Package 'lsr'

July 22, 2025

Title Companion to ``Learning Statistics with R''

Version 0.5.2

Description A collection of tools intended to make introductory statistics easier to teach, including wrappers for common hypothesis tests and basic data manipulation. It accompanies Navarro, D. J. (2015). Learning Statistics with R: A Tutorial for Psychology Students and Other Beginners, Version 0.6.

License MIT + file LICENSE

URL https://github.com/djnavarro/lsr

BugReports https://github.com/djnavarro/lsr/issues

Encoding UTF-8

LazyLoad yes

Imports graphics, grDevices, methods, stats

RoxygenNote 7.1.1

Suggests testthat (>= 3.0.0)

Config/testthat/edition 3

NeedsCompilation no

Author Danielle Navarro [aut, cre] (ORCID: <https://orcid.org/0000-0001-7648-6578>)

Maintainer Danielle Navarro <djnavarro@protonmail.com>

Repository CRAN

Date/Publication 2021-12-01 08:00:06 UTC

Contents

---

aad *Mean (average) absolute deviation from the mean*

Description

Calculates the mean absolute deviation from the sample mean

Usage

```
aad(x, na.rm = FALSE)
```

Arguments

x	A vector containing the observations.
na.rm	A logical value indicating whether or not missing values should be removed. Defaults to FALSE

Details

The aad function calculates the average (i.e. mean) absolute deviation from the mean value of x, removing NA values if requested by the user. It exists primarily to simplify the discussion of descriptive statistics during an introductory stats class.

Value

Numeric

Examples

```
# basic usage
X <- c(1, 3, 6)  # data
aad(X)           # returns a value of 1.777

# removing missing data
X <- c(1, 3, NA, 6)   # data
aad(X)                # returns NA
aad(X, na.rm = TRUE)  # returns 1.777
```

---

associationTest *Chi-square test of association / independence*

Description

Convenience function that runs a chi-square test of association/independence. This is a wrapper function intended to be used for pedagogical purposes only.

Usage

```
associationTest(formula, data = NULL)
```

Arguments

formula	One-sided formula specifying the two variables (required).
data	Optional data frame containing the variables.

Details

The associationTest function runs the chi-square test of association on the variables specified in the formula argument. The formula must be a one-sided formula of the form ~variable1 + variable2, and both variables must be factors.

Value

An object of class 'assocTest'. When printed, the output is organised into six short sections. The first section lists the name of the test and the variables included. The second lists the null and alternative hypotheses for the test. The third shows the observed contingency table, and the fourth shows the expected contingency table under the null. The fifth prints out the test results, and the sixth reports an estimate of effect size.

Examples

```
df <- data.frame(
gender=factor(c("male","male","male","male","female","female","female")),
answer=factor(c("heads","heads","heads","heads","tails","tails","heads"))
)

associationTest( ~ gender + answer, df )
```

---

bars *Grouped Bar Plots with Error Bars*

Description

Grouped bar plots with error bars

Usage

```
bars(
  formula,
  data = NULL,
  heightFun = mean,
  errorFun = ciMean,
  yLabel = NULL,
  xLabels = NULL,
  main = "",
  ylim = NULL,
  barFillColour = NULL,
  barLineWidth = 2,
  barLineColour = "black",
  barSpaceSmall = 0.2,
  barSpaceBig = 1,
  legendLabels = NULL,
  legendDownShift = 0,
  legendLeftShift = 0,
  errorBarLineWidth = 1,
  errorBarLineColour = "grey40",
  errorBarWhiskerWidth = 0.2
)
```

