Package 'nuggets'

February 3, 2026

Title Extensible Framework for Data Pattern Exploration

Version 2.1.1

Date 2026-02-03

Maintainer Michal Burda <michal.burda@osu.cz>

Description A framework for systematic exploration of association rules (Agrawal et al., 1994, <https://www.vldb.org/conf/1994/P487.PDF>), contrast patterns (Chen, 2022, <doi:10.48550/arXiv.2209.13556>), emerging patterns (Dong et al., 1999, <doi:10.1145/312129.312191>), subgroup discovery (Atzmueller, 2015, <doi:10.1002/widm.1144>), and conditional correlations (Hájek, 1978, <doi:10.1007/978-3-642-66943-9>). User-defined functions may also be supplied to guide custom pattern searches. Supports both crisp (Boolean) and fuzzy data. Generates candidate conditions expressed as elementary conjunctions, evaluates them on a dataset, and inspects the induced sub-data for statistical, logical, or structural properties such as associations, correlations, or contrasts. Includes methods for visualization of logical structures and supports interactive exploration through integrated Shiny applications.

URL https://beerda.github.io/nuggets/, https://github.com/beerda/nuggets/

BugReports https://github.com/beerda/nuggets/issues/

License GPL (>= 3)

Encoding UTF-8

RoxygenNote 7.3.3

Language en-US

Depends R (>= 4.1.0)

Imports classInt, cli, dplyr, fastmatch, generics, ggplot2, grid, lifecycle, methods, purrr, Rcpp, rlang, stats, stringr, tibble, tidyr, tidyselect, utils

LinkingTo cli, Rcpp, testthat

SystemRequirements C++17

Suggests arules, DT, htmltools, htmlwidgets, jsonlite, shiny, shinyjs, shinyWidgets, testthat (>= 3.0.0), xml2, withr, knitr, rmarkdown

Config/testthat/edition 3

VignetteBuilder knitr

NeedsCompilation yes

Author Michal Burda [aut, cre] (ORCID: <https://orcid.org/0000-0002-4182-4407>)

Repository CRAN

Date/Publication 2026-02-03 10:00:02 UTC

Contents

`add_interest.associations` *Add additional interest measures for association rules*

Description

[Experimental]

This function calculates various additional interest measures for association rules based on their contingency table counts.

Usage

```
## S3 method for class 'associations'
add_interest(x, measures = NULL, smooth_counts = 0, p = 0.5, ...)

add_interest(x, ...)
```

Arguments

`x`	A nugget of flavour `associations`, typically created with `dig_associations()`.
`measures`	A character vector specifying which interest measures to calculate. If `NULL` (the default), all supported measures are calculated. See the Details section for the list of supported measures.
`smooth_counts`	A non-negative numeric value specifying the amount of Laplace smoothing to apply to the contingency table counts before calculating the interest measures. Default is `0` (no smoothing). Positive values add the specified amount to each of the counts (`pp`, `pn`, `np`, `nn`), which can help avoid issues with undefined measures due to zero counts. Use `smooth_counts = 1` for standard Laplace smoothing. Use `smooth_counts = 0.5` for Haldane-Anscombe smoothing, which is often used for odds ratio estimation and in chi-squared tests.
`p`	A numeric value in the range `[0, 1]` representing the conditional probability of the consequent being true given that the antecedent is true. This parameter is used in the calculation of GUHA quantifiers `"lci"`, `"uci"`, `"dlci"`, `"duci"`, `"lce"`, and `"uce"`. The default value is `0.5`.
`...`	Currently unused.

Details

The input nugget object must contain the columns `pp` (positive antecedent & positive consequent), `pn` (positive antecedent & negative consequent), `np` (negative antecedent & positive consequent), and `nn` (negative antecedent & negative consequent), representing the counts from the contingency table. These columns are automatically produced by `dig_associations()`.

The supported interest measures that can be calculated include:

- Founded GUHA (General Unary Hypothesis Automaton) quantifiers:

  - `"fi"` - *Founded Implication*, which equals to the `"confidence"` measure calculated automatically by `dig_associations()`.
  - `"dfi"` - *Double Founded Implication* computed as $pp/(pp + pn + np)$
  - `"fe"` - *Founded Equivalence* computed as $(pp + nn)/(pp + pn + np + nn)$

- GUHA quantifiers based on binomial tests - these measures require the additional parameter p, which represents the conditional probability of the consequent being true given that the antecedent is true under the null hypothesis. The measures are computed as one-sided p-values from the Clopper-Pearson confidence interval for the binomial proportion:
  - `"lci"` - *Lower Critical Implication* computed as $\sum_{i=pp}^{pp+pn} \frac{(pp+pn)!}{i!(pp+pn-i)!} p^i (1-p)^{pp+pn-i}$
  - `"uci"` - *Upper Critical Implication* computed as $\sum_{i=0}^{pp} \frac{(pp+pn)!}{i!(pp+pn-i)!} p^i (1-p)^{pp+pn-i}$
  - `"dlci"` - *Double Lower Critical Implication* computed as $\sum_{i=pp}^{pp+pn+np} \frac{(pp+pn+np)!}{i!(pp+pn+np-i)!} p^i (1-p)^{pp+pn+np-i}$
  - `"duci"` - *Double Upper Critical Implication* computed as $\sum_{i=0}^{pp} \frac{(pp+pn+np)!}{i!(pp+pn+np-i)!} p^i (1-p)^{pp+pn+np-i}$
  - `"lce"` - *Lower Critical Equivalence* computed as $\sum_{i=pp}^{pp+pn+np+nn} \frac{(pp+pn+np+nn)!}{i!(pp+pn+np+nn-i)!} p^i (1-p)^{pp+pn+np+nn-i}$
  - `"uce"` - *Upper Critical Equivalence* computed as $\sum_{i=0}^{pp} \frac{(pp+pn+np+nn)!}{i!(pp+pn+np+nn-i)!} p^i (1-p)^{pp+pn+np+nn-i}$

- measures adopted from the `arules` package:
  - `"added_value"` - *Added Value*, see https://mhahsler.github.io/arules/docs/measures#addedvalue for details
  - `"casual_confidence"` - *Casual Confidence*, see https://mhahsler.github.io/arules/docs/measures#casualconfidence for details
  - `"casual_support"` - *Casual Support*, see https://mhahsler.github.io/arules/docs/measures#casualsupport for details
  - `"centered_confidence"` - *Centered Confidence*, see https://mhahsler.github.io/arules/docs/measures#centeredconfidence for details
  - `"certainty"` - *Certainty Factor*, see https://mhahsler.github.io/arules/docs/measures#certainty for details
  - `"collective_strength"` - *Collective Strength*, see https://mhahsler.github.io/arules/docs/measures#collectivestrength for details
  - `"confirmed_confidence"` - *Descriptive Confirmed Confidence*, see https://mhahsler.github.io/arules/docs/measures#confirmedconfidence for details
  - `"conviction"` - *Conviction*, see https://mhahsler.github.io/arules/docs/measures#conviction for details
  - `"cosine"` - *Cosine*, see https://mhahsler.github.io/arules/docs/measures#cosine for details
  - `"counterexample"` - *Example and Counter-Example Rate*, see https://mhahsler.github.io/arules/docs/measures#counterexample for details
  - `"doc"` - *Difference of Confidence*, see https://mhahsler.github.io/arules/docs/measures#doc for details
  - `"gini"` - *Gini Index*, see https://mhahsler.github.io/arules/docs/measures#gini for details

- "imbalance" - *Imbalance Ratio*, see https://mhahsler.github.io/arules/docs/measures#imbalance for details
- "implication_index" - *Implication Index*, see https://mhahsler.github.io/arules/docs/measures#implicationindex for details
- "importance" - *Importance*, see https://mhahsler.github.io/arules/docs/measures#importance for details
- "j_measure" - *J-Measure*, see https://mhahsler.github.io/arules/docs/measures#jmeasure for details
- "jaccard" - *Jaccard Coefficient*, see https://mhahsler.github.io/arules/docs/measures#jaccard for details
- "kappa" - *Kappa*, see https://mhahsler.github.io/arules/docs/measures#kappa for details
- "kulczynski" - *Kulczynski*, see https://mhahsler.github.io/arules/docs/measures#kulczynski for details
- "lambda" - *Lambda*, see https://mhahsler.github.io/arules/docs/measures#lambda for details
- "least_contradiction" - *Least Contradiction*, see https://mhahsler.github.io/arules/docs/measures#leastcontradiction for details
- "lerman" - *Lerman Similarity*, see https://mhahsler.github.io/arules/docs/measures#lerman for details
- "leverage" - *Leverage*, see https://mhahsler.github.io/arules/docs/measures#leverage for details
- "maxconfidence" - *Max Confidence*, see https://mhahsler.github.io/arules/docs/measures#maxconfidence for details
- "mutual_information" - *Mutual Information*, see https://mhahsler.github.io/arules/docs/measures#mutualinformation for details
- "odds_ratio" - *Odds Ratio*, see https://mhahsler.github.io/arules/docs/measures#oddsratio for details
- "phi" - *Phi Correlation Coefficient*, see https://mhahsler.github.io/arules/docs/measures#phi for details
- "ralambondrainy" - *Ralambondrainy*, see https://mhahsler.github.io/arules/docs/measures#ralambondrainy for details
- "relative_risk" - *Relative Risk*, see https://mhahsler.github.io/arules/docs/measures#relativerisk for details
- "rule_power_factor" - *Rule Power Factor*, see https://mhahsler.github.io/arules/docs/measures#rulepowerfactor for details
- "sebag" - *Sebag-Schoenauer*, see https://mhahsler.github.io/arules/docs/measures#sebag for details
- "varying_liaison" - *Varying Rates Liaison*, see https://mhahsler.github.io/arules/docs/measures#varyingliaison for details
- "yule_q" - *Yule's Q*, see https://mhahsler.github.io/arules/docs/measures#yuleq for details
- "yule_y" - *Yule's Y*, see https://mhahsler.github.io/arules/docs/measures#yuley for details

All the above measures are primarily intended for use with binary (logical) data. While they can be computed for numerical data as well, their interpretations may not be meaningful in that context - users should exercise caution when applying these measures to non-binary data.

Many measures are based on the contingency table counts, and some may be undefined for certain combinations of counts (e.g., division by zero). This issue can be mitigated by applying smoothing using the `smooth_counts` argument.

Value

An S3 object which is an instance of `associations` and `nugget` classes and which is a tibble containing all the columns of the input nugget `x`, plus additional columns for each of the requested interest measures.

Author(s)

Michal Burda

See Also

`dig_associations()`

Examples

```
d <- partition(mtcars, .breaks = 2)
rules <- dig_associations(d,
                          antecedent = !starts_with("mpg"),
                          consequent = starts_with("mpg"),
                          min_support = 0.3,
                          min_confidence = 0.8)
rules <- add_interest(rules,
                      measures = c("conviction", "leverage", "jaccard"))
```

---

association_matrix *Create an association matrix from a nugget of flavour* associations.

Description

The association matrix is a matrix where rows correspond to antecedents, columns correspond to consequents, and the values are taken from a specified column of the nugget. Missing values are filled with zeros.

A pair of antecedent and consequent must be unique in the nugget. If there are multiple rows with the same pair, an error is raised.

Usage

```
association_matrix(
  x,
  value,
  error_context = list(arg_x = "x", arg_value = "value", call = current_env())
)
```

Arguments

x	A nugget of flavour associations.
value	A tidyselect expression (see tidyselect syntax) specifying the column to use for filling the matrix values.
error_context	A list of details to be used in error messages. It must contain: - arg_x: the name of the x argument; - arg_value: the name of the value argument; - call: an environment in which to evaluate the error messages. Defaults to the current environment.

Value

A numeric matrix with row names corresponding to antecedents and column names corresponding to consequents. Values are taken from the column specified by value. Missing values are filled with zeros.

Author(s)

Michal Burda

Examples

```
d <- partition(mtcars, .breaks = 2)
rules <- dig_associations(d,
                          antecedent = everything(),
                          consequent = everything(),
                          min_support = 0.3)
association_matrix(rules, confidence)
```

---

bound_range *Bound a range of numeric values*

Description

This function computes the range of numeric values in a vector and adjusts the bounds to "nice" rounded numbers. Specifically, it rounds the lower bound downwards (similar to floor()) and the upper bound upwards (similar to ceiling()) to the specified number of digits. This can be useful when preparing data ranges for axis labels, plotting, or reporting. The function returns a numeric vector of length two, containing the adjusted lower and upper bounds.

Usage

```
bound_range(x, digits = 0, na_rm = FALSE)
```

Arguments

x	A numeric vector to be bounded.
digits	An integer scalar specifying the number of digits to round the bounds to. A positive value determines the number of decimal places used. A negative value rounds to the nearest 10, 100, etc. If digits is NULL, no rounding is performed and the exact range is returned.
na_rm	A logical flag indicating whether NA values should be removed before computing the range. If TRUE, the range is computed from non-NA values only. If FALSE and x contains any NA values, the function returns c(NA, NA).

Value

A numeric vector of length two with the rounded lower and upper bounds of the range of x. The lower bound is always rounded down, and the upper bound is always rounded up. If x is NULL or has length zero, the function returns NULL.

