Package: quest (via r-universe)

October 31, 2024

Type Package

Title Prepare Questionnaire Data for Analysis

Version 0.2.0

Description Offers a suite of functions to prepare questionnaire data for analysis (perhaps other types of data as well). By data preparation, I mean data analytic tasks to get your raw data ready for statistical modeling (e.g., regression). There are functions to investigate missing data, reshape data, validate responses, recode variables, score questionnaires, center variables, aggregate by groups, shift scores (i.e., leads or lags), etc. It provides functions for both single level and multilevel (i.e., grouped) data. With a few exceptions (e.g., ncases()), functions without an ``s" at the end of their primary word (e.g., center_by()) act on atomic vectors, while functions with an ``s" at the end of their primary word (e.g., centers_by()) act on multiple columns of a data.frame.

Depends R (>= 4.0.0), datasets, stats, utils, methods

Imports str2str, abind, checkmate, plyr, car, psych, boot, MBESS, nlme, lme4, multilevel, lavaan

Suggests reshape, psychTools, lmeInfo, semTools

License GPL (>= 2)

Encoding UTF-8

RoxygenNote 7.2.3

Collate 'quest_functions.R' 'psymet_functions.R' 'describes_functions.R' 'diary_functions.R' 'mia_functions.R'

NeedsCompilation no

Author David Disabato [aut, cre] (<https://orcid.org/0000-0001-7094-4996>)

Maintainer David Disabato <ddisab01@gmail.com>

Date/Publication 2023-12-05 00:10:02 UTC

Repository https://ddisab01.r-universe.dev

RemoteUrl https://github.com/cran/quest

RemoteRef HEAD

RemoteSha 1568a5217fdc553ede6b7c5b4a2059939626d71e

Contents

quest-package . . . 4
.cronbach . . . 6
.cronbachs . . . 7
.gtheory . . . 8
.gtheorys . . . 9
add_sig . . . 10
add_sig_cor . . . 11
agg . . . 13
aggs . . . 15
agg_dfm . . . 17
amd_bi . . . 20
amd_multi . . . 21
amd_uni . . . 22
auto_by . . . 23
ave_dfm . . . 25
boot_ci . . . 26
by2 . . . 28
center . . . 29
centers . . . 30
centers_by . . . 31
center_by . . . 32
change . . . 33
changes . . . 34
changes_by . . . 35
change_by . . . 37
colMeans_if . . . 38
colNA . . . 39
colSums_if . . . 40
composite . . . 41
composites . . . 44
confint2 . . . 46
confint2.boot . . . 47
confint2.default . . . 49
corp . . . 50
corp_by . . . 52
corp_miss . . . 54
corp_ml . . . 56
cor_by . . . 58
cor_miss . . . 59
cor_ml . . . 61
covs_test . . . 62

cronbach 64
cronbachs 66
decompose 68
decomposes 69
deff 71
deffs 72
describe_ml 73
dum2nom 75
freq 76
freqs 78
freqs_by 79
freq_by 81
gtheory 83
gtheorys 85
gtheorys_ml 87
gtheory_ml 89
iccs_11 91
icc_11 92
icc_all_by 94
lengths_by 96
length_by 97
long2wide 98
make.dummy 100
make.dumNA 101
make.fun_if 102
make.latent 103
make.product 104
means_change 106
means_compare 109
means_diff 112
means_test 115
mean_change 118
mean_compare 120
mean_diff 123
mean_if 126
mean_test 128
mode2 130
ncases 131
ncases_by 132
ncases_desc 133
ncases_ml 135
ngrp 137
nhst 138
nom2dum 139
nrow_by 140
nrow_ml 141
n_compare 142
partial.cases 143

pomp . . . 144
pomps . . . 145
props_compare . . . 147
props_diff . . . 150
props_test . . . 153
prop_compare . . . 156
prop_diff . . . 159
prop_test . . . 162
recode2other . . . 164
recodes . . . 166
renames . . . 167
reorders . . . 169
revalid . . . 170
revalids . . . 171
reverse . . . 172
reverses . . . 173
rowMeans_if . . . 174
rowNA . . . 175
rowsNA . . . 176
rowSums_if . . . 177
score . . . 178
scores . . . 180
shift . . . 181
shifts . . . 183
shifts_by . . . 184
shift_by . . . 185
summary_ucfa . . . 186
sum_if . . . 189
tapply2 . . . 190
ucfa . . . 192
valids_test . . . 194
valid_test . . . 195
vecNA . . . 196
wide2long . . . 197
winsor . . . 200
winsors . . . 201

Index **203**

`quest-package` *Pre-processing Questionnaire Data*

Description

quest is a package for pre-processing questionnaire data to get it ready for statistical modeling. It contains functions for investigating missing data (e.g., rowNA), reshaping data (e.g., wide2long), validating responses (e.g., revalids), recoding variables (e.g., recodes), scoring (e.g., scores), centering (e.g., centers), aggregating (e.g., aggs), shifting (e.g., shifts), etc. Functions whose first phrases end with an s are vectorized versions of their functions without an s at the end of the first phrase. For example, center inputs a (atomic) vector and outputs a atomic vector to center and/or scale a single variable; centers inputs a data.frame and outputs a data.frame to center and/or scale multiple variables. Functions that end in _by are calculated by group. For example, center does grand-mean centering while center_by does group-mean centering. Putting the two together, centers_by inputs a data.frame and outputs a data.frame to center and/or scale multiple variables by group. Functions that end in _ml calculate a "multilevel" result with a within-group result and between-group result. Functions that end in _if are calculated dependent on the frequency of observed values (aka amount of missing data). The quest package uses the str2str package internally to convert R objects from one structure to another. See str2str for details.

Types of functions

There are three main types of functions. 1) Helper functions that primarily exist to save a few lines of code and are primarily for convenience (e.g., vecNA). 2) Functions for wrangling questionnaire data (e.g., nom2dum, reverses). 3) Functions for preliminary statistical calculation (e.g., means_diff, corp_by).

Abbreviations

See the table below

vrb variable

grp group

nm names

NA missing values

ov observed values

prop proportion

sep separator

cor correlations

id identifier

rtn return

fun function

dfm data.frame

fct factor

nom nominal variable

bin binary variable

dum dummy variable

pomp percentage of maximum possible

std standardize

wth within-groups

btw between-groups

Author(s)

Maintainer: David Disabato <ddisab01@gmail.com> (ORCID)

`.cronbach` *Bootstrap Function for* `cronbach()` *Function*

Description

`.cronbach` is the function used by the `boot` function within the `cronbach` function. It is primarily created to increase the computational efficiency of bootstrap confidence intervals within the `cronbach` function by doing only the minimal computations needed to compute cronbach's alpha.

Usage

```
.cronbach(dat, i, use)
```

Arguments

`dat`	data.frame with only the items you wish to include in the cronbach's alpha computation and no other variables.
`i`	integer vector of length = `nrow(dat)` specifying which rows should be included in the computation. When used by the `boot::boot` function this argument will change with every new bootstrapped resample.
`use`	character vector of length 1 specifying how missing data should be handled when computing covariances. See `cov` for details.

Value

double vector of length 1 providing cronbach's alpha

Examples

```
.cronbach(dat = attitude,
  i = sample(x = 1:nrow(attitude), size = nrow(attitude), replace = TRUE), use = "pairwise")
```

`.cronbachs` *Bootstrap Function for* `cronbachs()` *Function*

Description

`.cronbachs` is the function used by the `boot` function within the `cronbachs` function. It is primarily created to increase the computational efficiency of bootstrap confidence intervals within the `cronbachs` function by doing only the minimal computations needed to compute cronbach's alpha for each set of variables/items.

Usage

```
.cronbachs(dat, i, nm.list, use)
```

Arguments

`dat`	data.frame of data. It can contain variables other than those used for cronbach's alpha calculation.
`i`	integer vector of length = `nrow(dat)` specifying which rows should be included in the computation. When used by the `boot::boot` function this argument will change with every new bootstrapped resample.
`nm.list`	list of character vectors specifying the sets of variables/items associated with each of the cronbach's alpha calculations.
`use`	character vector of length 1 specifying how missing data should be handled when computing covariances. See `cov` for details.

Value

double vector of length = `length(nm.list)` providing cronbach's alpha for each set of variables/items.

Examples

```
dat0 <- psych::bfi[1:250, ]
dat1 <- str2str::pick(x = dat0, val = c("A1","C4","C5","E1","E2","O2","O5",
   "gender","education","age"), not = TRUE, nm = TRUE)
vrb_nm_list <- lapply(X = str2str::sn(c("E","N","C","A","O")), FUN = function(nm) {
   str2str::pick(x = names(dat1), val = nm, pat = TRUE)})
.cronbachs(dat = dat1,
   i = sample(x = 1:nrow(dat1), size = nrow(dat1), replace = TRUE),
   nm.list = vrb_nm_list, use = "pairwise")
```

.gtheory *Bootstrap Function for* gtheory() *Function*

Description

.gtheory is the function used by the boot function within the gtheory function. It is primarily created to increase the computational efficiency of bootstrap confidence intervals within the gtheory function by doing only the minimal computations needed to compute the generalizability theory coefficient.

Usage

```
.gtheory(dat, i, cross.vrb)
```

Arguments

dat	data.frame with only the variables/items you wish to include in the generalizability theory coefficient and no other variables/items.
i	integer vector of length = nrow(dat) specifying which rows should be included in the computation. When used by the boot::boot function this argument will change with every new bootstrapped resample.
cross.vrb	logical vector of length 1 specifying whether the variables/items should be crossed when computing the generalizability theory coefficient. If TRUE, then only the covariance structure of the variables/items will be incorperated into the estimate of reliability. If FALSE, then the mean structure of the variables/items will be incorperated.

Value

double vector of length 1 providing the generalizability theory coefficient.

See Also

.gtheorys gtheory

Examples

```
.gtheory(dat = attitude,
   i = sample(x = 1:nrow(attitude), size = nrow(attitude), replace = TRUE),
   cross.vrb = TRUE)
.gtheory(dat = attitude,
   i = sample(x = 1:nrow(attitude), size = nrow(attitude), replace = TRUE),
   cross.vrb = FALSE)
```

`.gtheorys` *Bootstrap Function for* `gtheorys()` *Function*

Description

`.gtheorys` is the function used by the `boot` function within the `gtheorys` function. It is primarily created to increase the computational efficiency of bootstrap confidence intervals within the `gtheorys` function by doing only the minimal computations needed to compute the generalizability theory coefficient.

Usage

```
.gtheorys(dat, i, nm.list, cross.vrb)
```

Arguments

`dat`	data.frame of data. It can contain variables other than those used for generalizability theory coefficient calculation.
`i`	integer vector of length = `nrow(dat)` specifying which rows should be included in the computation. When used by the `boot::boot` function this argument will change with every new bootstrapped resample.
`nm.list`	list of character vectors specifying the sets of variables/items associated with each of the generalizability theory coefficient calculations.
`cross.vrb`	logical vector of length 1 specifying whether the variables/items should be crossed when computing the generalizability theory coefficient. If TRUE, then only the covariance structure of the variables/items will be incorperated into the estimate of reliability. If FALSE, then the mean structure of the variables/items will be incorperated.

Value

double vector of length = `length(nm.list)` providing the generalizability theory coefficients.

See Also

`.gtheory` `gtheorys`

Examples

```
dat0 <- psych::bfi[1:250, ]
dat1 <- str2str::pick(x = dat0, val = c("A1","C4","C5","E1","E2","O2","O5",
   "gender","education","age"), not = TRUE, nm = TRUE)
vrb_nm_list <- lapply(X = str2str::sn(c("E","N","C","A","O")), FUN = function(nm) {
   str2str::pick(x = names(dat1), val = nm, pat = TRUE)})
.gtheorys(dat = dat1,
   i = sample(x = 1:nrow(dat1), size = nrow(dat1), replace = TRUE),
   nm.list = vrb_nm_list, cross.vrb = TRUE)
```

```
.gtheorys(dat = dat1,
   i = sample(x = 1:nrow(dat1), size = nrow(dat1), replace = TRUE),
   nm.list = vrb_nm_list, cross.vrb = FALSE)
```

---

add_sig *Add Significance Symbols to a (Atomic) Vector, Matrix, or Array*

Description

add_sig adds symbols for various p-values cutoffs of statistical significance. The function inputs a numeric vector, matrix, or array of effect sizes (e.g., correlation matrix) and a numeric vector, matrix, or array of p-values that correspond to the effect size (i.e., each row and column match) and then returns a character vector, matrix, or array of effect sizes with appended significance symbols. One of the primary applications of this function is use within corp corp_by, and corp_ml for correlation matrices.

Usage

```
add_sig(
  x,
  p,
  digits = 3,
  p.10 = "",
  p.05 = "*",
  p.01 = "**",
  p.001 = "***",
  lead.zero = FALSE,
  trail.zero = TRUE,
  plus = FALSE
)
```

Arguments

x	double numeric vector of effect sizes for which statistical significance is available.
p	double matrix of p-values for the effect sizes in x that are matched by element index for vectors, by row and column index with matrices, by row, column, and layer index for 3D arrays, etc. For example, the p-value in the first row and second column of p is associated with the effect size in the first row and second column of x. If x and p do not have the same dimensions, an error is returned.
digits	integer vector of length 1 specifying the number of decimals to round to.
p.10	character vector of length 1 specifying which symbol to append to the end of any effect size significant at the p < .10 level.
p.05	character vector of length 1 specifying which symbol to append to the end of any effect size significant at the p < .05 level.

p.01	character vector of length 1 specifying which symbol to append to the end of any effect size significant at the p < .01 level.
p.001	character vector of length 1 specifying which symbol to append to the end of any effect size significant at the p < .001 level.
lead.zero	logical vector of length 1 specifying whether to retain a zero in front of the decimal place if the effect size is within 1 or -1.
trail.zero	logical vector of length 1 specifying whether to retain zeros after the decimal place (due to rounding).
plus	logical vector of length 1 specifying whether to include a plus sign in front of positive effect sizes (minus signs are always in front of negative effect sizes).

Details

There are several functions out there that do similar things. Here is one posted to R-bloggers that does it for correlation matrices using the `corr` function from the `Hmisc` package: https://www.r-bloggers.com/2020/07/create-a-publication-ready-correlation-matrix-with-significance-levels-in-r

Value

character vector, matrix, or array with the same dimensions as `x` and `p` containing the effect sizes with their significance symbols appended to the end of each value.

Examples

```
corr_test <- psych::corr.test(mtcars[1:5])
r <- corr_test[["r"]]
p <- corr_test[["p"]]
add_sig(x = r, p = p)
add_sig(x = r, p = p, digits = 2)
add_sig(x = r, p = p, lead.zero = TRUE, trail.zero = FALSE)
add_sig(x = r, p = p, plus = TRUE)
noquote(add_sig(x = r, p = p)) # no quotes for character elements
```

`add_sig_cor` *Add Significance Symbols to a Correlation Matrix*

Description

`add_sig_cor` adds symbols for various p-values cutoffs of statistical significance. The function inputs a correlation matrix and a numeric matrix of p-values that correspond to the correlations (i.e., each row and column match) and then returns a data.frame of correlations with appended significance symbols. One of the primary applications of this function is use within `corp` `corp_by`, and `corp_ml` for correlation matrices.

Usage

```
add_sig_cor(
  r,
  p,
  digits = 3,
  p.10 = "",
  p.05 = "*",
  p.01 = "**",
  p.001 = "***",
  lead.zero = FALSE,
  trail.zero = TRUE,
  plus = FALSE,
  diags = FALSE,
  lower = TRUE,
  upper = FALSE
)
```

Arguments

r	double numeric matrix of correlation coefficients for which statistical significance is available. Since its a correlation matrix, it must be symmetrical and is expected to be a full matrix with all elements included (not just lower or upper diagonals values included).
p	double matrix of p-values for the correlations in r that are matched by row and column index. For example, the p-value in the first row and second column of p is associated with the correlation in the first row and second column of r. If r and p do not have the same dimensions, an error is returned.
digits	integer vector of length 1 specifying the number of decimals to round to.
p.10	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .10 level.
p.05	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .05 level.
p.01	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .01 level.
p.001	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .001 level.
lead.zero	logical vector of length 1 specifying whether to retain a zero in front of the decimal place.
trail.zero	logical vector of length 1 specifying whether to retain zeros after the decimal place (due to rounding).
plus	logical vector of length 1 specifying whether to include a plus sign in front of positive correlations (minus signs are always in front of negative correlations).
diags	logical vector of length 1 specifying whether to retain the values in the diagonal of the correlation matrix. If TRUE, then the diagonal will be 1s with digits number of zeros after the decimal place (and no significant symbols). If FALSE, then the diagonal will be NA.

`lower` logical vector of length 1 specifying whether to retain the lower triangle of the correlation matrix. If TRUE, then the lower triangle correlations and their significance symbols are retained. If FAlSE, then the lower triangle will all be NA.

`upper` logical vector of length 1 specifying whether to retain the upper triangle of the correlation matrix. If TRUE, then the upper triangle correlations and their significance symbols are retained. If FAlSE, then the upper triangle will all be NA.

Details

There are several functions out there that do similar things. Here is one posted to R-bloggers that uses the `corr` function from the `Hmisc` package: https://www.r-bloggers.com/2020/07/create-a-publication-ready-correlation-matrix-with-significance-levels-in-r/.

Value

data.frame with the same dimensions as `r` containing the correlations and their significance symbols. Elements may or may not contain NA values depending on the arguments `diags`, `lower`, and `upper`.

Examples

```
corr_test <- psych::corr.test(mtcars[1:5])
r <- corr_test[["r"]]
p <- corr_test[["p"]]
add_sig_cor(r = r, p = p)
add_sig_cor(r = r, p = p, digits = 2)
add_sig_cor(r = r, p = p, diags = TRUE)
add_sig_cor(r = r, p = p, lower = FALSE, upper = TRUE)
add_sig_cor(r = r, p = p, lead.zero = TRUE, trail.zero = FALSE)
add_sig_cor(r = r, p = p, plus = TRUE)
```

---

agg *Aggregate an Atomic Vector by Group*

Description

`agg` evaluates a function separately for each group and combines the results back together into an atomic vector of data.frame that is returned. Depending on the argument `rep`, the results of `fun` are repeated for each element of `x` in the group (TRUE) or only once for each group (FALSE). Depending on the argument `rtn.grp`, the return object is a data.frame and the groups within `grp` are included in the data.frame as columns (TRUE) or the return object is an atomic vector and the groups are the names (FALSE).

Usage

```
agg(x, grp, rep = TRUE, rtn.grp = !rep, sep = "_", fun, ...)
```

Arguments

x	atomic vector.
grp	atomic vector or list of atomic vectors (e.g., data.frame) specifying the groups. The atomic vector(s) must be the length of x or else an error is returned.
rep	logical vector of length 1 specifying whether the result of fun should be repeated for every instance of the group in x (TRUE) or only once for each group (FALSE).
rtn.grp	logical vector of length 1 specifying whether the groups (i.e., grp) should be included in the return object as columns. The default is the opposite of rep as traditionally it is most important to return the group columns when rep = FALSE.
sep	character vector of length 1 specifying what string should separate different group values when naming the return object. This argument is only used if grp is a list of atomic vectors (e.g., data.frame) AND rep = FALSE AND rtn.grp = FALSE.
fun	function to use for aggregation. This function is expected to return an atomic vector of length 1.
...	additional named arguments to fun.

Details

If rep = TRUE, then agg calls ave; if rep = FALSE, then agg calls aggregate.

Value

result of fun applied to x for each group within grp. The structure of the return object depends on the arguments rep and rtn.grp:

If rep = TRUE and rtn.grp = TRUE: then the return object is a data.frame with nrow = nrow(data) where the first columns are grp and the last column is the result of fun. If grp is not a list with names, then its colnames will be "Group.1", "Group.2", "Group.3" etc. similar to aggregate's return object. The colname for the result of fun will be "x".

If rep = TRUE and rtn.grp = FALSE: then the return object is an atomic vector with length = length(x) where the values are the result of fun and the names = names(x).

If rep = FALSE and rtn.grp = TRUE: then the return object is a data.frame with nrow = length(levels(interaction(gr where the first columns are the unique group combinations in grp and the last column is the result of fun. If grp is not a list with names, then its colnames will be "Group.1", "Group.2", "Group.3" etc. similar to aggregate's return object. The colname for the result of fun will be "x".

If rep = FALSE and rtn.grp = FALSE: then the return object is an atomic vector with length length(levels(interaction(grp))) where the values are the result of fun and the names are each group value pasted together by sep if there are multiple grouping variables within grp (i.e., is.list(grp) && length(grp) > 2).

See Also

aggs, agg_dfm, ave, aggregate,

Examples

```
# one grouping variable
agg(x = airquality$"Solar.R", grp = airquality$"Month", fun = mean)
agg(x = airquality$"Solar.R", grp = airquality$"Month", fun = mean,
   na.rm = TRUE) # ignoring missing values
agg(x = setNames(airquality$"Solar.R", nm = row.names(airquality)), grp = airquality$"Month",
   fun = mean, na.rm = TRUE) # keeps the names in the return object
agg(x = airquality$"Solar.R", grp = airquality$"Month", rep = FALSE,
   fun = mean, na.rm = TRUE) # do NOT repeat aggregated values
agg(x = airquality$"Solar.R", grp = airquality$"Month", rep = FALSE, rtn.grp = FALSE,
   fun = mean, na.rm = TRUE) # groups are the names of the returned atomic vector

# two grouping variables
tmp_nm <- c("vs","am") # Roxygen2 doesn't like a c() within a []
agg(x = mtcars$"mpg", grp = mtcars[tmp_nm], rep = TRUE, fun = sd)
agg(x = mtcars$"mpg", grp = mtcars[tmp_nm], rep = FALSE,
   fun = sd) # do NOT repeat aggregated values
agg(x = mtcars$"mpg", grp = mtcars[tmp_nm], rep = FALSE, rtn.grp = FALSE,
   fun = sd) # groups are the names of the returned atomic vector
agg(x = mtcars$"mpg", grp = mtcars[tmp_nm], rep = FALSE, rtn.grp = FALSE,
   sep = ".", fun = sd) # change the separater for naming

# error messages
## Not run:
   agg(x = airquality$"Solar.R", grp = mtcars[tmp_nm]) # error returned
   # b/c  atomic vectors within \code{grp} not having the same length as \code{x}

## End(Not run)
```

---

aggs *Aggregate Data by Group*

Description

aggs evaluates a function separately for each group and combines the results back together into a data.frame that is returned. Depending on rep, the results of fun are repeated for each element of data[vrb.nm] in the group (TRUE) or only once for each group (FALSE). Note, aggs evaluates fun separately for each variable vrb.nm within data. If instead, you want to evaluate fun for variables as a set data[vrb.nm], then use agg_dfm.

Usage

```
aggs(
  data,
  vrb.nm,
  grp.nm,
  rep = TRUE,
  rtn.grp = !rep,
```

```
  sep = "_",
  suffix = "_a",
  fun,
  ...
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.
grp.nm	character vector of colnames from data specifying the groups.
rep	logical vector of length 1 specifying whether the result of fun should be repeated for every instance of the group in data[vrb.nm] (TRUE) or only once for each group (FALSE).
rtn.grp	logical vector of length 1 specifying whether the group columns (i.e., data[grp.nm]) should be included in the return object as columns. The default is the opposite of rep as traditionally it is most important to return the group columns when rep = FALSE.
sep	character vector of length 1 specifying what string should separate different group values when naming the return object. This argument is only used if grp.nm has length > 1 AND rep = FALSE AND rtn.grp = FALSE.
suffix	character vector of length 1 specifying the string to append to the end of the colnames in the return object.
fun	function to use for aggregation. This function is expected to return an atomic vector of length 1.
...	additional named arguments to fun.

Details

If rep = TRUE, then agg calls ave; if rep = FALSE, then agg calls aggregate.

Value

data.frame of aggregated values. If rep is TRUE, then nrow = nrow(data). If rep = FALSE, then nrow = length(levels(interaction(data[grp.nm]))). The names are specified by paste0(vrb.nm, suffix). If rtn.grp = TRUE, then the group columns are appended to the begining of the data.frame.

See Also

agg, agg_dfm, ave, aggregate,

Examples

```
aggs(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
   fun = mean, na.rm = TRUE)
aggs(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
```

```
   rtn.grp = TRUE, fun = mean, na.rm = TRUE) # include the group columns
aggs(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
   rep = FALSE, fun = mean, na.rm = TRUE) # do NOT repeat aggregated values
aggs(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   rep = FALSE, fun = mean, na.rm = TRUE) # with multiple group columns
aggs(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   rep = FALSE, rtn.grp = FALSE, fun = mean, na.rm = TRUE) # without returning groups
```

---

agg_dfm *Data Information by Group*

Description

agg_dfm evaluates a function on a set of variables in a data.frame separately for each group and combines the results back together. The rep and rtn.grp arguments determine exactly how the results are combined together. If rep = TRUE, then the result of fun is repeated for every row of the group in data[grp.nm]; If rep = FALSE, then the result of fun for each unique combination of data[grp.nm] is returned once. If rtn.grp = TRUE, then the results are returned in a data.frame where the first columns are the groups from data[grp.nm]; If rtn.grp = FALSE, then the results are returned in an atomic vector. Note, agg_dfm evaluates fun on all the variables in data[vrb.nm] as a whole, If instead, you want to evaluate fun separately for variable vrb.nm in data, then use Agg.

Usage

```
agg_dfm(
  data,
  vrb.nm,
  grp.nm,
  rep = FALSE,
  rtn.grp = !rep,
  sep = ".",
  rtn.result.nm = "result",
  fun,
  ...
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the set of variables to evaluate fun on.
grp.nm	character vector of colnames from data specifying the groups.
rep	logical vector of length 1 specifying whether the result of fun should be repeated for every instance of the group in data[vrb.nm] (TRUE) or only once for each group (FALSE).

rtn.grp	logical vector of length 1 specifying whether the group columns (i.e., data[grp.nm]) should be included in the return object as columns. The default is the opposite of rep as traditionally it is most important to return the group columns when rep = FALSE.
sep	character vector of length 1 specifying the string to paste the group values together with when there are multiple grouping variables (i.e., length(grp.nm) > 1). Only used if rep = FALSE and rtn.grp = FALSE.
rtn.result.nm	character vector of length 1 specifying the name for the column of results in the return object. Only used if rtn.grp = TRUE.
fun	function to evaluate each grouping of data[vrb.nm] by. This function must return an atomic vector of length 1. If not, then consider using by2 or plyr::dlply.
...	additional named arguments to fun.

Details

If rep = TRUE, then agg_dfm calls ave_dfm; if rep = FALSE, then agg_dfm calls by. When rep = FALSE and rtn.grp = TRUE, agg_dfm is very similar to plyr::ddply; when rep = FALSE and rtn.grp = FALSE, then agg_dfm is very similar to plyr::daply.

Value

result of fun applied to each grouping of data[vrb.nm]. The structure of the return object depends on the arguments rep and rtn.grp.

If rep = TRUE and rtn.grp = TRUE: then the return object is a data.frame with nrow = nrow(data) where the first columns are data[grp.nm] and the last column is the result of fun with colname = rtn.result.nm.

If rep = TRUE and rtn.grp = FALSE: then the return object is an atomic vector with length = nrow(data) where the values are the result of fun and the names = row.names(data).

If rep = FALSE and rtn.grp = TRUE: then the return object is a data.frame with nrow = length(levels(interaction(da where the first columns are the unique group combinations in data[grp.nm] and the last column is the result of fun with colname = rtn.result.nm.

If rep = FALSE and rtn.grp = FALSE: then the return object is an atomic vector with length length(levels(interaction(data[grp.nm]))) where the values are the result of fun and the names are each group value pasted together by sep if there are multiple grouping variables (i.e., length(grp.nm) > 2).

See Also

agg aggs by2 ddply daply

Examples

```
### one grouping variable

## by in base R
by(data = airquality[c("Ozone","Solar.R")], INDICES = airquality["Month"],
   simplify = FALSE, FUN = function(dat) cor(dat, use = "complete")[1,2])
```

```
## rep = TRUE

# rtn.group = TRUE
agg_dfm(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
   rep = TRUE, rtn.grp = TRUE, fun = function(dat) cor(dat, use = "complete")[1,2])

# rtn.group = FALSE
agg_dfm(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
   rep = TRUE, rtn.grp = FALSE, fun = function(dat) cor(dat, use = "complete")[1,2])

## rep = FALSE

# rtn.group = TRUE
agg_dfm(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
   rep = FALSE, rtn.grp = TRUE, fun = function(dat) cor(dat, use = "complete")[1,2])
suppressWarnings(plyr::ddply(.data = airquality[c("Ozone","Solar.R","Month")],
   .variables = "Month", .fun = function(dat) cor(dat, use = "complete")[1,2]))

# rtn.group = FALSE
agg_dfm(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
   rep = FALSE, rtn.grp = FALSE, fun = function(dat) cor(dat, use = "complete")[1,2])
suppressWarnings(plyr::daply(.data = airquality[c("Ozone","Solar.R","Month")],
   .variables = "Month", .fun = function(dat) cor(dat, use = "complete")[1,2]))

### two grouping variables

## by in base R
by(data = mtcars[c("mpg","cyl","disp")], INDICES = mtcars[c("vs","am")],
   FUN = nrow, simplify = FALSE) # with multiple group columns

## rep = TRUE

# rtn.grp = TRUE
agg_dfm(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   rep = TRUE, rtn.grp = TRUE, fun = nrow)

# rtn.grp = FALSE
agg_dfm(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   rep = TRUE, rtn.grp = FALSE, fun = nrow)

## rep = FALSE

# rtn.grp = TRUE
agg_dfm(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   rep = FALSE, rtn.grp = TRUE, fun = nrow)
agg_dfm(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   rep = FALSE, rtn.grp = TRUE, rtn.result.nm = "value", fun = nrow)

# rtn.grp = FALSE
agg_dfm(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   rep = FALSE, rtn.grp = FALSE, fun = nrow)
agg_dfm(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
```

```
rep = FALSE, rtn.grp = FALSE, sep = "_", fun = nrow)
```

---

amd_bi *Amount of Missing Data - Bivariate (Pairwise Deletion)*

Description

amd_bi by default computes the proportion of missing data for pairs of variables in a data.frame, with arguments to allow for counts instead of proportions (i.e., prop) or observed data rather than missing data (i.e., ov). It is bivariate in that each pair of variables is treated in isolation.

Usage

```
amd_bi(data, vrb.nm, prop = TRUE, ov = FALSE)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of the colnames from data specifying the variables.
prop	logical vector of length 1 specifying whether the frequency of missing values should be returned as a proportion (TRUE) or a count (FALSE).
ov	logical vector of length 1 specifying whether the frequency of observed values (TRUE) should be returned rather than the frequency of missing values (FALSE).

Value

data.frame of nrow = ncol = length(vrb.nm) and rowames = colnames = vrb.nm providing the frequency of missing (or observed if ov = TRUE) values per pair of variables. If prop = TRUE, the values will range from 0 to 1. If prop = FALSE, the values will range from 0 to nrow(data).

See Also

amd_bi amd_multi

Examples

```
amd_bi(data = airquality, vrb.nm = names(airquality)) # proportion of missing data
amd_bi(data = airquality, vrb.nm = names(airquality),
   ov = TRUE) # proportion of observed data
amd_bi(data = airquality, vrb.nm = names(airquality),
   prop = FALSE) # count of missing data
amd_bi(data = airquality, vrb.nm = names(airquality),
   prop = FALSE, ov = TRUE) # count of observed data
```

amd_multi *Amount of Missing Data - Multivariate (Listwise Deletion)*

Description

amd_multi by default computes the proportion of missing data from listwise deletion for a set of variables in a data.frame, with arguments to allow for counts instead of proportions (i.e., prop) or observed data rather than missing data (i.e., ov). It is multivariate in that the variables are treated together as a set.

Usage

```
amd_multi(data, vrb.nm, prop = TRUE, ov = FALSE)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of the colnames from data specifying the variables.
prop	logical vector of length 1 specifying whether the frequency of missing values should be returned as a proportion (TRUE) or a count (FALSE).
ov	logical vector of length 1 specifying whether the frequency of observed values (TRUE) should be returned rather than the frequency of missing values (FALSE).

Value

numeric vector of length 1 providing the frequency of missing (or observed if ov = TRUE) rows from listwise deletion for the set of variables vrb.nm. If prop = TRUE, the value will range from 0 to 1. If prop = FALSE, the value will range from 0 to nrow(data).

See Also

amd_uni amd_bi

Examples

```
amd_multi(airquality, vrb.nm = names(airquality)) # proportion of missing data
amd_multi(airquality, vrb.nm = names(airquality),
   ov = TRUE) # proportion of observed data
amd_multi(airquality, vrb.nm = names(airquality),
   prop = FALSE) # count of missing data
amd_multi(airquality, vrb.nm = names(airquality),
   prop = FALSE, ov = TRUE) # count of observed data
```

amd_uni *Amount of Missing Data - Univariate*

Description

amd_uni by default computes the proportion of missing data for variables in a data.frame, with arguments to allow for counts instead of proportions (i.e., prop) or observed data rather than missing data (i.e., ov). It is univariate in that each variable is treated in isolation. amd_uni is a simple wrapper for colNA.

Usage

```
amd_uni(data, vrb.nm, prop = TRUE, ov = FALSE)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of the colnames from data specifying the variables.
prop	logical vector of length 1 specifying whether the frequency of missing values should be returned as a proportion (TRUE) or a count (FALSE).
ov	logical vector of length 1 specifying whether the frequency of observed values (TRUE) should be returned rather than the frequency of missing values (FALSE).

Value

numeric vector of length = length(vrb.nm) and names = vrb.nm providing the frequency of missing (or observed if ov = TRUE) values per variable. If prop = TRUE, the values will range from 0 to 1. If prop = FALSE, the values will range from 0 to nrow(data).

See Also

amd_bi amd_multi

Examples

```
amd_uni(data = airquality, vrb.nm = names(airquality)) # proportion of missing data
amd_uni(data = airquality, vrb.nm = names(airquality),
   ov = TRUE) # proportion of observed data
amd_uni(data = airquality, vrb.nm = names(airquality),
   prop = FALSE) # count of missing data
amd_uni(data = airquality, vrb.nm = names(airquality),
   prop = FALSE, ov = TRUE) # count of observed data
```

auto_by *Autoregressive Coefficient by Group*

Description

auto_by computes the autoregressive coefficient by group for longitudinal data where each observation within the group represents a different timepoint. The function assumes the data are already sorted by time.

Usage

```
auto_by(
  x,
  grp,
  n = -1L,
  how = "cor",
  cw = TRUE,
  method = "pearson",
  use = "na.or.complete",
  REML = TRUE,
  control = NULL,
  sep = "."
)
```

Arguments

x	numeric vector.
grp	list of atomic vector(s) and/or factor(s) (e.g., data.frame), which each have same length as x. It can also be an atomic vector or factor, which will then be made the first element of a list.
n	integer vector with length 1. Specifies the direction and magnitude of the shift. See shift for details. The default is -1L, which is a one-lag autoregressive coefficient' +2L would be a two-lead autoregressive coefficient. The sign of n only affects the results for how = "lm", "lme", or "lmer".
how	character vector of length 1 specifying how to compute the autoregressive coefficients. The options are 1) "cor" for correlation with the cor function, 2) "cov" for covariance with the cov function, 3) "lm" for the linear regression slope with the lm function, 4) "lme" for empirical Bayes estimates from a linear mixed effects model with the lme function, 5) "lmer" for empirical Bayes estimates from a linear mixed effects model with the lmer function.
cw	logical vector of length 1 specifying whether the shifted vector should be group-mean centered (TRUE) or not (FALSE). This only affects the results for how = "lme" or "lmer".
method	character vector of length 1 specifying the type of correlation or covariance to compute. Only used when how = "cor" or "cov". See cor for details.

use	character vector of length 1 specifying how to handle missing data. Only used when how = "cor" or "cov". See cor for details.
REML	logical vector of length 1 specifying whether to use restricted estimated maximum liklihood (TRUE) rather than traditional maximum likelihood (FALSE). Only used when how = "lme" or "lmer".
control	list of control parameters for lme or lmer when how = "lme" or "lmer", respectively. See lmeControl and lmerControl for details.
sep	character vector of length 1 specifying what string should separate different group values when naming the return object. This argument is only used if grp is a list of atomic vectors (e.g., data.frame).

Details

There are several different ways to estimate the autoregressive parameter. This function offers a variety of ways with the how and cw arguments. Note, that a recent simulation suggests that group-mean centering via cw is the best approach when using linear mixed effects modeling via how = "lme" or "lmer" (Hamaker & Grasman, 2015).

Value

numeric vector of autoregressive coefficients with length = `length(levels(interaction(grp)))` and names = pasteing of the grouping value(s) together separated by sep.

References

Hamaker, E. L., & Grasman, R. P. (2015). To center or not to center? Investigating inertia with a multilevel autoregressive model. Frontiers in Psychology, 5, 1492.

Examples

```
# cor
auto_by(x = airquality$"Ozone", grp = airquality$"Month", how = "cor")
auto_by(x = airquality$"Ozone", grp = airquality$"Month",
   n = -2L, how = "cor") # lag across 2 timepoints
auto_by(x = airquality$"Ozone", grp = airquality$"Month",
   n = +1L, how = "cor") # lag and lead identical for cor
auto_by(x = airquality$"Ozone", grp = airquality$"Month", how = "cor",
   cw = FALSE) # centering within-person identical for cor

# cov
auto_by(x = airquality$"Ozone", grp = airquality$"Month", how = "cov")
auto_by(x = airquality$"Ozone", grp = airquality$"Month",
   n = -2L, how = "cov") # lag across 2 timepoints
auto_by(x = airquality$"Ozone", grp = airquality$"Month",
   n = +1L, how = "cov") # lag and lead identical for cov
auto_by(x = airquality$"Ozone", grp = airquality$"Month", how = "cov",
   cw = FALSE) # centering within-person identical for cov

# lm
auto_by(x = airquality$"Ozone", grp = airquality$"Month", how = "lm")
```

```
auto_by(x = airquality$"Ozone", grp = airquality$"Month",
   n = -2L, how = "lm") # lag across 2 timepoints
auto_by(x = airquality$"Ozone", grp = airquality$"Month",
   n = +1L, how = "lm") # lag and lead NOT identical for lm
auto_by(x = airquality$"Ozone", grp = airquality$"Month", how = "lm",
   cw = FALSE) # centering within-person identical for lm

# lme
chick_weight <- as.data.frame(ChickWeight)
auto_by(x = chick_weight$"weight", grp = chick_weight$"Chick", how = "lme")
control_lme <- nlme::lmeControl(maxIter = 250L, msMaxIter = 250L,
   tolerance = 1e-3, msTol = 1e-3) # custom controls
auto_by(x = chick_weight$"weight", grp = chick_weight$"Chick", how = "lme",
   control = control_lme)
auto_by(x = chick_weight$"weight", grp = chick_weight$"Chick",
   n = -2L, how = "lme") # lag across 2 timepoints
auto_by(x = chick_weight$"weight", grp = chick_weight$"Chick",
   n = +1L, how = "lme") # lag and lead NOT identical for lme
auto_by(x = chick_weight$"weight", grp = chick_weight$"Chick", how = "lme",
   cw = FALSE) # centering within-person NOT identical for lme

# lmer
bryant_2016 <- as.data.frame(lmeInfo::Bryant2016)
## Not run:
auto_by(x = bryant_2016$"outcome", grp = bryant_2016$"case", how = "lmer")
control_lmer <- lme4::lmerControl(check.conv.grad = lme4:::.makeCC("stop",
   tol = 2e-3, relTol = NULL), check.conv.singular = lme4:::.makeCC("stop",
  tol = formals(lme4::isSingular)$"tol"), check.conv.hess = lme4:::.makeCC(action = "stop",
   tol = 1e-6)) # custom controls
auto_by(x = bryant_2016$"outcome", grp = bryant_2016$"case", how = "lmer",
   control = control_lmer) # TODO: for some reason lmer doesn't like this
   # and is not taking into account the custom controls
auto_by(x = bryant_2016$"outcome", grp = bryant_2016$"case",
   n = -2L, how = "lmer") # lag across 2 timepoints
auto_by(x = bryant_2016$"outcome", grp = bryant_2016$"case",
   n = +1L, how = "lmer") # lag and lead NOT identical for lmer
auto_by(x = bryant_2016$"outcome", grp = bryant_2016$"case", how = "lmer",
   cw = FALSE) # centering within-person NOT identical for lmer

## End(Not run)
```

---

ave_dfm *Repeated Group Statistics for a Data-Frame*

Description

ave_dfm evaluates a function on a set of variables vrb.nm separately for each group within grp.nm. The results are combined back together in line with the rows of data similar to ave. ave_dfm is different than ave or agg because it operates on a data.frame, not an atomic vector.

Usage

```
ave_dfm(data, vrb.nm, grp.nm, fun, ...)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames in data specifying the variables to use for the aggregation function fun.
grp.nm	character vector of colnames in data specifying the grouping variables.
fun	function that returns an atomic vector of length 1. Probably makes sense to ensure the function always returns the same typeof as well.
...	additional named arguments to fun.

Value

atomic vector of length = nrow(data) providing the result of the function fun for the subset of data with that group value (i.e., data[levels(interaction(data[grp.nm]))[i], vrb.nm]) for that row.

See Also

ave for the same functionality with atomic vector inputs agg_dfm for similar functionality with data.frames, but can return the result for each group once rather than repeating the result for each group value in the data.frame

Examples

```
# one grouping variables
ave_dfm(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month",
   fun = function(dat) cor(dat, use = "complete")[1,2])

# two grouping variables
ave_dfm(data = mtcars, vrb.nm = c("mpg","cyl","disp"), grp.nm = c("vs","am"),
   fun = nrow) # with multiple group columns
```

---

boot_ci *Bootstrapped Confidence Intervals from a Matrix of Coefficients*

Description

boot_ci computes bootstrapped confidence intervals from a matrix of coefficients (or any statistical information of interest). The function is an alternative to confint2.boot for when the user does not have an object of class boot, but rather creates their own matrix of coefficients. It has limited types of bootstrapped confidence intervals at the moment, but future versions are expected to have more options.

Usage

```
boot_ci(coef, est = colMeans(coef), boot.ci.type = "perc2", level = 0.95)
```

Arguments

coef	numeric matrix (or data.frame of numeric columns) of coefficients. The rows correspond to each bootstrapped resample and the columns to different coefficients. This is the equivalent of the "t" element in a boot object.
est	numeric vector of observed coefficients from the full sample. This is the equivalent of the "t0" element in a boot object. The default takes the mean of each coefficient across bootstrapped resamples; however, this usually results in small amount of bias in the coefficients.
boot.ci.type	character vector of length 1 specifying the type of bootstrapped confidence interval to compute. The options are 1) "perc2" for the naive percentile method using quantile, and 2) "norm2" for the normal method that uses the bootstrapped standard error to construct symmetrical confidence intervals with the classic formula around the estimate, The options have a "2" after them because, although they are conceptually similar to the "perc" and "norm" methods in the boot.ci function, they are slightly different mathematically.
level	double vector of length 1 specifying the confidence level. Must be between 0 and 1.

Value

data.frame will be returned with nrow equal to the number of coefficients bootstrapped and columns specified below. The rownames are the colnames in the coef argument or the names in the est argument (default data.frame rownames if neither have any names). The columns are the following:

est original parameter estimates

se bootstrapped standard errors (does not differ by boot.ci.type)

lwr lower bound of the bootstrapped confidence intervals

upr upper bound of the bootstrapped confidence intervals

See Also

boot.ci for the confidence interval function in the boot package, confint.boot for an alternative function with boot objects

Examples

```
tmp <- replicate(n = 100, expr = {
   i <- sample.int(nrow(attitude), replace = TRUE)
   colMeans(attitude[i, ])
}, simplify = FALSE)
mat <- str2str::lv2m(tmp, along = 1)
boot_ci(mat, est = colMeans(attitude))
```

by2 *Apply a Function to Data by Group*

Description

by2 applies a function to data by group and is an alternative to the base R function by. The function is apart of the split-apply-combine type of function discussed in the plyr R package and is very similar to dlply. It splits up one data.frame .data[.vrb.nm]into a data.frame for each group in .data[.grp.nm], applies a function .fun to each data.frame, and then returns the results as a list with names equal to the group values unique(interaction(.data[.grp.nm], sep = .sep)). by2 is simply split.data.frame + lapply. Similar to dlply, The arguments all start with . so that they do not conflict with arguments from the function .fun. If you want to apply a function a (atomic) vector rather than data.frame, then use tapply2.

Usage

```
by2(.data, .vrb.nm, .grp.nm, .sep = ".", .fun, ...)
```

Arguments

.data	data.frame of data.
.vrb.nm	character vector specifying the colnames of .data to select the set of variables to apply .fun to.
.grp.nm	character vector specifying the colnames of .data to select the grouping variables.
.sep	character vector of length 1 specifying the string to combine the group values together with. .sep is only used if there are multiple grouping variables (i.e., length(.grp.nm) > 1).
.fun	function to apply to the set of variables .data[.vrb.nm] for each group.
...	additional named arguments to pass to .fun.

Value

list of objects containing the return object of .fun for each group. The names are the unique combinations of the grouping variables (i.e., unique(interaction(.data[.grp.nm], sep = .sep))).

