topics documented:

Author  Indrajeet Patil [aut] (<https://orcid.org/0000-0003-1995-6531>,
@patilindrajeet),
Etienne Bacher [aut, cre] (<https://orcid.org/0000-0002-9271-5075>),
Dominique Makowski [aut] (<https://orcid.org/0000-0001-5375-9967>,
@Dom_Makowski),
Daniel Lüdecke [aut] (<https://orcid.org/0000-0002-8895-3206>,
@strengejacke),
Mattan S. Ben-Shachar [aut] (<https://orcid.org/0000-0002-4287-4801>),
Brenton M. Wiernik [aut] (<https://orcid.org/0000-0001-9560-6336>,
@bmwiernik),
Rémi Thériault [ctb] (<https://orcid.org/0000-0003-4315-6788>,
@rempsyc),
Thomas J. Faulkenberry [rev],
Robert Garrett [rev]

Repository  CRAN

Date/Publication  2023-06-16 21:40:03 UTC

R topics documented:

adjust ................................................................. 3
assign_labels ..................................................... 6
categorize .......................................................... 9
center ................................................................. 13
coeff_var ............................................................ 17
goce_to_numeric .................................................. 18
convert_na_to ....................................................... 19
convert_to_na ...................................................... 22
data_addprefix .................................................... 24
data_arrange ....................................................... 27
data_codebook ..................................................... 28
data_duplicated ................................................... 31
data_extract ....................................................... 33
data_group .......................................................... 35
data_match .......................................................... 37
data_merge .......................................................... 40
data_modify .......................................................... 43
data_partition ..................................................... 45
datapeek .............................................................. 47
data_read .............................................................. 49
data_relocate ....................................................... 51
data_restoretype ................................................... 54
data_rotate ........................................................... 55
data_separate ....................................................... 56
data_tabulate ....................................................... 60
data_to_long ......................................................... 63
data_to_wide ......................................................... 66
data_unique .......................................................... 69
adjust

Adjust data for the effect of other variable(s)

Description

This function can be used to adjust the data for the effect of other variables present in the dataset. It is based on an underlying fitting of regressions models, allowing for quite some flexibility, such as including factors as random effects in mixed models (multilevel partialization), continuous variables as smooth terms in general additive models (non-linear partialization) and/or fitting these models under a Bayesian framework. The values returned by this function are the residuals of the regression models. Note that a regular correlation between two "adjusted" variables is equivalent to the partial correlation between them.
adjust

Usage

adjust(
  data,
  effect = NULL,
  select = is.numeric,
  exclude = NULL,
  multilevel = FALSE,
  additive = FALSE,
  bayesian = FALSE,
  keep_intercept = FALSE,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = FALSE
)

data_adjust(
  data,
  effect = NULL,
  select = is.numeric,
  exclude = NULL,
  multilevel = FALSE,
  additive = FALSE,
  bayesian = FALSE,
  keep_intercept = FALSE,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = FALSE
)

Arguments

data
  A data frame.

effect
  Character vector of column names to be adjusted for (regressed out). If NULL
  (the default), all variables will be selected.

select
  Variables that will be included when performing the required tasks. Can be either

  • a variable specified as a literal variable name (e.g., column_name),
  • a string with the variable name (e.g., "column_name"), or a character vector
    of variable names (e.g., c("col1", "col2", "col3")),
  • a formula with variable names (e.g., ~column_1 + column_2),
  • a vector of positive integers, giving the positions counting from the left (e.g.
    1 or c(1, 3, 5)),
  • a vector of negative integers, giving the positions counting from the right
    (e.g., -1 or -1:-3),
  • one of the following select-helpers: starts_with(), ends_with(), contains(),
    a range using : or regex("""). starts_with(), ends_with(), and contains()
    accept several patterns, e.g starts_with("Sep", "Petal").
• or a function testing for logical conditions, e.g. `is.numeric()` (or `is.numeric`), or any user-defined function that selects the variables for which the function returns `TRUE` (like: `foo <- function(x) mean(x) > 3`),

• ranges specified via literal variable names, select-helpers (except `regex()`) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a `-`, e.g. `-ends_with("\""), -is.numeric or -(Sepal.Width:Petal.Length). **Note:** Negation means that matches are **excluded**, and thus, the exclude argument can be used alternatively. For instance, `select=-ends_with("Length")` (with `-`) is equivalent to `exclude=ends_with("Length")` (no `-`). In case negation should not work as expected, use the exclude argument instead.

If `NULL`, selects all columns. Patterns that found no matches are silently ignored, e.g. `find_columns(iris, select = c("Species", "Test"))` will just return "Species".

`exclude` See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If `NULL` (the default), excludes no columns.

`multilevel` If `TRUE`, the factors are included as random factors. Else, if `FALSE` (default), they are included as fixed effects in the simple regression model.

`additive` If `TRUE`, continuous variables as included as smooth terms in additive models. The goal is to regress-out potential non-linear effects.

`bayesian` If `TRUE`, the models are fitted under the Bayesian framework using `rstanarm`.

`keep_intercept` If `FALSE` (default), the intercept of the model is re-added. This avoids the centering around 0 that happens by default when regressing out another variable (see the examples below for a visual representation of this).

`ignore_case` Logical, if `TRUE` and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

`regex` Logical, if `TRUE`, the search pattern from select will be treated as regular expression. When `regex = TRUE`, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. `regex = TRUE` is comparable to using one of the two select-helpers, `select = contains("\"")` or `select = regex("\"\")`, however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.

`verbose` Toggle warnings.

Value

A data frame comparable to `data`, with adjusted variables.

Examples

```r
adjusted_all <- adjust(attitude)
head(adjusted_all)
adjusted_one <- adjust(attitude, effect = "complaints", select = "rating")
```
head(adjusted_one)

adjust(attitude, effect = "complaints", select = "rating", bayesian = TRUE)
adjust(attitude, effect = "complaints", select = "rating", additive = TRUE)
attitude$complaints_LMH <- cut(attitude$complaints, 3)
adjust(attitude, effect = "complaints_LMH", select = "rating", multilevel = TRUE)

# Generate data
data <- simulate_correlation(n = 100, r = 0.7)
data$V2 <- (5 * data$V2) + 20 # Add intercept

# Adjust
adjusted <- adjust(data, effect = "V1", select = "V2")
adjusted_icpt <- adjust(data, effect = "V1", select = "V2", keep_intercept = TRUE)

# Visualize
plot(data$V1, data$V2, pch = 19, col = "blue", ylim = c(min(adjusted$V2), max(data$V2)),
     main = "Original (blue), adjusted (green), and adjusted - intercept kept (red) data")
abline(lm(V2 ~ V1, data = data), col = "blue")
points(adjusted$V1, adjusted$V2, pch = 19, col = "green")
abline(lm(V2 ~ V1, data = adjusted), col = "green")
points(adjusted_icpt$V1, adjusted_icpt$V2, pch = 19, col = "red")
abline(lm(V2 ~ V1, data = adjusted_icpt), col = "red")

---

assign_labels

Assign variable and value labels

Description

Assign variable and values labels to a variable or variables in a data frame. Labels are stored as attributes ("label" for variable labels and "labels") for value labels.

Usage

assign_labels(x, ...)

## S3 method for class 'numeric'
assign_labels(x, variable = NULL, values = NULL, ...)

## S3 method for class 'data.frame'
assign_labels(
x,
    select = NULL,
    exclude = NULL,
Arguments

x
A data frame, factor or vector.

... Currently not used.

variable The variable label as string.

values The value labels as (named) character vector. If values is not a named vector, the length of labels must be equal to the length of unique values. For a named vector, the left-hand side (LHS) is the value in x, the right-hand side (RHS) the associated value label. Non-matching labels are omitted.

select Variables that will be included when performing the required tasks. Can be either

• a variable specified as a literal variable name (e.g., column_name),
• a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
• a formula with variable names (e.g., ~column_1 + column_2),
• a vector of positive integers, giving the positions counting from the left (e.g. 1 or c(1, 3, 5)),
• a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
• one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal").
• or a function testing for logical conditions, e.g. is.numeric() (or is.numeric), or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
• ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. -ends_with(""), -is.numeric or -(Sepal.Width:Petal.Length). Note: Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

exclude See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.
assign_labels

ignore_case Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

regex Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains("") or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see ‘Details’), this argument may be used as workaround.

verbose Toggle warnings.

Value
A labelled variable, or a data frame of labelled variables.

Selection of variables - the select argument

For most functions that have a select argument (including this function), the complete input data frame is returned, even when select only selects a range of variables. That is, the function is only applied to those variables that have a match in select, while all other variables remain unchanged. In other words: for this function, select will not omit any non-included variables, so that the returned data frame will include all variables from the input data frame.

Examples

```r
x <- 1:3
# labelling by providing required number of labels
assign_labels(  
  x,  
  variable = "My x",  
  values = c("one", "two", "three")
)

# labelling using named vectors
data(iris)
out <- assign_labels(  
  iris$Species,  
  variable = "Labelled Species",  
  values = c('setosa' = "Spec1", 'versicolor' = "Spec2", 'virginica' = "Spec3")
)
str(out)

# data frame example
out <- assign_labels(  
  iris,  
  select = "Species",  
  variable = "Labelled Species",  
  values = c('setosa' = "Spec1", 'versicolor' = "Spec2", 'virginica' = "Spec3")
)
str(out$Species)
```
# Partial labelling
x <- 1:5
assign_labels(
  x,
  variable = "My x",
  values = c(1 = "lowest", 5 = "highest")
)

categorize  
Rencode (or "cut" / "bin") data into groups of values.

Description
This function divides the range of variables into intervals and recodes the values inside these intervals according to their related interval. It is basically a wrapper around base R’s cut(), providing a simplified and more accessible way to define the interval breaks (cut-off values).

Usage
categorize(x, ...)

## S3 method for class 'numeric'
categorize(
  x,
  split = "median",
  n_groups = NULL,
  range = NULL,
  lowest = 1,
  labels = NULL,
  verbose = TRUE,
  ...
)

## S3 method for class 'data.frame'
categorize(
  x,
  select = NULL,
  exclude = NULL,
  split = "median",
  n_groups = NULL,
  range = NULL,
  lowest = 1,
  labels = NULL,
  append = FALSE,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = TRUE,
categorize

Arguments

x  A (grouped) data frame, numeric vector or factor.

...  not used.

split  Character vector, indicating at which breaks to split variables, or numeric values with values indicating breaks. If character, may be one of "median", "mean", "quantile", "equal_length", or "equal_range". "median" or "mean" will return dichotomous variables, split at their mean or median, respectively. "quantile" and "equal_length" will split the variable into n_groups groups, where each group refers to an interval of a specific range of values. Thus, the length of each interval will be based on the number of groups. "equal_range" also splits the variable into multiple groups, however, the length of the interval is given, and the number of resulting groups (and hence, the number of breaks) will be determined by how many intervals can be generated, based on the full range of the variable.

n_groups  If split is "quantile" or "equal_length", this defines the number of requested groups (i.e. resulting number of levels or values) for the recoded variable(s). "quantile" will define intervals based on the distribution of the variable, while "equal_length" tries to divide the range of the variable into pieces of equal length.

range  If split = "equal_range", this defines the range of values that are recoded into a new value.

lowest  Minimum value of the recoded variable(s). If NULL (the default), for numeric variables, the minimum of the original input is preserved. For factors, the default minimum is 1. For split = "equal_range", the default minimum is always 1, unless specified otherwise in lowest.

labels  Character vector of value labels. If not NULL, categorize() will returns factors instead of numeric variables, with labels used for labelling the factor levels. Can also be "mean" or "median" for a factor with labels as the mean/median of each groups.

verbose  Toggle warnings.

select  Variables that will be included when performing the required tasks. Can be either

- a variable specified as a literal variable name (e.g., column_name),
- a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
- a formula with variable names (e.g., ~column_1 + column_2),
- a vector of positive integers, giving the positions counting from the left (e.g., 1 or c(1, 3, 5)),
- a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
• one of the following select-helpers: `starts_with()`, `ends_with()`, `contains()`, a range using `or_regex()`, `starts_with()`, `ends_with()`, and `contains()` accept several patterns, e.g. `starts_with("Sep", "Petal")`.
• or a function testing for logical conditions, e.g. `is.numeric()` (or `is.numeric`), or any user-defined function that selects the variables for which the function returns `TRUE` (like: `foo <- function(x) mean(x) > 3`).
• ranges specified via literal variable names, select-helpers (except `regex()`) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a `-`, e.g. `-ends_with(""), -is.numeric or -(Sepal.Width:Petal.Length). **Note:** Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, `select=-ends_with("Length") (with -)` is equivalent to `exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. `find_columns(iris, select = c("Species", "Test"))` will just return "Species".

**exclude**  
See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

**append**  
Logical or string. If TRUE, recoded or converted variables get new column names and are appended (column bind) to `x`, thus returning both the original and the recoded variables. The new columns get a suffix, based on the calling function: 
"_r" for recode functions, "_n" for `to_numeric()`, "_f" for `to_factor()`, or "_s" for `slide()`. If `append=FALSE`, original variables in `x` will be overwritten by their recoded versions. If a character value, recoded variables are appended with new column names (using the defined suffix) to the original data frame.

**ignore_case**  
Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

**regex**  
Logical, if TRUE, the search pattern from select will be treated as regular expression. When `regex = TRUE`, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. `regex = TRUE` is comparable to using one of the two select-helpers, `select = contains("")` or `select = regex("")`, however, since the select-helpers may not work when called from inside other functions (see ‘Details’), this argument may be used as workaround.

**Value**  
`x`, recoded into groups. By default `x` is numeric, unless `labels` is specified. In this case, a factor is returned, where the factor levels (i.e. recoded groups are labelled accordingly.

**Splits and breaks (cut-off values)**  
Breaks are in general **exclusive**, this means that these values indicate the lower bound of the next group or interval to begin. Take a simple example, a numeric variable with values from 1 to 9. The median would be 5, thus the first interval ranges from 1-4 and is recoded into 1, while 5-9
would turn into 2 (compare `cbind(1:9, categorize(1:9))}). The same variable, using `split = "quantile"` and `n_groups = 3` would define breaks at 3.67 and 6.33 (see `quantile(1:9, probs = c(1/3, 2/3))`), which means that values from 1 to 3 belong to the first interval and are recoded into 1 (because the next interval starts at 3.67), 4 to 6 into 2 and 7 to 9 into 3.

**Recoding into groups with equal size or range**

`split = "equal_length"` and `split = "equal_range"` try to divide the range of x into intervals of similar (or same) length. The difference is that `split = "equal_length"` will divide the range of x into `n_groups` pieces and thereby defining the intervals used as breaks (hence, it is equivalent to `cut(x, breaks = n_groups)`), while `split = "equal_range"` will cut x into intervals that all have the length of range, where the first interval by defaults starts at 1. The lowest (or starting) value of that interval can be defined using the `lowest` argument.

**Selection of variables - the `select` argument**

For most functions that have a `select` argument (including this function), the complete input data frame is returned, even when `select` only selects a range of variables. That is, the function is only applied to those variables that have a match in `select`, while all other variables remain unchanged. In other words: for this function, `select` will not omit any non-included variables, so that the returned data frame will include all variables from the input data frame.

**See Also**

- Functions to rename stuff: `data_rename()`, `data_rename_rows()`, `data_addprefix()`, `data_addsuffix()`
- Functions to reorder or remove columns: `data_reorder()`, `data_relocate()`, `data_remove()`
- Functions to reshape, pivot or rotate data frames: `data_to_long()`, `data_to_wide()`, `data_rotate()`
- Functions to recode data: `rescale()`, `reverse()`, `categorize()`, `recode_values()`, `slide()`
- Functions to standardize, normalize, rank-transform: `center()`, `standardize()`, `normalize()`, `ranktransform()`, `winsorize()`
- Split and merge data frames: `data_partition()`, `data_merge()`
- Functions to find or select columns: `data_select()`, `data_find()`
- Functions to filter rows: `data_match()`, `data_filter()`

**Examples**

```r
set.seed(123)
x <- sample(1:10, size = 50, replace = TRUE)
table(x)

# by default, at median
table(categorize(x))

# into 3 groups, based on distribution (quantiles)
table(categorize(x, split = "quantile", n_groups = 3))

# into 3 groups, user-defined break
```
table(categorize(x, split = c(3, 5)))

set.seed(123)
x <- sample(1:100, size = 500, replace = TRUE)

# into 5 groups, try to recode into intervals of similar length,
# i.e. the range within groups is the same for all groups
table(categorize(x, split = "equal_length", n_groups = 5))

# into 5 groups, try to return same range within groups
# i.e. 1-20, 21-40, 41-60, etc. Since the range of "x" is
# 1-100, and we have a range of 20, this results into 5
# groups, and thus is for this particular case identical
# to the previous result.
table(categorize(x, split = "equal_range", range = 20))

# return factor with value labels instead of numeric value
set.seed(123)
x <- sample(1:10, size = 30, replace = TRUE)
categorize(x, "equal_length", n_groups = 3)
categorize(x, "equal_length", n_groups = 3, labels = c("low", "mid", "high"))

# cut numeric into groups with the mean or median as a label name
x <- sample(1:10, size = 30, replace = TRUE)
categorize(x, "equal_length", n_groups = 3, labels = "mean")
categorize(x, "equal_length", n_groups = 3, labels = "median")

---

center

Centering (Grand-Mean Centering)

Description

Performs a grand-mean centering of data.

Usage

center(x, ...)

centre(x, ...)

## S3 method for class 'numeric'
center(
  x,
  robust = FALSE,
  weights = NULL,
  reference = NULL,
  center = NULL,
  verbose = TRUE,
  ...
)
## S3 method for class 'data.frame'
center(
  x,
  select = NULL,
  exclude = NULL,
  robust = FALSE,
  weights = NULL,
  reference = NULL,
  center = NULL,
  force = FALSE,
  remove_na = c("none", "selected", "all"),
  append = FALSE,
  ignore_case = FALSE,
  verbose = TRUE,
  regex = FALSE,
  ...
)

### Arguments

- **x**: A (grouped) data frame, a (numeric or character) vector or a factor.
- **...**: Currently not used.
- **robust**: Logical, if TRUE, centering is done by subtracting the median from the variables. If FALSE, variables are centered by subtracting the mean.
- **weights**: Can be NULL (for no weighting), or:
  - For data frames: a numeric vector of weights, or a character of the name of a column in the data.frame that contains the weights.
  - For numeric vectors: a numeric vector of weights.
- **reference**: A data frame or variable from which the centrality and deviation will be computed instead of from the input variable. Useful for standardizing a subset or new data according to another data frame.
- **center**: Numeric value, which can be used as alternative to reference to define a reference centrality. If center is of length 1, it will be recycled to match the length of selected variables for centering. Else, center must be of same length as the number of selected variables. Values in center will be matched to selected variables in the provided order, unless a named vector is given. In this case, names are matched against the names of the selected variables.
- **verbose**: Toggle warnings and messages.
- **select**: Variables that will be included when performing the required tasks. Can be either
  - a variable specified as a literal variable name (e.g., column_name),
  - a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
  - a formula with variable names (e.g., ~column_1 + column_2),
  - a character vector of variable names (e.g., c("col1", "col2", "col3"),
  - a string with the variable name (e.g., "column_name"), or a character vector
- a vector of positive integers, giving the positions counting from the left (e.g. 1 or c(1, 3, 5)),
- a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
- one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""). starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal"),
- or a function testing for logical conditions, e.g. is.numeric() (or is.numeric), or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
- ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. -ends_with(""), -is.numeric or -(Sepal.Width:Petal.Length). **Note:** Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

**exclude**
See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

**force**
Logical, if TRUE, forces centering of factors as well. Factors are converted to numerical values, with the lowest level being the value 1 (unless the factor has numeric levels, which are converted to the corresponding numeric value).

**remove_na**
How should missing values (NA) be treated: if "none" (default): each column’s standardization is done separately, ignoring NAs. Else, rows with NA in the columns selected with select / exclude ("selected") or in all columns ("all") are dropped before standardization, and the resulting data frame does not include these cases.

**append**
Logical or string. If TRUE, centered variables get new column names (with the suffix "_c") and are appended (column bind) to x, thus returning both the original and the centered variables. If FALSE, original variables in x will be overwritten by their centered versions. If a character value, centered variables are appended with new column names (using the defined suffix) to the original data frame.

**ignore_case**
Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

**regex**
Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains("") or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see ‘Details’), this argument may be used as workaround.
Value

The centered variables.

Selection of variables - the select argument

For most functions that have a select argument (including this function), the complete input data frame is returned, even when select only selects a range of variables. That is, the function is only applied to those variables that have a match in select, while all other variables remain unchanged. In other words: for this function, select will not omit any non-included variables, so that the returned data frame will include all variables from the input data frame.

Note

Difference between centering and standardizing: Standardized variables are computed by subtracting the mean of the variable and then dividing it by the standard deviation, while centering variables involves only the subtraction.

See Also

If centering within-clusters (instead of grand-mean centering) is required, see demean(). For standardizing, see standardize(), and makepredictcall.dw_transformer() for use in model formulas.

Examples

data(iris)

# entire data frame or a vector
head(iris$Sepal.Width)
head(center(iris$Sepal.Width))
head(center(iris))
head(center(iris, force = TRUE))

# only the selected columns from a data frame
center(anscombe, select = c("x1", "x3"))
center(anscombe, exclude = c("x1", "x3"))

# centering with reference center and scale
d <- data.frame(
  a = c(-2, -1, 0, 1, 2),
  b = c(3, 4, 5, 6, 7)
)

# default centering at mean
center(d)

# centering, using 0 as mean
center(d, center = 0)

# centering, using -5 as mean
center(d, center = -5)
Compute the coefficient of variation

Description

Compute the coefficient of variation (CV, ratio of the standard deviation to the mean, $\sigma/\mu$) for a set of numeric values.

Usage

\[
\text{coef_var}(x, \ldots)
\]

\[
distribution\_coef\_var(x, \ldots)
\]

## S3 method for class 'numeric'
\[
\text{coef_var}(x, \ldots)
\]

Arguments

- **x**: A numeric vector of ratio scale (see details), or vector of values that can be coerced to one.
- **\ldots**: Further arguments passed to computation functions.
- **mu**: A numeric vector of mean values to use to compute the coefficient of variation. If supplied, x is not used to compute the mean.
- **sigma**: A numeric vector of standard deviation values to use to compute the coefficient of variation. If supplied, x is not used to compute the SD.
- **method**: Method to use to compute the CV. Can be "standard" to compute by dividing the standard deviation by the mean, "unbiased" for the unbiased estimator for normally distributed data, or one of two robust alternatives: "median_mad" to divide the median by the `stats::mad()` function, or "qcd" (quartile coefficient of dispersion, interquartile range divided by the sum of the quartiles [twice the midhinge]: $(Q_3 - Q_1)/(Q_3 + Q_1)$).
- **trim**: The fraction (0 to 0.5) of values to be trimmed from each end of x before the mean and standard deviation (or other measures) are computed. Values of trim outside the range of (0 to 0.5) are taken as the nearest endpoint.
Details

CV is only applicable of values taken on a ratio scale: values that have a fixed meaningfully defined 0 (which is either the lowest or highest possible value), and that ratios between them are interpretable. For example, how many sandwiches have I eaten this week? 0 means "none" and 20 sandwiches is 4 times more than 5 sandwiches. If I were to center the number of sandwiches, it will no longer be on a ratio scale (0 is no "none" it is the mean, and the ratio between 4 and -2 is not meaningful). Scaling a ratio scale still results in a ratio scale. So I can redefine "how many half sandwiches did I eat this week ( = sandwiches * 0.5) and 0 would still mean "none", and 20 half-sandwiches is still 4 times more than 5 half-sandwiches.

This means that CV is NOT invariant to shifting, but it is to scaling:

```r
sandwiches <- c(0, 4, 15, 0, 5, 2, 7)
coef_var(sandwiches)
#> [1] 1.239094
c coef_var(sandwiches / 2) # same
#> [1] 1.239094
c coef_var(sandwiches + 4) # different! 0 is no longer meaningful!
#> [1] 0.6290784
```

Value

The computed coefficient of variation for x.