Author(s)

Michal Burda

See Also

floor(), ceiling()

Examples

```
bound_range(c(1.9, 2, 3.1), digits = 0)      # returns c(1, 4)
bound_range(c(190, 200, 301), digits = -2)   # returns c(100, 400)
```

---

cluster_associations *Cluster association rules*

Description

This function clusters association rules based on the selected numeric attribute by (e.g., confidence or lift) and summarizes the clusters. The clustering is performed using the k-means algorithm.

Each cluster is represented by a label consisting of the number of rules in the cluster and the most common predicates in the antecedents of those rules.

Usage

```
cluster_associations(
  x,
  n,
  by,
  algorithm = "Hartigan-Wong",
  predicates_in_label = 2
)
```

Arguments

x	A nugget of flavour associations, typically the output of dig_associations().
n	The number of clusters to create. Must be a positive integer.
by	A tidyselect expression (see tidyselect syntax) specifying the numeric column to use for clustering.
algorithm	The k-means algorithm to use. One of "Hartigan-Wong" (the default), "Lloyd", "Forgy", or "MacQueen". See stats::kmeans() for details.
predicates_in_label	The number of most common predicates to include in the cluster label. The default is 2.

Value

A tibble with one row per (cluster, consequent) pair. The columns are:

- cluster: the cluster number;
- cluster_label: a label for the cluster, consisting of the number of rules in the cluster and the most common predicates in the antecedents of those rules;
- consequent: consequents of the rules;
- other numeric columns from the input nugget, aggregated by mean within each cluster.

Author(s)

Michal Burda

See Also

dig_associations(), association_matrix() stats::kmeans()

Examples

```
# Prepare the data
cars <- mtcars |>
    partition(cyl, vs:gear, .method = "dummy") |>
    partition(carb, .method = "crisp", .breaks = c(0, 3, 10)) |>
    partition(mpg, disp:qsec, .method = "triangle", .breaks = 3)

# Search for associations
rules <- dig_associations(cars,
                          antecedent = everything(),
                          consequent = everything(),
                          max_length = 3,
                          min_support = 0.2)

# Cluster the found rules
clu <- cluster_associations(rules, 10, "lift")

# Print the number of clusters
length(unique(clu$cluster))
```

```
## Not run:
# Plot the clustered rules
library(ggplot2)

ggplot(clu) +
   aes(x = cluster_label, y = consequent, color = lift, size = support) +
   geom_point() +
   xlab("predicates in antecedent groups") +
   scale_y_discrete(limits = rev) +
   theme(axis.text.x = element_text(angle = 45, hjust = 1))

## End(Not run)
```

`dig` *Search for patterns of a custom type*

Description

A general function for searching for patterns of a custom type. The function allows selection of columns of `x` to be used as condition predicates. It enumerates all possible conditions in the form of elementary conjunctions of selected predicates, and for each condition executes a user-defined callback function `f`. The callback is expected to perform some analysis and return an object (often a list) representing a pattern or patterns related to the condition. The results of all calls are returned as a list.

The callback function `f` may accept a number of arguments (see `f` argument description). The algorithm automatically provides condition-related information to `f` based on which arguments are present.

In addition to conditions, the function can evaluate *focus* predicates (foci). Foci are specified separately and are tested within each generated condition. Extra information about them is then passed to `f`.

Restrictions may be imposed on generated conditions, such as:

- minimum and maximum condition length (`min_length`, `max_length`);
- minimum condition support (`min_support`);
- minimum focus support (`min_focus_support`), i.e. support of rows where both the condition and the focus hold.

Usage

```
dig(
  x,
  f,
  condition = everything(),
  focus = NULL,
  disjoint = var_names(colnames(x)),
  excluded = NULL,
```

```
  min_length = 0,
  max_length = Inf,
  min_support = 0,
  min_focus_support = 0,
  min_conditional_focus_support = 0,
  max_support = 1,
  filter_empty_foci = FALSE,
  t_norm = "goguen",
  max_results = Inf,
  verbose = FALSE,
  threads = 1L,
 error_context = list(arg_x = "x", arg_f = "f", arg_condition = "condition", arg_focus =
  "focus", arg_disjoint = "disjoint", arg_excluded = "excluded", arg_min_length =
   "min_length", arg_max_length = "max_length", arg_min_support = "min_support",
  arg_min_focus_support = "min_focus_support", arg_min_conditional_focus_support =
   "min_conditional_focus_support", arg_max_support = "max_support",
  arg_filter_empty_foci = "filter_empty_foci", arg_t_norm = "t_norm", arg_max_results =
   "max_results", arg_verbose = "verbose",
    arg_threads = "threads", call =
   current_env())
)
```

Arguments

x	A matrix or data frame. If a matrix, it must be numeric (double) or logical. If a data frame, all columns must be numeric (double) or logical.
f	A callback function executed for each generated condition. It may declare any subset of the arguments listed below. The algorithm detects which arguments are present and provides only those values to `f`. This design allows the user to control both the amount of information received and the computational cost, as some arguments are more expensive to compute than others. The function `f` is expected to return an object (typically a list) representing a pattern or patterns related to the condition. The results of all calls of `f` are collected and returned as a list. Possible arguments are: `condition`, `sum`, `support`, `indices`, `weights`, pp, pn, np, nn, or `foci_supports` (deprecated), which are thoroughly described below in the "Details" section.
condition	tidyselect expression (see tidyselect syntax) specifying columns of `x` to use as condition predicates
focus	tidyselect expression (see tidyselect syntax) specifying columns of `x` to use as focus predicates
disjoint	An atomic vector (length = number of columns in `x`) defining groups of predicates. Columns in the same group cannot appear together in a condition. With data from `partition()`, use `var_names()` on column names to construct `disjoint`.
excluded	`NULL` or a list of character vectors, each representing an implication formula. In each vector, all but the last element form the antecedent and the last element is the consequent. These formulae are treated as *tautologies* and used to filter out generated conditions. If a condition contains both the antecedent and the

consequent of any such formula, it is not passed to the callback function `f`. Likewise, if a condition contains the antecedent, the corresponding focus (the consequent) is not passed to `f`.

`min_length` Minimum number of predicates in a condition required to trigger the callback `f`. Must be ≥ 0. If set to 0, the empty condition also triggers the callback.

`max_length` Maximum number of predicates allowed in a condition. Conditions longer than `max_length` are not generated. If `Inf`, the only limit is the total number of available predicates. Must be ≥ 0 and $\geq min_length$. This setting strongly influences both the number of generated conditions and the speed of the search.

`min_support` Minimum support of a condition required to trigger `f`. Support is the relative frequency of the condition in `x`. For logical data, this is the proportion of rows where all condition predicates are `TRUE`. For numeric (double) data, support is the mean (over all rows) of the products of predicate values. Must be in $[0, 1]$. If a condition's support falls below `min_support`, recursive generation of its extensions is stopped. Thus, `min_support` directly affects search speed and the number of callback calls.

`min_focus_support`
Minimum support of a focus required for it to be passed to `f`. For logical data, this is the proportion of rows where both the condition and the focus are `TRUE`. For numeric (double) data, support is computed as the mean (over all rows) of a t-norm of predicate values (the t-norm is selected by `t_norm`). Must be in $[0, 1]$. Foci with support below this threshold are excluded. Together with `filter_empty_foci`, this parameter influences both search speed and the number of triggered calls of `f`.

`min_conditional_focus_support`
Minimum conditional support of a focus within a condition. Defined as the relative frequency of rows where the focus is `TRUE` among those where the condition is `TRUE`. If sum (see `support` in *Details*) is the number of rows (or sum of truth degrees for fuzzy data) satisfying the condition, and pp (see `pp[i]` in *Details*) is the sum of truth degrees where both the condition and the focus hold, then conditional support is pp/sum. Must be in $[0, 1]$. Foci below this threshold are not passed to `f`. Together with `filter_empty_foci`, this parameter influences search speed and the number of callback calls.

`max_support` Maximum support of a condition to trigger `f`. Conditions with support above this threshold are skipped, but recursive generation of their supersets continues. Must be in $[0, 1]$.

`filter_empty_foci`
Logical; controls whether `f` is triggered for conditions with no remaining foci after filtering by `min_focus_support` or `min_conditional_focus_support`. If `TRUE`, `f` is called only when at least one focus remains. If `FALSE`, `f` is called regardless.

`t_norm` T-norm used for conjunction of weights: `"goedel"` (minimum), `"goguen"` (product), or `"lukas"` (Łukasiewicz).

`max_results` Maximum number of results (objects returned by the callback `f`) to store and return in the output list. When this limit is reached, generation of further conditions stops. Use a positive integer to enable early stopping; set to `Inf` to remove the cap.

verbose — Logical; if TRUE, print progress messages.

threads — Number of threads for parallel computation.

error_context — A list of details to be used when constructing error messages. This is mainly useful when dig() is called from another function and errors should refer to the caller's argument names rather than those of dig(). The list must contain:

- arg_x – name of the argument x as a character string
- arg_f – name of the argument f as a character string
- arg_condition – name of the argument condition
- arg_focus – name of the argument focus
- arg_disjoint – name of the argument disjoint
- arg_excluded – name of the argument excluded
- arg_min_length – name of the argument min_length
- arg_max_length – name of the argument max_length
- arg_min_support – name of the argument min_support
- arg_min_focus_support – name of the argument min_focus_support
- arg_min_conditional_focus_support – name of the argument min_conditional_focus_suppor
- arg_max_support – name of the argument max_support
- arg_filter_empty_foci – name of the argument filter_empty_foci
- arg_t_norm – name of the argument t_norm
- arg_threads – name of the argument threads
- call – environment in which to evaluate error messages

Details

Let P be the set of condition predicates selected by condition and E be the set of focus predicates selected by focus. The function generates all possible conditions as elementary conjunctions of distinct predicates from P. These conditions are filtered using disjoint, excluded, min_length, max_length, min_support, and max_support.

For each remaining condition, all foci from E are tested and filtered using min_focus_support and min_conditional_focus_support. If at least one focus remains (or if filter_empty_foci = FALSE), the callback f is executed with details of the condition and foci. Results of all calls are collected and returned as a list.

Let C be a condition ($C \subseteq P$), F the set of filtered foci ($F \subseteq E$), R the set of rows of x, and $\mu_C(r)$ the truth degree of condition C on row r. The parameters passed to f are defined as:

- condition: a named integer vector of column indices representing the predicates of C. Names correspond to column names.
- sum: a numeric scalar value of the number of rows satisfying C for logical data, or the sum of truth degrees for fuzzy data, $sum = \sum_{r \in R} \mu_C(r)$.
- support: a numeric scalar value of relative frequency of rows satisfying C, $supp = sum/|R|$.
- pp, pn, np, nn: a numeric vector of entries of a contingency table for C and F, satisfying the Ruspini condition $pp + pn + np + nn = |R|$. The i-th elements of these vectors correspond to the i-th focus F_i from F and are defined as:
  - pp[i]: rows satisfying both C and F_i, $pp_i = \sum_{r \in R} \mu_{C \wedge F_i}(r)$.

- pn[i]: rows satisfying C but not F_i, $pn_i = \sum_{r \in R} \mu_C(r) - pp_i$.
- np[i]: rows satisfying F_i but not C, $np_i = \sum_{r \in R} \mu_{F_i}(r) - pp_i$.
- nn[i]: rows satisfying neither C nor F_i, $nn_i = |R| - (pp_i + pn_i + np_i)$.

Value

A list of results returned by the callback function f.

Author(s)

Michal Burda

See Also

partition(), var_names(), dig_grid()

Examples

```
library(tibble)

# Prepare iris data
d <- partition(iris, .breaks = 2)

# Simple callback: return formatted condition names
dig(x = d,
    f = function(condition) format_condition(names(condition)),
    min_support = 0.5)

# Callback returning condition and support
res <- dig(x = d,
           f = function(condition, support) {
               list(condition = format_condition(names(condition)),
                    support = support)
           },
           min_support = 0.5)
do.call(rbind, lapply(res, as_tibble))

# Within each condition, evaluate also supports of columns starting with
# "Species"
res <- dig(x = d,
           f = function(condition, support, pp) {
               c(list(condition = format_condition(names(condition))),
                 list(condition_support = support),
                 as.list(pp / nrow(d)))
           },
           condition = !starts_with("Species"),
           focus = starts_with("Species"),
           min_support = 0.5,
           min_focus_support = 0)
do.call(rbind, lapply(res, as_tibble))

# Multiple patterns per condition based on foci
```

```
res <- dig(x = d,
           f = function(condition, support, pp) {
               lapply(seq_along(pp), function(i) {
                   list(condition = format_condition(names(condition)),
                        condition_support = support,
                        focus = names(pp)[i],
                        focus_support = pp[[i]] / nrow(d))
               })
           },
           condition = !starts_with("Species"),
           focus = starts_with("Species"),
           min_support = 0.5,
           min_focus_support = 0)

# Flatten result and convert to tibble
res <- unlist(res, recursive = FALSE)
do.call(rbind, lapply(res, as_tibble))
```

`dig_associations` *Search for association rules*

Description

[Experimental]

Association rules identify conditions (*antecedents*) under which a specific feature (*consequent*) is present very often.

Scheme: `A => C`

If condition `A` is satisfied, then the feature `C` is present very often.

Example: `university_edu & middle_age & IT_industry => high_income`

People in *middle age* with *university education* working in IT industry have very likely a *high income*.