See Also

by tapply2 dlply

Examples

```
# one grouping variable
by2(mtcars, .vrb.nm = c("mpg","cyl","disp"), .grp.nm = "vs",
   .fun = cov, use = "complete.obs")
```

```
# two grouping variables
x <- by2(mtcars, .vrb.nm = c("mpg","cyl","disp"), .grp.nm = c("vs","am"),
   .fun = cov, use = "complete.obs")
print(x)
str(x)

# compare to by
vrb_nm <- c("mpg","cyl","disp") # Roxygen runs the whole script if I put a c() in a []
grp_nm <- c("vs","am") # Roxygen runs the whole script if I put a c() in a []
y <- by(mtcars[vrb_nm], INDICES = mtcars[grp_nm],
   FUN = cov, use = "complete.obs", simplify = FALSE)
str(y) # has dimnames rather than names
```

`center` *Centering and/or Standardizing a Numeric Vector*

Description

`center` centers and/or standardized a numeric vector. It is an alternative to `scale.default` that returns a numeric vector rather than a numeric matrix.

Usage

```
center(x, center = TRUE, scale = FALSE)
```

Arguments

`x`	numeric vector.
`center`	logical vector with length 1 specifying whether grand-mean centering should be done.
`scale`	logical vector with length 1 specifying whether grand-SD scaling should be done.

Details

`center` first coerces `x` to a matrix in preparation for the call to `scale.default`. If the coercion results in a non-numeric matrix (e.g., `x` is a character vector or factor), then an error is returned.

Value

numeric vector of `x` centered and/or standardized with the same names as `x`.

See Also

`centers` `center_by` `centers_by` `scale.default`

Examples

```
center(x = mtcars$"disp")
center(x = mtcars$"disp", scale = TRUE)
center(x = mtcars$"disp", center = FALSE, scale = TRUE)
center(x = setNames(mtcars$"disp", nm = row.names(mtcars)))
```

---

centers *Centering and/or Standardizing Numeric Data*

Description

`centers` centers and/or standardized data. It is an alternative to `scale.default` that returns a data.frame rather than a numeric matrix.

Usage

```
centers(data, vrb.nm, center = TRUE, scale = FALSE, suffix)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`center`	logical vector with length 1 specifying whether grand-mean centering should be done.
`scale`	logical vector with length 1 specifying whether grand-SD scaling should be done.
`suffix`	character vector with a single element specifying the string to append to the end of the colnames of the return object. The default depends on the `center` and `scale` arguments: 1)if `center` = TRUE and `scale` = FALSE, then `suffix` = "_c", 2) if `center` = FALSE and `scale` = TRUE, then `suffix` = "_s", 3) if `center` = TRUE and `scale` = TRUE, then `suffix` = "_z", 4) if `center` = FALSE and `scale` = FALSE, then `suffix` = "".

Details

`centers` first coerces `data[vrb.nm]` to a matrix in preparation for the call to `scale.default`. If the coercion results in a non-numeric matrix (e.g., any columns in `data[vrb.nm]` are character vectors or factors), then an error is returned.

Value

data.frame of centered and/or standardized variables with colnames specified by `paste0(vrb.nm, suffix)`.

See Also

`center` `centers_by` `center_by` `scale.default`

Examples

```
centers(data = mtcars, vrb.nm = c("disp","hp","drat","wt","qsec"))
centers(data = mtcars, vrb.nm = c("disp","hp","drat","wt","qsec"),
   scale = TRUE)
centers(data = mtcars, vrb.nm = c("disp","hp","drat","wt","qsec"),
   center = FALSE, scale = TRUE)
centers(data = mtcars, vrb.nm = c("disp","hp","drat","wt","qsec"),
   scale = TRUE, suffix = "_std")
```

---

centers_by *Centering and/or Standardizing Numeric Data by Group*

Description

centers_by centers and/or standardized data by group. This is sometimes called group-mean centering and/or group-SD standardizing. The groups can be specified by multiple columns in data (e.g., grp.nm with length > 1), and interaction will be implicitly called to create the groups.

Usage

```
centers_by(data, vrb.nm, grp.nm, center = TRUE, scale = FALSE, suffix)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.
grp.nm	character vector of colnames from data specifying the groups.
center	logical vector with length 1 specifying whether group-mean centering should be done.
scale	logical vector with length 1 specifying whether group-SD scaling should be done.
suffix	character vector with a single element specifying the string to append to the end of the colnames of the return object. The default depends on the center and scale arguments: 1)if center = TRUE and scale = FALSE, then suffix = "_cw", 2) if center = FALSE and scale = TRUE, then suffix = "_sw", 3) if center = TRUE and scale = TRUE, then suffix = "_zw", 4) if center = FALSE and scale = FALSE, then suffix = "".

Details

centers_by first coerces data[vrb.nm] to a matrix in preparation for the core of the function, which is essentially lapply(X = split(x = data[vrb.nm], f = data[grp.nm]), FUN = scale.default) If the coercion results in a non-numeric matrix (e.g., any columns in data[vrb.nm] are character vectors or factors), then an error is returned.

Value

data.frame of centered and/or standardized variables by group with colnames specified by `paste0(vrb.nm, suffix)`.

See Also

`center_by` `centers` `center` `scale.default`

Examples

```
ChickWeight2 <- as.data.frame(ChickWeight) # because the "groupedData" class calls
   # `[.groupedData`, which is different than `[.data.frame`
row.names(ChickWeight2) <- as.numeric(row.names(ChickWeight)) / 1000
centers_by(data = ChickWeight2, vrb.nm = c("weight","Time"), grp.nm = "Chick")
centers_by(data = ChickWeight2, vrb.nm = c("weight","Time"), grp.nm = "Chick",
   scale = TRUE, suffix = "_within")
centers_by(data = as.data.frame(CO2), vrb.nm = c("conc","uptake"),
   grp.nm = c("Type","Treatment"), scale = TRUE) # multiple grouping columns
```

---

center_by *Centering and/or Standardizing a Numeric Vector by Group*

Description

`center_by` centers and/or standardized a numeric vector by group. This is sometimes called group-mean centering and/or group-SD standardizing.

Usage

```
center_by(x, grp, center = TRUE, scale = FALSE)
```

Arguments

`x`	numeric vector.
`grp`	list of atomic vector(s) and/or factor(s) (e.g., data.frame) containing the groups. They should each have same length as `x`. It can also be an atomic vector or factor, which will then be made the first element of a list internally.
`center`	logical vector with length 1 specifying whether group-mean centering should be done.
`scale`	logical vector with length 1 specifying whether group-SD scaling should be done.

Details

`center_by` first coerces `x` to a matrix in preparation for the core of the function, which is essentially: `lapply(X = split(x = x, f = grp), FUN = scale.default)`. If the coercion results in a non-numeric matrix (e.g., `x` is a character vector or factor), then an error is returned. An error is also returned if `x` and the elements of `grp` do not have the same length.

Value

numeric vector of x centered and/or standardized by group with the same names as x.

See Also

centers_by center centers scale.default

Examples

```
chick_data <- as.data.frame(ChickWeight) # because the "groupedData" class calls
   # `[.groupedData`, which is different than `[.data.frame`
center_by(x = ChickWeight[["weight"]], grp = ChickWeight[["Chick"]])
center_by(x = setNames(obj = ChickWeight[["weight"]], nm = row.names(ChickWeight)),
   grp = ChickWeight[["Chick"]]) # with names
tmp_nm <- c("Type","Treatment") # b/c Roxygen2 doesn't like a c() within a []
center_by(x = as.data.frame(CO2)[["uptake"]], grp = as.data.frame(CO2)[tmp_nm],
   scale = TRUE) # multiple grouping vectors
```

---

change *Change Score from a Numeric Vector*

Description

change creates a change score (aka difference score) from a numeric vector. It is assumed that the vector is already sorted by time such that the first element is earliest in time and the last element is the latest in time.

Usage

```
change(x, n, undefined = NA)
```

Arguments

x	numeric vector.
n	integer vector with length 1. Specifies how the change score is calculated. If n is positive, then the change score is calculated from lead - original; if n is negative, then the change score is calculated from original - lag. The magnitude of n determines how many elements are shifted for the lead/lag within the calculation. If n is zero, then change simply returns a vector or zeros. See details of shift.
undefined	atomic vector with length 1 (probably makes sense to be the same typeof as x). Specifies what to insert for undefined values after the shifting takes place. See details of shift.

Details

It is recommended to use L when specifying n to prevent problems with floating point numbers. shift tries to circumvent this issue by a call to round within shift if n is not an integer; however that is not a complete fail safe. The problem is that as.integer(n) implicit in shift truncates rather than rounds. See details of shift.

Value

an atomic vector of the same length as x that is the change score. If x and undefined are different typeofs, then the return will be coerced to the most complex typeof (i.e., complex to simple: character, double, integer, logical).

See Also

changes change_by changes_by shift

Examples

```
change(x = attitude[[1]], n = -1L) # use L to prevent problems with floating point numbers
change(x = attitude[[1]], n = -2L) # can specify any integer up to the length of `x`
change(x = attitude[[1]], n = +1L) # can specify negative or positive integers
change(x = attitude[[1]], n = +2L, undefined = -999) # user-specified indefined value
change(x = attitude[[1]], n = -2L, undefined = -999) # user-specified indefined value
change(x = attitude[[1]], n = 0L) # returns a vector of zeros
## Not run:
change(x = setNames(object = letters, nm = LETTERS), n = 3L) # character vector returns an error

## End(Not run)
```

---

changes *Change Scores from Numeric Data*

Description

changes creates change scores (aka difference scores) from numeric data. It is assumed that the data is already sorted by time such that the first row is earliest in time and the last row is the latest in time. changes is a multivariate version of change that operates on multiple variabes rather than just one.

Usage

```
changes(data, vrb.nm, n, undefined = NA, suffix)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.
n	integer vector with length 1. Specifies how the change score is calculated. If n is positive, then the change score is calculated from lead - original; if n is negative, then the change score is calculated from original - lag. The magnitude of n determines how many rows are shifted for the lead/lag within the calculation. See details of shifts.
undefined	atomic vector with length 1 (probably makes sense to be the same typeof as x). Specifies what to insert for undefined values after the shifting takes place. See details of shifts.

suffix — character vector of length 1 specifying the string to append to the end of the colnames of the return object. The default depends on the n argument: 1) if n < 0, then suffix = paste0("_hg", -n), 2) if n > 0, then suffix = paste0("_hd", +n), 3) if n = 0, then suffix = "".

Details

It is recommended to use L when specifying n to prevent problems with floating point numbers. shifts tries to circumvent this issue by a call to round within shifts if n is not an integer; however that is not a complete fail safe. The problem is that as.integer(n) implicit in shifts truncates rather than rounds. See details of shifts.

Value

data.frame of change scores with colnames specified by paste0(vrb.nm, suffix).

See Also

change changes_by change_by shifts

Examples

```
changes(attitude, vrb.nm = names(attitude),
   n = -1L) # use L to prevent problems with floating point numbers
changes(attitude, vrb.nm = names(attitude),
   n = -2L) # can specify any integer up to the length of `x`
changes(attitude, vrb.nm = names(attitude),
   n = +1L) # can specify negative or positive integers
changes(attitude, vrb.nm = names(attitude),
   n = +2L, undefined = -999) # user-specified indefined value
changes(attitude, vrb.nm = names(attitude),
   n = -2L, undefined = -999) # user-specified indefined value
## Not run:
changes(str2str::d2d(InsectSprays), names(InsectSprays),
  n = 3L) # character vector returns an error

## End(Not run)
```

---

changes_by *Change Scores from Numeric Data by Group*

Description

changes_by creates change scores (aka difference scores) from numeric data separately for each group. It is assumed that the data is already sorted within each group by time such that the first row for that group is earliest in time and the last row for that group is the latest in time.

Usage

```
changes_by(data, vrb.nm, grp.nm, n, undefined = NA, suffix)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.
grp.nm	character vector of colnames from data specifying the groups.
n	integer vector with length 1. Specifies how the change score is calculated. If n is positive, then the change score is calculated from lead - original; if n is negative, then the change score is calculated from original - lag. The magnitude of n determines how many rows are shifted for the lead/lag within the calculation. See details of shifts_by.
undefined	atomic vector with length 1 (probably makes sense to be the same typeof as x). Specifies what to insert for undefined values after the shifting takes place. See details of shifts_by.
suffix	character vector of length 1 specifying the string to append to the end of the colnames of the return object. The default depends on the n argument: 1) if n < 0, then suffix = paste0("_hgw", -n), 2) if n > 0, then suffix = paste0("_hdw", +n), 3) if n = 0, then suffix = "".

Details

It is recommended to use L when specifying n to prevent problems with floating point numbers. shifts_by tries to circumvent this issue by a call to round within shifts_by if n is not an integer; however that is not a complete fail safe. The problem is that as.integer(n) implicit in shifts_by truncates rather than rounds. See details of shifts_by.

Value

data.frame of change scores by group with colnames specified by paste0(vrb.nm, suffix).

See Also

change_by changes change shifts_by

Examples

```
changes_by(data = ChickWeight, vrb.nm = c("weight","Time"), grp.nm = "Chick", n = -1L)
changes_by(data = mtcars, vrb.nm = c("disp","mpg"), grp.nm = c("vs","am"), n = 1L)
changes_by(data = as.data.frame(CO2), vrb.nm = c("conc","uptake"),
   grp.nm = c("Type","Treatment"), n = 2L) # multiple grouping columns
```

change_by *Change Scores from a Numeric Vector by Group*

Description

change_by creates a change score (aka difference score) from a numeric vector separately for each group. It is assumed that the vector is already sorted within each group by time such that the first element for that group is earliest in time and the last element for that group is the latest in time.

Usage

```
change_by(x, grp, n, undefined = NA)
```

Arguments

x	numeric vector.
grp	list of atomic vector(s) and/or factor(s) (e.g., data.frame), which each have same length as x. It can also be an atomic vector or factor, which will then be made the first element of a list internally.
n	integer vector with length 1. Specifies how the change score is calculated. If n is positive, then the change score is calculated from lead - original; if n is negative, then the change score is calculated from original - lag. The magnitude of n determines how many rows are shifted for the lead/lag within the calculation. See details of shift_by.
undefined	atomic vector with length 1 (probably makes sense to be the same typeof as x). Specifies what to insert for undefined values after the shifting takes place. See details of shift_by.

Details

It is recommended to use L when specifying n to prevent problems with floating point numbers. shift_by tries to circumvent this issue by a call to round within shift_by if n is not an integer; however that is not a complete fail safe. The problem is that as.integer(n) implicit in shift_by truncates rather than rounds. See details of shift_by.

Value

an atomic vector of the same length as x that is the change score by group. If x and undefined are different typeofs, then the return will be coerced to the more complex typoof (i.e., complex to simple: character, double, integer, logical).

See Also

changes_by change changes shift_by

Examples

```
change_by(x = ChickWeight[["Time"]], grp = ChickWeight[["Chick"]], n = -1L)
tmp_nm <- c("vs","am") # multiple grouping vectors
change_by(x = mtcars[["disp"]], grp = mtcars[tmp_nm], n = +1L)
tmp_nm <- c("Type","Treatment") # multiple grouping vectors
change_by(x = as.data.frame(CO2)[["uptake"]], grp = as.data.frame(CO2)[tmp_nm], n = 2L)
```

---

colMeans_if *Column Means Conditional on Frequency of Observed Values*

Description

`colMeans_if` calculates the mean of every column in a numeric or logical matrix conditional on the frequency of observed data. If the frequency of observed values in that column is less than (or equal to) that specified by `ov.min`, then NA is returned for that row.

Usage

```
colMeans_if(x, ov.min = 1, prop = TRUE, inclusive = TRUE)
```

Arguments

`x`	numeric or logical matrix. If not a matrix, it will be coerced to one.
`ov.min`	minimum frequency of observed values required per column. If `prop` = TRUE, then this is a decimal between 0 and 1. If `prop` = FALSE, then this is a integer between 0 and `nrow(x)`.
`prop`	logical vector of length 1 specifying whether `ov.min` should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
`inclusive`	logical vector of length 1 specifying whether the mean should be calculated if the frequency of observed values in a column is exactly equal to `ov.min`.

Details

Conceptually this function does: `apply(X = x, MARGIN = 2, FUN = mean_if, ov.min = ov.min, prop = prop, inclusive = inclusive)`. But for computational efficiency purposes it does not because then the missing values conditioning would not be vectorized. Instead, it uses `colMeans` and then inserts NAs for columns that have too few observed values.

Value

numeric vector of length = `ncol(x)` with names = `colnames(x)` providing the mean of each column or NA depending on the frequency of observed values.

See Also

`colSums_if` `rowMeans_if` `rowSums_if` `colMeans`

Examples

```
colMeans_if(airquality)
colMeans_if(x = airquality, ov.min = 150, prop = FALSE)
```

---

colNA *Frequency of Missing Values by Column*

Description

rowNA compute the frequency of missing values in a matrix by column. This function essentially does apply(X = x, MARGIN = 2, FUN = vecNA). It is also used by other functions in the quest package related to missing values (e.g., colMeans_if).

Usage

```
colNA(x, prop = FALSE, ov = FALSE)
```

Arguments

x	matrix with any typeof. If not a matrix, it will be coerced to a matrix via as.matrix. The function allows for colnames to carry over for non-matrix objects (e.g., data.frames).
prop	logical vector of length 1 specifying whether the frequency of missing values should be returned as a proportion (TRUE) or a count (FALSE).
ov	logical vector of length 1 specifying whether the frequency of observed values (TRUE) should be returned rather than the frequency of missing values (FALSE).

Value

numeric vector of length = ncol(x), and names = colnames(x) providing the frequency of missing values (or observed values if ov = TRUE) per column. If prop = TRUE, the values will range from 0 to 1. If prop = FALSE, the values will range from 1 to nrow(x).

See Also

is.na vecNA rowNA rowsNA

Examples

```
colNA(as.matrix(airquality)) # count of missing values
colNA(as.matrix(airquality), prop = TRUE) # proportion of missing values
colNA(as.matrix(airquality), ov = TRUE) # count of observed values
colNA(as.data.frame(airquality), prop = TRUE, ov = TRUE) # proportion of observed values
```

colSums_if *Column Sums Conditional on Frequency of Observed Values*

Description

colSums_if calculates the sum of every column in a numeric or logical matrix conditional on the frequency of observed data. If the frequency of observed values in that column is less than (or equal to) that specified by ov.min, then NA is returned for that column. It also has the option to return a value other than 0 (e.g., NA) when all columns are NA, which differs from colSums(x, na.rm = TRUE).

Usage

```
colSums_if(
  x,
  ov.min = 1,
  prop = TRUE,
  inclusive = TRUE,
  impute = TRUE,
  allNA = NA_real_
)
```

Arguments

x	numeric or logical matrix. If not a matrix, it will be coerced to one.
ov.min	minimum frequency of observed values required per column. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and nrow(x).
prop	logical vector of length 1 specifying whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
inclusive	logical vector of length 1 specifying whether the sum should be calculated if the frequency of observed values in a column is exactly equal to ov.min.
impute	logical vector of length 1 specifying if missing values should be imputed with the mean of observed values of x[, i]. If TRUE (default), this will make sums over the same rows with different amounts of observed data comparable.
allNA	numeric vector of length 1 specifying what value should be returned for columns that are all NA. This is most applicable when ov.min = 0 and inclusive = TRUE. The default is NA, which differs from colSums with na.rm = TRUE where 0 is returned. Note, the value is overwritten by NA if the frequency of observed values in that column is less than (or equal to) that specified by ov.min.

Details

Conceptually this function does: apply(X = x, MARGIN = 2, FUN = sum_if,ov.min = ov.min, prop = prop, inclusive = inclusive). But for computational efficiency purposes it does not because then the observed values conditioning would not be vectorized. Instead, it uses colSums and then inserts NAs for columns that have too few observed values.

Value

numeric vector of length = `ncol(x)` with names = `colnames(x)` providing the sum of each column or NA depending on the frequency of observed values.

See Also

`colMeans_if` `rowSums_if` `rowMeans_if` `colSums`

Examples

```
colSums_if(airquality)
colSums_if(x = airquality, ov.min = 150, prop = FALSE)
x <- data.frame("x" = c(1, 2, NA), "y" = c(1, NA, NA), "z" = c(NA, NA, NA))
colSums_if(x)
colSums_if(x, ov.min = 0)
colSums_if(x, ov.min = 0, allNA = 0)
identical(x = colSums(x, na.rm = TRUE),
   y = colSums_if(x, impute = FALSE, ov.min = 0, allNA = 0)) # identical to
   # colSums(x, na.rm = TRUE)
```

---

composite *Composite Reliability of a Score*

Description

`composite` computes the composite reliability coefficient (sometimes referred to as omega) for a set of variables/items. The composite reliability computed in `composite` assumes a undimensional factor model with no error covariances. In addition to the coefficient itself, its standard error and confidence interval are returned, the average standardized factor loading from the factor model and number of variables/items, and (optional) model fit indices of the factor model. Note, any reverse coded items need to be recoded ahead of time so that all variables/items are keyed in the same direction.

Usage

```
composite(
  data,
  vrb.nm,
  level = 0.95,
  std = FALSE,
  ci.type = "delta",
  boot.ci.type = "bca.simple",
  R = 200L,
  fit.measures = c("chisq", "df", "tli", "cfi", "rmsea", "srmr"),
  se = "standard",
  test = "standard",
  missing = "fiml",
  ...
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames in data specifying the set of variables/items.
level	double vector of length 1 with a value between 0 and 1 specifying what confidence level to use.
std	logical element of length 1 specifying if the composite reliability should be computed for the standardized version of the variables data[vrb.nm].
ci.type	character vector of length 1 specifying which type of confidence interval to compute. The "delta" option uses the delta method to compute a standard error and a symmetrical confidence interval. The "boot" option uses bootstrapping to compute an asymmetrical confidence interval as well as a (pseudo) standard error.
boot.ci.type	character vector of length 1 specifying which type of bootstrapped confidence interval to compute. The options are: 1) "norm", 2) "basic", 3) "perc", 4) "bca.simple". Only used if ci.type = "boot". See parameterEstimates and its boot.ci.type argument for details.
R	integer vector of length 1 specifying how many bootstrapped resamples to compute. Note, as the number of bootstrapped resamples increases, the computation time will increase. Only used if ci.type is "boot".
fit.measures	character vector specifying which model fit indices to include in the return object. The default option includes the chi-square test statistic ("chisq"), degrees of freedom ("df"), tucker-lewis index ("tli"), comparative fit index ("cfi"), root mean square error of approximation ("rmsea"), and standardized root mean residual ("srmr"). If NULL, then no model fit indices are included in the return object. See fitMeasures for details.
se	character vector of length 1 specifying which type of standard errors to compute. If ci.type = "boot", then the input value is ignored and set to "bootstrap". See lavOptions and its se argument for details.
test	character vector of length 1 specifying which type of test statistic to compute. If ci.type = "boot", then the input value is ignored and set to "bootstrap". See lavOptions and its test argument for details.
missing	character vector of length 1 specifying how to handle missing data. The default is "fiml" for full information maximum likelihood). See lavOptions and its missing argument for details.
...	other arguments passed to cfa. Use at your own peril as some argument values could cause the function to break.

Details

The factor model is estimated using the R package lavaan. The reliability coefficients are calculated based on the square of the sum of the factor loadings divided by the sum of the square of the sum of the factors loadings and the sum of the error variances (Raykov, 2001).

composite is only able to use the "ML" estimator at the moment and cannot model items as categorical/ordinal. However, different versions of standard errors and test statistics are possible. For example, the "MLM" estimator can be specified by se = "robust.sem" and test = "satorra.bentler"; the "MLR" estimator can be specified by se = "robust.huber.white" and test = "yuan.bentler.mplus". See lavOptions and scroll down to Estimation options.

Value

double vector where the first element is the composite reliability coefficient ("est") followed by its standard error ("se"), then its confidence interval ("lwr" and "upr"), the average standardized factor loading of the factor model ("average_l") and number of variables ("nvrb"), and finally any of the `fit.measures` requested.

References

Raykov, T. (2001). Estimation of congeneric scale reliability using covariance structure analysis with nonlinear constraints. British Journal of Mathematical and Statistical Psychology, 54(2), 315–323.

See Also

`composites` `cronbach`

Examples

```
# data
dat <- psych::bfi[1:250, 2:5] # the first item is reverse coded

# delta method CI
composite(data = dat, vrb.nm = names(dat), ci.type = "delta")
composite(data = dat, vrb.nm = names(dat), ci.type = "delta", level = 0.99)
composite(data = dat, vrb.nm = names(dat), ci.type = "delta", std = TRUE)
composite(data = dat, vrb.nm = names(dat), ci.type = "delta", fit.measures = NULL)
composite(data = dat, vrb.nm = names(dat), ci.type = "delta",
   se = "robust.sem", test = "satorra.bentler", missing = "listwise") # MLM estimator
composite(data = dat, vrb.nm = names(dat), ci.type = "delta",
  se = "robust.huber.white", test = "yuan.bentler.mplus", missing = "fiml") # MLR estimator

## Not run:
# bootstrapped CI
composite(data = dat, vrb.nm = names(dat), level = 0.95,
   ci.type = "boot") # slightly different estimate for some reason...
composite(data = dat, vrb.nm = names(dat), level = 0.95, ci.type = "boot",
  boot.ci.type = "perc", R = 250L) # probably want to use more resamples - this is just an example

## End(Not run)

# compare to semTools::reliability
psymet_obj <- composite(data = dat, vrb.nm = names(dat))
psymet_est <- unname(psymet_obj["est"])
lavaan_obj <- lavaan::cfa(model = make.latent(names(dat)), data = dat,
   std.lv = TRUE, missing = "fiml")
semTools_obj <- semTools::reliability(lavaan_obj)
semTools_est <- semTools_obj["omega", "latent"]
all.equal(psymet_est, semTools_est)
```

composites *Composite Reliability of Multiple Scores*

Description

composites computes the composite reliability coefficient (sometimes referred to as omega) for multiple sets of variables/items. The composite reliability computed in composites assumes a undimensional factor model for each set of variables/items with no error covariances. In addition to the coefficients themselves, their standard errors and confidence intervals are returned, the average standardized factor loading from the factor models and number of variables/items in each set, and (optional) model fit indices of the factor models. Note, any reverse coded items need to be recoded ahead of time so that all items are keyed in the same direction for each set of variables/items.

Usage

```
composites(
  data,
  vrb.nm.list,
  level = 0.95,
  std = FALSE,
  ci.type = "delta",
  boot.ci.type = "bca.simple",
  R = 200L,
  fit.measures = c("chisq", "df", "tli", "cfi", "rmsea", "srmr"),
  se = "standard",
  test = "standard",
  missing = "fiml",
  ...
)
```

Arguments

data	data.frame of data.
vrb.nm.list	list of character vectors containing colnames in data specifying the multiple sets of variables/items.
level	double vector of length 1 with a value between 0 and 1 specifying what confidence level to use.
std	logical element of length 1 specifying if the composite reliability should be computed for the standardized version of the variables/items data[unlist(vrb.nm.list)].
ci.type	character vector of length 1 specifying which type of confidence interval to compute. The "delta" option uses the delta method to compute a standard error and a symmetrical confidence interval. The "boot" option uses bootstrapping to compute an asymmetrical confidence interval as well as a (pseudo) standard error.
boot.ci.type	character vector of length 1 specifying which type of bootstrapped confidence interval to compute. The options are: 1) "norm", 2) "basic", 3) "perc", 4)

	"bca.simple". Only used if `ci.type` = "boot". See `parameterEstimates` and its `boot.ci.type` argument for details.
`R`	integer vector of length 1 specifying how many bootstrapped resamples to compute. Note, as the number of bootstrapped resamples increases, the computation time will increase. Only used if `ci.type` is "boot".
`fit.measures`	character vector specifying which model fit indices to include in the return object. The default option includes the chi-square test statistic ("chisq"), degrees of freedom ("df"), tucker-lewis index ("tli"), comparative fit index ("cfi"), root mean square error of approximation ("rmsea"), and standardized root mean residual ("srmr"). If NULL, then no model fit indices are included in the return object. See `fitMeasures` for details.
`se`	character vector of length 1 specifying which type of standard errors to compute. If ci.type = "boot", then the input value is ignored and implicitly set to "bootstrap". See `lavOptions` and its `se` argument for details.
`test`	character vector of length 1 specifying which type of test statistic to compute. If ci.type = "boot", then the input value is ignored and implicitly set to "bootstrap". See `lavOptions` and its `test` argument for details.
`missing`	character vector of length 1 specifying how to handle missing data. The default is "fiml" for full information maximum likelihood. See `lavOptions` and its `missing` argument for details.
`...`	other arguments passed to `cfa`. Use at your own peril as some argument values could cause the function to break.

Details

The factor models are estimated using the R package `lavaan`. The reliability coefficients are calculated based on the square of the sum of the factor loadings divided by the sum of the square of the sum of the factors loadings and the sum of the error variances (Raykov, 2001).

`composites` is only able to use the "ML" estimator at the moment and cannot model items as categorical/ordinal. However, different versions of standard errors and test statistics are possible. For example, the "MLM" estimator can be specified by `se` = "robust.sem" and `test` = "satorra.bentler"; the "MLR" estimator can be specified by `se` = "robust.huber.white" and `test` = "yuan.bentler.mplus". See `lavOptions` and scroll down to Estimation options for details.

Value

data.frame containing the composite reliability of each set of variables/items.

est estimate of the reliability coefficient

se standard error of the reliability coefficient

lwr lower bound of the confidence interval of the reliability coefficient

upr upper bound of the confidence interval of the reliability coefficient

average_l average standardized factor loading from the factor model

nvrb number of variables/items

??? any model fit indices requested by the `fit.measures` argument

References

Raykov, T. (2001). Estimation of congeneric scale reliability using covariance structure analysis with nonlinear constraints. British Journal of Mathematical and Statistical Psychology, 54(2), 315–323.

See Also

composite cronbachs

Examples

```
dat0 <- psych::bfi[1:250, ]
dat1 <- str2str::pick(x = dat0, val = c("A1","C4","C5","E1","E2","O2","O5",
   "gender","education","age"), not = TRUE, nm = TRUE)
vrb_nm_list <- lapply(X = str2str::sn(c("E","N","C","A","O")), FUN = function(nm) {
   str2str::pick(x = names(dat1), val = nm, pat = TRUE)})
composites(data = dat1, vrb.nm.list = vrb_nm_list)
## Not run:
start_time <- Sys.time()
composites(data = dat1, vrb.nm.list = vrb_nm_list, ci.type = "boot",
   R = 5000L) # the function is not optimized for speed at the moment
   # since it will bootstrap separately for each set of variables/items
end_time <- Sys.time()
print(end_time - start_time) # takes 10 minutes on my laptop

## End(Not run)
composites(data = attitude,
   vrb.nm.list = list(names(attitude))) # also works with only one set of variables/items
```

---

confint2 *Confidence Intervals from Statistical Information*

Description

confint2 is a generic function for creating confidence intervals from various statistical information (e.g., confint2.default) or object classes (e.g., confint2.boot). It is an alternative to the original confint generic function in the stats package.

Usage

```
confint2(obj, ...)
```

Arguments

obj	object of a particular class (e.g., "boot") or the first argument in the default method (e.g., the obj argument in confint2.default)
...	additional arguments specific to the particular method of confint2.

Value

depends on the particular method of `confint2`, but usually a data.frame with a column for the parameter estimate ("est"), standard error ("se"), lower bound of the confidence interval ("lwr"), and upper bound of the confidence interval ("upr").

See Also

`confint2.default` for the default method, `confint2.boot` for the `boot` method,

`confint2.boot` *Bootstrapped Confidence Intervals from a* `boot` *Object*

Description

`confint2.boot` is the `boot` method for the generic function `confint2` and computes bootstrapped confidence intervals from an object of class `boot` (aka an object returned by the function `boot`. The function is a simple wrapper for the car boot methods for the `summary` and `confint` generics. See `hist.boot` for details on those methods.

Usage

```
## S3 method for class 'boot'
confint2(obj, boot.ci.type = "perc", level = 0.95, ...)
```

Arguments

`obj`	an object of class `boot` (aka an object returned by the function `boot`).
`boot.ci.type`	character vector of length 1 specifying the type of bootstrapped confidence interval to compute. The options are 1) "perc" for the regular percentile method, 2) "bca" for bias-corrected and accelerated percentile method, 3) "norm" for the normal method that uses the bootstrapped standard error to construct symmetrical confidence intervals with the classic formula around the bias-corrected estimate, and 4) "basic" for the basic method. Note, "stud" for the studentized method is NOT an option. See `boot.ci` for details. Although a more informative link is the following blogpost on bootstrapped confidence intervals with the boot package https://www.r-bloggers.com/2019/09/understanding-bootstrap-confidence-interva
`level`	double vector of length 1 specifying the confidence level. Must be between 0 and 1.
`...`	This argument has no use. Technically, it is additional arguments for `confint2.boot`, but is only included for Roxygen2 to satisfy "checking S3 generic/method consistency".

Details

The bias-corrected and accelerated percentile method (`boot.ci.type` = "bca") will often fail if the number of bootstrapped resamples is less than the sample size. Even still, it can fail for other reasons. Following `car:::confint.boot`, `confint2.boot` gives a warning if the bias-corrected and accelerated percentile method fails for any statistic, and implicitly switches to the regular percentile method to prevent an error. When multiple statistics were bootstrapped, it might be that the bias-corrected and accelerated percentile method succeeded for most of the statistics and only failed for one statistic; however, `confint2.boot` will switch to using the regular percentile method for ALL the statistics. This may change in the future.

Value

data.frame will be returned with nrow equal to the number of statistics bootstrapped and columns specified below. The rownames are the names in the "t0" element of the `boot` object (default data.frame rownames if the "t0" element does not have any names). The columns are the following:

est original parameter estimates

se bootstrapped standard errors (does not differ by `boot.ci.type`)

lwr lower bound of the bootstrapped confidence intervals

upr upper bound of the bootstrapped confidence intervals

See Also

`boot.ci` `hist.boot`

Examples

```
# a single statistic
mean2 <- function(x, i) mean(x[i], na.rm = TRUE)
boot_obj <- boot::boot(data = attitude[[1]], statistic = mean2, R = 200L)
confint2.boot(boot_obj)
confint2.boot(boot_obj, boot.ci.type = "bca")
confint2.boot(boot_obj, level = 0.99)

# multiple statistics
colMeans2 <- function(dat, i) colMeans(dat[i, ], na.rm = TRUE)
boot_obj <- boot::boot(data = attitude, statistic = colMeans2, R = 200L)
confint2.boot(boot_obj)
confint2.boot(boot_obj, boot.ci.type = "bca")
confint2.boot(boot_obj, level = 0.99)
```

confint2.default *Confidence Intervals from Parameter Estimates and Standard Errors*

Description

confint2.default is the default method for the generic function confint2 and computes the statistical information for confidence intervals from parameter estimates, standard errors, and degrees of freedom. If degrees of freedom are not applicable or available, then df can be set to Inf (the default) and critical z-values rather than critical t-values will be used.

Usage

```
## Default S3 method:
confint2(obj, se, df = Inf, level = 0.95, ...)
```

Arguments

obj	numeric vector of parameter estimates. A better name for this argument would be est; however, uses of S3 generic functions requires the first argument to be the same name (i.e., obj) across methods.
se	numeric vector of standard errors. Must be the same length as obj.
df	numeric vector of degrees of freedom. Must have length 1 or the same length as obj and se. If degrees of freedom are not applicable or available, then df can be set to Inf (the default) and critical z-values rather than critical t-values will be used.
level	double vector of length 1 specifying the confidence level. Must be between 0 and 1.
...	This argument has no use. Technically, it is additional arguments for confint2.default, but is only included for Roxygen2 to satisfy "checking S3 generic/method consistency".

Value

data.frame with nrow equal to the lengths of obj and se. The rownames are taken from obj, unless obj does not have any names and then the rownames are taken from the names of se. If neither have names, then the rownames are automatic (i.e., 1:nrow()). The columns are the following:

est parameter estimates

se standard errors

lwr lower bound of the confidence intervals

upr upper bound of the confidence intervals

See Also

confint2.boot nhst

Examples

```
# single estimate
confint2.default(obj = 10, se = 3)

# multiple estimates
est <- colMeans(attitude)
se <- apply(X = str2str::d2m(attitude), MARGIN = 2, FUN = function(vec)
   sqrt(var(vec) / length(vec)))
df <- nrow(attitude) - 1
confint2.default(obj = est, se = se, df = df)
confint2.default(obj = est, se = se) # default is df = Inf and use of ctitical z-values
confint2.default(obj = est, se = se, df = df, level = 0.99)

# error
## Not run:
confint2.default(obj = c(10, 12), se = c(3, 4, 5))

## End(Not run)
```

---

corp *Bivariate Correlations with Significant Symbols*

Description

corp computes bivariate correlations and their associated p-values. The function is primarily for preparing a correlation table for publication: the correlations are appended by significant symbols (e.g., asterixis), corp is simply corr.test + add_sig_cor.

Usage

```
corp(
  data,
  vrb.nm,
  use = "pairwise.complete.obs",
  method = "pearson",
  digits = 3L,
  p.10 = "",
  p.05 = "*",
  p.01 = "**",
  p.001 = "***",
  lead.zero = FALSE,
  trail.zero = TRUE,
  plus = FALSE,
  diags = FALSE,
  lower = TRUE,
  upper = FALSE
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variable columns.
use	character vector of length 1 specifying how to handle missing data when computing the correlations. The options are 1) "pairwise.complete.obs", 2) "complete.obs", 3) "na.or.complete", 4) "all.obs", or 5) "everything". See details of cor.
method	character vector of length 1 specifying the type of correlations to be computed. The options are 1) "pearson", 2) "kendall", or 3) "spearman". See details of cor.
digits	integer vector of length 1 specifying the number of decimals to round to.
p.10	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .10 level.
p.05	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .05 level.
p.01	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .01 level.
p.001	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .001 level.
lead.zero	logical vector of length 1 specifying whether to retain a zero in front of the decimal place.
trail.zero	logical vector of length 1 specifying whether to retain zeros after the decimal place (due to rounding).
plus	logical vector of length 1 specifying whether to include a plus sign in front of positive correlations (minus signs are always in front of negative correlations).
diags	logical vector of length 1 specifying whether to retain the values in the diagonal of the correlation matrix. If TRUE, then the diagonal will be 1s with digits number of zeros after the decimal place (and no significant symbols). If FALSE, then the diagonal will be NA.
lower	logical vector of length 1 specifying whether to retain the lower triangle of the correlation matrix. If TRUE, then the lower triangle correlations and their significance symbols are retained. If FAlSE, then the lower triangle will all be NA.
upper	logical vector of length 1 specifying whether to retain the upper triangle of the correlation matrix. If TRUE, then the upper triangle correlations and their significance symbols are retained. If FAlSE, then the upper triangle will all be NA.

Value

data.frame with rownames and colnames equal to vrb.nm containing the bivariate correlations with significance symbols after the correlation value, specified by the arguments p.10, p.05, p.01, and p.001 arguments. The specific elements of the return object are determined by the other arguments.

See Also

add_sig_cor for adding significant symbols to a correlation matrix, add_sig for adding significant symbols to any (atomic) vector, matrix, or (3D+) array, cor for computing only the correlation coefficients themselves corr.test for a function providing confidence intervals as well

Examples

```
corp(data = mtcars, vrb.nm = c("mpg","cyl","disp","hp","drat")) # no quotes b/c a data.frame
corp(data = attitude, vrb.nm = colnames(attitude))
corp(data = attitude, vrb.nm = colnames(attitude), p.10 = "'") # advance & privileges
corp(data = airquality, vrb.nm = colnames(airquality), plus = TRUE)
```

---

corp_by *Bivariate Correlations with Significant Symbols by Group*

Description

corp_by computes a correlation data.frame for each group within numeric data. The correlation coefficients are appended by their significant symbols based on their associated p-values. If only the correlation coefficients are desired, use cor_by which returns a list of numeric matrices. corp_by is simply corp + by2.

Usage

```
corp_by(
  data,
  vrb.nm,
  grp.nm,
  use = "pairwise.complete.obs",
  method = "pearson",
  sep = ".",
  digits = 3L,
  p.10 = "",
  p.05 = "*",
  p.01 = "**",
  p.001 = "***",
  lead.zero = FALSE,
  trail.zero = TRUE,
  plus = FALSE,
  diags = FALSE,
  lower = TRUE,
  upper = FALSE
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.
grp.nm	character vector of colnames from data specifying the groups.
use	character vector of length 1 specifying how to handle missing data when computing the correlations. The options are 1) "pairwise.complete.obs", 2) "complete.obs", 3) "na.or.complete", 4) "all.obs", or 5) "everything". See details of cor.
method	character vector of length 1 specifying the type of correlations to be computed. The options are 1) "pearson", 2) "kendall", or 3) "spearman". See details of cor.
sep	character vector of length 1 specifying the string to combine the group values together with. sep is only used if there are multiple grouping variables (i.e., length(grp.nm) > 1).
digits	integer vector of length 1 specifying the number of decimals to round to.
p.10	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .10 level.
p.05	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .05 level.
p.01	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .01 level.
p.001	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .001 level.
lead.zero	logical vector of length 1 specifying whether to retain a zero in front of the decimal place.
trail.zero	logical vector of length 1 specifying whether to retain zeros after the decimal place (due to rounding).
plus	logical vector of length 1 specifying whether to include a plus sign in front of positive correlations (minus signs are always in front of negative correlations).
diags	logical vector of length 1 specifying whether to retain the values in the diagonal of the correlation matrix. If TRUE, then the diagonal will be 1s with digits number of zeros after the decimal place (and no significant symbols). If FALSE, then the diagonal will be NA.
lower	logical vector of length 1 specifying whether to retain the lower triangle of the correlation matrix. If TRUE, then the lower triangle correlations and their significance symbols are retained. If FAlSE, then the lower triangle will all be NA.
upper	logical vector of length 1 specifying whether to retain the upper triangle of the correlation matrix. If TRUE, then the upper triangle correlations and their significance symbols are retained. If FAlSE, then the upper triangle will all be NA.

Value

list of data.frames containing the correlation coefficients and their appended significance symbols based upon their associated p-values. The listnames are the unique combinations of the grouping variables, separated by "sep" if multiple grouping variables (i.e., `length(grp.nm) > 1`) are input: `unique(interaction(data[grp.nm], sep = sep))`. For each data.frame, the rownames and colnames = `vrb.nm`. The significance symbols are specified by the arguments `p.10`, `p.05`, `p.01`, and `p.001`, after the correlation value. The specific elements of the return object are determined by the other arguments.

See Also

corp cor_by cor

Examples

```
# one grouping variable
corp_by(airquality, vrb.nm = c("Ozone","Solar.R","Wind"), grp.nm = "Month")
corp_by(airquality, vrb.nm = c("Ozone","Solar.R","Wind"), grp.nm = "Month",
   use = "complete.obs", method = "spearman")

# two grouping variables
corp_by(mtcars, vrb.nm = c("mpg","disp","drat","wt"), grp.nm = c("vs","am"))
corp_by(mtcars, vrb.nm = c("mpg","disp","drat","wt"), grp.nm = c("vs","am"),
   use = "complete.obs", method = "spearman", sep = "_")
```

---

corp_miss *Point-biserial Correlations of Missingness With Significant Symbols*

Description

`corp_miss` computes (point-biserial) correlations between missingness on data columns and scores on other data columns. It also appends significance symbols at the end of the correlations.

Usage

```
corp_miss(
  data,
  x.nm,
  m.nm,
  ov = FALSE,
  use = "pairwise.complete.obs",
  method = "pearson",
  m.suffix = if (ov) "_ov" else "_na",
  digits = 3L,
  p.10 = "",
  p.05 = "*",
  p.01 = "**",
```

```
  p.001 = "***",
  lead.zero = FALSE,
  trail.zero = TRUE,
  plus = FALSE
)
```

Arguments

data	data.frame of data.
x.nm	character vector of colnames in data to be the predictors of missingness.
m.nm	character vector of colnames in data to specify missing data on.
ov	logical vector of length 1 specifying whether the correlations should be with "observedness" rather than missingness.
use	character vector of length 1 specifying how to deal with missing data in the predictor columns. See cor.
method	character vector of length 1 specifying what type of correlations to compute. See cor.
m.suffix	character vector of length 1 specifying a string to oppend to the end of the colnames to clarify whether they refer to missingness or "observedness". Default is "_na" if ov = FALSE and "_ov" if ov = TRUE.
digits	integer vector of length 1 specifying the number of decimals to round to.
p.10	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .10 level.
p.05	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .05 level.
p.01	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .01 level.
p.001	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .001 level.
lead.zero	logical vector of length 1 specifying whether to retain a zero in front of the decimal place.
trail.zero	logical vector of length 1 specifying whether to retain zeros after the decimal place (due to rounding).
plus	logical vector of length 1 specifying whether to include a plus sign in front of positive correlations (minus signs are always in front of negative correlations).

Details

cor_miss calls make.dumNA to create dummy vectors representing missingness on the data[m.nm] columns.

Value

numeric matrix of (point-biserial) correlations between rows of predictors and columns of missingness.

Examples

```
corp_miss(data = airquality, x.nm = c("Wind","Temp","Month","Day"),
   m.nm = c("Ozone","Solar.R"))
corp_miss(data = airquality, x.nm = c("Wind","Temp","Month","Day"),
   m.nm = c("Ozone","Solar.R"), ov = TRUE) # correlations with "observedness"
corp_miss(data = airquality, x.nm = c("Wind","Temp","Month","Day"),
   m.nm = c("Ozone","Solar.R"), use = "complete.obs", method = "kendall")
```

---

corp_ml — corp_ml *decomposes correlations from multilevel data into within-group and between-group correlations as well as adds significance symbols to the end of each value. The workhorse of the function is* statsBy. corp_ml *is simply a combination of* cor_ml *and* add_sig_cor.

Description

corp_ml decomposes correlations from multilevel data into within-group and between-group correlations as well as adds significance symbols to the end of each value. The workhorse of the function is statsBy. corp_ml is simply a combination of cor_ml and add_sig_cor.

Usage

```
corp_ml(
  data,
  vrb.nm,
  grp.nm,
  use = "pairwise.complete.obs",
  method = "pearson",
  digits = 3L,
  p.10 = "",
  p.05 = "*",
  p.01 = "**",
  p.001 = "***",
  lead.zero = FALSE,
  trail.zero = TRUE,
  plus = FALSE,
  diags = FALSE,
  lower = TRUE,
  upper = FALSE
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variable columns.

grp.nm	character vector of length 1 of a colname from data specifying the grouping column.
use	character vector of length 1 specifying how to handle missing values when computing the correlations. The options are: 1) "pairwise.complete.obs" which uses pairwise deletion, 2) "complete.obs" which uses listwise deletion, and 3) "everything" which uses all cases and returns NA for any correlations from columns in data[vrb.nm] with missing values.
method	character vector of length 1 specifying which type of correlations to compute. The options are: 1) "pearson" for traditional Pearson product-moment correlations, 2) "kendall" for Kendall rank correlations, and 3) "spearman" for Spearman rank correlations.
digits	integer vector of length 1 specifying the number of decimals to round to.
p.10	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .10 level.
p.05	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .05 level.
p.01	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .01 level.
p.001	character vector of length 1 specifying which symbol to append to the end of any correlation significant at the p < .001 level.
lead.zero	logical vector of length 1 specifying whether to retain a zero in front of the decimal place.
trail.zero	logical vector of length 1 specifying whether to retain zeros after the decimal place (due to rounding).
plus	logical vector of length 1 specifying whether to include a plus sign in front of positive correlations (minus signs are always in front of negative correlations).
diags	logical vector of length 1 specifying whether to retain the values in the diagonal of the correlation matrix. If TRUE, then the diagonal will be 1s with digits number of zeros after the decimal place (and no significant symbols). If FALSE, then the diagonal will be NA.
lower	logical vector of length 1 specifying whether to retain the lower triangle of the correlation matrix. If TRUE, then the lower triangle correlations and their significance symbols are retained. If FAlSE, then the lower triangle will all be NA.
upper	logical vector of length 1 specifying whether to retain the upper triangle of the correlation matrix. If TRUE, then the upper triangle correlations and their significance symbols are retained. If FAlSE, then the upper triangle will all be NA.