Examples

```r
c coef_var(1:10)
c coef_var(c(1:10, 100), method = "median_mad")
c coef_var(c(1:10, 100), method = "qcd")
c coef_var(mu = 10, sigma = 20)
c coef_var(mu = 10, sigma = 20, method = "unbiased", n = 30)
```

---

**coerce_to_numeric**

*Convert to Numeric (if possible)*

**Description**

Tries to convert vector to numeric if possible (if no warnings or errors). Otherwise, leaves it as is.
**Usage**

coerce_to_numeric(x)

**Arguments**

x  A vector to be converted.

**Value**

Numeric vector (if possible)

**Examples**

c coerce_to_numeric(c("1", "2"))
c coerce_to_numeric(c("1", "2", "A"))

---

**Description**

Replace missing values in a variable or a data frame.

**Usage**

convert_na_to(x, ...)

```r
## S3 method for class 'numeric'
convert_na_to(x, replacement = NULL, verbose = TRUE, ...)

## S3 method for class 'character'
convert_na_to(x, replacement = NULL, verbose = TRUE, ...)

## S3 method for class 'data.frame'
convert_na_to(
  x,
  select = NULL,
  exclude = NULL,
  replacement = NULL,
  replace_num = replacement,
  replace_char = replacement,
  replace_fac = replacement,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = TRUE,
  ...
)
```
Arguments

x A numeric, factor, or character vector, or a data frame.
... Not used.
replacement Numeric or character value that will be used to replace NA.
verbose Toggle warnings.
select Variables that will be included when performing the required tasks. Can be either

• a variable specified as a literal variable name (e.g., column_name),
• a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
• a formula with variable names (e.g., ~column_1 + column_2),
• a vector of positive integers, giving the positions counting from the left (e.g., 1 or c(1, 3, 5)),
• a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
• one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""). starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal").
• or a function testing for logical conditions, e.g. is.numeric() (or is.numeric),
• or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
• ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. -ends_with(""), -is.numeric or -(Sepal.Width:Petal.Length). Note: Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

exclude See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

replace_num Value to replace NA when variable is of type numeric.
replace_char Value to replace NA when variable is of type character.
replace_fac Value to replace NA when variable is of type factor.
ignore_case Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

regex Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported
select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains(""") or select = regex(""""), however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.

Value

x, where NA values are replaced by replacement.

Selection of variables - the select argument

For most functions that have a select argument (including this function), the complete input data frame is returned, even when select only selects a range of variables. That is, the function is only applied to those variables that have a match in select, while all other variables remain unchanged. In other words: for this function, select will not omit any non-included variables, so that the returned data frame will include all variables from the input data frame.

Examples

# Convert NA to 0 in a numeric vector
convert_na_to(
  c(9, 3, NA, 2, 3, 1, NA, 8),
  replacement = 0
)

# Convert NA to "missing" in a character vector
convert_na_to(
  c("a", NA, "d", "z", NA, "t"),
  replacement = "missing"
)

### For data frames

test_df <- data.frame(
  x = c(1, 2, NA),
  x2 = c(4, 5, NA),
  y = c("a", "b", NA)
)

# Convert all NA to 0 in numeric variables, and all NA to "missing" in # character variables
convert_na_to(
  test_df,
  replace_num = 0,
  replace_char = "missing"
)

# Convert a specific variable in the data frame
convert_na_to(
  test_df,
  replace_num = 0,
  replace_char = "missing"하였습니다.
select = "x"
)

# Convert all variables starting with "x"
convert_na_to(
  test_df,
  replace_num = 0,
  replace_char = "missing",
  select = starts_with("x")
)

# Convert NA to 1 in variable 'x2' and to 0 in all other numeric variables
convert_na_to(
  test_df,
  replace_num = 0,
  select = list(x2 = 1)
)

---

**convert_to_na**

*Convert non-missing values in a variable into missing values.*

**Description**

Convert non-missing values in a variable into missing values.

**Usage**

```r
convert_to_na(x, ...)
```

`## S3 method for class 'numeric'
convert_to_na(x, na = NULL, verbose = TRUE, ...)

`## S3 method for class 'factor'
convert_to_na(x, na = NULL, drop_levels = FALSE, verbose = TRUE, ...)

`## S3 method for class 'data.frame'
convert_to_na(
  x,
  select = NULL,
  exclude = NULL,
  na = NULL,
  drop_levels = FALSE,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = TRUE,
  ...)
```
Arguments

x
A vector, factor or a data frame.

... Not used.

na Numeric, character vector or logical (or a list of numeric, character vectors or
logicals) with values that should be converted to NA. Numeric values applied to
numeric vectors, character values are used for factors, character vectors or date
variables, and logical values for logical vectors.

verbose Toggle warnings.

drop_levels Logical, for factors, when specific levels are replaced by NA, should unused lev-
els be dropped?

select Variables that will be included when performing the required tasks. Can be
either
• a variable specified as a literal variable name (e.g., column.name),
• a string with the variable name (e.g., "column.name"), or a character vector
of variable names (e.g., c("col1", "col2", "col3")),
• a formula with variable names (e.g., ~column.1 + column.2),
• a vector of positive integers, giving the positions counting from the left (e.g.
1 or c(1, 3, 5)),
• a vector of negative integers, giving the positions counting from the right (e.g.,
-1 or -1:-3),
• one of the following select-helpers: starts_with(), ends_with(), contains(),
a range using : or regex("". starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal").
• or a function testing for logical conditions, e.g. is.numeric() (or is.numeric
or any user-defined function that selects the variables for which the function
returns TRUE (like: foo <- function(x) mean(x) > 3),
• ranges specified via literal variable names, select-helpers (except regex())
and (user-defined) functions can be negated, i.e. return non-matching el-
ements, when prefixed with a -, e.g. -ends_with(""), -is.numeric or
-(Sepal.Width:Petal.Length). **Note:** Negation means that matches are
excluded, and thus, the exclude argument can be used alternatively.
For instance, select=-ends_with("Length") (with -) is equivalent to
exclude=ends_with("Length") (no -). In case negation should not work
as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored,
e.g. find_columns(iris, select = c("Species", "Test")) will just return
"Species".

exclude See select, however, column names matched by the pattern from exclude will
be excluded instead of selected. If NULL (the default), excludes no columns.

ignore_case Logical, if TRUE and when one of the select-helpers or a regular expression is
used in select, ignores lower/upper case in the search pattern when matching
against variable names.

regex Logical, if TRUE, the search pattern from select will be treated as regular ex-
pression. When regex = TRUE, select must be a character string (or a variable
containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains("") or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see ‘Details’), this argument may be used as workaround.

Value

x, where all values in na are converted to NA.

Examples

x <- sample(1:6, size = 30, replace = TRUE)
x
# values 4 and 5 to NA
convert_to_na(x, na = 4:5)

# data frames
set.seed(123)
x <- data.frame(
a = sample(1:6, size = 20, replace = TRUE),
b = sample(letters[1:6], size = 20, replace = TRUE),
c = sample(c(30:33, 99), size = 20, replace = TRUE)
)
# for all numerics, convert 5 to NA. Character/factor will be ignored.
convert_to_na(x, na = 5)

# for numerics, 5 to NA, for character/factor, "f" to NA
convert_to_na(x, na = list(6, "f"))

# select specific variables
convert_to_na(x, select = c("a", "b"), na = list(6, "f"))

data_addprefix

Rename columns and variable names

Description

Safe and intuitive functions to rename variables or rows in data frames. data_rename() will rename column names, i.e. it facilitates renaming variables data_addprefix() or data_addsuffix() add prefixes or suffixes to column names. data_rename_rows() is a convenient shortcut to add or rename row names of a data frame, but unlike row.names(), its input and output is a data frame, thus, integrating smoothly into a possible pipe-workflow.

Usage

data_addprefix(
data,  
pattern,
data_addprefix

    select = NULL,
    exclude = NULL,
    ignore_case = FALSE,
    regex = FALSE,
    verbose = TRUE,
    ...
)

data_addsuffix(
    data,
    pattern,
    select = NULL,
    exclude = NULL,
    ignore_case = FALSE,
    regex = FALSE,
    verbose = TRUE,
    ...
)

data_rename(
    data,
    pattern = NULL,
    replacement = NULL,
    safe = TRUE,
    verbose = TRUE,
    ...
)

data_rename_rows(data, rows = NULL)

Arguments

data A data frame, or an object that can be coerced to a data frame.

pattern Character vector. For data_rename(), indicates columns that should be selected for renaming. Can be NULL (in which case all columns are selected). For data_addprefix() or data_addsuffix(), a character string, which will be added as prefix or suffix to the column names.

select Variables that will be included when performing the required tasks. Can be either
- a variable specified as a literal variable name (e.g., column_name),
- a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
- a formula with variable names (e.g., ~column_1 + column_2),
- a vector of positive integers, giving the positions counting from the left (e.g. 1 or c(1, 3, 5)),
- a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
one of the following select-helpers: starts_with(), ends_with(), contains(),
a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g. starts_with("Sep", "Petal").
• or a function testing for logical conditions, e.g. is.numeric() (or is.numeric),
or any user-defined function that selects the variables for which the function
returns TRUE (like: foo <- function(x) mean(x) > 3),
• ranges specified via literal variable names, select-helpers (except regex())
and (user-defined) functions can be negated, i.e. return non-matching ele-
ments, when prefixed with a -, e.g. ~ends_with(""), ~is.numeric or
~(Sepal.Width:Petal.Length). Note: Negation means that matches are
ecluded, and thus, the exclude argument can be used alternatively.
For instance, select=-ends_with("Length") (with ~) is equivalent to
exclude=ends_with("Length") (no ~). In case negation should not work
as expected, use the exclude argument instead.
If NULL, selects all columns. Patterns that found no matches are silently ignored,
e.g. find_columns(iris, select = c("Species", "Test"))) will just return
"Species".
exclude See select, however, column names matched by the pattern from exclude will
be excluded instead of selected. If NULL (the default), excludes no columns.
ignore_case Logical, if TRUE and when one of the select-helpers or a regular expression is
used in select, ignores lower/upper case in the search pattern when matching
against variable names.
regex Logical, if TRUE, the search pattern from select will be treated as regular ex-
pression. When regex = TRUE, select must be a character string (or a variable
containing a character string) and is not allowed to be one of the supported
select-helpers or a character vector of length > 1. regex = TRUE is compara-
tible to using one of the two select-helpers, select = contains("") or select
= regex(""), however, since the select-helpers may not work when called from
inside other functions (see 'Details'), this argument may be used as workaround.
verbose Toggle warnings and messages.
... Other arguments passed to or from other functions.
replacement Character vector. Indicates the new name of the columns selected in pattern.
Can be NULL (in which case column are numbered in sequential order). If not
NULL, pattern and replacement must be of the same length.
safe Do not throw error if for instance the variable to be renamed/removed doesn’t
exist.
rows Vector of row names.

Value
A modified data frame.

See Also
• Functions to rename stuff: data_rename(), data_rename_rows(), data_addprefix(), data_addsuffix()
• Functions to reorder or remove columns: data_reorder(), data_relocate(), data_remove()
Functions to reshape, pivot or rotate data frames: \texttt{data_to_long()}, \texttt{data_to_wide()}, \texttt{data_rotate()}

Functions to recode data: \texttt{rescale()}, \texttt{reverse()}, \texttt{categorize()}, \texttt{recode_values()}, \texttt{slide()}

Functions to standardize, normalize, rank-transform: \texttt{center()}, \texttt{standardize()}, \texttt{normalize()}, \texttt{ranktransform()}, \texttt{winsorize()}

Split and merge data frames: \texttt{data_partition()}, \texttt{data_merge()}

Functions to find or select columns: \texttt{data_select()}, \texttt{data_find()}

Functions to filter rows: \texttt{data_match()}, \texttt{data_filter()}

\subsection*{Examples}

\begin{verbatim}
# Add prefix / suffix to all columns
head(data_addprefix(iris, "NEW_"))
head(data_addsuffix(iris, "_OLD"))

# Rename columns
head(data_rename(iris, "Sepal.Length", "length"))
# data_rename(iris, "FakeCol", "length", safe=FALSE)  # This fails
head(data_rename(iris, "FakeCol", "length")) # This doesn't
head(data_rename(iris, c("Sepal.Length", "Sepal.Width"), c("length", "width")))

# Reset names
head(data_rename(iris, NULL))

# Change all
head(data_rename(iris, replacement = paste0("Var", 1:5)))
\end{verbatim}

\subsection*{data_arrange}

\textit{Arrange rows by column values}

\subsection*{Description}

\texttt{data_arrange()} orders the rows of a data frame by the values of selected columns.

\subsection*{Usage}

\texttt{data_arrange(data, select = NULL, safe = TRUE)}

\subsection*{Arguments}

\begin{description}
\item[data] A data frame, or an object that can be coerced to a data frame.
\item[select] Character vector of column names. Use a dash just before column name to arrange in decreasing order, for example "-x1".
\item[safe] Do not throw an error if one of the variables specified doesn’t exist.
\end{description}

\subsection*{Value}

A data frame.
Examples

# Arrange using several variables
data_arrange(head(mtcars), c("gear", "carb"))

# Arrange in decreasing order
data_arrange(head(mtcars), "-carb")

# Throw an error if one of the variables specified doesn't exist
try(data_arrange(head(mtcars), c("gear", "foo"), safe = FALSE))

data_codebook

Generate a codebook of a data frame.

data_codebook()
generates codebooks from data frames, i.e. overviews of all variables and some
more information about each variable (like labels, values or value range, frequencies, amount of
missing values).

Usage

data_codebook(
  data,
  select = NULL,
  exclude = NULL,
  variable_label_width = NULL,
  value_label_width = NULL,
  max_values = 10,
  range_at = 6,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = TRUE,
  ...
)

## S3 method for class 'data_codebook'
print_html(
  x,
  font_size = "100%",
  line_padding = 3,
  row_color = "#eeeeee",
  ...
)
Arguments

**data**  
A data frame, or an object that can be coerced to a data frame.

**select**  
Variables that will be included when performing the required tasks. Can be either

- a variable specified as a literal variable name (e.g., column_name),
- a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
- a formula with variable names (e.g., ~column_1 + column_2),
- a vector of positive integers, giving the positions counting from the left (e.g. 1 or c(1, 3, 5)),
- a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
- one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal").
- or a function testing for logical conditions, e.g. is.numeric() (or is.numeric), or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
- ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. -ends_with(""), -is.numeric or -(Sepal.Width:Petal.Length). **Note:** Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

**exclude**  
See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

**variable_label_width**  
Length of variable labels. Longer labels will be wrapped at variable_label_width chars. If NULL, longer labels will not be split into multiple lines. Only applies to labelled data.

**value_label_width**  
Length of value labels. Longer labels will be shortened, where the remaining part is truncated. Only applies to labelled data or factor levels.

**max_values**  
Number of maximum values that should be displayed. Can be used to avoid too many rows when variables have lots of unique values.

**range_at**  
Indicates how many unique values in a numeric vector are needed in order to print a range for that variable instead of a frequency table for all numeric values. Can be useful if the data contains numeric variables with only a few unique values and where full frequency tables instead of value ranges should be displayed.
ignore_case
Logical, if TRUE and when one of the select-helpers or a regular expression is
used in select, ignores lower/upper case in the search pattern when matching
against variable names.

regex
Logical, if TRUE, the search pattern from select will be treated as regular ex-
pression. When regex = TRUE, select must be a character string (or a variable
containing a character string) and is not allowed to be one of the supported
select-helpers or a character vector of length > 1. regex = TRUE is compar-
able to using one of the two select-helpers, select = contains(""), or select
= regex(""), however, since the select-helpers may not work when called from
inside other functions (see ‘Details’), this argument may be used as workaround.

verbose
Toggle warnings and messages on or off.

... Arguments passed to or from other methods.

x
A (grouped) data frame, a vector or a statistical model (for unstandardize()
cannot be a model).

font_size
For HTML tables, the font size.

line_padding
For HTML tables, the distance (in pixel) between lines.

row_color
For HTML tables, the fill color for odd rows.

Value
A formatted data frame, summarizing the content of the data frame. Returned columns include
the column index of the variables in the original data frame (ID), column name, variable label (if
data is labelled), type of variable, number of missing values, unique values (or value range), value
labels (for labelled data), and a frequency table (N for each value). Most columns are formatted as
character vectors.

Note
There are methods to print() the data frame in a nicer output, as well methods for printing in
markdown or HTML format (print_md() and print_html()).

Examples

data(iris)
data_codebook(iris, select = starts_with("Sepal"))

data(efc)
data_codebook(efc)

# shorten labels
data_codebook(efc, variable_label_width = 20, value_label_width = 15)

# automatic range for numerics at more than 5 unique values
data(mtcars)
data_codebook(mtcars, select = starts_with("c"))

# force all values to be displayed
data_codebook(mtcars, select = starts_with("c"), range_at = 100)
data_duplicated  

Extract all duplicates

Description

Extract all duplicates, for visual inspection. Note that it also contains the first occurrence of future duplicates, unlike duplicated() or dplyr::distinct(). Also contains an additional column reporting the number of missing values for that row, to help in the decision-making when selecting which duplicates to keep.

Usage

data_duplicated(
  data,
  select = NULL,
  exclude = NULL,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = TRUE
)

Arguments

data  A data frame.

select  Variables that will be included when performing the required tasks. Can be either

• a variable specified as a literal variable name (e.g., column_name),

• a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),

• a formula with variable names (e.g., ~column_1 + column_2),

• a vector of positive integers, giving the positions counting from the left (e.g., 1 or c(1, 3, 5)),

• a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),

• one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal")

• or a function testing for logical conditions, e.g. is.numeric() (or is.numeric), or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),

• ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a ~, e.g. ~ends_with(""), ~is.numeric or ~(Sepal.Width:Petal.Length). **Note:** Negation means that matches are excluded, and thus, the exclude argument can be used alternatively.
For instance, `select=-ends_with("Length")` (with `-`) is equivalent to `exclude=ends_with("Length")` (no `-`). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. `find_columns(iris, select = c("Species", "Test"))` will just return "Species".

**exclude**  
See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

**ignore_case**  
Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

**regex**  
Logical, if TRUE, the search pattern from select will be treated as regular expression. When `regex = TRUE`, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. `regex = TRUE` is comparable to using one of the two select-helpers, select = contains("") or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.

**verbose**  
Toggle warnings.

**Value**  
A dataframe, containing all duplicates.

**See Also**

`data_unique()`

**Examples**

df1 <- data.frame(  
id = c(1, 2, 3, 1, 3),  
year = c(2022, 2022, 2022, 2022, 2000),  
item1 = c(NA, 1, 1, 2, 3),  
item2 = c(NA, 1, 1, 2, 3),  
item3 = c(NA, 1, 1, 2, 3)
)

data_duplicated(df1, select = "id")

data_duplicated(df1, select = c("id", "year"))

# Filter to exclude duplicates  
df2 <- df1[-c(1, 5), ]  
df2
Description

data_extract() (or its alias extract()) is similar to $. It extracts either a single column or element from an object (e.g., a data frame, list), or multiple columns resp. elements.

Usage

data_extract(data, select, ...)  

## S3 method for class 'data.frame'  
data_extract(  
  data,  
  select,  
  name = NULL,  
  extract = "all",  
  as_data_frame = FALSE,  
  ignore_case = FALSE,  
  regex = FALSE,  
  verbose = TRUE,  
  ...  
)

Arguments

data  
The object to subset. Methods are currently available for data frames and data frame extensions (e.g., tibbles).

select  
Variables that will be included when performing the required tasks. Can be either  
• a variable specified as a literal variable name (e.g., column_name),  
• a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),  
• a formula with variable names (e.g., ~column_1 + column_2),  
• a vector of positive integers, giving the positions counting from the left (e.g. 1 or c(1, 3, 5)),  
• a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),  
• one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal").  
• or a function testing for logical conditions, e.g. is.numeric() (or is.numeric), or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
• ranges specified via literal variable names, select-helpers (except `regex()`)
  and (user-defined) functions can be negated, i.e. return non-matching elements,
  when prefixed with a -, e.g. `-ends_with("")`, `-is.numeric` or
  `-Sepal.Width:Petal.Length`). **Note:** Negation means that matches
  are excluded, and thus, the exclude argument can be used alternatively.
  For instance, `select=-ends_with("Length")` (with -) is equivalent to
  `exclude=ends_with("Length")` (no -). In case negation should not work
  as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored,
  e.g. `find_columns(iris, select = c("Species", "Test"))` will just return
  "Species".

... For use by future methods.

name An optional argument that specifies the column to be used as names for the
  vector elements after extraction. Must be specified either as literal variable name
  (e.g., `column_name`) or as string (`"column_name"`). name will be ignored when
  a data frame is returned.

extract String, indicating which element will be extracted when select matches mul-
  tiple variables. Can be "all" (the default) to return all matched variables,
  "first" or "last" to return the first or last match, or "odd" and "even" to return
  all odd-numbered or even-numbered matches. Note that "first" or "last"
  return a vector (unless `as_data_frame = TRUE`), while "all" can return a vector
  (if only one match was found) or a data frame (for more than one match). Type
  safe return values are only possible when `extract` is "first" or "last" (will
  always return a vector) or when `as_data_frame = TRUE` (always returns a data
  frame).

as_data_frame Logical, if TRUE, will always return a data frame, even if only one variable was
  matched. If FALSE, either returns a vector or a data frame. See `extract`
  for details.

ignore_case Logical, if TRUE and when one of the select-helpers or a regular expression is
  used in select, ignores lower/upper case in the search pattern when matching
  against variable names.

regex Logical, if TRUE, the search pattern from select will be treated as regular ex-
  pression. When `regex = TRUE`, select must be a character string (or a variable
  containing a character string) and is not allowed to be one of the supported
  select-helpers or a character vector of length > 1. `regex = TRUE` is compar-
  able to using one of the two select-helpers, `select = contains("")` or select
  = `regex("")`, however, since the select-helpers may not work when called from
  inside other functions (see `Details`), this argument may be used as workaround.

verbose Toggle warnings.

**Details**

`data_extract()` can be used to select multiple variables or pull a single variable from a data frame.
  Thus, the return value is by default not type safe - `data_extract()` either returns a vector or a data
  frame.
**Extracting single variables (vectors):** When `select` is the name of a single column, or when `select` only matches one column, a vector is returned. A single variable is also returned when `extract` is either "first" or "last". Setting `as_data_frame` to `TRUE` overrides this behaviour and always returns a data frame.

**Extracting a data frame of variables:** When `select` is a character vector containing more than one column name (or a numeric vector with more than one valid column indices), or when `select` uses one of the supported select-helpers that match multiple columns, a data frame is returned. Setting `as_data_frame` to `TRUE` always returns a data frame.

**Value**

A vector (or a data frame) containing the extracted element, or NULL if no matching variable was found.

**Examples**

```r
# single variable
data_extract(mtcars, cyl, name = gear)
data_extract(mtcars, "cyl", name = gear)
data_extract(mtcars, -1, name = gear)
data_extract(mtcars, cyl, name = 0)
data_extract(mtcars, cyl, name = "row.names")

# selecting multiple variables
head(data_extract(iris, starts_with("Sepal")))
head(data_extract(iris, ends_with("Width")))
head(data_extract(iris, 2:4))

# select first of multiple variables
data_extract(iris, starts_with("Sepal"), extract = "first")

# select first of multiple variables, return as data frame
data_extract(iris, starts_with("Sepal"), extract = "first", as_data_frame = TRUE)
```

---

**Description**

This function is comparable to `dplyr::group_by()`, but just following the `datawizard` function design. `data_ungroup()` removes the grouping information from a grouped data frame.

**Usage**

```r
data_group(
data,
  select = NULL,
  exclude = NULL,
```
data_ungroup(data, verbose = TRUE, ...)

Arguments

data  A data frame

select  Variables that will be included when performing the required tasks. Can be either
  • a variable specified as a literal variable name (e.g., column_name),
  • a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
  • a formula with variable names (e.g., ~column_1 + column_2),
  • a vector of positive integers, giving the positions counting from the left (e.g., 1 or c(1, 3, 5)),
  • a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
  • one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal"),
  • or a function testing for logical conditions, e.g. is.numeric() (or is.numeric), or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
  • ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. ~ends_with(""), ~is.numeric or ~(Sepal.Width:Petal.Length). Note: Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

exclude  See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

ignore_case  Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

regex  Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported
data_match

select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains(""") or select = regex(""""), however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.

verbose

Toggle warnings.

... Arguments passed down to other functions. Mostly not used yet.

Value

A grouped data frame, i.e. a data frame with additional information about the grouping structure saved as attributes.

Examples

data(efc)
suppressPackageStartupMessages(library(poorman, quietly = TRUE))

# total mean
efc %>%
  summarize(mean_hours = mean(c12hour, na.rm = TRUE))

# mean by educational level
efc %>%
  data_group(c172code) %>%
  data_group(c172code) %>
  summarize(mean_hours = mean(c12hour, na.rm = TRUE))

Description

Return a filtered (or sliced) data frame or row indices of a data frame that match a specific condition. data_filter() works like data_match(), but works with logical expressions or row indices of a data frame to specify matching conditions.

Usage

data_match(x, to, match = "and", return_indices = FALSE, drop_na = TRUE, ...)

data_filter(x, ...)

Return filtered or sliced data frame, or row indices
**Arguments**

- **x**: A data frame.
- **to**: A data frame matching the specified conditions. Note that if `match` is a value other than "and", the original row order might be changed. See 'Details'.
- **match**: String, indicating with which logical operation matching conditions should be combined. Can be "and" (or "&"), "or" (or "|") or "not" (or "!").
- **return_indices**: Logical, if FALSE, return the vector of rows that can be used to filter the original data frame. If FALSE (default), returns directly the filtered data frame instead of the row indices.
- **drop_na**: Logical, if TRUE, missing values (NAs) are removed before filtering the data. This is the default behaviour, however, sometimes when row indices are requested (i.e. return_indices=TRUE), it might be useful to preserve NA values, so returned row indices match the row indices of the original data frame.
- **...**: A sequence of logical expressions indicating which rows to keep, or a numeric vector indicating the row indices of rows to keep. Can also be a string representation of a logical expression (e.g. "x > 4"), a character vector (e.g. c("x > 4", "y == 2")) or a variable that contains the string representation of a logical expression. These might be useful when used in packages to avoid defining undefined global variables.

**Details**

For `data_match()`, if `match` is either "or" or "not", the original row order from `x` might be changed. If preserving row order is required, use `data_filter()` instead.