Antecedent `A` is usually a set of predicates, and consequent `C` is a single predicate.

For the following explanations we need a mathematical function $supp(I)$, which is defined for a set I of predicates as a relative frequency of rows satisfying all predicates from I. For logical data, $supp(I)$ equals to the relative frequency of rows, for which all predicates $i_1, i_2, \ldots, i_n$ from I are TRUE. For numerical (double) input, $supp(I)$ is computed as the mean (over all rows) of truth degrees of the formula `i_1 AND i_2 AND ... AND i_n`, where `AND` is a triangular norm selected by the `t_norm` argument.

Association rules are characterized with the following quality measures.

Length of a rule is the number of elements in the antecedent.

Coverage of a rule is equal to $supp(A)$.

Consequent support of a rule is equal to $supp(\{c\})$.

Support of a rule is equal to $supp(A \cup \{c\})$.

Confidence of a rule is the fraction $supp(A)/supp(A \cup \{c\})$.

Lift of a rule is the ratio of its support to the expected support assuming antecedent and consequent are independent, i.e., $supp(A \cup \{c\})/(supp(A) * supp(\{c\}))$.

Usage

```
dig_associations(
  x,
  antecedent = everything(),
  consequent = everything(),
  disjoint = var_names(colnames(x)),
  excluded = NULL,
  min_length = 0L,
  max_length = Inf,
  min_coverage = 0,
  min_support = 0,
  min_confidence = 0,
  contingency_table = deprecated(),
  t_norm = "goguen",
  max_results = Inf,
  verbose = FALSE,
  threads = 1,
 error_context = list(arg_x = "x", arg_antecedent = "antecedent", arg_consequent =
  "consequent", arg_disjoint = "disjoint", arg_excluded = "excluded", arg_min_length =
  "min_length", arg_max_length = "max_length", arg_min_coverage = "min_coverage",
    arg_min_support = "min_support", arg_min_confidence = "min_confidence",
  arg_contingency_table = "contingency_table", arg_t_norm = "t_norm", arg_max_results =
    "max_results", arg_verbose = "verbose", arg_threads = "threads", call =
    current_env())
)
```

Arguments

x	a matrix or data frame with data to search in. The matrix must be numeric (double) or logical. If x is a data frame then each column must be either numeric (double) or logical.
antecedent	a tidyselect expression (see tidyselect syntax) specifying the columns to use in the antecedent (left) part of the rules
consequent	a tidyselect expression (see tidyselect syntax) specifying the columns to use in the consequent (right) part of the rules
disjoint	an atomic vector of size equal to the number of columns of x that specifies the groups of predicates: if some elements of the disjoint vector are equal, then the corresponding columns of x will NOT be present together in a single condition. If x is prepared with partition(), using the var_names() function on x's column names is a convenient way to create the disjoint vector.
excluded	NULL or a list of character vectors, where each character vector contains the names of columns that must not appear together in a single antecedent.

`min_length` the minimum length, i.e., the minimum number of predicates in the antecedent, of a rule to be generated. Value must be greater or equal to 0. If 0, rules with empty antecedent are generated in the first place.

`max_length` The maximum length, i.e., the maximum number of predicates in the antecedent, of a rule to be generated. If equal to Inf, the maximum length is limited only by the number of available predicates.

`min_coverage` the minimum coverage of a rule in the dataset `x`. (See Description for the definition of *coverage*.)

`min_support` the minimum support of a rule in the dataset `x`. (See Description for the definition of *support*.)

`min_confidence` the minimum confidence of a rule in the dataset `x`. (See Description for the definition of *confidence*.)

`contingency_table`
(Deprecated. Contingency table is always added to the result.) A logical value indicating whether to provide a contingency table for each rule. If `TRUE`, the columns `pp`, `pn`, `np`, and `nn` are added to the output table. These columns contain the number of rows satisfying the antecedent and the consequent, the antecedent but not the consequent, the consequent but not the antecedent, and neither the antecedent nor the consequent, respectively.

`t_norm` a t-norm used to compute conjunction of weights. It must be one of `"goedel"` (minimum t-norm), `"goguen"` (product t-norm), or `"lukas"` (Łukasiewicz t-norm).

`max_results` the maximum number of generated conditions to execute the callback function on. If the number of found conditions exceeds `max_results`, the function stops generating new conditions and returns the results. To avoid long computations during the search, it is recommended to set `max_results` to a reasonable positive value. Setting `max_results` to `Inf` will generate all possible conditions.

`verbose` a logical value indicating whether to print progress messages.

`threads` the number of threads to use for parallel computation.

`error_context` a named list providing context for error messages. This is mainly useful when `dig_associations()` is called from another function and you want error messages to refer to the argument names of that calling function. The list must contain the following elements:

- `arg_x` - name of the argument `x`
- `arg_antecedent` - name of the argument `antecedent`
- `arg_consequent` - name of the argument `consequent`
- `arg_disjoint` - name of the argument `disjoint`
- `arg_excluded` - name of the argument `excluded`
- `arg_min_length` - name of the argument `min_length`
- `arg_max_length` - name of the argument `max_length`
- `arg_min_coverage` - name of the argument `min_coverage`
- `arg_min_support` - name of the argument `min_support`
- `arg_min_confidence` - name of the argument `min_confidence`
- `arg_contingency_table` - name of the argument `contingency_table`

- `arg_t_norm` - name of the argument `t_norm`
- `arg_max_results` - name of the argument `max_results`
- `arg_verbose` - name of the argument `verbose`
- `arg_threads` - name of the argument `threads`

Value

An S3 object, which is an instance of `associations` and `nugget` classes, and which is a tibble with found patterns and computed quality measures.

Author(s)

Michal Burda

See Also

`partition()`, `var_names()`, `dig()`

Examples

```
d <- partition(mtcars, .breaks = 2)
dig_associations(d,
                 antecedent = !starts_with("mpg"),
                 consequent = starts_with("mpg"),
                 min_support = 0.3,
                 min_confidence = 0.8)
```

---

dig_baseline_contrasts

Search for conditions that yield in statistically significant one-sample test in selected variables.

Description

[Experimental]

Baseline contrast patterns identify conditions under which a specific feature is significantly different from a given value by performing a one-sample statistical test.

Scheme: `var != 0 | C`

Variable `var` is (in average) significantly different from 0 under the condition `C`.

Example: `(measure_error != 0 | measure_tool_A`

If measuring with measure tool *A*, the average measure error is significantly different from 0.

The baseline contrast is computed using a one-sample statistical test, which is specified by the `method` argument. The function computes the contrast between all variables specified by the `vars` argument. Baseline contrasts are computed in sub-data corresponding to conditions generated from the `condition` columns. Function `dig_baseline_contrasts()` supports crisp conditions only, i.e., the condition columns in `x` must be logical.

Usage

```
dig_baseline_contrasts(
  x,
  condition = where(is.logical),
  vars = where(is.numeric),
  disjoint = var_names(colnames(x)),
  excluded = NULL,
  min_length = 0L,
  max_length = Inf,
  min_support = 0,
  max_support = 1,
  method = "t",
  alternative = "two.sided",
  h0 = 0,
  conf_level = 0.95,
  max_p_value = 0.05,
  wilcox_exact = FALSE,
  wilcox_correct = TRUE,
  wilcox_tol_root = 1e-04,
  wilcox_digits_rank = Inf,
  max_results = Inf,
  verbose = FALSE,
  threads = 1
)
```

Arguments

`x`	a matrix or data frame with data to search the patterns in.
`condition`	a tidyselect expression (see tidyselect syntax) specifying the columns to use as condition predicates
`vars`	a tidyselect expression (see tidyselect syntax) specifying the columns to use for computation of contrasts
`disjoint`	an atomic vector of size equal to the number of columns of `x` that specifies the groups of predicates: if some elements of the `disjoint` vector are equal, then the corresponding columns of `x` will NOT be present together in a single condition. If `x` is prepared with `partition()`, using the `var_names()` function on `x`'s column names is a convenient way to create the `disjoint` vector.
`excluded`	NULL or a list of character vectors, where each character vector contains the names of columns that must not appear together in a single condition.

`min_length` the minimum size (the minimum number of predicates) of the condition to be generated (must be greater or equal to 0). If 0, the empty condition is generated in the first place.

`max_length` The maximum size (the maximum number of predicates) of the condition to be generated. If equal to Inf, the maximum length of conditions is limited only by the number of available predicates.

`min_support` the minimum support of a condition to trigger the callback function for it. The support of the condition is the relative frequency of the condition in the dataset `x`. For logical data, it equals to the relative frequency of rows such that all condition predicates are TRUE on it. For numerical (double) input, the support is computed as the mean (over all rows) of multiplications of predicate values.

`max_support` the maximum support of a condition to trigger the callback function for it. See argument `min_support` for details of what is the support of a condition.

`method` a character string indicating which contrast to compute. One of `"t"`, for parametric, or `"wilcox"`, for non-parametric test on equality in position.

`alternative` indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. `"greater"` corresponds to positive association, `"less"` to negative association.

`h0` a numeric value specifying the null hypothesis for the test. For the `"t"` method, it is the value of the mean. For the `"wilcox"` method, it is the value of the median. The default value is 0.

`conf_level` a numeric value specifying the level of the confidence interval. The default value is 0.95.

`max_p_value` the maximum p-value of a test for the pattern to be considered significant. If the p-value of the test is greater than `max_p_value`, the pattern is not included in the result.

`wilcox_exact` (used for the `"wilcox"` method only) a logical value indicating whether the exact p-value should be computed. If `NULL`, the exact p-value is computed for sample sizes less than 50. See `wilcox.test()` and its `exact` argument for more information. Contrary to the behavior of `wilcox.test()`, the default value is `FALSE`.

`wilcox_correct` (used for the `"wilcox"` method only) a logical value indicating whether the continuity correction should be applied in the normal approximation for the p-value, if `wilcox_exact` is `FALSE`. See `wilcox.test()` and its `correct` argument for more information.

`wilcox_tol_root` (used for the `"wilcox"` method only) a numeric value specifying the tolerance for the root-finding algorithm used to compute the exact p-value. See `wilcox.test()` and its `tol.root` argument for more information.

`wilcox_digits_rank` (used for the `"wilcox"` method only) a numeric value specifying the number of digits to round the ranks to. See `wilcox.test()` and its `digits.rank` argument for more information.

`max_results` the maximum number of generated conditions to execute the callback function on. If the number of found conditions exceeds `max_results`, the function stops

	generating new conditions and returns the results. To avoid long computations during the search, it is recommended to set `max_results` to a reasonable positive value. Setting `max_results` to `Inf` will generate all possible conditions.
`verbose`	a logical scalar indicating whether to print progress messages.
`threads`	the number of threads to use for parallel computation.

Value

An S3 object which is an instance of `baseline_contrasts` and `nugget` classes and which is a tibble with found patterns in rows. The following columns are always present:

`condition`	the condition of the pattern as a character string in the form `{p1 & p2 & ... & pn}` where `p1`, `p2`, ..., `pn` are `x`'s column names.
`support`	the support of the condition, i.e., the relative frequency of the condition in the dataset `x`.
`var`	the name of the contrast variable.
`estimate`	the estimated mean or median of variable `var`.
`statistic`	the statistic of the selected test.
`p_value`	the p-value of the underlying test.
`n`	the number of rows in the sub-data corresponding to the condition.
`conf_int_lo`	the lower bound of the confidence interval of the estimate.
`conf_int_hi`	the upper bound of the confidence interval of the estimate.
`alternative`	a character string indicating the alternative hypothesis. The value must be one of `"two.sided"`, `"greater"`, or `"less"`.
`method`	a character string indicating the method used for the test.
`comment`	a character string with additional information about the test (mainly error messages on failure).

For the `"t"` method, the following additional columns are also present (see also `t.test()`):

`df`	the degrees of freedom of the t test.
`stderr`	the standard error of the mean.

Author(s)

Michal Burda

See Also

`dig_paired_baseline_contrasts()`, `dig_complement_contrasts()`, `dig()`, `dig_grid()`, `stats::t.test()`, `stats::wilcox.test()`

Examples

```
d <- partition(mtcars, .breaks = 2, .keep = TRUE)
dig_baseline_contrasts(d,
                       condition = where(is.logical),
                       vars = where(is.numeric),
                       min_support = 0.3,
                       max_length = 2)
```

`dig_complement_contrasts`

Search for conditions that provide significant differences in selected variables to the rest of the data table

---

Description

[Experimental]

Complement contrast patterns identify conditions under which there is a significant difference in some numerical variable between elements that satisfy the identified condition and the rest of the data table.

Scheme: `(var | C) != (var | not C)`

> There is a statistically significant difference in variable `var` between group of elements that satisfy condition `C` and a group of elements that do not satisfy condition `C`.

Example: `(life_expectancy | smoker) < (life_expectancy | non-smoker)`

> The life expectancy in people that smoke cigarettes is in average significantly lower than in people that do not smoke.

The complement contrast is computed using a two-sample statistical test, which is specified by the `method` argument. The function computes the complement contrast in all variables specified by the `vars` argument. Complement contrasts are computed based on sub-data corresponding to conditions generated from the `condition` columns and the rest of the data table. Function #' `dig_complement_contrasts()` supports crisp conditions only, i.e., the condition columns in `x` must be logical.