Value

list of two elements that are data.frames with names "within" and "between". The first data.frame has the within-group correlations with their significance symbols at the end of the statistically significant correlations based on their associated p-value. The second data.frame has the between-group correlations with their significance symbols at the end of the statistically significant correlations based on their associated p-values. The rownames and colnames of each dataframe are vrb.nm. The formatting of the two data.frames depends on several of the arguments.

See Also

cor_ml for multilevel correlations without significance symbols, corp_by for correlations with significance symbols by group, statsBy the workhorse for the corp_ml function, add_sig_cor for adding significant symbols to correlation matrices,

Examples

```
# traditional use
tmp <- c("outcome","case","session","trt_time") # roxygen2 does not like c() inside []
dat <- as.data.frame(lmeInfo::Bryant2016)[tmp]
stats_by <- psych::statsBy(dat, group = "case") # requires you to include "case" column in dat
corp_ml(data = dat, vrb.nm = c("outcome","session","trt_time"), grp.nm = "case")

# varying the `use` and `method` arguments
corp_ml(data = airquality, vrb.nm = c("Ozone","Solar.R","Wind","Temp"), grp.nm = "Month",
   use = "pairwise", method = "pearson")
corp_ml(data = airquality, vrb.nm = c("Ozone","Solar.R","Wind","Temp"), grp.nm = "Month",
   use = "complete", method = "kendall")
corp_ml(data = airquality, vrb.nm = c("Ozone","Solar.R","Wind","Temp"), grp.nm = "Month",
   use = "everything", method = "spearman")
```

---

cor_by *Correlation Matrix by Group*

Description

cor_by computes a correlation matrix for each group within numeric data. Only the correlation coefficients are determined and not any NHST information. If that is desired, use corp_by which includes significance symbols. cor_by is simply cor + by2.

Usage

```
cor_by(
  data,
  vrb.nm,
  grp.nm,
  use = "pairwise.complete.obs",
  method = "pearson",
  sep = ".",
  check = TRUE
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.

grp.nm	character vector of colnames from data specifying the groups.
use	character vector of length 1 specifying how to handle missing data when computing the correlations. The options are 1) "pairwise.complete.obs", 2) "complete.obs", 3) "na.or.complete", 4) "all.obs", or 5) "everything". See details of cor.
method	character vector of length 1 specifying the type of correlations to be computed. The options are 1) "pearson", 2) "kendall", or 3) "spearman". See details of cor.
sep	character vector of length 1 specifying the string to combine the group values together with. sep is only used if there are multiple grouping variables (i.e., length(grp.nm) > 1).
check	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether data[vrb.nm] are all mode numeric. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

list of numeric matrices containing the correlations from each group. The listnames are the unique combinations of the grouping variables, separated by "sep" if multiple grouping variables (i.e., length(grp.nm) > 1) are input: unique(interaction(data[grp.nm], sep = sep)). The rownames and colnames of each numeric matrix are vrb.nm.

See Also

cor for full sample correlation matrixes, corp for full sample correlation data.frames with significance symbols, corp_by for full sample correlation data.farmes with significance symbols by group.

Examples

```
# one grouping variable
cor_by(airquality, vrb.nm = c("Ozone","Solar.R","Wind"), grp.nm = "Month")
cor_by(airquality, vrb.nm = c("Ozone","Solar.R","Wind"), grp.nm = "Month",
   use = "complete.obs", method = "spearman")

# two grouping variables
cor_by(mtcars, vrb.nm = c("mpg","disp","drat","wt"), grp.nm = c("vs","am"))
cor_by(mtcars, vrb.nm = c("mpg","disp","drat","wt"), grp.nm = c("vs","am"),
   use = "complete.obs", method = "spearman", sep = "_")
```

---

cor_miss *Point-biserial Correlations of Missingness*

Description

cor_miss computes (point-biserial) correlations between missingness on data columns and scores on other data columns.

Usage

```
cor_miss(
  data,
  x.nm,
  m.nm,
  ov = FALSE,
  use = "pairwise.complete.obs",
  method = "pearson"
)
```

Arguments

data	data.frame of data.
x.nm	character vector of colnames in data to be the predictors of missingness.
m.nm	character vector of colnames in data to specify missing data on.
ov	logical vector of length 1 specifying whether the correlations should be with "observedness" rather than missingness.
use	character vector of length 1 specifying how to deal with missing data in the predictor columns. See cor.
method	character vector of length 1 specifying what type of correlations to compute. See cor.

Details

cor_miss calls make.dumNA to create dummy vectors representing missingness on the data[m.nm] columns.

Value

numeric matrix of (point-biserial) correlations between rows of predictors and columns of missingness.

Examples

```
cor_miss(data = airquality, x.nm = c("Wind","Temp","Month","Day"),
   m.nm = c("Ozone","Solar.R"))
cor_miss(data = airquality, x.nm = c("Wind","Temp","Month","Day"),
   m.nm = c("Ozone","Solar.R"), ov = TRUE) # correlations with "observedness"
cor_miss(data = airquality, x.nm = c("Wind","Temp","Month","Day"),
   m.nm = c("Ozone","Solar.R"), use = "complete.obs", method = "kendall")
```

cor_ml *Multilevel Correlation Matrices*

Description

cor_ml decomposes correlations from multilevel data into within-group and between-group correlations. The workhorse of the function is statsBy.

Usage

```
cor_ml(data, vrb.nm, grp.nm, use = "pairwise.complete.obs", method = "pearson")
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variable columns.
grp.nm	character vector of length 1 of a colname from data specifying the grouping column.
use	character vector of length 1 specifying how to handle missing values when computing the correlations. The options are: 1. "pairwise.complete.obs" which uses pairwise deletion, 2. "complete.obs" which uses listwise deletion, and 3. "everything" which uses all cases and returns NA for any correlations from columns in data[vrb.nm] with missing values.
method	character vector of length 1 specifying which type of correlations to compute. The options are: 1. "pearson" for traditional Pearson product-moment correlations, 2. "kendall" for Kendall rank correlations, and 3. "spearman" for Spearman rank correlations.

Value

list with two elements named "within" and "between" each containing a numeric matrix. The first "within" matrix is the within-group correlation matrix and the second "between" matrix is the between-group correlation matrix. The rownames and colnames of each numeric matrix are vrb.nm.

See Also

corp_ml for multilevel correlations with significance symbols, cor_by for correlation matrices by group, cor for traditional, single-level correlation matrices, statsBy the workhorse for the cor_ml function,

Examples

```
# traditional use
tmp <- c("outcome","case","session","trt_time") # roxygen2 does not like c() inside []
dat <- as.data.frame(lmeInfo::Bryant2016)[tmp]
stats_by <- psych::statsBy(dat, group = "case") # requires you to include "case" column in dat
```

```
cor_ml(data = dat, vrb.nm = c("outcome","session","trt_time"), grp.nm = "case")

# varying the \code{use} and \code{method} arguments
cor_ml(data = airquality, vrb.nm = c("Ozone","Solar.R","Wind","Temp"), grp.nm = "Month",
   use = "pairwise", method = "pearson")
cor_ml(data = airquality, vrb.nm = c("Ozone","Solar.R","Wind","Temp"), grp.nm = "Month",
   use = "complete", method = "kendall")
cor_ml(data = airquality, vrb.nm = c("Ozone","Solar.R","Wind","Temp"), grp.nm = "Month",
   use = "everything", method = "spearman")
```

`covs_test` *Covariances Test of Significance*

Description

`covs_test` computes sample covariances and tests for their significance with the Pearson method assuming multivariate normality of the data. Note, the normal-theory significance test for the covariance is much more sensitive to departures from normality than the significant test for the mean. This function is the covariance analogue to the `psych::corr.test()` function for correlations.

Usage

```
covs_test(data, vrb.nm, use = "pairwise", ci.level = 0.95, rtn.dfm = FALSE)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames specifying the variables in `data` to conduct the significant test of the covariances.
`use`	character vector of length 1 specifying how missing values are handled. Currently, there are only two options: 1) "pairwise" for pairwise deletion (i.e., `cov(use = "pairwise.complete.obs")`), or 2) "complete" for listwise deletion (i.e., `cov(use = "complete.obs")`).
`ci.level`	numeric vector of length 1 specifying the confidence level. It must be between 0 and 1 - or it can be NULL in which case confidence intervals are not computed and the return object does not have "lwr" or "upr" columns.
`rtn.dfm`	logical vector of length 1 specifying whether the return object should be an array (FALSE) or data.frame (TRUE). If an array, then the first two dimensions are the matrix dimensions from the covariance matrix and the 3rd dimension (aka layers) contains the statistical information (e.g., est, se, t). If data.frame, then the first two columns are the matrix dimensions from the covariance matrix expanded and the rest of the columns contain the statistical information (e.g., est, se, t).

Value

If `rtn.dfm = FALSE`, an array where its first two dimensions are the matrix dimensions from the covariance matrix and the 3rd dimension (aka layers) contains the statistical information detailed below. If `rtn.dfm = TRUE`, a data.frame where its first two columns are the expanded matrix dimensions from the covariance matrix and the rest of the columns contain the statistical information detailed below:

cov sample covariances

se standard errors of the covariances

t t-values

df degrees of freedom (n - 2)

p two-sided p-values

lwr lower bound of the confidence intervals (excluded if `ci.level = NULL`)

upr upper bound of the confidence intervals (excluded if `ci.level = NULL`)

See Also

`cov` for covariance matrix estimates, `corr.test` for correlation matrix significant testing,

Examples

```
# traditional use
covs_test(data = attitude, vrb.nm = names(attitude))
covs_test(data = attitude, vrb.nm = names(attitude),
   ci.level = NULL) # no confidence intervals
covs_test(data = attitude, vrb.nm = names(attitude),
   rtn.dfm = TRUE) # return object as data.frame

# NOT same as simple linear regression slope
covTest <- covs_test(data = attitude, vrb.nm = names(attitude),
   ci.level = NULL, rtn.dfm = TRUE)
x <- covTest[with(covTest, rownames == "rating" & colnames == "complaints"), ]
lm_obj <- lm(rating ~ complaints, data = attitude)
y <- coef(summary(lm_obj))["complaints", , drop = FALSE]
print(x); print(y)
z <- x[, "cov"] / var(attitude$"complaints")
print(z) # dividing by variance of the predictor gives you the regression slope
# but the t-values and p-values are still different

# NOT same as correlation coefficient
covTest <- covs_test(data = attitude, vrb.nm = names(attitude),
   ci.level = NULL, rtn.dfm = TRUE)
x <- covTest[with(covTest, rownames == "rating" & colnames == "complaints"), ]
cor_test <- cor.test(x = attitude[[1]], y = attitude[[2]])
print(x); print(cor_test)
z <- x[, "cov"] / sqrt(var(attitude$"rating") * var(attitude$"complaints"))
print(z) # dividing by sqrt of the variances gives you the correlation
# but the t-values and p-values are still different
```

cronbach *Cronbach's Alpha of a Set of Variables/Items*

Description

cronbach computes Cronbach's alpha for a set of variables/items as an estimate of reliability for a score. There are three different options for confidence intervals. Missing data can be handled by either pairwise deletion (use = "pairwise.complete.obs") or listwise deletion (use = "complete.obs"). cronbach is a wrapper for the alpha function in the psych package.

Usage

```
cronbach(
  data,
  vrb.nm,
  ci.type = "delta",
  level = 0.95,
  use = "pairwise.complete.obs",
  stats = c("average_r", "nvrb"),
  R = 200L,
  boot.ci.type = "perc"
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames of data specifying the variables/items.
ci.type	character vector of length 1 specifying the type of confidence interval to compute. The options are 1) "classic" is the Feldt et al. (1987) procedure using only the mean covariance, 2) "delta" is the Duhhacheck & Iacobucci (2004) procedure using the delta method of the covariance matrix, or 3) "boot" is bootstrapped confidence intervals with the method specified by boot.ci.type.
level	double vector of length 1 with a value between 0 and 1 specifying what confidence level to use.
use	character vector of length 1 specifying how to handle missing data when computing the covariances. The options are 1) "pairwise.complete.obs", 2) "complete.obs", 3) "na.or.complete", 4) "all.obs", or 5) "everything". See details of cov.
stats	character vector specifying the additional statistical information you could like related to cronbach's alpha. Options are: 1) "std.alpha" = cronbach's alpha of the standardized variables/items, 2) "G6(smc)" = Guttman's Lambda 6 reliability, 3) "average_r" = mean correlation between the variables/items, 4) "median_r" = median correlation between the variables/items, 5) "mean" = mean of the the score from averaging the variables/items together, 6) "sd" = standard deviation of the scores from averaging the variables/items together, 7) "nvrb" = number of variables/items. The default is "average_r" and "nvrb".

R — integer vector of length 1 specifying the number of bootstrapped resamples to do. Only used when ci.type = "boot".

boot.ci.type — character vector of length 1 specifying the type of bootstrapped confidence interval to compute. The options are 1) "perc" for the regular percentile method, 2) "bca" for bias-corrected and accelerated percentile method, 3) "norm" for the normal method that uses the bootstrapped standard error to construct symmetrical confidence intervals with the classic formula around the bias-corrected estimate, and 4) "basic" for the basic method. Note, "stud" for the studentized method is NOT an option. See boot.ci as well as confint2.boot for details.

Details

When ci.type = "classic" the confidence interval is based on the mean covariance. It is the same as the confidence interval used by alpha.ci (Feldt, Woodruff, & Salih, 1987). When ci.type = "delta" the confidence interval is based on the delta method of the covariance matrix. It is based on the standard error returned by alpha (Duhachek & Iacobucci, 2004).

Value

double vector containing Cronbach's alpha, it's standard error, and it's confidence interval, followed by any statistics requested via the stats argument.

References

Feldt, L. S., Woodruff, D. J., & Salih, F. A. (1987). Statistical inference for coefficient alpha. Applied Psychological Measurement (11) 93-103.

Duhachek, A. and Iacobucci, D. (2004). Alpha's standard error (ase): An accurate and precise confidence interval estimate. Journal of Applied Psychology, 89(5):792-808.

See Also

cronbachs composite

Examples

```
tmp_nm <- c("A2","A3","A4","A5")
psych::alpha(psych::bfi[tmp_nm])[["total"]]
a <- suppressMessages(psych::alpha(attitude))[["total"]]["raw_alpha"]
a.ci <- psych::alpha.ci(a, n.obs = 30,
   n.var = 7, digits = 7) # n.var is optional and only needed to find r.bar
cronbach(data = psych::bfi, vrb.nm = c("A2","A3","A4","A5"), ci.type = "classic")
cronbach(data = psych::bfi, vrb.nm = c("A2","A3","A4","A5"), ci.type = "delta")
cronbach(data = psych::bfi, vrb.nm = c("A2","A3","A4","A5"), ci.type = "boot")
cronbach(data = psych::bfi, vrb.nm = c("A2","A3","A4","A5"), stats = NULL)

## Not run:
cronbach(data = psych::bfi, vrb.nm = c("A2","A3","A4","A5"), ci.type = "boot",
   boot.ci.type = "bca") # will automatically convert to "perc" when "bca" fails

## End(Not run)
```

cronbachs *Cronbach's Alpha for Multiple Sets of Variables/Items*

Description

cronbachs computes Cronbach's alpha for multiple sets of variables/items as an estimate of reliability for multiple scores. There are three different options for confidence intervals. Missing data can be handled by either pairwise deletion (use = "pairwise.complete.obs") or listwise deletion (use = "complete.obs"). cronbachs is a wrapper for the alpha function in the psych package.

Usage

```
cronbachs(
  data,
  vrb.nm.list,
  ci.type = "delta",
  level = 0.95,
  use = "pairwise.complete.obs",
  stats = c("average_r", "nvrb"),
  R = 200L,
  boot.ci.type = "perc"
)
```

Arguments

data	data.frame of data.
vrb.nm.list	list of character vectors specifying the sets of variables/items. Each element of vrb.nm.list provides the colnames of data for that set of variables/items.
ci.type	character vector of length 1 specifying the type of confidence interval to compute. The options are 1) "classic" = the Feldt et al. (1987) procedure using only the mean covariance, 2) "delta" = the Duhhacheck & Iacobucci (2004) procedure using the delta method of the covariance matrix, or 3) "boot" = bootstrapped confidence intervals with the method specified by boot.ci.type.
level	double vector of length 1 with a value between 0 and 1 specifying what confidence level to use.
use	character vector of length 1 specifying how to handle missing data when computing the covariances. The options are 1) "pairwise.complete.obs", 2) "complete.obs", 3) "na.or.complete", 4) "all.obs", or 5) "everything". See details of cov.
stats	character vector specifying the additional statistical information you could like related to cronbach's alpha. Options are: 1) "std.alpha" = cronbach's alpha of the standardized variables/items, 2) "G6(smc)" = Guttman's Lambda 6 reliability, 3) "average_r" = mean correlation between the variables/items, 4) "median_r" = median correlation between the variables/items, 5) "mean" = mean of the the scores from averaging the variables/items together, 6) "sd" = standard deviation

of the scores from averaging the variables/items together, 7) "nvrb" = number of variables/items. The default is "average_r" and "nvrb".

`R` integer vector of length 1 specifying the number of bootstrapped resamples to do. Only used when `ci.type` = "boot".

`boot.ci.type` character vector of length 1 specifying the type of bootstrapped confidence interval to compute. The options are 1) "perc" for the regular percentile method, 2) "bca" for bias-corrected and accelerated percentile method, 3) "norm" for the normal method that uses the bootstrapped standard error to construct symmetrical confidence intervals with the classic formula around the bias-corrected estimate, and 4) "basic" for the basic method. Note, "stud" for the studentized method is NOT an option. See `boot.ci` as well as `confint2.boot` for details.

Details

When `ci.type` = "classic" the confidence interval is based on the mean covariance. It is the same as the confidence interval used by `alpha.ci` (Feldt, Woodruff, & Salih, 1987). When `ci.type` = "delta" the confidence interval is based on the delta method of the covariance matrix. It is based on the standard error returned by `alpha` (Duhachek & Iacobucci, 2004).

Value

data.frame containing the following columns:

est Cronbach's alpha itself

se standard error for Cronbach's alpha

lwr lower bound of the confidence interval of Cronbach's alpha

upr upper bound for the confidence interval of Cronbach's alpha,

??? any statistics requested via the `stats` argument

References

Feldt, L. S., Woodruff, D. J., & Salih, F. A. (1987). Statistical inference for coefficient alpha. Applied Psychological Measurement (11) 93-103.

Duhachek, A. and Iacobucci, D. (2004). Alpha's standard error (ase): An accurate and precise confidence interval estimate. Journal of Applied Psychology, 89(5):792-808.

See Also

`cronbach` `composites`

Examples

```
dat0 <- psych::bfi
dat1 <- str2str::pick(x = dat0, val = c("A1","C4","C5","E1","E2","O2","O5",
   "gender","education","age"), not = TRUE, nm = TRUE)
vrb_nm_list <- lapply(X = str2str::sn(c("E","N","C","A","O")), FUN = function(nm) {
   str2str::pick(x = names(dat1), val = nm, pat = TRUE)})
cronbachs(data = dat1, vrb.nm.list = vrb_nm_list, ci.type = "classic")
```

```
cronbachs(data = dat1, vrb.nm.list = vrb_nm_list, ci.type = "delta")
cronbachs(data = dat1, vrb.nm.list = vrb_nm_list, ci.type = "boot")
suppressMessages(cronbachs(data = attitude, vrb.nm.list =
   list(names(attitude)))) # also works with only one set of variables/items
```

`decompose` *Decompose a Numeric Vector by Group*

Description

`decompose` decomposes a numeric vector into within-group and between-group components via within-group centering and group-mean aggregation. There is an option to create a grand-mean centered version of the between-person component as well as lead/lag versions of the original vector and the within-group component.

Usage

```
decompose(x, grp, grand = TRUE, n.shift = NULL, undefined = NA)
```

Arguments

`x`	numeric vector.
`grp`	list of atomic vector(s) and/or factor(s) (e.g., data.frame), which each have same length as `x`. It can also be an atomic vector or factor, which will then be made the first element of a list internally.
`grand`	logical vector of length 1 specifying whether a grand-mean centered version of the the between-group component should be computed.
`n.shift`	integer vector specifying the direction and magnitude of the shifts. For example a one-lead is +1 and a two-lag is -2. See `shift` details.
`undefined`	atomic vector with length 1 (probably makes sense to be the same typeof as `x`). Specifies what to insert for undefined values after the shifting takes place. See `shift` details.

Value

data.frame with nrow = `length(x)` and `row.names` = `names(x)`. The first two columns correspond to the within-group component (i.e., "wth") and the between-group component (i.e., "btw"). If grand = TRUE, then the third column corresponds to the grand-mean centered between-group component (i.e., "btw_c"). If shift != NULL, then the last columns are the shifts indicated by n.shift, where the shifts of `x` are first (i.e., "tot") and then the shifts of the within-group component are second (i.e., "wth"). The naming of the shifted columns is based on the default behavior of `Shift_by`. See the details of `Shift_by`. If you don't like the default naming, then call `Decompose` instead and use the different suffix arguments.

See Also

`decomposes` `center_by` `agg` `shift_by`

Examples

```
# single grouping variable
chick_data <- as.data.frame(ChickWeight) # because the "groupedData" class
   # calls `[.groupedData`, which is different than `[.data.frame`
decompose(x = ChickWeight[["weight"]], grp = ChickWeight[["Chick"]])
decompose(x = ChickWeight[["weight"]], grp = ChickWeight[["Chick"]],
   grand = FALSE) # no grand-mean centering
decompose(x = setNames(obj = ChickWeight[["weight"]],
  nm = paste0(row.names(ChickWeight),"_row")), grp = ChickWeight[["Chick"]]) # with names

# multiple grouping variables
tmp_nm <- c("Type","Treatment") # b/c Roxygen2 doesn't like c() in a []
decompose(x = as.data.frame(CO2)[["uptake"]], grp = as.data.frame(CO2)[tmp_nm])
decompose(x = as.data.frame(CO2)[["uptake"]], grp = as.data.frame(CO2)[tmp_nm],
   n.shift = 1)
decompose(x = as.data.frame(CO2)[["uptake"]], grp = as.data.frame(CO2)[tmp_nm],
   n.shift = c(+2, +1, -1, -2))
```

---

decomposes *Decompose Numeric Data by Group*

Description

decomposes decomposes numeric data by group into within-group and between- group components via within-group centering and group-mean aggregation. There is an option to create a grand-mean centered version of the between-group components.

Usage

```
decomposes(
  data,
  vrb.nm,
  grp.nm,
  grand = TRUE,
  n.shift = NULL,
  undefined = NA,
  suffix.wth = "_w",
  suffix.btw = "_b",
  suffix.grand = "c",
  suffix.lead = "_dw",
  suffix.lag = "_gw"
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.

grp.nm	character vector of colnames from data specifying the groups.
grand	logical vector of length 1 specifying whether grand-mean centered versions of the the between-group components should be computed.
n.shift	integer vector specifying the direction and magnitude of the shifts. For example a one-lead is +1 and a two-lag is -2. See Shift_by details.
undefined	atomic vector of length 1 (probably makes sense to be the same typeof as the vectors in data[vrb.nm]). Specifies what to insert for undefined values after the shifting takes place. See details of Shift_by.
suffix.wth	character vector with a single element specifying the string to append to the end of the within-group component colnames of the return object.
suffix.btw	character vector with a single element specifying the string to append to the end of the between-group component colnames of the return object.
suffix.grand	character vector with a single element specifying the string to append to the end of the grand-mean centered version of the between-group component colnames of the return object. Note, this is a string that is appended after suffix.btw has already been appended.
suffix.lead	character vector with a single element specifying the string to append to the end of the positive shift colnames of the return object. Note, decomposes will add abs(n.shift) to the end of suffix.lead.
suffix.lag	character vector with a single element specifying the string to append to the end of the negative shift colnames of the return object. Note, decomposes will add abs(n.shift) to the end of suffix.lag.

Value

data.frame with nrow = nrow(data and rownames = row.names(data). The first set of columns correspond to the within-group components, followed by the between-group components. If grand = TRUE, then the next set of columns correspond to the grand-mean centered between-group components. If shift != NULL, then the last columns are the shifts by group indicated by n.shift, where the shifts of data[vrb.nm] are first and then the shifts of the within-group components are second.

See Also

decompose centers_by aggs shifts_by

Examples

```
ChickWeight2 <- as.data.frame(ChickWeight)
row.names(ChickWeight2) <- as.numeric(row.names(ChickWeight)) / 1000
decomposes(data = ChickWeight2, vrb.nm = c("weight","Time"), grp.nm = "Chick")
decomposes(data = ChickWeight2, vrb.nm = c("weight","Time"), grp.nm = "Chick",
   suffix.wth = ".wth", suffix.btw = ".btw", suffix.grand = ".grand")
decomposes(data = as.data.frame(CO2), vrb.nm = c("conc","uptake"),
   grp.nm = c("Type","Treatment")) # multiple grouping columns
decomposes(data = as.data.frame(CO2), vrb.nm = c("conc","uptake"),
   grp.nm = c("Type","Treatment"), n.shift = 1) # with lead
decomposes(data = as.data.frame(CO2), vrb.nm = c("conc","uptake"), grp.nm = c("Type","Treatment"),
   n.shift = c(+2, +1, -1, -2)) # with multiple lead/lags
```

deff *Design Effect from Multilevel Numeric Vector*

Description

deff computes the design effect for a multilevel numeric vector. Design effects summarize how much larger sampling variances (i.e., squared standard errors) are due to the multilevel structure of the data. By taking the square root, the value summarizes how much larger standard errors are due to the multilevel structure of the data.

Usage

```
deff(x, grp, how = "lme", REML = TRUE)
```

Arguments

x	numeric vector.
grp	atomic vector the same length as x providing the grouping variable.
how	character vector of length 1 specifying how the ICC(1,1) should be calculated. There are four options: 1) "lme" uses a linear mixed effects model with the function lme from the package nlme, 2) "lmer" uses a linear mixed effects modeling with the function lmer from the package lme4, 3) "aov" uses a one-way analysis of variance with the function aov, and 4) "raw" uses the observed variances, which provides a biased estimate of the ICC(1,1) and is not recommended (It is only included for teaching purposes).
REML	logical vector of length 1 specifying whether restricted maximum likelihood estimation (TRUE) should be used rather than traditional maximum likelihood estimation (FALSE). Only used for linear mixed effects models if how = "lme" or how = "lmer".

Details

Design effects are a function of both the intraclass correlation (ICC) and the average group size. Design effects can be large due to large ICCs and small group sizes or small ICCs and large group sizes. For example, with an ICC = .01 and average group size of 100, the design effect would be 2.0, whose square root is 1.41. For more information, see myths 1 and 2 in Huang (2018).

Value

double vector of lenght 1 providing the design effect.

References

Huang, F. L. (2018). Multilevel modeling myths School Psychology Quarterly, 33(3), 492-499.

See Also

icc_11 deffs

Examples

```
icc_11(x = airquality$"Ozone", grp = airquality$"Month")
length_by(x = airquality$"Ozone", grp = airquality$"Month", na.rm = TRUE)
deff(x = airquality$"Ozone", grp = airquality$"Month")
sqrt(deff(x = airquality$"Ozone", grp = airquality$"Month")) # how much SE inflated
```

---

deffs *Design Effects from Multilevel Numeric Data*

Description

deffs computes the design effects for multilevel numeric data. Design effects summarize how much larger sampling variances (i.e., squared standard errors) are due to the multilevel structure of the data. By taking the square root, the value summarizes how much larger standard errors are due to the multilevel structure of the data.

Usage

```
deffs(data, vrb.nm, grp.nm, how = "lme", REML = FALSE)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variable columns.
grp.nm	character vector of length 1 of a colname from data specifying the grouping column.
how	character vector of length 1 specifying how the ICC(1,1) should be calculated. There are four options: 1) "lme" uses a linear mixed effects model with the function lme from the package nlme, 2) "lmer" uses a linear mixed effects modeling with the function lmer from the package lme4, 3) "aov" uses a one-way analysis of variance with the function aov, and 4) "raw" uses the observed variances, which provides a biased estimate of the ICC(1,1) and is not recommended (It is only included for teaching purposes).
REML	logical vector of length 1 specifying whether restricted maximum likelihood estimation (TRUE) should be used rather than traditional maximum likelihood estimation (FALSE). Only used for linear mixed effects models if how = "lme" or how = "lmer".

Details

Design effects are a function of both the intraclass correlation (ICC) and the average group size. Design effects can be large due to large ICCs and small group sizes or small ICCs and large group sizes. For example, with an ICC = .01 and average group size of 100, the design effect would be 2.0, whose square root is 1.41. For more information, see myths 1 and 2 in Huang (2018).

Value

double vector providing the design effects with names = `vrb.nm`.

References

Huang, F. L. (2018). Multilevel modeling myths School Psychology Quarterly, 33(3), 492-499.

See Also

`iccs_11` `deff`

Examples

```
iccs_11(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month")
lengths_by(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month", na.rm = TRUE)
deffs(data = airquality, vrb.nm = c("Ozone","Solar.R"), grp.nm = "Month")
```

---

describe_ml *Multilevel Descriptive Statistics*

Description

`describe_ml` decomposes descriptive statistics from multilevel data into within-group and between-group descriptives. The data is first separated out into within-group components via `centers_by` and between-group components via `aggs`. Then the `psych` function `describe` is applied to both.

Usage

```
describe_ml(
  data,
  vrb.nm,
  grp.nm,
  na.rm = TRUE,
  interp = FALSE,
  skew = TRUE,
  ranges = TRUE,
  trim = 0.1,
  type = 3,
  quant = NULL,
  IQR = FALSE
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variable columns.
grp.nm	character vector of length 1 of a colname from data specifying the grouping column.
na.rm	logical vector of length 1 specifying whether missing values should be removed before calculating the descriptive statistics. See psych::describe.
interp	logical vector of length 1 specifying whether the median should be standard (FALSE) or interpolated (TRUE).
skew	logical vector of length 1 specifying whether skewness and kurtosis should be calculated (TRUE) or not (FALSE).
ranges	logical vector of length 1 specifying whether the minimum, maximum, and range (i.e., maximum - minimum) should be calculated (TRUE) or not (FALSE). Note, if ranges = FALSE, the trimmed mean and median absolute deviation is also not computed as per the psych::describe function behavior.
trim	numeric vector of length 1 specifying the top and bottom quantiles of data that are to be excluded when calculating the trimmed mean. For example, the default value of 0.1 means that only data within the 10th - 90th quantiles are used for calculating the trimmed mean.
type	numeric vector of length 1 specifying the type of skewness and kurtosis coefficients to compute. See the details of psych::describe. The options are 1, 2, or 3.
quant	numeric vector specifying the quantiles to compute. Foe example, the default value of c(0.25, 0.75) computes the 25th and 75th quantiles of the group number of cases. If quant = NULL, then no quantiles are returned.
IQR	logical vector of length 1 specifying whether to compute the Interquartile Range (TRUE) or not (FALSE), which is simply the 75th quantil - 25th quantile.

Value

list of two elements each containing a data.frame of descriptive statistics, the first for the within-person components ("within") and the second for the between-person components ("between").

See Also

describe

Examples

```
tmp_nm <- c("outcome","case","session","trt_time")
dat <- as.data.frame(lmeInfo::Bryant2016)[tmp_nm]
stats_by <- psych::statsBy(dat, group = "case") # requires you to include "case" column in dat
describe_ml(data = dat, vrb.nm = c("outcome","session","trt_time"), grp.nm = "case")
```

dum2nom *Dummy Variables to a Nominal Variable*

Description

dum2nom converts dummy variables to a nominal variable. The information from the dummy columns in a data.frame are combined into a character vector (or factor if rtn.fct = TRUE) representing a nominal variable. The unique values of the nominal variable will be the dummy colnames (i.e., dum.nm). Note, *all* the dummy variables associated with a nominal variable are required for this function to work properly. In regression-like models, data analysts will exclude one dummy variable for the category that is the reference group. If d = number of categories in the nominal variable, then that leads to d - 1 dummy variables in the model. dum2nom requires all d dummy variables.

Usage

```
dum2nom(data, dum.nm, yes = 1L, rtn.fct = FALSE)
```

Arguments

data	data.frame of data.
dum.nm	character vector of colnames from data specifying the dummy variables.
yes	atomic vector of length 1 specifying the unique value of the category in each dummy column. This must be the same value for all the dummy variables.
rtn.fct	logical vector of length 1 specifying whether the return object should be a factor (TRUE) or a character vector (FALSE).

Details

dum2nom tests to ensure that data[dum.nm] are indeed a set of dummy columns. First, the dummy columns are expected to have the same mode such that there is one yes unique value across the dummy columns. Second, each row in data[dum.nm] is expected to have either 0 or 1 instance of yes. If there is more than one instance of yes in a row, then an error is returned. If there is 0 instances of yes in a row (e.g., all missing values), NA is returned for that row. Note, any value other than yes will be treated as a no.

Value

character vector (or factor if rtn.fct = TRUE) containing the unique values of dum.nm - one for each dummy variable.

See Also

nom2dum

Examples

```
dum <- data.frame(
   "Quebec_nonchilled" = ifelse(CO2$"Type" == "Quebec" & CO2$"Treatment" == "nonchilled",
      yes = 1L, no = 0L),
   "Quebec_chilled" = ifelse(CO2$"Type" == "Quebec" & CO2$"Treatment" == "chilled",
      yes = 1L, no = 0L),
  "Mississippi_nonchilled" = ifelse(CO2$"Type" == "Mississippi" & CO2$"Treatment" == "nonchilled",
      yes = 1L, no = 0L),
  "Mississippi_chilled" = ifelse(CO2$"Type" == "Mississippi" & CO2$"Treatment" == "chilled",
      yes = 1L, no = 0L)
)
dum2nom(data = dum, dum.nm = names(dum)) # default
dum2nom(data = dum, dum.nm = names(dum), rtn.fct = TRUE) # return as a factor
## Not run:
dum2nom(data = npk, dum.nm = c("N","P","K")) # error due to overlapping dummy columns
dum2nom(data = mtcars, dum.nm = c("vs","am"))# error due to overlapping dummy columns

## End(Not run)
```

---

freq *Univariate Frequency Table*

Description

`freq` creates univariate frequency tables similar to `table`. It differs from `table` by allowing for custom sorting by something other than the alphanumerics of the unique values as well as returning an atomic vector rather than a 1D-array.

Usage

```
freq(
  x,
  exclude = if (useNA == "no") c(NA, NaN),
  useNA = "always",
  prop = FALSE,
  sort = "frequency",
  decreasing = TRUE,
  na.last = TRUE
)
```

Arguments

`x`	atomic vector or list vector. If not a vector, it will be coerced to a vector via `as.vector`.
`exclude`	unique values of `x` to exclude from the returned table. If NULL, then missing values are always included in the returned table. See `table` for documentation on the same argument.

useNA	character vector of length 1 specifying how to handle missing values (i.e., whether to include NA as an element in the returned table). There are three options: 1) "no" = don't include missing values in the table, 2) "ifany" = include missing values if there are any, 3) "always" = include missing values in the table, regardless of whether there are any or not. See table for documentation on the same argument.
prop	logical vector of length 1 specifying whether the returned table should include counts (FALSE) or proportions (TRUE). If NAs are excluded (e.g., useNA = "no" or exclude = c(NA, NaN)), then the proportions will be based on the number of observed elements.
sort	character vector of length 1 specifying how the returned table will be sorted. There are three options: 1) "frequency" = the frequency of the unique values in x, 2) "position" = the position when each unique value first appears in x, 3) "alphanum" = alphanumeric ordering of the unique values in x (the sorting used by table). When "frequency" is specified and there are ties, then the ties are sorted alphanumerically.
decreasing	logical vector of length 1 specifying whether the table should be sorted in decreasing (TRUE) or increasing (FALSE) order.
na.last	logical vector of length 1 specifying whether the table should have NAs last or in whatever position they end up at. This argument is only relevant if NAs exist in x and are included in the table (e.g., useNA = "always" or exclude = NULL).

Details

The name for the table element giving the frequency of missing values is "(NA)". This is different from table where the name is NA_character_. This change allows for the sorting of tables that include missing values, as subsetting in R is not possible with NA_character_ names. In future versions of the package, this might change as it should be possible to avoid this issue by subetting with a logical vector or integer indices instead of names. However, it is convenient to be able to subset the return object fully by names.

Value

numeric vector of frequencies as either counts (if prop = FALSE) or proportions (if prop = TRUE) with the unique values of x as names (missing values have name = "(NA)"). Note, this is different from table, which returns a 1D-array and has class "table".

See Also

freqs freq_by freqs_by table

Examples

```
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = FALSE,
   sort = "frequency", decreasing = TRUE, na.last = TRUE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = FALSE,
   sort = "frequency", decreasing = TRUE, na.last = FALSE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = TRUE,
   sort = "frequency", decreasing = FALSE, na.last = TRUE)
```

```
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = TRUE,
   sort = "frequency", decreasing = FALSE, na.last = FALSE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = FALSE,
   sort = "position", decreasing = TRUE, na.last = TRUE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = FALSE,
   sort = "position", decreasing = TRUE, na.last = FALSE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = TRUE,
   sort = "position", decreasing = FALSE, na.last = TRUE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = TRUE,
   sort = "position", decreasing = FALSE, na.last = FALSE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = FALSE,
   sort = "alphanum", decreasing = TRUE, na.last = TRUE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = FALSE,
   sort = "alphanum", decreasing = TRUE, na.last = FALSE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = TRUE,
   sort = "alphanum", decreasing = FALSE, na.last = TRUE)
freq(c(mtcars$"carb", NA, NA, mtcars$"gear"), prop = TRUE,
   sort = "alphanum", decreasing = FALSE, na.last = FALSE)
```

`freqs` *Multiple Univariate Frequency Tables*

Description

`freqs` creates a frequency table for a set of variables in a data.frame. Depending on `total`, frequencies for all the variables together can be returned. The function probably makes the most sense for sets of variables with similar unique values (e.g., items from a questionnaire with similar response options).

Usage

```
freqs(data, vrb.nm, prop = FALSE, useNA = "always", total = "no")
```

Arguments

`data`	data.fame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`prop`	logical vector of length 1 specifying whether the frequencies should be counts (FALSE) or proportions (TRUE). Note, whether the proportions include missing values depends on the `useNA` argument.
`useNA`	character vector of length 1 specifying how missing values should be handled. The three options are 1) "no" = do not include NA frequencies in the return object, 2) "ifany" = only NA frequencies if there are any missing values (in any variable from `data[vrb.nm]`), or 3) "always" = do include NA frequencies regardless of whether there are missing values or not.

`total` character vector of length 1 specifying whether the frequencies for the set of variables as a whole should be returned. The name "total" refers to tabulating the frequencies for the variables from `data[vrb.nm]` together as a set. The three options are 1) "no" = do not include a row for the total frequencies in the return object, 2) "yes" = do include the total frequencies as the first row in the return object, or 3) "only" = only include the total frequencies as a single row in the return object and do not include rows for each of the individual column frequencies in `data[vrb.nm]`.

Details

`freqs` uses `plyr::rbind.fill` to combine the results from `table` applied to each variable into a single data.frame. If a variable from `data[vrb.nm]` does not have values present in other variables from `data[vrb.nm]`, then the frequencies in the return object will be 0.

The name for the table element giving the frequency of missing values is "(NA)". This is different from `table` where the name is `NA_character_`. This change allows for the sorting of tables that include missing values, as subsetting in R is not possible with `NA_character_` names. In future versions of the package, this might change as it should be possible to avoid this issue by subetting with a logical vector or integer indices instead of names. However, it is convenient to be able to subset the return object fully by names.

Value

data.frame of frequencies for the variables in `data[vrb.nm]`. Depending on `prop`, the frequencies are either counts (FALSE) or proportions (TRUE). Depending on `total`, the nrow is either 1) `length(vrb.nm)` (if `total` = "no"), 1 + `length(vrb.nm)` (if `total` = "yes"), or 3) 1 (if `total` = "only"). The rownames are `vrb.nm` for each variable in `data[vrb.nm]` and "_total_" for the total row (if present). The colnames are the unique values present in `data[vrb.nm]`, potentially including "(NA)" depending on `useNA`.

See Also

`freq` `freqs_by` `freq_by` `table`

Examples

```
vrb_nm <- str2str::inbtw(names(psych::bfi), "A1","O5")
freqs(data = psych::bfi, vrb.nm = vrb_nm) # default
freqs(data = psych::bfi, vrb.nm = vrb_nm, prop = TRUE) # proportions by row
freqs(data = psych::bfi, vrb.nm = vrb_nm, useNA = "no") # without NA counts
freqs(data = psych::bfi, vrb.nm = vrb_nm, total = "yes") # include total counts
```

`freqs_by` *Multiple Univariate Frequency Tables*

Description

freqs_by creates a frequency table for a set of variables in a data.frame by group. Depending on total, frequencies for all the variables together can be returned by group. The function probably makes the most sense for sets of variables with similar unique values (e.g., items from a questionnaire with similar response options).

Usage