```r
# mimics subset() behaviour, preserving original row order
head(data_filter(mtcars[c("mpg", "vs", "am")], vs == 0 | am == 1))
#> mpg vs am
#> Mazda RX4 21.0 0 1
#> Mazda RX4 Wag 21.0 0 1
#> Datsun 710 22.8 1 1
#> Hornet Sportabout 18.7 0 0
#> Duster 360 14.3 0 0
#> Merc 450SE 16.4 0 0

# re-sorting rows
data.frame(vs = 0, am = 1)
```

```r
# mimics subset() behaviour, preserving original row order
data.frame(vs = 0, am = 1)
```

```r
# re-sorting rows
head(data_match(mtcars[c("mpg", "vs", "am")],
                   data.frame(vs = 0, am = 1),
                   match = "or")
#> mpg vs am
#> Mazda RX4 21.0 0 1
#> Mazda RX4 Wag 21.0 0 1
#> Hornet Sportabout 18.7 0 0
#> Duster 360 14.3 0 0
#> Merc 450SE 16.4 0 0
#> Merc 450SL 17.3 0 0
```
While `data_match()` works with data frames to match conditions against, `data_filter()` is basically a wrapper around `subset(subset = <filter>)`. However, unlike `subset()`, it preserves label attributes and is useful when working with labelled data.

**Value**

A filtered data frame, or the row indices that match the specified configuration.

**See Also**

- Functions to rename stuff: `data_rename()`, `data_rename_rows()`, `data_addprefix()`, `data_addsuffix()`
- Functions to reorder or remove columns: `data_reorder()`, `data_relocate()`, `data_remove()`
- Functions to reshape, pivot or rotate data frames: `data_to_long()`, `data_to_wide()`, `data_rotate()`
- Functions to recode data: `rescale()`, `reverse()`, `categorize()`, `recode_values()`, `slide()`
- Functions to standardize, normalize, rank-transform: `center()`, `standardize()`, `normalize()`, `ranktransform()`, `winsorize()`
- Split and merge data frames: `data_partition()`, `data_merge()`
- Functions to find or select columns: `data_select()`, `data_find()`
- Functions to filter rows: `data_match()`, `data_filter()`

**Examples**

data_match(mtcars, data.frame(vs = 0, am = 1))
data_match(mtcars, data.frame(vs = 0, am = c(0, 1)))

# observations where "vs" is NOT 0 AND "am" is NOT 1
data_match(mtcars, data.frame(vs = 0, am = 1), match = "not")
# equivalent to
data_filter(mtcars, vs != 0 & am != 1)

# observations where EITHER "vs" is 0 OR "am" is 1
data_match(mtcars, data.frame(vs = 0, am = 1), match = "or")
# equivalent to
data_filter(mtcars, vs == 0 | am == 1)

# slice data frame by row indices
data_filter(mtcars, 5:10)

# Define a custom function containing data_filter()
my_filter <- function(data, variable) {
  data_filter(data, variable)
}
my_filter(mtcars, "cyl == 6")

# Pass complete filter-condition as string.
my_filter <- function(data, condition) {
  data_filter(data, condition)
}
my_filter(mtcars, "am != 0")
# string can also be used directly as argument
data_filter(mtcars, "am != 0")

# or as variable
fl <- "am != 0"
data_filter(mtcars, fl)

data_merge

Merge (join) two data frames, or a list of data frames

Description

Merge (join) two data frames, or a list of data frames. However, unlike base R’s `merge()`, `data_merge()` offers a few more methods to join data frames, and it does not drop data frame nor column attributes.

Usage

data_merge(x, ...)
data_join(x, ...)

## S3 method for class 'data.frame'
data_merge(x, y, join = "left", by = NULL, id = NULL, verbose = TRUE, ...)

## S3 method for class 'list'
data_merge(x, join = "left", by = NULL, id = NULL, verbose = TRUE, ...)

Arguments

x, y A data frame to merge. x may also be a list of data frames that will be merged. Note that the list-method has no y argument.

... Not used.

join Character vector, indicating the method of joining the data frames. Can be "full", "left" (default), "right", "inner", "anti", "semi" or "bind". See details below.

by Specifications of the columns used for merging.

id Optional name for ID column that will be created to indicate the source data frames for appended rows. Only applies if join = "bind".

verbose Toggle warnings.

Details

Merging data frames: Merging data frames is performed by adding rows (cases), columns (variables) or both from the source data frame (y) to the target data frame (x). This usually requires one or more variables which are included in both data frames and that are used for merging, typically indicated with the by argument. When by contains a variable present in both data frames, cases are matched and filtered by identical values of by in x and y.
Left- and right-joins: Left- and right joins usually don’t add new rows (cases), but only new columns (variables) for existing cases in \( x \). For join = "left" or join = "right" to work, by must indicate one or more columns that are included in both data frames. For join = "left", if by is an identifier variable, which is included in both \( x \) and \( y \), all variables from \( y \) are copied to \( x \), but only those cases from \( y \) that have matching values in their identifier variable in \( x \) (i.e. all cases in \( x \) that are also found in \( y \) get the related values from the new columns in \( y \)). If there is no match between identifiers in \( x \) and \( y \), the copied variable from \( y \) will get a NA value for this particular case. Other variables that occur both in \( x \) and \( y \), but are not used as identifiers (with by), will be renamed to avoid multiple identical variable names. Cases in \( y \) where values from the identifier have no match in \( x \)’s identifier are removed. join = "right" works in a similar way as join = "left", just that only cases from \( x \) that have matching values in their identifier variable in \( y \) are chosen.

In base R, these are equivalent to `merge(x, y, all.x = TRUE)` and `merge(x, y, all.y = TRUE)`.

Full joins: Full joins copy all cases from \( y \) to \( x \). For matching cases in both data frames, values for new variables are copied from \( y \) to \( x \). For cases in \( y \) not present in \( x \), these will be added as new rows to \( x \). Thus, full joins not only add new columns (variables), but also might add new rows (cases).

In base R, this is equivalent to `merge(x, y, all = TRUE)`.

Inner joins: Inner joins merge two data frames, however, only those rows (cases) are kept that are present in both data frames. Thus, inner joins usually add new columns (variables), but also remove rows (cases) that only occur in one data frame.

In base R, this is equivalent to `merge(x, y)`.

Binds: join = "bind" row-binds the complete second data frame \( y \) to \( x \). Unlike simple `rbind()`, which requires the same columns for both data frames, join = "bind" will bind shared columns from \( y \) to \( x \), and add new columns from \( y \) to \( x \).

Value

A merged data frame.

See Also

- Functions to rename stuff: `data_rename()`, `data_rename_rows()`, `data_addprefix()`, `data_addsuffix()`
- Functions to reorder or remove columns: `data_reorder()`, `data_relocate()`, `data_remove()`
- Functions to reshape, pivot or rotate data frames: `data_to_long()`, `data_to_wide()`, `data_rotate()`
- Functions to recode data: `rescale()`, `reverse()`, `categorize()`, `recode_values()`, `slide()`
- Functions to standardize, normalize, rank-transform: `center()`, `standardize()`, `normalize()`, `ranktransform()`, `winsorize()`
- Split and merge data frames: `data_partition()`, `data_merge()`
- Functions to find or select columns: `data_select()`, `data_find()`
- Functions to filter rows: `data_match()`, `data_filter()`
Examples

```r
x <- data.frame(a = 1:3, b = c("a", "b", "c"), c = 5:7, id = 1:3)
y <- data.frame(c = 6:8, d = c("f", "g", "h"), e = 100:102, id = 2:4)
x
y

# "by" will default to all shared columns, i.e. "c" and "id". new columns
# "d" and "e" will be copied from "y" to "x", but there are only two cases
# in "x" that have the same values for "c" and "id" in "y". only those cases
# have values in the copied columns, the other case gets "NA".
data_merge(x, y, join = "left")

# we change the id-value here
x <- data.frame(a = 1:3, b = c("a", "b", "c"), c = 5:7, id = 1:3)
y <- data.frame(c = 6:8, d = c("f", "g", "h"), e = 100:102, id = 3:5)
x
y

# no cases in "y" have the same matching "c" and "id" as in "x", thus
# copied variables from "y" to "x" copy no values, all get NA.
data_merge(x, y, join = "left")

# one case in "y" has a match in "id" with "x", thus values for this
# case from the remaining variables in "y" are copied to "x", all other
# values (cases) in those remaining variables get NA
data_merge(x, y, join = "left", by = "id")

data(mtcars)
x <- mtcars[1:5, 1:3]
y <- mtcars[28:32, 4:6]

# add ID common column
x$id <- 1:5
y$id <- 3:7

# left-join, add new variables and copy values from y to x,
# where "id" values match
data_merge(x, y)

# right-join, add new variables and copy values from x to y,
# where "id" values match
data_merge(x, y, join = "right")

# full-join
data_merge(x, y, join = "full")

data(mtcars)
x <- mtcars[1:5, 1:3]
```
y <- mtcars[28:32, c(1, 4:5)]

# add ID common column
x$id <- 1:5
y$id <- 3:7

# left-join, no matching rows (because columns "id" and "disp" are used)
# new variables get all NA values
data_merge(x, y)

# one common value in "mpg", so one row from y is copied to x
data_merge(x, y, by = "mpg")

# only keep rows with matching values in by-column
data_merge(x, y, join = "semi", by = "mpg")

# only keep rows with non-matching values in by-column
data_merge(x, y, join = "anti", by = "mpg")

# merge list of data frames. can be of different rows
x <- mtcars[1:5, 1:3]
y <- mtcars[28:31, 3:5]
z <- mtcars[11:18, c(1, 3:4, 6:8)]
x$id <- 1:5
y$id <- 4:7
z$id <- 3:10
data_merge(list(x, y, z), join = "bind", by = "id", id = "source")

---

data_modify  

Create new variables in a data frame

Description

Create new variables or modify existing variables in a data frame. Unlike base::transform(), data_modify() can be used on grouped data frames, and newly created variables can be directly used.

Usage

data_modify(data, ...)

Arguments

data  
A data frame

...  
One or more expressions that define the new variable name and the values or recoding of those new variables. These expressions can be one of:

- A sequence of named, literal expressions, where the left-hand side refers to the name of the new variable, while the right-hand side represent the values of the new variable. Example: Sepal.Width = center(Sepal.Width).
• A sequence of string values, representing expressions.
• A variable that contains a string representation of the expression. Example:
  \[
a \leftarrow "2 \times \text{Sepal.Width}"
\]
  \[
data\_modify(\text{iris}, a)\]
• A character vector of expressions. Example:
  \[
  \text{c("SW\_double = 2 \times \text{Sepal.Width}",}
  \text{"SW\_fraction = SW\_double / 10")}. \text{This type of expression cannot be}
  \text{mixed with other expressions, i.e. if a character vector is provided, you}
  \text{may not add further elements to ... .}
• Using NULL as right-hand side removes a variable from the data frame. Ex-
  ample: Petal.Width = NULL.

Note that newly created variables can be used in subsequent expressions. See
also 'Examples'.

Note

\text{data\_modify()} \text{can also be used inside functions. However, it is recommended to pass the recode-
expression as character vector or list of characters.}

Examples

\text{data(efc)}
\text{new\_efc} \leftarrow \text{data\_modify(}
  \text{efc},
  \text{c12hour\_c} = \text{center(c12hour)},
  \text{c12hour\_z} = \text{c12hour\_c} / \text{sd(c12hour, na.rm = TRUE)},
  \text{c12hour\_z2} = \text{standardize(c12hour)}
\text{)}
\text{head(new\_efc)}

# using strings instead of literal expressions
\text{new\_efc} \leftarrow \text{data\_modify(}
  \text{efc},
  "c12hour\_c = \text{center(c12hour)}",
  "c12hour\_z = \text{c12hour\_c} / \text{sd(c12hour, na.rm = TRUE)}",
  "c12hour\_z2 = \text{standardize(c12hour)}"
\text{)}
\text{head(new\_efc)}

# using character strings, provided as variable
\text{stand} \leftarrow "\text{c12hour\_c} / \text{sd(c12hour, na.rm = TRUE)}"
\text{new\_efc} \leftarrow \text{data\_modify(}
  \text{efc},
  \text{c12hour\_c} = \text{center(c12hour)},
  \text{c12hour\_z} = \text{stand}
\text{)}
\text{head(new\_efc)}

# providing expressions as character vector
\text{new\_exp} \leftarrow \text{c(}
  "\text{c12hour\_c} = \text{center(c12hour)}",
  "\text{c12hour\_z2} = \text{standardize(c12hour)}",
  "\text{c12hour\_z} = \text{c12hour\_c} / \text{sd(c12hour, na.rm = TRUE)}"
\text{)}
\text{head(new\_exp)}
"c12hour_z = c12hour_c / sd(c12hour, na.rm = TRUE)"
)
new_efc <- data_modify(efc, new_exp)
head(new_efc)

# attributes - in this case, value and variable labels - are preserved
str(new_efc)

# overwrite existing variable, remove old variable
out <- data_modify(iris, Petal.Length = 1 / Sepal.Length, Sepal.Length = NULL)
head(out)

# works on grouped data
grouped_efc <- data_group(efc, "c172code")
new_efc <- data_modify(
  grouped_efc,
  c12hour_c = center(c12hour),
  c12hour_z = c12hour_c / sd(c12hour, na.rm = TRUE),
  c12hour_z2 = standardize(c12hour)
)
head(new_efc)

# works from inside functions
foo <- function(data, z) {
  head(data_modify(data, z))
}
foo(iris, "var_a = Sepal.Width / 10")

new_exp <- c("SW_double = 2 * Sepal.Width", "SW_fraction = SW_double / 10")
foo(iris, new_exp)

---

**data_partition**

**Partition data**

Creates data partitions (for instance, a training and a test set) based on a data frame that can also be stratified (i.e., evenly spread a given factor) using the `group` argument.

**Usage**

data_partition(
data,
  proportion = 0.7,
  group = NULL,
  seed = NULL,
  row_id = ".row_id",
  verbose = TRUE,
  training_proportion = proportion,
  ...
)
data_partition

Arguments

- **data**: A data frame, or an object that can be coerced to a data frame.
- **proportion**: Scalar (between 0 and 1) or numeric vector, indicating the proportion(s) of the training set(s). The sum of proportion must not be greater than 1. The remaining part will be used for the test set.
- **group**: A character vector indicating the name(s) of the column(s) used for stratified partitioning.
- **seed**: A random number generator seed. Enter an integer (e.g., 123) so that the random sampling will be the same each time you run the function.
- **row_id**: Character string, indicating the name of the column that contains the row-id’s.
- **verbose**: Toggle messages and warnings.
- **training_proportion**: Deprecated, please use proportion.
- **...**: Other arguments passed to or from other functions.

Value

A list of data frames. The list includes one training set per given proportion and the remaining data as test set. List elements of training sets are named after the given proportions (e.g., $p_{0.7}$), the test set is named $test$.

See Also

- Functions to rename stuff: `data_rename()`, `data_rename_rows()`, `data_addprefix()`, `data_addsuffix()`
- Functions to reorder or remove columns: `data_reorder()`, `data_relocate()`, `data_remove()`
- Functions to reshape, pivot or rotate data frames: `data_to_long()`, `data_to_wide()`, `data_rotate()`
- Functions to recode data: `rescale()`, `reverse()`, `categorize()`, `recode_values()`, `slide()`
- Functions to standardize, normalize, rank-transform: `center()`, `standardize()`, `normalize()`, `ranktransform()`, `winsorize()`
- Split and merge data frames: `data_partition()`, `data_merge()`
- Functions to find or select columns: `data_select()`, `data_find()`
- Functions to filter rows: `data_match()`, `data_filter()`

Examples

data(iris)
out <- data_partition(iris, proportion = 0.9)
out$test
nrow(out$p_0.9)

# Stratify by group (equal proportions of each species)
out <- data_partition(iris, proportion = 0.9, group = "Species")
out$test

# Create multiple partitions
data_peek

out <- data_partition(iris, proportion = c(0.3, 0.3))
lapply(out, head)

# Create multiple partitions, stratified by group - 30% equally sampled
# from species in first training set, 50% in second training set and
# remaining 20% equally sampled from each species in test set.
out <- data_partition(iris, proportion = c(0.3, 0.5), group = "Species")
lapply(out, function(i) table(i$Species))

data_peek

---

**Peek at values and type of variables in a data frame**

Description

This function creates a table a data frame, showing all column names, variable types and the first values (as many as fit into the screen).

Usage

```r
data_peek(x, ...)
```

### S3 method for class 'data.frame'

```r
data_peek(
  x,
  select = NULL,
  exclude = NULL,
  ignore_case = FALSE,
  regex = FALSE,
  width = NULL,
  verbose = TRUE,
  ...
)
```

Arguments

- **x** A data frame.
- **...** not used.
- **select** Variables that will be included when performing the required tasks. Can be either
  - a variable specified as a literal variable name (e.g., `column_name`),
  - a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., `c("col1", "col2", "col3")`),
  - a formula with variable names (e.g., `~column_1 + column_2`),
  - a vector of positive integers, giving the positions counting from the left (e.g., 1 or `c(1, 3, 5)`),
  - if `select` is `NULL`, no variables will be included.
- **exclude** Variables that will be excluded when performing the required tasks.
- **ignore_case** Ignores case when matching variable names.
- **regex** Uses regular expressions for matching variable names.
- **width** sets character width for variable names.
• a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
• one of the following select-helpers: starts_with(), ends_with(), contains(),
   a range using : or regex(""), starts_with(), ends_with(), and contains()
   accept several patterns, e.g starts_with("Sep", "Petal")
• or a function testing for logical conditions, e.g. is.numeric() (or is.numeric),
   or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
• ranges specified via literal variable names, select-helpers (except regex())
   and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. ~ends_with(""), ~is.numeric or
   ~(Sepal.Width:Petal.Length). Note: Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=~ends_with("Length") (with ~) is equivalent to exclude=ends_with("Length") (no ~). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

exclude
   See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

ignore_case
   Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

regex
   Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains("") or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.

width
   Maximum width of line length to display. If NULL, width will be determined using options()$width.

verbose
   Toggle warnings.

Value
   A data frame with three columns, containing information about the name, type and first values of the input data frame.

Note
   To show only specific or a limited number of variables, use the select argument, e.g. select = 1:5 to show only the first five variables.
**Examples**

```r
data(efc)
data.peek(efc)
  # show variables two to four
data.peek(efc, select = 2:4)
```

---

**data_read**  
*Read (import) data files from various sources*

**Description**

This function imports data from various file types. It is a small wrapper around `haven::read_spss()`, `haven::read_stata()`, `haven::read_sas()`, `readxl::read_excel()` and `data.table::fread()` resp. `readr::read_delim()` (the latter if package `data.table` is not installed). Thus, supported file types for importing data are data files from SPSS, SAS or Stata, Excel files or text files (like `.csv` files). All other file types are passed to `rio::import()`. `data_write()` works in a similar way.

**Usage**

```r
data_read(
  path,
  path_catalog = NULL,
  encoding = NULL,
  convert_factors = TRUE,
  verbose = TRUE,
  ...
)

data_write(
  data,
  path,
  delimiter = ",",
  convert_factors = FALSE,
  save_labels = FALSE,
  verbose = TRUE,
  ...
)
```

**Arguments**

- **path**  
  Character string, the file path to the data file.

- **path_catalog**  
  Character string, path to the catalog file. Only relevant for SAS data files.

- **encoding**  
  The character encoding used for the file. Usually not needed.

- **convert_factors**  
  If TRUE (default), numeric variables, where all values have a value label, are assumed to be categorical and converted into factors. If FALSE, no variable types
are guessed and no conversion of numeric variables into factors will be performed. See also section 'Differences to other packages'. For `data_write()`, this argument only applies to the text (e.g. .txt or .csv) or spreadsheet file formats (like .xlsx). Converting to factors might be useful for these formats because labelled numeric variables are then converted into factors and exported as character columns - else, value labels would be lost and only numeric values are written to the file.

**verbose**

Toggle warnings and messages.

... Arguments passed to the related read_*() or write_*() functions.

**data**

The data frame that should be written to a file.

**delimiter**

For CSV-files, specifies the delimiter. Defaults to ",", but in particular in European regions, ";" might be a useful alternative, especially when exported CSV-files should be opened in Excel.

**save_labels**

Only applies to CSV files. If TRUE, value and variable labels (if any) will be saved as additional CSV file. This file has the same file name as the exported CSV file, but includes a ",_labels" suffix (i.e. when the file name is "mydat.csv", the additional file with value and variable labels is named "mydat_labels.csv").

**Value**

A data frame.

**Supported file types**

- `data_read()` is a wrapper around the haven, data.table, readr readxl and rio packages. Currently supported file types are .txt, .csv, .xls, .xlsx, .sav, .por, .dta and .sas (and related files). All other file types are passed to rio::import().

- `data_write()` is a wrapper around haven, readr and rio packages, and supports writing files into all formats supported by these packages.

**Compressed files (zip) and URLs**

data_read() can also read the above mentioned files from URLs or from inside zip-compressed files. Thus, `path` can also be a URL to a file like "http://www.url.com/file.csv". When `path` points to a zip-compressed file, and there are multiple files inside the zip-archive, then the first supported file is extracted and loaded.

**General behaviour**

data_read() detects the appropriate read_*() function based on the file-extension of the data file. Thus, in most cases it should be enough to only specify the `path` argument. However, if more control is needed, all arguments in ... are passed down to the related read_*() function. The same applies to `data_write()`, i.e. based on the file extension provided in `path`, the appropriate write_*() function is used automatically.
SPSS specific behaviour

`data_read()` does not import user-defined ("tagged") NA values from SPSS, i.e. argument `user_na` is always set to `FALSE` when importing SPSS data with the `haven` package. Use `convert_to_na()` to define missing values in the imported data, if necessary. Furthermore, `data_write()` compresses SPSS files by default. If this causes problems with (older) SPSS versions, use `compress = "none"`, for example `data_write(data, "myfile.sav", compress = "none")`.

Differences to other packages that read foreign data formats

`data_read()` is most comparable to `rio::import()`. For data files from SPSS, SAS or Stata, which support labelled data, variables are converted into their most appropriate type. The major difference to `rio::import()` is that `data_read()` automatically converts fully labelled numeric variables into factors, where imported value labels will be set as factor levels. If a numeric variable has no value labels or less value labels than values, it is not converted to factor. In this case, value labels are preserved as "labels" attribute. Character vectors are preserved. Use `convert_factors = FALSE` to remove the automatic conversion of numeric variables to factors.

---

**data_relocate**

---

Relocate (reorder) columns of a data frame

**Description**

`data_relocate()` will reorder columns to specific positions, indicated by `before` or `after`. `data_reorder()` will instead move selected columns to the beginning of a data frame. Finally, `data_remove()` removes columns from a data frame. All functions support select-helpers that allow flexible specification of a search pattern to find matching columns, which should be reordered or removed.

**Usage**

```r
data_relocate(
  data,
  select,
  before = NULL,
  after = NULL,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = TRUE,
  ...
)
```

```r
data_reorder(
  data,
  select,
  exclude = NULL,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = TRUE,
)```

data_remove(
    data,
    select = NULL,
    exclude = NULL,
    ignore_case = FALSE,
    regex = FALSE,
    verbose = FALSE,

Arguments

data       A data frame.
select     Variables that will be included when performing the required tasks. Can be either
• a variable specified as a literal variable name (e.g., column_name),
• a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
• a formula with variable names (e.g., ~column_1 + column_2),
• a vector of positive integers, giving the positions counting from the left (e.g., 1 or c(1, 3, 5)),
• a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
• one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal")..
• or a function testing for logical conditions, e.g. is.numeric() (or is.numeric), or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
• ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. -ends_with(""), -is.numeric or -(Sepal.Width:Petal.Length). Note: Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

before, after Destination of columns. Supplying neither will move columns to the left-hand side; specifying both is an error. Can be a character vector, indicating the name of the destination column, or a numeric value, indicating the index number of the destination column. If -1, will be added before or after the last column.
**data_relocate**

- **ignore_case**: Logical, if TRUE and when one of the select-helpers or a regular expression is used in `select`, ignores lower/upper case in the search pattern when matching against variable names.