Usage

```
dig_complement_contrasts(
  x,
  condition = where(is.logical),
  vars = where(is.numeric),
  disjoint = var_names(colnames(x)),
  excluded = NULL,
  min_length = 0L,
  max_length = Inf,
  min_support = 0,
```

```
  max_support = 1 - min_support,
  method = "t",
  alternative = "two.sided",
  h0 = if (method == "var") 1 else 0,
  conf_level = 0.95,
  max_p_value = 0.05,
  t_var_equal = FALSE,
  wilcox_exact = FALSE,
  wilcox_correct = TRUE,
  wilcox_tol_root = 1e-04,
  wilcox_digits_rank = Inf,
  max_results = Inf,
  verbose = FALSE,
  threads = 1L
)
```

Arguments

x	a matrix or data frame with data to search the patterns in.
condition	a tidyselect expression (see tidyselect syntax) specifying the columns to use as condition predicates
vars	a tidyselect expression (see tidyselect syntax) specifying the columns to use for computation of contrasts
disjoint	an atomic vector of size equal to the number of columns of x that specifies the groups of predicates: if some elements of the disjoint vector are equal, then the corresponding columns of x will NOT be present together in a single condition. If x is prepared with partition(), using the var_names() function on x's column names is a convenient way to create the disjoint vector.
excluded	NULL or a list of character vectors, where each character vector contains the names of columns that must not appear together in a single condition.
min_length	the minimum size (the minimum number of predicates) of the condition to be generated (must be greater or equal to 0). If 0, the empty condition is generated in the first place.
max_length	The maximum size (the maximum number of predicates) of the condition to be generated. If equal to Inf, the maximum length of conditions is limited only by the number of available predicates.
min_support	the minimum support of a condition to trigger the callback function for it. The support of the condition is the relative frequency of the condition in the dataset x. For logical data, it equals to the relative frequency of rows such that all condition predicates are TRUE on it. For numerical (double) input, the support is computed as the mean (over all rows) of multiplications of predicate values.
max_support	the maximum support of a condition to trigger the callback function for it. See argument min_support for details of what is the support of a condition.
method	a character string indicating which contrast to compute. One of "t", for parametric, or "wilcox", for non-parametric test on equality in position, and "var" for F-test on comparison of variances of two populations.

`alternative` indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. `"greater"` corresponds to positive association, `"less"` to negative association.

`h0` a numeric value specifying the null hypothesis for the test. For the `"t"` method, it is the difference in means. For the `"wilcox"` method, it is the difference in medians. For the `"var"` method, it is the hypothesized ratio of the population variances. The default value is 1 for `"var"` method, and 0 otherwise.

`conf_level` a numeric value specifying the level of the confidence interval. The default value is 0.95.

`max_p_value` the maximum p-value of a test for the pattern to be considered significant. If the p-value of the test is greater than `max_p_value`, the pattern is not included in the result.

`t_var_equal` (used for the `"t"` method only) a logical value indicating whether the variances of the two samples are assumed to be equal. If `TRUE`, the pooled variance is used to estimate the variance in the t-test. If `FALSE`, the Welch (or Satterthwaite) approximation to the degrees of freedom is used. See `t.test()` and its `var.equal` argument for more information.

`wilcox_exact` (used for the `"wilcox"` method only) a logical value indicating whether the exact p-value should be computed. If `NULL`, the exact p-value is computed for sample sizes less than 50. See `wilcox.test()` and its `exact` argument for more information. Contrary to the behavior of `wilcox.test()`, the default value is `FALSE`.

`wilcox_correct` (used for the `"wilcox"` method only) a logical value indicating whether the continuity correction should be applied in the normal approximation for the p-value, if `wilcox_exact` is `FALSE`. See `wilcox.test()` and its `correct` argument for more information.

`wilcox_tol_root` (used for the `"wilcox"` method only) a numeric value specifying the tolerance for the root-finding algorithm used to compute the exact p-value. See `wilcox.test()` and its `tol.root` argument for more information.

`wilcox_digits_rank` (used for the `"wilcox"` method only) a numeric value specifying the number of digits to round the ranks to. See `wilcox.test()` and its `digits.rank` argument for more information.

`max_results` the maximum number of generated conditions to execute the callback function on. If the number of found conditions exceeds `max_results`, the function stops generating new conditions and returns the results. To avoid long computations during the search, it is recommended to set `max_results` to a reasonable positive value. Setting `max_results` to `Inf` will generate all possible conditions.

`verbose` a logical scalar indicating whether to print progress messages.

`threads` the number of threads to use for parallel computation.

Value

An S3 object which is an instance of `complement_contrasts` and `nugget` classes and which is a tibble with found patterns in rows. The following columns are always present:

condition	the condition of the pattern as a character string in the form {p1 & p2 & ... & pn} where p1, p2, ..., pn are x's column names.
support	the support of the condition, i.e., the relative frequency of the condition in the dataset x.
var	the name of the contrast variable.
estimate	the estimate value (see the underlying test.
statistic	the statistic of the selected test.
p_value	the p-value of the underlying test.
n_x	the number of rows in the sub-data corresponding to the condition.
n_y	the number of rows in the sub-data corresponding to the negation of the condition.
conf_int_lo	the lower bound of the confidence interval of the estimate.
conf_int_hi	the upper bound of the confidence interval of the estimate.
alternative	a character string indicating the alternative hypothesis. The value must be one of "two.sided", "greater", or "less".
method	a character string indicating the method used for the test.
comment	a character string with additional information about the test (mainly error messages on failure).

For the "t" method, the following additional columns are also present (see also t.test()):

df	the degrees of freedom of the t test.
stderr	the standard error of the mean difference.

Author(s)

Michal Burda

See Also

dig_baseline_contrasts(), dig_paired_baseline_contrasts(), dig(), dig_grid(), stats::t.test(), stats::wilcox.test(), stats::var.test()

Examples

```
d <- partition(mtcars, .breaks = 2, .keep = TRUE)
dig_complement_contrasts(d,
                         condition = where(is.logical),
                         vars = where(is.numeric),
                         min_support = 0.3,
                         max_length = 2)
```

`dig_correlations` *Search for conditional correlations*

Description

[Experimental]

Conditional correlations are patterns that identify strong relationships between pairs of numeric variables under specific conditions.

Scheme: `xvar ~ yvar | C`

`xvar` and `yvar` highly correlates in data that satisfy the condition `C`.

Example: `study_time ~ test_score | hard_exam`

For *hard exams*, the amount of *study time* is highly correlated with the obtained exam's *test score*.

The function computes correlations between all combinations of `xvars` and `yvars` columns of `x` in multiple sub-data corresponding to conditions generated from `condition` columns.

Usage

```
dig_correlations(
  x,
  condition = where(is.logical),
  xvars = where(is.numeric),
  yvars = where(is.numeric),
  disjoint = var_names(colnames(x)),
  excluded = NULL,
  method = "pearson",
  alternative = "two.sided",
  exact = NULL,
  min_length = 0L,
  max_length = Inf,
  min_support = 0,
  max_support = 1,
  max_results = Inf,
  verbose = FALSE,
  threads = 1
)
```

Arguments

`x`	a matrix or data frame with data to search in.
`condition`	a tidyselect expression (see tidyselect syntax) specifying the columns to use as condition predicates

xvars	a tidyselect expression (see tidyselect syntax) specifying the columns to use for computation of correlations
yvars	a tidyselect expression (see tidyselect syntax) specifying the columns to use for computation of correlations
disjoint	an atomic vector of size equal to the number of columns of x that specifies the groups of predicates: if some elements of the disjoint vector are equal, then the corresponding columns of x will NOT be present together in a single condition. If x is prepared with partition(), using the var_names() function on x's column names is a convenient way to create the disjoint vector.
excluded	NULL or a list of character vectors, where each character vector contains the names of columns that must not appear together in a single condition.
method	a character string indicating which correlation coefficient is to be used for the test. One of "pearson", "kendall", or "spearman"
alternative	indicates the alternative hypothesis and must be one of "two.sided", "greater" or "less". "greater" corresponds to positive association, "less" to negative association.
exact	a logical indicating whether an exact p-value should be computed. Used for Kendall's *tau* and Spearman's *rho*. See stats::cor.test() for more information.
min_length	the minimum size (the minimum number of predicates) of the condition to be generated (must be greater or equal to 0). If 0, the empty condition is generated in the first place.
max_length	The maximum size (the maximum number of predicates) of the condition to be generated. If equal to Inf, the maximum length of conditions is limited only by the number of available predicates.
min_support	the minimum support of a condition to trigger the callback function for it. The support of the condition is the relative frequency of the condition in the dataset x. For logical data, it equals to the relative frequency of rows such that all condition predicates are TRUE on it. For numerical (double) input, the support is computed as the mean (over all rows) of multiplications of predicate values.
max_support	the maximum support of a condition to trigger the callback function for it. See argument min_support for details of what is the support of a condition.
max_results	the maximum number of generated conditions to execute the callback function on. If the number of found conditions exceeds max_results, the function stops generating new conditions and returns the results. To avoid long computations during the search, it is recommended to set max_results to a reasonable positive value. Setting max_results to Inf will generate all possible conditions.
verbose	a logical scalar indicating whether to print progress messages.
threads	the number of threads to use for parallel computation.

Value

An S3 object which is an instance of correlations and nugget classes and which is tibble with found patterns.

Author(s)

Michal Burda

See Also

dig(), stats::cor.test()

Examples

```
# convert iris$Species into dummy logical variables
d <- partition(iris, Species)

# find conditional correlations between all pairs of numeric variables
dig_correlations(d,
                 condition = where(is.logical),
                 xvars = Sepal.Length:Petal.Width,
                 yvars = Sepal.Length:Petal.Width)

# With `condition = NULL`, dig_correlations() computes correlations between
# all pairs of numeric variables on the whole dataset only, which is an
# alternative way of computing the correlation matrix
dig_correlations(iris,
                 condition = NULL,
                 xvars = Sepal.Length:Petal.Width,
                 yvars = Sepal.Length:Petal.Width)
```

---

dig_grid *Search for grid-based rules*

Description

[Experimental]

This function creates a grid column names specified by `xvars` and `yvars` (see var_grid()). After that, it enumerates all conditions created from data in `x` (by calling dig()) and for each such condition and for each row of the grid of combinations, a user-defined function `f` is executed on each sub-data created from `x` by selecting all rows of `x` that satisfy the generated condition and by selecting the columns in the grid's row.

Function is useful for searching for patterns that are based on the relationships between pairs of columns, such as in dig_correlations().

Usage

```
dig_grid(
  x,
  f,
  condition = where(is.logical),
  xvars = where(is.numeric),
```

```
  yvars = where(is.numeric),
  disjoint = var_names(colnames(x)),
  excluded = NULL,
  allow = "all",
  na_rm = FALSE,
  type = "crisp",
  min_length = 0L,
  max_length = Inf,
  min_support = 0,
  max_support = 1,
  max_results = Inf,
  verbose = FALSE,
  threads = 1L,
 error_context = list(arg_x = "x", arg_f = "f", arg_condition = "condition", arg_xvars =
  "xvars", arg_yvars = "yvars", arg_disjoint = "disjoint", arg_excluded = "excluded",
   arg_allow = "allow", arg_na_rm = "na_rm", arg_type = "type", arg_min_length =
   "min_length", arg_max_length = "max_length", arg_min_support = "min_support",
  arg_max_support = "max_support", arg_max_results = "max_results", arg_verbose =
    "verbose", arg_threads = "threads", call = current_env())
)
```

Arguments

x	a matrix or data frame with data to search in.
f	the callback function to be executed for each generated condition. The arguments of the callback function differ based on the value of the type argument (see below): • If type = "crisp" (that is, boolean), the callback function f must accept a single argument pd of type data.frame with single (if yvars == NULL) or two (if yvars != NULL) columns, accessible as pd[[1]] and pd[[2]]. Data frame pd is a subset of the original data frame x with all rows that satisfy the generated condition. Optionally, the callback function may accept an argument nd that is a subset of the original data frame x with all rows that do not satisfy the generated condition. • If type = "fuzzy", the callback function f must accept an argument d of type data.frame with single (if yvars == NULL) or two (if yvars != NULL) columns, accessible as d[[1]] and d[[2]], and a numeric argument weights with the same length as the number of rows in d. The weights argument contains the truth degree of the generated condition for each row of d. The truth degree is a number in the interval $[0, 1]$ that represents the degree of satisfaction of the condition in the original data row. In all cases, the function must return a list of scalar values, which will be converted into a single row of result of final tibble.
condition	a tidyselect expression (see tidyselect syntax) specifying the columns to use as condition predicates. The selected columns must be logical or numeric. If numeric, fuzzy conditions are considered.
xvars	a tidyselect expression (see tidyselect syntax) specifying the columns of x, whose names will be used as a domain for combinations use at the first place (xvar)

yvars — NULL or a tidyselect expression (see tidyselect syntax) specifying the columns of x, whose names will be used as a domain for combinations use at the second place (yvar)

disjoint — an atomic vector of size equal to the number of columns of x that specifies the groups of predicates: if some elements of the disjoint vector are equal, then the corresponding columns of x will NEITHER be present together in a single condition NOR in a single combination of xvars and yvars. If x is prepared with partition(), using the var_names() function on x's column names is a convenient way to create the disjoint vector.