Arguments

formula	A two-sided formula specifying the response variable and the grouping factors
data	An optional data frame containing the variables
heightFun	The function used to calculate the bar height for a group (default=mean)
errorFun	The function used to calculate the error bar for a group (default=ciMean). No bars drawn if errorFun=FALSE
yLabel	The y-axis label (defaults to the name of the response variable)
xLabels	The x-axis bar labels (defaults to factor labels of the appropriate grouping variable)
main	The plot title
ylim	The y-axis limit: lower bound defaults to 0, default upper bound estimated
barFillColour	The colours to fill the bars (defaults to a rainbow palette with saturation .3)
barLineWidth	The width of the bar border lines (default=2)
barLineColour	The colour of the bar border lines (default="black")
barSpaceSmall	The size of the gap between bars within a cluster, as a proportion of bar width (default=.2)
barSpaceBig	The size of the gap separating clusters of bars, as a proportion of bar width (default=1)
legendLabels	The text for the legend (defaults to factor labels of the appropriate grouping variable). No legends drawn if legendLabels=FALSE or if only one grouping variable is specified
legendDownShift	How far below the top is the legend, as proportion of plot height? (default=0)
legendLeftShift	How far away from the right edge is the legend, as proportion of plot? (default=0)
errorBarLineWidth	The line width for the error bars (default=1)
errorBarLineColour	The colour of the error bars (default="grey40")
errorBarWhiskerWidth	The width of error bar whiskers, as proportion of bar width (default=.2)

Details

Plots group means (or other function, if specified) broken down by one or two grouping factors. Confidence intervals (or other function) are plotted. User specifies a two sided formula of the form response ~ group1 + group2, where response must be numeric and group1 and group2 are factors. The group1 variable defines the primary separation on the x-axis, and the x-axis labels by default print out the levels of this factor. The group2 variable defines the finer grain separation, and the legend labels correspond to the levels of this factor. Note that group2 is optional.

Value

Invisibly returns a data frame containing the factor levels, group means and confidence intervals. Note that this function is usually called for its side effects.

`ciMean` *Confidence interval around the mean*

Description

Calculates confidence intervals for the mean of a normally-distributed variable.

Usage

```
ciMean(x, conf = 0.95, na.rm = FALSE)
```

Arguments

`x`	A numeric vector, data frame or matrix containing the observations.
`conf`	The level of confidence desired. Defaults to a 95% confidence interval
`na.rm`	Logical value indicating whether missing values are to be removed. Defaults to `FALSE`.

Details

This function calculates the confidence interval for the mean of a variable (or set of variables in a data frame or matrix), under the standard assumption that the data are normally distributed. By default it returns a 95% confidence interval (`conf = 0.95`) and does not remove missing values (`na.rm = FALSE`).

Value

The output is a matrix containing the lower and upper ends of the confidence interval for each variable. If a data frame is specified as input and contains non-numeric variables, the corresponding rows in the output matrix have NA values.

Examples

```
X <- c(1, 3, 6)          # data
ciMean(X)                # 95 percent confidence interval
ciMean(X, conf = .8)     # 80 percent confidence interval

confint( lm(X ~ 1) )     # for comparison purposes

X <- c(1, 3, NA, 6)      # data with missing values
ciMean(X, na.rm = TRUE)  # remove missing values
```

cohensD *Cohen's d*

Description

Calculates the Cohen's d measure of effect size.

Usage

```
cohensD(
  x = NULL,
  y = NULL,
  data = NULL,
  method = "pooled",
  mu = 0,
  formula = NULL
)
```

Arguments

x	A numeric variable containing the data for group 1, or possibly a formula of the form `outcome ~ group`
y	If `x` is a numeric variable, the `y` argument should be a numeric variable containing the data for group 2. If a one-sample calculation is desired, then no value for `y` should be specified.
data	If `x` is a formula, then `data` is an optional argument specifying data frame containing the variables in the formula.
method	Which version of the d statistic should we calculate? Possible values are `"pooled"` (the default), `"x.sd"`, `"y.sd"`, `"corrected"`, `"raw"`, `"paired"` and `"unequal"`. See below for specifics.
mu	The "null" value against which the effect size should be measured. This is almost always 0 (the default), so this argument is rarely specified.
formula	An alias for `x` if a formula input is used. Included for the sake of consistency with the `t.test` function.