- **regex**: Logical, if TRUE, the search pattern from `select` will be treated as regular expression. When `regex = TRUE`, `select` must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. `regex = TRUE` is comparable to using one of the two select-helpers, `select = contains("")` or `select = regex("")`, however, since the select-helpers may not work when called from inside other functions (see ‘Details’), this argument may be used as workaround.

- **verbose**: Toggle warnings.

- **exclude**: See `select`, however, column names matched by the pattern from `exclude` will be excluded instead of selected. If NULL (the default), excludes no columns.

**Value**

A data frame with reordered columns.

**See Also**

- Functions to rename stuff: `data_rename()`, `data_rename_rows()`, `data_addprefix()`, `data_addsuffix()`
- Functions to reorder or remove columns: `data_reorder()`, `data_relocate()`, `data_remove()`
- Functions to reshape, pivot or rotate data frames: `data_to_long()`, `data_to_wide()`, `data_rotate()`
- Functions to recode data: `rescale()`, `reverse()`, `categorize()`, `recode_values()`, `slide()`
- Functions to standardize, normalize, rank-transform: `center()`, `standardize()`, `normalize()`, `ranktransform()`, `winsorize()`
- Split and merge data frames: `data_partition()`, `data_merge()`
- Functions to find or select columns: `data_select()`, `data_find()`
- Functions to filter rows: `data_match()`, `data_filter()`

**Examples**

```r
# Reorder columns
head(data_relocate(iris, select = "Species", before = "Sepal.Length"))
head(data_relocate(iris, select = "Species", before = "Sepal.Width"))
head(data_relocate(iris, select = "Sepal.Width", after = "Species"))
# which is same as
head(data_relocate(iris, select = "Sepal.Width", after = -1))

# Reorder multiple columns
head(data_relocate(iris, select = c("Species", "Petal.Length"), after = "Sepal.Width"))
# which is same as
head(data_relocate(iris, select = c("Species", "Petal.Length"), after = 2))

# Reorder columns
head(data_reorder(iris, c("Species", "Sepal.Length")))
```
# Remove columns
head(data_remove(iris, "Sepal.Length"))
head(data_remove(iris, starts_with("Sepal")))

---

## data_restoretype

**Restore the type of columns according to a reference data frame**

### Description

Restore the type of columns according to a reference data frame.

### Usage

```r
data_restoretype(data, reference = NULL, ...)
```

### Arguments

- `data`: A data frame to pivot.
- `reference`: A reference data frame from which to find the correct column types. If `NULL`, each column is converted to numeric if it doesn’t generate NAs. For example, `c("1", "2")` can be converted to numeric but not `c("Sepal.Length")`.
- `...`: Currently not used.

### Value

A data frame with columns whose types have been restored based on the reference data frame.

### Examples

```r
data <- data.frame(
  Sepal.Length = c("1", "3", "2"),
  Species = c("setosa", "versicolor", "setosa"),
  New = c("1", "3", "4")
)

fixed <- data_restoretype(data, reference = iris)
summary(fixed)
```
**data_rotate**

**Rotate a data frame**

**Description**

This function rotates a data frame, i.e. columns become rows and vice versa. It’s the equivalent of using `t()` but restores the `data.frame` class, preserves attributes and prints a warning if the data type is modified (see example).

**Usage**

```r
data_rotate(data, rownames = NULL, colnames = FALSE, verbose = TRUE)
data_transpose(data, rownames = NULL, colnames = FALSE, verbose = TRUE)
```

**Arguments**

- `data`: A data frame.
- `rownames`: Character vector (optional). If not `NULL`, the data frame’s rownames will be added as (first) column to the output, with `rownames` being the name of this column.
- `colnames`: Logical or character vector (optional). If TRUE, the values of the first column in `x` will be used as column names in the rotated data frame. If a character vector, values from that column are used as column names.
- `verbose`: Toggle warnings.

**Value**

A (rotated) data frame.

**See Also**

- Functions to rename stuff: `data_rename()`, `data_rename_rows()`, `data_addprefix()`, `data_addsuffix()`
- Functions to reorder or remove columns: `data_reorder()`, `data_relocate()`, `data_remove()`
- Functions to reshape, pivot or rotate data frames: `data_to_long()`, `data_to_wide()`, `data_rotate()`
- Functions to recode data: `rescale()`, `reverse()`, `categorize()`, `recode_values()`, `slide()`
- Functions to standardize, normalize, rank-transform: `center()`, `standardize()`, `normalize()`, `ranktransform()`, `winsorize()`
- Split and merge data frames: `data_partition()`, `data_merge()`
- Functions to find or select columns: `data_select()`, `data_find()`
- Functions to filter rows: `data_match()`, `data_filter()`
Examples

```r
x <- mtcars[1:3, 1:4]

x

data_rotate(x)
data_rotate(x, rownames = "property")

# use values in 1. column as column name
data_rotate(x, colnames = TRUE)
data_rotate(x, rownames = "property", colnames = TRUE)

# use either first column or specific column for column names
x <- data.frame(a = 1:5, b = 11:15, c = 21:25)
data_rotate(x, colnames = TRUE)
data_rotate(x, colnames = "c")
```

---

data_separate

Separate single variable into multiple variables

Description

Separates a single variable into multiple new variables.

Usage

```r
data_separate(
  data,
  select = NULL,
  new_columns = NULL,
  separator = "[^[[:alnum:]]+",
  guess_columns = NULL,
  merge_multiple = FALSE,
  merge_separator = "",
  fill = "right",
  extra = "drop_right",
  convert_na = TRUE,
  exclude = NULL,
  append = FALSE,
  ignore_case = FALSE,
  verbose = TRUE,
  regex = FALSE,
  ...
)
```
data_separate

Arguments

data  A data frame.

select  Variables that will be included when performing the required tasks. Can be either
  • a variable specified as a literal variable name (e.g., column_name),
  • a string with the variable name (e.g., "column_name"), or a character vector
    of variable names (e.g., c("col1", "col2", "col3")),
  • a formula with variable names (e.g., ~column_1 + column_2),
  • a vector of positive integers, giving the positions counting from the left (e.g.
    1 or c(1, 3, 5)),
  • a vector of negative integers, giving the positions counting from the right
    (e.g., -1 or -1:-3),
  • one of the following select-helpers: starts_with(), ends_with(), contains(),
    a range using : or regex("""). starts_with(), ends_with(), and contains()
    accept several patterns, e.g starts_with("Sep", "Petal").
  • or a function testing for logical conditions, e.g. is.numeric() (or is.numeric),
    or any user-defined function that selects the variables for which the function
    returns TRUE (like: foo <- function(x) mean(x) > 3),
  • ranges specified via literal variable names, select-helpers (except regex())
    and (user-defined) functions can be negated, i.e. return non-matching el-
    ements, when prefixed with a -, e.g. -ends_with(""), -is.numeric or
    -(Sepal.Width:Petal.Length). Note: Negation means that matches are
    excluded, and thus, the exclude argument can be used alternatively.
    For instance, select=-ends_with("Length") (with -) is equivalent to
    exclude=ends_with("Length") (no -). In case negation should not work
    as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored,
  e.g. find_columns(iris, select = c("Species", "Test")) will just return
  "Species".

new_columns  The names of the new columns, as character vector. If more than one variable
  was selected (in select), the new names are prefixed with the name of the
  original column. new_columns can also be a list of (named) character vectors
  when multiple variables should be separated. See 'Examples'.

separator  Separator between columns. Can be a character vector, which is then treated
  as regular expression, or a numeric vector that indicates at which positions the
  string values will be split.

guess_columns  If new_columns is not given, the required number of new columns is guessed
  based on the results of value splitting. For example, if a variable is split into three
  new columns, this will be considered as the required number of new columns,
  and columns are named "split_1", "split_2" and "split_3". When values
  from a variable are split into different amount of new columns, the guess_column
  can be either "mode" (number of new columns is based on the most common
  number of splits), "min" or "max" to use the minimum resp. maximum number
  of possible splits as required number of columns.
merge_multiple Logical, if TRUE and more than one variable is selected for separating, new columns can be merged. Value pairs of all split variables are merged.

merge_separator Separator string when merge_multiple = TRUE. Defines the string that is used to merge values together.

fill How to deal with values that return fewer new columns after splitting? Can be "left" (fill missing columns from the left with NA), "right" (fill missing columns from the right with NA) or "value_left" or "value_right" to fill missing columns from left or right with the left-most or right-most values.

extra How to deal with values that return too many new columns after splitting? Can be "drop_left" or "drop_right" to drop the left-most or right-most values, or "merge_left" or "merge_right" to merge the left- or right-most value together, and keeping all remaining values as is.

convert_na Logical, if TRUE, character "NA" values are converted into real NA values.

exclude See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

append Logical, if FALSE (default), removes original columns that were separated. If TRUE, all columns are preserved and the new columns are appended to the data frame.

ignore_case Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

verbose Toggle warnings.

regex Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains("") or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.

... Currently not used.

Value
A data frame with the newly created variable(s), or - when append = TRUE - data including new variables.

See Also
data_unite()

Examples
# simple case
d <- data.frame(
  x = c("1.a.6", "2.b.7", "3.c.8"),

stringsAsFactors = FALSE
)
d
# guess number of columns
d <- data.frame(
  x = c("1.a.6", NA, "2.b.6.7", "3.c", "x.y.z"),
  stringsAsFactors = FALSE
)
d
# drop left-most column
data_separate(d, guess_columns = "mode", extra = "drop_left")

# merge right-most column
data_separate(d, guess_columns = "mode", extra = "merge_right")

# fill columns with fewer values with left-most values
data_separate(d, guess_columns = "mode", fill = "value_left")

# fill and merge
data_separate(
  d,
  guess_columns = "mode",
  fill = "value_left",
  extra = "merge_right"
)

# multiple columns to split

d <- data.frame(
  x = c("1.a.6", "2.b.7", "3.c.8"),
  y = c("x.y.z", "10.11.12", "m.n.o"),
  stringsAsFactors = FALSE
)
d
# split two columns, default column names
data_separate(d, guess_columns = "mode")

# split into new named columns, repeating column names
data_separate(d, new_columns = c("a", "b", "c"))

# split selected variable new columns
data_separate(d, select = "y", new_columns = c("a", "b", "c"))

# merge multiple split columns
data_separate(
  d,
  new_columns = c("a", "b", "c"),
  merge_multiple = TRUE
)
data_tabulate

Create frequency tables of variables

Description

This function creates frequency tables of variables, including the number of levels/values as well as the distribution of raw, valid and cumulative percentages.

Usage

```r
data_tabulate(x, ...)
```

## Default S3 method:
data_tabulate(x, drop_levels = FALSE, name = NULL, verbose = TRUE, ...)

## S3 method for class 'data.frame'
data_tabulate(
x, 
select = NULL, 
exclude = NULL, 
ignore_case = FALSE, 
regex = FALSE, 
```
collapse = FALSE,
drop_levels = FALSE,
verbose = TRUE,
...)

Arguments

x 
A (grouped) data frame, a vector or factor.

... not used.
drop_levels 
Logical, if TRUE, factor levels that do not occur in the data are included in the table (with frequency of zero), else unused factor levels are dropped from the frequency table.

name 
Optional character string, which includes the name that is used for printing.

verbose 
Toggle warnings.

select 
Variables that will be included when performing the required tasks. Can be either

- a variable specified as a literal variable name (e.g., column_name),
- a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
- a formula with variable names (e.g., ~column_1 + column_2),
- a vector of positive integers, giving the positions counting from the left (e.g., 1 or c(1, 3, 5)),
- a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
- one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal").
- or a function testing for logical conditions, e.g. is.numeric() (or is.numeric), or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
- ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. -ends_with(""), -is.numeric or -(Sepal.Width:Petal.Length). Note: Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

exclude 
See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.
**ignore_case**

Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

**regex**

Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains(""_sur) or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.

**collapse**

Logical, if TRUE collapses multiple tables into one larger table for printing. This affects only printing, not the returned object.

**Value**

A data frame, or a list of data frames, with one frequency table as data frame per variable.

**Examples**

```r
data(efc)

# vector/factor
data_tabulate(efc$c172code)

# data frame
data_tabulate(efc, c("e42dep", "c172code"))

# grouped data frame
suppressPackageStartupMessages(library(poorman, quietly = TRUE))
efc %>%
group_by(c172code) %>%
data_tabulate("e16sex")

# collapse tables
efc %>%
group_by(c172code) %>%
data_tabulate("e16sex", collapse = TRUE)

# for larger N's (> 100000), a big mark is automatically added
set.seed(123)
x <- sample(1:3, 1e6, TRUE)
data_tabulate(x, name = "Large Number")

# to remove the big mark, use "print(..., big_mark = "")"
print(data_tabulate(x), big_mark = "")
```
Data to Long

Reshape (pivot) data from wide to long

Description

This function "lengthens" data, increasing the number of rows and decreasing the number of columns. This is a dependency-free base-R equivalent of `tidyr::pivot_longer()`.

Usage

```r
data_to_long(
  data,
  select = "all",
  names_to = "name",
  names_prefix = NULL,
  names_sep = NULL,
  names_pattern = NULL,
  values_to = "value",
  values_drop_na = FALSE,
  rows_to = NULL,
  ignore_case = FALSE,
  regex = FALSE,
  ...
)
```

```r
reshape_longer(
  data,
  select = "all",
  names_to = "name",
  names_prefix = NULL,
  names_sep = NULL,
  names_pattern = NULL,
  values_to = "value",
  values_drop_na = FALSE,
  rows_to = NULL,
  ignore_case = FALSE,
  regex = FALSE,
  ...
)
```

Arguments

- `data`: A data frame to pivot.
Variables that will be included when performing the required tasks. Can be either

- a variable specified as a literal variable name (e.g., `column_name`),
- a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., `c("col1", "col2", "col3")`),
- a formula with variable names (e.g., `~column_1 + column_2`),
- a vector of positive integers, giving the positions counting from the left (e.g., `1` or `c(1, 3, 5)`),
- a vector of negative integers, giving the positions counting from the right (e.g., `-1` or `-1:-3`),
- one of the following select-helpers: `starts_with()`, `ends_with()`, `contains()`, a range using `:`, or `regex("")`. `starts_with()`, `ends_with()`, and `contains()` accept several patterns, e.g `starts_with("Sep", "Petal")`. 
- or a function testing for logical conditions, e.g. `is.numeric()` or `is.numeric`, or any user-defined function that selects the variables for which the function returns `TRUE` (like: `foo <- function(x) mean(x) > 3`),
- ranges specified via literal variable names, select-helpers (except `regex()`) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a `-`, e.g. `~ends_with("")`, `~is.numeric` or `~(Sepal.Width:Petal.Length)`. **Note:** Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, `select=~/ends_with("Length")` (with `-`) is equivalent to `exclude=ends_with("Length")` (no `-`). In case negation should not work as expected, use the exclude argument instead.

If `NULL`, selects all columns. Patterns that found no matches are silently ignored, e.g. `find_columns(iris, select = c("Species", "Test"))` will just return "Species".

The name of the new column that will contain the column names.

A regular expression used to remove matching text from the start of each variable name.

If `names_to` contains multiple values, this argument controls how the column name is broken up. `names_pattern` takes a regular expression containing matching groups, i.e. "()".

The name of the new column that will contain the values of the pivoted variables.

If `TRUE`, will drop rows that contain only `NA` in the `values_to` column. This effectively converts explicit missing values to implicit missing values, and should generally be used only when missing values in data were created by its structure.

The name of the column that will contain the row names or row numbers from the original data. If `NULL`, will be removed.

Logical, if `TRUE` and when one of the select-helpers or a regular expression is used in `select`, ignores lower/upper case in the search pattern when matching against variable names.

Logical, if `TRUE`, the search pattern from `select` will be treated as regular expression. When `regex = TRUE`, select must be a character string (or a variable
containing a character string) and is not allowed to be one of the supported
select-helpers or a character vector of length > 1. regex = TRUE is compara-
tible to using one of the two select-helpers, select = contains("") or select
= regex(""), however, since the select-helpers may not work when called from
inside other functions (see ‘Details’), this argument may be used as workaround.

... Currently not used.

cols Identical to select. This argument is here to ensure compatibility with tidyr::pivot_longer().
If both select and cols are provided, cols is used.

colnames_to Deprecated. Use names_to instead.

Value

If a tibble was provided as input, reshape_longer() also returns a tibble. Otherwise, it returns a
data frame.

See Also

• Functions to rename stuff: data_rename(), data_rename_rows(), data_addprefix(), data_addsuffix()
• Functions to reorder or remove columns: data_reorder(), data_relocate(), data_remove()
• Functions to reshape, pivot or rotate data frames: data_to_long(), data_to_wide(), data_rotate()
• Functions to recode data: rescale(), reverse(), categorize(), recode_values(), slide()
• Functions to standardize, normalize, rank-transform: center(), standardize(), normalize(),
  ranktransform(), winsorize()
• Split and merge data frames: data_partition(), data_merge()
• Functions to find or select columns: data_select(), data_find()
• Functions to filter rows: data_match(), data_filter()

Examples

wide_data <- data.frame(replicate(5, rnorm(10)))

# Default behaviour (equivalent to tidyr::pivot_longer(wide_data, cols = 1:5))
data_to_long(wide_data)

# Customizing the names
data_to_long(wide_data,
  select = c(1, 2),
  names_to = "Column",
  values_to = "Numbers",
  rows_to = "Row"
)

# Full example
# ------------------
data <- psych::bfi # Wide format with one row per participant's personality test

# Pivot long format
data_to_long(data,
  select = regex("\d"), # Select all columns that contain a digit
  names_to = "Item",
  values_to = "Score",
  rows_to = "Participant"
)

data_to_long(
  tidyr::who,
  select = new_sp_m014:newrel_f65,
  names_to = c("diagnosis", "gender", "age"),
  names_pattern = "new_?(.*)_(.)\.(.*)",
  values_to = "count"
)

data_to_wide
Reshape (pivot) data from long to wide

Description
This function "widens" data, increasing the number of columns and decreasing the number of rows.
This is a dependency-free base-R equivalent of tidyr::pivot_wider().

Usage
data_to_wide(
  data,
  id_cols = NULL,
  values_from = "Value",
  names_from = "Name",
  names_sep = "_",
  names_prefix = "",
  names_glue = NULL,
  values_fill = NULL,
  verbose = TRUE,
  ..., 
  colnames_from, 
  rows_from,,
  sep 
)

reshape_wider(
  data,
  id_cols = NULL,
  values_from = "Value",
  names_from = "Name",
  names_sep = "_",
data_to_wide

```r
names_prefix = "", names_glue = NULL, values_fill = NULL, verbose = TRUE,
...
colnames_from,
rows_from,
sep
)
```

**Arguments**

- **data**
  - A data frame to pivot.
- **id_cols**
  - The name of the column that identifies the rows. If NULL, it will use all the unique rows.
- **values_from**
  - The name of the column that contains the values to be used as future variable values.
- **names_from**
  - The name of the column that contains the levels to be used as future column names.
- **names_sep**
  - If names_from or values_from contains multiple variables, this will be used to join their values together into a single string to use as a column name.
- **names_prefix**
  - String added to the start of every variable name. This is particularly useful if names_from is a numeric vector and you want to create syntactic variable names.
- **names_glue**
  - Instead of names_sep and names_prefix, you can supply a glue specification that uses the names_from columns to create custom column names. Note that the only delimiters supported by names_glue are curly brackets, { and }.
- **values_fill**
  - Optionally, a (scalar) value that will be used to replace missing values in the new columns created.
- **verbose**
  - Toggle warnings.
- **...**
  - Not used for now.
- **colnames_from**
  - Deprecated. Use names_from instead.
- **rows_from**
  - Deprecated. Use id_cols instead.
- **sep**
  - Deprecated. Use names_sep instead.

**Value**

If a tibble was provided as input, reshape_wider() also returns a tibble. Otherwise, it returns a data frame.

**See Also**

- Functions to rename stuff: data_rename(), data_rename_rows(), data_addprefix(), data_addsuffix()
- Functions to reorder or remove columns: data_reorder(), data_relocate(), data_remove()
- Functions to reshape, pivot or rotate data frames: data_to_long(), data_to_wide(), data_rotate()
• Functions to recode data: `rescale()`, `reverse()`, `categorize()`, `recode_values()`, `slide()`

• Functions to standardize, normalize, rank-transform: `center()`, `standardize()`, `normalize()`, `ranktransform()`, `winsorize()`

• Split and merge data frames: `data_partition()`, `data_merge()`

• Functions to find or select columns: `data_select()`, `data_find()`

• Functions to filter rows: `data_match()`, `data_filter()`

Examples

data_long <- read.table(header = TRUE, text = "
subject sex condition measurement
1  M  control  7.9
1  M  cond1  12.3
1  M  cond2  10.7
2  F  control  6.3
2  F  cond1  10.6
2  F  cond2  11.1
3  F  control  9.5
3  F  cond1  13.1
3  F  cond2  13.8
4  M  control  11.5
4  M  cond1  13.4
4  M  cond2  12.9"
)

data_to_wide(
data_long, id_cols = "subject", names_from = "condition", values_from = "measurement"
)

data_to_wide(
data_long, id_cols = "subject", names_from = "condition", values_from = "measurement", names_prefix = "Var.", names_sep = "."
)

production <- expand.grid(
  product = c("A", "B"),
  country = c("AI", "EI"),
  year = 2000:2014
)
production <- data_filter(production, (product == "A" & country == "AI") | product == "B")

production$production <- rnorm(nrow(production))

data_to_wide(production)
Keep only one row from all with duplicated IDs

**Description**

From all rows with at least one duplicated ID, keep only one. Methods for selecting the duplicated row are either the first duplicate, the last duplicate, or the "best" duplicate (default), based on the duplicate with the smallest number of NA. In case of ties, it picks the first duplicate, as it is the one most likely to be valid and authentic, given practice effects.

Contrarily to dplyr::distinct(), data_unique() keeps all columns.

**Usage**