excluded — NULL or a list of character vectors, where each character vector contains the names of columns that must not appear together in a single condition.

allow — a character string specifying which columns are allowed to be selected by xvars and yvars arguments. Possible values are:

- "all" - all columns are allowed to be selected
- "numeric" - only numeric columns are allowed to be selected

na_rm — a logical value indicating whether to remove rows with missing values from sub-data before the callback function f is called

type — a character string specifying the type of conditions to be processed. The "crisp" type accepts only logical columns as condition predicates. The "fuzzy" type accepts both logical and numeric columns as condition predicates where numeric data are in the interval $[0, 1]$. The callback function f differs based on the value of the type argument (see the description of f above).

min_length — the minimum size (the minimum number of predicates) of the condition to be generated (must be greater or equal to 0). If 0, the empty condition is generated in the first place.

max_length — the maximum size (the maximum number of predicates) of the condition to be generated. If equal to Inf, the maximum length of conditions is limited only by the number of available predicates.

min_support — the minimum support of a condition to trigger the callback function for it. The support of the condition is the relative frequency of the condition in the dataset x. For logical data, it equals to the relative frequency of rows such that all condition predicates are TRUE on it. For numerical (double) input, the support is computed as the mean (over all rows) of multiplications of predicate values.

max_support — the maximum support of a condition to trigger the callback function for it. See argument min_support for details of what is the support of a condition.

max_results — the maximum number of generated conditions to execute the callback function on. If the number of found conditions exceeds max_results, the function stops generating new conditions and returns the results. To avoid long computations during the search, it is recommended to set max_results to a reasonable positive value. Setting max_results to Inf will generate all possible conditions.

verbose — a logical scalar indicating whether to print progress messages.

threads — the number of threads to use for parallel computation.

error_context — a list of details to be used in error messages. This argument is useful when dig_grid() is called from another function to provide error messages, which

refer to arguments of the calling function. The list must contain the following elements:

- `arg_x` - the name of the argument `x` as a character string
- `arg_condition` - the name of the argument `condition` as a character string
- `arg_xvars` - the name of the argument `xvars` as a character string
- `arg_yvars` - the name of the argument `yvars` as a character string
- `call` - an environment in which to evaluate the error messages.

Value

An S3 object, which is an instance of `nugget` class, and which is a tibble with found patterns. Each row represents a single call of the callback function `f`.

Author(s)

Michal Burda

See Also

`dig()`, `var_grid()`; see also `dig_correlations()` and `dig_paired_baseline_contrasts()`, as they are using this function internally.

Examples

```
# *** Example of crisp (boolean) patterns:
# dichotomize iris$Species
crispIris <- partition(iris, Species)

# a simple callback function that computes mean difference of `xvar` and `yvar`
f <- function(pd) {
    list(m = mean(pd[[1]] - pd[[2]]),
         n = nrow(pd))
    }

# call f() for each condition created from column `Species`
dig_grid(crispIris,
         f,
         condition = starts_with("Species"),
         xvars = starts_with("Sepal"),
         yvars = starts_with("Petal"),
         type = "crisp")

# *** Example of fuzzy patterns:
# create fuzzy sets from Sepal columns
fuzzyIris <- partition(iris,
                       starts_with("Sepal"),
                       .method = "triangle",
                       .breaks = 3)

# a simple callback function that computes a weighted mean of a difference of
# `xvar` and `yvar`
```

```
f <- function(d, weights) {
    list(m = weighted.mean(d[[1]] - d[[2]], w = weights),
         w = sum(weights))
}

# call f() for each fuzzy condition created from column fuzzy sets whose
# names start with "Sepal"
dig_grid(fuzzyIris,
         f,
         condition = starts_with("Sepal"),
         xvars = Petal.Length,
         yvars = Petal.Width,
         type = "fuzzy")
```

`dig_paired_baseline_contrasts` *Search for conditions that provide significant differences between paired variables*

Description

[Experimental]

Paired baseline contrast patterns identify conditions under which there is a significant difference in some statistical feature between two paired numeric variables.

Scheme: `(xvar - yvar) != 0 | C`

> There is a statistically significant difference between paired variables `xvar` and `yvar` under the condition `C`.

Example: `(daily_ice_cream_income - daily_tea_income) > 0 | sunny`

> Under the condition of *sunny weather*, the paired test shows that *daily ice-cream income* is significantly higher than the *daily tea income*.

The paired baseline contrast is computed using a paired version of a statistical test, which is specified by the `method` argument. The function computes the paired contrast between all pairs of variables, where the first variable is specified by the `xvars` argument and the second variable is specified by the `yvars` argument. Paired baseline contrasts are computed in sub-data corresponding to conditions generated from the `condition` columns. Function `dig_paired_baseline_contrasts()` supports crisp conditions only, i.e., the condition columns in `x` must be logical.

Usage

```
dig_paired_baseline_contrasts(
  x,
  condition = where(is.logical),
  xvars = where(is.numeric),
  yvars = where(is.numeric),
```

```
  disjoint = var_names(colnames(x)),
  excluded = NULL,
  min_length = 0L,
  max_length = Inf,
  min_support = 0,
  max_support = 1,
  method = "t",
  alternative = "two.sided",
  h0 = 0,
  conf_level = 0.95,
  max_p_value = 1,
  t_var_equal = FALSE,
  wilcox_exact = FALSE,
  wilcox_correct = TRUE,
  wilcox_tol_root = 1e-04,
  wilcox_digits_rank = Inf,
  max_results = Inf,
  verbose = FALSE,
  threads = 1
)
```

Arguments

x	a matrix or data frame with data to search the patterns in.
condition	a tidyselect expression (see tidyselect syntax) specifying the columns to use as condition predicates
xvars	a tidyselect expression (see tidyselect syntax) specifying the columns to use for computation of contrasts
yvars	a tidyselect expression (see tidyselect syntax) specifying the columns to use for computation of contrasts
disjoint	an atomic vector of size equal to the number of columns of x that specifies the groups of predicates: if some elements of the disjoint vector are equal, then the corresponding columns of x will NOT be present together in a single condition. If x is prepared with partition(), using the var_names() function on x's column names is a convenient way to create the disjoint vector.
excluded	NULL or a list of character vectors, where each character vector contains the names of columns that must not appear together in a single condition.
min_length	the minimum size (the minimum number of predicates) of the condition to be generated (must be greater or equal to 0). If 0, the empty condition is generated in the first place.
max_length	The maximum size (the maximum number of predicates) of the condition to be generated. If equal to Inf, the maximum length of conditions is limited only by the number of available predicates.
min_support	the minimum support of a condition to trigger the callback function for it. The support of the condition is the relative frequency of the condition in the dataset x. For logical data, it equals to the relative frequency of rows such that all

condition predicates are TRUE on it. For numerical (double) input, the support is computed as the mean (over all rows) of multiplications of predicate values.

`max_support` the maximum support of a condition to trigger the callback function for it. See argument `min_support` for details of what is the support of a condition.

`method` a character string indicating which contrast to compute. One of `"t"`, for parametric, or `"wilcox"`, for non-parametric test on equality in position.

`alternative` indicates the alternative hypothesis and must be one of `"two.sided"`, `"greater"` or `"less"`. `"greater"` corresponds to positive association, `"less"` to negative association.

`h0` a numeric value specifying the null hypothesis for the test. For the `"t"` method, it is the difference in means. For the `"wilcox"` method, it is the difference in medians. The default value is 0.

`conf_level` a numeric value specifying the level of the confidence interval. The default value is 0.95.

`max_p_value` the maximum p-value of a test for the pattern to be considered significant. If the p-value of the test is greater than `max_p_value`, the pattern is not included in the result.

`t_var_equal` (used for the `"t"` method only) a logical value indicating whether the variances of the two samples are assumed to be equal. If `TRUE`, the pooled variance is used to estimate the variance in the t-test. If `FALSE`, the Welch (or Satterthwaite) approximation to the degrees of freedom is used. See `t.test()` and its `var.equal` argument for more information.

`wilcox_exact` (used for the `"wilcox"` method only) a logical value indicating whether the exact p-value should be computed. If `NULL`, the exact p-value is computed for sample sizes less than 50. See `wilcox.test()` and its `exact` argument for more information. Contrary to the behavior of `wilcox.test()`, the default value is `FALSE`.

`wilcox_correct` (used for the `"wilcox"` method only) a logical value indicating whether the continuity correction should be applied in the normal approximation for the p-value, if `wilcox_exact` is `FALSE`. See `wilcox.test()` and its `correct` argument for more information.

`wilcox_tol_root`
(used for the `"wilcox"` method only) a numeric value specifying the tolerance for the root-finding algorithm used to compute the exact p-value. See `wilcox.test()` and its `tol.root` argument for more information.

`wilcox_digits_rank`
(used for the `"wilcox"` method only) a numeric value specifying the number of digits to round the ranks to. See `wilcox.test()` and its `digits.rank` argument for more information.

`max_results` the maximum number of generated conditions to execute the callback function on. If the number of found conditions exceeds `max_results`, the function stops generating new conditions and returns the results. To avoid long computations during the search, it is recommended to set `max_results` to a reasonable positive value. Setting `max_results` to `Inf` will generate all possible conditions.

`verbose` a logical scalar indicating whether to print progress messages.

`threads` the number of threads to use for parallel computation.

Value

An S3 object which is an instance of `paired_baseline_contrasts` and `nugget` classes and which is a tibble with found patterns in rows. The following columns are always present:

`condition`	the condition of the pattern as a character string in the form `{p1 & p2 & ... & pn}` where `p1`, `p2`, ..., `pn` are `x`'s column names.
`support`	the support of the condition, i.e., the relative frequency of the condition in the dataset `x`.
`xvar`	the name of the first variable in the contrast.
`yvar`	the name of the second variable in the contrast.
`estimate`	the estimated difference of variable `var`.
`statistic`	the statistic of the selected test.
`p_value`	the p-value of the underlying test.
`n`	the number of rows in the sub-data corresponding to the condition.
`conf_int_lo`	the lower bound of the confidence interval of the estimate.
`conf_int_hi`	the upper bound of the confidence interval of the estimate.
`alternative`	a character string indicating the alternative hypothesis. The value must be one of `"two.sided"`, `"greater"`, or `"less"`.
`method`	a character string indicating the method used for the test.
`comment`	a character string with additional information about the test (mainly error messages on failure).

For the `"t"` method, the following additional columns are also present (see also `t.test()`):

`df`	the degrees of freedom of the t test.
`stderr`	the standard error of the mean difference.

Author(s)

Michal Burda

See Also

`dig_baseline_contrasts()`, `dig_complement_contrasts()`, `dig()`, `dig_grid()`, `stats::t.test()`, `stats::wilcox.test()`

Examples

```
# Compute ratio of sepal and petal length and width for iris dataset
crispIris <- iris
crispIris$Sepal.Ratio <- iris$Sepal.Length / iris$Sepal.Width
crispIris$Petal.Ratio <- iris$Petal.Length / iris$Petal.Width

# Create predicates from the Species column
crispIris <- partition(crispIris, Species)
```

```
# Compute paired contrasts for ratios of sepal and petal length and width
dig_paired_baseline_contrasts(crispIris,
                              condition = where(is.logical),
                              xvars = Sepal.Ratio,
                              yvars = Petal.Ratio,
                              method = "t",
                              min_support = 0.1)
```

`dig_tautologies` *Find tautologies or "almost tautologies" in a dataset*

Description

This function finds tautologies in a dataset, i.e., rules of the form `{a1 & a2 & ... & an} => {c}` where `a1`, `a2`, ..., `an` are antecedents and `c` is a consequent. The intent of searching for tautologies is to find rules that are always true, which may be used for filtering of further generated conditions. The resulting rules may be used as a basis for the list of `excluded` formulae (see the `excluded` argument of `dig()`).

The search for tautologies is performed by iteratively searching for rules with increasing length of the antecedent. Rules found in previous iterations are used as `excluded` argument in the next iteration.

Usage

```
dig_tautologies(
  x,
  antecedent = everything(),
  consequent = everything(),
  disjoint = var_names(colnames(x)),
  max_length = Inf,
  min_coverage = 0,
  min_support = 0,
  min_confidence = 0,
  contingency_table = deprecated(),
  t_norm = "goguen",
  max_results = Inf,
  verbose = FALSE,
  threads = 1
)
```

Arguments

`x`	a matrix or data frame with data to search in. The matrix must be numeric (double) or logical. If `x` is a data frame then each column must be either numeric (double) or logical.
`antecedent`	a tidyselect expression (see tidyselect syntax) specifying the columns to use in the antecedent (left) part of the rules

consequent — a tidyselect expression (see tidyselect syntax) specifying the columns to use in the consequent (right) part of the rules

disjoint — an atomic vector of size equal to the number of columns of x that specifies the groups of predicates: if some elements of the disjoint vector are equal, then the corresponding columns of x will NOT be present together in a single condition. If x is prepared with partition(), using the var_names() function on x's column names is a convenient way to create the disjoint vector.

max_length — The maximum length, i.e., the maximum number of predicates in the antecedent, of a rule to be generated. If equal to Inf, the maximum length is limited only by the number of available predicates.

min_coverage — the minimum coverage of a rule in the dataset x. (See Description for the definition of *coverage*.)

min_support — the minimum support of a rule in the dataset x. (See Description for the definition of *support*.)

min_confidence — the minimum confidence of a rule in the dataset x. (See Description for the definition of *confidence*.)

contingency_table — (Deprecated.) A logical value indicating whether to provide a contingency table for each rule. If TRUE, the columns pp, pn, np, and nn are added to the output table. These columns contain the number of rows satisfying the antecedent and the consequent, the antecedent but not the consequent, the consequent but not the antecedent, and neither the antecedent nor the consequent, respectively.

t_norm — a t-norm used to compute conjunction of weights. It must be one of "goedel" (minimum t-norm), "goguen" (product t-norm), or "lukas" (Łukasiewicz t-norm).

max_results — the maximum number of generated conditions to execute the callback function on. If the number of found conditions exceeds max_results, the function stops generating new conditions and returns the results. To avoid long computations during the search, it is recommended to set max_results to a reasonable positive value. Setting max_results to Inf will generate all possible conditions.

verbose — a logical value indicating whether to print progress messages.

threads — the number of threads to use for parallel computation.