Details

The `cohensD` function calculates the Cohen's d measure of effect size in one of several different formats. The function is intended to be called in one of two different ways, mirroring the `t.test` function. That is, the first input argument `x` is a formula, then a command of the form `cohensD(x = outcome~group, data = data.frame)` is expected, whereas if `x` is a numeric variable, then a command of the form `cohensD(x = group1, y = group2)` is expected.

The `method` argument allows the user to select one of several different variants of Cohen's d. Assuming that the original t-test for which an effect size is desired was an independent samples t-test (i.e., not one sample or paired samples t-test), then there are several possibilities for how the normalising term (i.e., the standard deviation estimate) in Cohen's d should be calculated. The most

commonly used method is to use the same pooled standard deviation estimate that is used in a Student t-test (`method = "pooled"`, the default). If `method = "raw"` is used, then the same pooled standard deviation estimate is used, except that the sample standard deviation is used (divide by N) rather than the unbiased estimate of the population standard deviation (divide by N-2). Alternatively, there may be reasons to use only one of the two groups to estimate the standard deviation. To do so, use `method = "x.sd"` to select the `x` variable, or the first group listed in the grouping factor; and `method = "y.sd"` to normalise by `y`, or the second group listed in the grouping factor. The last of the "Student t-test" based measures is the unbiased estimator of d (`method = "corrected"`), which multiplies the "pooled" version by (N-3)/(N-2.25).

For other versions of the t-test, there are two possibilities implemented. If the original t-test did not make a homogeneity of variance assumption, as per the Welch test, the normalising term should mirror the Welch test (`method = "unequal"`). Or, if the original t-test was a paired samples t-test, and the effect size desired is intended to be based on the standard deviation of the differences, then `method = "paired"` should be used.

The last argument to `cohensD` is `mu`, which represents the mean against which one sample Cohen's d calculation should be assessed. Note that this is a slightly narrower usage of `mu` than the `t.test` function allows. `cohensD` does not currently support the use of a non-zero `mu` value for a paired-samples calculation.

Value

Numeric variable containing the effect size, d. Note that it does not show the direction of the effect, only the magnitude. That is, the value of d returned by the function is always positive or zero.

References

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

Examples

```
# calculate Cohen's d for two independent samples:
gradesA <- c(55, 65, 65, 68, 70) # 5 students with teacher A
gradesB <- c(56, 60, 62, 66)     # 4 students with teacher B
cohensD(gradesA, gradesB)

# calculate Cohen's d for the same data, described differently:
grade <- c(55, 65, 65, 68, 70, 56, 60, 62, 66) # grades for all students
teacher <- c("A", "A", "A", "A", "A", "B", "B", "B", "B") # teacher for each student
cohensD(grade ~ teacher)

# calculate Cohen's d for two paired samples:
pre  <- c(100, 122, 97, 25, 274) # a pre-treatment measure for 5 cases
post <- c(104, 125, 99, 29, 277) # the post-treatment measure for the same 5 cases
cohensD(pre, post, method = "paired") # ... explicitly indicate that it's paired, or else
cohensD(post - pre)  # ... do a "single-sample" calculation on the difference

# support for data frames:
exams <- data.frame(grade, teacher)
cohensD(exams$grade ~ exams$teacher)    # using $
```

```
cohensD(grade ~ teacher, data = exams)  # using the 'data' argument
```

---

copy *Copies a vector into a matrix*

Description

Copies a vector into a matrix

Usage

```
colCopy(x, times, dimnames = NULL)

rowCopy(x, times, dimnames = NULL)
```

Arguments

x	The vector to be copied
times	Number of copies of the vector to bind together
dimnames	List specifying row and column names

Details

This is a convenience function for binding together multiple copies of the same vector. The intended usage is for situations where one might ordinarily use `rbind` or `cbind`, but the work is done by the `matrix` function. Instead of needing to input multiple copies of the input vector `x` (as one would for `rbind`), one only needs to specify the number of `times` that the vector should be copied.