```r
data_unique(
  data,
  select = NULL,
  keep = "best",
  exclude = NULL,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = TRUE
)
```

**Arguments**

- **data**: A data frame.
- **select**: Variables that will be included when performing the required tasks. Can be either
  - a variable specified as a literal variable name (e.g., `column_name`),
  - a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., `c("col1", "col2", "col3")`),
  - a formula with variable names (e.g., `~column_1 + column_2`),
  - a vector of positive integers, giving the positions counting from the left (e.g., `1` or `c(1, 3, 5)`),
  - a vector of negative integers, giving the positions counting from the right (e.g., `-1` or `-1:-3`),
  - one of the following select-helpers: `starts_with()`, `ends_with()`, `contains()`, `starts_with()`, `ends_with()`, and `contains()`, accept several patterns, e.g `starts_with("Sep", "Petal")`. 

• or a function testing for logical conditions, e.g. `is.numeric()` (or `is.numeric`), or any user-defined function that selects the variables for which the function returns TRUE (like: `foo <- function(x) mean(x) > 3`),

• ranges specified via literal variable names, select-helpers (except `regex()`) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a `-`, e.g. `-ends_with(""), -is.numeric or -(Sepal.Width:Petal.Length). **Note:** Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, `select=-ends_with("Length")` (with `=`) is equivalent to `exclude=ends_with("Length")` (no `-`). In case negation should not work as expected, use the exclude argument instead.

If `NULL`, selects all columns. Patterns that found no matches are silently ignored, e.g. `find_columns(iris, select = c("Species", "Test"))` will just return "Species".

- **keep**
  The method to be used for duplicate selection, either "best" (the default), "first", or "last".

- **exclude**
  See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If `NULL` (the default), excludes no columns.

- **ignore_case**
  Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

- **regex**
  Logical, if TRUE, the search pattern from select will be treated as regular expression. When `regex = TRUE`, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. `regex = TRUE` is comparable to using one of the two select-helpers, `select = contains("")` or `select = regex("")`, however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.

- **verbose**
  Toggle warnings.

**Value**

A data frame, containing only the chosen duplicates.

**See Also**

`data_duplicated()`

**Examples**

```r
df1 <- data.frame(
  id = c(1, 2, 3, 1, 3),
  item1 = c(NA, 1, 1, 2, 3),
  item2 = c(NA, 1, 1, 2, 3),
  item3 = c(NA, 1, 1, 2, 3)
)
data_unique(df1, select = "id")
```
data_unite  

Unite ("merge") multiple variables

Description
Merge values of multiple variables per observation into one new variable.

Usage

data_unite(
  data,
  new_column = NULL,
  select = NULL,
  exclude = NULL,
  separator = " ",
  append = FALSE,
  remove_na = FALSE,
  ignore_case = FALSE,
  verbose = TRUE,
  regex = FALSE,
  ...
)

Arguments

data  A data frame.
new_column  The name of the new column, as a string.
select  Variables that will be included when performing the required tasks. Can be either

  • a variable specified as a literal variable name (e.g., column_name),
  • a string with the variable name (e.g., "column_name"), or a character vector
    of variable names (e.g., c("col1", "col2", "col3")),
  • a formula with variable names (e.g., ~column_1 + column_2),
  • a vector of positive integers, giving the positions counting from the left (e.g. 1 or c(1, 3, 5)),
  • a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
  • one of the following select-helpers: starts_with(), ends_with(), contains(),
    a range using : or regex(" "). starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal").
  • or a function testing for logical conditions, e.g. is.numeric() (or is.numeric),
    or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
ranges specified via literal variable names, select-helpers (except `regex()`) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. `-ends_with(""), -is.numeric or -(Sepal.Width:Petal.Length). Note: Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, `select=-ends_with("Length")` (with -) is equivalent to `exclude=ends_with("Length")` (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. `find_columns(iris, select = c("Species", "Test"))` will just return "Species".

#### exclude
See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

#### separator
A character to use between values.

#### append
Logical, if FALSE (default), removes original columns that were united. If TRUE, all columns are preserved and the new column is appended to the data frame.

#### remove_na
Logical, if TRUE, missing values (NA) are not included in the united values. If FALSE, missing values are represented as "NA" in the united values.

#### ignore_case
Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

#### verbose
Toggle warnings.

#### regex
Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, `select = contains("")` or `select = regex("")`, however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.

#### Value
Data, with a newly created variable.

#### See Also
`data_separate()`

#### Examples
```r
d <- data.frame(
  x = 1:3,
  y = letters[1:3],
  z = 6:8
)
d
```
`demean` computes group- and de-meaned versions of a variable that can be used in regression analysis to model the between- and within-subject effect. `degroup()` is more generic in terms of the centering-operation. While `demean()` always uses mean-centering, `degroup()` can also use the mode or median for centering.

**Usage**

```r
demean(
  x,
  select,
  group,
  suffix_demean = "_within",
  suffix_groupmean = "_between",
  add_attributes = TRUE,
  verbose = TRUE
)

degroup(
  x,
  select,
  group,
  center = "mean",
  suffix_demean = "_within",
  suffix_groupmean = "_between",
  add_attributes = TRUE,
  verbose = TRUE
)

detrend(
  x,
  select,
  group,
  center = "mean",
  suffix_demean = "_within",
  suffix_groupmean = "_between",
  add_attributes = TRUE,
  verbose = TRUE
)
```
Arguments

- **x**: A data frame.
- **select**: Character vector (or formula) with names of variables to select that should be group- and de-meaned.
- **group**: Character vector (or formula) with the name of the variable that indicates the group- or cluster-ID.
- **suffix_demean, suffix_groupmean**: String value, will be appended to the names of the group-meaned and de-meaned variables of `x`. By default, de-meaned variables will be suffixed with ".within" and grouped-meaned variables with ".between".
- **add_attributes**: Logical, if TRUE, the returned variables gain attributes to indicate the within- and between-effects. This is only relevant when printing `model_parameters()` - in such cases, the within- and between-effects are printed in separated blocks.
- **verbose**: Toggle warnings and messages.
- **center**: Method for centering. `demean()` always performs mean-centering, while `degroup()` can use `center = "median"` or `center = "mode"` for median- or mode-centering, and also "min" or "max".

Details

**Heterogeneity Bias**: Mixed models include different levels of sources of variability, i.e. error terms at each level. When macro-indicators (or level-2 predictors, or higher-level units, or more general: *group-level predictors that vary within and across groups*) are included as fixed effects (i.e. treated as covariate at level-1), the variance that is left unaccounted for this covariate will be absorbed into the error terms of level-1 and level-2 (Bafumi and Gelman 2006; Gelman and Hill 2007, Chapter 12.6): “Such covariates contain two parts: one that is specific to the higher-level entity that does not vary between occasions, and one that represents the difference between occasions, within higher-level entities” (Bell et al. 2015). Hence, the error terms will be correlated with the covariate, which violates one of the assumptions of mixed models (iid, independent and identically distributed error terms). This bias is also called the *heterogeneity bias* (Bell et al. 2015). To resolve this problem, level-2 predictors used as (level-1) covariates should be separated into their "within" and "between" effects by "de-meaning" and "group-meaning": After demeaning time-varying predictors, “at the higher level, the mean term is no longer constrained by Level 1 effects, so it is free to account for all the higher-level variance associated with that variable” (Bell et al. 2015).

**Panel data and correlating fixed and group effects**: `demean()` is intended to create group- and de-meaned variables for panel regression models (fixed effects models), or for complex random-effect-within-between models (see Bell et al. 2015, 2018), where group-effects (random effects) and fixed effects correlate (see Bafumi and Gelman 2006). This can happen, for instance, when analyzing panel data, which can lead to *Heterogeneity Bias*. To control for correlating predictors and group effects, it is recommended to include the group-meaned and de-meaned version of *time-varying covariates* (and group-meaned version of *time-invariant covariates* that are on a higher level, e.g. level-2 predictors) in the model. By this, one can fit complex multilevel models for panel data, including time-varying predictors, time-invariant predictors and random effects.
Why mixed models are preferred over fixed effects models: A mixed models approach can model the causes of endogeneity explicitly by including the (separated) within- and between-effects of time-varying fixed effects and including time-constant fixed effects. Furthermore, mixed models also include random effects, thus a mixed models approach is superior to classic fixed-effects models, which lack information of variation in the group-effects or between-subject effects. Furthermore, fixed effects regression cannot include random slopes, which means that fixed effects regressions are neglecting "cross-cluster differences in the effects of lower-level controls (which) reduces the precision of estimated context effects, resulting in unnecessarily wide confidence intervals and low statistical power" (Heisig et al. 2017).

Terminology: The group-meaned variable is simply the mean of an independent variable within each group (or id-level or cluster) represented by group. It represents the cluster-mean of an independent variable. The regression coefficient of a group-meaned variable is the between-subject-effect. The de-meaned variable is then the centered version of the group-meaned variable. De-meaning is sometimes also called person-mean centering or centering within clusters. The regression coefficient of a de-meaned variable represents the within-subject-effect.

De-meaning with continuous predictors: For continuous time-varying predictors, the recommendation is to include both their de-meaned and group-meaned versions as fixed effects, but not the raw (untransformed) time-varying predictors themselves. The de-meaned predictor should also be included as random effect (random slope). In regression models, the coefficient of the de-meaned predictors indicates the within-subject effect, while the coefficient of the group-meaned predictor indicates the between-subject effect.

De-meaning with binary predictors: For binary time-varying predictors, there are two recommendations. First is to include the raw (untransformed) binary predictor as fixed effect only and the de-meaned variable as random effect (random slope). The alternative would be to add the de-meaned version(s) of binary time-varying covariates as additional fixed effect as well (instead of adding it as random slope). Centering time-varying binary variables to obtain within-effects (level 1) isn’t necessary. They have a sensible interpretation when left in the typical 0/1 format (Hoffmann 2015, chapter 8-2.1). demean() will thus coerce categorical time-varying predictors to numeric to compute the de- and group-meaned versions for these variables, where the raw (untransformed) binary predictor and the de-meaned version should be added to the model.

De-meaning of factors with more than 2 levels: Factors with more than two levels are demeaned in two ways: first, these are also converted to numeric and de-meaned; second, dummy variables are created (binary, with 0/1 coding for each level) and these binary dummy-variables are de-meaned in the same way (as described above). Packages like panelr internally convert factors to dummies before demeaning, so this behaviour can be mimicked here.

De-meaning interaction terms: There are multiple ways to deal with interaction terms of within- and between-effects. A classical approach is to simply use the product term of the de-meaned variables (i.e. introducing the de-meaned variables as interaction term in the model formula, e.g. y ~ x_within * time_within). This approach, however, might be subject to bias (see Gieselmann & Schmidt-Catran 2020).

Another option is to first calculate the product term and then apply the de-meaning to it. This approach produces an estimator “that reflects unit-level differences of interacted variables whose moderators vary within units”, which is desirable if no within interaction of two time-dependent variables is required.
A third option, when the interaction should result in a genuine within estimator, is to "double de-mean" the interaction terms (Giesselmann & Schmidt-Catran 2018), however, this is currently not supported by `demean()`. If this is required, the `wmb()` function from the `panelr` package should be used.

To de-mean interaction terms for within-between models, simply specify the term as interaction for the `select`-argument, e.g. `select = "a*b"` (see 'Examples').

**Analysing panel data with mixed models using lme4:** A description of how to translate the formulas described in Bell et al. 2018 into R using `lmer()` from `lme4` can be found in this vignette.

**Value**

A data frame with the group-/de-meaned variables, which get the suffix "_between" (for the group-meaned variable) and "_within" (for the de-meaned variable) by default.

**References**


See Also

If grand-mean centering (instead of centering within-clusters) is required, see `center()`.

**Examples**

```r
data(iris)
iris$ID <- sample(1:4, nrow(iris), replace = TRUE) # fake-ID
iris$binary <- as.factor(rbinom(150, 1, .35)) # binary variable
```
describe_distribution

x <- demean(iris, select = c("Sepal.Length", "Petal.Length"), group = "ID")
head(x)

x <- demean(iris, select = c("Sepal.Length", "binary", "Species"), group = "ID")
head(x)

# demean interaction term x*y
dat <- data.frame(
a = c(1, 2, 3, 4, 1, 2, 3, 4),
x = c(4, 3, 3, 4, 1, 2, 1, 2),
y = c(1, 2, 1, 2, 4, 3, 2, 1),
ID = c(1, 2, 3, 1, 2, 3, 1, 2)
)
demean(dat, select = c("a", "x*y"), group = "ID")

# or in formula-notation
demean(dat, select = ~ a + x * y, group = ~ID)

describe_distribution  Describe a distribution

Description

This function describes a distribution by a set of indices (e.g., measures of centrality, dispersion, range, skewness, kurtosis).

Usage

describe_distribution(x, ...)

## S3 method for class 'numeric'
describe_distribution(
x, centrality = "mean", dispersion = TRUE, iqr = TRUE, range = TRUE, quartiles = FALSE, ci = NULL, iterations = 100, threshold = 0.1, verbose = TRUE, ...)

## S3 method for class 'factor'
describe_distribution(x, dispersion = TRUE, range = TRUE, verbose = TRUE, ...)
describe_distribution

## S3 method for class 'data.frame'

```r
describe_distribution(
  x,
  select = NULL,
  exclude = NULL,
  centrality = "mean",
  dispersion = TRUE,
  iqr = TRUE,
  range = TRUE,
  quartiles = FALSE,
  include_factors = FALSE,
  ci = NULL,
  iterations = 100,
  threshold = 0.1,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = TRUE,
  ...
)
```

**Arguments**

- **x**: A numeric vector, a character vector, a data frame, or a list. See Details.
- **...**: Additional arguments to be passed to or from methods.
- **centrality**: The point-estimates (centrality indices) to compute. Character (vector) or list with one or more of these options: "median", "mean", "MAP" or "all".
- **dispersion**: Logical, if TRUE, computes indices of dispersion related to the estimate(s) (SD and MAD for mean and median, respectively).
- **iqr**: Logical, if TRUE, the interquartile range is calculated (based on `stats::IQR()`, using type = 6).
- **range**: Return the range (min and max).
- **quartiles**: Return the first and third quartiles (25th and 75th percentiles).
- **ci**: Confidence Interval (CI) level. Default is NULL, i.e. no confidence intervals are computed. If not NULL, confidence intervals are based on bootstrap replicates (see iterations). If centrality = "all", the bootstrapped confidence interval refers to the first centrality index (which is typically the median).
- **iterations**: The number of bootstrap replicates for computing confidence intervals. Only applies when ci is not NULL.
- **threshold**: For centrality = "trimmed" (i.e. trimmed mean), indicates the fraction (0 to 0.5) of observations to be trimmed from each end of the vector before the mean is computed.
- **verbose**: Toggle warnings and messages.
- **select**: Variables that will be included when performing the required tasks. Can be either
describe_distribution

- a variable specified as a literal variable name (e.g., column_name),
- a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
- a formula with variable names (e.g., ~column_1 + column_2),
- a vector of positive integers, giving the positions counting from the left (e.g. 1 or c(1, 3, 5)),
- a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
- one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal").
- or a function testing for logical conditions, e.g. is.numeric() (or is.numeric), or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
- ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. -ends_with(""), -is.numeric or -(Sepal.Width:Petal.Length). **Note:** Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

**exclude**
See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

**include_factors**
Logical, if TRUE, factors are included in the output, however, only columns for range (first and last factor levels) as well as n and missing will contain information.

**ignore_case**
Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

**regex**
Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains("") or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see ‘Details‘), this argument may be used as workaround.

**Details**
If x is a data frame, only numeric variables are kept and will be displayed in the summary.
If x is a list, the behavior is different whether x is a stored list. If x is stored (for example, describe_distribution(mylist) where mylist was created before), artificial variable names
are used in the summary (Var_1, Var_2, etc.). If `x` is an unquoted list (for example, `describe_distribution(list(mtcars$mpg))`), then "mtcars$mpg" is used as variable name.

**Value**

A data frame with columns that describe the properties of the variables.

**Note**

There is also a `plot()`-method implemented in the `see-package`.

**Examples**

```r
describe_distribution(rnorm(100))

data(iris)
describe_distribution(iris)
describe_distribution(iris, include_factors = TRUE, quartiles = TRUE)
describe_distribution(list(mtcars$mpg, mtcars$cyl))
```

---

**distribution_mode**

*Compute mode for a statistical distribution*

**Description**

Compute mode for a statistical distribution

**Usage**

```r
distribution_mode(x)
```

**Arguments**

- `x`  
  An atomic vector, a list, or a data frame.

**Value**

The value that appears most frequently in the provided data. The returned data structure will be the same as the entered one.

**See Also**

For continuous variables, the **Highest Maximum a Posteriori probability estimate** (MAP) may be a more useful way to estimate the most commonly-observed value than the mode. See `bayestestR::map_estimate()`.  
Examples

distribution_mode(c(1, 2, 3, 3, 4, 5))
distribution_mode(c(1.5, 2.3, 3.7, 3.7, 4.0, 5))

---

Sample dataset from the EFC Survey

Description

Selected variables from the EUROFAMCARE survey. Useful when testing on "real-life" data sets, including random missing values. This data set also has value and variable label attributes.

---

Find or get columns in a data frame based on search patterns

Description

find_columns() returns column names from a data set that match a certain search pattern, while get_columns() returns the found data. data_select() is an alias for get_columns(), and data_find() is an alias for find_columns().

Usage

find_columns(
  data,
  select = NULL,
  exclude = NULL,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = TRUE,
  ...
)

data_find(
  data,
  select = NULL,
  exclude = NULL,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = TRUE,
  ...
)
find_columns

get_columns(
  data, 
  select = NULL, 
  exclude = NULL, 
  ignore_case = FALSE, 
  regex = FALSE, 
  verbose = TRUE, 
  ...
)

data_select(
  data, 
  select = NULL, 
  exclude = NULL, 
  ignore_case = FALSE, 
  regex = FALSE, 
  verbose = TRUE, 
  ...
)

Arguments

data A data frame.

select Variables that will be included when performing the required tasks. Can be either

• a variable specified as a literal variable name (e.g., column_name),
• a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
• a formula with variable names (e.g., ~column_1 + column_2),
• a vector of positive integers, giving the positions counting from the left (e.g. 1 or c(1, 3, 5)),
• a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
• one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal").
• or a function testing for logical conditions, e.g. is.numeric() (or is.numeric), or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
• ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. -ends_with(""), -is.numeric or -(Sepal.Width:Petal.Length). Note: Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.
find_columns

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

exclude
See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

ignore_case
Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

regex
Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains(""), or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.

verbose
Toggle warnings.

... Arguments passed down to other functions. Mostly not used yet.

Details

Note that it is possible to either pass an entire select helper or only the pattern inside a select helper as a function argument:

foo <- function(data, pattern) {
  find_columns(data, select = starts_with(pattern))
}
foo(iris, pattern = "Sep")

foo2 <- function(data, pattern) {
  find_columns(data, select = pattern)
}
foo2(iris, pattern = starts_with("Sep"))

This means that it is also possible to use loop values as arguments or patterns:

for (i in c("Sepal", "Sp")) {
  head(iris) |> find_columns(select = starts_with(i)) |>
  print()
}

However, this behavior is limited to a "single-level function". It will not work in nested functions, like below:

inner <- function(data, arg) {
  find_columns(data, select = arg)
}
outer <- function(data, arg) {
    inner(data, starts_with(arg))
}  
outer(iris, "Sep")

In this case, it is better to pass the whole select helper as the argument of outer():

outer <- function(data, arg) {
    inner(data, arg)
}  
outer(iris, starts_with("Sep"))

Value

find_columns() returns a character vector with column names that matched the pattern in select and exclude, or NULL if no matching column name was found. get_columns() returns a data frame with matching columns.

See Also

• Functions to rename stuff: data_rename(), data_rename_rows(), data_addprefix(), data_addsuffix()
• Functions to reorder or remove columns: data_reorder(), data_relocate(), data_remove()
• Functions to reshape, pivot or rotate data frames: data_to_long(), data_to_wide(), data_rotate()
• Functions to recode data: rescale(), reverse(), categorize(), recode_values(), slide()
• Functions to standardize, normalize, rank-transform: center(), standardize(), normalize(), ranktransform(), winsorize()
• Split and merge data frames: data_partition(), data_merge()
• Functions to find or select columns: data_select(), data_find()
• Functions to filter rows: data_match(), data_filter()

Examples

# Find columns names by pattern
find_columns(iris, starts_with("Sepal"))
find_columns(iris, ends_with("Width"))
find_columns(iris, regex("\\.\""))
find_columns(iris, c("Petal.Width", "Sepal.Length"))

# starts with "Sepal", but not allowed to end with "width"
find_columns(iris, starts_with("Sepal"), exclude = contains("Width"))

# find numeric with mean > 3.5
numeric_mean_35 <- function(x) is.numeric(x) & mean(x, na.rm = TRUE) > 3.5
find_columns(iris, numeric_mean_35)
labels_to_levels

Convert value labels into factor levels

Description

Convert value labels into factor levels

Usage

labels_to_levels(x, ...)

## S3 method for class 'factor'
labels_to_levels(x, verbose = TRUE, ...)

## S3 method for class 'data.frame'
labels_to_levels(
  x,
  select = NULL,
  exclude = NULL,
  ignore_case = FALSE,
  append = FALSE,
  regex = FALSE,
  verbose = TRUE,
  ...
)

Arguments

- **x**
  - A data frame or factor. Other variable types (e.g. numerics) are not allowed.
- **...**
  - Currently not used.
- **verbose**
  - Toggle warnings.
- **select**
  - Variables that will be included when performing the required tasks. Can be either
    - a variable specified as a literal variable name (e.g., column_name),
    - a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
    - a formula with variable names (e.g., ~column_1 + column_2),
    - a vector of positive integers, giving the positions counting from the left (e.g. 1 or c(1, 3, 5)),
    - a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
    - one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal").
• or a function testing for logical conditions, e.g. `is.numeric()` (or `is.numeric`), or any user-defined function that selects the variables for which the function returns TRUE (like: `foo <- function(x) mean(x) > 3`),

• ranges specified via literal variable names, select-helpers (except `regex()`) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a `-`, e.g. `-ends_with(""`), `-is.numeric` or `-Sepal.Width:Petal.Length). **Note:** Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, `select=-ends_with("Length")` (with `-`) is equivalent to `exclude=ends_with("Length")` (no `-`). In case negation should not work as expected, use the exclude argument instead.

If `NULL`, selects all columns. Patterns that found no matches are silently ignored, e.g. `find_columns(iris, select = c("Species", "Test"))` will just return "Species".

### exclude
See `select`, however, column names matched by the pattern from exclude will be excluded instead of selected. If `NULL` (the default), excludes no columns.

### ignore_case
Logical, if TRUE and when one of the select-helpers or a regular expression is used in `select`, ignores lower/upper case in the search pattern when matching against variable names.

### append
Logical or string. If TRUE, recoded or converted variables get new column names and are appended (column bind) to `x`, thus returning both the original and the recoded variables. The new columns get a suffix, based on the calling function: "_r" for `recode` functions, "_n" for `to_numeric()`, "_f" for `to_factor()`, or "_s" for `slide()`. If append=FALSE, original variables in `x` will be overwritten by their recoded versions. If a character value, recoded variables are appended with new column names (using the defined suffix) to the original data frame.

### regex
Logical, if TRUE, the search pattern from `select` will be treated as regular expression. When `regex = TRUE`, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. `regex = TRUE` is comparable to using one of the two select-helpers, `select = contains("")` or `select = regex\\("")`, however, since the select-helpers may not work when called from inside other functions (see ‘Details’), this argument may be used as workaround.

### Details

`labels_to_levels()` allows to use value labels of factors as their levels.

### Value

`x`, where for all factors former levels are replaced by their value labels.

### Examples

```r
data(efc)
# create factor
x <- as.factor(efc$c172code)
# add value labels - these are not factor levels yet
```
```r
x <- assign_labels(x, values = c(`1` = "low", `2` = "mid", `3` = "high"))
levels(x)
data_tabulate(x)

x <- labels_to_levels(x)
levels(x)
data_tabulate(x)
```

---

**makepredictcall.dw_transformer**

*Utility Function for Safe Prediction with datawizard transformers*

### Description

This function allows for the use of (some of) datawizard’s transformers inside a model formula. See examples below.

Currently, `center()`, `standardize()`, `normalize()`, & `rescale()` are supported.

### Usage

```r
## S3 method for class 'dw_transformer'
makepredictcall(var, call)
```

### Arguments

- `var`: A variable.
- `call`: The term in the formula, as a call.

### Value

A replacement for `call` for the `predvars` attribute of the terms.

### See Also

`stats::makepredictcall()`

### Examples

```r
data("mtcars")
train <- mtcars[1:30, ]

m1 <- lm(mpg ~ center(hp), data = train)
predict(m1, newdata = test) # Data is "centered" before the prediction is made,
# according to the center of the old data

m2 <- lm(mpg ~ standardize(hp), data = train)
```
m3 <- lm(mpg ~ scale(hp), data = train) # same as above
predict(m2, newdata = test) # Data is "standardized" before the prediction is made.
predict(m3, newdata = test) # Data is "standardized" before the prediction is made.

m4 <- lm(mpg ~ normalize(hp), data = mtcars)
m5 <- lm(mpg ~ rescale(hp, to = c(-3, 3)), data = mtcars)

(newdata <- data.frame(hp = c(range(mtcars$hp), 400))) # 400 is outside original range!

model.frame(delete.response(terms(m4)), data = newdata)
model.frame(delete.response(terms(m5)), data = newdata)

---

### mean_sd

**Summary Helpers**

**Description**

Summary Helpers

**Usage**

```r
mean_sd(x, times = 1L, remove_na = TRUE, named = TRUE, na.rm = TRUE, ...)
```

```r
median_mad(
  x,
  times = 1L,
  remove_na = TRUE,
  constant = 1.4826,
  named = TRUE,
  na.rm = TRUE,
  ...
)
```

**Arguments**

- `x` A numeric vector (or one that can be coerced to one via `as.numeric()`) to be summarized.
- `times` How many SDs above and below the Mean (or MADs around the Median)
- `remove_na` Logical. Should NA values be removed before computing (TRUE) or not (FALSE, default)?
- `named` Should the vector be named? (E.g., c("-SD" = -1, Mean = 1, "+SD" = 2).)
- `na.rm` Deprecated. Please use `remove_na` instead.
- `...` Not used.
- `constant` scale factor.
Value

A (possibly named) numeric vector of length \(2 \times \text{times} + 1\) of SDs below the mean, the mean, and SDs above the mean (or median and MAD).

Examples

```
mean_sd(mtcars$mpg)
mean_sd(mtcars$mpg, times = 2L)
median_mad(mtcars$mpg)
```

---

nhanes_sample

Sample dataset from the National Health and Nutrition Examination Survey

---

Description

Selected variables from the National Health and Nutrition Examination Survey that are used in the example from Lumley (2010), Appendix E.

References


---

normalize

Normalize numeric variable to 0-1 range

Description

Performs a normalization of data, i.e., it scales variables in the range 0 - 1. This is a special case of `rescale()`. `unnormalize()` is the counterpart, but only works for variables that have been normalized with `normalize()`.

Usage

```
normalize(x, ...)
```

`## S3 method for class 'numeric'`
```
normalize(x, include_bounds = TRUE, verbose = TRUE, ...)
```

`## S3 method for class 'data.frame'`
```
normalize(
  x,
  select = NULL,
)```
Arguments

x A numeric vector, (grouped) data frame, or matrix. See 'Details'.