Value

An S3 object which is an instance of associations and nugget classes and which is a tibble with found tautologies in the format equal to the output of dig_associations().

Author(s)

Michal Burda

Examples

```
d <- partition(mtcars, .breaks = 2)
dig_tautologies(d,
```

```
                antecedent = everything(),
                consequent = everything(),
                min_confidence = 0.99)
```

`explore.associations` *Show interactive application to explore association rules*

Description

[Experimental]

Launches an interactive Shiny application for visual exploration of mined association rules. The explorer provides tools for inspecting rule quality, comparing interestingness measures, and interactively filtering subsets of rules. When the original dataset is supplied, the application also allows for contextual exploration of rules with respect to the underlying data.

Usage

```
## S3 method for class 'associations'
explore(x, data = NULL, ...)
```

Arguments

`x`	An object of S3 class `associations`, typically created with `dig_associations()`.
`data`	An optional data frame containing the dataset from which the rules were mined. Providing this enables additional contextual features in the explorer, such as examining supporting records.
`...`	Currently ignored.

Value

An object of class `shiny.appobj` representing the Shiny application. When "printed" in an interactive R session, the application is launched immediately in the default web browser.

Author(s)

Michal Burda

See Also

`dig_associations()`

Examples

```
## Not run:
data("iris")
# convert all columns into dummy logical variables
part <- partition(iris, .breaks = 3)

# find association rules
rules <- dig_associations(part)

# launch the interactive explorer
explore(rules, data = part)

## End(Not run)
```

`fire` *Obtain truth-degrees of conditions*

Description

Given a data frame or matrix of truth values for predicates, compute the truth values of a set of conditions expressed as elementary conjunctions.

Each element of `condition` must be a character string of the format `"{p1,p2,p3}"`, where `"p1"`, `"p2"`, and `"p3"` are predicate names. The data object `x` must contain columns whose names correspond exactly to all predicates referenced in the conditions. Each condition is evaluated for every row of `x` as a conjunction of its predicates, with the conjunction operation determined by the `t_norm` argument. An empty condition (`"{}"`) is always evaluated as 1 (i.e., fully true).

Usage

```
fire(x, condition, t_norm = "goguen")
```

Arguments

`x`	A matrix or data frame containing predicate truth values. If `x` is a matrix, it must be numeric (double) or logical. If `x` is a data frame, all columns must be numeric (double) or logical.
`condition`	A character vector of conditions, each formatted according to `format_condition()`. For example, `"{p1,p2,p3}"` represents a condition composed of three predicates `"p1"`, `"p2"`, and `"p3"`. Every predicate mentioned in `condition` must be present as a column in `x`.
`t_norm`	A string specifying the triangular norm (t-norm) used to compute conjunctions of predicate values. Must be one of `"goedel"` (minimum t-norm), `"goguen"` (product t-norm), or `"lukas"` (Łukasiewicz t-norm).

Value

A numeric matrix with entries in the interval $[0, 1]$ giving the truth degrees of the conditions. The matrix has `nrow(x)` rows and `length(condition)` columns. The element in row *i* and column *j* corresponds to the truth degree of the *j*-th condition evaluated on the *i*-th row of `x`.

Author(s)

Michal Burda

See Also

`format_condition()`, `partition()`

Examples

```
d <- data.frame(
  a = c(1, 0.8, 0.5, 0.2, 0),
  b = c(0.5, 1, 0.5, 0, 1),
  c = c(0.9, 0.9, 0.1, 0.8, 0.7)
)

# Evaluate conditions with different t-norms
fire(d, c("{a,c}", "{}", "{a,b,c}"), t_norm = "goguen")
fire(d, c("{a,c}", "{a,b}"), t_norm = "goedel")
fire(d, c("{b,c}"), t_norm = "lukas")
```

---

format_condition *Format a vector of predicates into a condition string*

Description

Convert a character vector of predicate names into a standardized string representation of a condition. Predicates are concatenated with commas and enclosed in curly braces. This formatting ensures consistency when storing or comparing conditions in other functions.

Usage

```
format_condition(condition)
```

Arguments

`condition`	A character vector of predicate names to be formatted. If `NULL` or of length zero, the result is `"{}"`, representing an empty condition that is always true.

Value

A character scalar containing the formatted condition string.

Author(s)

Michal Burda

See Also

parse_condition(), fire()

Examples

```
format_condition(NULL)
format_condition(character(0))
format_condition(c("a", "b", "c"))
```

---

geom_diamond *Geom for drawing diamond plots of lattice structures*

Description

Create a custom ggplot2 geom for visualizing lattice structures as *diamond plots*. This geom is particularly useful for displaying association rules and their ancestor–descendant relationships in a clear, compact graphical form.

In a diamond plot, nodes (diamonds) represent items or conditions within the lattice, while edges denote inclusion (subset) relationships between them. The geom combines node and edge rendering with flexible control over aesthetics such as labels, color, and size.

Usage

```
geom_diamond(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  na.rm = FALSE,
  linetype = "solid",
  linewidth = NA,
  nudge_x = 0,
  nudge_y = 0.125,
  show.legend = NA,
  inherit.aes = TRUE,
  ...
)
```

Arguments

mapping	Aesthetic mappings, usually created with ggplot2::aes().
data	A data frame representing the lattice structure to plot.
stat	Statistical transformation to apply; defaults to "identity".
position	Position adjustment for the geom; defaults to "identity".
na.rm	Logical; if TRUE, missing values are silently removed.
linetype	Line type for edges; defaults to "solid".
linewidth	Width of edges connecting parent and child nodes. If set to NA, edge widths are determined by the linewidth aesthetic. If no aesthetic is provided, a default width of 0.5 is used.
nudge_x	Horizontal nudge applied to labels.
nudge_y	Vertical nudge applied to labels.
show.legend	Logical; whether to include a legend. Defaults to FALSE.
inherit.aes	Logical; whether to inherit aesthetics from the plot. Defaults to TRUE.
...	Additional arguments passed to ggplot2::layer().

Details

Concept overview

A *lattice* represents inclusion relationships between conditions. Each node corresponds to a condition, and a line connects a condition to its direct descendants:

```
      {a}          <- ancestor (parent)
     /   \
{a,b}   {a,c}     <- direct descendants (children)
   \     /
   {a,b,c}        <- leaf condition
```

The layout positions broader (more general) conditions above their descendants. This helps visualize hierarchical structures such as those produced by association rule mining or subset lattices.

Supported aesthetics

- condition – character vector of conditions formatted with format_condition(). Each condition defines one node in the lattice. The hierarchy is determined by subset inclusion: a condition X is a descendant of Y if $Y \subset X$. Each condition must be unique.
- label – optional text label for each node. If omitted, the condition string is used.
- colour – border color of the node.
- fill – interior color of the node.
- size – size of nodes.
- shape – node shape.
- alpha – transparency of nodes.
- stroke – border line width of nodes.
- linewidth – edge width between parent and child nodes, computed as the difference of this aesthetic between them.

Value

A ggplot2 layer object that adds a diamond lattice visualization to an existing plot.

Author(s)

Michal Burda

Examples

```
## Not run:
library(ggplot2)

# Prepare data by partitioning numeric columns into fuzzy or crisp sets
part <- partition(iris, .breaks = 3)

# Find all antecedents with "Sepal" for rules with consequent "Species=setosa"
rules <- dig_associations(part,
                          antecedent = starts_with("Sepal"),
                          consequent = `Species=setosa`,
                          min_length = 0,
                          max_length = Inf,
                          min_coverage = 0,
                          min_support = 0,
                          min_confidence = 0,
                          measures = c("lift", "conviction"),
                          max_results = Inf)

# Add abbreviated labels for readability
rules$abbrev <- shorten_condition(rules$antecedent)

# Plot the lattice of rules as a diamond diagram
ggplot(rules) +
  aes(condition = antecedent,
      fill = confidence,
      linewidth = confidence,
      size = coverage,
      label = abbrev) +
  geom_diamond()

## End(Not run)
```

---

is_almost_constant *Test whether a vector is almost constant*

Description

Check if a vector contains (almost) the same value in the majority of its elements. The function returns TRUE if the proportion of the most frequent value in x is greater than or equal to the specified threshold.

This is useful for detecting low-variability or degenerate variables, which may be uninformative in modeling or analysis.

Usage

```
is_almost_constant(x, threshold = 1, na_rm = FALSE)
```

Arguments

x	A vector to be tested.
threshold	A numeric scalar in the interval $[0, 1]$ specifying the minimum required proportion of the most frequent value. Defaults to 1.
na_rm	Logical; if TRUE, NA values are removed before computing proportions. If FALSE, NA is treated as an ordinary value, so a large number of NAs can cause the function to return TRUE.

Value

A logical scalar. Returns TRUE in the following cases:

- x is empty or has length one.
- x contains only NA values.
- The proportion of the most frequent value in x is greater than or equal to threshold. Otherwise, returns FALSE.

Author(s)

Michal Burda

See Also

remove_almost_constant(), unique(), table()

Examples

```
is_almost_constant(1)
is_almost_constant(1:10)
is_almost_constant(c(NA, NA, NA), na_rm = TRUE)
is_almost_constant(c(NA, NA, NA), na_rm = FALSE)
is_almost_constant(c(NA, NA, NA, 1, 2), threshold = 0.5, na_rm = FALSE)
is_almost_constant(c(NA, NA, NA, 1, 2), threshold = 0.5, na_rm = TRUE)
```

`is_condition` *Check whether a list of character vectors contains valid conditions*

Description

A valid condition is a character vector of predicate names, where each predicate corresponds to a column name in a given data frame or matrix. This function verifies that each element of a list `x` contains only valid predicates that match column names of `data`.

Special cases:

- An empty character vector (`character(0)`) is considered a valid condition and always passes the check.
- A `NULL` element is treated the same as an empty character vector, i.e., it is also a valid condition.

Usage

```
is_condition(x, data)
```

Arguments

`x`	A list of character vectors, each representing a condition.
`data`	A matrix or data frame whose column names define valid predicates.

Value

A logical vector with one element for each condition in `x`. An element is `TRUE` if the corresponding condition is valid, i.e. all of its predicates are column names of `data`. Otherwise, it is `FALSE`.

Author(s)

Michal Burda

See Also

`remove_ill_conditions()`, `format_condition()`

Examples

```
d <- data.frame(foo = 1:5, bar = 1:5, blah = 1:5)

is_condition(list("foo"), d)
is_condition(list(c("bar", "blah"), NULL, c("foo", "bzz")), d)
```

`is_degree` *Test whether an object contains numeric values from the interval* $[0, 1]$

Description

Check if the input consists only of numeric values between 0 and 1, inclusive. This is often useful when validating truth degrees, membership values in fuzzy sets, or probabilities.

Usage

```
is_degree(x, na_rm = FALSE)
```

Arguments

`x`	The object to be tested. Can be a numeric vector, matrix, or array.
`na_rm`	Logical; whether to ignore `NA` values. If `TRUE`, `NA`s are treated as valid values. If `FALSE` and `x` contains any `NA`s, the function immediately returns `FALSE`.

Value

A logical scalar. Returns `TRUE` if all (non-`NA`) elements of `x` are numeric and lie within the closed interval $[0, 1]$. Returns `FALSE` if:

- `x` contains any `NA` values and `na_rm = FALSE`
- any element is outside the interval $[0, 1]$
- `x` is not numeric
- `x` is empty (`length(x) == 0`)

Author(s)

Michal Burda

See Also

`is.numeric()`

Examples

```
is_degree(0.5)
is_degree(c(0, 0.2, 1))
is_degree(c(0.5, NA), na_rm = TRUE)   # TRUE
is_degree(c(0.5, NA), na_rm = FALSE)  # FALSE
is_degree(c(-0.1, 0.5))               # FALSE
is_degree(numeric(0))                 # FALSE
```

`is_nugget` *Test whether an object is a nugget*

Description

Check if the given object is a nugget, i.e. an object created by `nugget()`. If a `flavour` is specified, the function returns `TRUE` only if the object is a nugget of the given flavour.

Technically, nuggets are implemented as S3 objects. An object is considered a nugget if it inherits from the S3 class `"nugget"`. It is a nugget of a given flavour if it inherits from both the specified `flavour` class and the `"nugget"` class.