```
freqs_by(
  data,
  vrb.nm,
  grp.nm,
  prop = FALSE,
  useNA = "always",
  total = "no",
  sep = "."
)
```

Arguments

data	data.fame of data.
vrb.nm	character vector of colnames from data specifying the variables.
grp.nm	character vector of colnames from data specifying the groups.
prop	logical vector of length 1 specifying whether the frequencies should be counts (FALSE) or proportions (TRUE). Note, whether the proportions include missing values depends on the useNA argument.
useNA	character vector of length 1 specifying how missing values should be handled. The three options are 1) "no" = do not include NA frequencies in the return object, 2) "ifany" = only NA frequencies if there are any missing values (in any variable from data[vrb.nm]), or 3) "always" = do include NA frequencies regardless of whether there are missing values or not.
total	character vector of length 1 specifying whether the frequencies for the set of variables as a whole should be returned. The name "total" refers to tabulating the frequencies for the variables from data[vrb.nm] together as a set. The three options are 1) "no" = do not include a row for the total frequencies in the return object, 2) "yes" = do include the total frequencies as the first row in the return object, or 3) "only" = only include the total frequencies as a single row in the return object and do not include rows for each of the individual column frequencies in data[vrb.nm].
sep	character vector of length 1 specifying the string to combine the group values together with. sep is only used if there are multiple grouping variables (i.e., length(grp.nm) > 1).

Details

freqs_by uses plyr::rbind.fill to combine the results from table applied to each variable into a single data.frame for each group. If a variable from data[vrb.nm] for each group does not have

values present in other variables from data[vrb.nm] for that group, then the frequencies in the return object will be 0.

The name for the table element giving the frequency of missing values is "(NA)". This is different from table where the name is NA_character_. This change allows for the sorting of tables that include missing values, as subsetting in R is not possible with NA_character_ names. In future versions of the package, this might change as it should be possible to avoid this issue by subetting with a logical vector or integer indices instead of names. However, it is convenient to be able to subset the return object fully by names.

Value

list of data.frames containing the frequencies for the variables in data[vrb.nm] by group. The number of list elements are the groups specified by unique(interaction(data[grp.nm], sep = sep)). Depending on prop, the frequencies are either counts (FALSE) or proportions (TRUE) by group. Depending on total, the nrow for each data.frame is either 1) length(vrb.nm) (if total = "no"), 1 + length(vrb.nm) (if total = "yes"), or 3) 1 (if total = "only"). The rownames are vrb.nm for each variable in data[vrb.nm] and "_total_" for the total row (if present). The colnames for each data.frame are the unique values present in data[vrb.nm], potentially including "(NA)" depending on useNA.

See Also

freqs freq_by freqs_by table

Examples

```
vrb_nm <- str2str::inbtw(names(psych::bfi), "A1","O5")
freqs_by(data = psych::bfi, vrb.nm = vrb_nm, grp.nm = "gender") # default
freqs_by(data = psych::bfi, vrb.nm = vrb_nm, grp.nm = "gender",
   prop = TRUE) # proportions by row
freqs_by(data = psych::bfi, vrb.nm = vrb_nm, grp.nm = "gender",
   useNA = "no") # without NA counts
freqs_by(data = psych::bfi, vrb.nm = vrb_nm, grp.nm = "gender",
   total = "yes") # include total counts
freqs_by(data = psych::bfi, vrb.nm = vrb_nm,
   grp.nm = c("gender","education")) # multiple grouping variables
```

---

freq_by *Univariate Frequency Table By Group*

Description

tables_by creates a frequency table for a set of variables in a data.frame by group. Depending on total, frequencies for all the variables together can be returned by group. The function probably makes the most sense for sets of variables with similar unique values (e.g., items from a questionnaire with similar response options).

Usage

```
freq_by(
  x,
  grp,
  exclude = if (useNA == "no") c(NA, NaN),
  useNA = "always",
  prop = FALSE,
  sort = "frequency",
  decreasing = TRUE,
  na.last = TRUE
)
```

Arguments

x	atomic vector.
grp	atomic vector or list of atomic vectors (e.g., data.frame) specifying the groups. The atomic vector(s) must be the length of x or else an error is returned.
exclude	unique values of x to exclude from the returned table. If NULL, then missing values are always included in the returned table. See table for documentation on the same argument.
useNA	character vector of length 1 specifying how to handle missing values (i.e., whether to include NA as an element in the returned table). There are three options: 1) "no" = don't include missing values in the table, 2) "ifany" = include missing values if there are any, 3) "always" = include missing values in the table, regardless of whether there are any or not. See table for documentation on the same argument.
prop	logical vector of length 1 specifying whether the returned table should include counts (FALSE) or proportions (TRUE). If NAs are excluded (e.g., useNA = "no" or exclude = c(NA, NaN)), then the proportions will be based on the number of observed elements.
sort	character vector of length 1 specifying how the returned table will be sorted. There are three options: 1) "frequency" = the frequency of the unique values in x, 2) "position" = the position when each unique value first appears in x, 3) "alphanum" = alphanumeric ordering of the unique values in x (the sorting used by table). When "frequency" is specified and there are ties, then the ties are sorted alphanumerically.
decreasing	logical vector of length 1 specifying whether the table should be sorted in decreasing (TRUE) or increasing (FALSE) order.
na.last	logical vector of length 1 specifying whether the table should have NAs last or in whatever position they end up at. This argument is only relevant if NAs exist in x and are included in the table (e.g., useNA = "always" or exclude = NULL).

Details

tables_by uses plyr::rbind.fill to combine the results from table applied to each variable into a single data.frame for each group. If a variable from data[vrb.nm] for each group does not

have values present in other variables from data[vrb.nm] for that group, then the frequencies in the return object will be 0.

The name for the table element giving the frequency of missing values is "(NA)". This is different from table where the name is NA_character_. This change allows for the sorting of tables that include missing values, as subsetting in R is not possible with NA_character_ names. In future versions of the package, this might change as it should be possible to avoid this issue by subetting with a logical vector or integer indices instead of names. However, it is convenient to be able to subset the return object fully by names.

Value

list of numeric vector of frequencies by group. The number of list elements are the groups specified by unique(interaction(grp, sep = sep)). The frequencies either counts (if prop = FALSE) or proportions (if prop = TRUE) with the unique values of x as names (missing values have name = "(NA)"). Note, this is different from table, which returns a 1D-array and has class "table".

See Also

freq freq_by freqs_by table

Examples

```
x <- freq_by(mtcars$"gear", grp = mtcars$"vs")
str(x)
y <- freq_by(mtcars$"am", grp = mtcars$"vs", useNA = "no")
str(y)
str2str::lv2m(lapply(X = y, FUN = rev), along = 1) # ready to pass to prop.test()
```

---

gtheory *Generalizability Theory Reliability of a Score*

Description

gtheory uses generalizability theory to compute the reliability coefficient of a score. It assumes single-level data where the rows are cases and the columns are variables/items. Generaliability theory coefficients in this case are the same as intraclass correlations (ICC). The default computes ICC(3,k), which is identical to cronbach's alpha, from cross.vrb = TRUE. When cross.vrb is FALSE, ICC(2,k) is computed, which takes mean differences between variables/items into account. gtheory is a wrapper function for ICC.

Usage

```
gtheory(
  data,
  vrb.nm,
  ci.type = "classic",
  level = 0.95,
  cross.vrb = TRUE,
```

```
  R = 200L,
  boot.ci.type = "perc"
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables/items.
ci.type	character vector of length = 1 specifying the type of confidence interval to compute. There are currently two options: 1) "classic" = traditional ICC-based confidence intervals (see details), 2) "boot" = bootstrapped confidence intervals.
level	double vector of length 1 specifying the confidence level from 0 to 1.
cross.vrb	logical vector of length 1 specifying whether the variables/items should be crossed when computing the generalizability theory coefficient. If TRUE, then only the covariance structure of the variables/items will be incorperated into the estimate of reliability. If FALSE, then the mean structure of the variables/items will be incorperated.
R	integer vector of length 1 specifying the number of bootstrapped resamples to use. Only used if ci.type = "boot".
boot.ci.type	character vector of length 1 specifying the type of bootstrapped confidence interval to compute. The options are 1) "perc" for the regular percentile method, 2) "bca" for bias-corrected and accelerated percentile method, 3) "norm" for the normal method that uses the bootstrapped standard error to construct symmetrical confidence intervals with the classic formula around the bias-corrected estimate, and 4) "basic" for the basic method. Note, "stud" for the studentized method is NOT an option. See boot.ci as well as confint2.boot for details.

Details

When ci.type = "classic" the confidence intervals are computed according to the formulas laid out by McGraw, Kenneth, and Wong, (1996). These are taken from the ICC function in the psych package. They are appropriately non-symmetrical given ICCs are not unbounded and range from 0 to 1. Therefore, there is no standard error associated with the coefficient. Note, they differ from the confidence intervals available in the cronbach function. When ci.type = "boot" the standard deviation of the empirical sampling distribution is returned as the standard error, which may or may not be trustworthy depending on the value of the ICC and sample size.

Value

double vector containing the generalizability theory coefficient, it's standard error (if ci.type = "boot"), and it's confidence interval.

References

McGraw, Kenneth O. and Wong, S. P. (1996), Forming inferences about some intraclass correlation coefficients. Psychological Methods, 1, 30-46. + errata on page 390.

See Also

gtheorys gtheory_ml cronbach

Examples

```
gtheory(attitude, vrb.nm = names(attitude), ci.type = "classic")
## Not run:
gtheory(attitude, vrb.nm = names(attitude), ci.type = "boot")
gtheory(attitude, vrb.nm = names(attitude), ci.type = "boot",
   R = 250L, boot.ci.type = "bca")

## End(Not run)

# comparison to cronbach's alpha:
gtheory(attitude, names(attitude))
gtheory(attitude, names(attitude), cross.vrb = FALSE)
a <- suppressMessages(psych::alpha(attitude)[["total"]]["raw_alpha"])
psych::alpha.ci(a, n.obs = 30, n.var = 7, digits = 7) # slightly different confidence interval
```

---

gtheorys *Generalizability Theory Reliability of Multiple Scores*

Description

gtheorys uses generalizability theory to compute the reliability coefficient of multiple scores. It assumes single-level data where the rows are cases and the columns are variables/items. Generaliability theory coefficients in this case are the same as intraclass correlations (ICC). The default computes ICC(3,k), which is identical to cronbach's alpha, from cross.vrb = TRUE. When cross.vrb is FALSE, ICC(2,k) is computed, which takes mean differences between variables/items into account. gtheorys is a wrapper function for ICC.

Usage

```
gtheorys(
  data,
  vrb.nm.list,
  ci.type = "classic",
  level = 0.95,
  cross.vrb = TRUE,
  R = 200L,
  boot.ci.type = "perc"
)
```

Arguments

data	data.frame of data.
vrb.nm.list	list of character vectors containing colnames from data specifying each set of variables/items.
ci.type	character vector of length = 1 specifying the type of confidence interval to compute. There are currently two options: 1) "classic" = traditional ICC-based confidence intervals (see details), 2) "boot" = bootstrapped confidence intervals.
level	double vector of length 1 specifying the confidence level from 0 to 1.
cross.vrb	logical vector of length 1 specifying whether the variables/items should be crossed when computing the generalizability theory coefficients. If TRUE, then only the covariance structure of the variables/items will be incorperated into the estimates of reliability. If FALSE, then the mean structure of the variables/items will be incorperated.
R	integer vector of length 1 specifying the number of bootstrapped resamples to use. Only used if ci.type = "boot".
boot.ci.type	character vector of length 1 specifying the type of bootstrapped confidence interval to compute. The options are 1) "perc" for the regular percentile method, 2) "bca" for bias-corrected and accelerated percentile method, 3) "norm" for the normal method that uses the bootstrapped standard error to construct symmetrical confidence intervals with the classic formula around the bias-corrected estimate, and 4) "basic" for the basic method. Note, "stud" for the studentized method is NOT an option. See boot.ci as well as confint2.boot for details.

Details

When ci.type = "classic" the confidence intervals are computed according to the formulas laid out by McGraw, Kenneth and Wong (1996). These are taken from the ICC function in the psych package. They are appropriately non-symmetrical given ICCs are not unbounded and range from 0 to 1. Therefore, there is no standard error associated with the coefficient. Note, they differ from the confidence intervals available in the cronbachs function. When ci.type = "boot" the standard deviation of the empirical sampling distribution is returned as the standard error, which may or may not be trustworthy depending on the value of the ICC and sample size.

Value

data.frame containing the generalizability theory statistical information. The columns are as follows:

est the generalizability theory coefficient itself

se standard error of the reliability coefficient

lwr lower bound of the confidence interval for the reliability coefficient

lwr lower bound of the confidence interval for the reliability coefficient

References

McGraw, Kenneth O. and Wong, S. P. (1996), Forming inferences about some intraclass correlation coefficients. Psychological Methods, 1, 30-46. + errata on page 390.

See Also

gtheory gtheorys_ml cronbachs

Examples

```
dat0 <- psych::bfi[1:100, ] # reduce number of rows
   # to reduce computational time of boot examples
dat1 <- str2str::pick(x = dat0, val = c("A1","C4","C5","E1","E2","O2","O5",
   "gender","education","age"), not = TRUE, nm = TRUE)
vrb_nm_list <- lapply(X = str2str::sn(c("E","N","C","A","O")), FUN = function(nm) {
   str2str::pick(x = names(dat1), val = nm, pat = TRUE)})
gtheorys(data = dat1, vrb.nm.list = vrb_nm_list)
## Not run:
gtheorys(data = dat1, vrb.nm.list = vrb_nm_list, ci.type = "boot") # singular messages
gtheorys(data = dat1, vrb.nm.list = vrb_nm_list, ci.type = "boot",
   R = 250L, boot.ci.type = "bca")

## End(Not run)
gtheorys(data = attitude,
   vrb.nm.list = list(names(attitude))) # also works with only one set of variables/items
```

---

gtheorys_ml *Generalizability Theory Reliability of Multiple Multilevel Scores*

Description

gtheorys_ml uses generalizability theory to compute the reliability coefficients of multiple multilevel score. It computes within-group coefficients that assess the reliability of the group-deviated scores (e.g., after calling centers_by) and between-group coefficients that assess the reliability of the mean aggregate scores (e.g., after calling aggs). It assumes two-level data where the rows are in long format and the columns are the variables/items of the score. Generaliability theory coefficients with multilevel data are analagous to intraclass correlations (ICC), but add an additional grouping variable. The default computes a multilevel version of ICC(3,k) from cross.obs = TRUE. When cross.obs = FALSE, a multilevel version of ICC(2,k) is computed, which takes mean differences between variables/items into account. gtheorys_ml is a wrapper function for mlr. Note, this function can take several minutes to run if you have a moderate to large dataset.

Usage

```
gtheorys_ml(data, vrb.nm.list, grp.nm, obs.nm, cross.obs = TRUE)
```

Arguments

data	data.frame of data.
vrb.nm.list	list of character vectors of colnames from data specifying the sets of variables/items.

`grp.nm` character vector of length 1 with colname from `data` specifying the grouping variable. Because `gtheorys_ml` is specific to two-level data, this can only be one variable.

`obs.nm` character vector of of length 1 with colname from `data` specifying the observation variable. In this context, observation refers to comparable cases across groups. In a longitudinal study, the groups are people and the observations are timepoints. For example, each person has a timepoint 1, timepoint 2, timepoint 3, etc. In an school study, the groups are classrooms and the observations are students. For example, each classroom has a student 1, student 2, student 3, etc. While longitudinal studies often have a time variable in their data, school studies don't have always a student variable. You would then have to create a student variable to be able to use this function.

`cross.obs` logical vector of length 1 specifying whether the observations should be crossed when computing the generalizability theory coefficients. If TRUE, the observations are treated as fixed; if FALSE, they are treated as random. See details.

Details

`gtheorys_ml` uses `mlr`, which is based on the formulas in Shrout, Patrick, and Lane (2012). When `cross.obs` = TRUE, the within-group coefficient is Rc and the between-group coefficient is RkF. When `cross.obs` = FALSE, the within-group coefficient is Rcn and the between-group coefficient is RkRn.

`gtheorys_ml` does not currently have standard errors or confidence intervals. I am not aware of mathematical formulas for analytical confidence intervals, and because the generaliability theory coefficients can take several minutes to estimate, bootstraped confidence intervals seem too time-intensive to be useful at the moment.

`gtheorys_ml` does not work with multiple single variable/item scores. You can still use generalizability theory to estimate between-group reliability in that instance though. To do so, reshape the multiple single variables/items from long to wide (e.g., `long2wide`) so that you have a column for each observation of that single variable/item and the rows are the groups. Then you can use `gtheorys` and treat each observation as a "different" variable/item.

Value

list with two elements. The first is named "within" and refers to the within-group reliability. The second is named "between" and refers to the between-group reliability. Each contains a data.frame with the following columns:

est generalizability theory reliability coefficient itself

average_r the average correlation at each level of the data based on `cor_ml` (which is a wrapper for `statsBy`)

nvrb number of variables/items that make up that score

The later two columns are included because even though the reliability coefficients are calculated from variance components, they are indirectly based on the average correlation and number of variables/items similar to Cronbach's alpha.

References

Shrout, Patrick and Lane, Sean P (2012), Psychometrics. In M.R. Mehl and T.S. Conner (eds) Handbook of research methods for studying daily life, (p 302-320) New York. Guilford Press

See Also

gtheory_ml gtheorys

Examples

```
dat <- psychTools::sai[psychTools::sai$"study" == "VALE", ] # 4 timepoints
vrb_nm_list <- list("positive_affect" = c("calm","secure","at.ease","rested",
   "comfortable","confident"), # extra: "relaxed","content","joyful"
   "negative_affect" = c("tense","regretful","upset","worrying","anxious",
      "nervous")) # extra: "jittery","high.strung","worried","rattled"
suppressMessages(gtheorys_ml(data = dat, vrb.nm.list = vrb_nm_list, grp.nm = "id",
   obs.nm = "time", cross.obs = TRUE))
suppressMessages(gtheorys_ml(data = dat, vrb.nm.list = vrb_nm_list, grp.nm = "id",
   obs.nm = "time", cross.obs = FALSE))
gtheorys_ml(data = dat, vrb.nm.list = vrb_nm_list["positive_affect"], grp.nm = "id",
   obs.nm = "time") # also works with only one set of variables/items
```

---

gtheory_ml *Generalizability Theory Reliability of a Multilevel Score*

Description

gtheory_ml uses generalizability theory to compute the reliability coefficients of a multilevel score. It computes a within-group coefficient that assesses the reliability of the group-deviated score (e.g., after calling center_by) and a between-group coefficient that assess the reliability of the mean aggregate score (e.g., after calling agg). It assumes two-level data where the rows are in long format and the columns are the variables/items of the score. Generaliability theory coefficients with multilevel data are analagous to intraclass correlations (ICC), but add an additional grouping variable. The default computes a multilevel version of ICC(3,k) from cross.obs = TRUE. When cross.obs = FALSE, a multilevel version of ICC(2,k) is computed, which takes mean differences between variables/items into account. gtheory_ml is a wrapper function for mlr. Note, this function can take several minutes to run if you have a moderate to large dataset.

Usage

```
gtheory_ml(data, vrb.nm, grp.nm, obs.nm, cross.obs = TRUE)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables/items.

`grp.nm` character vector of length 1 with colname from `data` specifying the grouping variable. Because `gtheory_ml` is specific to two-level data, this can only be one variable.

`obs.nm` character vector of of length 1 with colname from `data` specifying the observation variable. In this context, observation refers to comparable cases across groups. In a longitudinal study, the groups are people and the observations are timepoints. For example, each person has a timepoint 1, timepoint 2, timepoint 3, etc. In an school study, the groups are classrooms and the observations are students. For example, each classroom has a student 1, student 2, student 3, etc. While longitudinal studies often have a time variable in their data, school studies don't always have a student variable. You would then have to create a student variable to be able to use `gtheory_ml`.

`cross.obs` logical vector of length 1 specifying whether the observations should be crossed when computing the generalizability theory coefficient. If TRUE, the observations are treated as fixed; if FALSE, they are treated as random. See details.

Details

`gtheory_ml` uses `mlr`, which is based on the formulas in Shrout, Patrick, and Lane (2012). When `cross.obs` = TRUE, the within-group coefficient is Rc and the between-group coefficient is RkF. When `cross.obs` = FALSE, the within-group coefficient is Rcn and the between-group coefficient is RkRn.

`gtheory_ml` does not currently have standard errors or confidence intervals. I am not aware of mathematical formulas for analytical confidence intervals, and because the generaliability theory coefficients can take several minutes to estimate, bootstraped confidence intervals seem too time-intensive to be useful at the moment.

`gtheory_ml` does not work with a single variable/item. You can still use generalizability theory to estimate between-group reliability in that instance though. To do so, reshape the variable/item from long to wide (e.g., `unstack2`) so that you have a column for each observation of that single variable/item and the rows are the groups. Then you can use `gtheory` and treat each observation as a "different" variable/item.

Value

list with two elements. The first is named "within" and refers to the within-group reliability. The second is named "between" and refers to the between-group reliability. Each contains a double vector where the first element is named "est" and contains the generalizability theory coefficient itself. The second element is named "average_r" and contains the average correlation at that level of the data based on `cor_ml` (which is a wrapper for `statsBy`). The third element is named "nvrb" and contains the number of variables/items. These later two elements are included because even though the reliability coefficients are calculated from variance components, they are indirectly based on the average correlation and number of variables/items, similar to Cronbach's alpha.

References

Shrout, Patrick and Lane, Sean P (2012), Psychometrics. In M.R. Mehl and T.S. Conner (eds) Handbook of research methods for studying daily life, (p 302-320) New York. Guilford Press

See Also

gtheorys_ml gtheory

Examples

```
shrout <- structure(list(Person = c(1L, 2L, 3L, 4L, 5L, 1L, 2L, 3L, 4L,
   5L, 1L, 2L, 3L, 4L, 5L, 1L, 2L, 3L, 4L, 5L), Time = c(1L, 1L,
      1L, 1L, 1L, 2L, 2L, 2L, 2L, 2L, 3L, 3L, 3L, 3L, 3L, 4L, 4L, 4L,
      4L, 4L), Item1 = c(2L, 3L, 6L, 3L, 7L, 3L, 5L, 6L, 3L, 8L, 4L,
         4L, 7L, 5L, 6L, 1L, 5L, 8L, 8L, 6L), Item2 = c(3L, 4L, 6L, 4L,
            8L, 3L, 7L, 7L, 5L, 8L, 2L, 6L, 8L, 6L, 7L, 3L, 9L, 9L, 7L, 8L
         ), Item3 = c(6L, 4L, 5L, 3L, 7L, 4L, 7L, 8L, 9L, 9L, 5L, 7L,
            9L, 7L, 8L, 4L, 7L, 9L, 9L, 6L)), .Names = c("Person", "Time",
               "Item1", "Item2", "Item3"), class = "data.frame", row.names = c(NA,
                  -20L))
mlr_obj <- psych::mlr(x = shrout, grp = "Person", Time = "Time",
   items = c("Item1", "Item2", "Item3"),
   alpha = FALSE, icc = FALSE, aov = FALSE, lmer = TRUE, lme = FALSE,
   long = FALSE, plot = FALSE)
gtheory_ml(data = shrout, vrb.nm = c("Item1", "Item2", "Item3"),
   grp.nm = "Person", obs.nm = "Time", cross.obs = TRUE) # crossed time
gtheory_ml(data = shrout, vrb.nm = c("Item1", "Item2", "Item3"),
   grp.nm = "Person", obs.nm = "Time", cross.obs = FALSE) # nested time
```

---

iccs_11 *Intraclass Correlation for Multiple Variables for Multilevel Analysis: ICC(1,1)*

Description

iccs_11 computes the intraclass correlation (ICC) for multiple variables based on a single rater with a single dimension, aka ICC(1,1). Traditionally, this is the type of ICC used for multilevel analysis where the value is interpreted as the proportion of variance accounted for by group membership. In other words, ICC(1,1) = the proportion of between-group variance; 1 - ICC(1,1) = the proportion of within-group variance.

Usage

```
iccs_11(data, vrb.nm, grp.nm, how = "lme", REML = FALSE)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variable columns.
grp.nm	character vector of length 1 of a colname from data specifying the grouping column.

how — character vector of length 1 specifying how the ICC(1,1) should be calculated. There are four options: 1) "lme" uses a linear mixed effects model with the function lme from the package nlme, 2) "lmer" uses a linear mixed effects modeling with the function lmer from the package lme4, 3) "aov" uses a one-way analysis of variance with the function aov, and 4) "raw" uses the observed variances, which provides a biased estimate of the ICC(1,1) and is not recommended (It is only included for teaching purposes).

REML — logical vector of length 1 specifying whether restricted maximum likelihood estimation (TRUE) should be used rather than traditional maximum likelihood (FALSE). This is only applicable to linear mixed effects models when how is "lme" or "lmer".

Value

double vector containing ICC(1, 1) of the vrb.nm columns in data with names of the return object equal to vrb.nm.

See Also

icc_11 # ICC(1,1) for a single variable, icc_all_by # all six types of ICCs by group, lme # how = "lme" function, lmer # how = "lmer" function, aov # how = "aov" function,

Examples

```
tmp_nm <- c("outcome","case","session","trt_time")
dat <- as.data.frame(lmeInfo::Bryant2016)[tmp_nm]
stats_by <- psych::statsBy(dat,
   group = "case") # requires you to include "case" column in dat
iccs_11(data = dat, vrb.nm = c("outcome","session","trt_time"), grp.nm = "case")
```

---

icc_11 — *Intraclass Correlation for Multilevel Analysis: ICC(1,1)*

Description

icc_11 computes the intraclass correlation (ICC) based on a single rater with a single dimension, aka ICC(1,1). Traditionally, this is the type of ICC used for multilevel analysis where the value is interpreted as the proportion of variance accounted for by group membership. In other words, ICC(1,1) = the proportion of between-group variance; 1 - ICC(1,1) = the proportion of within-group variance.

Usage

```
icc_11(x, grp, how = "lme", REML = TRUE)
```

Arguments

x	numeric vector.
grp	atomic vector the same length as x providing the grouping variable.
how	character vector of length 1 specifying how the ICC(1,1) should be calculated. There are four options: 1) "lme" uses a linear mixed effects model with the function lme from the package nlme, 2) "lmer" uses a linear mixed effects modeling with the function lmer from the package lme4, 3) "aov" uses a one-way analysis of variance with the function aov, and 4) "raw" uses the observed variances, which provides a biased estimate of the ICC(1,1) and is not recommended (It is only included for teaching purposes).
REML	logical vector of length 1 specifying whether restricted maximum likelihood estimation (TRUE) should be used rather than traditional maximum likelihood estimation (FALSE). Only used for linear mixed effects models if how = "lme" or how = "lmer".

Value

numeric vector of length 1 providing ICC(1,1) and computed based on the how argument.

See Also

iccs_11 # ICC(1,1) for multiple variables, icc_all_by # all six types of ICCs by group, lme # how = "lme" function, lmer # how = "lmer" function, aov # how = "aov" function,

Examples

```
# BALANCED DATA (how = "aov" and "lme"/"lmer" do YES provide the same value)

str(InsectSprays)
icc_11(x = InsectSprays$"count", grp = InsectSprays$"spray", how = "aov")
icc_11(x = InsectSprays$"count", grp = InsectSprays$"spray", how = "lme")
icc_11(x = InsectSprays$"count", grp = InsectSprays$"spray", how = "lmer")
icc_11(x = InsectSprays$"count", grp = InsectSprays$"spray",
  how = "raw") # biased estimator and not recommended. Only available for teaching purposes.

# UN-BALANCED DATA (how = "aov" and "lme"/"lmer" do NOT provide the same value)

dat <- as.data.frame(lmeInfo::Bryant2016)
icc_11(x = dat$"outcome", grp = dat$"case", how = "aov")
icc_11(x = dat$"outcome", grp = dat$"case", how = "lme")
icc_11(x = dat$"outcome", grp = dat$"case", how = "lmer")
icc_11(x = dat$"outcome", grp = dat$"case", how = "lme", REML = FALSE)
icc_11(x = dat$"outcome", grp = dat$"case", how = "lmer", REML = FALSE)

# how = "lme" does not account for any correlation structure
lme_obj <- nlme::lme(outcome ~ 1, random = ~ 1 | case,
   data = dat, na.action = na.exclude,
   correlation = nlme::corAR1(form = ~ 1 | case), method = "REML")
var_corr <- nlme::VarCorr(lme_obj) # VarCorr.lme
vars <- as.double(var_corr[, "Variance"])
```

```
btw <- vars[1]
wth <- vars[2]
btw / (btw + wth)
```

`icc_all_by` *All Six Intraclass Correlations by Group*

Description

`icc_all_by` computes each of the six intraclass correlations (ICC) in Shrout & Fleiss (1979) by group. The ICCs differ by whether they treat dimensions as fixed or random and whether they are for a single variable in `data[vrb.nm]` of the set of variables `data[vrb.nm]`. `icc_all_by` also returns information about the linear mixed effects modeling (using `lmer`) used to compute the ICCs as well as any warning or error messages by group. For an understanding of the six different ICCs, see the following blogpost: http://www.daviddisabato.com/blog/2021/10/1/the-six-different-types-of-intraclass-correlations-iccs. `icc_all_by` is a combination of `by2` + `try_fun` + `ICC` (`ICC` calls `lmer` internally).

Usage

```
icc_all_by(data, vrb.nm, grp.nm, ci.level = 0.95, check = TRUE)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`grp.nm`	character vector of colnames from `data` specifying the groups.
`ci.level`	double vector of length 1 specifying the confidence level. It must range from 0 to 1.
`check`	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `data[vrb.nm]` are all typeof numeric. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Details

`icc_all_by` internally suppresses any messages, warnings, or errors returned by `lmer` (e.g., "boundary (singular) fit: see ?isSingular") because that information is provided in the returned data.frame.

Value

data.frame containing the unique combinations of the grouping variables `data[grp.nm]` and each group's intraclass correlations (ICCs), their confidence intervals, information about the `merMod` object from the linear mixed effects model, and any warning or error messages from `lmer`. For an understanding of the six different ICCs, see the following blogpost: http://www.daviddisabato.com/blog/2021/10/1/the-six-different-types-of-intraclass-correlations-iccs. The

first columns are always `unique.data.frame(data[vrb.nm])`. All other columns are in the following order with the following colnames:

icc11_est ICC(1,1) parameter estimate

icc11_lwr ICC(1,1) lower bound of the confidence interval

icc11_upr ICC(1,1) lower bound of the confidence interval

icc21_est ICC(2,1) parameter estimate

icc21_lwr ICC(2,1) lower bound of the confidence interval

icc21_upr ICC(2,1) lower bound of the confidence interval

icc31_est ICC(3,1) parameter estimate

icc31_lwr ICC(3,1) lower bound of the confidence interval

icc31_upr ICC(3,1) lower bound of the confidence interval

icc1k_est ICC(1,k) parameter estimate

icc1k_lwr ICC(1,k) lower bound of the confidence interval

icc1k_upr ICC(1,k) lower bound of the confidence interval

icc2k_est ICC(2,k) parameter estimate

icc2k_lwr ICC(2,k) lower bound of the confidence interval

icc2k_upr ICC(2,k) lower bound of the confidence interval

icc3k_est ICC(3,k) parameter estimate

icc3k_lwr ICC(3,k) lower bound of the confidence interval

icc3k_upr ICC(3,k) lower bound of the confidence interval

lmer_nobs number of observations used for the linear mixed effects model. Note, this is the number of (non-missing) rows after `data[vrb.nm]` has been stacked together via `stack`.

lmer_ngrps number of groups used for the linear mixed effects model. This is the number of unique combinations of the grouping variables after `data[grp.nm]`.

lmer_logLik logLik of the linear mixed effects model

lmer_sing binary variable where 1 = the linear mixed effects model had a singularity in the random effects covariance matrix or 0 = it did not

lmer_warn binary variable where 1 = the linear mixed effects model returned a warning or 0 = it did not

lmer_err binary variable where 1 = the linear mixed effects model returned an error or 0 = it did not

warn_mssg character vector providing the warning messages for any warnings. If a group did not generate a warning, then the value is NA

err_mssg character vector providing the error messages for any warnings. If a group did not generate an error, then the value is NA

References

Shrout, P.E., & Fleiss, J.L. (1979). Intraclass correlations: Uses in assessing rater reliability. Psychological Bulletin, 86(2), 420-428.

See Also

ICC lmer

Examples

```
# one grouping variable
x <- icc_all_by(data = psych::bfi, vrb.nm = c("A2","A3","A4","A5"),
   grp.nm = "gender")

# two grouping variables
y <- icc_all_by(data = psych::bfi, vrb.nm = c("A2","A3","A4","A5"),
   grp.nm = c("gender","education"))

# with errors
z <- icc_all_by(data = psych::bfi, vrb.nm = c("A2","A3","A4","A5"),
   grp.nm = c("age")) # NA for all ICC columns when there is an error
```

---

lengths_by *Length of Data Columns by Group*

Description

lengths_by computes the the length of multiple columns in a data.frame by group. The argument na.rm can be used to include (FALSE) or exclude (TRUE) missing values. Through the use of na.rm = TRUE, the number of observed values for each variable by each group can be computed.

Usage

```
lengths_by(data, vrb.nm, grp.nm, na.rm = FALSE, sep = ".")
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.
grp.nm	character vector of colnames from data specifying the groups.
na.rm	logical vector of length 1 specifying whether to include (FALSE) or exclude (TRUE) missing values.
sep	character vector of length 1 specifying what string should separate different group values when naming the return object. This argument is only used if grp is a list of atomic vectors (e.g., data.frame).

Value

data.frame with colnames = vrb.nm and rownames = length(levels(interaction(grp))) providing the number of elements (excluding missing values if na.rm = TRUE) in each column by group.

See Also

length_by length colNA

Examples

```
lengths_by(mtcars, vrb.nm = c("mpg","cyl","disp"), grp = "gear")
lengths_by(mtcars, vrb.nm = c("mpg","cyl","disp"),
   grp = c("gear","vs")) # can handle multiple grouping variables
lengths_by(mtcars, vrb.nm = c("mpg","cyl","disp"),
   grp = c("gear","am")) # can handle zero lengths
lengths_by(airquality, c("Ozone","Solar.R","Wind"), grp = "Month",
   na.rm = FALSE) # include missing values
lengths_by(airquality, c("Ozone","Solar.R","Wind"), grp = "Month",
   na.rm = TRUE) # exclude missing values
```

---

length_by *Length of a (Atomic) Vector by Group*

Description

length_by computes the the length of a (atomic) vector by group. The argument na.rm can be used to include (FALSE) or exclude (TRUE) missing values.

Usage

```
length_by(x, grp, na.rm = FALSE, sep = ".")
```

Arguments

x	atomic vector.
grp	atomic vector or list of atomic vectors (e.g., data.frame) specifying the groups. The atomic vector(s) must be the length of x or else an error is returned.
na.rm	logical vector of length 1 specifying whether to include (FALSE) or exclude (TRUE) missing values.
sep	character vector of length 1 specifying what string should separate different group values when naming the return object. This argument is only used if grp is a list of atomic vectors (e.g., data.frame).

Value

integer vector of length = length(levels(interaction(grp))) with names = length(levels(interaction(grp))) providing the number of elements (excluding missing values if na.rm = TRUE) in each group.

See Also

lengths_by length agg

Examples

```
length_by(x = mtcars$"mpg", grp = mtcars$"gear")
length_by(x = airquality$"Ozone", grp = airquality$"Month", na.rm = FALSE)
length_by(x = airquality$"Ozone", grp = airquality$"Month", na.rm = TRUE)
```

---

long2wide *Reshape Multiple Scores From Long to Wide*

Description

long2wide reshapes data from long to wide. This if often necessary to do with multilevel data where variables in the long format seek to be reshaped to multiple sets of variables in the wide format. If only one column needs to be reshaped, then you can use unstack2 or cast - but that does not work for *multiple* columns.

Usage

```
long2wide(
  data,
  vrb.nm,
  grp.nm,
  obs.nm,
  sep = ".",
  colnames.by.obs = TRUE,
  keep.attr = FALSE
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables to be reshaped. In longitudinal panel data, this would be the scores.
grp.nm	character vector of colnames from data specifying the groups. In longitudnal panel data, this would be the participant ID variable.
obs.nm	character vector of length 1 with a colname from data specifying the observation within each group. In longitudinal panel data, this would be the time variable.
sep	character vector of length 1 specifying the string that separates the name prefix (e.g., score) from it's number suffix (e.g., timepoint). If sep = "", then that implies there is no string separating the name prefix and the number suffix (e.g., "outcome1").
colnames.by.obs	logical vector of length 1 specifying whether to sort the return object colnames by the observation label (TRUE) or by the order of vrb.nm. See the example at the end of the "MULTIPLE GROUPING VARIABLES" section of the examples.

keep.attr — logical vector of length 1 specifying whether to keep the "reshapeWide" attribute (from reshape) in the return object.

Details

long2wide uses reshape(direction = "wide") to reshape the data. It attempts to streamline the task of reshaping long to wide as the reshape arguments can be confusing because the same arguments are used for wide vs. long reshaping. See reshape if you are curious.

Value

data.frame with nrow equal to nrow(unique(data[grp.nm])) and number of reshaped columns equal to length(vrb.nm) * unique(data[[obs.nm]]). The colnames will have the structure paste0(vrb.nm, sep, unique(data[[obs.nm]])). The reshaped colnames are sorted by the observation labels if colnames.by.obs = TRUE and sorted by vrb.nm if colnames.by.obs = FALSE. Overall, the columns are in the following order: 1) grp.nm of the groups, 2) reshaped columns, 3) additional columns that were not reshaped.

See Also

wide2long reshape unstack2

Examples

```
# SINGLE GROUPING VARIABLE
dat_long <- as.data.frame(ChickWeight) # b/c groupedData class does weird things...
w1 <- long2wide(data = dat_long, vrb.nm = "weight", grp.nm = "Chick",
   obs.nm = "Time") # NAs inserted for missing observations in some groups
w2 <- long2wide(data = dat_long, vrb.nm = "weight", grp.nm = "Chick",
   obs.nm = "Time", sep = "_")
head(w1); head(w2)
w3 <- long2wide(data = dat_long, vrb.nm = "weight", grp.nm = "Chick",
   obs.nm = "Time", sep = "_T", keep.attr = TRUE)
attributes(w3)

# MULTIPLE GROUPING VARIABLE
tmp <- psychTools::sai
grps <- interaction(tmp[1:3], drop = TRUE)
dups <- duplicated(grps)
dat_long <- tmp[!(dups), ] # for some reason there are duplicate groups in the data
vrb_nm <- str2str::pick(names(dat_long), val = c("study","time","id"), not = TRUE)
w4 <- long2wide(data = dat_long, vrb.nm = vrb_nm, grp.nm = c("study","id"),
   obs.nm = "time")
w5 <- long2wide(data = dat_long, vrb.nm = vrb_nm, grp.nm = c("study","id"),
   obs.nm = "time", colnames.by.obs = FALSE) # colnames sorted by `vrb.nm` instead
head(w4); head(w5)
```

make.dummy *Make Dummy Columns*

Description

make.dummy creates dummy columns (i.e., dichotomous numeric vectors coded 0 and 1) from logical conditions. If you want to make logical conditions from columns of a data.frame, you will need to call the data.frame and its columns explicitly as this function does not use non-standard evaluation.

Usage

```
make.dummy(..., rtn.lgl = FALSE)
```

Arguments

...	logical conditions that evaluate to logical vectors of the same length. If the logical vectors are not the same length, an error is returned. The names of the arguments are the colnames in the return object. If unnamed, then default R data.frame naming is used, which can get ugly.
rtn.lgl	logical vector of length 1 specifying whether the dummy columns should be logical vectors (TRUE) rather than numeric vectors (FALSE).

Value

data.frame of dummy columns based on the logical conditions n ... . If rtn.lgl = TRUE, then the columns are logical vectors. If out.lgl = FALSE, then the columns are numeric vectors where 0 = FALSE and 1 = TRUE. The colnames are the names of the arguments in ... If not specified, then default data.frame names are created from the logical conditions themselves (which can get ugly).

See Also

make.dumNA

Examples

```
make.dummy(attitude$"rating" > 50) # ugly colnames
make.dummy("rating_50plus" = attitude$"rating" > 50,
   "advance_50minus" = attitude$"advance" < 50)
make.dummy("rating_50plus" = attitude$"rating" > 50,
   "advance_50minus" = attitude$"advance" < 50, rtn.lgl = TRUE)
## Not run:
   make.dummy("rating_50plus" = attitude$"rating" > 50,
      "mpg_20plus" = mtcars$"mpg" > 20)

## End(Not run)
```

make.dumNA *Make Dummy Columns For Missing Data.*

Description

make.dumNA makes dummy columns (i.e., dichomotous numeric vectors coded 0 and 1) for missing data. Each variable is treated in isolation.

Usage

```
make.dumNA(data, vrb.nm, ov = FALSE, rtn.lgl = FALSE, suffix = "_m")
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.
ov	logical vector of length 1 specifying whether the dummy columns should be reverse coded such that missing values = 0/FALSE and observed values = 1/TRUE.
rtn.lgl	logical vector of length 1 specifying whether the dummy columns should be logical vectors (TRUE) rather than numeric vectors (FALSE).
suffix	character vector of length 1 specifying the string that should be appended to the end of the colnames in the return object.

Value

data.frame of numeric (logical if rtn.lgl = TRUE) columns where missing = 1 and observed = 0 (flipped if ov = TRUE) for each variable. The colnames are created by paste0(vrb.nm, suffix).

See Also

make.dummy

Examples

```
make.dumNA(data = airquality, vrb.nm = c("Ozone","Solar.R"))
make.dumNA(data = airquality, vrb.nm = c("Ozone","Solar.R"),
   rtn.lgl = TRUE) # logical vectors returned
make.dumNA(data = airquality, vrb.nm = c("Ozone","Solar.R"),
   ov = TRUE, suffix = "_o") # 1 = observed value
```

make.fun_if *Make a Function Conditional on Frequency of Observed Values*

Description

make.fun_if makes a function that evaluates conditional on a specified minimum frequency of observed values. Within the function, if the frequency of observed values is less than (or equal to) ov.min, then false is returned rather than the return value.

Usage

```
make.fun_if(
  fun,
  ...,
  ov.min.default = 1,
  prop.default = TRUE,
  inclusive.default = TRUE,
  false = NA
)
```

Arguments

fun	function that takes an atomic vector as its first argument. The first argument does not have to be named "x" within fun, but it will be named "x" in the returned function.
...	additional arguments with parameters to fun. This would be similar to impute in sum_if. However in the current version of make.fun_if, the parameters you provide will always be used within the returned function and cannot be specified by the user of the returned function. Unfortunately, I cannot figure out how to include user-specified arguments (with defaults) within the returned function other than ov.min.default, prop.default, and inclusive.default.
ov.min.default	numeric vector of length 1 specifying what the default should be for the argument ov.min within the returned function, which specifies the minimum frequency of observed values required. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and length(x).
prop.default	logical vector of length 1 specifying what the default should be for the argument prop within the returned function, which specifies whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
inclusive.default	logical vector of length 1 speicfying what the default should be for the argument inclusive within the returned function, which specifies whether the function should be evaluated if the frequency of observed values is exactly equal to ov.min.
false	vector of length 1 specifying what should be returned if the observed values condition is not met within the returned function. The default is NA. Whatever the value is, it will be coerced to the same mode as x within the returned function.

Value

function that takes an atomic vector `x` as its first argument, `...` as other arguments, ending with `ov.min`, `prop`, and `inclusive` as final arguments with defaults specified by `ov.min.default`, `prop.default`, and `inclusive.default`, respectively.

See Also

`sum_if` `mean_if`

Examples

```
# SD
sd_if <- make.fun_if(fun = sd, na.rm = TRUE) # always have na.rm = TRUE
sd_if(x = airquality[[1]], ov.min = .75) # proportion of observed values
sd_if(x = airquality[[1]], ov.min = 116,
   prop = FALSE) # count of observed values
sd_if(x = airquality[[1]], ov.min = 116, prop = FALSE,
   inclusive = FALSE) # not include ov.min values itself

# skewness
skew_if <- make.fun_if(fun = psych::skew, type = 1) # always have type = 1
skew_if(x = airquality[[1]], ov.min = .75) # proportion of observed values
skew_if(x = airquality[[1]], ov.min = 116,
   prop = FALSE) # count of observed values
skew_if(x = airquality[[1]], ov.min = 116, prop = FALSE,
   inclusive = FALSE) # not include ov.min values itself

# mode
popular <- function(x) names(sort(table(x), decreasing = TRUE))[1]
popular_if <- make.fun_if(fun = popular) # works with character vectors too
popular_if(x = c(unlist(dimnames(HairEyeColor)), rep.int(x = NA, times = 10)),
   ov.min = .50)
popular_if(x = c(unlist(dimnames(HairEyeColor)), rep.int(x = NA, times = 10)),
   ov.min = .60)
```

---

make.latent *Make Model Syntax for a Latent Factor in Lavaan*

Description

`make.latent` makes the model syntax for a latent factor in `lavaan`. The return object can be used as apart of the model syntax for calls to `lavaan`, `sem`, `cfa`, etc.

Usage

```
make.latent(
  x,
  nm.latent = "latent",
  error.var = FALSE,
```

```
  nm.par = FALSE,
  suffix.load = "_l",
  suffix.error = "_e"
)
```

Arguments

x	character vector specifying the colnames in your data that correspond to the variables indicating the latent factor (e.g., questionnaire items).
nm.latent	character vector of length 1 specifying what the latent factor should be labeled as in the return object.
error.var	logical vector of length 1 specifying whether the model syntax for the error variances should be included in the return object.
nm.par	logical vector of length 1 specifying whether the model syntax should include names for the factor loading (and error variance) parameters.
suffix.load	character vector of length 1 specifying what string should be appended to the end of the elements of x when creating names for the factor loading parameters. Only used if nm.par is TRUE.
suffix.error	character vector of length 1 specifying what string should be appended to the end of the elements of x when creating names for the error variance parameters. Only used if nm.par is TRUE.

Value

character vector of length 1 providing the model syntax. The regular expression "\n" is used to delineate new lines within the model syntax.