... Arguments passed to or from other methods.

include_bounds Numeric or logical. Using this can be useful in case of beta-regression, where the response variable is not allowed to include zeros and ones. If TRUE, the input is normalized to a range that includes zero and one. If FALSE, the return value is compressed, using Smithson and Verkuilen's (2006) formula \((x * (n - 1) + 0.5) / n\), to avoid zeros and ones in the normalized variables. Else, if numeric (e.g., 0.001), include_bounds defines the "distance" to the lower and upper bound, i.e. the normalized vectors are rescaled to a range from 0 + include_bounds to 1 - include_bounds.

verbose Toggle warnings and messages on or off.

select Variables that will be included when performing the required tasks. Can be either

• a variable specified as a literal variable name (e.g., column_name),
• a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
• a formula with variable names (e.g., ~column_1 + column_2),
• a vector of positive integers, giving the positions counting from the left (e.g. 1 or c(1, 3, 5)),
**normalize**

- a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
- one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal")
- or a function testing for logical conditions, e.g. is.numeric() or is.numeric, or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
- ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. -ends_with(""), -is.numeric or -(Sepal.Width:Petal.Length). **Note:** Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

**exclude**
See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

**append**
Logical or string. If TRUE, standardized variables get new column names (with the suffix ":z") and are appended (column bind) to x, thus returning both the original and the standardized variables. If FALSE, original variables in x will be overwritten by their standardized versions. If a character value, standardized variables are appended with new column names (using the defined suffix) to the original data frame.

**ignore_case**
Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

**regex**
Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains(""") or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.

**Details**

- If x is a matrix, normalization is performed across all values (not column- or row-wise). For column-wise normalization, convert the matrix to a data.frame.
- If x is a grouped data frame (grouped_df), normalization is performed separately for each group.

**Value**

A normalized object.
Selection of variables - the `select` argument

For most functions that have a `select` argument (including this function), the complete input data frame is returned, even when `select` only selects a range of variables. That is, the function is only applied to those variables that have a match in `select`, while all other variables remain unchanged. In other words: for this function, `select` will not omit any non-included variables, so that the returned data frame will include all variables from the input data frame.

References


See Also

See `makepredictcall.dw_transformer()` for use in model formulas.

Other transform utilities: `ranktransform()`, `rescale()`, `reverse()`, `standardize()`

Examples

```r
normalize(c(0, 1, 5, -5, -2))
normalize(c(0, 1, 5, -5, -2), include.bounds = FALSE)
# use a value defining the bounds
normalize(c(0, 1, 5, -5, -2), include.bounds = .001)

head(normalize(trees))
```

---

**ranktransform**

(Signed) rank transformation

**Description**

Transform numeric values with the integers of their rank (i.e., 1st smallest, 2nd smallest, 3rd smallest, etc.). Setting the `sign` argument to `TRUE` will give you signed ranks, where the ranking is done according to absolute size but where the sign is preserved (i.e., 2, 1, -3, 4).

**Usage**

```r
ranktransform(x, ...)
```

## S3 method for class 'numeric'
```r
ranktransform(x, sign = FALSE, method = "average", verbose = TRUE, ...)
```

## S3 method for class 'data.frame'
```r
ranktransform(
  x,
```
ranktransform

select = NULL,
exclude = NULL,
sign = FALSE,
method = "average",
ignore_case = FALSE,
regex = FALSE,
verbose = TRUE,
...
)

Arguments

x

Object.

... Arguments passed to or from other methods.

sign Logical, if TRUE, return signed ranks.

method Treatment of ties. Can be one of "average" (default), "first", "last", "random", "max" or "min". See rank() for details.

verbose Toggle warnings.

select Variables that will be included when performing the required tasks. Can be either

• a variable specified as a literal variable name (e.g., column_name),
• a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
• a formula with variable names (e.g., ~column_1 + column_2),
• a vector of positive integers, giving the positions counting from the left (e.g., 1 or c(1, 3, 5)),
• a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
• one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""). starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal").
• or a function testing for logical conditions, e.g. is.numeric() (or is.numeric),
   or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
• ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. ~ends_with(""), ~is.numeric or ~(Sepal.Width:Petal.Length). Note: Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".
For most functions that have a select argument (including this function), the complete input data frame is returned, even when select only selects a range of variables. That is, the function is only applied to those variables that have a match in select, while all other variables remain unchanged. In other words: for this function, select will not omit any non-included variables, so that the returned data frame will include all variables from the input data frame.

See Also
Other transform utilities: normalize(), rescale(), reverse(), standardize()

Examples
ranktransform(c(0, 1, 5, -5, -2))
# Won't work
# ranktransform(c(0, 1, 5, -5, -2), sign = TRUE)
head(ranktransform(trees))

recode_into
Recode values from one or more variables into a new variable

Description
This function recodes values from one or more variables into a new variable. It is a convenient function to avoid nested ifelse() statements, which is similar to dplyr::case_when().
Usage
recode_into(..., data = NULL, default = NA, verbose = TRUE)

Arguments

... A sequence of two-sided formulas, where the left hand side (LHS) is a logical
matching condition that determines which values match this case. The LHS of
this formula is also called "recode pattern" (e.g., in messages). The right hand
side (RHS) indicates the replacement value.

data Optional, name of a data frame. This can be used to avoid writing the data name
multiple times in .... See 'Examples'.

default Indicates the default value that is chosen when no match in the formulas in ... is
found. If not provided, NA is used as default value.

verbose Toggle warnings.

Value
A vector with recoded values.

Examples

```r
x <- 1:30
recode_into(
  x > 15 ~ "a",
  x > 10 & x <= 15 ~ "b",
  default = "c"
)
```

```r
set.seed(123)
d <- data.frame(
  x = sample(1:5, 30, TRUE),
  y = sample(letters[1:5], 30, TRUE),
  stringsAsFactors = FALSE
)

# from different variables into new vector
recode_into(
  d$x %in% 1:3 & d$y %in% c("a", "b") ~ 1,
  d$x > 3 ~ 2,
  default = 0
)
```

```r
# no need to write name of data frame each time
recode_into(
  x %in% 1:3 & y %in% c("a", "b") ~ 1,
  x > 3 ~ 2,
  data = d,
  default = 0
)
```
recode_values

Recode old values of variables into new values

Description

This function recodes old values into new values and can be used to recode numeric or character vectors, or factors.

Usage

recode_values(x, ...)

## S3 method for class 'numeric'
recode_values(
  x,
  recode = NULL,
  default = NULL,
  preserve_na = TRUE,
  verbose = TRUE,
  ...
)

## S3 method for class 'data.frame'
recode_values(
  x,
  select = NULL,
  exclude = NULL,
  recode = NULL,
  default = NULL,
  preserve_na = TRUE,
  append = FALSE,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = TRUE,
  ...
)

change_code(x, ...)

Arguments

x
  A data frame, numeric or character vector, or factor.

...  
  not used.

recode  
  A list of named vectors, which indicate the recode pairs. The names of the list-elements (i.e. the left-hand side) represent the new values, while the values of the list-elements indicate the original (old) values that should be replaced. When recoding numeric vectors, element names have to be surrounded in backticks. For
example, recode=list(`0`=1) would recode all 1 into 0 in a numeric vector. See also 'Examples' and 'Details'.

default
Defines the default value for all values that have no match in the recode-pairs. Note that, if preserve_na=FALSE, missing values (NA) are also captured by the default argument, and thus will also be recoded into the specified value. See 'Examples' and 'Details'.

preserve_na
Logical, if TRUE, NA (missing values) are preserved. This overrides any other arguments, including default. Hence, if preserve_na=TRUE, default will no longer convert NA into the specified default value.

verbose
Toggle warnings.

select
Variables that will be included when performing the required tasks. Can be either
  • a variable specified as a literal variable name (e.g., column_name),
  • a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
  • a formula with variable names (e.g., ~column_1 + column_2),
  • a vector of positive integers, giving the positions counting from the left (e.g., 1 or c(1, 3, 5)),
  • a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
  • one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g. starts_with("Sep", "Petal"),
  • or a function testing for logical conditions, e.g. is.numeric() (or is.numeric), or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
  • ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. ~ends_with(""), ~is.numeric or ~(Sepal.Width:Petal.Length). Note: Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=~ends_with("Length") (with ~) is equivalent to exclude=ends_with("Length") (no ~). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

exclude
See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

append
Logical or string. If TRUE, recoded or converted variables get new column names and are appended (column bind) to x, thus returning both the original and the recoded variables. The new columns get a suffix, based on the calling function: 
  • _r" for recode functions, "_n" for to_numeric(). "_f" for to_factor(). or "_s" for slide(). If append=FALSE, original variables in x will be overwritten by their recoded versions. If a character value, recoded variables are appended with new column names (using the defined suffix) to the original data frame.
ignore_case

Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

regex

Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains("") or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.

Details

This section describes the pattern of the recode arguments, which also provides some shortcuts, in particular when recoding numeric values.

- Single values
  Single values either need to be wrapped in backticks (in case of numeric values) or "as is" (for character or factor levels). Example: recode=list(`0` = 1, `1` = 2) would recode 1 into 0, and 2 into 1. For factors or character vectors, an example is: recode=list(x="a", y="b") (recode "a" into "x" and "b" into "y").

- Multiple values
  Multiple values that should be recoded into a new value can be separated with comma. Example: recode=list(`1` = c(1, 4), `2` = c(2, 3)) would recode the values 1 and 4 into 1, and 2 and 3 into 2. It is also possible to define the old values as a character string, like: recode=list(`1` = "1, 4", `2` = "2, 3") For factors or character vectors, an example is: recode=list(x=c("a", "b"), y = c("c", "d"))

- Value range
  Numeric value ranges can be defined using the \:. Example: recode=list(`1` = 1:3, `2` = 4:6) would recode all values from 1 to 3 into 1, and 4 to 6 into 2.

- min and max
  placeholder to use the minimum or maximum value of the (numeric) variable. Useful, e.g., when recoding ranges of values. Example: recode=list(`1` = "min:10", `2` = "11:max")

- default values
  The default argument defines the default value for all values that have no match in the recode-pairs. For example, recode=list(`1` = c(1,2), `2` = c(3,4)), default=9 would recode values 1 and 2 into 1, 3 and 4 into 2, and all other values into 9. If preserve_na is set to FALSE, NA (missing values) will also be recoded into the specified default value.

- Reversing and rescaling
  See reverse() and rescale().

Value

x, where old values are replaced by new values.
Selection of variables - the select argument

For most functions that have a select argument (including this function), the complete input data frame is returned, even when select only selects a range of variables. That is, the function is only applied to those variables that have a match in select, while all other variables remain unchanged. In other words: for this function, select will not omit any non-included variables, so that the returned data frame will include all variables from the input data frame.

Note

You can use options(data_recode_pattern = "old=new") to switch the behaviour of the recode-argument, i.e. recode-pairs are now following the pattern old values = new values, e.g. ifgetOption("data_recode_pattern") is set to "old=new", then recode(\'1\'=0) would recode all 1 into 0. The default for recode(\'1\'=0) is to recode all 0 into 1.

See Also

• Functions to rename stuff: data_rename(), data_rename_rows(), data_addprefix(), data_addsuffix()
• Functions to reorder or remove columns: data_reorder(), data_relocate(), data_remove()
• Functions to reshape, pivot or rotate data frames: data_to_long(), data_to_wide(), data_rotate()
• Functions to recode data: rescale(), reverse(), categorize(), recode_values(), slide()
• Functions to standardize, normalize, rank-transform: center(), standardize(), normalize(), ranktransform(), winsorize()
• Split and merge data frames: data_partition(), data_merge()
• Functions to find or select columns: data_select(), data_find()
• Functions to filter rows: data_match(), data_filter()

Examples

# numeric ---------
set.seed(123)
x <- sample(c(1:4, NA), 15, TRUE)
table(x, useNA = "always")

out <- recode_values(x, list(c(0) = 1, c(1) = 2:3, c(2) = 4))
table(out, useNA = "always")

# to recode NA values, set preserve_na to FALSE
out <- recode_values(x, list(c(0) = 1, c(1) = 2:3, c(2) = 4, c(9) = NA), preserve_na = FALSE)
table(out, useNA = "always")

# preserve na --------
out <- recode_values(x, list(c(0) = 1, c(1) = 2:3), default = 77)
out

table(out, useNA = "always")

# recode na into default ---------
out <- recode_values(
  x,
  list("0" = 1, "1" = 2:3),
  default = 77,
  preserve_na = FALSE
)
out

table(out, useNA = "always")

# factors (character vectors are similar) --------
set.seed(123)
x <- as.factor(sample(c("a", "b", "c"), 15, TRUE))
table(x)

out <- recode_values(x, list(x = "a", y = c("b", "c")))
out

table(out)

out <- recode_values(x, list(x = "a", y = "b", z = "c"))
out

table(out)

out <- recode_values(x, list(y = "b,c"), default = 77)
# same as
# recode_values(x, list(y = c("b", "c")), default = 77)
out

table(out)

# data frames -------
set.seed(123)
d <- data.frame(
  x = sample(c(1:4, NA), 12, TRUE),
  y = as.factor(sample(c("a", "b", "c"), 12, TRUE)),
  stringsAsFactors = FALSE
)
recode_values(
  d,
  recode = list("0" = 1, "1" = 2:3, "2" = 4, x = "a", y = c("b", "c")),
  append = TRUE
)

# switch recode pattern to "old=new" --------
options(data_recode_pattern = "old=new")

# numeric
```
set.seed(123)
x <- sample(c(1:4, NA), 15, TRUE)
table(x, useNA = "always")

out <- recode_values(x, list('1' = 0, '2:3' = 1, '4' = 2))
table(out, useNA = "always")

# factors (character vectors are similar)
set.seed(123)
x <- as.factor(sample(c("a", "b", "c"), 15, TRUE))
table(x)

out <- recode_values(x, list(a = "x", 'b', 'c' = "y"))
table(out)

# reset options
options(data_recode_pattern = NULL)
```

---

**remove_empty**  
*Return or remove variables or observations that are completely missing*

**Description**

These functions check which rows or columns of a data frame completely contain missing values, i.e. which observations or variables completely have missing values, and either (1) returns their indices; or (2) removes them from the data frame.

**Usage**

```r
empty_columns(x)
empty_rows(x)
remove_empty_columns(x)
remove_empty_rows(x)
remove_empty(x)
```

**Arguments**

- `x`  
  A data frame.

**Details**

For character vectors, empty string values (i.e. "") are also considered as missing value. Thus, if a character vector only contains NA and """, it is considered as empty variable and will be removed. Same applies
**Value**

- For `empty_columns()` and `empty_rows()`, a numeric (named) vector with row or column indices of those variables that completely have missing values.
- For `remove_empty_columns()` and `remove_empty_rows()`, a data frame with "empty" columns or rows removed, respectively.
- For `remove_empty()`, both empty rows and columns will be removed.

**Examples**

```r
tmp <- data.frame(
  a = c(1, 2, 3, NA, 5),
  b = c(1, NA, 3, NA, 5),
  c = c(NA, NA, NA, NA, NA),
  d = c(1, NA, 3, NA, 5)
)

tmp
```

# indices of empty columns or rows
empty_columns(tmp)
empty_rows(tmp)

# remove empty columns or rows
remove_empty_columns(tmp)
remove_empty_rows(tmp)

# remove empty columns and rows
remove_empty(tmp)

# also remove "empty" character vectors
```r
tmp <- data.frame(
  a = c(1, 2, 3, NA, 5),
  b = c(1, NA, 3, NA, 5),
  c = c("", ",", ",", ",", "),
  stringsAsFactors = FALSE
)
```

**replace_nan_inf**

Convert infinite or NaN values into NA

**Description**

Replaces all infinite (Inf and -Inf) or NaN values with NA.

**Usage**

`replace_nan_inf(x, ...)`
Arguments

x A vector or a dataframe

Value

Data with Inf, -Inf, and NaN converted to NA.

Examples

# a vector
x <- c(1, 2, NA, 3, NaN, 4, NA, 5, Inf, -Inf, 6, 7)
replace_nan_inf(x)

# a data frame
df <- data.frame(
  x = c(1, NA, 5, Inf, 2, NA),
  y = c(3, NaN, 4, -Inf, 6, 7),
  stringsAsFactors = FALSE
)
replace_nan_inf(df)

rescale

Rescale Variables to a New Range

Description

Rescale variables to a new range. Can also be used to reverse-score variables (change the keying/scoring direction).

Usage

rescale(x, ...)

calculate_scale(x, ...)

## S3 method for class 'numeric'
rescale(x, to = c(0, 100), range = NULL, verbose = TRUE, ...)

## S3 method for class 'data.frame'
rescale(
  x,
  select = NULL,
  exclude = NULL,
  to = c(0, 100),
  range = NULL,
  append = FALSE,
)
ignore_case = FALSE,
regex = FALSE,
verbose = FALSE,
...
)

Arguments

Arguments passed to or from other methods.

x
A (grouped) data frame, numeric vector or factor.

... to
Numeric vector of length 2 giving the new range that the variable will have after rescaling. To reverse-score a variable, the range should be given with the maximum value first. See examples.

range
Initial (old) range of values. If NULL, will take the range of the input vector (range(x)).

verbose
Toggle warnings.

select
Variables that will be included when performing the required tasks. Can be either

- a variable specified as a literal variable name (e.g., column_name),
- a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
- a formula with variable names (e.g., ~column_1 + column_2),
- a vector of positive integers, giving the positions counting from the left (e.g., 1 or c(1, 3, 5)),
- a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
- one of the following select-helpers: starts_with(), ends_with(), contains(), or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g. starts_with("Sep", "Petal").
- or a function testing for logical conditions, e.g. is.numeric() (or is.numeric), or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
- ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. -ends_with(""), -is.numeric or -(Sepal.Width:Petal.Length). **Note:** Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

exclude
See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.
Rescale variables

**Append**
Logical or string. If TRUE, recoded or converted variables get new column names and are appended (column bind) to x, thus returning both the original and the recoded variables. The new columns get a suffix, based on the calling function: 
"_r" for recode functions, "_n" for to_numeric(), "_f" for to_factor(), or "_s" for slide(). If append=FALSE, original variables in x will be overwritten by their recoded versions. If a character value, recoded variables are appended with new column names (using the defined suffix) to the original data frame.

**Ignore Case**
Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

**Regex**
Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains(""), or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.

**Value**
A rescaled object.

**Selection of variables - the select argument**
For most functions that have a select argument (including this function), the complete input data frame is returned, even when select only selects a range of variables. That is, the function is only applied to those variables that have a match in select, while all other variables remain unchanged. In other words: for this function, select will not omit any non-included variables, so that the returned data frame will include all variables from the input data frame.

**See Also**
See `makepredictcall.dw_transformer()` for use in model formulas.
Other transform utilities: `normalize()`, `ranktransform()`, `reverse()`, `standardize()`

**Examples**
```
rescale(c(0, 1, 5, -5, -2))
rescale(c(0, 1, 5, -5, -2), to = c(-5, 5))
rescale(c(1, 2, 3, 4, 5), to = c(-2, 2))

# Specify the "theoretical" range of the input vector
rescale(c(1, 3, 4), to = c(0, 40), range = c(0, 4))

# Reverse-score a variable
rescale(c(1, 2, 3, 4, 5), to = c(5, 1))
rescale(c(1, 2, 3, 4, 5), to = c(2, -2))

# Data frames
```
head(rescale(iris, to = c(0, 1)))
head(rescale(iris, to = c(0, 1), select = "Sepal.Length"))

# One can specify a list of ranges
head(rescale(iris, to = list(
  "Sepal.Length" = c(0, 1),
  "Petal.Length" = c(-1, 0)
)))

---

### rescale_weights

**Rescale design weights for multilevel analysis**

**Description**

Most functions to fit multilevel and mixed effects models only allow to specify frequency weights, but not design (i.e. sampling or probability) weights, which should be used when analyzing complex samples and survey data. `rescale_weights()` implements an algorithm proposed by Asparouhov (2006) and Carle (2009) to rescale design weights in survey data to account for the grouping structure of multilevel models, which then can be used for multilevel modelling.

**Usage**

```r
rescale_weights(data, group, probability_weights, nest = FALSE)
```

**Arguments**

- `data`: A data frame.
- `group`: Variable names (as character vector, or as formula), indicating the grouping structure (strata) of the survey data (level-2-cluster variable). It is also possible to create weights for multiple group variables; in such cases, each created weighting variable will be suffixed by the name of the group variable.
- `probability_weights`: Variable indicating the probability (design or sampling) weights of the survey data (level-1-weight).
- `nest`: Logical, if TRUE and group indicates at least two group variables, then groups are "nested", i.e. groups are now a combination from each group level of the variables in group.