Value

For `rowCopy`, the output is a matrix with `times` rows and `length(x)` columns, in which each row contains the vector `x`. For `colCopy`, each column corresponds to the vector `x`.

Examples

```
#Example 1: basic usage
data <- c(3,1,4,1,5)
rowCopy( data, 4 )
colCopy( data, 4 )

#Example 2: attach dimension names
dnames <- list( rows = c("r1","r2","r3"), cols = c("c1","c2","c3","c4","c5") )
rowCopy( data,3,dnames )
```

`correlate` *Correlation matrices*

Description

Computes a correlation matrix and runs hypothesis tests with corrections for multiple comparisons

Usage

```
correlate(
  x,
  y = NULL,
  test = FALSE,
  corr.method = "pearson",
  p.adjust.method = "holm"
)
```

Arguments

`x`	Matrix or data frame containing variables to be correlated
`y`	Optionally, a second set of variables to be correlated with those in `x`
`test`	Should hypothesis tests be displayed? (Default=`FALSE`)
`corr.method`	What kind of correlations should be computed? Default is `"pearson"`, but `"spearman"` and `"kendall"` are also supported
`p.adjust.method`	What method should be used to correct for multiple comparisons. Default value is `"holm"`, and the allowable values are the same as for `p.adjust`

Details

The `correlate` function calculates a correlation matrix between all pairs of variables. Much like the `cor` function, if the user inputs only one set of variables (`x`) then it computes all pairwise correlations between the variables in `x`. If the user specifies both `x` and `y` it correlates the variables in `x` with the variables in `y`.

Unlike the `cor` function, `correlate` does not generate an error if some of the variables are categorical (i.e., factors). Variables that are not numeric (or integer) class are simply ignored. They appear in the output, but no correlations are reported for those variables. The decision to have the `correlate` function allow the user a little leniency when the input contains non-numeric variables should be explained. The motivation is pedagogical rather than statistical. It is sometimes the case in psychology that students need to work with correlation matrices before they are comfortable subsetting a data frame, so it is convenient to allow them to type commands like `correlate(data)` even when `data` contains variables for which Pearson/Spearman correlations are not appropriate. (It is also useful to use the output of `correlate` to illustrate the fact that Pearson correlations should not be used for categorical variables).

A second difference between `cor` and `correlate` is that `correlate` runs hypothesis tests for all correlations in the correlation matrix (using the `cor.test` function to do the work). The results of

the tests are only displayed to the user if `test=TRUE`. This is a pragmatic choice, given the (perhaps unfortunate) fact that psychologists often want to see the results of these tests: it is probably not coincidental that the `corr.test` function in the **psych** package already provides this functionality (though the output is difficult for novices to read).

The concern with running hypothesis tests for all elements of a correlation matrix is inflated Type I error rates. To minimise this risk, reported p-values are adjusted using the Holm method. The user can change this setting by specifying `p.adjust.method`. See `p.adjust` for details.

Missing data are handled using pairwise complete cases.

Value

The printed output shows the correlation matrix, and if tests are requested it also reports a matrix of p-values and sample sizes associated with each correlation (these can vary if there are missing data). The underlying data structure is an object of class `correlate` (an S3 class). It is effectively a list containing four elements: `correlation` is the correlation matrix, `p.value` is the matrix of p-values, `sample.size` is the matrix of sample sizes, and `args` is a vector that stores information about what the user requested.

Examples

```
# data frame with factors and missing values
data <- data.frame(
  anxiety = c(1.31,2.72,3.18,4.21,5.55,NA),
  stress = c(2.01,3.45,1.99,3.25,4.27,6.80),
  depression = c(2.51,1.77,3.34,5.83,9.01,7.74),
  happiness = c(4.02,3.66,5.23,6.37,7.83,1.18),
  gender = factor( c("male","female","female","male","female","female") ),
  ssri = factor( c("no","no","no",NA,"yes","yes") )
)

# default output is just the (Pearson) correlation matrix
correlate( data )

# other types of correlation:
correlate( data, corr.method="spearman" )

# two meaningful subsets to be correlated:
nervous <- data[,c("anxiety","stress")]
happy <- data[,c("happiness","depression","ssri")]

# default output for two matrix input
correlate( nervous, happy )

# the same examples, with Holm-corrected p-values
correlate( data, test=TRUE )
correlate( nervous, happy, test=TRUE )
```

cramersV *Cramer's V*

Description

Calculate the Cramer's V measure of association

Usage

```
cramersV(...)
```

Arguments

...	Arguments to be passed to the chisq.test function.

