")
x <- factor(sample(categories, 20, replace = TRUE),
            levels = categories, ordered = TRUE)
sort(x)
#  [1] An       An       An       example  example  example  example
#  [8] example  example  example  example  that has that has that has
# [15] spaces   spaces   spaces   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 2 2 2 2 2 2

# single quotes work, but are not necessary
recode(sort(x), "An : 'that has' = 1; spaces = 2")
#  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2

# same using cut values
recode(sort(x), cut = "that has")
#  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2

# modifying the output values
recode(sort(x), cut = "that has", values = 0:1)
#  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1

# more treatment of "else" values
x <- 10:20

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

# all other values copied
recode(x, "8:15=1; else=copy")
#  [1]  1  1  1  1  1  1 16 17 18 19 20
```

---

recreate *Facilitate expression substitution*

Description

Utility function based on `substitute()`, to recover an unquoted input.

Usage

```
recreate(x, snames = NULL)
```

Arguments

`x`	A substituted input.
`snames`	A character string containing set names.

Details

This function is especially useful when users have to provide lots of quoted inputs, such as the name of the columns from a data frame to be considered for a particular function.

This is actually one of the main uses of the base function `substitute()`, but here it can be employed to also detect SOP (sum of products) expressions, explained for instance in function `translate()`.

Such SOP expressions are usually used in contexts of sufficieny and necessity, which are indicated with the usual signs `->` and `<-`. These are both allowed by the R parser, indicating standard assignment. Due to the R's internal parsing system, a sufficient expression using `->` is automatically flipped to a necessity statement `<-` with reversed LHS to RHS, but this function is able to determine what is the expression and what is the output.

The other necessity code `<=` is also recognized, but the equivalent sufficiency code `=>` is not allowed in unquoted expressions.

Value

A quoted, equivalent expression or a substituted object.

Author(s)

Adrian Dusa

See Also

`substitute`, `simplify`

Examples

```
recreate(substitute(A + ~B*C))

foo <- function(x, ...) recreate(substitute(list(...)))

foo(arg1 = 3, arg2 = A + ~B*C)

df <- data.frame(A = 1, B = 2, C = 3, Y = 4)

# substitute from the global environment
```

```
# the result is the builtin C() function
res <- recreate(substitute(C))

is.function(res) # TRUE

# search first within the column name space from df
recreate(substitute(C), colnames(df))
# "C"

# necessity well recognized
recreate(substitute(A <- B))

# but sufficiency is flipped
recreate(substitute(A -> B))

# more complex SOP expressions are still recovered
recreate(substitute(A + ~B*C -> Y))
```

`replaceText` *Replace text in a string*

Description

Provides an improved method to replace strings, compared to function gsub() in package **base**.

Usage

```
replaceText(expression = "", target = "", replacement = "", boolean = FALSE, ...)
```

Arguments

`expression`	Character string, usually a SOP - sum of products expression.
`target`	Character vector or a string containing the text to be replaced.
`replacement`	Character vector or a string containing the text to replace with.
`boolean`	Treat characters in a boolean way, using upper and lower case letters.
`...`	Other arguments, from and to other functions.

Details

If the input expression is "J*JSR", and the task is to replace "J" with "A" and "JSR" with "B", function gsub() is not very useful since the letter "J" is found in multiple places, including the second target.

This function finds the exact location(s) of each target in the input string, starting with those having the largest number of characters, making sure the locations are unique. For instance, the target "JSR" is found on the location from 3 to 5, while the target "J" is is found on two locations 1 and 3, but 3 was already identified in the previously found location for the larger target.

In addition, this function can also deal with target strings containing spaces.

Value

The original string, replacing the target text with its replacement.

Author(s)

Adrian Dusa

Examples

```
replaceText("J*JSR", "J, JSR", "A, B")

# same output, on input expresions containing spaces
replaceText("J*JS R", "J, JS R", "A, B")

# works even with Boolean expressions, where lower case
# letters signal the absence of the causal condition
replaceText("DEV + urb*LIT", "DEV, URB, LIT", "A, B, C", boolean = TRUE)
```

`Tildae operations` *Tildae operations*

Description

Checks and changes expressions containing set negations using a tilde.

Usage

```
hastilde(x)
notilde(x)
tilde1st(x)
```

Arguments

x	A vector of values

Details

Boolean expressions can be negated in various ways. For binary crisp and fuzzy sets, one of the most straightforward ways to invert the set membership scores is to subtract them from 1. This is both possible using R vectors and also often used to signal a negation in SOP (sum of products) expressions.

Some other times, SOP expressions can signal a set negation (also known as the absence of a causal condition) by using lower case letters, while upper case letters are used to signal the presence of a causal condition. SOP expressions also use a tilde to signal a set negation, immediately preceding the set name.

This set of functions detect when and if a set present in a SOP expression contains a tilde (function `hastilde`), whether the entire expression begins with a tilde (function `tilde1st`).

Author(s)

Adrian Dusa

Examples

```
hastilde("~A")
```

`tryCatchWEM` *Try functions to capture warnings, errors and messages.*

Description

This function combines the base functions `tryCatch()` and `withCallingHandlers()` for the specific purpose of capturing not only errors and warnings but messages as well.

Usage

```
tryCatchWEM(expr, capture = FALSE)
```

Arguments

`expr`	Expression to be evaluated.
`capture`	Logical, capture the visible output.

Details

In some situations it might be important not only to test a function, but also to capture everything that is written in the R console, be it an error, a warning or simply a message.

For instance package **QCA** (version 3.4) has a Graphical User Interface that simulates an R console embedded into a web based **shiny** app.

It is not intended to replace function `tryCatch()` in any way, especially not evaluating an expression before returning or exiting, it simply captures everything that is printed on the console (the visible output).

Value

A list, if anything would be printed on the screen, or an empty (NULL) object otherwise.

Author(s)

Adrian Dusa

Index