**Details**

Rescaling is based on two methods: For `pweights_a`, the sample weights `probability_weights` are adjusted by a factor that represents the proportion of group size divided by the sum of sampling weights within each group. The adjustment factor for `pweights_b` is the sum of sample weights within each group divided by the sum of squared sample weights within each group (see Carle (2009), Appendix B). In other words, `pweights_a` "scales the weights so that the new weights sum to the cluster sample size" while `pweights_b" scales the weights so that the new weights sum to the effective cluster size".
Regarding the choice between scaling methods A and B, Carle suggests that "analysts who wish to discuss point estimates should report results based on weighting method A. For analysts more interested in residual between-group variance, method B may generally provide the least biased estimates". In general, it is recommended to fit a non-weighted model and weighted models with both scaling methods and when comparing the models, see whether the "inferential decisions converge", to gain confidence in the results.

Though the bias of scaled weights decreases with increasing group size, method A is preferred when insufficient or low group size is a concern.

The group ID and probably PSU may be used as random effects (e.g. nested design, or group and PSU as varying intercepts), depending on the survey design that should be mimicked.

**Value**

data, including the new weighting variables: pweights_a and pweights_b, which represent the rescaled design weights to use in multilevel models (use these variables for the weights argument).

**References**


**Examples**

```r
if (require("lme4")) {
  data(nhanes_sample)
  head(rescale_weights(nhanes_sample, "SDMVSTRA", "WTINT2YR"))
  # also works with multiple group-variables
  head(rescale_weights(nhanes_sample, c("SDMVSTRA", "SDMVPSU"), "WTINT2YR"))
  # or nested structures.
  x <- rescale_weights(
    data = nhanes_sample,
    group = c("SDMVSTRA", "SDMVPSU"),
    probability_weights = "WTINT2YR",
    nest = TRUE
  )
  head(x)

  nhanes_sample <- rescale_weights(nhanes_sample, "SDMVSTRA", "WTINT2YR")

  glmer(
    total ~ factor(RIAGENDR) * (log(age) + factor(RIDRETH1)) + (1 | SDMVPSU),
    family = poisson(),
    data = nhanes_sample,
    weights = pweights_a
  )
}
```
reshape_ci

Reshape CI between wide/long formats

Description

Reshape CI between wide/long formats.

Usage

reshape_ci(x, ci_type = "CI")

Arguments

x
A data frame containing columns named CI_low and CI_high (or similar, see ci_type).

 ci_type
String indicating the "type" (i.e. prefix) of the interval columns. Per easystats convention, confidence or credible intervals are named CI_low and CI_high, and the related ci_type would be "CI". If column names for other intervals differ, ci_type can be used to indicate the name, e.g. ci_type = "SI" can be used for support intervals, where the column names in the data frame would be SI_low and SI_high.

Value

A data frame with columns corresponding to confidence intervals reshaped either to wide or long format.

Examples

```r
x <- data.frame(
  Parameter = c("Term 1", "Term 2", "Term 1", "Term 2"),
  CI = c(.8, .8, .9, .9),
  CI_low = c(.2, .3, .1, .15),
  CI_high = c(.5, .6, .8, .85),
  stringsAsFactors = FALSE
)

reshape_ci(x)
reshape_ci(reshape_ci(x))
```
**Description**

Reverse-score variables (change the keying/scoring direction).

**Usage**

```r
reverse(x, ...) reverse_scale(x, ...)
```

```r
## S3 method for class 'numeric'
reverse(x, range = NULL, verbose = TRUE, ...)
```

```r
## S3 method for class 'data.frame'
reverse(
  x,
  select = NULL,
  exclude = NULL,
  range = NULL,
  append = FALSE,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = FALSE,
  ...
)
```

**Arguments**

- `x` A (grouped) data frame, numeric vector or factor.
- `...` Arguments passed to or from other methods.
- `range` Range of values that is used as reference for reversing the scale. For numeric variables, can be `NULL` or a numeric vector of length two, indicating the lowest and highest value of the reference range. If `NULL`, will take the range of the input vector (`range(x)`). For factors, range can be `NULL`, a numeric vector of length two, or a (numeric) vector of at least the same length as factor levels (i.e. must be equal to or larger than `nlevels(x)`). Note that providing a range for factors usually only makes sense when factor levels are numeric, not characters.
- `verbose` Toggle warnings.
- `select` Variables that will be included when performing the required tasks. Can be either
  - a variable specified as a literal variable name (e.g., `column_name`),
  - a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., `c("col1", "col2", "col3")`).
• a formula with variable names (e.g., \(-\text{column}_1 + \text{column}_2\)),
• a vector of positive integers, giving the positions counting from the left (e.g. 1 or c(1, 3, 5)),
• a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
• one of the following select-helpers: starts_with(), ends_with(), contains(),
  a range using : or \text{regex("")}, starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal").
• or a function testing for logical conditions, e.g. is.numeric() (or is.numeric),
  or any user-defined function that selects the variables for which the function returns TRUE (like: \text{foo <- function(x) mean(x) > 3}),
• ranges specified via literal variable names, select-helpers (except \text{regex()}) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. -ends_with(""), -is.numeric or -\text{(Sepal.Width:Petal.Length)}. \textbf{Note:} Negation means that matches are \textit{excluded}, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. \text{find_columns(iris, select = c("Species", "Test"))} will just return "Species".

\textbf{exclude} \hspace{2cm} See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

\textbf{append} \hspace{2cm} Logical or string. If TRUE, recoded or converted variables get new column names and are appended (column bind) to \text{x}, thus returning both the original and the recoded variables. The new columns get a suffix, based on the calling function: "_r" for recode functions, "_n" for \text{to_numeric()}, "_f" for \text{to_factor()}, or "_s" for \text{slide()}. If append=FALSE, original variables in \text{x} will be overwritten by their recoded versions. If a character value, recoded variables are appended with new column names (using the defined suffix) to the original data frame.

\textbf{ignore_case} \hspace{2cm} Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

\textbf{regex} \hspace{2cm} Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains("") or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.

\textbf{Value} 

A reverse-scored object.
Selection of variables - the select argument

For most functions that have a select argument (including this function), the complete input data frame is returned, even when select only selects a range of variables. That is, the function is only applied to those variables that have a match in select, while all other variables remain unchanged. In other words: for this function, select will not omit any non-included variables, so that the returned data frame will include all variables from the input data frame.

See Also

Other transform utilities: `normalize()`, `ranktransform()`, `rescale()`, `standardize()`

Examples

```r
reverse(c(1, 2, 3, 4, 5))
reverse(c(-2, -1, 0, 2, 1))

# Specify the "theoretical" range of the input vector
reverse(c(1, 3, 4), range = c(0, 4))

# Factor variables
reverse(factor(c(1, 2, 3, 4, 5)))
reverse(factor(c(1, 2, 3, 4, 5)), range = 0:10)

# Data frames
head(reverse(iris))
head(reverse(iris, select = "Sepal.Length"))
```

rownames_as_column  
Tools for working with row names or row ids

Description

Tools for working with row names or row ids

Usage

```r
rownames_as_column(x, var = "rowname")

column_as_rownames(x, var = "rowname")

rowid_as_column(x, var = "rowid")
```

Arguments

- `x`: A data frame.
- `var`: Name of column to use for row names/ids. For `column_as_rownames()`, this argument can be the variable name or the column number. For `rownames_as_column()` and `rowid_as_column()`, the column name must not already exist in the data.
Details

These are similar to tibble's functions column_to_rownames(), rownames_to_column() and rowid_to_column(). Note that the behavior of rowid_as_column() is different for grouped dataframe: instead of making the rowid unique across the full dataframe, it creates rowid per group. Therefore, there can be several rows with the same rowid if they belong to different groups.

If you are familiar with dplyr, this is similar to doing the following:

```
data |>
  group_by(grp) |>
  mutate(id = row_number()) |>
  ungroup()
```

Value

A data frame.

Examples

# Convert between row names and column --------------------------------
test <- rownames_as_column(mtcars, var = "car")
test
test_data <- head(iris)
rowid_as_column(test_data)
rowid_as_column(test_data, var = "my_id")

---

row_to_colnames | Tools for working with column names

Description

Tools for working with column names

Usage

```
row_to_colnames(x, row = 1, na_prefix = "x", verbose = TRUE)
```

```
colnames_to_row(x, prefix = "x")
```

Arguments

- **x**: A data frame.
- **row**: Row to use as column names.
- **na_prefix**: Prefix to give to the column name if the row has an NA. Default is 'x', and it will be incremented at each NA (x1, x2, etc.).
skewness

verbose        Toggle warnings.
prefix        Prefix to give to the column name. Default is 'x', and it will be incremented at
              each column (x1, x2, etc.).

Value

row_to_colnames() and colnames_to_row() both return a data frame.

Examples

# Convert a row to column names --------------------------------

test <- data.frame(
  a = c("iso", 2, 5),
  b = c("year", 3, 6),
  c = c("value", 5, 7)
)
test
row_to_colnames(test)

# Convert column names to row --------------------------------

test <- data.frame(
  ARG = c("BRA", "FRA"),
  `1960` = c(1960, 1960),
)
test
colnames_to_row(test)

skewness

Compute Skewness and (Excess) Kurtosis

Description

Compute Skewness and (Excess) Kurtosis

Usage

skewness(x, ...)

## S3 method for class 'numeric'

skewness(
  x,
  remove_na = TRUE,
  type = "2",
  iterations = NULL,
  verbose = TRUE,
  na.rm = TRUE,
  ...
)
kurtosis(x, ...)

## S3 method for class 'numeric'
kurtosis(
  x,
  remove_na = TRUE,
  type = "2",
  iterations = NULL,
  verbose = TRUE,
  na.rm = TRUE,
  ...
)

## S3 method for class 'parameters_kurtosis'
print(x, digits = 3, test = FALSE, ...)

## S3 method for class 'parameters_skewness'
print(x, digits = 3, test = FALSE, ...)

## S3 method for class 'parameters_skewness'
summary(object, test = FALSE, ...)

## S3 method for class 'parameters_kurtosis'
summary(object, test = FALSE, ...)

Arguments

x
A numeric vector or data.frame.

... Arguments passed to or from other methods.

remove_na Logical. Should NA values be removed before computing (TRUE) or not (FALSE, default)?

type Type of algorithm for computing skewness. May be one of 1 (or "1", "I" or "classic"), 2 (or "2", "II" or "SPSS" or "SAS") or 3 (or "3", "III" or "Minitab"). See 'Details'.

iterations The number of bootstrap replicates for computing standard errors. If NULL (default), parametric standard errors are computed.

verbose Toggle warnings and messages.

na.rm Deprecated. Please use remove_na instead.

digits Number of decimal places.

test Logical, if TRUE, tests if skewness or kurtosis is significantly different from zero.

object An object returned by skewness() or kurtosis().
Details

**Skewness:** Symmetric distributions have a skewness around zero, while a negative skewness values indicates a "left-skewed" distribution, and a positive skewness values indicates a "right-skewed" distribution. Examples for the relationship of skewness and distributions are:

- Normal distribution (and other symmetric distribution) has a skewness of 0
- Half-normal distribution has a skewness just below 1
- Exponential distribution has a skewness of 2
- Lognormal distribution can have a skewness of any positive value, depending on its parameters


**Types of Skewness:** skewness() supports three different methods for estimating skewness, as discussed in Joanes and Gill (1988):

- Type "1" is the "classical" method, which is \( g1 = \frac{\text{sum}((x - \text{mean}(x))^3) / n)}{\text{sum}((x - \text{mean}(x))^2) / n}^{1.5} \)
- Type "2" first calculates the type-1 skewness, then adjusts the result: \( G1 = g1 * \sqrt{n * (n - 1)} / (n - 2). \) This is what SAS and SPSS usually return.
- Type "3" first calculates the type-1 skewness, then adjusts the result: \( b1 = g1 * ((1 - 1 / n)^{1.5} \).

**Kurtosis:** The kurtosis is a measure of "tailedness" of a distribution. A distribution with a kurtosis values of about zero is called "mesokurtic". A kurtosis value larger than zero indicates a "leptokurtic" distribution with fatter tails. A kurtosis value below zero indicates a "platykurtic" distribution with thinner tails (https://en.wikipedia.org/wiki/Kurtosis).

**Types of Kurtosis:** kurtosis() supports three different methods for estimating kurtosis, as discussed in Joanes and Gill (1988):

- Type "1" is the "classical" method, which is \( g2 = n * \text{sum}((x - \text{mean}(x))^4) / (\text{sum}((x - \text{mean}(x))^2)^2 - 3. \)
- Type "2" first calculates the type-1 kurtosis, then adjusts the result: \( G2 = (((n + 1) * g2 + 6) \)
- Type "3" first calculates the type-1 kurtosis, then adjusts the result: \( b2 = (g2 + 3) * (1 - 1 / n)^{2} - 3. \) This is what Minitab usually returns.

**Standard Errors:** It is recommended to compute empirical (bootstrapped) standard errors (via the iterations argument) than relying on analytic standard errors (Wright & Herrington, 2011).

Value

Values of skewness or kurtosis.

References

Examples

```r
skewness(rnorm(1000))
kurtosis(rnorm(1000))
```

---

### slide

**Shift numeric value range**

**Description**

This function shifts the value range of a numeric variable, so that the new range starts at a given value.

**Usage**

```r
slide(x, ...)
```

```r
## S3 method for class 'numeric'
slide(x, lowest = 0, ...)
```

```r
## S3 method for class 'data.frame'
slide(
  x,
  select = NULL,
  exclude = NULL,
  lowest = 0,
  append = FALSE,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = TRUE,
  ...
)
```

**Arguments**

- `x` A data frame or numeric vector.
- `...` not used.
- `lowest` Numeric, indicating the lowest (minimum) value when converting factors or character vectors to numeric values.
- `select` Variables that will be included when performing the required tasks. Can be either
  - a variable specified as a literal variable name (e.g., column_name),
  - a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
  - a formula with variable names (e.g., ~column_1 + column_2).
• a vector of positive integers, giving the positions counting from the left (e.g. 1 or c(1, 3, 5)),
• a vector of negative integers, giving the positions counting from the right (e.g. -1 or -1:-3),
• one of the following select-helpers: starts_with(), ends_with(), contains(),
  a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal")
• or a function testing for logical conditions, e.g. is.numeric() (or is.numeric),
  or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
• ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a - e.g. -ends_with(""), -is.numeric or -(Sepal.Width:Petal.Length). **Note:** Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

**exclude**

See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

**append**

Logical or string. If TRUE, recoded or converted variables get new column names and are appended (column bind) to x, thus returning both the original and the recoded variables. The new columns get a suffix, based on the calling function: "+r" for recode functions, "+n" for to_numeric(), "+f" for to_factor(), or "+s" for slide(). If append=FALSE, original variables in x will be overwritten by their recoded versions. If a character value, recoded variables are appended with new column names (using the defined suffix) to the original data frame.

**ignore_case**

Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

**regex**

Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains("") or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see ’Details’), this argument may be used as workaround.

**verbose**

Toggle warnings.

**Value**

x, where the range of numeric variables starts at a new value.
Selection of variables - the select argument

For most functions that have a select argument (including this function), the complete input data frame is returned, even when select only selects a range of variables. That is, the function is only applied to those variables that have a match in select, while all other variables remain unchanged. In other words: for this function, select will not omit any non-included variables, so that the returned data frame will include all variables from the input data frame.

See Also

- Functions to rename stuff: `data_rename()`, `data_rename_rows()`, `data_addprefix()`, `data_addsuffix()`
- Functions to reorder or remove columns: `data_reorder()`, `data_relocate()`, `data_remove()`
- Functions to reshape, pivot or rotate data frames: `data_to_long()`, `data_to_wide()`, `data_rotate()`
- Functions to recode data: `rescale()`, `reverse()`, `categorize()`, `recode_values()`, `slide()`
- Functions to standardize, normalize, rank-transform: `center()`, `standardize()`, `normalize()`, `ranktransform()`, `winsorize()`
- Split and merge data frames: `data_partition()`, `data_merge()`
- Functions to find or select columns: `data_select()`, `data_find()`
- Functions to filter rows: `data_match()`, `data_filter()`

Examples

```r
# numeric
head(mtcars$gear)
head(slide(mtcars$gear))
head(slide(mtcars$gear, lowest = 10))

# data frame
sapply(slide(mtcars, lowest = 1), min)
sapply(mtcars, min)
```

smoothness

Quantify the smoothness of a vector

Description

Quantify the smoothness of a vector

Usage

```r
smoothness(x, method = "cor", lag = 1, iterations = NULL, ...)
```
**standardize**

### Arguments

- **x**: Numeric vector (similar to a time series).
- **method**: Can be "diff" (the standard deviation of the standardized differences) or "cor" (default, lag-one autocorrelation).
- **lag**: An integer indicating which lag to use. If less than 1, will be interpreted as expressed in percentage of the length of the vector.
- **iterations**: The number of bootstrap replicates for computing standard errors. If NULL (default), parametric standard errors are computed.
- **...**: Arguments passed to or from other methods.

### Value

Value of smoothness.

### References


### Examples

```r
x <- (-10:10)^3 + rnorm(21, 0, 100)
plot(x)
smoothness(x, method = "cor")
smoothness(x, method = "diff")
```

---

**standardize**  
*Standardization (Z-scoring)*

### Description

Performs a standardization of data (z-scoring), i.e., centering and scaling, so that the data is expressed in terms of standard deviation (i.e., mean = 0, SD = 1) or Median Absolute Deviance (median = 0, MAD = 1). When applied to a statistical model, this function extracts the dataset, standardizes it, and refits the model with this standardized version of the dataset. The `normalize()` function can also be used to scale all numeric variables within the 0 - 1 range.

For model standardization, see `standardize.default()`.

### Usage

```r
standardize(x, ...)
standardise(x, ...)
```

## S3 method for class 'numeric'
```r
standardize(
```
## S3 method for class 'factor'
standardize(
  x,
  robust = FALSE,
  two_sd = FALSE,
  weights = NULL,
  force = FALSE,
  verbose = TRUE,
  ...
)

## S3 method for class 'data.frame'
standardize(
  x,
  select = NULL,
  exclude = NULL,
  robust = FALSE,
  two_sd = FALSE,
  weights = NULL,
  reference = NULL,
  center = NULL,
  scale = NULL,
  remove_na = c("none", "selected", "all"),
  force = FALSE,
  append = FALSE,
  ignore_case = FALSE,
  regex = FALSE,
  verbose = TRUE,
  ...
)

unstandardize(x, ...)

unstandardise(x, ...)

## S3 method for class 'numeric'
unstandardize(}
Arguments

- **x**
  - A (grouped) data frame, a vector or a statistical model (for `unstandardize()` cannot be a model).
- **...**
  - Arguments passed to or from other methods.
- **robust**
  - Logical, if `TRUE`, centering is done by subtracting the median from the variables and dividing it by the median absolute deviation (MAD). If `FALSE`, variables are standardized by subtracting the mean and dividing it by the standard deviation (SD).
- **two_sd**
  - If `TRUE`, the variables are scaled by two times the deviation (SD or MAD depending on `robust`). This method can be useful to obtain model coefficients of continuous parameters comparable to coefficients related to binary predictors, when applied to the **predictors** (not the outcome) (Gelman, 2008).
- **weights**
  - Can be `NULL` (for no weighting), or:
    - For model: if `TRUE` (default), a weighted-standardization is carried out.
    - For data frames: a numeric vector of weights, or a character of the name of a column in the data frame that contains the weights.
    - For numeric vectors: a numeric vector of weights.
- **reference**
  - A data frame or variable from which the centrality and deviation will be computed instead of from the input variable. Useful for standardizing a subset or new data according to another data frame.
center, scale

- For `standardize()`:
  Numeric values, which can be used as alternative to reference to define a reference centrality and deviation. If scale and center are of length 1, they will be recycled to match the length of selected variables for standardization. Else, center and scale must be of same length as the number of selected variables. Values in center and scale will be matched to selected variables in the provided order, unless a named vector is given. In this case, names are matched against the names of the selected variables.

- For `unstandardize()`:
  center and scale correspond to the center (the mean / median) and the scale (SD / MAD) of the original non-standardized data (for data frames, should be named, or have column order correspond to the numeric column). However, one can also directly provide the original data through reference, from which the center and the scale will be computed (according to robust and two_sd). Alternatively, if the input contains the attributes center and scale (as does the output of `standardize()`), it will take it from there if the rest of the arguments are absent.

`verbose`  
Toggle warnings and messages on or off.

`force`  
Logical, if TRUE, forces recoding of factors and character vectors as well.

`select`  
Variables that will be included when performing the required tasks. Can be either

- a variable specified as a literal variable name (e.g., `column_name`),
- a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., `c("col1", "col2", "col3")`),
- a formula with variable names (e.g., `~column_1 + column_2`),
- a vector of positive integers, giving the positions counting from the left (e.g., `1` or `c(1, 3, 5)`),
- a vector of negative integers, giving the positions counting from the right (e.g., `-1` or `-1:-3`),
- one of the following select-helpers: `starts_with()`, `ends_with()`, `contains()`, `starts_with()`, `ends_with()`, and `contains()` accept several patterns, e.g `starts_with("Sep", "Petal")`.
- or a function testing for logical conditions, e.g. `is.numeric()` (or `is.numeric`), or any user-defined function that selects the variables for which the function returns TRUE (like: `foo <- function(x) mean(x) > 3`),
- ranges specified via literal variable names, select-helpers (except `regex()`) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. `~ends_with("")`, `~is.numeric` or `~(Sepal.Width:Petal.Length)`. **Note:** Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, `select=~ends_with("Length")` (with ~) is equivalent to `exclude=ends_with("Length")` (no ~). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. `find_columns(iris, select = c("Species", "Test"))` will just return "Species".
exclude See `select`, however, column names matched by the pattern from `exclude` will be excluded instead of selected. If NULL (the default), excludes no columns.

remove_na How should missing values (NA) be treated: if "none" (default): each column’s standardization is done separately, ignoring NAs. Else, rows with NA in the columns selected with `select / exclude` ("selected") or in all columns ("all") are dropped before standardization, and the resulting data frame does not include these cases.

append Logical or string. If TRUE, standardized variables get new column names (with the suffix "_z") and are appended (column bind) to `x`, thus returning both the original and the standardized variables. If FALSE, original variables in `x` will be overwritten by their standardized versions. If a character value, standardized variables are appended with new column names (using the defined suffix) to the original data frame.

ignore_case Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

regex Logical, if TRUE, the search pattern from `select` will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains("") or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.

Value

The standardized object (either a standardize data frame or a statistical model fitted on standardized data).

Selection of variables - the `select` argument

For most functions that have a select argument (including this function), the complete input data frame is returned, even when select only selects a range of variables. That is, the function is only applied to those variables that have a match in select, while all other variables remain unchanged. In other words: for this function, select will not omit any non-included variables, so that the returned data frame will include all variables from the input data frame.

Note

When `x` is a vector or a data frame with `remove_na = "none"`, missing values are preserved, so the return value has the same length / number of rows as the original input.

See Also

See `center()` for grand-mean centering of variables, and `makepredictcall.dw_transformer()` for use in model formulas.

Other transform utilities: `normalize()`, `ranktransform()`, `rescale()`, `reverse()`

Other standardize: `standardize.default()`
Examples

d <- iris[1:4, ]

# vectors
standardise(d$Petal.Length)

# Data frames
# overwrite
standardise(d, select = c("Sepal.Length", "Sepal.Width"))

# append
standardise(d, select = c("Sepal.Length", "Sepal.Width"), append = TRUE)

# append, suffix
standardise(d, select = c("Sepal.Length", "Sepal.Width"), append = "_std")

# standardizing with reference center and scale
d <- data.frame(
  a = c(-2, -1, 0, 1, 2),
  b = c(3, 4, 5, 6, 7)
)

# default standardization, based on mean and sd of each variable
standardize(d) # means are 0 and 5, sd ~ 1.581139

# standardization, based on mean and sd set to the same values
standardize(d, center = c(0, 5), scale = c(1.581, 1.581))

# standardization, mean and sd for each variable newly defined
standardize(d, center = c(3, 4), scale = c(2, 4))

# standardization, taking same mean and sd for each variable
standardize(d, center = 1, scale = 3)

---

standardize.default  Re-fit a model with standardized data

Description

Performs a standardization of data (z-scoring) using `standardize()` and then re-fits the model to the standardized data.