Usage

```
is_nugget(x, flavour = NULL)
```

Arguments

`x`	An object to be tested.
`flavour`	Optional character string specifying the required flavour of the nugget. If `NULL` (default), the function checks only whether `x` is a nugget of any flavour.

Value

A logical scalar: `TRUE` if `x` is a nugget (and of the specified flavour, if given), otherwise `FALSE`.

Author(s)

Michal Burda

See Also

`nugget()`

Examples

```
d <- partition(mtcars, .breaks = 2)
rules <- dig_associations(d, min_support = 0.3)
is_nugget(rules)
is_nugget(rules, "associations")
is_nugget(mtcars)
```

`is_subset` *Determine whether one vector is a subset of another*

Description

Check if all elements of `x` are also contained in `y`. This is equivalent to testing whether `setdiff(x, y)` is empty.

Usage

```
is_subset(x, y)
```

Arguments

`x`	The first vector.
`y`	The second vector.

Details

- If `x` is empty, the result is always `TRUE` (the empty set is a subset of any set).
- If `y` is empty and `x` is not, the result is `FALSE`.
- Duplicates in `x` are ignored; only set membership is tested.
- `NA` values are treated as ordinary elements. In particular, `NA` in `x` is considered a subset element only if `NA` is also present in `y`.

Value

A logical scalar. Returns `TRUE` if `x` is a subset of `y`, i.e. all elements of `x` are also elements of `y`. Returns `FALSE` otherwise.

Author(s)

Michal Burda

See Also

`generics::setdiff()`, `generics::intersect()`, `generics::union()`

Examples

```
is_subset(1:3, 1:5)               # TRUE
is_subset(c(2, 5), 1:4)           # FALSE
is_subset(numeric(0), 1:5)        # TRUE
is_subset(1:3, numeric(0))        # FALSE
is_subset(c(1, NA), c(1, 2, NA))  # TRUE
is_subset(c(NA), 1:5)             # FALSE
```

nugget *Create a nugget object of a given flavour*

Description

Construct a nugget object, which is an S3 object used to store and represent results (e.g., rules or patterns) in the nuggets framework.

A nugget is technically a tibble (or data frame) that inherits from both the "nugget" class and, optionally, a flavour-specific S3 class. This allows distinguishing different types of nuggets (flavours) while still supporting generic methods for all nuggets.

Usage

```
nugget(x, flavour, call_function, call_data, call_args)
```

Arguments

x	An object with rules or patterns, typically a tibble or data frame. If NULL, it will be converted to an empty tibble.
flavour	A character string specifying the flavour of the nugget, or NULL if no flavour should be assigned. If given, the returned object will inherit from both "nugget" and the specified flavour class.
call_function	A character scalar giving the name of the function that created the nugget. Stored as an attribute for provenance.
call_data	A list containing information about the data that was passed to the function which created the nugget. Stored as an attribute for reproducibility.
call_args	A list of arguments that were passed to the function which created the nugget. Stored as an attribute for reproducibility.

Details

Each nugget stores additional provenance information in attributes:

- "call_function" — the name of the function that created the nugget.
- "call_args" — the list of arguments passed to that function.

These attributes make it possible to reconstruct or track how the nugget was created, which supports reproducibility, transparency, and debugging. For example, one can inspect attr(n, "call_args") to recover the original parameters used to mine the patterns.

Value

A tibble object that is an S3 subclass of "nugget" and, if specified, the given flavour class. The object also contains attributes "call_function" and "call_args" describing its provenance.

Author(s)

Michal Burda

See Also

`is_nugget()`

Examples

```
df <- data.frame(lhs = c("a", "b"), rhs = c("c", "d"))
n <- nugget(df,
            flavour = "rules",
            call_function = "example_function",
            call_data = list(ncol = 2,
                             nrow = 2,
                             colnames = c("lhs", "rhs")),
            call_args = list(data = "mydata"))

inherits(n, "nugget")      # TRUE
inherits(n, "rules")       # TRUE
attr(n, "call_function")   # "dig_example_function"
attr(n, "call_args")       # list(data = "mydata")
```

---

parse_condition *Convert condition strings into lists of predicate vectors*

Description

Parse a character vector of conditions into a list of predicate vectors. Each element of the list corresponds to one condition. A condition is a string of predicates separated by commas and enclosed in curly braces, as produced by `format_condition()`. The function splits each string into its component predicates.

If multiple vectors of conditions are provided via `...`, they are combined element-wise. The result is a single list where each element is formed by merging the predicates from the corresponding elements of all input vectors. If the input vectors differ in length, shorter ones are recycled.

Empty conditions (`"{}"`) are parsed as empty character vectors (`character(0)`).

Usage

```
parse_condition(..., .sort = FALSE)
```

Arguments

`...`	One or more character vectors of conditions to be parsed.
`.sort`	Logical flag indicating whether the predicates in each result should be sorted alphabetically. Defaults to `FALSE`.

Value

A list of character vectors, where each element corresponds to one condition and contains the parsed predicates.

Author(s)

Michal Burda

See Also

`format_condition()`, `is_condition()`, `fire()`

Examples

```
parse_condition(c("{a}", "{x=1, z=2, y=3}", "{}"))

# Merge conditions from multiple vectors element-wise
parse_condition(c("{b}", "{x=1, z=2, y=3}", "{q}", "{}"),
                c("{a}", "{v=10, w=11}",    "{}",  "{r,s,t}"))

# Sorting predicates within each condition
parse_condition("{z,y,x}", .sort = TRUE)
```

`partition` *Convert columns of a data frame to Boolean or fuzzy sets (triangular, trapezoidal, or raised-cosine)*

Description

Transform selected columns of a data frame into either dummy logical variables or membership degrees of fuzzy sets, while leaving all remaining columns unchanged. Each transformed column typically produces multiple new columns in the output.

These transformations are most often used as a preprocessing step before calling `dig()` or one of its derivatives, such as `dig_correlations()`, `dig_paired_baseline_contrasts()`, or `dig_associations()`.

The transformation depends on the column type:

- **logical** column `x` is expanded into two logical columns: `x=TRUE` and `x=FALSE`;
- **factor** column `x` with levels `l1`, `l2`, `l3` becomes three logical columns: `x=l1`, `x=l2`, and `x=l3`;
- **numeric** column `x` is transformed according to `.method`:
  - `.method = "dummy"`: the column is treated as a factor with one level per unique value, then expanded into dummy columns;
  - `.method = "crisp"`: the column is discretized into intervals (defined by `.breaks`, `.style`, and `.style_params`) and expanded into dummy columns representing those intervals;
  - `.method = "triangle"` or `.method = "raisedcos"`: the column is converted into one or more fuzzy sets, each represented by membership degrees in $[0, 1]$ (triangular or raised-cosine shaped).

Details of numeric transformations are controlled by `.breaks`, `.labels`, `.style`, `.style_params`, `.right`, `.span`, and `.inc`.

Usage

```
partition(
  .data,
  .what = everything(),
  ...,
  .breaks = NULL,
  .labels = NULL,
  .na = TRUE,
  .keep = FALSE,
  .method = "crisp",
  .style = "equal",
  .style_params = list(),
  .right = TRUE,
  .span = 1,
  .inc = 1
)
```

Arguments

`.data`	A data frame to be processed.
`.what`	A tidyselect expression (see tidyselect syntax) selecting the columns to transform.
`...`	Additional tidyselect expressions selecting more columns.
`.breaks`	Ignored if `.method = "dummy"`. For other methods, either an integer (number of intervals/sets) or a numeric vector of breakpoints.
`.labels`	Optional character vector with labels used for new column names. If `NULL`, labels are generated automatically.
`.na`	If `TRUE`, adds an extra logical column for each source column containing `NA` values (e.g., `x=NA`).
`.keep`	If `TRUE`, keep original columns in the output.
`.method`	Transformation method for numeric columns: `"dummy"`, `"crisp"`, `"triangle"`, or `"raisedcos"`.
`.style`	Controls how breakpoints are determined when `.breaks` is an integer. Values correspond to methods in `classInt::classIntervals()`, e.g., `"equal"`, `"quantile"`, `"kmeans"`, `"sd"`, `"hclust"`, `"bclust"`, `"fisher"`, `"jenks"`, `"dpih"`, `"headtails"`, `"maximum"`, `"box"`. Defaults to `"equal"`. Used only if `.method = "crisp"` and `.breaks` is a single integer.
`.style_params`	A named list of parameters passed to the interval computation method specified by `.style`. Used only if `.method = "crisp"` and `.breaks` is an integer.
`.right`	For `"crisp"`, whether intervals are right-closed and left-open (`TRUE`), or left-closed and right-open (`FALSE`).

`.span`	Number of consecutive breaks forming a set. For `"crisp"`, controls interval width. For `"triangle"`/`"raisedcos"`, `.span = 1` produces triangular sets, `.span = 2` trapezoidal sets.
`.inc`	Step size for shifting breaks when generating successive sets. With `.inc = 1`, all possible sets are created; larger values skip sets.

Details

- Crisp partitioning is efficient and works well when attributes have distinct categories or clear boundaries.
- Fuzzy partitioning is recommended for modeling gradual changes or uncertainty, allowing smooth category transitions at a higher computational cost.

Value

A tibble with `.data` transformed into Boolean or fuzzy predicates.

Crisp transformation of numeric data

For `.method = "crisp"`, numeric columns are discretized into a set of dummy logical variables, each representing one interval of values.

- If `.breaks` is an integer, it specifies the number of intervals into which the column should be divided. The intervals are determined using the `.style` and `.style_params` arguments, allowing not only equal-width but also data-driven breakpoints (e.g., quantile or k-means based). The first and last intervals automatically extend to infinity.
- If `.breaks` is a numeric vector, it specifies interval boundaries directly. Infinite values are allowed.

The `.style` argument defines *how* breakpoints are computed when `.breaks` is an integer. Supported methods (from `classInt::classIntervals()`) include:

- `"equal"` – equal-width intervals across the column range (default);
- `"quantile"` – equal-frequency intervals (see `quantile()` for additional parameters that may be passed through `.style_params`; note that the probs parameter is set automatically and should not be included in `.style_params`);
- `"kmeans"` – intervals found by 1D k-means clustering (see `kmeans()` for additional parameters);
- `"sd"` – intervals based on standard deviations from the mean;
- `"hclust"` – hierarchical clustering intervals (see `hclust()` for additional parameters);
- `"bclust"` – model-based clustering intervals (see `e1071::bclust()` for additional parameters);
- `"fisher"` / `"jenks"` – Fisher–Jenks optimal partitioning;
- `"dpih"` – kernel-based density partitioning (see `KernSmooth::dpih()` for additional parameters);
- `"headtails"` – head/tails natural breaks;
- `"maximum"` – maximization-based partitioning;

- "box" – breaks at boxplot hinges.

Additional parameters for these methods can be passed through .style_params, which should be a named list of arguments accepted by the respective algorithm in classInt::classIntervals(). For example, when .style = "kmeans", one can specify .style_params = list(algorithm = "Lloyd") to request Lloyd's algorithm for k-means clustering.

With .span = 1 and .inc = 1, the generated intervals are consecutive and non-overlapping. For example, with .breaks = c(1, 3, 5, 7, 9, 11) and .right = TRUE, the intervals are $(1; 3]$, $(3; 5]$, $(5; 7]$, $(7; 9]$, and $(9; 11]$. If .right = FALSE, the intervals are left-closed: $[1; 3)$, $[3; 5)$, etc.

Larger .span values produce overlapping intervals. For example, with .span = 2, .inc = 1, and .right = TRUE, intervals are $(1; 5]$, $(3; 7]$, $(5; 9]$, $(7; 11]$.

The .inc argument controls how far the window shifts along .breaks.

- .span = 1, .inc = 2 $\rightarrow$ $(1; 3]$, $(5; 7]$, $(9; 11]$.
- .span = 2, .inc = 3 $\rightarrow$ $(1; 5]$, $(9; 11]$.

Fuzzy transformation of numeric data

For .method = "triangle" or .method = "raisedcos", numeric columns are converted into fuzzy membership degrees in $[0, 1]$.

- If .breaks is an integer, it specifies the number of fuzzy sets.
- If .breaks is a numeric vector, it directly defines fuzzy set boundaries. Infinite values produce open-ended sets.

With .span = 1, each fuzzy set is defined by three consecutive breaks: membership is 0 outside the outer breaks, rises to 1 at the middle break, then decreases back to 0 — yielding triangular or raised-cosine sets.

With .span > 1, fuzzy sets use four consecutive breaks: membership increases between the first two, remains 1 between the middle two, and decreases between the last two — creating trapezoidal sets. Border shapes are linear for .method = "triangle" and cosine for .method = "raisedcos".

The .inc argument defines the step between break windows:

- .span = 1, .inc = 1 $\rightarrow$ $(1; 3; 5)$, $(3; 5; 7)$, $(5; 7; 9)$, $(7; 9; 11)$.
- .span = 2, .inc = 1 $\rightarrow$ $(1; 3; 5; 7)$, $(3; 5; 7; 9)$, $(5; 7; 9; 11)$.
- .span = 1, .inc = 3 $\rightarrow$ $(1; 3; 5)$, $(7; 9; 11)$.