Details

Calculates the Cramer's V measure of effect size for chi-square tests of association and goodness of fit. The arguments to the cramersV function are all passed straight to the chisq.test function, and should have the same format.

Value

A numeric variable with a single element corresponding to the value of V.

Examples

```
# Consider an experiment with two conditions, each with 100
# participants. Each participant chooses between one of three
# options. Possible data for this experiment:

condition1 <- c(30, 20, 50)
condition2 <- c(35, 30, 35)
X <- cbind( condition1, condition2 )
rownames(X) <- c( 'choice1', 'choice2', 'choice3' )
print(X)

# To test the null hypothesis that the distribution of choices
# is identical in the two conditions, we would run a chi-square
# test:
chisq.test(X)

# To estimate the effect size we can use Cramer's V:
cramersV( X )  # returns a value of 0.159
```

`etaSquared` *Effect size calculations for ANOVAs*

Description

Calculates eta-squared and partial eta-squared

Usage

```
etaSquared(x, type = 2, anova = FALSE)
```

Arguments

`x`	An analysis of variance (aov) object.
`type`	What type of sum of squares to calculate?
`anova`	Should the full ANOVA table be printed out in addition to the effect sizes?

Details

Calculates the eta-squared and partial eta-squared measures of effect size that are commonly used in analysis of variance. The input `x` should be the analysis of variance object itself.

For unbalanced designs, the default in `etaSquared` is to compute Type II sums of squares (`type=2`), in keeping with the `Anova` function in the `car` package. It is possible to revert to the Type I SS values (`type=1`) to be consistent with `anova`, but this rarely tests hypotheses of interest. Type III SS values (`type=3`) can also be computed.

Value

If `anova=FALSE`, the output is an M x 2 matrix. Each of the M rows corresponds to one of the terms in the ANOVA (e.g., main effect 1, main effect 2, interaction, etc), and each of the columns corresponds to a different measure of effect size. Column 1 contains the eta-squared values, and column 2 contains partial eta-squared values. If `anova=TRUE`, the output contains additional columns containing the sums of squares, mean squares, degrees of freedom, F-statistics and p-values.

Examples

```
# Example 1: one-way ANOVA

outcome <- c( 1.4,2.1,3.0,2.1,3.2,4.7,3.5,4.5,5.4 )  # data
treatment1 <- factor( c( 1,1,1,2,2,2,3,3,3 ))        # grouping variable
anova1 <- aov( outcome ~ treatment1 )                # run the ANOVA
summary( anova1 )                                    # print the ANOVA table
etaSquared( anova1 )                                 # effect size

# Example 2: two-way ANOVA

treatment2 <- factor( c( 1,2,3,1,2,3,1,2,3 ))      # second grouping variable
anova2 <- aov( outcome ~ treatment1 + treatment2 ) # run the ANOVA
```

```
summary( anova2 )                    # print the ANOVA table
etaSquared( anova2 )                 # effect size
```

`expandFactors` *Expand factors to a set of contrasts*

Description

Substitutes all factors in a data frame with the set of contrasts with which that factor is associated

Usage

```
expandFactors(data, ...)
```

Arguments

`data`	A data frame.
`...`	Additional arguments to be passed to model.matrix

Details

The `expandFactors` function replaces all of the factors in a data frame with the set of contrasts output by the `contrasts` function or `model.matrix`. It may be useful for teaching purposes when explaining relationship between ANOVA and regression.