Examples

```
make.latent(x = names(psych::bfi)[1:5], error.var = FALSE, nm.par = FALSE)
make.latent(x = names(psych::bfi)[1:5], error.var = FALSE, nm.par = TRUE)
make.latent(x = names(psych::bfi)[1:5], error.var = TRUE, nm.par = FALSE)
make.latent(x = names(psych::bfi)[1:5], error.var = TRUE, nm.par = TRUE)
```

---

make.product *Make Product Terms (e.g., interactions)*

Description

make.product creates product terms (i.e., interactions) from various components. make.product uses Center for the optional of centering and/or scaling the predictors and/or moderators before making the product terms.

Usage

```
make.product(
  data,
  x.nm,
  m.nm,
  center.x = FALSE,
  center.m = FALSE,
  scale.x = FALSE,
  scale.m = FALSE,
  suffix.x = "",
  suffix.m = "",
  sep = ":",
  combo = TRUE
)
```

Arguments

data	data.frame of data.
x.nm	character vector of colnames from data specifying the predictor columns.
m.nm	character vector of colnames from data specifying the moderator columns.
center.x	logical vector of length 1 specifying whether the predictor columns should be grand-mean centered before making the product terms.
center.m	logical vector of length 1 specifying whether the moderator columns should be grand-mean centered before making the product terms.
scale.x	logical vector of length 1 specifying whether the predictor columns should be grand-SD scaled before making the product terms.
scale.m	logical vector of length 1 specifying whether the moderator columns should be grand-SD scaled before making the product terms.
suffix.x	character vector of length 1 specifying any suffix to add to the end of the predictor colnames x.nm when creating the colnames of the return object.
suffix.m	character vector of length 1 specifying any suffix to add to the end of the moderator colnames m.nm when creating the colnames of the return object.
sep	character vector of length 1 specifying the string to connect x.nm and m.nm when specifying the colnames of the return object.
combo	logical vector of length 1 specifying whether all combinations of the predictors and moderators should be calculated or only those in parallel to each other (i.e., x.nm[i] and m.nm[i]). This argument is only applicable when multiple predictors AND multiple moderators are given.

Value

data.frame with product terms (e.g., interactions) as columns. The colnames are created by paste(paste0(x.nm, suffix.x), paste0(m.nm, suffix.m), sep = sep).

Examples

```
make.product(data = attitude, x.nm = c("complaints","privileges"),
   m.nm = "learning", center.x = TRUE, center.m = TRUE,
   suffix.x = "_c", suffix.m = "_c") # with grand-mean centering
make.product(data = attitude, x.nm = c("complaints","privileges"),
   m.nm = c("learning","raises"), combo = TRUE) # all possible combinations
make.product(data = attitude, x.nm = c("complaints","privileges"),
   m.nm = c("learning","raises"), combo = FALSE) # only combinations "in parallel"
```

---

means_change *Mean Changes Across Two Timepoints For Multiple PrePost Pairs of Variables (dependent two-samples t-tests)*

Description

means_change tests for mean changes across two timepoints for multiple prepost pairs of variables via dependent two-samples t-tests. The function also calculates the descriptive statistics for the timepoints and the standardized mean differences (i.e., Cohen's d) based on either the standard deviation of the pre-timepoint, pooled standard deviation of the pre-timepoint and post-timepoint, or the standard deviation of the change score (post - pre). means_change is simply a wrapper for t.test plus some extra calculations.

Usage

```
means_change(
  data,
  prepost.nm.list,
  standardizer = "pre",
  d.ci.type = "unbiased",
  ci.level = 0.95,
  check = TRUE
)
```

Arguments

data	data.frame of data.
prepost.nm.list	list of length-2 character vectors specifying the colnames from data corresponding to the prepost pairs of variables. For each element of the list, the character vector should have length 2 where the first element corresponds to the pre-timepoint variable colname of that prepost pair and the second element corresponds to the post-timepoint variable colname of that prepost pair. The names of the list will be the rownames in the data.frames of the return object. See examples. prepost.nm.list can also be a single length-2 character vector for the case of a single pre-post pair of variables, which is functionally equivalent to mean_change.

`standardizer` chararacter vector of length 1 specifying what to use for standardization when computing the standardized mean difference (i.e., Cohen's d). There are three options: 1. "pre" for the standard deviation of the pre-timepoint, 2. "pooled" for the pooled standard deviation of the pre-timepoint and post-timepoint, 3. "change" for the standard deviation of the change score (post - pre). The default is "pre", which I believe makes the most theoretical sense (see Cumming, 2012); however, "change" is the traditional choice originally proposed by Jacob Cohen (Cohen, 1988).

`d.ci.type` character vector of lenth 1 specifying how to compute the confidence intervals (and standard errors) of the standardized mean differences. There are currently two options: 1. "unbiased" which calculates the unbiased standard error of Cohen's d based on the formulas in Viechtbauer (2007). If `standardizer` = "pre" or "pooled", then equation 36 from Table 2 is used. If `standardizer` = "change", then equation 25 from Table 1 is used. A symmetrical confidence interval is then calculated based on the standard error. 2. "classic" which calculates the confidence interval of Cohen's d based on the confidence interval of the mean change itself. The lower and upper confidence bounds are divided by the `standardizer`. Technically, this confidence interval is biased due to not taking into account the uncertainty of the `standardizer`. No standard error is calculated for this option and NA is returned for "d_se" in the return object.

`ci.level` double vector of length 1 specifying the confidence level. `ci.level` must range from 0 to 1.

`check` logical vector of length 1 specifying whether the input arguments should be checked for errors. For example, checking whether `prepost.nm.list` is a list of length-2 character vectors. This is a tradeoff between computational efficiency (FALSE) and more useful error messages (TRUE).

Details

For each prepost pair of variables, `means_change` calculates the mean change as `data[[ prepost.nm.list[[i]][2] ]] - data[[ prepost.nm.list[[i]][1] ]]` (which corresponds to post - pre) such that increases over time have a positive mean change estimate and decreases over time have a negative mean change estimate. This would be as if the post-timepoint was `x` and the pre-timepoint `y` in `t.test(paired = TRUE)`.

Value

list of data.frames containing statistical information about the mean change for each prepost pair of variables (the rownames of the data.frames are the names of `prepost.nm.list`): 1) nhst = dependent two-samples t-test stat info in a data.frame, 2) desc = descriptive statistics stat info in a data.frame, 3) std = standardized mean difference stat info in a data.frame,

1) nhst = dependent two-samples t-test stat info in a data.frame

est mean change estimate (i.e., post - pre)

se standard error

t t-value

df degrees of freedom

p two-sided p-value

lwr lower bound of the confidence interval

upr upper bound of the confidence interval

2) desc = descriptive statistics stat info in a data.frame

mean_post mean of the post variable

mean_pre mean of the pre variable

sd_post standard deviation of of the post variable

sd_pre standard deviation of the pre variable

n sample size of the change score

r Pearson correlation between the pre and post variables

3) std = standardized mean difference stat info in a data.frame

d_est Cohen's d estimate

d_se Cohen's d standard error

d_lwr Cohen's d lower bound of the confidence interval

d_upr Cohen's d upper bound of the confidence interval

References

Cohen, J. (1988). Statistical power analysis for the behavioral sciences, 2nd ed. Hillsdale, NJ: Erlbaum.

Cumming, G. (2012). Understanding the new statistics: Effect sizes, confidence intervals, and meta-analysis. New York, NY: Rouledge.

Viechtbauer, W. (2007). Approximate confidence intervals for standardized effect sizes in the two-independent and two-dependent samples design. Journal of Educational and Behavioral Statistics, 32(1), 39-60.

See Also

`mean_change` for a single pair of prepost variables, `t.test` fixes the table of contents for some unknown reason, `means_diff` for multiple independent two-sample t-tests, `means_test` for multiple one-sample t-tests,

Examples

```
# dependent two-sample t-tests
prepost_nm_list <- list("first_pair" = c("disp","hp"), "second_pair" = c("carb","gear"))
means_change(mtcars, prepost.nm.list = prepost_nm_list)
means_change(mtcars, prepost.nm.list = prepost_nm_list, d.ci.type = "classic")
means_change(mtcars, prepost.nm.list = prepost_nm_list, standardizer = "change")
means_change(mtcars, prepost.nm.list = prepost_nm_list, ci.level = 0.99)

# same as intercept-only regression with the change score
means_change(data = mtcars, prepost.nm.list = c("disp","hp"))
```

```
lm_obj <- lm(hp - disp ~ 1, data = mtcars)
coef(summary(lm_obj))
```

`means_compare` *Mean differences for multiple variables across 3+ independent groups (one-way ANOVAs)*

Description

`means_compare` compares means across 3+ independent groups with a separate one-way ANOVA for each variable. The function also calculates the descriptive statistics for each group and the variance explained (i.e., R^2 - aka eta^2) by the nominal grouping variable. `means_compare` is simply a wrapper for `oneway.test` plus some extra calculations. `mean_compare` will work with 2 independent groups; however it arguably makes more sense to use `mean_diff` in that case.

Usage

```
means_compare(
  data,
  vrb.nm,
  nom.nm,
  lvl = levels(as.factor(data[[nom.nm]])),
  var.equal = TRUE,
  r2.ci.type = "classic",
  ci.level = 0.95,
  rtn.table = TRUE,
  check = TRUE
)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of length 1 with colnames from `data` specifying the variables.
`nom.nm`	character vector of length 1 with colnames from `data` specifying the nominal variable. It identifies the 3+ groups with 3+ unique values (other than missing values).
`lvl`	character vector with length 3+ specifying the unique values for the 3+ groups. If `nom` is a factor, then `lvl` should be the factor levels rather than the underlying integer codes. This argument allows you to specify the order of the descriptive statistics in the return object, which will be opposite the order of `lvl` for consistency with `mean_diff` and `mean_change`.
`var.equal`	logical vector of length 1 specifying whether the variances of the groups are assumed to be equal (TRUE) or not (FALSE). If TRUE, a traditional one-way ANOVA is computed; if FALSE, Welch's ANOVA is computed. These two tests differ by their denominator degrees of freedoms, F-values, and p-values.

`r2.ci.type` character vector with length 1 specifying the type of confidence intervals to compute for the variance explained (i.e., R^2 or eta^2). There are currently two options: 1) "Fdist" which calculates a non-symmetrical confidence interval based on the non-central F distribution (pg. 38, Smithson, 2003), 2) "classic" which calculates the confidence interval based on a large-sample theory standard error (eq. 3.6.3 in Cohen, Cohen, West, & Aiken, 2003), which is taken from Olkin & Finn (1995) - just above eq. 10. The confidence intervals for R^2-adjusted use the same formula as R^2, but replace R^2 with R^2 adjusted. Technically, the R^2 adjusted confidence intervals can have poor coverage (pg. 54, Smithson, 2003)

`ci.level` numeric vector of length 1 specifying the confidence level. `ci.level` must range from 0 to 1.

`rtn.table` logical vector of length 1 specifying whether the traditional ANOVA tables should be returned as the last element of the return object.

`check` logical vector of length 1 specifying whether the input arguments should be checked for errors. For example, if `vrb.nm` are not colnames within `data`. This is a tradeoff between computational efficiency (FALSE) and more useful error messages (TRUE).

Value

list of data.frames containing statistical information about the mean comparisons for each variable (the rows of the data.frames are `vrb.nm`): 1) nhst = one-way ANOVA stat info in a data.frame, 2) desc = descriptive statistics stat info in a data.frame, 3) std = standardized effect sizes stat info in a data.frame, 4) anova = traditional ANOVA table in a numeric 3D array (only returned if rtn.table = TRUE)

1) nhst = one-way ANOVA stat info in a data.frame

diff_avg average mean difference across group pairs

se NA to remind the user there is no standard error for the average mean difference

F F-value

df_num numerator degrees of freedom

df_den denominator degrees of freedom

p two-sided p-value

2) desc = descriptive statistics stat info in a data.frame (note there could be more than 3 groups - groups i, j, and k are just provided as an example)

mean_'lvl[k '] mean of group k

mean_'lvl[j '] mean of group j

mean_'lvl[i '] mean of group i

sd_'lvl[k '] standard deviation of group k

sd_'lvl[j '] standard deviation of group j

sd_'lvl[i '] standard deviation of group i

n_'lvl[k '] sample size of group k

n_'lvl[j '] sample size of group j

n_'lvl[i '] sample size of group i

3) std = standardized effect sizes stat info in a data.frame

r2_reg_est R^2 estimate

r2_reg_se R^2 standard error (only available if `r2.ci.type` = "classic")

r2_reg_lwr R^2 lower bound of the confidence interval

r2_reg_upr R^2 upper bound of the confidence interval

r2_adj_est R^2-adjusted estimate

r2_adj_se R^2-adjusted standard error (only available if `r2.ci.type` = "classic")

r2_adj_lwr R^2-adjusted lower bound of the confidence interval

r2_adj_upr R^2-adjusted upper bound of the confidence interval

4) anova = traditional ANOVA table in a numeric 3D array (only returned if rtn.table = TRUE).

The dimlabels of the array are "effect" for the rows, "info" for the columns, and "vrb" for the layers. There are two rows with rownames 1. "nom" and 2. "Residuals" where "nom" refers to the between-group effect of the nominal variable and "Residuals" refers to the within-group residual errors. There are 5 columns with colnames 1. "SS" = sum of squares, 2. "df" = degrees of freedom, 3. "MS" = mean squares, 4. "F" = F-value. and 5. "p" = p-value. Note the F-value and p-value will differ from the "nhst" returned vector if `var.equal` = FALSE because the traditional ANOVA table always assumes variances are equal (i.e. `var.equal` = TRUE). There are as many layers as `length(vrb.nm)` with the laynames equal to `vrb.nm`.

References

Cohen, J., Cohen, P., West, A. G., & Aiken, L. S. (2003). Applied Multiple Regression/Correlation Analysis for the Behavioral Science - third edition. New York, NY: Routledge.

Olkin, I., & Finn, J. D. (1995). Correlations redux. Psychological Bulletin, 118(1), 155-164.

Smithson, M. (2003). Confidence intervals. Thousand Oaks, CA: Sage Publications.

See Also

`oneway.test` the workhorse for `means_compare`, `mean_compare` for a single variable across the same 3+ groups, `ci.R2` for confidence intervals of the variance explained, `means_diff` for multiple variables across only 2 groups,

Examples

```
means_compare(mtcars, vrb.nm = c("mpg","wt","qsec"), nom.nm = "gear")
means_compare(mtcars, vrb.nm = c("mpg","wt","qsec"), nom.nm = "gear",
   var.equal = FALSE)
means_compare(mtcars, vrb.nm = c("mpg","wt","qsec"), nom.nm = "gear",
   rtn.table = FALSE)
means_compare(mtcars, vrb.nm = "mpg", nom.nm = "gear")
```

means_diff *Mean differences across two independent groups (independent two-samples t-tests)*

Description

means_diff tests for mean differences across two independent groups with independent two-samples t-tests. The function also calculates the descriptive statistics for each group and the standardized mean differences (i.e., Cohen's d) based on the pooled standard deviations. mean_diff is simply a wrapper for t.test plus some extra calculations.

Usage

```
means_diff(
  data,
  vrb.nm,
  bin.nm,
  lvl = levels(as.factor(data[[bin.nm]])),
  var.equal = TRUE,
  d.ci.type = "unbiased",
  ci.level = 0.95,
  check = TRUE
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames specifying the variables in data to conduct the independent two-sample t-tests for.
bin.nm	character vector of length 1 specifying the binary variable in data. It identifies the two groups with two (and only two) unique values (other than missing values).
lvl	character vector with length 2 specifying the unique values for the two groups. If data[[bin.nm]] is a factor, then lvl should be the factor levels rather than the underlying integer codes. This argument allows you to specify the direction of the mean difference. means_diff calculates the mean differences as data[[vrb.nm]][data[[bin.nm]] == lvl[2],] - data[[vrb.nm]][data[[bin.nm]] == lvl[1],] such that it is group 2 - group 1. By changing which group is group 1 vs. group 2, the direction of the mean difference can be changed. See details.
var.equal	logical vector of length 1 specifying whether the variances of the groups are assumed to be equal (TRUE) or not (FALSE). If TRUE, a traditional independent two-samples t-test is computed; if FALSE, Welch's t-test is computed. These two tests differ by their degrees of freedom and p-values.

d.ci.type — character vector with length 1 specifying the type of confidence intervals to compute for the standardized mean difference (i.e., Cohen's d). There are currently three options: 1) "unbiased" which calculates the unbiased standard error of Cohen's d based on formula 25 in Viechtbauer (2007). A symmetrical confidence interval is then calculated based on the standard error. 2) "tdist" which calculates the confidence intervals based on the t-distribution using the function cohen.d.ci, 3) "classic" which calculates the confidence interval of Cohen's d based on the confidence interval of the mean difference itself. The lower and upper confidence bounds are divided by the pooled standard deviation. Technically, this confidence interval is biased due to not taking into account the uncertainty of the standard deviations. No standard error is calculated for this option and NA is returned for "d_se" in the return object.

ci.level — numeric vector of length 1 specifying the confidence level. ci.level must range from 0 to 1.

check — logical vector of length 1 specifying whether the input arguments should be checked for errors. For example, if data[[bin.nm]] has more than 2 unique values (other than missing values) or if bin.nm is not a colname in data. This is a tradeoff between computational efficiency (FALSE) and more useful error messages (TRUE).

Details

means_diff calculates the mean differences as data[[vrb.nm]][data[[bin.nm]] == lvl[2],] - data[[vrb.nm]][data[[bin.nm]] == lvl[1],] such that it is group 2 - group 1. Group 1 corresponds to the first factor level of data[[bin.nm]] (after being coerced to a factor). Group 2 correspond to the second factor level of data[[bin.nm]] (after being coerced to a factor). This was set up to handle dummy coded treatment variables in a desirable way. For example, if data[[bin.nm]] is a numeric vector with values 0 and 1, the default factor coersion will have the first factor level be "0" and the second factor level "1". This would result will correspond to 1 - 0. However, if the first factor level of data[[bin.nm]] is "treatment" and the second factor level is "control", the result will correspond to control - treatment. If the opposite is desired (e.g., treatment - control), this can be reversed within the function by specifying the lvl argument as c("control","treatment"). Note, means_diff diverts from t.test by calculating the mean difference as group 2 - group 1 (as opposed to the group 1 - group 2 that t.test does). However, group 2 - group 1 is the convention that psych::cohen.d uses as well.

means_diff calculates the pooled standard deviation in a different way than cohen.d. Therefore, the Cohen's d estimates (and confidence intervals if d.ci.type == "tdist") differ from those in cohen.d. means_diff uses the total degrees of freedom in the denomenator while cohen.d uses the total sample size in the denomenator - based on the notation in McGrath & Meyer (2006). However, almost every introduction to statistics textbook uses the total degrees of freedom in the denomenator and that is what makes more sense to me. See examples.

Value

list of data.frames vectors containing statistical information about the mean differences (the rownames of each data.frame are vrb.nm): 1) nhst = independent two-samples t-test stat info in a data.frame, 2) desc = descriptive statistics stat info in a data.frame, 3) std = standardized mean difference stat info in a data.frame

1) nhst = independent two-samples t-test stat info in a data.frame

est mean difference estimate (i.e., group 2 - group 1)

se standard error

t t-value

df degrees of freedom

p two-sided p-value

lwr lower bound of the confidence interval

upr upper bound of the confidence interval

2) desc = descriptive statistics stat info in a data.frame

mean_'lvl[2 '] mean of group 2

mean_'lvl[1 '] mean of group 1

sd_'lvl[2 '] standard deviation of group 2

sd_'lvl[1 '] standard deviation of group 1

n_'lvl[2 '] sample size of group 2

n_'lvl[1 '] sample size of group 1

3) std = standardized mean difference stat info in a data.frame

d_est Cohen's d estimate

d_se Cohen's d standard error

d_lwr Cohen's d lower bound of the confidence interval

d_upr Cohen's d upper bound of the confidence interval

References

McGrath, R. E., & Meyer, G. J. (2006). When effect sizes disagree: the case of r and d. Psychological Methods, 11(4), 386-401.

Viechtbauer, W. (2007). Approximate confidence intervals for standardized effect sizes in the two-independent and two-dependent samples design. Journal of Educational and Behavioral Statistics, 32(1), 39-60.

See Also

`means_diff` for independent two-sample t-test of a single variable, `t.test` the workhorse for `mean_diff`, `cohen.d` for another standardized mean difference function, `means_change` for dependent two-sample t-tests, `means_test` for one-sample t-tests,

Examples

```
# independent two-samples t-tests
means_diff(data = mtcars, vrb.nm = c("mpg","cyl","disp"), bin.nm = "vs")
means_diff(data = mtcars, vrb.nm = c("mpg","cyl","disp"), bin.nm = "vs",
   d.ci.type = "classic")
means_diff(data = mtcars, vrb.nm = c("mpg","cyl","disp"), bin.nm = "vs",
   lvl = c("1","0")) # signs are reversed
means_diff(data = mtcars, vrb.nm = c("mpg","cyl","disp"), bin.nm = "vs",
   lvl = c(1,0)) # can provide numeric levels for dummy variables

# compare to psych::cohen.d()
means_diff(data = mtcars, vrb.nm = c("mpg","cyl","disp"), bin.nm = "vs",
   d.ci.type = "tdist")
tmp_nm <- c("mpg","cyl","disp","vs") # so that Roxygen2 doesn't freak out
cohend_obj <- psych::cohen.d(mtcars[tmp_nm], group = "vs")
as.data.frame(cohend_obj[["cohen.d"]]) # different estimate of cohen's d
   # of course, this also leads to different confidence interval bounds as well

# same as intercept-only regression when var.equal = TRUE
means_diff(data = mtcars, vrb.nm = "mpg", bin.nm = "vs")
lm_obj <- lm(mpg ~ vs, data = mtcars)
coef(summary(lm_obj))

# if levels are not unique values in data[[bin.nm]]
## Not run:
means_diff(data = mtcars, vrb.nm = c("mpg","cyl","disp"), bin.nm = "vs",
   lvl = c("zero", "1")) # an error message is returned
means_diff(data = mtcars, vrb.nm = c("mpg","cyl","disp"), bin.nm = "vs",
   lvl = c("0", "one")) # an error message is returned

## End(Not run)
```

`means_test` *Test for Multiple Sample Means Against Mu (one-sample t-tests)*

Description

`means_test` computes sample means and compares them against specified population `mu` values. These are sometimes referred to as one-sample t-tests. It provides the same results as `t.test`, but provides the confidence intervals for the mean differences from mu rather than the mean itself. The function also calculates the descriptive statistics and the standardized mean differences (i.e., Cohen's d) based on the sample standard deviations.

Usage

```
means_test(
  data,
  vrb.nm,
```

```
  mu = 0,
  d.ci.type = "tdist",
  ci.level = 0.95,
  check = TRUE
)
```

Arguments

`data`	data.frame or data.
`vrb.nm`	character vector of colnames specifying the variables in `data` to conduct the one-sample t-tests for.
`mu`	numeric vector of length = `length(vrb.nm)` or length 1 specifying the population mean values to compare the sample means against. The order of the values should be the same as the order in `vrb.nm`. When length 1, the same population mean value is used for all the variables.
`d.ci.type`	character vector with length 1 of specifying the type of confidence intervals to compute for the standardized mean differences (i.e., Cohen's d). There are currently two options: 1. "tdist" which calculates the confidence intervals based on the t-distribution using the function `cohen.d.ci`. No standard error is calculated for this option and NA is returned for "d_se" in the return object. 2. "classic" which calculates the confidence intervals of Cohen's d based on the confidence interval of the mean difference itself. The lower and upper confidence bounds are divided by the sample standard deviation. Technically, this confidence interval is biased due to not taking into account the uncertainty of the standard deviations. No standard error is calculated for this option and NA is returned for "d_se" in the return object.
`ci.level`	numeric vector of length 1 specifying the confidence level. It must be between 0 and 1.
`check`	logical vector of length 1 specifying whether the input arguments should be checked for errors. For example, checking whether `ci.level` is between 0 and 1. This is a tradeoff between computational efficiency (FALSE) and more useful error messages (TRUE).

Value

list of data.frames containing statistical information about the sample means (the rownames of the data.frames are `vrb.nm`): 1) nhst = one-sample t-test stat info in a data.frame, 2) desc = descriptive statistics stat info in a data.frame, 3) std = standardized mean difference stat info in a data.frame

1) nhst = one-sample t-test stat info in a data.frame

est mean - mu estimate

se standard error

t t-value

df degrees of freedom

p two-sided p-value

lwr lower bound of the confidence interval

upr upper bound of the confidence interval

2) desc = descriptive statistics stat info in a data.frame

mean mean of x

mu population value of comparison

sd standard deviation of x

n sample size of x

3) std = standardized mean difference stat info in a data.frame

d_est Cohen's d estimate

d_se Cohen's d standard error

d_lwr Cohen's d lower bound of the confidence interval

d_upr Cohen's d upper bound of the confidence interval

See Also

`mean_test` one-sample t-test for a single variable, `t.test` same results, `means_diff` independent two-sample t-tests for multiple variables, `means_change` dependent two-sample t-tests for multiple variables,

Examples

```
# one-sample t-tests
means_test(data = attitude, vrb.nm = names(attitude), mu = 50)
means_test(data = attitude, vrb.nm = c("rating","complaints","privileges"),
   mu = c(60, 55, 50))
means_test(data = attitude, vrb.nm = names(attitude), mu = 50, ci.level = 0.90)
means_test(airquality, vrb.nm = names(airquality)) # different df and n due to missing data

# compare to t.test
means_test(data = attitude, vrb.nm = "rating", mu = 50, ci.level = .99)
t.test(attitude$"rating", mu = 50, conf.level = .99)

# same as intercept-only regression
means_test(data = attitude, vrb.nm = "rating")
lm_obj <- lm(rating ~ 1, data = attitude)
coef(summary(lm_obj))
```

mean_change *Mean Change Across Two Timepoints (dependent two-samples t-test)*

Description

mean_change tests for mean change across two timepoints with a dependent two-samples t-test. The function also calculates the descriptive statistics for the timepoints and the standardized mean difference (i.e., Cohen's d) based on either the standard deviation of the pre-timepoint, pooled standard deviation of the pre-timepoint and post-timepoint, or the standard deviation of the change score (post - pre). mean_change is simply a wrapper for t.test plus some extra calculations.

Usage

```
mean_change(
  pre,
  post,
  standardizer = "pre",
  d.ci.type = "unbiased",
  ci.level = 0.95,
  check = TRUE
)
```

Arguments

pre	numeric vector of the variable at the pre-timepoint.
post	numeric vector of the variable at the post-timepoint. The elements must correspond to the same cases in pre as pairs by position. Thus, the length of post must be the same as pre. Note, missing values in post are expected and handled with listwise deletion.
standardizer	chararacter vector of length 1 specifying what to use for standardization when computing the standardized mean difference (i.e., Cohen's d). There are three options: 1. "pre" for the standard deviation of the pre-timepoint, 2. "pooled" for the pooled standard deviation of the pre-timepoint and post-timepoint, 3. "change" for the standard deviation of the change score (post - pre). The default is "pre", which I believe makes the most theoretical sense (see Cumming, 2012); however, "change" is the traditional choice originally proposed by Jacob Cohen (Cohen, 1988).
d.ci.type	character vector of lenth 1 specifying how to compute the confidence interval (and standard error) of the standardized mean difference. There are currently two options: 1. "unbiased" which calculates the unbiased standard error of Cohen's d based on the formulas in Viechtbauer (2007). If standardizer = "pre" or "pooled", then equation 36 from Table 2 is used. If standardizer = "change", then equation 25 from Table 1 is used. A symmetrical confidence interval is then calculated based on the standard error. 2. "classic" which calculates the confidence interval of Cohen's d based on the confidence interval of the mean change itself. The lower and upper confidence bounds are divided

by the standardizer. Technically, this confidence interval is biased due to not taking into account the uncertainty of the standardizer. No standard error is calculated for this option and NA is returned for "d_se" in the return object.

ci.level — double vector of length 1 specifying the confidence level. ci.level must range from 0 to 1.

check — logical vector of length 1 specifying whether the input arguments should be checked for errors. For example, checking whether post is the same length as pre. This is a tradeoff between computational efficiency (FALSE) and more useful error messages (TRUE).

Details

mean_change calculates the mean change as post - pre such that increases over time have a positive mean change estimate and decreases over time have a negative mean change estimate. This would be as if the post-timepoint was x and the pre-timepoint was y in t.test(paired = TRUE).

Value

list of numeric vectors containing statistical information about the mean change: 1) nhst = dependent two-samples t-test stat info in a numeric vector, 2) desc = descriptive statistics stat info in a numeric vector, 3) std = standardized mean difference stat info in a numeric vector

1) nhst = dependent two-samples t-test stat info in a numeric vector

est mean change estimate (i.e., post - pre)

se standard error

t t-value

df degrees of freedom

p two-sided p-value

lwr lower bound of the confidence interval

upr upper bound of the confidence interval

2) desc = descriptive statistics stat info in a numeric vector

mean_post mean of the post variable

mean_pre mean of the pre variable

sd_post standard deviation of of the post variable

sd_pre standard deviation of the pre variable

n sample size of the change score

r Pearson correlation between the pre and post variables

3) std = standardized mean difference stat info in a numeric vector

d_est Cohen's d estimate

d_se Cohen's d standard error

d_lwr Cohen's d lower bound of the confidence interval

d_upr Cohen's d upper bound of the confidence interval

References

Cohen, J. (1988). Statistical power analysis for the behavioral sciences, 2nd ed. Hillsdale, NJ: Erlbaum.

Cumming, G. (2012). Understanding the new statistics: Effect sizes, confidence intervals, and meta-analysis. New York, NY: Rouledge.

Viechtbauer, W. (2007). Approximate confidence intervals for standardized effect sizes in the two-independent and two-dependent samples design. Journal of Educational and Behavioral Statistics, 32(1), 39-60.

See Also

means_change for multiple sets of prepost pairs of variables, t.test the workhorse for mean_change, mean_diff for a independent two-samples t-test, mean_test for a one-sample t-test,

Examples

```
# dependent two-sample t-test
mean_change(pre = mtcars$"disp", post = mtcars$"hp") # standardizer = "pre"
mean_change(pre = mtcars$"disp", post = mtcars$"hp", d.ci.type = "classic")
mean_change(pre = mtcars$"disp", post = mtcars$"hp", standardizer = "pooled")
mean_change(pre = mtcars$"disp", post = mtcars$"hp", ci.level = 0.99)
mean_change(pre = mtcars$"hp", post = mtcars$"disp",
   ci.level = 0.99) # note, when flipping pre and post, the cohen's d estimate
   # changes with standardizer = "pre" because the "pre" variable is different.
   # This does not happen for standardizer = "pooled" or "change". For example...
mean_change(pre = mtcars$"disp", post = mtcars$"hp", standardizer = "pooled")
mean_change(pre = mtcars$"hp", post = mtcars$"disp", standardizer = "pooled")
mean_change(pre = mtcars$"disp", post = mtcars$"hp", standardizer = "change")
mean_change(pre = mtcars$"hp", post = mtcars$"disp", standardizer = "change")

# same as intercept-only regression with the change score
mean_change(pre = mtcars$"disp", post = mtcars$"hp")
lm_obj <- lm(hp - disp ~ 1, data = mtcars)
coef(summary(lm_obj))
```

---

mean_compare *Mean differences for a single variable across 3+ independent groups (one-way ANOVA)*

Description

mean_compare compares means across 3+ independent groups with a one-way ANOVA. The function also calculates the descriptive statistics for each group and the variance explained (i.e., R^2 aka eta^2) by the nominal grouping variable. mean_compare is simply a wrapper for oneway.test plus some extra calculations. mean_compare will work with 2 independent groups; however it arguably makes more sense to use mean_diff in that case.

Usage

```
mean_compare(
  x,
  nom,
  lvl = levels(as.factor(nom)),
  var.equal = TRUE,
  r2.ci.type = "Fdist",
  ci.level = 0.95,
  rtn.table = TRUE,
  check = TRUE
)
```

Arguments

x	numeric vector.
nom	atomic vector (e.g., factor) the same length as x that is a nominal variable. It identifies the 3+ groups with 3+ unique values (other than missing values).
lvl	character vector with length 3+ specifying the unique values for the 3+ groups. If nom is a factor, then lvl should be the factor levels rather than the underlying integer codes. This argument allows you to specify the order of the descriptive statistics in the return object, which will be opposite the order of lvl for consistency with mean_diff and mean_change.
var.equal	logical vector of length 1 specifying whether the variances of the groups are assumed to be equal (TRUE) or not (FALSE). If TRUE, a traditional one-way ANOVA is computed; if FALSE, Welch's ANOVA is computed. These two tests differ by their denominator degrees of freedom, F-value, and p-value.
r2.ci.type	character vector with length 1 specifying the type of confidence intervals to compute for the variance explained (i.e., R^2 aka eta^2). There are currently two options: 1) "Fdist" which calculates a non-symmetrical confidence interval based on the non-central F distribution (pg. 38, Smithson, 2003), 2) "classic" which calculates the confidence interval based on a large-sample theory standard error (eq. 3.6.3 in Cohen, Cohen, West, & Aiken, 2003), which is taken from Olkin & Finn (1995) - just above eq. 10. The confidence intervals for R^2-adjusted use the same formula as R^2, but replace R^2 with R^2 adjusted. Technically, the R^2 adjusted confidence intervals can have poor coverage (pg. 54, Smithson, 2003)
ci.level	numeric vector of length 1 specifying the confidence level. ci.level must range from 0 to 1.
rtn.table	logical vector of length 1 specifying whether the traditional ANOVA table should be returned as the last element of the return object.
check	logical vector of length 1 specifying whether the input arguments should be checked for errors. For example, if nom has length different than the length of x. This is a tradeoff between computational efficiency (FALSE) and more useful error messages (TRUE).

Value

list of numeric vectors containing statistical information about the mean comparison: 1) nhst = one-way ANOVA stat info in a numeric vector, 2) desc = descriptive statistics stat info in a numeric vector, 3) std = standardized effect sizes stat info in a numeric vector, 4) anova = traditional ANOVA table in a numeric matrix (only returned if rtn.table = TRUE).

1) nhst = one-way ANOVA stat info in a numeric vector

diff_avg average mean difference across group pairs

se NA to remind the user there is no standard error for the average mean difference

F F-value

df_num numerator degrees of freedom

df_den denominator degrees of freedom

p two-sided p-value

2) desc = descriptive statistics stat info in a numeric vector (note there could be more than 3 groups - groups i, j, and k are just provided as an example)

mean_'lvl[k '] mean of group k

mean_'lvl[j '] mean of group j

mean_'lvl[i '] mean of group i

sd_'lvl[k '] standard deviation of group k

sd_'lvl[j '] standard deviation of group j

sd_'lvl[i '] standard deviation of group i

n_'lvl[k '] sample size of group k

n_'lvl[j '] sample size of group j

n_'lvl[i '] sample size of group i

3) std = standardized effect sizes stat info in a numeric vector

r2_reg_est R^2 estimate

r2_reg_se R^2 standard error (only available if `r2.ci.type` = "classic")

r2_reg_lwr R^2 lower bound of the confidence interval

r2_reg_upr R^2 upper bound of the confidence interval

r2_adj_est R^2-adjusted estimate

r2_adj_se R^2-adjusted standard error (only available if `r2.ci.type` = "classic")

r2_adj_lwr R^2-adjusted lower bound of the confidence interval

r2_adj_upr R^2-adjusted upper bound of the confidence interval

4) anova = traditional ANOVA table in a numeric matrix (only returned if rtn.table = TRUE).

The dimlabels of the matrix was "effect" for the rows and "info" for the columns. There are two rows with rownames 1. "nom" and 2. "Residuals" where "nom" refers to the between-group effect of the nominal variable and "Residuals" refers to the within-group residual errors. There are 5 columns with colnames 1. "SS" = sum of squares, 2. "df" = degrees of freedom, 3. "MS" = mean squares, 4. "F" = F-value. and 5. "p" = p-value. Note the F-value and p-value will differ from the "nhst" returned vector if `var.equal` = FALSE because the traditional ANOVA table always assumes variances are equal (i.e. `var.equal` = TRUE).

References

Cohen, J., Cohen, P., West, A. G., & Aiken, L. S. (2003). Applied Multiple Regression/Correlation Analysis for the Behavioral Science - third edition. New York, NY: Routledge.

Olkin, I., & Finn, J. D. (1995). Correlations redux. Psychological Bulletin, 118(1), 155-164.

Smithson, M. (2003). Confidence intervals. Thousand Oaks, CA: Sage Publications.

See Also

`oneway.test` the workhorse for `mean_compare`, `means_compare` for multiple variables across the same 3+ groups, `ci.R2` for confidence intervals of the variance explained, `mean_diff` for a single variable across only 2 groups,

Examples

```
mean_compare(x = mtcars$"mpg", nom = mtcars$"gear")
mean_compare(x = mtcars$"mpg", nom = mtcars$"gear", var.equal = FALSE)
mean_compare(x = mtcars$"mpg", nom = mtcars$"gear", rtn.table = FALSE)
mean_compare(x = mtcars$"mpg", nom = mtcars$"gear", r2.ci.type = "classic")
```

`mean_diff` *Mean difference across two independent groups (independent two-samples t-test)*

Description

`mean_diff` tests for mean differences across two independent groups with an independent two-samples t-test. The function also calculates the descriptive statistics for each group and the standardized mean difference (i.e., Cohen's d) based on the pooled standard deviation. `mean_diff` is simply a wrapper for `t.test` plus some extra calculations.

Usage

```
mean_diff(
  x,
  bin,
  lvl = levels(as.factor(bin)),
  var.equal = TRUE,
  d.ci.type = "unbiased",
  ci.level = 0.95,
  check = TRUE
)
```

Arguments

x	numeric vector.
bin	atomic vector (e.g., factor) the same length as x that is a binary variable. It identifies the two groups with two (and only two) unique values (other than missing values).
lvl	character vector with length 2 specifying the unique values for the two groups. If bin is a factor, then lvl should be the factor levels rather than the underlying integer codes. This argument allows you to specify the direction of the mean difference. mean_diff calculates the mean difference as x[bin == lvl[2]] - x[bin == lvl[1]] such that it is group 2 - group 1. By changing which group is group 1 vs. group 2, the direction of the mean difference can be changed. See details.
var.equal	logical vector of length 1 specifying whether the variances of the groups are assumed to be equal (TRUE) or not (FALSE). If TRUE, a traditional independent two-samples t-test is computed; if FALSE, Welch's t-test is computed. These two tests differ by their degrees of freedom and p-values.
d.ci.type	character vector with length 1 of specifying the type of confidence intervals to compute for the standardized mean difference (i.e., Cohen's d). There are currently three options: 1) "unbiased" which calculates the unbiased standard error of Cohen's d based on formula 25 in Viechtbauer (2007). A symmetrical confidence interval is then calculated based on the standard error. 2) "tdist" which calculates the confidence intervals based on the t-distribution using the function cohen.d.ci, 3) "classic" which calculates the confidence interval of Cohen's d based on the confidence interval of the mean difference itself. The lower and upper confidence bounds are divided by the pooled standard deviation. Technically, this confidence interval is biased due to not taking into account the uncertainty of the standard deviations. No standard error is calculated for this option and NA is returned for "d_se" in the return object.
ci.level	numeric vector of length 1 specifying the confidence level. ci.level must range from 0 to 1.
check	logical vector of length 1 specifying whether the input arguments should be checked for errors. For example, if bin has more than 2 unique values (other than missing values) or if bin has length different than the length of x. This is a tradeoff between computational efficiency (FALSE) and more useful error messages (TRUE).

Details

mean_diff calculates the mean difference as x[bin == lvl[2]] - x[bin == lvl[1]] such that it is group 2 - group 1. Group 1 corresponds to the first factor level of bin (after being coerced to a factor). Group 2 correspond to the second factor level bin (after being coerced to a factor). This was set up to handle dummy coded treatment variables in a desirable way. For example, if bin is a numeric vector with values 0 and 1, the default factor coersion will have the first factor level be "0" and the second factor level "1". This would result will correspond to 1 - 0. However, if the first factor level of bin is "treatment" and the second factor level is "control", the result will correspond to control - treatment. If the opposite is desired (e.g., treatment - control), this can

be reversed within the function by specifying the `lvl` argument as `c("control","treatment")`. Note, `mean_diff` diverts from `t.test` by calculating the mean difference as group 2 - group 1 (as opposed to the group 1 - group 2 that `t.test` does). However, group 2 - group 1 is the convention that `psych::cohen.d` uses as well.

`mean_diff` calculates the pooled standard deviation in a different way than `cohen.d`. Therefore, the Cohen's d estimates (and confidence intervals if d.ci.type == "tdist") differ from those in `cohen.d`. `mean_diff` uses the total degrees of freedom in the denomenator while `cohen.d` uses the total sample size in the denomenator - based on the notation in McGrath & Meyer (2006). However, almost every introduction to statistics textbook uses the total degrees of freedom in the denomenator and that is what makes more sense to me. See examples.

Value

list of numeric vectors containing statistical information about the mean difference: 1) nhst = independent two-samples t-test stat info in a numeric vector, 2) desc = descriptive statistics stat info in a numeric vector, 3) std = standardized mean difference stat info in a numeric vector

1) nhst = independent two-samples t-test stat info in a numeric vector

est mean difference estimate (i.e., group 2 - group 1)

se standard error

t t-value

df degrees of freedom

p two-sided p-value

lwr lower bound of the confidence interval

upr upper bound of the confidence interval

2) desc = descriptive statistics stat info in a numeric vector

mean_'lvl[2 '] mean of group 2

mean_'lvl[1 '] mean of group 1

sd_'lvl[2 '] standard deviation of group 2

sd_'lvl[1 '] standard deviation of group 1

n_'lvl[2 '] sample size of group 2

n_'lvl[1 '] sample size of group 1

3) std = standardized mean difference stat info in a numeric vector

d_est Cohen's d estimate

d_se Cohen's d standard error

d_lwr Cohen's d lower bound of the confidence interval

d_upr Cohen's d upper bound of the confidence interval

References

McGrath, R. E., & Meyer, G. J. (2006). When effect sizes disagree: the case of r and d. Psychological Methods, 11(4), 386-401.

Viechtbauer, W. (2007). Approximate confidence intervals for standardized effect sizes in the two-independent and two-dependent samples design. Journal of Educational and Behavioral Statistics, 32(1), 39-60.

See Also

`t.test` the workhorse for `mean_diff`, `means_diff` for multiple variables across the same two groups, `cohen.d` for another standardized mean difference function, `mean_change` for dependent two-sample t-test, `mean_test` for one-sample t-test,

Examples

```
# independent two-samples t-test
mean_diff(x = mtcars$"mpg", bin = mtcars$"vs")
mean_diff(x = mtcars$"mpg", bin = mtcars$"vs", lvl = c("1","0"))
mean_diff(x = mtcars$"mpg", bin = mtcars$"vs", lvl = c(1, 0)) # levels don't have to be character
mean_diff(x = mtcars$"mpg", bin = mtcars$"vs", d.ci.type = "classic")

# compare to psych::cohen.d()
mean_diff(x = mtcars$"mpg", bin = mtcars$"vs", d.ci.type = "tdist")
tmp_nm <- c("mpg","vs") # because otherwise Roxygen2 gets upset
cohend_obj <- psych::cohen.d(mtcars[tmp_nm], group = "vs")
as.data.frame(cohend_obj[["cohen.d"]]) # different estimate of cohen's d
   # of course, this also leads to different confidence interval bounds as well

# same as intercept-only regression when var.equal = TRUE
mean_diff(x = mtcars$"mpg", bin = mtcars$"vs", d.ci.type = "tdist")
lm_obj <- lm(mpg ~ vs, data = mtcars)
coef(summary(lm_obj))

# errors
## Not run:
mean_diff(x = mtcars$"mpg",
   bin = attitude$"ratings") # `bin` has length different than `x`
mean_diff(x = mtcars$"mpg",
  bin = mtcars$"gear") # `bin` has more than two unique values (other than missing values)

## End(Not run)
```

`mean_if` *Mean Conditional on Minimum Frequency of Observed Values*

Description

mean_if calculates the mean of a numeric or logical vector conditional on a specified minimum frequency of observed values. If the frequency of observed values is less than (or equal to) ov.min, then NA is returned rather than the mean.

Usage

```
mean_if(x, trim = 0, ov.min = 1, prop = TRUE, inclusive = TRUE)
```

Arguments

x	numeric or logical vector.
trim	numeric vector of length 1 specifying the proportion of values from each end of x to trim. Trimmed values are recoded to their endpoint for calculation of the mean. See mean.default.
ov.min	minimum frequency of observed values required. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and length(x).
prop	logical vector of length 1 specifying whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
inclusive	logical vector of length 1 specifying whether the mean should be calculated if the frequency of observed values is exactly equal to ov.min.

Value

numeric vector of length 1 providing the mean of x or NA conditional on if the frequency of observed data is greater than (or equal to) ov.min.