Author(s)

Michal Burda

Examples

```
# Crisp transformation using equal-width bins
partition(CO2, conc, .method = "crisp", .breaks = 4)

# Crisp transformation using quantile-based bins
partition(CO2, conc, .method = "crisp", .breaks = 4, .style = "quantile")
```

```
# Crisp transformation using k-means clustering for breakpoints
partition(CO2, conc, .method = "crisp", .breaks = 4, .style = "kmeans")

# Crisp transformation using Lloyd algorithm for k-means clustering for breakpoints
partition(CO2, conc, .method = "crisp", .breaks = 4, .style = "kmeans",
          .style_params = list(algorithm = "Lloyd"))

# Fuzzy triangular transformation (default)
partition(CO2, conc:uptake, .method = "triangle", .breaks = 3)

# Raised-cosine fuzzy sets
partition(CO2, conc:uptake, .method = "raisedcos", .breaks = 3)

# Overlapping trapezoidal fuzzy sets (Ruspini condition)
partition(CO2, conc:uptake, .method = "triangle", .breaks = 3,
          .span = 2, .inc = 2)

# Different settings per column
CO2 |>
  partition(Plant:Treatment) |>
  partition(conc,
            .method = "raisedcos",
            .breaks = c(-Inf, 95, 175, 350, 675, 1000, Inf)) |>
  partition(uptake,
            .method = "triangle",
            .breaks = c(-Inf, 7.7, 28.3, 45.5, Inf),
            .labels = c("low", "medium", "high"))
```

`remove_almost_constant` *Remove almost constant columns from a data frame*

Description

Test all columns specified by `.what` and remove those that are almost constant. A column is considered almost constant if the proportion of its most frequent value is greater than or equal to the threshold specified by `.threshold`. See `is_almost_constant()` for further details.

Usage

```
remove_almost_constant(
  .data,
  .what = everything(),
  ...,
  .threshold = 1,
  .na_rm = FALSE,
  .verbose = FALSE
)
```

Arguments

.data	A data frame.
.what	A tidyselect expression (see tidyselect syntax) specifying the columns to process.
...	Additional tidyselect expressions selecting more columns.
.threshold	Numeric scalar in the interval $[0, 1]$ giving the minimum required proportion of the most frequent value for a column to be considered almost constant.
.na_rm	Logical; if TRUE, NA values are removed before computing proportions. If FALSE, NA is treated as a regular value. See is_almost_constant() for details.
.verbose	Logical; if TRUE, print a message listing the removed columns.

Value

A data frame with all selected columns removed that meet the definition of being almost constant.

Author(s)

Michal Burda

See Also

is_almost_constant(), remove_ill_conditions()

Examples

```
d <- data.frame(a1 = 1:10,
                a2 = c(1:9, NA),
                b1 = "b",
                b2 = NA,
                c1 = rep(c(TRUE, FALSE), 5),
                c2 = rep(c(TRUE, NA), 5),
                d  = c(rep(TRUE, 4), rep(FALSE, 4), NA, NA))

# Remove columns that are constant (threshold = 1)
remove_almost_constant(d, .threshold = 1.0, .na_rm = FALSE)
remove_almost_constant(d, .threshold = 1.0, .na_rm = TRUE)

# Remove columns where the majority value occurs in >= 50% of rows
remove_almost_constant(d, .threshold = 0.5, .na_rm = FALSE)
remove_almost_constant(d, .threshold = 0.5, .na_rm = TRUE)

# Restrict check to a subset of columns
remove_almost_constant(d, a1:b2, .threshold = 0.5, .na_rm = TRUE)
```

`remove_ill_conditions` *Remove invalid conditions from a list*

Description

From a given list of character vectors, remove those elements that are not valid conditions.

A valid condition is a character vector of predicates, where each predicate corresponds to a column name in the supplied data frame or matrix. Empty character vectors and `NULL` elements are also considered valid conditions.

Usage

```
remove_ill_conditions(x, data)
```

Arguments

`x`	A list of character vectors, each representing a condition.
`data`	A matrix or data frame whose column names define valid predicates.

Details

This function acts as a simple filter around `is_condition()`. It checks each element of `x` against the column names of `data` and removes those that contain invalid predicates. The result preserves only valid conditions and discards the invalid ones.

Value

A list containing only those elements of `x` that are valid conditions.

Author(s)

Michal Burda

See Also

`is_condition()`

Examples

```
d <- data.frame(foo = 1:5, bar = 1:5, blah = 1:5)

conds <- list(c("foo", "bar"), "blah", "invalid", character(0), NULL)
remove_ill_conditions(conds, d)
# keeps "foo","bar"; "blah"; empty; NULL
```

shorten_condition *Shorten predicates within conditions*

Description

This function takes a character vector of conditions and shortens the predicates within each condition according to a specified method.

Each element of `x` must be a condition formatted as a string, e.g. `"{a=1,b=100,c=3}"` (see `format_condition()`). The function then shortens the predicates in each condition based on the selected `method`:

- `"letters"`: predicates are replaced with single letters from the English alphabet, starting with `A` for the first distinct predicate;
- `"abbrev4"`: predicates are abbreviated to at most 4 characters using `base::abbreviate()`;
- `"abbrev8"`: predicates are abbreviated to at most 8 characters using `base::abbreviate()`;
- `"none"`: no shortening is applied; predicates remain unchanged.

Usage

```
shorten_condition(x, method = "letters")
```

Arguments

`x`	A character vector of conditions, each formatted as a string (e.g., `"{a=1,b=100,c=3}"`).
`method`	A character scalar specifying the shortening method. Must be one of `"letters"`, `"abbrev4"`, `"abbrev8"`, or `"none"`. Defaults to `"letters"`.

Details

Predicate shortening is useful for visualization or reporting, especially when original predicate names are long or complex. Note that shortening is applied consistently across all conditions in `x`.

Value

A character vector of conditions with predicates shortened according to the specified method.

Author(s)

Michal Burda

See Also

`format_condition()`, `parse_condition()`, `is_condition()`, `remove_ill_conditions()`, `base::abbreviate()`

Examples

```
shorten_condition(c("{a=1,b=100,c=3}", "{a=2}", "{b=100,c=3}"),
                  method = "letters")

shorten_condition(c("{helloWorld=1}", "{helloWorld=2}", "{c=3,helloWorld=1}"),
                  method = "abbrev4")

shorten_condition(c("{helloWorld=1}", "{helloWorld=2}", "{c=3,helloWorld=1}"),
                  method = "abbrev8")

shorten_condition(c("{helloWorld=1}", "{helloWorld=2}"),
                  method = "none")
```

---

values *Extract values from predicate names*

Description

This function extracts the value part from a character vector of predicate names. Each element of `x` is expected to follow the pattern `<varname>=<value>`, where `<varname>` is a variable name and `<value>` is the associated value.

If an element does not contain an equal sign (`=`), the function returns an empty string for that element.

Usage

```
values(x)
```

Arguments

`x`	A character vector of predicate names.

Details

This function is the counterpart to `var_names()`, which extracts the variable part of predicates. Together, `var_names()` and `values()` provide a convenient way to split predicate strings into their variable and value components.

Value

A character vector containing the `<value>` parts of predicate names in `x`. Elements without an equal sign return an empty string. If `x` is `NULL`, the function returns `NULL`. If `x` is an empty vector (`character(0)`), the function returns an empty vector (`character(0)`).

Author(s)

Michal Burda

See Also

var_names()

Examples

```
values(c("a=1", "a=2", "b=x", "b=y"))
# returns c("1", "2", "x", "y")

values(c("a", "b=3"))
# returns c("", "3")
```

---

var_grid *Create a tibble of combinations of selected column names*

Description

The xvars and yvars arguments are tidyselect expressions (see tidyselect syntax) that specify the columns of x whose names will be used to form combinations.

If yvars is NULL, the function creates a tibble with one column, var, enumerating all column names selected by the xvars expression.

If yvars is not NULL, the function creates a tibble with two columns, xvar and yvar, whose rows enumerate all combinations of column names specified by xvars and yvars.

It is allowed to specify the same column in both xvars and yvars. In such cases, self-combinations (a column paired with itself) are removed from the result.

In other words, the function creates a grid of all possible pairs (xx, yy) where $xx \in xvars$, $yy \in yvars$, and $xx \neq yy$.

Usage

```
var_grid(
  x,
  xvars = everything(),
  yvars = everything(),
  allow = "all",
  disjoint = var_names(colnames(x)),
  xvar_name = if (quo_is_null(enquo(yvars))) "var" else "xvar",
  yvar_name = "yvar",
 error_context = list(arg_x = "x", arg_xvars = "xvars", arg_yvars = "yvars", arg_allow =
   "allow", arg_disjoint = "disjoint", arg_xvar_name = "xvar_name", arg_yvar_name =
    "yvar_name", call = current_env())
)
```

Arguments

x	A data frame or matrix.
xvars	A tidyselect expression specifying the columns of x whose names will be used in the first position (xvar) of the combinations.
yvars	NULL or a tidyselect expression specifying the columns of x whose names will be used in the second position (yvar) of the combinations.
allow	A character string specifying which columns may be selected by xvars and yvars. Possible values are: • "all" – all columns may be selected; • "numeric" – only numeric columns may be selected.
disjoint	An atomic vector of length equal to the number of columns in x that specifies disjoint groups of predicates. Columns belonging to the same group (i.e. having the same value in disjoint) will not appear together in a single combination of xvars and yvars. Ignored if yvars is NULL.
xvar_name	A character string specifying the name of the first column (xvar) in the output tibble.
yvar_name	A character string specifying the name of the second column (yvar) in the output tibble. This column is omitted if yvars is NULL.
error_context	A list providing details for error messages. This is useful when var_grid() is called from another function, allowing error messages to reference the caller's argument names. The list must contain: • arg_x – name of the argument x; • arg_xvars – name of the argument xvars; • arg_yvars – name of the argument yvars; • arg_allow – name of the argument allow; • arg_xvar_name – name of the xvar column in the output; • arg_yvar_name – name of the yvar column in the output; • call – the calling environment for evaluating error messages.

Details

var_grid() is typically used when a function requires a systematic list of variables or variable pairs to analyze. For example, it can be used to generate all pairs of variables for correlation, association, or contrast analysis. The flexibility of xvars and yvars makes it possible to restrict the grid to specific subsets of variables while ensuring that invalid or redundant combinations (e.g., self-pairs or disjoint groups) are excluded automatically.

The allow argument can be used to restrict the selection of columns to numeric columns only. This is useful when the resulting variable combinations will be used in analyses that require numeric data, such as correlation or contrast tests.

The disjoint argument allows specifying groups of columns that should not appear together in a single combination. This is useful when certain columns represent mutually exclusive categories or measurements that should not be analyzed together. For example, if disjoint groups columns by measurement type, the function will ensure that no combination includes two columns from the same type.

Value

If yvars is NULL, a tibble with a single column (var). If yvars is not NULL, a tibble with two columns (xvar, yvar) enumerating all valid combinations of column names selected by xvars and yvars. The order of variables in the result follows the order in which they are selected by xvars and yvars.

Author(s)

Michal Burda

Examples

```
# Grid of all pairwise column combinations in CO2
var_grid(CO2)

# Grid of combinations where the first column is Plant, Type, or Treatment,
# and the second column is conc or uptake
var_grid(CO2, xvars = Plant:Treatment, yvars = conc:uptake)

# Prevent variables from the same disjoint group from being paired together
d <- data.frame(a = 1:5, b = 6:10, c = 11:15, d = 16:20)
# Group (a, b) together and (c, d) together
var_grid(d, xvars = everything(), yvars = everything(),
         disjoint = c(1, 1, 2, 2))
```

---

var_names *Extract variable names from predicate names*

Description

This function extracts the variable part from a character vector of predicate names. Each element of x is expected to follow the pattern <varname>=<value>, where <varname> is a variable name and <value> is the associated value.

If an element does not contain an equal sign (=), the entire string is returned unchanged.

Usage

```
var_names(x)
```

Arguments

x	A character vector of predicate names.

Details

This function is the counterpart to values(), which extracts the value part of predicates. Together, var_names() and values() provide a convenient way to split predicate strings into their variable and value components.

Value

A character vector containing the <varname> parts of predicate names in x. If an element does not contain =, the entire string is returned as is. If x is NULL, the function returns NULL. If x has length zero (character(0)), the function returns character(0).

Author(s)

Michal Burda

See Also

values()

Examples

```
var_names(c("a=1", "a=2", "b=x", "b=y"))
# returns c("a", "a", "b", "b")

var_names(c("a", "b=3"))
# returns c("a", "b")

var_names(character(0))
# returns character(0)

var_names(NULL)
# returns character(0)
```

---

which_antichain *Return indices of first elements of the list, which are incomparable with preceding elements.*

Description

The function returns indices of elements from the given list x, which are incomparable (i.e., it is neither subset nor superset) with any preceding element. The first element is always selected. The next element is selected only if it is incomparable with all previously selected elements.

Usage

```
which_antichain(x, distance = 0)
```

Arguments

x	a list of integerish vectors
distance	a non-negative integer, which specifies the allowed discrepancy between compared sets

Value

an integer vector of indices of selected (incomparable) elements.

Author(s)

Michal Burda

Examples

```
# Create a list of integerish vectors
x <- list(c(1, 2), c(1, 2, 3), c(2, 3), c(1, 3), c(4, 5))

# Find incomparable elements
which_antichain(x)
```

Index