Value

A data frame.

Examples

```
grading <- data.frame( teacher = factor( c("Amy","Amy","Ben","Ben","Cat") ),
                       gender = factor( c("male","female","female","male","male") ),
                       grade = c(75,80,45,50,65) )

# expand factors using the default contrasts (usually treatment contrasts)
expandFactors( grading )

# specify the contrasts using the contrasts.arg argument to model.matrix
my.contrasts <- list( teacher = "contr.helmert", gender = "contr.treatment" )
expandFactors( grading, contrasts.arg = my.contrasts )
```

goodnessOfFitTest *Chi-square test against specified probabilities*

Description

Convenience function that runs a chi-square goodness of fit test against specified probabilities. This is a wrapper function intended to be used for pedagogical purposes only.

Usage

```
goodnessOfFitTest(x, p = NULL)
```

Arguments

x	Factor variable containing the raw outcomes.
p	Numeric variable containing the null-hypothesis probabilities (default = all outcomes equally likely)

Details

The goodnessOfFitTest function runs the chi-square goodness of fit test of the hypothesis that the outcomes in the factor x were generated according to the probabilities in the vector p. The probability vector p must be a numeric variable of length nlevels(x). If no probabilities are specified, all outcomes are assumed to be equally likely.

Value

An object of class 'gofTest'. When printed, the output is organised into four short sections. The first section lists the name of the test and the variables included. The second lists the null and alternative hypotheses for the test. The third shows the observed frequency table, the expected frequency table under the null hypothesis, and the probabilities specified by the null. The fourth prints out the test results.

See Also

chisq.test, associationTest, cramersV

Examples

```
# raw data
gender <- factor(
  c( "male","male","male","male","female","female",
     "female","male","male","male" ))

# goodness of fit test against the hypothesis that males and
# females occur with equal frequency
goodnessOfFitTest( gender )
```

```
# goodness of fit test against the hypothesis that males appear
# with probability .6 and females with probability .4.
goodnessOfFitTest( gender, p=c(.4,.6) )
goodnessOfFitTest( gender, p=c(female=.4,male=.6) )
goodnessOfFitTest( gender, p=c(male=.6,female=.4) )
```

`importList` *Import a list*

Description

Creates variables in the workspace corresponding to the elements of a list

Usage

```
importList(x, ask = TRUE)
```

Arguments

`x`	List to be imported
`ask`	Should R ask the user to confirm the new variables before creating them? (default is `TRUE`)

Details

The `importList` function creates variables in the parent environment (generally the global workspace) corresponding to each of the elements of the list object `x`. If the names of these elements do not correspond to legitimate variables names they are converted using the `make.names` functions to valid variables names.

Value

Invisibly returns `0` if the user chooses not to import the variables, otherwise invisibly returns `1`.

See Also

`unlist`, `attach`

Examples

```
# data set organised into two groups
data <- c(1,2,3,4,5)
group <- c("group A","group A","group B","group B","group B")

# the split function creates a list with two elements
# named "group A" and "group B", each containing the
# data for the respective groups
data.list <- split( data, group )
```

```
# The data.list variable looks like this:

#   $`group A`
#   [1] 1 2
#
#   $`group B`
#   [1] 3 4 5

# importing the list with the default value of ask = TRUE will
# cause R to wait on the user's approval. Typing this:

#   importList( data.list )

# would produce the following output:

#   Names of variables to be created:
#   [1] "group.A" "group.B"
#   Create these variables? [y/n]

# If the user then types y, the new variables are created.

# this version will silently import the variables.
importList( x = data.list, ask = FALSE )
```

`independentSamplesTTest` *Independent samples t-test*

Description

Convenience function that runs an independent samples t-test. This is a wrapper function intended to be used for pedagogical purposes only.