See Also

mean.default sum_if make.fun_if

Examples

```
mean_if(x = airquality[[1]], ov.min = .75) # proportion of observed values
mean_if(x = airquality[[1]], ov.min = 116,
   prop = FALSE) # count of observe values
mean_if(x = airquality[[1]], ov.min = 116, prop = FALSE,
   inclusive = FALSE) # not include ov.min value itself
mean_if(x = c(TRUE, NA, FALSE, NA),
   ov.min = .50) # works with logical vectors as well as numeric
```

mean_test *Test for Sample Mean Against Mu (one-sample t-test)*

Description

mean_test computes the sample mean and compares it against a specified population mu value. This is sometimes referred to as a one-sample t-test. It provides the same results as t.test, but provides the confidence interval for the mean difference from mu rather than the mean itself. The function also calculates the descriptive statistics and the standardized mean difference (i.e., Cohen's d) based on the sample standard deviation.

Usage

```
mean_test(x, mu = 0, d.ci.type = "tdist", ci.level = 0.95, check = TRUE)
```

Arguments

x	numeric vector.
mu	numeric vector of length 1 specifying the population mean value to compare the sample mean against.
d.ci.type	character vector with length 1 specifying the type of confidence interval to compute for the standardized mean difference (i.e., Cohen's d). There are currently two options: 1. "tdist" which calculates the confidence intervals based on the t-distribution using the function cohen.d.ci. No standard error is calculated for this option and NA is returned for "d_se" in the return object. 2. "classic" which calculates the confidence interval of Cohen's d based on the confidence interval of the mean difference itself. The lower and upper confidence bounds are divided by the sample standard deviation. Technically, this confidence interval is biased due to not taking into account the uncertainty of the standard deviations. No standard error is calculated for this option and NA is returned for "d_se" in the return object.
ci.level	numeric vector of length 1 specifying the confidence level. It must be between 0 and 1.
check	logical vector of length 1 specifying whether the input arguments should be checked for errors. For example, checking whether x is a numeric vector. This is a tradeoff between computational efficiency (FALSE) and more useful error messages (TRUE).

Value

list of numeric vectors containing statistical information about the sample mean: 1) nhst = one-sample t-test stat info in a numeric vector, 2) desc = descriptive statistics stat info in a numeric vector, 3) std = standardized mean difference stat info in a numeric vector

1) nhst = one-sample t-test stat info in a numeric vector

est mean - mu estimate

se standard error

t t-value

df degrees of freedom

p two-sided p-value

lwr lower bound of the confidence interval

upr upper bound of the confidence interval

2) desc = descriptive statistics stat info in a numeric vector

mean mean of x

mu population value of comparison

sd standard deviation of x

n sample size of x

3) std = standardized mean difference stat info in a numeric vector

d_est Cohen's d estimate

d_se Cohen's d standard error

d_lwr Cohen's d lower bound of the confidence interval

d_upr Cohen's d upper bound of the confidence interval

See Also

means_test one-sample t-tests for multiple variables, t.test same results, mean_diff independent two-sample t-test, mean_change dependent two-sample t-test,

Examples

```
# one-sample t-test
mean_test(x = mtcars$"mpg")
mean_test(x = attitude$"rating", mu = 50)
mean_test(x = attitude$"rating", mu = 50, d.ci.type = "classic")

# compare to t.test()
mean_test(x = attitude$"rating", mu = 50, ci.level = .99)
t.test(attitude$"rating", mu = 50, conf.level = .99)

# same as intercept-only regression when mu = 0
mean_test(x = mtcars$"mpg")
lm_obj <- lm(mpg ~ 1, data = mtcars)
coef(summary(lm_obj))
```

mode2 *Statistical Mode of a Numeric Vector*

Description

mode2 calculates the statistical mode - a measure of central tendancy - of a numeric vector. This is in contrast to mode in base R, which returns the storage mode of an object. In the case multiple modes exist, the multiple argument allows the user to specify if they want the multiple modes returned or just one.

Usage

```
mode2(x, na.rm = FALSE, multiple = FALSE)
```

Arguments

x	atomic vector
na.rm	logical vector of length 1 specifying if missing values should be removed from x before calculating its frequencies.
multiple	logical vector of length 1 specifying if multiple modes should be returned in the case they exist. If multiple modes exist and multiple = TRUE, the multiple modes will be returned in alphanumeric order. If multiple modes exist and multiple = TRUE, the first mode in alphanumeric order will be returned. Note, NA is always last in the alphanumeric order. If only one mode exists, then the multiple argument is not used.

Value

atomic vector of the same storage mode as x providing the statistical mode(s).

See Also

freq table

Examples

```
# ONE MODE
vec <- c(7,8,9,7,8,9,9)
mode2(vec)
mode2(vec, multiple = TRUE)

# TWO MODES
vec <- c(7,8,9,7,8,9,8,9)
mode2(vec)
mode2(vec, multiple = TRUE)

# WITH NA
vec <- c(7,8,9,7,8,9,NA,9)
```

```
mode2(vec)
mode2(vec, na.rm = TRUE)
vec <- c(7,8,9,7,8,9,NA,9,NA,NA)
mode2(vec)
mode2(vec, multiple = TRUE)
```

---

ncases *Number of Cases in Data*

Description

ncases counts how many cases in a data.frame there are that have a specified frequency of observed values across a set of columns. This function is similar to nrow and is essentially partial.cases + sum. The user can have ncases return the number of complete cases by calling ov.min = 1, prop = TRUE, and inclusive = TRUE (the default).

Usage

```
ncases(data, vrb.nm = names(data), ov.min = 1, prop = TRUE, inclusive = TRUE)
```

Arguments

data	data.frame or matrix of data.
vrb.nm	a character vector of colnames from data specifying the variables.
ov.min	minimum frequency of observed values required per row. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and length(vrb.nm).
prop	logical vector of length 1 specifying whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
inclusive	logical vector of length 1 specifying whether the case should be included if the frequency of observed values in a row is exactly equal to ov.min.

Value

integer vector of length 1 providing the nrow in data with the given amount of observed values.

See Also

partial.cases nrow

Examples

```
vrb_nm <- c("Ozone","Solar.R","Wind")
nrow(airquality[vrb_nm]) # number of cases regardless of missing data
sum(complete.cases(airquality[vrb_nm])) # number of complete cases
ncases(data = airquality, vrb.nm = c("Ozone","Solar.R","Wind"),
   ov.min = 2/3) # number of rows with at least 2 of the 3 variables observed
```

ncases_by *Number of Cases in Data by Group*

Description

ncases_by computes the ncases of a data.frame by group. Through the use of the ov.min, prop, and inclusive arguments, the user can specify how many missing values are allowed in a row for it to be counted. ncases_by is simply a wrapper for ncases + agg_dfm.

Usage

```
ncases_by(
  data,
  vrb.nm = str2str::pick(names(data), val = grp.nm, not = TRUE),
  grp.nm,
  sep = ".",
  ov.min = 1L,
  prop = TRUE,
  inclusive = TRUE
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the set of variables to base the ncases on.
grp.nm	character vector of colnames from data specifying the grouping variables.
sep	character vector of length 1 specifying what string to use to separate the groups when naming the return object. sep is only used if grp.nm has length > 1 (aka multiple grouping variables)
ov.min	minimum frequency of observed values required per row. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and length(vrb.nm).
prop	logical vector of length 1 specifying whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
inclusive	logical vector of length 1 specifying whether the case should be included if the frequency of observed values in a row is exactly equal to ov.min.

Value

atomic vector with names = unique(interaction(data[grp.nm], sep = sep)) and length = length(unique(interactio sep = sep))) providing the ncases for each group.

See Also

nrow_by ncases agg_dfm

Examples

```
# one grouping variables
tmp_nm <- c("outcome","case","session","trt_time")
dat <- as.data.frame(lmeInfo::Bryant2016)[tmp_nm]
stats_by <- psych::statsBy(dat,
   group = "case") # requires you to include "case" column in dat
ncases_by(data = dat, grp.nm = "case")
dat2 <- as.data.frame(ChickWeight)
ncases_by(data = dat2, grp.nm = "Chick")

# two grouping variables
tmp <- reshape(psych::bfi[1:10, ], varying = 1:25, timevar = "item",
   ids = row.names(psych::bfi)[1:10], direction = "long", sep = "")
tmp_nm <- c("id","item","N","E","C","A","O") # Roxygen runs the whole script
dat3 <- str2str::stack2(tmp[tmp_nm], select.nm = c("N","E","C","A","O"),
   keep.nm = c("id","item"))
ncases_by(dat3, grp.nm = c("id","vrb_names"))
```

---

ncases_desc *Describe Number of Cases in Data by Group*

Description

ncases_desc computes descriptive statistics about the number of cases by group in a data.frame. This is often done in diary studies to obtain information about compliance for the sample. Through the use of the ov.min, prop, and inclusive arguments, the user can specify how many missing values are allowed in a row for it to be counted. ncases_desc is simply ncases_by + psych::describe.

Usage

```
ncases_desc(
  data,
  vrb.nm = str2str::pick(names(data), val = grp.nm, not = TRUE),
  grp.nm,
  ov.min = 1,
  prop = TRUE,
  inclusive = TRUE,
  interp = FALSE,
  skew = TRUE,
  ranges = TRUE,
  trim = 0.1,
  type = 3,
  quant = c(0.25, 0.75),
  IQR = FALSE
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the set of variables to base the ncases on.
grp.nm	character vector of colnames from data specifying the grouping variables.
ov.min	minimum frequency of observed values required per row. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and length(vrb.nm).
prop	logical vector of length 1 specifying whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
inclusive	logical vector of length 1 specifying whether the case should be included if the frequency of observed values in a row is exactly equal to ov.min.
interp	logical vector of length 1 specifying whether the median should be standard (FALSE) or interpolated (TRUE).
skew	logical vector of length 1 specifying whether skewness and kurtosis should be calculated (TRUE) or not (FALSE).
ranges	logical vector of length 1 specifying whether the minimum, maximum, and range (i.e., maximum - minimum) should be calculated (TRUE) or not (FALSE). Note, if ranges = FALSE, the trimmed mean and median absolute deviation is also not computed as per the psych::describe function behavior.
trim	numeric vector of length 1 specifying the top and bottom quantiles of data that are to be excluded when calculating the trimmed mean. For example, the default value of 0.1 means that only data within the 10th - 90th quantiles are used for calculating the trimmed mean.
type	numeric vector of length 1 specifying the type of skewness and kurtosis coefficients to compute. See the details of psych::describe. The options are 1, 2, or 3.
quant	numeric vector specifying the quantiles to compute. Foe example, the default value of c(0.25, 0.75) computes the 25th and 75th quantiles of the group number of cases. If quant = NULL, then no quantiles are returned.
IQR	logical vector of length 1 specifying whether to compute the Interquartile Range (TRUE) or not (FALSE), which is simply the 75th quantil - 25th quantile.

Value

numeric vector containing descriptive statistics about number of cases by group. Note, which elements are returned depends on the arguments. See each argument's description.

n number of groups

mean mean

sd standard deviation

median median (standard if interp = FALSE, interpolated if interp = TRUE)

trimmed trimmed mean based on trim

mad median absolute difference

min minimum

max maximum

range maximum - minumum

skew skewness

kurtosis kurtosis

se standard error of the mean

IQR 75th quantile - 25th quantile

QX.XX quantiles, which are named by quant (e.g., 0.25 = "Q0.25")

See Also

ncases_by describe

Examples

```
tmp_nm <- c("outcome","case","session","trt_time")
dat <- as.data.frame(lmeInfo::Bryant2016)[tmp_nm]
stats_by <- psych::statsBy(dat, group = "case") # doesn't include everything you want
ncases_desc(data = dat, grp.nm = "case")
dat2 <- as.data.frame(ChickWeight)
ncases_desc(data = dat2, grp.nm = "Chick")
ncases_desc(data = dat2, grp.nm = "Chick", trim = .05)
ncases_desc(data = dat2, grp.nm = "Chick", ranges = FALSE)
ncases_desc(data = dat2, grp.nm = "Chick", quant = NULL)
ncases_desc(data = dat2, grp.nm = "Chick", IQR = TRUE)
```

---

ncases_ml *Multilevel Number of Cases*

Description

ncases_ml computes the number cases and number of groups in the data that are at least partially observed, given a specified frequency of observed values across a set of columns. ncases_ml allows the user to specify the frequency of columns that need to be observed in order to count the case. Groups can be excluded if no rows in the data for a group have enough observed values to be counted as cases. This is simply a combination of partial.cases + nrow_ml. Note, ncases_ml is essentially a version of nrow_ml that accounts for missing data.

Usage

```
ncases_ml(
  data,
  vrb.nm = str2str::pick(names(data), val = grp.nm, not = TRUE),
  grp.nm,
  ov.min = 1L,
```

```
  prop = TRUE,
  inclusive = TRUE
)
```

Arguments

data	data.frame of data.
vrb.nm	a character vector of colnames from data specifying the variables which will be used to determine the partially observed cases.
grp.nm	character vector of colnames from data specifying the grouping variables.
ov.min	minimum frequency of observed values required per row. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and length(vrb.nm).
prop	logical vector of length 1 specifying whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
inclusive	logical vector of length 1 specifying whether the case should be included if the frequency of observed values in a row is exactly equal to ov.min.

Value

list with two elements providing the sample sizes (accouning for missing data). The first element is named "within" and contains the number of cases in the data. The second element is named "between" and contains the number of groups in the data. Cases are counted if if the frequency of observed values is greater than (or equal to, if inclusive = TRUE).

See Also

nrow_ml ncases_by partial.cases

Examples

```
# NO MISSING DATA

# one grouping variable
ncases_ml(data = as.data.frame(ChickWeight), grp.nm = "Chick")

# multiple grouping variables
ncases_ml(data = mtcars, grp.nm = c("vs","am"))

# YES MISSING DATA

# only within
nrow_ml(data = airquality, grp.nm = "Month")
ncases_ml(data = airquality, grp.nm = "Month")

# both within and between
airquality2 <- airquality
airquality2[airquality2$"Month" == 6, "Ozone"] <- NA
nrow_ml(data = airquality2, grp.nm = "Month")
```

```
ncases_ml(data = airquality2, grp.nm = "Month")
```

---

ngrp *Number of Groups in Data*

Description

ngrp computes the number of groups in data given one or more grouping variables. This is simply a combination of unique.data.frame + nrow.

Usage

```
ngrp(data, grp.nm)
```

Arguments

data	data.frame of data.
grp.nm	character vector of colnames from data specifying the grouping variables.

Value

integer vector of length 1 specifying the number of groups.

See Also

nrow_ml ncases_ml nrow_by ncases_by

Examples

```
# one grouping variable
Orthodont2 <- as.data.frame(nlme::Orthodont)
ngrp(Orthodont2, grp.nm = "Subject")
length(unique(Orthodont2$"Subject"))

# two grouping variable
co2 <- as.data.frame(CO2)
ngrp(co2, grp.nm = c("Plant"))
grp_nm <- c("Type","Treatment")
ngrp(co2, grp.nm = grp_nm)
unique.data.frame(co2[grp_nm])

#TODO: how does it handle factor levels with no cases?
```

nhst *Null Hypothesis Significance Testing*

Description

nhst computes the statistical information for null hypothesis significance testing (NHST), t-values, p-values, etc., from parameter estimates, standard errors, and degrees of freedom. If degrees of freedom are not applicable or available, then df can be set to Inf (the default) and z-values rather than t-values will be computed.

Usage

```
nhst(est, se, df = Inf, ci.level = 0.95, p.value = "two.sided")
```

Arguments

est	numeric vector of parameter estimates.
se	numeric vector of standard errors. Must be the same length as est.
df	numeric vector of degrees of freedom. Must be length of 1 or have same length as est and se. If degrees of freedom are not applicable or available, then df can be set to Inf (the default) and z-values rather than t-values will be computed. Note, df can be non-integers with decimals.
ci.level	double vector of length 1 specifying the confidence level. Must be between 0 and 1 - or can be NULL in which case no confidence intervals are computed and the return object does not have the columns "lwr" or "upr".
p.value	character vector of length 1 specifying the type of p-values to compute. The options are 1) "two.sided" which computed non-directional, two-tailed p-values, 2) "less", which computes negative-directional, one-tailed p-values, or 3) "greater", which computes positive-directional, one-tailed p-values.

Value

data.frame with nrow equal to the lengths of est and se. The rownames are taken from est, unless est does not have any names and then the rownames are taken from the names of se. If neither have names, then the rownames are automatic (i.e., 1:nrow()). The columns are the following:

est parameter estimates

se standard errors

t t-values (z-values if df = Inf)

df degrees of freedom

p p-values

lwr lower bound of the confidence intervals (excluded if ci.level = NULL)

upr upper bound of the confidence intervals (excluded if ci.level = NULL)

See Also

confint2.default

Examples

```
est <- colMeans(attitude)
se <- apply(X = str2str::d2m(attitude), MARGIN = 2, FUN = function(vec)
   sqrt(var(vec) / length(vec)))
df <- nrow(attitude) - 1
nhst(est = est, se = se, df = df)
nhst(est = est, se = se) # default is df = Inf resulting in z-values
nhst(est = est, se = se, df = df, ci.level = NULL) # no "lwr" or "upr" columns
nhst(est = est, se = se, df = df, ci.level = 0.99)
```

---

nom2dum *Nominal Variable to Dummy Variables*

Description

nom2dum converts a nominal variable into a set of dummy variables. There is one dummy variable for each unique value in the nominal variable. Note, base R does this recoding internally through the model.matrix.default function, but it is used in the context of regression-like models and it is not clear how to simplify it for general use cases outside that context.

Usage

```
nom2dum(nom, yes = 1L, no = 0L, prefix = "", rtn.fct = FALSE)
```

Arguments

nom	character vector (or any atomic vector, including factors, which will be then coerced to a character vector) specifying the nominal variable.
yes	atomic vector of length 1 specifying what unique value should represent rows when the nominal category of interest is present. For a traditional dummy variable this value would be 1.
no	atomic vector of length 1 specifying what unique value should represent rows when the nominal category of interest is absent. For a traditional dummy variable this value would be 0.
prefix	character vector of length 1 specifying the string that should be appended to the beginning of each colname in the return object.
rtn.fct	logical vector of length 1 specifying whether the columns of the return object should be factors where the first level is no and the second level is yes.

Details

Note, that yes and no are assumed to be the same typeof. If they are not, then the columns in the return object will be coerced to the most complex typeof (i.e., most to least: character, double, integer, logical).

Value

data.frame of dummy columns with colnames specified by paste0(prefix, unique(nom)) and rownames specified by names(nom) or default data.frame rownames (i.e., c("1","2","3", etc.) if names(nom) is NULL.

See Also

model.matrix.default dum2nom

Examples

```
nom2dum(infert$"education") # default
nom2dum(infert$"education", prefix = "edu_") # use of the `prefix` argument
nom2dum(nom = infert$"education", yes = "one", no = "zero",
   rtn.fct = TRUE) # returns factor columns
```

---

nrow_by *Number of Rows in Data by Group*

Description

nrow_by computes the nrow of a data.frame by group. nrow_by is simply a wrapper for nrow + agg_dfm.

Usage

```
nrow_by(data, grp.nm, sep = ".")
```

Arguments

data	data.frame of data.
grp.nm	character vector of colnames from data specifying the grouping variables.
sep	character vector of length 1 specifying what string to use to separate the groups when naming the return object. sep is only used if grp.nm has length > 1 (aka multiple grouping variables)

Value

atomic vector with names = unique(interaction(data[grp.nm], sep = sep)) and length = length(unique(interactio
sep = sep))) providing the nrow for each group.

See Also

ncases_by nrow agg_dfm

Examples

```
# one grouping variables
tmp_nm <- c("outcome","case","session","trt_time")
dat <- as.data.frame(lmeInfo::Bryant2016)[tmp_nm]
stats_by <- psych::statsBy(dat,
   group = "case") # requires you to include "case" column in dat
nrow_by(data = dat, grp.nm = "case")
dat2 <- as.data.frame(ChickWeight)
nrow_by(data = dat2, grp.nm = "Chick")

# two grouping variables
tmp <- reshape(psych::bfi[1:10, ], varying = 1:25, timevar = "item",
   ids = row.names(psych::bfi)[1:10], direction = "long", sep = "")
tmp_nm <- c("id","item","N","E","C","A","O") # Roxygen runs the whole script
dat3 <- str2str::stack2(tmp[tmp_nm], select.nm = c("N","E","C","A","O"),
   keep.nm = c("id","item"))
nrow_by(dat3, grp.nm = c("id","vrb_names"))
```

---

nrow_ml *Multilevel Number of Rows*

Description

nrow_ml computes the number rows in the data as well as the number of groups in the data. This corresponds to the within-group sample size and between-group sample size (ignoring any missing data). This is simply a combination of nrow + ngrp.

Usage

```
nrow_ml(data, grp.nm)
```

Arguments

data	data.frame of data.
grp.nm	character vector of colnames from data specifying the grouping variables.

Value

list with two elements providing the sample sizes (ignoring missing data). The first element is named "within" and contains the number of rows in the data. The second element is named "between" and contains the number of groups in the data.

See Also

ncases_ml nrow_by ncases_by ngrp

Examples

```
# one grouping variable
nrow_ml(data = as.data.frame(ChickWeight), grp.nm = "Chick")

# multiple grouping variables
nrow_ml(data = mtcars, grp.nm = c("vs","am"))
```

---

n_compare *Test for Equal Frequency of Values (chi-square test of goodness of fit)*

Description

n_compare tests whether all the values for a variable have equal frequency with a chi-square test of goodness of fit. n_compare does not currently allow for user-specified unequal frequencies of values; this is possible with chisq.test. The function also calculates the counts and overall percentages for the value frequencies. prop_test is simply a wrapper for chisq.test plus some extra calculations.

Usage

```
n_compare(x, simulate.p.value = FALSE, B = 2000)
```

Arguments

x	atomic vector. Probably makes sense to contain relatively few unique values.
simulate.p.value	logial vector of length 1 specifying whether the p-value should be based on a Monte Carlo simulation rather than the classic formula. See chisq.test for details.
B	integer vector of length 1 specifying how much Monte Carlo simulations run. Only used if simulate.p.value = TRUE. See chisq.test for details.

Value

list of numeric vectors containing statistical information about the frequency comparison: 1) nhst = chi-square test of goodness of fit stat info in a numeric vector, 2) count = numeric vector of length 3 with table of counts, 3) percent = numeric vector of length 3 with table of overall percentages

1) nhst = chi-square test of goodness of fit stat info in a numeric vector

diff_avg average difference in subsample sizes (i.e., |ni - nj|)

se NA (to remind the user there is no standard error for the test)

X2 chi-square value

df degrees of freedom (# of unique values = 1)

p two-sided p-value

2) count = numeric vector of length 3 with table of counts with an additional element for the total. The names are 1. "n_‘lvl[k]‘", 2. "n_‘lvl[j]‘", 3. "n_‘lvl[i]‘", ..., X = "total"

3) percent = numeric vector of length 3 with table of overall percentages with an additional element for the total. The names are 1. "n_‘lvl[k]‘", 2. "n_‘lvl[j]‘", 3. "n_‘lvl[i]‘", ..., X = "total"

See Also

chisq.test the workhorse for n_compare, props_test for multiple dummy variables, prop_diff for chi-square test of independence,

Examples

```
n_compare(mtcars$"cyl")
n_compare(mtcars$"gear")
n_compare(mtcars$"cyl", simulate.p.value = TRUE)

# compare to chisq.test()
n_compare(mtcars$"cyl")
chisq.test(table(mtcars$"cyl"))
```

---

partial.cases *Find Partial Cases*

Description

partial.cases indicates which cases are at least partially observed, given a specified frequency of observed values across a set of columns. This function builds off complete.cases. While complete.cases requires completely observed cases, partial.cases allows the user to specify the frequency of columns required to be observed. The default arguments are equal to complete.cases.

Usage

```
partial.cases(data, vrb.nm, ov.min = 1, prop = TRUE, inclusive = TRUE)
```

Arguments

data	data.frame or matrix of data.
vrb.nm	a character vector of colnames from data specifying the variables which will be used to determine the partially observed cases.
ov.min	minimum frequency of observed values required per row. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and length(vrb.nm).

prop	logical vector of length 1 specifying whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
inclusive	logical vector of length 1 specifying whether the case should be included if the frequency of observed values in a row is exactly equal to ov.min.

Value

logical vector of length = nrow(data) with names = rownames(data) specifying if the frequency of observed values is greater than (or equal to, if inclusive = TRUE) ov.min.

See Also

complete.cases rowNA ncases

Examples

```
cases2keep <- partial.cases(data = airquality,
   vrb.nm = c("Ozone","Solar.R","Wind"), ov.min = .66)
airquality2 <- airquality[cases2keep, ] # all cases with 2/3 variables observed
cases2keep <- partial.cases(data = airquality,
   vrb.nm = c("Ozone","Solar.R","Wind"), ov.min = 1, prop = TRUE, inclusive = TRUE)
complete_cases <- complete.cases(airquality)
identical(x = unname(cases2keep),
   y = complete_cases) # partial.cases(ov.min = 1, prop = TRUE,
   # inclusive = TRUE) = complete.cases()
```

---

pomp *Recode a Numeric Vector to Percentage of Maximum Possible (POMP) Units*

Description

pomp recodes a numeric vector to percentage of maximum possible (POMP) units. This can be useful when data is measured with arbitrary units (e.g., Likert scale).

Usage

```
pomp(x, mini, maxi, relative = FALSE, unit = 1)
```

Arguments

x	numeric vector.
mini	numeric vector of length 1 specifying the minimum numeric value possible.
maxi	numeric vector of length 1 specifying the maximum numeric value possible.
relative	logical vector of length 1 specifying whether relative POMP scores (rather than absolute POMP scores) should be created. If TRUE, then the mini and maxi arguments are ignored. See details for the distinction between absolute and relative POMP scores.

`unit` numeric vector of length 1 specifying how many percentage points is desired for the units. Traditionally, POMP scores use `unit = 1` (default) such that one unit is one percentage point. However, another option is to use `unit = 100` such that one unit is all 100 percentage points (i.e., proportion of maximum possible). This argument also gives the flexibility of specifying units in between 1 and 100 percentage points. For example, `unit = 50` would mean that one unit represents going from low (i.e., 25th percentile) to high (i.e., 75th percentile) on the variable.

Details

There are too common approaches to POMP scores: 1) absolute POMP units where the minimum and maximum are the smallest/largest values possible from the measurement instrument (e.g., 1 to 7 on a Likert scale) and 2) relative POMP units where the minimum and maximum are the smallest/largest values observed in the data (e.g., 1.3 to 6.8 on a Likert scale). Both will be correlated perfectly with the original units as they are each linear transformations.

Value

numeric vector from recoding `x` to percentage of maximum possible (pomp) with units specified by `unit`.

See Also

`pomps`

Examples

```
vec <- psych::bfi[[1]]
pomp(x = vec, mini = 1, maxi = 6) # absolute POMP units
pomp(x = vec, relative = TRUE) # relative POMP units
pomp(x = vec, mini = 1, maxi = 6, unit = 100) # unit = 100
pomp(x = vec, mini = 1, maxi = 6, unit = 50) # unit = 50
```

---

pomps *Recode Numeric Data to Percentage of Maximum Possible (POMP) Units*

Description

`pomps` recodes numeric data to percentage of maximum possible (POMP) units. This can be useful when data is measured with arbitrary units (e.g., Likert scale).

Usage

```
pomps(
  data,
  vrb.nm,
  mini,
  maxi,
  relative = FALSE,
  unit = 1,
  suffix = paste0("_p", unit)
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.
mini	numeric vector of length 1 specifying the minimum numeric value possible. Note, this is assumed to be the same for each variable.
maxi	numeric vector of length 1 specifying the maximum numeric value possible. Note, this is assumed to be the same for each variable.
relative	logical vector of length 1 specifying whether relative POMP scores (rather than absolute POMP scores) should be created. If TRUE, then the mini and maxi arguments are ignored. See details for the distinction between absolute and relative POMP scores.
unit	numeric vector of length 1 specifying how many percentage points is desired for the units. Traditionally, POMP scores use unit = 1 (default) such that one unit is one percentage point. However, another option is to use unit = 100 such that one unit is all 100 percentage points (i.e., proportion of maximum possible). This argument also gives the flexibility of specifying units in between 1 and 100 percentage points. For example, unit = 50 would mean that one unit represents going from low (i.e., 25th percentile) to high (i.e., 75th percentile) on the variable.
suffix	character vector of length 1 specifying the string to add to the end of the column names in the return object.

Details

There are too common approaches to POMP scores: 1) absolute POMP units where the minimum and maximum are the smallest/largest values possible from the measurement instrument (e.g., 1 to 7 on a Likert scale) and 2) relative POMP units where the minimum and maximum are the smallest/largest values observed in the data (e.g., 1.3 to 6.8 on a Likert scale). Both will be correlated perfectly with the original units as they are each linear transformations.

Value

data.frame of variables recoded to percentage of maximum possible (pomp) with units specified by unit and names specified by paste0(vrb.nm, suffix).

See Also

pomp

Examples

```
vrb_nm <- names(psych::bfi)[grepl(pattern = "A", x = names(psych::bfi))]
pomps(data = psych::bfi, vrb.nm = vrb_nm, min = 1, max = 6) # absolute POMP units
pomps(data = psych::bfi, vrb.nm = vrb_nm, relative = TRUE) # relative POMP units
pomps(data = psych::bfi, vrb.nm = vrb_nm, min = 1, max = 6, unit = 100) # unit = 100
pomps(data = psych::bfi, vrb.nm = vrb_nm, min = 1, max = 6, unit = 50) # unit = 50
pomps(data = psych::bfi, vrb.nm = vrb_nm, min = 1, max = 6, suffix = "_pomp")
```

---

props_compare *Proportion Comparisons for Multiple Variables across 3+ Independent Groups (Chi-square Tests of Independence)*

Description

`prop_compare` tests for proportion differences across 3+ independent groups with chi-square tests of independence. The function also calculates the descriptive statistics for each group, Cramer's V and its confidence interval as a standardized effect size, and can provide the X by 2 contingency tables. `prop_compare` is simply a wrapper for `prop.test` plus some extra calculations.

Usage

```
props_compare(
  data,
  vrb.nm,
  nom.nm,
  lvl = levels(as.factor(data[[nom.nm]])),
  yates = TRUE,
  ci.level = 0.95,
  rtn.table = TRUE,
  check = TRUE
)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the dummy variables, in other words, variables that only have values of 0 or 1 (or missing values).
`nom.nm`	character vector of length 1 specifying the colname in `data` containing a nominal variable that takes on three or more unordered values (or missing values).
`lvl`	character vector with length 3+ specifying the unique values for the 3+ independent groups. If `nom` is a factor, then `lvl` should be the factor levels rather than the underlying integer codes. This argument allows you to specify order of the proportions in the return object.

`yates` logical vector of length 1 specifying whether the Yate's continuity correction should be applied for small samples. See `chisq.test` for details.

`ci.level` numeric vector of length 1 specifying the confidence level. `ci.level` must range from 0 to 1.

`rtn.table` logical vector of lengh 1 specifying whether the return object should include the X by 2 contingency table of counts with totals for each dummy variable and the X by 2 overall percentages table with totals for each dummy variable. If TRUE, then the last two elements of the return object are "count" containing an array of counts and "percent" containing an array of overall percentages.

`check` logical vector of length 1 specifying whether the input arguments should be checked for errors. For example, if `lvl` has values that are not present in `data[[nom.nm]]`. This is a tradeoff between computational efficiency (FALSE) and more useful error messages (TRUE).

Details

The confidence interval for Cramer's V is calculated with fisher's r to z transformation as Cramer's V is a kind of multiple correlation coefficient. Cramer's V is transformed to fisher's z units, a symmetric confidence interval for fisher's z is calculated, and then the lower and upper bounds are back-transformed to Cramer's V units.

Value

list of data.frames containing statistical information about the proportion comparisons: 1) nhst = chi-square test of independence stat info in a data.frame, 2) desc = descriptive statistics stat info in a data.frame (note there could be more than 3 groups - groups i, j, and k are just provided as an example), 3) std = standardized effect size and its confidence interval in a data.frame, 4) count = numeric array with dim = `[X+1, 3, length(vrb.nm)]` of the X by 2 contingency table of counts for each dummy variable with an additional row and column for totals (if `rtn.table` = TRUE), 5) percent = numeric array with dim = `[X+1, 3, length(vrb.nm)]` of the X by 2 contingency table of overall percentages for each dummy variable with an additional row and column for totals (if `rtn.table` = TRUE).

1) nhst = chi-square test of independence stat info in a data.frame

est average proportion difference absolute value (i.e., |group j - group i|)

se NA (to remind the user there is no standard error for the test)

X2 chi-square value

df degrees of freedom (of the nominal variable)

p two-sided p-value

2) desc = descriptive statistics stat info in a data.frame (note there could be more than 3 groups - groups i, j, and k are just provided as an example):

prop_'lvl[k '] proportion of group k

prop_'lvl[j '] proportion of group j

prop_'lvl[i '] proportion of group i

sd_'lvl[k '] standard deviation of group k

sd_'lvl[j '] standard deviation of group j

sd_'lvl[i '] standard deviation of group i

n_'lvl[k '] sample size of group k

n_'lvl[j '] sample size of group j

n_'lvl[i '] sample size of group i

3) std = standardized effect size and its confidence interval in a data.frame

cramer Cramer's V estimate

lwr lower bound of Cramer's V confidence interval

upr upper bound of Cramer's V confidence interval

4) count = numeric array with dim = `[X+1, 3, length(vrb.nm)]` of the X by 2 contingency table of counts for each dummy variable with an additional row and column for totals (if `rtn.table` = TRUE).

The 3+ unique observed values of `data[[nom.nm]]` - plus the total - are the rows and the two unique observed values of `data[[vrb.nm]]` (i.e., 0 and 1) - plus the total - are the columns. The variables in `data[vrb.nm]` are the layers. The dimlabels are "nom" for the rows and "x" for the columns and "vrb" for the layers. The rownames are 1. 'lvl[i]', 2. 'lvl[j]', 3. 'lvl[k]', 4. "total". The colnames are 1. "0", 2. "1", 3. "total". The laynames are `vrb.nm`.

5) percent = numeric array with dim = `[X+1, 3, length(vrb.nm)]` of the X by 2 contingency table of overall percentages for each dummy variable with an additional row and column for totals (if `rtn.table` = TRUE).

The 3+ unique observed values of `data[[nom.nm]]` - plus the total - are the rows and the two unique observed values of `data[[vrb.nm]]` (i.e., 0 and 1) - plus the total - are the columns. The variables in `data[vrb.nm]` are the layers. The dimlabels are "nom" for the rows, "x" for the columns, and "vrb" for the layers. The rownames are 1. 'lvl[i]', 2. 'lvl[j]', 3. 'lvl[k]', 4. "total". The colnames are 1. "0", 2. "1", 3. "total". The laynames are `vrb.nm`.

See Also

`prop.test` the workhorse for `prop_compare`, `prop_compare` for a single dummy variable, `props_diff` for only 2 independent groups (aka binary variable),

Examples

```
# rtn.table = TRUE (default)

# multiple variables
tmp <- replicate(n = 10, expr = mtcars, simplify = FALSE)
mtcars2 <- str2str::ld2d(tmp)
mtcars2$"gear_dum" <- ifelse(mtcars2$"gear" > 3, yes = 1L, no = 0L)
mtcars2$"carb_dum" <- ifelse(mtcars2$"carb" > 3, yes = 1L, no = 0L)
vrb_nm <- c("am","gear_dum","carb_dum") # dummy variables
lapply(X = vrb_nm, FUN = function(nm) {
   tmp <- c("cyl", nm)
```

```
   table(mtcars2[tmp])
})
props_compare(data = mtcars2, vrb.nm = c("am","gear_dum","carb_dum"), nom.nm = "cyl")

# single variable
props_compare(mtcars2, vrb.nm = "am", nom.nm = "cyl")

# rtn.table = FALSE (no "count" or "percent" list elements)

# multiple variables
props_compare(data = mtcars2, vrb.nm = c("am","gear_dum","carb_dum"), nom.nm = "cyl",
   rtn.table = FALSE)

# single variable
props_compare(mtcars2, vrb.nm = "am", nom.nm = "cyl",
   rtn.table = FALSE)

# more than 3 groups
airquality2 <- airquality
airquality2$"Wind_dum" <- ifelse(airquality$"Wind" >= 10, yes = 1, no = 0)
airquality2$"Solar.R_dum" <- ifelse(airquality$"Solar.R" >= 100, yes = 1, no = 0)
props_compare(airquality2, vrb.nm = c("Wind_dum","Solar.R_dum"), nom.nm = "Month")
props_compare(airquality2, vrb.nm = "Wind_dum", nom.nm = "Month")
```

`props_diff` *Proportion Difference of Multiple Variables Across Two Independent Groups (Chi-square Tests of Independence)*

Description

`props_diff` tests the proportion difference of multiple variables across two independent groups with chi-square tests of independence. The function also calculates the descriptive statistics for each group, various standardized effect sizes (e.g., Cramer's V), and can provide the 2x2 contingency tables. `props_diff` is simply a wrapper for `prop.test` plus some extra calculations.

Usage

```
props_diff(
  data,
  vrb.nm,
  bin.nm,
  lvl = levels(as.factor(data[[bin.nm]])),
  yates = TRUE,
  zero.cell = 0.05,
  smooth = TRUE,
  ci.level = 0.95,
  rtn.table = TRUE,
  check = TRUE
)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector specifying the colnames in `data` for the variables. Since we are testing proportions, the variables must be dummy codes such that they only have values of 0 or 1 (or missing values).
`bin.nm`	character vector of length 1 specifying the colname in `data` for the binary variable that only takes on two values (or missing values), specifying the two independent groups.
`lvl`	character vector with length 2 specifying the unique values for the two groups. If `bin` is a factor, then `lvl` should be the factor levels rather than the underlying integer codes. This argument allows you to specify the direction of the prop difference. `prop_diff` calculates the prop differences as `x[ bin == lvl[2] ] - x[ bin == lvl[1] ]` such that it is group 2 - group 1. By changing which group is group 1 vs. group 2, the direction of the prop differences can be changed. See details of `prop_diff`.
`yates`	logical vector of length 1 specifying whether the Yate's continuity correction should be applied for small samples. See `chisq.test` for details.
`zero.cell`	numeric vector of length 1 specifying what value to impute for zero cell counts in the 2x2 contingency table when computing the tetrachoric correlations. See `tetrachoric` for details.
`smooth`	logical vector of length 1 specifying whether a smoothing algorithm should be applied when estimating the tetrachoric correlations. See `tetrachoric` for details.
`ci.level`	numeric vector of length 1 specifying the confidence level. `ci.level` must range from 0 to 1.
`rtn.table`	logical vector of lengh 1 specifying whether the return object should include the 2x2 contingency table of counts with totals and the 2x2 overall percentages table. If TRUE, then the last two elements of the return object are "count" containing a 3D array of counts and "percent" containing a 3D array of overall percentages.
`check`	logical vector of length 1 specifying whether the input arguments should be checked for errors. For example, if `data[[bin.nm]]` has more than 2 unique values (other than missing values). This is a tradeoff between computational efficiency (FALSE) and more useful error messages (TRUE).

Value

list of data.frames containing statistical information about the prop differences (the rownames of each data.frame are `vrb.nm`): 1) chisqtest = chi-square tests of independence stat info in a data.frame, 2) describes = descriptive statistics stat info in a data.frame, 3) effects = various standardized effect sizes in a data.frame, 4) count = numeric 3D array with dim = `[3, 3, length(vrb.nm)]` of the 2x2 contingency tables of counts with additional rows and columns for totals (if `rtn.table` = TRUE), 5) percent = numeric 3D array with dim = `[3, 3, length(vrb.nm)]` of the 2x2 contingency tables of overall percentages with additional rows and columns for totals (if `rtn.table` = TRUE).

1) chisqtest = chi-square tests of independence stat info in a data.frame

est mean difference estimate (i.e., group 2 - group 1)

se NA (to remind the user there is no standard error for the test)

X2 chi-square value

df degrees of freedom (will always be 1)

p two-sided p-value

lwr lower bound of the confidence interval

upr upper bound of the confidence interval

2) describes = descriptive statistics stat info in a data.frame

prop_'lvl[2 '] proportion of group 2

prop_'lvl[1 '] proportion of group 1

sd_'lvl[2 '] standard deviation of group 2

sd_'lvl[1 '] standard deviation of group 1

n_'lvl[2 '] sample size of group 2

n_'lvl[1 '] sample size of group 1

3) effects = various standardized effect sizes in a data.frame

cramer Cramer's V estimate

h Cohen's h estimate

phi Phi coefficient estimate

yule Yule coefficient estimate

tetra Tetrachoric correlation estimate

OR odds ratio estimate

RR risk ratio estimate calculated as (i.e., group 2 / group 1). Note this value will often differ when recoding variables (as it should).

4) count = numeric 3D array with dim = `[3, 3, length(vrb.nm)]` of the 2x2 contingency tables of counts with additional rows and columns for totals (if `rtn.table` = TRUE).

The two unique observed values of `data[vrb.nm]` (i.e., 0 and 1) - plus the total - are the rows and the two unique observed values of `data[[bin.nm]]` - plus the total - are the columns. The variables themselves as the layers (i.e., 3rd dimension of the array). The dimlabels are "bin" for the rows, "x" for the columns, and "vrb" for the layers. The rownames are 1. "0", 2. "1", 3. "total". The colnames are 1. 'lvl[1]', 2. 'lvl[2]', 3. "total". The laynames are `vrb.nm`.

5) percent = numeric 3D array with dim = `[3, 3, length(vrb.nm)]` of the 2x2 contingency tables of overall percentages with additional rows and columns for totals (if `rtn.table` = TRUE).

The two unique observed values of `data[vrb.nm]` (i.e., 0 and 1) - plus the total - are the rows and the two unique observed values of `data[[bin]]` - plus the total - are the columns. The variables themselves as the layers (i.e., 3rd dimension of the array). The dimlabels are "bin" for the rows, "x" for the columns, and "vrb" for the layers. The rownames are 1. "0", 2. "1", 3. "total". The colnames are 1. 'lvl[1]', 2. 'lvl[2]', 3. "total". The laynames are `vrb.nm`.

See Also

prop.test the workhorse for props_diff, prop_diff for a single dummy variable, phi for another phi coefficient function Yule for another yule coefficient function tetrachoric for another tetrachoric coefficient function

Examples

```
# rtn.table = TRUE (default)

# multiple variables
mtcars2 <- mtcars
mtcars2$"vs_bin" <- ifelse(mtcars$"vs" == 1, yes = "yes", no = "no")
mtcars2$"gear_dum" <- ifelse(mtcars2$"gear" > 3, yes = 1L, no = 0L)
mtcars2$"carb_dum" <- ifelse(mtcars2$"carb" > 3, yes = 1L, no = 0L)
vrb_nm <- c("am","gear_dum","carb_dum") # dummy variables
lapply(X = vrb_nm, FUN = function(nm) {
   tmp <- c("vs_bin", nm)
   table(mtcars2[tmp])
})
props_diff(data = mtcars2, vrb.nm = c("am","gear_dum","carb_dum"), bin.nm = "vs_bin")

# single variable
props_diff(mtcars2, vrb.nm = "am", bin.nm = "vs_bin")

# rtn.table = FALSE (no "count" or "percent" list elements)

# multiple variables
props_diff(data = mtcars2, vrb.nm = c("am","gear_dum","carb_dum"), bin.nm = "vs",
   rtn.table = FALSE)

# single variable
props_diff(mtcars, vrb.nm = "am", bin.nm = "vs",
   rtn.table = FALSE)
```

---

props_test *Test for Multiple Sample Proportion Against Pi (Chi-square Tests of Goodness of Fit)*

Description

props_test tests for multiple sample proportion difference from population proportions with chi-square tests of goodness of fit. The default is that the goodness of fit is consistent with a population proportion Pi of 0.50. The function also calculates the descriptive statistics, various standardized effect sizes (e.g., Cramer's V), and can provide the 1x2 contingency tables. props_test is simply a wrapper for prop.test plus some extra calculations.

Usage