Standardization is done by completely refitting the model on the standardized data. Hence, this approach is equal to standardizing the variables before fitting the model and will return a new model object. This method is particularly recommended for complex models that include interactions or transformations (e.g., polynomial or spline terms). The robust (default to FALSE) argument enables a robust standardization of data, based on the median and the MAD instead of the mean and the SD.
Usage

```r
## Default S3 method:
standardize(
  x,
  robust = FALSE,
  two_sd = FALSE,
  weights = TRUE,
  verbose = TRUE,
  include_response = TRUE,
  ...
)
```

Arguments

- **x**: A statistical model.
- **robust**: Logical, if `TRUE`, centering is done by subtracting the median from the variables and dividing it by the median absolute deviation (MAD). If `FALSE`, variables are standardized by subtracting the mean and dividing it by the standard deviation (SD).
- **two_sd**: If `TRUE`, the variables are scaled by two times the deviation (SD or MAD depending on `robust`). This method can be useful to obtain model coefficients of continuous parameters comparable to coefficients related to binary predictors, when applied to the predictors (not the outcome) (Gelman, 2008).
- **weights**: If `TRUE` (default), a weighted-standardization is carried out.
- **verbose**: Toggle warnings and messages on or off.
- **include_response**: If `TRUE` (default), the response value will also be standardized. If `FALSE`, only the predictors will be standardized.
  - Note that for GLMs and models with non-linear link functions, the response value will not be standardized, to make re-fitting the model work.
  - If the model contains an `stats::offset()`, the offset variable(s) will be standardized only if the response is standardized. If `two_sd = TRUE`, offsets are standardized by one-sd (similar to the response).
  - (For mediate models, the `include_response` refers to the outcome in the `y` model; m model’s response will always be standardized when possible).

Value

A statistical model fitted on standardized data

Generalized Linear Models

Standardization for generalized linear models (GLM, GLMM, etc) is done only with respect to the predictors (while the outcome remains as-is, unstandardized) - maintaining the interpretability of the coefficients (e.g., in a binomial model: the exponent of the standardized parameter is the OR of a change of 1 SD in the predictor, etc.)
Dealing with Factors

`standardize(model)` or `standardize_parameters(model, method = "refit")` do not standardize categorical predictors (i.e. factors) / their dummy-variables, which may be a different behaviour compared to other R packages (such as `lm.beta`) or other software packages (like SPSS). To mimic such behaviours, either use `standardize_parameters(model, method = "basic")` to obtain post-hoc standardized parameters, or standardize the data with `standardize(data, force = TRUE) before fitting the model.

Transformed Variables

When the model’s formula contains transformations (e.g. y ~ exp(X)) the transformation effectively takes place after standardization (e.g., `exp(scale(X))`). Since some transformations are undefined for none positive values, such as `log()` and `sqrt()`, the relevel variables are shifted (post standardization) by `Z - min(Z) + 1` or `Z - min(Z)` (respectively).

See Also

Other `standardize`: `standardize()`

Examples

```r
model <- lm(Infant.Mortality ~ Education * Fertility, data = swiss)
coef(standardize(model))
```

---

**Description**

Convenience functions to manipulate and format text.

**Usage**

```r
text_format(
  text,
  sep = "", ",",
  last = " and ",
  width = NULL,
  enclose = NULL,
  ...
)
```

```r
format_text(
  text,
  sep = "", ",",
  last = " and ",
```

```r
```

**Examples**

```r
model <- lm(Infant.Mortality ~ Education * Fertility, data = swiss)
coef(standardize(model))
```
text_format

width = NULL,
enclose = NULL,
...
)
text_fullstop(text)
text_lastchar(text, n = 1)
text_concatenate(text, sep = "", last = " and ", enclose = NULL)
text_paste(text, text2 = NULL, sep = "", enclose = NULL, ...)
text_remove(text, pattern = "", ...)
text_wrap(text, width = NULL, ...)

Arguments

text, text2  A character string.
sep          Separator.
last         Last separator.
width        Positive integer giving the target column width for wrapping lines in the output. Can be "auto", in which case it will select 90\ default width.
enclose      Character that will be used to wrap elements of text, so these can be, e.g., enclosed with quotes or backticks. If NULL (default), text elements will not be enclosed.
...          Other arguments to be passed to or from other functions.
n            The number of characters to find.
pattern      Character vector. For data_rename(), indicates columns that should be selected for renaming. Can be NULL (in which case all columns are selected). For data_addprefix() or data_addsuffix(), a character string, which will be added as prefix or suffix to the column names.

Value

A character string.

Examples

# Add full stop if missing
text_fullstop(c("something", "something else."))

# Find last characters
text_lastchar(c("ABC", "DEF"), n = 2)

# Smart concatenation
text_concatenate(c("First", "Second", "Last"))
text_concatenate(c("First", "Second", "Last"), last = " or ", enclose = "")

# Remove parts of string
text_remove(c("one!", "two", "three!"), "!")

# Wrap text
long_text <- paste(rep("abc ", 100), collapse = "")
cat(text_wrap(long_text, width = 50))

# Paste with optional separator
text_paste(c("A", "", "B"), c("42", "42", "42"))

---

**to_factor**  
*Convert data to factors*

**Description**
Convert data to factors

**Usage**
to_factor(x, ...)

## S3 method for class 'numeric'
to_factor(x, labels_to_levels = TRUE, verbose = TRUE, ...)

## S3 method for class 'data.frame'
to_factor(
x,
  select = NULL,
  exclude = NULL,
  ignore_case = FALSE,
  append = FALSE,
  regex = FALSE,
  verbose = TRUE,
  ...
)

**Arguments**

- **x**  
  A data frame or vector.

- **...**  
  Arguments passed to or from other methods.

- **labels_to_levels**
  Logical, if TRUE, value labels are used as factor levels after x was converted to factor. Else, factor levels are based on the values of x (i.e. as if using as.factor()).

- **verbose**
  Toggle warnings.
Variables that will be included when performing the required tasks. Can be either

- a variable specified as a literal variable name (e.g., column_name),
- a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
- a formula with variable names (e.g., ~column_1 + column_2),
- a vector of positive integers, giving the positions counting from the left (e.g., 1 or c(1, 3, 5)),
- a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
- one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g. starts_with("Sep", "Petal").
- or a function testing for logical conditions, e.g. is.numeric() (or is.numeric), or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
- ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. -ends_with(""), -is.numeric or - (Sepal.Width:Petal.Length). **Note:** Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

Logical or string. If TRUE, recoded or converted variables get new column names and are appended (column bind) to x, thus returning both the original and the recoded variables. The new columns get a suffix, based on the calling function: "+_r" for recode functions, "+_n" for to_numeric(), "+_f" for to_factor(), or "+_s" for slide(). If append=FALSE, original variables in x will be overwritten by their recoded versions. If a character value, recoded variables are appended with new column names (using the defined suffix) to the original data frame.

Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains("") or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see 'Details'), this argument may be used as workaround.
Details

Convert variables or data into factors. If the data is labelled, value labels will be used as factor levels. The counterpart to convert variables into numeric is to_numeric().

Value

A factor, or a data frame of factors.

Selection of variables - the select argument

For most functions that have a select argument (including this function), the complete input data frame is returned, even when select only selects a range of variables. That is, the function is only applied to those variables that have a match in select, while all other variables remain unchanged. In other words: for this function, select will not omit any non-included variables, so that the returned data frame will include all variables from the input data frame.

Note

Factors are ignored and returned as is. If you want to use value labels as levels for factors, use labels_to_levels() instead.

Examples

str(to_factor(iris))

# use labels as levels
data(efc)
str(efc$c172code)
head(to_factor(efc$c172code))

---

to_numeric

Convert data to numeric

Description

Convert data to numeric by converting characters to factors and factors to either numeric levels or dummy variables. The "counterpart" to convert variables into factors is to_factor().

Usage

to_numeric(x, ...)

## S3 method for class 'data.frame'
to_numeric(
  x,
  select = NULL,
  exclude = NULL,
  dummy_factors = TRUE,
to_numeric

preserve_levels = FALSE,
lowest = NULL,
append = FALSE,
ignore_case = FALSE,
regex = FALSE,
verbose = TRUE,
...
}

Arguments

x A data frame, factor or vector.

... Arguments passed to or from other methods.

select Variables that will be included when performing the required tasks. Can be either

- a variable specified as a literal variable name (e.g., column_name),
- a string with the variable name (e.g., "column_name"), or a character vector of variable names (e.g., c("col1", "col2", "col3")),
- a formula with variable names (e.g., ~column_1 + column_2),
- a vector of positive integers, giving the positions counting from the left (e.g. 1 or c(1, 3, 5)),
- a vector of negative integers, giving the positions counting from the right (e.g., -1 or -1:-3),
- one of the following select-helpers: starts_with(), ends_with(), contains(), a range using : or regex(""), starts_with(), ends_with(), and contains() accept several patterns, e.g starts_with("Sep", "Petal").
- or a function testing for logical conditions, e.g. is.numeric() (or is.numeric), or any user-defined function that selects the variables for which the function returns TRUE (like: foo <- function(x) mean(x) > 3),
- ranges specified via literal variable names, select-helpers (except regex()) and (user-defined) functions can be negated, i.e. return non-matching elements, when prefixed with a -, e.g. -ends_with(""), -is.numeric or -(Sepal.Width:Petal.Length). **Note:** Negation means that matches are excluded, and thus, the exclude argument can be used alternatively. For instance, select=-ends_with("Length") (with -) is equivalent to exclude=ends_with("Length") (no -). In case negation should not work as expected, use the exclude argument instead.

If NULL, selects all columns. Patterns that found no matches are silently ignored, e.g. find_columns(iris, select = c("Species", "Test")) will just return "Species".

exclude See select, however, column names matched by the pattern from exclude will be excluded instead of selected. If NULL (the default), excludes no columns.

dummy_factors Transform factors to dummy factors (all factor levels as different columns filled with a binary 0-1 value).
preserve_levels Logical, only applies if x is a factor. If TRUE, and x has numeric factor levels, these will be converted into the related numeric values. If this is not possible, the converted numeric values will start from 1 to number of levels.

lowest Numeric, indicating the lowest (minimum) value when converting factors or character vectors to numeric values.

append Logical or string. If TRUE, recoded or converted variables get new column names and are appended (column bind) to x, thus returning both the original and the recoded variables. The new columns get a suffix, based on the calling function: 
"_r" for recode functions, "_n" for to_numeric(), "_f" for to_factor(), or 
"_s" for slide(). If append=FALSE, original variables in x will be overwritten by their recoded versions. If a character value, recoded variables are appended with new column names (using the defined suffix) to the original data frame.

ignore_case Logical, if TRUE and when one of the select-helpers or a regular expression is used in select, ignores lower/upper case in the search pattern when matching against variable names.

regex Logical, if TRUE, the search pattern from select will be treated as regular expression. When regex = TRUE, select must be a character string (or a variable containing a character string) and is not allowed to be one of the supported select-helpers or a character vector of length > 1. regex = TRUE is comparable to using one of the two select-helpers, select = contains("") or select = regex(""), however, since the select-helpers may not work when called from inside other functions (see ‘Details’), this argument may be used as workaround.

verbose Toggle warnings.

Value
A data frame of numeric variables.

Selection of variables - select argument
For most functions that have a select argument the complete input data frame is returned, even when select only selects a range of variables. However, for to_numeric(), factors might be converted into dummies, thus, the number of variables of the returned data frame no longer match the input data frame. Hence, when select is used, only those variables (or their dummies) specified in select will be returned. Use append=TRUE to also include the original variables in the returned data frame.

Examples
```r
to_numeric(head(ToothGrowth))
to_numeric(head(ToothGrowth), dummy_factors = FALSE)

# factors
x <- as.factor(mtcars$gear)
to_numeric(x, dummy_factors = FALSE)
to_numeric(x, dummy_factors = FALSE, preserve_levels = TRUE)
```
visualisation_recipe  Prepare objects for visualisation

Description

This function prepares objects for visualisation by returning a list of layers with data and geoms that can be easily plotted using for instance ggplot2. If the see package is installed, the call to visualization_recipe() can be replaced by plot(), which will internally call the former and then plot it using ggplot. The resulting plot can be customized ad-hoc (by adding ggplot's geoms, theme or specifications), or via some of the arguments of visualization_recipe() that control the aesthetic parameters.

See the specific documentation page for your object's class:


Usage

```r
visualisation_recipe(x, ...)
```

Arguments

- `x` An easystats object.
- `...` Other arguments passed to other functions.

weighted_mean  Weighted Mean, Median, SD, and MAD

Description

Weighted Mean, Median, SD, and MAD

Usage

```r
weighted_mean(x, weights = NULL, remove_na = TRUE, verbose = TRUE, ...)
weighted_median(x, weights = NULL, remove_na = TRUE, verbose = TRUE, ...)
weighted_sd(x, weights = NULL, remove_na = TRUE, verbose = TRUE, ...)
weighted_mad(
  x,
...)```
weights = NULL,
constant = 1.4826,
remove_na = TRUE,
verbose = TRUE,
...
)

Arguments

x an object containing the values whose weighted mean is to be computed.
weights A numerical vector of weights the same length as x giving the weights to use for elements of x. If weights = NULL, x is passed to the non-weighted function.
remove_na Logical, if TRUE (default), removes missing (NA) and infinite values from x and weights.
verbose Show warning when weights are negative?
... arguments to be passed to or from methods.
constant scale factor.

Examples

## GPA from Siegel 1994
x <- c(3.7, 3.3, 3.5, 2.8)
wt <- c(5, 5, 4, 1) / 15

weighted_mean(x, wt)
weighted_median(x, wt)

weighted_sd(x, wt)
weighted_mad(x, wt)

winsorize

Winsorize data

Description

Winsorize data

Usage

winsorize(data, ...)

## S3 method for class 'numeric'
winsorize(
data,
threshold = 0.2,
method = "percentile",
...
winsorize

```r
robust = FALSE,
verbose = TRUE,
...
)
```

Arguments

- **data**: data frame or vector.
- **...**: Currently not used.
- **threshold**: The amount of winsorization, depends on the value of method:
  - For method = "percentile": the amount to winsorize from each tail. The value of threshold must be between 0 and 0.5 and of length 1.
  - For method = "zscore": the number of \( SD/MAD \)-deviations from the mean/median (see robust). The value of threshold must be greater than 0 and of length 1.
  - For method = "raw": a vector of length 2 with the lower and upper bound for winsorization.
- **method**: One of "percentile" (default), "zscore", or "raw".
- **robust**: Logical, if TRUE, winsorizing through the "zscore" method is done via the median and the median absolute deviation (MAD); if FALSE, via the mean and the standard deviation.
- **verbose**: Not used anymore since datawizard 0.6.6.

Details

Winsorizing or winsorization is the transformation of statistics by limiting extreme values in the statistical data to reduce the effect of possibly spurious outliers. The distribution of many statistics can be heavily influenced by outliers. A typical strategy is to set all outliers (values beyond a certain threshold) to a specified percentile of the data; for example, a 90% winsorization would see all data below the 5th percentile set to the 5th percentile, and data above the 95th percentile set to the 95th percentile. Winsorized estimators are usually more robust to outliers than their more standard forms.

Value

A data frame with winsorized columns or a winsorized vector.

See Also

- Functions to rename stuff: `data_rename()`, `data_rename_rows()`, `data_addprefix()`, `data_addsuffix()`
- Functions to reorder or remove columns: `data_reorder()`, `data_relocate()`, `data_remove()`
- Functions to reshape, pivot or rotate data frames: `data_to_long()`, `data_to_wide()`, `data_rotate()`
- Functions to recode data: `rescale()`, `reverse()`, `categorize()`, `recode_values()`, `slide()`
- Functions to standardize, normalize, rank-transform: `center()`, `standardize()`, `normalize()`, `ranktransform()`, `winsorize()`
- Split and merge data frames: `data_partition()`, `data_merge()`
- Functions to find or select columns: `data_select()`, `data_find()`
- Functions to filter rows: `data_match()`, `data_filter()`

**Examples**

```r
hist(iris$Sepal.Length, main = "Original data")

hist(winsorize(iris$Sepal.Length, threshold = 0.2),
     xlim = c(4, 8), main = "Percentile Winsorization")

hist(winsorize(iris$Sepal.Length, threshold = 1.5, method = "zscore"),
     xlim = c(4, 8), main = "Mean (+/- SD) Winsorization")

hist(winsorize(iris$Sepal.Length, threshold = 1.5, method = "zscore", robust = TRUE),
     xlim = c(4, 8), main = "Median (+/- MAD) Winsorization")

hist(winsorize(iris$Sepal.Length, threshold = c(5, 7.5), method = "raw"),
     xlim = c(4, 8), main = "Raw Thresholds")

# Also works on a data frame:
winsorize(iris, threshold = 0.2)
```
Index

* **datawizard-transformers**
  - `makepredictcall.dw_transformer`, 87
* **data**
  - `efc`, 81
  - `nhanes_sample`, 89
* **duplicates**
  - `data_duplicated`, 31
* **standardize**
  - `standardize`, 119
  - `standardize.default`, 124
* **transform utilities**
  - `normalize`, 89
  - `ranktransform`, 92
  - `rescale`, 103
  - `reverse`, 109
  - `standardize`, 119

adjust, 3
assign_labels, 6

`bayestestR::map_estimate()`, 80

categorize, 9
categorize(), 12, 27, 39, 41, 46, 53, 55, 65, 68, 84, 99, 118, 135
center, 13
center(), 12, 27, 39, 41, 46, 53, 55, 65, 68, 76, 84, 87, 99, 118, 123, 135
centre (center), 13
change_code (recode_values), 96
change_scale (rescale), 103
coeff_var, 17
coerce_to_numeric, 18
colnames_to_row (row_to_colnames), 112
column_as_rownames
  - `rownames_as_column`, 111
convert_na_to, 19
convert_to_na, 22
data_addprefix, 24
data_addprefix(), 12, 26, 39, 41, 46, 53, 55, 65, 67, 84, 99, 118, 135
data_addsuffix (data_addprefix), 24
data_addsuffix(), 12, 26, 39, 41, 46, 53, 55, 65, 67, 84, 99, 118, 135
data_adjust (adjust), 3
data_arrange, 27
data_codebook, 28
data_duplicated, 31
data_duplicated(), 70
data_extract, 33
data_filter (data_match), 37
data_filter(), 12, 27, 39, 41, 46, 53, 55, 65, 68, 84, 99, 118, 136
data_find (find_columns), 81
data_find(), 12, 27, 39, 41, 46, 53, 55, 65, 68, 84, 99, 118, 136
data_group, 35
data_join (data_merge), 40
data_match, 37
data_match(), 12, 27, 39, 41, 46, 53, 55, 65, 68, 84, 99, 118, 136
data_merge, 40
data_merge(), 12, 27, 39, 41, 46, 53, 55, 65, 68, 84, 99, 118, 135
data_modify, 43
data_partition, 45
data_partition(), 12, 27, 39, 41, 46, 53, 55, 65, 68, 84, 99, 118, 135
data.peek, 47
data_read, 49
data_relocate, 51
data_relocate(), 12, 26, 39, 41, 46, 53, 55, 65, 67, 84, 99, 118, 135
data_remove (data_relocate), 51
data_remove(), 12, 26, 39, 41, 46, 53, 55, 65, 67, 84, 99, 118, 135
data_rename (data_addprefix), 24
data_rename(), 12, 26, 39, 41, 46, 53, 55, 65,
ifelse(), 94

data_rename_rows (data_addprefix), 24
data_rename_rows(), 12, 26, 39, 41, 46, 53, 55, 63, 67, 84, 99, 118, 135
data_reorder (data_relocate), 51
data_reorder(), 12, 26, 39, 41, 46, 53, 55, 65, 67, 84, 99, 118, 135
data_restore_type, 54
data_rotate, 55
data_rotate(), 12, 27, 39, 41, 46, 53, 55, 65, 67, 84, 99, 118, 135
data_select (find_columns), 81
data_select(), 12, 27, 39, 41, 46, 53, 55, 65, 67, 84, 99, 118, 136
data_separate, 56
data_separate(), 72
data_tabulate, 60
data_to_long, 63
data_to_long(), 12, 27, 39, 41, 46, 53, 55, 65, 67, 84, 99, 118, 135
data_to_wide, 66
data_to_wide(), 12, 27, 39, 41, 46, 53, 55, 65, 67, 84, 99, 118, 135
data_transpose (data_rotate), 55
data_ungroup (data_group), 35
data_unique, 69
data_unique(), 32
data_unite, 71
data_unite(), 58
data_write (data_read), 49
degroup (demean), 73
demean, 73
demean(), 16
describe_distribution, 77
detrend (demean), 73
distribution_coef_var (coef_var), 17
distribution_cv (coef_var), 17
distribution_mode, 80
dplyr::distinct(), 31
duplicated(), 31
efc, 81
empty_columns (remove_empty), 101
empty_rows (remove_empty), 101
find_columns, 81
format_text (text_format), 126
gt_columns (find_columns), 81

kurtosis (skewness), 113
labels_to_levels, 85
labels_to_levels(), 130
makepredictcall. dw_transformer, 87
makepredictcall. dw_transformer(), 16, 92, 105, 123
mean_sd, 88
median_mad (mean_sd), 88

nhanes_sample, 89
normalize, 89, 94, 105, 111, 123
normalize(), 12, 27, 39, 41, 46, 53, 55, 65, 68, 84, 87, 99, 118, 119, 135

print_parameters_kurtosis (skewness), 113
print_parameters_skewness (skewness), 113
print_html.data_codebook
(data_codebook), 28
rank(), 93
ranktransform(), 92, 92, 105, 111, 123
ranktransform(), 12, 27, 39, 41, 46, 53, 55, 65, 68, 84, 87, 99, 118, 119, 135
recode_into, 94
recode_values, 96
recode_values(), 12, 27, 39, 41, 46, 53, 55, 65, 68, 84, 87, 99, 118, 119, 135
remove_empty, 101
remove_empty_columns (remove_empty), 101
remove_empty_rows (remove_empty), 101
replace_nan_inf, 102
rescale, 92, 94, 103, 111, 123
rescale(), 12, 27, 39, 41, 46, 53, 55, 65, 68, 84, 87, 89, 98, 99, 118, 135
rescale_weights, 106
reshape_ci, 108
reshape_longer (data_to_long), 63
reshape_wider (data_to_wide), 66
reverse, 92, 94, 105, 109, 123
reverse(), 12, 27, 39, 41, 46, 53, 55, 65, 68, 84, 98, 99, 118, 135
reverse_scale (reverse), 109
row_to_colnames, 112
INDEX

rowid_as_column (rownames_as_column), 111
rownames_as_column, 111

skewness, 113
slide, 116
slide(), 12, 27, 39, 41, 46, 53, 55, 65, 68, 84, 99, 118, 135

smoothness, 118
standardise (standardize), 119
standardize, 92, 94, 105, 111, 119, 126
standardize(), 12, 16, 27, 39, 41, 46, 53, 55, 65, 68, 84, 87, 99, 118, 124, 135
standardize.default, 123, 124
standardize.default(), 119

standardize_models (standardize.default), 124
stats::IQR(), 78
stats::mad(), 17
stats::makepredictcall(), 87
stats::offset(), 125

summary.parameters_kurtosis (skewness), 113
summary.parameters_skewness (skewness), 113

text_concatenate (text_format), 126
text_format, 126
text_fullstop (text_format), 126
text_lastchar (text_format), 126
text_paste (text_format), 126
text_remove (text_format), 126
text_wrap (text_format), 126
to_factor, 128
to_numeric, 130

unnormalize (normalize), 89
unstandardise (standardize), 119
unstandardize (standardize), 119

visualisation_recipe, 133

weighted_mad (weighted_mean), 133
weighted_mean, 133
weighted_median (weighted_mean), 133
weighted_sd (weighted_mean), 133
winsorize, 134
winsorize(), 12, 27, 39, 41, 46, 53, 55, 65, 68, 84, 99, 118, 135