Usage

```
independentSamplesTTest(
  formula,
  data = NULL,
  var.equal = FALSE,
  one.sided = FALSE,
  conf.level = 0.95
)
```

Arguments

`formula`	Formula specifying the outcome and the groups (required).
`data`	Optional data frame containing the variables.

var.equal	Should the test assume equal variances (default = FALSE).
one.sided	One sided or two sided hypothesis test (default = FALSE)
conf.level	The confidence level for the confidence interval (default = .95).

Details

The independentSamplesTTest function runs an independent-samples t-test and prints the results in a format that is easier for novices to handle than the output of t.test. All the actual calculations are done by the t.test and cohensD functions. The formula argument must be a two-sided formula of the form outcome ~ group. When var.equal=TRUE, a Student's t-test is run and the estimate of Cohen's d uses a pooled estimate of standard deviation. When var.equal=FALSE, the Welch test is used, and the estimate of Cohen's d uses the "unequal" method.

As with the t.test function, the default test is two sided, corresponding to a default value of one.sided = FALSE. To specify a one sided test, the one.sided argument must specify the name of the factor level that is hypothesised (under the alternative) to have the larger mean. For instance, if the outcome for "group2" is expected to be higher than for "group1", then the corresponding one sided test is specified by one.sided = "group2".

Value

An object of class 'TTest'. When printed, the output is organised into five short sections. The first section lists the name of the test and the variables included. The second provides means and standard deviations. The third states explicitly what the null and alternative hypotheses were. The fourth contains the test results: t-statistic, degrees of freedom and p-value. The final section includes the relevant confidence interval and an estimate of the effect size (i.e., Cohen's d)

See Also

t.test, oneSampleTTest, pairedSamplesTTest, cohensD

Examples

```
df <- data.frame(
  rt = c(451, 562, 704, 324, 505, 600, 829),
  cond = factor( x=c(1,1,1,2,2,2,2), labels=c("group1","group2")))

# Welch t-test
independentSamplesTTest( rt ~ cond, df )

# Student t-test
independentSamplesTTest( rt ~ cond, df, var.equal=TRUE )

# one sided test
independentSamplesTTest( rt ~ cond, df, one.sided="group1" )

# missing data
df$rt[1] <- NA
df$cond[7] <- NA
independentSamplesTTest( rt ~ cond, df )
```

longToWide *Reshape from long to wide*

Description

Reshape a data frame from long form to wide form

Usage

```
longToWide(data, formula, sep = "_")
```

Arguments

data	The data frame.
formula	A two-sided formula specifying measure variables and within-subject variables
sep	Separator string used in wide-form variable names

Details

The longToWide function is the companion function to wideToLong. The data argument is a "long form" data frame, in which each row corresponds to a single observation. The output is a "wide form" data frame, in which each row corresponds to a single experimental unit (e.g., a single subject).

The reshaping formula should list all of the measure variables on the left hand side, and all of the within-subject variables on the right hand side. All other variables are assumed to be between-subject variables. For example, if the accuracy of a participant's performance is measured at multiple time points, then the formula would be accuracy ~ time.

Multiple variables are supported on both sides of the formula. For example, suppose we measured the response time rt and accuracy of participants, across three separate days, and across three separate sessions within each day. In this case the formula would be rt + accuracy ~ days + sessions.

Value

The output is a "wide form" data frame in containing one row per subject (or experimental unit, more generally), with each observation of that subject corresponding to a separate variable. The naming scheme for these variables places the name of the measured variable first, followed by the levels of within-subjects variable(s), separated by the separator string sep. In the example above where the reshaping formula was accuracy ~ time, if the default separator of sep="_" was used, and the levels of the time variable are t1, t2 and t3, then the output would include the variables accuracy_t1, accuracy_t2 and accuracy_t3.

In the second example listed above, where the reshaping formula was rt + accuracy ~ days + sessions, the output variables would refer to levels of both within-subjects variables. For instance, rt_day1_session1, and accuracy_day2_session1 might be the names of two of the variables in the wide form data frame.