```
props_test(
  data,
  dum.nm,
  pi = 0.5,
  yates = TRUE,
  ci.level = 0.95,
  rtn.table = TRUE,
  check = TRUE
)
```

Arguments

data	data.frame of data.
dum.nm	character vector of length 1 specifying the colnames in data of the variables used to calculate the proportions. The variables must only have values of 0 or 1 (or missing values), or be otherwise known as dummy variables. See is.dummy.
pi	numeric vector of length = length(dum.nm) or length 1 specifying the population proportion values to compare the sample proportions against. The order of the values should be the same as the order in dum.nm. When length 1, the same population proportion value is used for all the variables.
yates	logical vector of length 1 specifying whether the Yate's continuity correction should be applied for small samples. See chisq.test for details.
ci.level	numeric vector of length 1 specifying the confidence level. ci.level must range from 0 to 1.
rtn.table	logical vector of lengh 1 specifying whether the return object should include the rbinded 1x2 contingency table of counts with totals and the rbinded 1x2 overall percentages table. If TRUE, then the last two elements of the return object are "count" containing a data.frame of counts and "percent" containing a data.frame of overall percentages.
check	logical vector of length 1 specifying whether the input arguments should be checked for errors. For example, if data[dum.nm] are all dummy variables that only take on values of 0 or 1 (or missing values). This is a tradeoff between computational efficiency (FALSE) and more useful error messages (TRUE).

Value

list of data.frames containing statistical information about the proportion differences from pi: 1) nhst = chi-square test of goodness of fit stat info in a data.frame, 2) desc = descriptive statistics stat info in a data.frame, 3) std = various standardized effect sizes in a data.frame, 4) count = data.frame containing the rbinded 1x2 tables of counts with an additional column for the total (if rtn.table = TRUE), 5) percent = data.frame containing the rbinded 1x2 tables of overall percentages with an additional column for the total (if rtn.table = TRUE)

1) nhst = chi-square test of goodness of fit stat info in a data.frame

est proportion difference estimate (i.e., sample proportion - pi)

se NA (to remind the user there is no standard error for the test)

X2 chi-square value

df degrees of freedom (will always be 1)

p two-sided p-value

2) desc = descriptive statistics stat info in a data.frame

prop sample proportion

pi popularion proportion provided by the user (or 0.50 by default)

sd standard deviation

n sample size

lwr lower bound of the confidence interval of the sample proportion itself

upr upper bound of the confidence interval of the sample proportion itself

3) std = various standardized effect sizes in a data.frame

cramer Cramer's V estimate

h Cohen's h estimate

4) count = data.frame containing the rbinded 1x2 tables of counts with an additional column for the total (if `rtn.table` = TRUE). The colnames are 1. "0", 2. "1", 3. "total"

5) percent = data.frame containing the rbinded 1x2 tables of overall percentages with an additional column for the total (if `rtn.table` = TRUE). The colnames are 1. "0", 2. "1", 3. "total"

See Also

`prop.test` the workhorse for `prop_test`, `prop_test` for a single dummy variables, `props_diff` for chi-square tests of independence,

Examples

```
# multiple variables
mtcars2 <- mtcars
mtcars2$"gear_dum" <- ifelse(mtcars2$"gear" > 3, yes = 1L, no = 0L)
mtcars2$"carb_dum" <- ifelse(mtcars2$"carb" > 3, yes = 1L, no = 0L)
vrb_nm <- c("am","gear_dum","carb_dum") # dummy variables
lapply(X = vrb_nm, FUN = function(nm) {
   table(mtcars2[nm])
})
props_test(data = mtcars2, dum.nm = c("am","gear_dum","carb_dum"))
props_test(data = mtcars2, dum.nm = c("am","gear_dum","carb_dum"),
   rtn.table = FALSE)

# single variable
props_test(data = mtcars2, dum.nm = "am")
props_test(data = mtcars2, dum.nm = "am", rtn.table = FALSE)

# error from non-dummy variables
```

```
## Not run:
props_test(data = mtcars2, dum.nm = c("am","gear","carb"))

## End(Not run)
```

`prop_compare` *Proportion Comparisons for a Single Variable across 3+ Independent Groups (Chi-square Test of Independence)*

Description

`prop_compare` tests for proportion differences across 3+ independent groups with a chi-square test of independence. The function also calculates the descriptive statistics for each group, Cramer's V and its confidence interval as a standardized effect size, and can provide the X by 2 contingency tables. `prop_compare` is simply a wrapper for `prop.test` plus some extra calculations.

Usage

```
prop_compare(
  x,
  nom,
  lvl = levels(as.factor(nom)),
  yates = TRUE,
  ci.level = 0.95,
  rtn.table = TRUE,
  check = TRUE
)
```

Arguments

`x`	numeric vector that only has values of 0 or 1 (or missing values), otherwise known as a dummy variable.
`nom`	atomic vector that takes on three or more unordered values (or missing values), otherwise known as a nominal variable.
`lvl`	character vector with length 2 specifying the unique values for the two groups. If `nom` is a factor, then `lvl` should be the factor levels rather than the underlying integer codes. This argument allows you to specify order of the proportions in the return object.
`yates`	logical vector of length 1 specifying whether the Yate's continuity correction should be applied for small samples. See `chisq.test` for details.
`ci.level`	numeric vector of length 1 specifying the confidence level. `ci.level` must range from 0 to 1.

`rtn.table` logical vector of lengh 1 specifying whether the return object should include the X by 2 contingency table of counts with totals and the X by 2 overall percentages table. If TRUE, then the last two elements of the return object are "count" containing a matrix of counts and "percent" containing a matrix of overall percentages.

`check` logical vector of length 1 specifying whether the input arguments should be checked for errors. For example, if `nom` has length different than the length of `x`. This is a tradeoff between computational efficiency (FALSE) and more useful error messages (TRUE).

Details

The confidence interval for Cramer's V is calculated with fisher's r to z transformation as Cramer's V is a kind of multiple correlation coefficient. Cramer's V is transformed to fisher's z units, a symmetric confidence interval for fisher's z is calculated, and then the lower and upper bounds are back-transformed to Cramer's V units.

Value

list of numeric vectors containing statistical information about the proportion comparisons: 1) nhst = chi-square test of independence stat info in a numeric vector, 2) desc = descriptive statistics stat info in a numeric vector, 3) std = standardized effect size and its confidence interval in a numeric vector, 4) count = numeric matrix with dim = `[X+1, 3]` of the X by 2 contingency table of counts with an additional row and column for totals (if `rtn.table` = TRUE), 5) percent = numeric matrix with dim = `[X+1, 3]` of the X by 2 contingency table of overall percentages with an additional row and column for totals (if `rtn.table` = TRUE).

1) nhst = chi-square test of independence stat info in a numeric vector

est average proportion difference absolute value (i.e., |group j - group i|)

se NA (to remind the user there is no standard error for the test)

X2 chi-square value

df degrees of freedom (of the nominal variable)

p two-sided p-value

2) desc = descriptive statistics stat info in a numeric vector (note there could be more than 3 groups - groups i, j, and k are just provided as an example):

prop_'lvl[k '] proportion of group k

prop_'lvl[j '] proportion of group j

prop_'lvl[i '] proportion of group i

sd_'lvl[k '] standard deviation of group k

sd_'lvl[j '] standard deviation of group j

sd_'lvl[i '] standard deviation of group i

n_'lvl[k '] sample size of group k

n_'lvl[j '] sample size of group j

n_‘lvl[i ‘] sample size of group i

3) std = standardized effect size and its confidence interval in a numeric vector

cramer Cramer's V estimate

lwr lower bound of Cramer's V confidence interval

upr upper bound of Cramer's V confidence interval

4) count = numeric matrix with dim = `[X+1, 3]` of the X by 2 contingency table of counts with an additional row and column for totals (if `rtn.table` = TRUE).

The 3+ unique observed values of `nom` - plus the total - are the rows and the two unique observed values of `x` (i.e., 0 and 1) - plus the total - are the columns. The dimlabels are "nom" for the rows and "x" for the columns. The rownames are 1. ‘lvl[i]‘, 2. ‘lvl[j]‘, 3. ‘lvl[k]‘, 4. "total". The colnames are 1. "0", 2. "1", 3. "total".

5) percent = numeric matrix with dim = `[X+1, 3]` of the X by 2 contingency table of overall percentages with an additional row and column for totals (if `rtn.table` = TRUE).

The 3+ unique observed values of `nom` - plus the total - are the rows and the two unique observed values of `x` (i.e., 0 and 1) - plus the total - are the columns. The dimlabels are "nom" for the rows and "x" for the columns. The rownames are 1. ‘lvl[i]‘, 2. ‘lvl[j]‘, 3. ‘lvl[k]‘, 4. "total". The rownames are 1. "0", 2. "1", 3. "total".

See Also

`prop.test` the workhorse for `prop_compare`, `props_compare` for multiple dummy variables, `prop_diff` for only 2 independent groups (aka binary variable),

Examples

```
tmp <- replicate(n = 10, expr = mtcars, simplify = FALSE)
mtcars2 <- str2str::ld2d(tmp)
mtcars2$"cyl_fct" <- car::recode(mtcars2$"cyl",
   recodes = "4='four'; 6='six'; 8='eight'", as.factor = TRUE)
prop_compare(x = mtcars2$"am", nom = mtcars2$"cyl_fct")
prop_compare(x = mtcars2$"am", nom = mtcars2$"cyl_fct",
   lvl = c("four","six","eight")) # specify order of levels in return object

# more than 3 groups
prop_compare(x = ifelse(airquality$"Wind" >= 10, yes = 1, no = 0), nom = airquality$"Month")
prop_compare(x = ifelse(airquality$"Wind" >= 10, yes = 1, no = 0), nom = airquality$"Month",
   rtn.table = FALSE) # no contingency tables
```

prop_diff *Proportion Difference for a Single Variable across Two Independent Groups (Chi-square Test of Independence)*

Description

prop_diff tests for proportion differences across two independent groups with a chi-square test of independence. The function also calculates the descriptive statistics for each group, various standardized effect sizes (e.g., Cramer's V), and can provide the 2x2 contingency tables. prop_diff is simply a wrapper for prop.test plus some extra calculations.

Usage

```
prop_diff(
  x,
  bin,
  lvl = levels(as.factor(bin)),
  yates = TRUE,
  zero.cell = 0.05,
  smooth = TRUE,
  ci.level = 0.95,
  rtn.table = TRUE,
  check = TRUE
)
```

Arguments

x	numeric vector that only has values of 0 or 1 (or missing values), otherwise known as a dummy variable.
bin	atomic vector that only takes on two values (or missing values), otherwise known as a binary variable.
lvl	character vector with length 2 specifying the unique values for the two groups. If bin is a factor, then lvl should be the factor levels rather than the underlying integer codes. This argument allows you to specify the direction of the prop difference. prop_diff calculates the prop difference as x[bin == lvl[2]] - x[bin == lvl[1]] such that it is group 2 - group 1. By changing which group is group 1 vs. group 2, the direction of the prop difference can be changed. See details.
yates	logical vector of length 1 specifying whether the Yate's continuity correction should be applied for small samples. See chisq.test for details.
zero.cell	numeric vector of length 1 specifying what value to impute for zero cell counts in the 2x2 contingency table when computing the tetrachoric correlation. See tetrachoric for details.
smooth	logical vector of length 1 specifying whether a smoothing algorithm should be applied when estimating the tetrachoric correlation. See tetrachoric for details.

`ci.level` numeric vector of length 1 specifying the confidence level. `ci.level` must range from 0 to 1.

`rtn.table` logical vector of lengh 1 specifying whether the return object should include the 2x2 contingency table of counts with totals and the 2x2 overall percentages table. If TRUE, then the last two elements of the return object are "count" containing a matrix of counts and "percent" containing a matrix of overall percentages.

`check` logical vector of length 1 specifying whether the input arguments should be checked for errors. For example, if `bin` has more than 2 unique values (other than missing values) or if `bin` has length different than the length of `x`. This is a tradeoff between computational efficiency (FALSE) and more useful error messages (TRUE).

Value

list of numeric vectors containing statistical information about the mean difference: 1) nhst = chi-square test of independence stat info in a numeric vector, 2) desc = descriptive statistics stat info in a numeric vector, 3) std = various standardized effect sizes in a numeric vector, 4) count = numeric matrix with dim = `[3, 3]` of the 2x2 contingency table of counts with an additional row and column for totals (if `rtn.table` = TRUE), 5) percent = numeric matrix with dim = `[3, 3]` of the 2x2 contingency table of overall percentages with an additional row and column for totals (if `rtn.table` = TRUE)

1) nhst = chi-square test of independence stat info in a numeric vector

est mean difference estimate (i.e., group 2 - group 1)

se NA (to remind the user there is no standard error for the test)

X2 chi-square value

df degrees of freedom (will always be 1)

p two-sided p-value

lwr lower bound of the confidence interval

upr upper bound of the confidence interval

2) desc = descriptive statistics stat info in a numeric vector

prop_'lvl[2 '] proportion of group 2

prop_'lvl[1 '] proportion of group 1

sd_'lvl[2 '] standard deviation of group 2

sd_'lvl[1 '] standard deviation of group 1

n_'lvl[2 '] sample size of group 2

n_'lvl[1 '] sample size of group 1

3) std = various standardized effect sizes in a numeric vector

cramer Cramer's V estimate

h Cohen's h estimate

phi Phi coefficient estimate

yule Yule coefficient estimate

tetra Tetrachoric correlation estimate

OR odds ratio estimate

RR risk ratio estimate calculated as (i.e., group 2 / group 1). Note this value will often differ when recoding variables (as it should).

4) count = numeric matrix with dim = `[3, 3]` of the 2x2 contingency table of counts with an additional row and column for totals (if `rtn.table` = TRUE).

The two unique observed values of `x` (i.e., 0 and 1) - plus the total - are the rows and the two unique observed values of `bin` - plus the total - are the columns. The dimlabels are "bin" for the rows and "x" for the columns. The rownames are 1. "0", 2. "1", 3. "total". The colnames are 1. 'lvl[1]', 2. 'lvl[2]', 3. "total"

5) percent = numeric matrix with dim = `[3, 3]` of the 2x2 contingency table of overall percentages with an additional row and column for totals (if `rtn.table` = TRUE).

The two unique observed values of `x` (i.e., 0 and 1) - plus the total - are the rows and the two unique observed values of `bin` - plus the total - are the columns. The dimlabels are "bin" for the rows and "x" for the columns. The rownames are 1. "0", 2. "1", 3. "total". The colnames are 1. 'lvl[1]', 2. 'lvl[2]', 3. "total"

See Also

`prop.test` the workhorse for `prop_diff`, `props_diff` for multiple dummy variables, `phi` for another phi coefficient function `Yule` for another yule coefficient function `tetrachoric` for another tetrachoric coefficient function

Examples

```
# chi-square test of independence
# x = "am", bin = "vs"
mtcars2 <- mtcars
mtcars2$"vs_bin" <- ifelse(mtcars$"vs" == 1, yes = "yes", no = "no")
agg(mtcars2$"am", grp = mtcars2$"vs_bin", rep = FALSE, fun = mean)
prop_diff(x = mtcars2$"am", bin = mtcars2$"vs_bin")
prop_diff(x = mtcars2$"am", bin = mtcars2$"vs")

# using \code{lvl} argument
prop_diff(x = mtcars2$"am", bin = mtcars2$"vs_bin")
prop_diff(x = mtcars2$"am", bin = mtcars2$"vs_bin",
   lvl = c("yes","no")) # reverses the direction of the effect
prop_diff(x = mtcars2$"am", bin = mtcars2$"vs",
   lvl = c(1, 0)) # levels don't have to be character

# recoding the variables
prop_diff(x = mtcars2$"am", bin = ifelse(mtcars2$"vs_bin" == "yes",
   yes = "no", no = "yes")) # reverses the direction of the effect
prop_diff(x = ifelse(mtcars2$"am" == 1, yes = 0, no = 1),
   bin = mtcars2$"vs") # reverses the direction of the effect
prop_diff(x = ifelse(mtcars2$"am" == 1, yes = 0, no = 1),
```

```
   bin = ifelse(mtcars2$"vs_bin" == "yes",
      yes = "no", no = "yes")) # double reverse means same direction of the effect

# compare to stats::prop.test
# x = "am", bin = "vs_bin" (binary as the rows; dummy as the columns)
tmp <- c("vs_bin","am") # b/c Roxygen2 will cause problems
table_obj <- table(mtcars2[tmp])
row_order <- nrow(table_obj):1
col_order <- ncol(table_obj):1
table_obj4prop <- table_obj[row_order, col_order]
prop.test(table_obj4prop)

# compare to stats:chisq.test
chisq.test(x = mtcars2$"am", y = mtcars2$"vs_bin")

# compare to psych::phi
cor(mtcars2$"am", mtcars$"vs")
psych::phi(table_obj, digits = 7)

# compare to psych::yule()
psych::Yule(table_obj)

# compare to psych::tetrachoric
psych::tetrachoric(table_obj)
# Note, I couldn't find a case where psych::tetrachoric() failed to compute
psych::tetrachoric(table_obj4prop)

# different than single logistic regression
summary(glm(am ~ vs, data = mtcars, family = binomial(link = "logit")))
```

`prop_test` *Test for Sample Proportion Against Pi (chi-square test of goodness of fit)*

Description

`prop_test` tests for a sample proportion difference from a population proportion with a chi-square test of goodness of fit. The default is that the goodness of fit is consistent with a population proportion Pi of 0.50. The function also calculates the descriptive statistics, various standardized effect sizes (e.g., Cramer's V), and can provide the 1x2 contingency tables. `prop_test` is simply a wrapper for `prop.test` plus some extra calculations.

Usage

```
prop_test(
  x,
  pi = 0.5,
  yates = TRUE,
```

```
  ci.level = 0.95,
  rtn.table = TRUE,
  check = TRUE
)
```

Arguments

x	numeric vector that only has values of 0 or 1 (or missing values), otherwise known as a dummy variable.
pi	numeric vector of length 1 specifying the population proportion value to compare the sample proportion against.
yates	logical vector of length 1 specifying whether the Yate's continuity correction should be applied for small samples. See chisq.test for details.
ci.level	numeric vector of length 1 specifying the confidence level. ci.level must range from 0 to 1.
rtn.table	logical vector of lengh 1 specifying whether the return object should include the 1x2 contingency table of counts with totals and the 1x2 overall percentages table. If TRUE, then the last two elements of the return object are "count" containing a vector of counts and "percent" containing a vector of overall percentages.
check	logical vector of length 1 specifying whether the input arguments should be checked for errors. For example, if x is a dummy variable that only takes on value of 0 or 1 (or missing values). This is a tradeoff between computational efficiency (FALSE) and more useful error messages (TRUE).

Value

list of numeric vectors containing statistical information about the proportion difference from pi: 1) nhst = chi-square test of goodness of fit stat info in a numeric vector, 2) desc = descriptive statistics stat info in a numeric vector, 3) std = various standardized effect sizes in a numeric vector, 4) count = numeric vector of length 3 with table of counts with an additional element for the total (if rtn.table = TRUE), 5) percent = numeric vector of length 3 with table of overall percentages with an element for the total (if rtn.table = TRUE)

1) nhst = chi-square test of goodness of fit stat info in a numeric vector

est proportion difference estimate (i.e., sample proportion - pi)

se NA (to remind the user there is no standard error for the test)

X2 chi-square value

df degrees of freedom (will always be 1)

p two-sided p-value

2) desc = descriptive statistics stat info in a numeric vector

prop sample proportion

pi popularion proportion provided by the user (or 0.50 by default)

sd standard deviation

n sample size

lwr lower bound of the confidence interval of the sample proportion itself

upr upper bound of the confidence interval of the sample proportion itself

3) std = various standardized effect sizes in a numeric vector

cramer Cramer's V estimate

h Cohen's h estimate

4) count = numeric vector of length 3 with table of counts with an additional element for the total (if `rtn.table` = TRUE). The names are 1. "0", 2. "1", 3. "total"

5) percent = numeric vector of length 3 with table of overall percentages with an element for the total (if `rtn.table` = TRUE). The names are 1. "0", 2. "1", 3. "total"

See Also

`prop.test` the workhorse for `prop_test`, `props_test` for multiple dummy variables, `prop_diff` for chi-square test of independence,

Examples

```
# chi-square test of goodness of fit
table(mtcars$"am")
prop_test(mtcars$"am")
prop_test(ifelse(mtcars$"am" == 1, yes = 0, no = 1))

# different than intercept only logistic regression
summary(glm(am ~ 1, data = mtcars, family = binomial(link = "logit")))

# error from non-dummy variable
## Not run:
prop_test(ifelse(mtcars$"am" == 1, yes = "1", no = "0"))
prop_test(ifelse(mtcars$"am" == 1, yes = 2, no = 1))

## End(Not run)
```

`recode2other` *Recode Unique Values in a Character Vector to 0ther (or NA)*

Description

`recode2other` recodes multiple unique values in a character vector to the same new value (e.g., "other", NA_character_). It's primary use is to recode based on the minimum frequency of the unique values so that low frequency values can be combined into the same category; however, it also allows for recoding particular unique values given by the user (see details). This function is a wrapper for `car::recode`, which can handle general recoding of character vectors.

Usage

```
recode2other(
  x,
  freq.min,
  prop = FALSE,
  inclusive = TRUE,
  other.nm = "other",
  extra.nm = NULL
)
```

Arguments

x	character vector. If not a character vector, it will be coarced to one via as.character.
freq.min	numeric vector of length 1 specifying the minimum frequency of a unique value to keep it unchanged and consequentially recode any unique values with frequencies less than (or equal to) it.
prop	logical vector of length 1 specifying if freq.min provides the frequency as a count (FALSE) or proportion (TRUE).
inclusive	logical vector of length 1 specifying whether the frequency of a unique value exactly equal to freq.min should be kept unchanged (and not recoded to other.nm).
other.nm	character vector of length 1 specifying what value the other unique values should be recoded to. This can be NA_character_ resulting in recoding to a missing value.
extra.nm	character vector specifying extra unique values that should be recoded to other.nm that are not included based on the minimum frequency from the combination of freq.min, prop, inclusive. The default is NULL, meaning no extra unique values are recoded.

Details

The extra.nm argument allows for recode2other to be used as simpler function that just recodes particular unique values to the same new value (although arguably this is easier to do using car::recode directly). To do so set freq.min = 0 and provide the unique values to extra.nm. Note, that the current version of this function does not allow for NA_character_ to be included in extra.nm as it will end up treating it as "NA" (see examples).

Value

character vector of the same length as x with unique values with frequency less than freq.nm recoded to other.nm as well as any unique values in extra.nm. While the current version of the function allows for recoding *to* NA values via other.nm, it does not allow for recoding *from* NA values via extra.nm (see examples).

See Also

recode ifelse

Examples

```
# based on minimum frequency unique values
state_region <- as.character(state.region)
recode2other(state_region, freq.min = 13) # freq.min as a count
recode2other(state_region, freq.min = 0.26, prop = TRUE) # freq.min as a proportion
recode2other(state_region, freq.min = 13, other.nm = "_blank_")
recode2other(state_region, freq.min = 13,
   other.nm = NA) # allows for other.nm to be NA
recode2other(state_region, freq.min = 13,
   extra.nm = "South") # add an extra unique value to recode
recode2other(state_region, freq.min = 13,
   inclusive = FALSE) # recodes "West" to "other"

# based on user given unique values
recode2other(state_region, freq.min = 0,
   extra.nm = c("South","West")) # recodes manually rather than by freq.min
# current version does NOT allow for NA to be a unique value that is converted to other
state_region2 <- c(NA, state_region, NA)
recode2other(state_region2, freq.min = 13) # NA remains in the character vector
recode2other(state_region2, freq.min = 0,
   extra.nm = c("South","West",NA)) # NA remains in the character vector
```

`recodes` *Recode Data*

Description

`recodes` recodes data based on specified recodes using the `car::recode` function. This can be used for numeric or character (including factors) data. See `recode` for details. The `levels` argument from `car::recode` is excluded because there is no easy way to vectorize it when only a subset of the variables are factors.

Usage

```
recodes(data, vrb.nm, recodes, suffix = "_r", as.factor, as.numeric = TRUE)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`recodes`	character vector of length 1 specifying the recodes. See details of `recode` for how to use this argument.
`suffix`	character vector of length 1 specifying the string to add to the end of the colnames in the return object.

as.factor — logical vector of length 1 specifying if the recoded columns should be returned as factors. The default depends on the column in data[vrb.nm]. If the column is a factor, then as.factor = TRUE for that column. If the column is not a factor, then as.factor = FALSE for that column. Any non-default, specified value for this argument will result in as.factor being universally applied to all columns in data[vrb.nm].

as.numeric — logical vector of length 1 specifying if the recoded columns should be returned as numeric vectors when possible. This can be useful when having character vectors converted to numeric, such that numbers with typeof character (e.g., "1") will be coerced to typeof numeric (e.g., 1). Note, this argument has no effect on columns in data[vrb.nm] which are typeof character and have letters in their values (e.g., "1a"). Note, this argument is often not needed as you can directly recode to a numeric by excluding quotes from the number in the recodes argument.

Value

data.frame of recoded variables with colnames specified by paste0(vrb.nm, suffix). In general, the columns of the data.frame are the same typeof as those in data except for instances when as.factor and/or as.numeric change the typeof.

See Also

recode reverses

Examples

```
recodes(data = psych::bfi, vrb.nm = c("A1","C4","C5","E1","E2","O2","O5"),
   recodes = "1=6; 2=5; 3=4; 4=3; 5=2; 6=1")
re_codes <- "'Quebec' = 'canada'; 'Mississippi' = 'usa'; 'nonchilled' = 'no'; 'chilled' = 'yes'"
recodes(data = CO2, vrb.nm = c("Type","Treatment"), recodes = re_codes,
   as.factor = FALSE) # convert from factors to characters
```

---

renames — *Rename Data Columns from a Codebook*

Description

renames renames columns in a data.frame from a codebook. The codebook is assumed to be a list of data.frames containing the old and new column names. See details for how the codebook should be structured. The idea is that the codebook has been imported as an excel workbook with different sets of column renaming information in different workbook sheets. This function is simply a wrapper for plyr::rename.

Usage

```
renames(
  data,
  codebook,
  old = 1L,
  new = 2L,
  warn_missing = TRUE,
  warn_duplicated = TRUE
)
```

Arguments

data	data.frame of data.
codebook	list of data.frames containing the old and new column names.
old	numeric vector or character vector of length 1 specifying the position or name of the column in the codebook data.frames that contains the old column names present in data.
new	numeric vector or character vector of length 1 specifying the position or name of the column in the codebook data.frames that contains the new column names to rename to in data.
warn_missing	logical vector of length 1 specifying whether renames should return a warning if any old names in codebook are not present in data.
warn_duplicated	logical vector of length 1 specifying whether renames should return a warning if the renaming process results in duplicate column names in the return object.

Details

codebook is a list of data.frames where one column refers to the old names and another column refers to the new names. Therefore, each row of the data.frames refers to a column in data. The position or names of the columns in the codebook data.frames that contain the old (i.e., old) and new (i.e., new) data columns must be the same for each data.frame in codebook.

Value

data.frame identical to data except that the old names in codebook have been replaced by the new names in codebook.

See Also

rename

Examples

```
code_book <- list(
   data.frame("old" = c("rating","complaints"), "new" = c("RATING","COMPLAINTS")),
   data.frame("old" = c("privileges","learning"), "new" = c("PRIVILEGES","LEARNING"))
)
renames(data = attitude, codebook = code_book, old = "old", new = "new")
```

reorders *Reorder Levels of Factor Data*

Description

reorders re-orders the levels of factor data. The factors are columns in a data.frame where the same reordering scheme is desired. This is often useful before using factor data in a statistical analysis (e.g., lm) or a graph (e.g., ggplot). It is essentially a vectorized version of reorder.default.

Usage

```
reorders(data, fct.nm, ord.nm = NULL, fun, ..., suffix = "_r")
```

Arguments

data	data.frame of data.
fct.nm	character vector of colnames in data that specify the factor columns. If any of the columns specified by fct.nm are not factors, then an error is returned.
ord.nm	character vector of length 1 or NULL. If a character vector of length 1, it is a colname in data specifying the column in data that will be used in conjunction with fun to re-order the factor columns. If NULL (default), it is assumed that each factor column itself will be used in conjunction with fun to re-order the factor columns.
fun	function that will be used to re-order the factor columns. The function is expected to input an atomic vector of length = nrow(data) and return an atomic vector of length 1. fun is applied to data[[ord.nm]] if ord.nm is a character vector of length 1 or applied to each column in data[fct.nm] if ord.nm = NULL.
...	additional named arguments used by fun. For example, if fun is mean, the user might specify an argument na.rm = TRUE to set the na.rm argument in the mean function.
suffix	character vector of length 1 specifying the string that will be appended to the end of the colnames in the return object.

Value

data.frame of re-ordered factor columns with colnames = paste0(fct.nm, suffix).

See Also

reorder.default

Examples

```
# factor vector
reorder(x = state.region, X = state.region,
   FUN = length) # least frequent to most frequent
reorder(x = state.region, X = state.region,
   FUN = function(vec) {-1 * length(vec)}) # most frequent to least frequent

# data.frame of factors
infert_fct <- infert
fct_nm <- c("education","parity","induced","case","spontaneous")
infert_fct[fct_nm] <- lapply(X = infert[fct_nm], FUN = as.factor)
x <- reorders(data = infert_fct, fct.nm = fct_nm,
   fun = length) # least frequent to most frequent
lapply(X = x, FUN = levels)
y <- reorders(data = infert_fct, fct.nm = fct_nm,
   fun = function(vec) {-1 * length(vec)}) # most frequent to least frequent
lapply(X = y, FUN = levels)
# ord.nm specified as a different column in data.frame
z <- reorders(data = infert_fct, fct.nm = fct_nm, ord.nm = "pooled.stratum",
   fun = mean) # category with highest mean for pooled.stratum to
   # category with lowest mean for pooled.stratum
lapply(X = z, FUN = levels)
```

---

revalid *Recode Invalid Values from a Vector*

Description

`revalid` recodes invalid data to specified values. For example, sometimes invalid values are present in a vector of data (e.g., age = -1). This function allows you to specify which values are possible and will then recode any impossible values to `undefined`. This function is a useful wrapper for the function `car::recode`, tailored for the specific use of recoding invalid values.

Usage

```
revalid(x, valid, undefined = NA)
```

Arguments

`x`	atomic vector.
`valid`	atomic vector of valid values for `x`.
`undefined`	atomic vector of length 1 specifying what the invalid values should be recoded to.

Value

atomic vector with the same typeof as `x` where any values not present in `valid` have been recoded to `undefined`.

See Also

revalids valid_test valids_test

Examples

```
revalid(x = attitude[[1]], valid = 25:75, undefined = NA) # numeric vector
revalid(x = as.character(ToothGrowth[["supp"]]), valid = c('VC'),
   undefined = NA) # character vector
revalid(x = ToothGrowth[["supp"]], valid = c('VC'),
   undefined = NA) # factor
```

---

revalids *Recode Invalid Values from Data*

Description

revalids recodes invalid data to specified values. For example, sometimes invalid values are present in a vector of data (e.g., age = -1). This function allows you to specify which values are possible and will then recode any impossible values to undefined. revalids is simply a vectorized version of revalid to more easily revalid multiple columns of a data.frame at the same time.

Usage

```
revalids(data, vrb.nm, valid, undefined = NA, suffix = "_v")
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.
valid	atomic vector of valid values for the data. Note, the valid values must be the same for each variable.
undefined	atomic vector of length 1 specifying what the invalid values should be recoded to.
suffix	character vector of length 1 specifying the string to add to the end of the colnames in the return object.

Value

data.frame of recoded variables where any values not present in valid have been recoded to undefined with colnames specified by paste0(vrb.nm, suffix).

See Also

revalid valids_test valid_test

Examples

```
revalids(data = attitude, vrb.nm = names(attitude),
   valid = 25:75) # numeric data
revalids(data = as.data.frame(CO2), vrb.nm = c("Type","Treatment"),
   valid = c('Quebec','nonchilled')) # factors
```

`reverse` *Reverse Code a Numeric Vector*

Description

`reverse` reverse codes a numeric vector based on minimum and maximum values. For example, say numerical values of response options can range from 1 to 4. The function will change 1 to 4, 2 to 3, 3 to 2, and 4 to 1. If there are an odd number of response options, the middle in the sequence will be unchanged.

Usage

```
reverse(x, mini, maxi)
```

Arguments

`x`	numeric vector.
`mini`	numeric vector of length 1 specifying the minimum numeric value.
`maxi`	numeric vector of length 1 specifying the maximum numeric value.

Value

numeric vector that correlates exactly -1 with `x`.

See Also

`reverses` `reverse.code` `recode`

Examples

```
x <- psych::bfi[[1]]
head(x, n = 15)
y <- reverse(x = psych::bfi[[1]], min = 1, max = 6)
head(y, n = 15)
cor(x, y, use = "complete.obs")
```

`reverses` *Reverse Code Numeric Data*

Description

`reverses` reverse codes numeric data based on minimum and maximum values. For example, say numerical values of response options can range from 1 to 4. The function will change 1 to 4, 2 to 3, 3 to 2, and 4 to 1. If there are an odd number of response options, the middle in the sequence will be unchanged.

Usage

```
reverses(data, vrb.nm, mini, maxi, suffix = "_r")
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables.
`mini`	numeric vector of length 1 specifying the minimum numeric value.
`maxi`	numeric vector of length 1 specifying the maximum numeric value.
`suffix`	character vector of length 1 specifying the string to add to the end of the colnames in the return object.

Details

`reverses` is simply a vectorized version of `reverse` to more easily reverse code multiple columns of a data.frame at the same time.

Value

data.frame of reverse coded variables with colnames specified by `paste0(vrb.nm, suffix)`.

See Also

`reverse` `reverse.code` `recodes`

Examples

```
tmp <- !(is.element(el = names(psych::bfi) , set = c("gender","education","age")))
vrb_nm <- names(psych::bfi)[tmp]
reverses(data = psych::bfi, vrb.nm = vrb_nm, mini = 1, maxi = 6)
```

rowMeans_if *Row Means Conditional on Frequency of Observed Values*

Description

rowMean_if calculates the mean of every row in a numeric or logical matrix conditional on the frequency of observed data. If the frequency of observed values in that row is less than (or equal to) that specified by ov.min, then NA is returned for that row.

Usage

```
rowMeans_if(x, ov.min = 1, prop = TRUE, inclusive = TRUE)
```

Arguments

x	numeric or logical matrix. If not a matrix, it will be coerced to one.
ov.min	minimum frequency of observed values required per row. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and ncol(x).
prop	logical vector of length 1 specifying whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
inclusive	logical vector of length 1 specifying whether the mean should be calculated if the frequency of observed values in a row is exactly equal to ov.min.

Details

Conceptually this function does: apply(X = x, MARGIN = 1, FUN = mean_if, ov.min = ov.min, prop = prop, inclusive = inclusive). But for computational efficiency purposes it does not because then the observed values conditioning would not be vectorized. Instead, it uses rowMeans and then inserts NAs for rows that have too few observed values

Value

numeric vector of length = nrow(x) with names = rownames(x) providing the mean of each row or NA depending on the frequency of observed values.

See Also

rowSums_if colMeans_if colSums_if rowMeans

Examples

```
rowMeans_if(airquality)
rowMeans_if(x = airquality, ov.min = 5, prop = FALSE)
```

rowNA *Frequency of Missing Values by Row*

Description

rowNA compute the frequency of missing values in a matrix by row. This function essentially does apply(X = x, MARGIN = 1, FUN = vecNA). It is also used by other functions in the quest package related to missing values (e.g., rowMeans_if).

Usage

```
rowNA(x, prop = FALSE, ov = FALSE)
```

Arguments

x	matrix with any typeof. If not a matrix, it will be coerced to a matrix via as.matrix. The argument rownames.force is set to TRUE to allow for rownames to carry over for non-matrix objects (e.g., data.frames).
prop	logical vector of length 1 specifying whether the frequency of missing values should be returned as a proportion (TRUE) or a count (FALSE).
ov	logical vector of length 1 specifying whether the frequency of observed values (TRUE) should be returned rather than the frequency of missing values (FALSE).

Value

numeric vector of length = nrow(x), and names = rownames(x), providing the frequency of missing values (or observed values if ov = TRUE) per row. If prop = TRUE, the values will range from 0 to 1. If prop = FALSE, the values will range from 1 to ncol(x).

See Also

is.na vecNA colNA rowsNA

Examples

```
rowNA(as.matrix(airquality)) # count of missing values
rowNA(as.data.frame(airquality)) # with rownames
rowNA(as.matrix(airquality), prop = TRUE) # proportion of missing values
rowNA(as.matrix(airquality), ov = TRUE) # count of observed values
rowNA(as.data.frame(airquality), prop = TRUE, ov = TRUE) # proportion of observed values
```

rowsNA *Frequency of Multiple Sets of Missing Values by Row*

Description

rowsNA computes the frequency of missing values for multiple sets of columns from a data.frame. The arguments prop and ov allow the user to specify if they want to sum or mean the missing values as well as compute the frequency of observed values rather than missing values. This function is essentially a vectorized version of rowNA that inputs and outputs a data.frame.

Usage

```
rowsNA(data, vrb.nm.list, prop = FALSE, ov = FALSE)
```

Arguments

data	data.frame of data.
vrb.nm.list	list where each element is a character vector of colnames in data specifying the variables for that set of columns. The names of vrb.nm.list will be the colnames of the return object.
prop	logical vector of length 1 specifying whether the frequency of missing values should be returned as a proportion (TRUE) or a count (FALSE).
ov	logical vector of length 1 specifying whether the frequency of observed values (TRUE) should be returned rather than the frequency of missing values (FALSE).

Value

data.frame with the frequency of missing values (or observed values if ov = TRUE) for each set of variables. The names are specified by names(vrb.nm.list); if vrb.nm.list does not have any names, then the first element from vrb.nm.list[[i]] is used.

See Also

rowNA colNA vecNA is.na

Examples

```
vrb_list <- lapply(X = c("O","C","E","A","N"), FUN = function(chr) {
   tmp <- grepl(pattern = chr, x = names(psych::bfi))
   names(psych::bfi)[tmp]
})
rowsNA(data = psych::bfi,
   vrb.nm.list = vrb_list) # names set to first elements in `vrb.nm.list`[[i]]
names(vrb_list) <- paste0(c("O","C","E","A","N"), "_m")
rowsNA(data = psych::bfi, vrb.nm.list = vrb_list) # names set to names(`vrb.nm.list`)
```

rowSums_if *Row Sums Conditional on Frequency of Observed Values*

Description

rowSums_if calculates the sum of every row in a numeric or logical matrix conditional on the frequency of observed data. If the frequency of observed values in that row is less than (or equal to) that specified by ov.min, then NA is returned for that row. It also has the option to return a value other than 0 (e.g., NA) when all rows are NA, which differs from rowSums(x, na.rm = TRUE).

Usage

```
rowSums_if(
  x,
  ov.min = 1,
  prop = TRUE,
  inclusive = TRUE,
  impute = TRUE,
  allNA = NA_real_
)
```

Arguments

x	numeric or logical matrix. If not a matrix, it will be coerced to one.
ov.min	minimum frequency of observed values required per row. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and ncol(x).
prop	logical vector of length 1 specifying whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
inclusive	logical vector of length 1 specifying whether the sum should be calculated if the frequency of observed values in a row is exactly equal to ov.min.
impute	logical vector of length 1 specifying if missing values should be imputed with the mean of observed values of x[i,]. If TRUE (default), this will make sums over the same columns with different amounts of observed data comparable.
allNA	numeric vector of length 1 specifying what value should be returned for rows that are all NA. This is most applicable when ov.min = 0 and inclusive = TRUE. The default is NA, which differs from rowSums with na.rm = TRUE where 0 is returned. Note, the value is overwritten by NA if the frequency of observed values in that row is less than (or equal to) that specified by ov.min.

Details

Conceptually this function is doing: apply(X = x, MARGIN = 1, FUN = sum_if, ov.min = ov.min, prop = prop, inclusive = inclusive). But for computational efficiency purposes it does not because then the observed values conditioning would not be vectorized. Instead, it uses rowSums and then inserts NAs for rows that have too few observed values.

Value

numeric vector of length = `nrow(x)` with names = `rownames(x)` providing the sum of each row or NA (or `allNA`) depending on the frequency of observed values.

See Also

`rowMeans_if` `colSums_if` `colMeans_if` `rowSums`

Examples

```
rowSums_if(airquality)
rowSums_if(x = airquality, ov.min = 5, prop = FALSE)
x <- data.frame("x" = c(1, 1, NA), "y" = c(2, NA, NA), "z" = c(NA, NA, NA))
rowSums_if(x)
rowSums_if(x, ov.min = 0)
rowSums_if(x, ov.min = 0, allNA = 0)
identical(x = rowSums(x, na.rm = TRUE),
   y = unname(rowSums_if(x, impute = FALSE, ov.min = 0, allNA = 0))) # identical to
   # rowSums(x, na.rm = TRUE)
```

---

score *Observed Unweighted Scoring of a Set of Variables/Items*

Description

`score` calculates observed unweighted scores across a set of variables/items. If a row's frequency of observed data is less than (or equal to) `ov.min`, then NA is returned for that row. `data[vrb.nm]` is coerced to a matrix before scoring. If the coercion leads to a character matrix, an error is returned.

Usage

```
score(
  data,
  vrb.nm,
  avg = TRUE,
  ov.min = 1,
  prop = TRUE,
  inclusive = TRUE,
  impute = TRUE,
  std = FALSE,
  std.data = std,
  std.score = std
)
```

Arguments

data	data.frame or numeric/logical matrix
vrb.nm	character vector of colnames in data specifying the set of variables/items.
avg	logical vector of length 1 specifying whether mean scores (TRUE) or sum scores (FALSE) should be created.
ov.min	minimum frequency of observed values required per row. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and length(vrb.nm).
prop	logical vector of length 1 specifying whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
inclusive	logical vector of length 1 specifying whether the score should be calculated (rather than NA) if the frequency of observed values in a row is exactly equal to ov.min.
impute	logical vector of length 1 specifying if missing values should be imputed with the mean of observed values from each row of data[vrb.nm] (i.e., row mean imputation). If TRUE (default), this will make sums over the same rows with different frequencies of missing values comparable. Note, this argument is only used when avg = FALSE since when avg = TRUE row mean imputation is always done implicitly.
std	logical vector of length 1 specifying whether 1) data[vrb.nm] should be standardized before scoring and 2) the score standardized after creation. This argument is for convenience as these two standardization processes are often used together. However, this argument will be overwritten by any non-default value for std.data and std.score.
std.data	logical vector of length 1 specifying whether data[vrb.nm] should be standardized before scoring.
std.score	logical vector of length 1 specifying whether the score should be standardized after creation.

Value

numeric vector of the mean/sum of each row or NA if the frequency of observed values is less than (or equal to) ov.min. The names are the rownames of data.

See Also

scores rowMeans_if rowSums_if scoreItems

Examples

```
score(data = attitude, vrb.nm = c("complaints","privileges","learning","raises"))
score(data = attitude, vrb.nm = c("complaints","privileges","learning","raises"),
   std = TRUE) # standardized scoring
score(data = airquality, vrb.nm = c("Ozone","Solar.R","Temp"),
   ov.min = 0.75) # conditional on observed values
```

scores *Observed Unweighted Scoring of Multiple Sets of Variables/Items*

Description

scores calculates observed unweighted scores across multiple sets of variables/items. If a row's frequency of observed data is less than (or equal to) ov.min, then NA is returned for that row. Each set of variables/items are coerced to a matrix before scoring. If the coercion leads to a character matrix, an error is returned. This can be tested with lapply(X = vrb.nm.list, FUN = function(nm) is.character(as.matrix(data[nm]))).

Usage

```
scores(
  data,
  vrb.nm.list,
  avg = TRUE,
  ov.min = 1,
  prop = TRUE,
  inclusive = TRUE,
  impute = TRUE,
  std = FALSE,
  std.data = std,
  std.score = std
)
```

Arguments

data	data.frame or numeric/logical matrix
vrb.nm.list	list where each element is a character vector of colnames in data specifying the variables/items for that score. The names of vrb.nm.list will be the names of the scores in the return object.
avg	logical vector of length 1 specifying whether mean scores (TRUE) or sum scores (FALSE) should be created.
ov.min	minimum frequency of observed values required per row. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and length(vrb.nm.list[[i]]).
prop	logical vector of length 1 specifying whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE). If the multiple sets of variables/items contain different numbers of variables, it probably makes the most sense to use the proportion of observed values (TRUE).
inclusive	logical vector of length 1 specifying whether the scores should be calculated (rather than NA) if the frequency of observed values in a row is exactly equal to ov.min.

impute	logical vector of length 1 specifying if missing values should be imputed with the mean of observed values from each row of data[vrb.nm.list[[i]]] (i.e., row mean imputation). If TRUE (default), this will make sums over the same rows with different frequencies of missing values comparable. Note, this argument is only used when avg = FALSE since when avg = TRUE row mean imputation is always done implicitly.
std	logical vector of length 1 specifying whether 1) the variables should be standardized before scoring and 2) the score standardized after creation. This argument is for convenience as these two standardization processes are often used together. However, this argument will be overwritten by any non-default value for std.data and std.score.
std.data	logical vector of length 1 specifying whether the variables/items should be standardized before scoring.
std.score	logical vector of length 1 specifying whether the scores should be standardized after creation.

Value

data.frame of mean/sum scores with NA for any row with the frequency of observed values less than (or equal to) ov.min. The colnames are specified by names(vrb.nm.list) and rownames by row.names(data).

See Also

score rowMeans_if rowSums_if scoreItems

Examples

```
list_colnames <- list("first" = c("rating","complaints","privileges"),
    "second" = c("learning","raises","critical"))
scores(data = attitude, vrb.nm.list = list_colnames)
list_colnames <- list("first" = c("Ozone","Wind"),
    "second" = c("Solar.R","Temp"))
scores(data = airquality, vrb.nm.list = list_colnames, ov.min = .50,
   inclusive = FALSE) # scoring conditional on observed values
```

---

shift *Shift a Vector (i.e., lag/lead)*

Description

shift shifts elements of a vector right (n < 0) for lags or left (n > 0) for leads replacing the undefined data with a user-defined value (e.g., NA). The number of elements shifted is equal to abs(n). It is assumed that x is already sorted by time such that the first element is earliest in time and the last element is the latest in time.

Usage

```
shift(x, n, undefined = NA)
```

Arguments

x	atomic vector or list vector.
n	integer vector with length 1. Specifies the direction and magnitude of the shift. See details.
undefined	atomic vector with length 1 (probably makes sense to be the same typeof as x). Specifies what to insert for undefined values after the shifting takes place. See details.

Details

If n is negative, then shift inserts undefined into the first abs(n) elements of x, shifting all other values of x to the right abs(n) positions, and then dropping the last abs(n) elements of x to preserve the original length of x. If n is positive, then shift drops the first abs(n) elements of x, shifting all other values of x left abs(n) positions, and then inserts undefined into the last abs(n) elements of x to preserve the original length of x. If n is zero, then shift simply returns x.

It is recommended to use L when specifying n to prevent problems with floating point numbers. shift tries to circumvent this issue by a call to round within shift if n is not an integer; however that is not a complete fail safe. The problem is that as.integer(n) implicit in shift truncates rather than rounds.

Value

an atomic vector of the same length as x that is shifted. If x and undefined are different typeofs, then the return will be coerced to the more complex typeof (i.e., complex to simple: character, double, integer, logical).

See Also

shifts shift_by shifts_by

Examples

```
shift(x = attitude[[1]], n = -1L) # use L to prevent problems with floating point numbers
shift(x = attitude[[1]], n = -2L) # can specify any integer up to the length of `x`
shift(x = attitude[[1]], n = +1L) # can specify negative or positive integers
shift(x = attitude[[1]], n = +2L, undefined = -999) # user-specified indefined value
shift(x = setNames(object = letters, nm = LETTERS), n = 3L) # names are kept
```

shifts *Shift Data (i.e., lag/lead)*

Description

shifts shifts rows of data down (n < 0) for lags or up (n > 0) for leads replacing the undefined data with a user-defined value (e.g., NA). The number of rows shifted is equal to abs(n). It is assumed that data[vrb.nm] is already sorted by time such that the first row is earliest in time and the last row is the latest in time.

Usage

```
shifts(data, vrb.nm, n, undefined = NA, suffix)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.
n	integer vector of length 1. Specifies the direction and magnitude of the shift. See details.
undefined	atomic vector of length 1 (probably makes sense to be the same typeof as the vectors in data[vrb.nm]). Specifies what to insert for undefined values after the shifting takes place. See details.
suffix	character vector of length 1 specifying the string to append to the end of the colnames of the return object. The default depends on the n argument: 1) if n < 0, then suffix = paste0("_g", -n), 2) if n > 0, then suffix = paste0("_d", +n), 3) if n = 0, then suffix = "".

Details

If n is negative, then shifts inserts undefined into the first abs(n) rows of data[vrb.nm], shifting all other rows of x down abs(n) positions, and then dropping the last abs(n) row of data[vrb.nm] to preserve the original nrow of data. If n is positive, then shifts drops the first abs(n) rows of x, shifting all other rows of data[vrb.nm] up abs(n) positions, and then inserts undefined into the last abs(n) rows of x to preserve the original length of data. If n is zero, then shifts simply returns data[vrb.nm].

It is recommended to use L when specifying n to prevent problems with floating point numbers. shifts tries to circumvent this issue by a call to round within shifts if n is not an integer; however that is not a complete fail safe. The problem is that as.integer(n) implicit in shifts truncates rather than rounds.

Value

data.frame of shifted data with colnames specified by suffix.

See Also

shift shifts_by shift_by

Examples

```
shifts(data = attitude, vrb.nm = colnames(attitude), n = -1L)
shifts(data = mtcars, vrb.nm = colnames(mtcars), n = 2L)
```

---

shifts_by *Shift Data (i.e., lag/lead) by Group*

Description

shifts_by shifts rows of data down (n < 0) for lags or up (n > 0) for leads replacing the undefined data with a user-defined value (e.g., NA). The number of rows shifted is equal to abs(n). It is assumed that data[vrb.nm] is already sorted within each group by time such that the first row for that group is earliest in time and the last row for that group is the latest in time. The groups can be specified by multiple columns in data (e.g., grp.nm with length > 1), and interaction will be implicitly called to create the groups.

Usage

```
shifts_by(data, vrb.nm, grp.nm, n, undefined = NA, suffix)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.
grp.nm	character vector of colnames from data specifying the groups.
n	integer vector of length 1. Specifies the direction and magnitude of the shift. See details.
undefined	atomic vector of length 1 (probably makes sense to be the same typeof as the vectors in data[vrb.nm]). Specifies what to insert for undefined values after the shifting takes place. See details.
suffix	character vector of length 1 specifying the string to append to the end of the colnames of the return object. The default depends on the n argument: 1) if n < 0, then suffix = paste0("_gw", -n), 2) if n > 0, then suffix = paste0("_dw", +n), 3) if n = 0, then suffix = "".

Details

If n is negative, then shifts_by inserts undefined into the first abs(n) rows of data[vrb.nm] for each group, shifting all other rows of x down abs(n) positions, and then dropping the last abs(n) row of data[vrb.nm] to preserve the original nrow of each group. If n is positive, then shifts_by drops the first abs(n) rows of x for each group, shifting all other rows of data[vrb.nm] up abs(n)

positions, and then inserts undefined into the last abs(n) rows of x to preserve the original length of each group. If n is zero, then shifts_by simply returns data[vrb.nm].

It is recommended to use L when specifying n to prevent problems with floating point numbers. shifts_by tries to circumvent this issue by a call to round within shifts_by if n is not an integer; however that is not a complete fail safe. The problem is that as.integer(n) implicit in shifts_by truncates rather than rounds.

Value

data.frame of shifted data by group with colnames specified by suffix.

See Also

shift_by shifts shift

Examples

```
shifts_by(data = ChickWeight, vrb.nm = c("weight","Time"), grp.nm = "Chick", n = -1L)
shifts_by(data = mtcars, vrb.nm = c("disp","mpg"), grp.nm = c("vs","am"), n = 1L)
shifts_by(data = as.data.frame(CO2), vrb.nm = c("conc","uptake"),
   grp.nm = c("Type","Treatment"), n = 2L) # multiple grouping columns
```

---

shift_by *Shift a Vector (i.e., lag/lead) by Group*

Description

shift_by shifts elements of a vector right (n < 0) for lags or left (n > 0) for leads by group, replacing the undefined data with a user-defined value (e.g., NA). The number of elements shifted is equal to abs(n). It is assumed that x is already sorted within each group by time such that the first element for that group is earliest in time and the last element for that group is the latest in time.

Usage

```
shift_by(x, grp, n, undefined = NA)
```

Arguments

x	atomic vector or list vector.
grp	list of atomic vector(s) and/or factor(s) (e.g., data.frame), which each have same length as x. It can also be an atomic vector or factor, which will then be made the first element of a list internally.
n	integer vector with length 1. Specifies the direction and magnitude of the shift. See details.
undefined	atomic vector with length 1 (probably makes sense to be the same typeof as x). Specifies what to insert for undefined values after the shifting takes place. See details.

Details

If n is negative, then shift_by inserts undefined into the first abs(n) elements of x for each group, shifting all other values of x to the right abs(n) positions, and then dropping the last abs(n) elements of x to preserve the original length of each group. If n is positive, then shift_by drops the first abs(n) elements of x for each group, shifting all other values of x left abs(n) positions, and then inserts undefined into the last abs(n) elements of x to preserve the original length of each group. If n is zero, then shift_by simply returns x.

It is recommended to use L when specifying n to prevent problems with floating point numbers. shift_by tries to circumvent this issue by a call to round within shift_by if n is not an integer; however that is not a complete fail safe. The problem is that as.integer(n) implicit in shift_by truncates rather than rounds.

Value

an atomic vector of the same length as x that is shifted by group. If x and undefined are different typeofs, then the return will be coerced to the most complex typeof (i.e., complex to simple: character, double, integer, logical).

See Also

shifts_by shift shifts

Examples

```
shift_by(x = ChickWeight[["Time"]], grp = ChickWeight[["Chick"]], n = -1L)
tmp_nm <- c("vs","am") # b/c Roxygen2 doesn't like c() in a []
shift_by(x = mtcars[["disp"]], grp = mtcars[tmp_nm], n = 1L)
tmp_nm <- c("Type","Treatment") # b/c Roxygen2 doesn't like c() in a []
shift_by(x = as.data.frame(CO2)[["uptake"]], grp = as.data.frame(CO2)[tmp_nm],
   n = 2L) # multiple grouping vectors
```

---

summary_ucfa *Summary of a Unidimensional Confirmatory Factor Analysis*

Description

summary_ucfa provides a summary of a unidimensional confirmatory factor analysis on a set of variables/items. Unidimensional meaning a one-factor model where all variables/items load on that factor. The function is a wrapper for cfa and returns a list with four vectors/matrices: 1) model info, 2) fit measures, 3) factor loadings, 4) covariance/correlation residuals. For details on all the cfa arguments see lavOptions.

Usage

```
summary_ucfa(
  data,
  vrb.nm,
  std.ov = FALSE,
  std.lv = TRUE,
  ordered = FALSE,
  meanstructure = TRUE,
  estimator = "ML",
  se = "standard",
  test = "standard",
  missing = "fiml",
  fit.measures = c("chisq", "df", "tli", "cfi", "rmsea", "srmr"),
  std.load = TRUE,
  resid.type = "cor.bollen",
  add.class = TRUE,
  ...
)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` providing the variables/items
`std.ov`	logical vector of length 1 specifying if the variables/items should be standardized
`std.lv`	logical vector of length 1 specifying if the latent factor should be standardized resulting in all factor loadings being estimated. If FALSE, then the first variable/item in `data[vrb.nm]` is fixed to a factor loading of 1.
`ordered`	logical vector of length 1 specifying if the variables/items should be treated as ordered categorical items where polychoric correlations are used.
`meanstructure`	logical vector of length 1 specifying if the mean structure of the factor model should be estimated. This would be the variable/item intercepts (and latent factor mean if `std.lv` = FALSE). Note, this must be true to use Full Information Maximum Likelihood (FIML) to handle missing data via `missing` = "fiml".
`estimator`	character vector of length 1 specifying the estimator to use for parameter estimation. Popular options are 1) "ML" = maximum likelihood estimation based on the multivariate normal distribution, 2) "DWLS" = diagonally weighted least squares which uses the diagnonal of the weight matrix, 3) "WLS" for weighted least squares whiches uses the full weight matrix (often results in computational problems), 4) "ULS" for unweighted least squares that doesn't use a weight matrix. "DWLS", "WLS", and "ULS" can each be used with ordered categorical items when `ordered` = TRUE.
`se`	character vector of length 1 specifying how standard errors should be calculated. Popular options are 1) "standard" for conventional standard errors from inverting the information matrix, 2) "robust.sem" for robust standard errors, 3) "robust.huber.white" for sandwich standard errors.

`test`	character vector of length 1 specifying how the omnibus test statistic should be calculated. Popular options are 1) "standard" for the conventional chi-square statistic, 2) "Satorra-Bentler" for the Satorra-Bentler test statistic, 3) "Yaun.Bentler.Mplus" for the version of the Yuan-Bentler test statistic that Mplus uses, 4) "mean.var.adjusted" for a mean and variance adjusted test statistic, 5) "scaled.shifted" for the version of the mean and variance adjusted test statistic Mplus uses.
`missing`	character vector of length 1 specifying how to handle missing data. Popular options are 1) "fiml" = Full Information Maximum Likelihood (FIML), 2) "pairwise" = pairwise deletion, 3) "listwise" = listwise deletion.
`fit.measures`	character vector specifying which model fit indices to include in the return object. The default option includes the chi-square test statistic ("chisq"), degrees of freedom ("df"), tucker-lewis index ("tli"), comparative fit index ("cfi"), root mean square error of approximation ("rmsea"), and standardized root mean residual ("srmr"). Note, if using robust corrections for `se` and `test`, you will probably want to call the scaled versions of model fit indices (e.g., "chisq.scaled"). See `fitMeasures` for details.
`std.load`	logical vector of length 1 specifying whether the factor loadings included in the return object should be standardized (TRUE) or not (FALSE).
`resid.type`	character vector of length 1 specifying the type of covariance/correlation residuals to include in the return object. Popular options are 1) "raw" for conventional covariance residuals, 2) "cor.bollen" for conventional correlation residuals, 3) "cor.bentler" for correlation residuals that standardizes the model-implied covariance matrix with the observed variances, 4) "standardized" for conventional z-scores of the covariance residuals.
`add.class`	logical vector of length 1 specifying whether the lavaan classes should be added to the returned vectors/matrices (TRUE) or not (FALSE). These classes do not change the underlying vector/matrix and only affect printing.
`...`	any other named arguments available in the `cfa` function. See `lavOptions` for the list of arguments.

Value

list of vectors/matrices providing statistical information about the unidimensional confirmatory factor analysis. If `add.class` = TRUE, then the elements have lavaan classes which affect printing (except for the first "model_info" element which always is just an integer vector). The four elements are:

model_info integer vector providing model information. The first element "converged" is 1 if the model converged and 0 if not. The second element "admissible" is 1 if the model is admissible (e.g., no negative variances) and 0 if not. The third element "nobs" is the number of observations used in the analysis. The fourth element "npar" is the number of parameter estimates.

fit_measures double vector providing model fit indices. The number and names of the fit indices is determined by the `fit.measures` argument.

factor_load 1-column double matrix providing factor loadings. The colname is "latent" and the rownames are the `vrb.nm` argument.

cov_resid covariance/correlation residuals for the model. Note, even though the name has "cov" in it, the residuals can be "cor" if the argument `resid.type` = "cor.bollen" or "cor.bentler".

See Also

ucfa cfa lavaan

Examples

```
# types of models
dat <- psych::bfi[1:250, 16:20] # nueroticism items
summary_ucfa(data = dat, vrb.nm = names(dat)) # default
summary_ucfa(data = dat, vrb.nm = names(dat), estimator = "ML", # MLR
   se = "robust.huber.white", test = "yuan.bentler.mplus", missing = "fiml",
   fit.measures = c("chisq.scaled","df.scaled","tli.scaled","cfi.scaled",
      "rmsea.scaled","srmr"))
summary_ucfa(data = dat, vrb.nm = names(dat), estimator = "ML", # MLM
   se = "robust.sem", test = "satorra.bentler", missing = "listwise",
   fit.measures = c("chisq.scaled","df.scaled","tli.scaled","cfi.scaled",
      "rmsea.scaled","srmr"))
summary_ucfa(data = dat, vrb.nm = names(dat), ordered = TRUE, estimator = "DWLS", # WLSMV
   se = "robust", test = "scaled.shifted", missing = "listwise",
   fit.measures = c("chisq.scaled","df.scaled","tli.scaled","cfi.scaled",
      "rmsea.scaled","wrmr"))

# types of info
dat <- psych::bfi[1:250, 16:20] # nueroticism items
w <- summary_ucfa(data = dat, vrb.nm = names(dat))
x <- summary_ucfa(data = dat, vrb.nm = names(dat), add.class = FALSE)
y <- summary_ucfa(data = dat, vrb.nm = names(dat),
   std.load = FALSE, resid.type = "raw")
z <- summary_ucfa(data = dat, vrb.nm = names(dat),
   std.load = FALSE, resid.type = "raw", add.class = FALSE)
lapply(w, class)
lapply(x, class)
lapply(y, class)
lapply(z, class)
```

---

sum_if *Sum Conditional on Minimum Frequency of Observed Values*

Description

sum_if calculates the sum of a numeric or logical vector conditional on a specified minimum frequency of observed values. If the amount of observed data is less than (or equal to) ov.min, then NA is returned rather than the sum.

Usage

```
sum_if(x, impute = TRUE, ov.min = 1, prop = TRUE, inclusive = TRUE)
```

Arguments

x	numeric or logical vector.
impute	logical vector of length 1 specifying if missing values should be imputed with the mean of observed values of x. If TRUE (default), this will make sums over the same vectors with different amounts of missing data comparable.
ov.min	minimum frequency of observed values required. If prop = TRUE, then this is a decimal between 0 and 1. If prop = FALSE, then this is a integer between 0 and length(x).
prop	logical vector of length 1 specifying whether ov.min should refer to the proportion of observed values (TRUE) or the count of observed values (FALSE).
inclusive	logical vector of length 1 specifying whether the sum should be calculated (rather than NA) if the frequency of observed values is exactly equal to ov.min.

Value

numeric vector of length 1 providing the sum of x or NA conditional on if the frequency of observed data is greater than (or equal to) ov.min.

See Also

sum mean_if make.fun_if

Examples

```
sum_if(x = airquality[[1]], ov.min = .75) # proportion of observed values
sum_if(x = airquality[[1]], ov.min = 116,
   prop = FALSE) # count of observe values
sum_if(x = airquality[[1]], ov.min = 116, prop = FALSE,
   inclusive = FALSE) # not include ov.min value itself
sum_if(x = c(TRUE, NA, FALSE, NA),
   ov.min = .50) # works with logical vectors as well as numeric
```

---

tapply2 *Apply a Function to a (Atomic) Vector by Group*

Description

tapply2 applies a function to a (atomic) vector by group and is an alternative to the base R function tapply. The function is apart of the split-apply-combine type of function discussed in the plyr R package and is somewhat similar to dlply. It splits up one (atomic) vector .xinto a (atomic) vector for each group in .grp, applies a function .fun to each (atomic) vector, and then returns the results as a list with names equal to the group values unique(interaction(.grp.nm, sep = .sep)). tapply2 is simply split.default + lapply. Similar to dlply, The arguments all start with . so that they do not conflict with arguments from the function .fun. If you want to apply a function a data.frame rather than a (atomic) vector, then use by2.

Usage

```
tapply2(.x, .grp, .sep = ".", .fun, ...)
```

Arguments

.x	atomic vector
.grp	list of atomic vector(s) and/or factor(s) (e.g., data.frame) containing the groups. They should each have same length as .x. It can also be an atomic vector or factor, which will then be made the first element of a list internally.
.sep	character vector of length 1 specifying the string to combine the group values together with. .sep is only used if there are multiple grouping variables (i.e., .grp is a list with multiple elements).
.fun	function to apply to .x for each group.
...	additional named arguments to pass to .fun.

Value

list of objects containing the return object of .fun for each group. The names are the unique combinations of the grouping variables (i.e., unique(interaction(.grp, sep = .sep))).

See Also

tapply by2 dlply

Examples

```
# one grouping variable
tapply2(mtcars$"cyl", .grp = mtcars$"vs", .fun = median, na.rm = TRUE)

# two grouping variables
grp_nm <- c("vs","am") # Roxygen runs the whole script if I put a c() in a []
x <- tapply2(mtcars$"cyl", .grp = mtcars[grp_nm], .fun = median, na.rm = TRUE)
print(x)
str(x)

# compare to tapply
grp_nm <- c("vs","am") # Roxygen runs the whole script if I put a c() in a []
y <- tapply(mtcars$"cyl", INDEX = mtcars[grp_nm],
   FUN = median, na.rm = TRUE, simplify = FALSE)
print(y)
str(y) # has dimnames rather than names
```

ucfa *Unidimensional Confirmatory Factor Analysis*

Description

ucfa conducts a unidimensional confirmatory factor analysis on a set of variables/items. Unidimensional meaning a one-factor model where all variables/items load on that factor. The function is a wrapper for cfa and returns an object of class "lavaan": lavaan. This then allows the user to extract statistical information from the object (e.g., lavInspect). For details on all the arguments see lavOptions.

Usage

```
ucfa(
  data,
  vrb.nm,
  std.ov = FALSE,
  std.lv = TRUE,
  ordered = FALSE,
  meanstructure = TRUE,
  estimator = "ML",
  se = "standard",
  test = "standard",
  missing = "fiml",
  ...
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data providing the variables/items
std.ov	logical vector of length 1 specifying if the variables/items should be standardized
std.lv	logical vector of length 1 specifying if the latent factor should be standardized resulting in all factor loadings being estimated. If FALSE, then the first variable/item in data[vrb.nm] is fixed to a factor loading of 1.
ordered	logical vector of length 1 specifying if the variables/items should be treated as ordered categorical items where polychoric correlations are used.
meanstructure	logical vector of length 1 specifying if the mean structure of the factor model should be estimated. This would be the variable/item intercepts (and latent factor mean if std.lv = FALSE). Note, this must be true to use Full Information Maximum Likelihood (FIML) to handle missing data via missing = "fiml".
estimator	character vector of length 1 specifying the estimator to use for parameter estimation. Popular options are 1) "ML" = maximum likelihood estimation based on the multivariate normal distribution, 2) "DWLS" = diagonally weighted least

squares which uses the diagnonal of the weight matrix, 3) "WLS" for weighted least squares whiches uses the full weight matrix (often results in computational problems), 4) "ULS" for unweighted least squares that doesn't use a weight matrix. "DWLS", "WLS", and "ULS" can each be used with ordered categorical items when `ordered` = TRUE.

`se` character vector of length 1 specifying how standard errors should be calculated. Popular options are 1) "standard" for conventional standard errors from inverting the information matrix, 2) "robust.sem" for robust standard errors, 3) "robust.huber.white" for sandwich standard errors.

`test` character vector of length 1 specifying how the omnibus test statistic should be calculated. Popular options are 1) "standard" for the conventional chi-square statistic, 2) "Satorra-Bentler" for the Satorra-Bentler test statistic, 3) "Yaun.Bentler.Mplus" for the version of the Yuan-Bentler test statistic that Mplus uses, 4) "mean.var.adjusted" for a mean and variance adjusted test statistic, 5) "scaled.shifted" for the version of the mean and variance adjusted test statistic Mplus uses.

`missing` character vector of length 1 specifying how to handle missing data. Popular options are 1) "fiml" = Full Information Maximum Likelihood (FIML), 2) "pairwise" = pairwise deletion, 3) "listwise" = listwise deletion.

`...` any other named arguments available in the `cfa` function. See `lavOptions` for the list of arguments.

Value

object of class "lavaan" `lavaan` providing the return object from a call to `cfa`.

See Also

`summary_ucfa` `cfa` `lavaan`

Examples

```
dat <- psych::bfi[1:250, 16:20] # nueroticism items
ucfa(data = dat, vrb.nm = names(dat))
ucfa(data = dat, vrb.nm = names(dat), std.ov = TRUE)
ucfa(data = dat, vrb.nm = names(dat), meanstructure = FALSE, missing = "pairwise")
ucfa(data = dat, vrb.nm = names(dat), estimator = "ML", # MLR
   se = "robust.huber.white", test = "yuan.bentler.mplus", missing = "fiml")
ucfa(data = dat, vrb.nm = names(dat), estimator = "ML", # MLM
   se = "robust.sem", test = "satorra.bentler", missing = "listwise")
ucfa(data = dat, vrb.nm = names(dat), ordered = TRUE, estimator = "DWLS", # WLSMV
   se = "robust", test = "scaled.shifted", missing = "listwise")
```

valids_test *Test for Invalid Elements in Data*

Description

`Valid.test` tests whether data has any invalid elements. Valid values are specified by `valid`. Each variable is tested independently. If the variable in `data[vrb.nm]` has any values other than `valid`, then FALSE is returned for that variable; If the variable in `data[vrb.nm]` only has values in `valid`, then TRUE is returned for that variable.

Usage

```
valids_test(data, vrb.nm, valid, na.rm = TRUE)
```

Arguments

`data`	data.frame of data.
`vrb.nm`	character vector of colnames from `data` specifying the variables
`valid`	atomic vector or list vector of valid values.
`na.rm`	logical vector of length 1 specifying whether NA should be ignored from the validity test. If TRUE (default), then any NAs are treated as valid.

Value

logical vector with length = `length(vrb.nm)` and names = `vrb.nm` specifying whether all elements in each variable of `data[vrb.nm]` are valid. If FALSE, then (at least one) invalid values are present in that variable of `data[vrb.nm]`.

See Also

`valid_test` `revalids` `revalid`

Examples

```
valids_test(data = psych::bfi, vrb.nm = names(psych::bfi)[1:25],
   valid = 1:6) # return TRUE
valids_test(data = psych::bfi, vrb.nm = names(psych::bfi)[1:25],
   valid = 0:5) # 6 is not present in `valid`
valids_test(data = psych::bfi, vrb.nm = names(psych::bfi)[1:25],
   valid = 1:6, na.rm = FALSE) # NA is not present in `valid`
valids_test(data = ToothGrowth, vrb.nm = c("supp","dose"),
   valid = list("VC", "OJ", 0.5, 1.0, 2.0)) # list vector as `valid` to allow for
   # elements of different typeof
```

valid_test *Test for Invalid Elements in a Vector*

Description

valid_test tests whether a vector has any invalid elements. Valid values are specified by valid. If the vector x has any values other than valid, then FALSE is returned; If the vector x only has values in valid, then TRUE is returned. This function can be useful for checking data after manual human entry.

Usage

```
valid_test(x, valid, na.rm = TRUE)
```

Arguments

x	atomic vector or list vector.
valid	atomic vector or list vector of valid values.
na.rm	logical vector of length 1 specifying whether NA should be ignored from the validity test. If TRUE (default), then any NAs are treated as valid.

Value

logical vector of length 1 specifying whether all elements in x are valid values. If FALSE, then (at least one) invalid values are present.

See Also

valids_test revalid revalids

Examples

```
valid_test(x = psych::bfi[[1]], valid = 1:6) # return TRUE
valid_test(x = psych::bfi[[1]], valid = 0:5) # 6 is not present in `valid`
valid_test(x = psych::bfi[[1]], valid = 1:6,
   na.rm = FALSE) # NA is not present in `valid`
```

vecNA *Frequency of Missing Values in a Vector*

Description

vecNA computes the frequency of missing values in an atomic vector. vecNA is essentially a wrapper for sum or mean + is.na or !is.na and can be useful for functional programming (e.g., lapply(FUN = vecNA)). It is also used by other functions in the quest package related to missing values (e.g., mean_if).

Usage

```
vecNA(x, prop = FALSE, ov = FALSE)
```

Arguments

x	atomic vector or list vector. If not a vector, it will be coerced to a vector via as.vector.
prop	logical vector of length 1 specifying whether the frequency of missing values should be returned as a proportion (TRUE) or a count (FALSE).
ov	logical vector of length 1 specifying whether the frequency of observed values (TRUE) should be returned rather than the frequency of missing values (FALSE).

Value

numeric vector of length 1 providing the frequency of missing values (or observed values if ov = TRUE). If prop = TRUE, the value will range from 0 to 1. If prop = FALSE, the value will range from 1 to length(x).

See Also

is.na rowNA colNA rowsNA

Examples

```
vecNA(airquality[[1]]) # count of missing values
vecNA(airquality[[1]], prop = TRUE) # proportion of missing values
vecNA(airquality[[1]], ov = TRUE) # count of observed values
vecNA(airquality[[1]], prop = TRUE, ov = TRUE) # proportion of observed values
```

wide2long *Reshape Multiple Sets of Variables From Wide to Long*

Description

wide2long reshapes data from wide to long. This if often necessary to do with multilevel data where multiple sets of variables in the wide format seek to be reshaped to multiple rows in the long format. If only one set of variables needs to be reshaped, then you can use stack2 or melt.data.frame - but that does not work for *multiple* sets of variables. See details for more information.

Usage

```
wide2long(
  data,
  vrb.nm.list,
  grp.nm = NULL,
  sep = ".",
  rtn.obs.nm = "obs",
  order.by.grp = TRUE,
  keep.attr = FALSE
)
```

Arguments

data	data.frame of multilevel data in the wide format.
vrb.nm.list	A unique argument for the quest package such that it can take on different types of inputs. The conventional use is to provide a list of character vectors specifying each set of colnames to be reshaped. In longitudinal panel data, each list element would contain a score with multiple timepoints. The advanced use is to provide a single character vector specifying the colnames to be reshaped (not organized by sets). See details.
grp.nm	character vector specifying the colnames in data corresponding to the groups. Because data is in the wide format, data[grp.nm] must have unique rows (aka groups); if this is not the case, an error is returned. grp.nm can be NULL, in which case the rownames of data will be used. In longitudinal panel data this variable would be the participant ID variable.
sep	character vector of length 1 specifying the string in the column names provided by vrb.nm.list that separates out the name prefix from the number suffix. If sep = "", then that implies there is no string separating the name prefix and the number suffix (e.g., "outcome1").
rtn.obs.nm	character vector of length 1 specifying the new colname in the return object indicating which observation within each group the row refers to. In longitudinal panel data, this would be the returned time variable.
order.by.grp	logical vector of length 1 specifying whether to sort the return object first by grp.nm and then obs.nm (TRUE) or by obs.nm and then grp.nm (FALSE).
keep.attr	logical vector of length 1 specifying whether to keep the "reshapeLong" attribute (from reshape) in the return object.

Details

wide2long uses reshape(direction = "long") to reshape the data. It attempts to streamline the task of reshaping wide to long as the reshape arguments can be confusing because the same arguments are used for wide vs. long reshaping. See reshape if you are curious.

IF vrb.nm.list IS A LIST OF CHARACTER VECTORS: The conventional use of vrb.nm.list is to provide a list of character vectors, which specify each set of variables to be reshaped. For example, if data contains data from a longitudinal panel study with the same scores at different waves, then there might be a column for each score at each wave. vrb.nm.list would then contain an element for each score with each element containing a character vector of the colnames for that score at each wave (see examples). The names of the list elements would then be the colnames in the return object for those scores.

IF vrb.nm.list IS A CHARACTER VECTOR: The advanced use of vrb.nm.list is to provide a single character vector, which specify the variables to be reshaped (not organized by sets). In this case (i.e., if vrb.nm.list is not a list), then wide2long (really reshape) will attempt to guess which colnames go together as a set. It is assumed the following column naming scheme has been used: 1) have the same name prefix for columns within a set, 2) have the same number suffixes for each set of columns, 3) use, *and only use*, sep in the colnames to separate the name prefix and the number suffix. For example, the name prefixes might be "predictor" and "outcome" while the number suffixes might be "0", "1", and "2", and the separator might be ".", resulting in column names such as "outcome.1". The name prefix could include separators other than sep (e.g., "outcome_item.1"), but it cannot include sep (e.g., "outcome.item.1"). So "outcome_item1.1" could be acceptable, but "outcome.item1.1" would not.

Value

data.frame with nrow equal to nrow(data) * length(vrb.nm.list[[1]]) if vrb.nm.list is a list (i.e., conventional use) or nrow(data) * number of unique number suffixes in vrb.nm.list if vrb.nm.list is not a list (i.e., advanced use). The columns will be in the following order: 1) grp.nm of the groups, 2) rtn.obs.nm of the observation labels, 3) the reshaped columns, 4) the additional columns that were not reshaped and instead repeated. How the returned data.frame is sorted depends on order.by.grp.

See Also

long2wide reshape stack2

Examples

```
# SINGLE GROUPING VARIABLE
dat_wide <- data.frame(
   x_1.1 = runif(5L),
   x_2.1 = runif(5L),
   x_3.1 = runif(5L),
   x_4.1 = runif(5L),
   x_1.2 = runif(5L),
   x_2.2 = runif(5L),
   x_3.2 = runif(5L),
   x_4.2 = runif(5L),
   x_1.3 = runif(5L),
```

```
   x_2.3 = runif(5L),
   x_3.3 = runif(5L),
   x_4.3 = runif(5L),
   y_1.1 = runif(5L),
   y_2.1 = runif(5L),
   y_1.2 = runif(5L),
   y_2.2 = runif(5L),
   y_1.3 = runif(5L),
   y_2.3 = runif(5L))
row.names(dat_wide) <- letters[1:5]
print(dat_wide)

# vrb.nm.list = list of character vectors (conventional use)
vrb_pat <- c("x_1","x_2","x_3","x_4","y_1","y_2")
vrb_nm_list <- lapply(X = setNames(vrb_pat, nm = vrb_pat), FUN = function(pat) {
   str2str::pick(x = names(dat_wide), val = pat, pat = TRUE)})
# without `grp.nm`
z1 <- wide2long(dat_wide, vrb.nm = vrb_nm_list)
# with `grp.nm`
dat_wide$"ID" <- letters[1:5]
z2 <- wide2long(dat_wide, vrb.nm = vrb_nm_list, grp.nm = "ID")
dat_wide$"ID" <- NULL

# vrb.nm.list = character vector + guessing (advanced use)
vrb_nm <- str2str::pick(x = names(dat_wide), val = "ID", not = TRUE)
# without `grp.nm`
z3 <- wide2long(dat_wide, vrb.nm.list = vrb_nm)
# with `grp.nm`
dat_wide$"ID" <- letters[1:5]
z4 <- wide2long(dat_wide, vrb.nm = vrb_nm, grp.nm = "ID")
dat_wide$"ID" <- NULL

# comparisons
head(z1); head(z3); head(z2); head(z4)
all.equal(z1, z3)
all.equal(z2, z4)
# keeping the reshapeLong attributes
z7 <- wide2long(dat_wide, vrb.nm = vrb_nm_list, keep.attr = TRUE)
attributes(z7)

# MULTIPLE GROUPING VARIABLES
bfi2 <- psych::bfi
bfi2$"person" <- unlist(lapply(X = 1:400, FUN = rep.int, times = 7))
bfi2$"day" <- rep.int(1:7, times = 400L)
head(bfi2, n = 15)

# vrb.nm.list = list of character vectors (conventional use)
vrb_pat <- c("A","C","E","N","O")
vrb_nm_list <- lapply(X = setNames(vrb_pat, nm = vrb_pat), FUN = function(pat) {
   str2str::pick(x = names(bfi2), val = pat, pat = TRUE)})
z5 <- wide2long(bfi2, vrb.nm.list = vrb_nm_list, grp = c("person","day"),
   rtn.obs.nm = "item")
```

```
# vrb.nm.list = character vector + guessing (advanced use)
vrb_nm <- str2str::pick(x = names(bfi2),
   val = c("person","day","gender","education","age"), not = TRUE)
z6 <- wide2long(bfi2, vrb.nm.list = vrb_nm, grp = c("person","day"),
   sep = "", rtn.obs.nm = "item") # need sep = "" because no character separating
   # scale name and item number
all.equal(z5, z6)
```

`winsor` *Winsorize a Numeric Vector*

Description

`winsor` winsorizes a numeric vector by recoding extreme values as a user-identified boundary value, which is defined by z-score units. The `to.na` argument provides the option of recoding the extreme values as missing.

Usage

```
winsor(x, z.min = -3, z.max = 3, rtn.int = FALSE, to.na = FALSE)
```

Arguments

`x`	numeric vector
`z.min`	numeric vector of length 1 specifying the lower boundary value in z-score units.
`z.max`	numeric vector of length 1 specifying the upper boundary value in z-score units.
`rtn.int`	logical vector of length 1 specifying whether the recoded values should be rounded to the nearest integer. This can be useful when working with count data and decimal values are impossible.
`to.na`	logical vector of length 1 specifying whether the extreme values should be recoded to NA rather than winsorized to the boundary values.

Details

Note, the psych package also has a function called `winsor`, which offers the option to winsorize a numeric vector by quantiles rather than z-scores. If you have both the quest package and the psych package attached in your current R session (e.g., using `library`), depending on which package you attached first, R might default to using the `winsor` function in either the quest package or the psych package. One way to deal with this issue is to explicitly call which package you want to use the `winsor` package from. You can do this using the `::` function in base R where the package name comes before the `::` and the function names comes after it (e.g., `quest::winsor`).

Value

numeric vector of the same length as `x` with extreme values recoded as either the boundary values or NA.

See Also

winsors winsor # psych package

Examples

```
# winsorize
table(quakes$"stations")
new <- winsor(quakes$"stations")
table(new)

# recode as NA
vecNA(quakes$"stations")
new <- winsor(quakes$"stations", to.na = TRUE)
vecNA(new)

# rtn.int = TRUE
winsor(x = cars[[1]], z.min = -2, z.max = 2, rtn.int = FALSE)
winsor(x = cars[[1]], z.min = -2, z.max = 2, rtn.int = TRUE)
```

---

winsors *Winsorize Numeric Data*

Description

winsors winsorizes numeric data by recoding extreme values as a user identified boundary value, which is defined by z-score units. The to.na argument provides the option of recoding the extreme values as missing.

Usage

```
winsors(
  data,
  vrb.nm,
  z.min = -3,
  z.max = 3,
  rtn.int = FALSE,
  to.na = FALSE,
  suffix = "_win"
)
```

Arguments

data	data.frame of data.
vrb.nm	character vector of colnames from data specifying the variables.
z.min	numeric vector of length 1 specifying the lower boundary value in z-score units.
z.max	numeric vector of length 1 specifying the upper boundary value in z-score units.

rtn.int	logical vector of length 1 specifying whether the recoded values should be rounded to the nearest integer. This can be useful when working with count data and decimal values are impossible.
to.na	logical vector of length 1 specifying whether the extreme values should be recoded to NA rather than winsorized to the boundary values.
suffix	character vector of length 1 specifying the string to append to the end of the colnames in the return object.

Value

data.frame of winsorized data with extreme values recoded as either the boundary values or NA and colnames = `paste0(vrb.nm, suffix)`.

See Also

`winsor` `winsor` # psych package

Examples

```
# winsorize
lapply(X = quakes[c("mag","stations")], FUN = table)
new <- winsors(quakes, vrb.nm = names(quakes))
lapply(X = new, FUN = table)

# recode as NA
vecNA(quakes)
new <- winsors(quakes, vrb.nm = names(quakes), to.na = TRUE)
vecNA(new)

# rtn.int = TRUE
winsors(data = cars, vrb.nm = names(cars), z.min = -2, z.max = 2, rtn.int = FALSE)
winsors(data = cars, vrb.nm = names(cars), z.min = -2, z.max = 2, rtn.int = TRUE)
```

Index

.cronbach, 6
.cronbachs, 7
.gtheory, 8, *9*
.gtheorys, *8*, 9

add_sig, 10, *52*
add_sig_cor, 11, *52*, *56*, *58*
agg, 13, *16*, *18*, *68*, *89*, *97*
agg_dfm, *14*, *16*, 17, *26*, *132*, *141*
aggregate, *14*, *16*
aggs, *5*, *14*, 15, *18*, *70*, *87*
alpha, *64–67*
alpha.ci, *65*, *67*
amd_bi, *20*, 20, *21, 22*
amd_multi, *20*, 21, *22*
amd_uni, *21*, 22
aov, *71, 72*, *92, 93*
as.vector, *76*, *196*
auto_by, 23
ave, *14*, *16*, *25, 26*
ave_dfm, 25

boot, *6–9*, *47*
boot.ci, *27*, *47, 48*, *65*, *67*, *84*, *86*
boot_ci, 26
by, *28*
by2, *18*, 28, *94*, *190, 191*

cast, *98*
center, 29, *30*, *32, 33*
center_by, *29, 30*, *32*, 32, *68*, *89*
centers, *5*, *29*, 30, *32, 33*
centers_by, *29, 30*, 31, *33*, *70*, *87*
cfa, *42*, *45*, *103*, *186*, *188, 189*, *192, 193*
change, 33, *34–37*
change_by, *34–36*, 37
changes, *34*, 34, *36, 37*
changes_by, *34, 35*, 35, *37*
chisq.test, *142, 143*, *148*, *151*, *154*, *156*, *159*, *163*
ci.R2, *111*, *123*
cohen.d, *113, 114*, *125, 126*
cohen.d.ci, *113*, *116*, *124*, *128*
colMeans, *38*
colMeans_if, 38, *39*, *41*, *174*, *178*
colNA, *22*, 39, *97*, *175, 176*, *196*
colSums, *41*
colSums_if, *38*, 40, *174*, *178*
complete.cases, *143, 144*
composite, 41, *46*, *65*
composites, *43*, 44, *67*
confint, *46*
confint.boot, *27*
confint2, 46, *47*, *49*
confint2.boot, *46, 47*, 47, *49*, *65*, *67*, *84*, *86*
confint2.default, *46, 47*, 49, *139*
cor, *23, 24*, *51–55*, *59–61*
cor_by, *54*, 58, *61*
cor_miss, 59
cor_ml, *56*, *58*, 61, *88*, *90*
corp, *10, 11*, 50, *54*, *59*
corp_by, *5*, *10, 11*, 52, *58, 59*
corp_miss, 54
corp_ml, *10, 11*, 56, *61*
corr.test, *50*, *52*, *63*
cov, *63, 64*, *66*
covs_test, 62
cronbach, *6*, *43*, 64, *67*, *84, 85*
cronbachs, *46*, *65*, 66, *86, 87*

daply, *18*
ddply, *18*
decompose, 68, *70*
decomposes, *68*, 69
deff, 71, *73*
deffs, *72*, 72
describe, *73, 74*, *135*
describe_ml, 73
dlply, *28*, *190, 191*
dum2nom, 75, *140*

fitMeasures, *42*, *45*, *188*
freq, 76, *79*, *83*, *130*
freq_by, *77*, *79*, *81*, 81, *83*
freqs, *77*, 78, *81*
freqs_by, *77*, *79*, 79, *81*, *83*

gtheory, *8*, 83, *87*, *91*
gtheory_ml, *85*, *89*, 89
gtheorys, *9*, *85*, 85, *89*
gtheorys_ml, *87*, 87, *91*

hist.boot, *47, 48*

ICC, *83–86*, *94*, *96*
icc_11, *72*, 92
icc_all_by, *92, 93*, 94
iccs_11, *73*, 91, *93*
ifelse, *165*
is.dummy, *154*
is.na, *39*, *175, 176*, *196*

lavaan, *103*, *189*, *192, 193*
lavInspect, *192*
lavOptions, *42*, *45*, *186*, *188*, *192, 193*
length, *97*
length_by, *97*, 97
lengths_by, 96, *97*
lme, *23*, *71, 72*, *92, 93*
lmeControl, *24*
lmer, *23*, *71, 72*, *92–94*, *96*
lmerControl, *24*
long2wide, *88*, 98, *198*

make.dummy, 100, *101*
make.dumNA, *60*, *100*, 101
make.fun_if, 102, *127*, *190*
make.latent, 103
make.product, 104
mean.default, *127*
mean_change, *106*, *108, 109*, 118, *121*, *126*, *129*
mean_compare, *111*, 120
mean_diff, *109*, *120, 121*, *123*, 123, *129*
mean_if, *103*, 126, *190*, *196*
mean_test, *117*, *120*, *126*, 128
means_change, 106, *114*, *117*, *120*
means_compare, 109, *123*
means_diff, *5*, *108*, *111*, 112, *114*, *117*, *126*
means_test, *108*, *114*, 115, *129*
melt.data.frame, *197*
mlr, *87–90*
mode, *130*
mode2, 130
model.matrix.default, *140*

n_compare, 142
ncases, 131, *132*, *144*
ncases_by, 132, *135–137*, *141, 142*
ncases_desc, 133
ncases_ml, 135, *137*, *142*
ngrp, 137, *142*
nhst, *49*, 138
nom2dum, *5*, *75*, 139
nrow, *131*, *141*
nrow_by, *132*, *137*, 140, *142*
nrow_ml, *135–137*, 141

oneway.test, *109*, *111*, *120*, *123*

parameterEstimates, *42*, *45*
partial.cases, *131*, *136*, 143
phi, *153*, *161*
pomp, 144, *147*
pomps, *145*, 145
prop.test, *147*, *149, 150*, *153*, *155, 156*, *158, 159*, *161, 162*, *164*
prop_compare, *149*, 156
prop_diff, *143*, *151*, *153*, *158*, 159, *164*
prop_test, *155*, 162
props_compare, 147, *158*
props_diff, *149*, 150, *155*, *161*
props_test, *143*, 153, *164*

quantile, *27*
quest *(*quest-package*)*, 4
quest-package, 4

recode, *165–167*, *172*
recode2other, 164
recodes, *5*, 166, *173*
rename, *168*
renames, 167
reorder.default, *169*
reorders, 169
reshape, *99*, *197, 198*
revalid, 170, *171*, *194, 195*
revalids, *5*, *171*, 171, *194, 195*
reverse, 172, *173*

reverse.code, *172, 173*
reverses, *5*, *167*, *172*, 173
rowMeans, *174*
rowMeans_if, *38*, *41*, 174, *175*, *178, 179*, *181*
rowNA, *5*, *39*, *144*, 175, *176*, *196*
rowsNA, *39*, *175*, 176, *196*
rowSums, *178*
rowSums_if, *38*, *41*, *174*, 177, *179*, *181*

scale.default, *29, 30*, *32, 33*
score, 178, *181*
scoreItems, *179*, *181*
scores, *5*, *179*, 180
sem, *103*
shift, *33, 34*, 181, *184–186*
shift_by, *37*, *68*, *182*, *184, 185*, 185
shifts, *5*, *34, 35*, *182*, 183, *185, 186*
shifts_by, *36*, *70*, *182*, *184*, 184, *186*
stack, *95*
stack2, *197, 198*
statsBy, *56*, *58*, *61*, *88*, *90*
str2str, *5*
sum, *190*
sum_if, *103*, *127*, 189
summary_ucfa, 186, *193*

t.test, *106*, *108*, *112*, *114, 115*, *117, 118*, *120*, *123*, *126*, *128, 129*
table, *76, 77*, *79*, *81–83*, *130*
tapply, *190, 191*
tapply2, *28*, 190
tetrachoric, *151*, *153*, *159*, *161*
try_fun, *94*

ucfa, *189*, 192
unstack2, *90*, *98, 99*

valid_test, *171*, *194*, 195
valids_test, *171*, 194, *195*
vecNA, *5*, *39*, *175, 176*, 196

wide2long, *5*, *99*, 197
winsor, 200, *201, 202*
winsors, *201*, 201

Yule, *153*, *161*