See Also

wideToLong

Examples

```
long <- data.frame(
  id = c(1, 2, 3, 1, 2, 3, 1, 2, 3),
  time = c("t1", "t1", "t1", "t2", "t2", "t2", "t3", "t3", "t3"),
  accuracy = c(.50, .03, .72, .94, .63, .49, .78, .71, .16)
)

longToWide(long, accuracy ~ time)
```

---

maxFreq *Sample mode*

Description

Calculate the mode of a sample: both modal value(s) and the corresponding frequency

Usage

```
maxFreq(x, na.rm = TRUE)

modeOf(x, na.rm = TRUE)
```

Arguments

x	A vector containing the observations.
na.rm	Logical value indicating whether NA values should be removed.

Details

These two functions can be used to calculate the mode (most frequently observed value) of a sample, and the actual frequency of the modal value. The only complication is in respect to missing data. If na.rm = FALSE, then there are multiple possibilities for how to calculate the mode. One possibility is to treat NA as another possible value for the elements of x, and therefore if NA is more frequent than any other value, then NA is the mode; and the modal frequency is equal to the number of missing values. This is the version that is currently implemented.

Another possibility is to treat NA as meaning "true value unknown", and to the mode of x is itself known only if the number of missing values is small enough that – regardless of what value they have – they cannot alter the sample mode. For instance, if x were c(1,1,1,1,2,2,NA), we know that the mode of x is 1 regardless of what the true value is for the one missing datum; and we know that the modal frequency is between 4 and 5. This is also a valid interpretation, depending on what precisely it is the user wants, but is not currently implemented.

Because of the ambiguity of how na.rm = FALSE should be interpreted, the default value has been set to na.rm = TRUE, which differs from the default value used elsewhere in the package.

Value

The `modeOf` function returns the mode of `x`. If there are ties, it returns a vector containing all values of `x` that have the modal frequency. The `maxFreq` function returns the modal frequency as a numeric value.

See Also

`mean`, `median`, `table`

Examples

```
# simple example
eyes <- c("green","green","brown","brown","blue")
modeOf(eyes)
maxFreq(eyes)

# vector with missing data
eyes <- c("green","green","brown","brown","blue",NA,NA,NA)

# returns NA as the modal value.
modeOf(eyes, na.rm = FALSE)
maxFreq(eyes, na.rm = FALSE)

# returns c("green", "brown") as the modes, as before
modeOf(eyes, na.rm = TRUE)
maxFreq(eyes, na.rm = TRUE)
```

---

oneSampleTTest *One sample t-test*

Description

Convenience function that runs a one sample t-test. This is a wrapper function intended to be used for pedagogical purposes only.

Usage

```
oneSampleTTest(x, mu, one.sided = FALSE, conf.level = 0.95)
```

Arguments

`x`	The variable to be tested (required).
`mu`	The value against which the mean should be tested (required).
`one.sided`	One sided or two sided hypothesis test (default = `FALSE`)
`conf.level`	The confidence level for the confidence interval (default = .95).

Details

The `oneSampleTTest` function runs a one-sample t-test on the data in `x`, and prints the results in a format that is easier for novices to handle than the output of `t.test`. All the actual calculations are done by the `t.test` and `cohensD` functions.

As with the `t.test` function, the default test is two sided, corresponding to a default value of `one.sided = FALSE`. To specify a one sided test in which the alternative hypothesis is that `x` is larger than `mu`, the input must be `one.sided = "greater"`. Similarly, if `one.sided="less"`, then the alternative hypothesis is that the mean of `x` is smaller than `mu`.

Value

An object of class 'TTest'. When printed, the output is organised into five short sections. The first section lists the name of the test and the variables included. The second provides means and standard deviations. The third states explicitly what the null and alternative hypotheses were. The fourth contains the test results: t-statistic, degrees of freedom and p-value. The final section includes the relevant confidence interval and an estimate of the effect size (i.e., Cohen's d).

See Also

`t.test`, `pairedSamplesTTest`, `independentSamplesTTest`, `cohensD`

Examples

```
likert <- c(3,1,4,1,4,6,7,2,6,6,7)

oneSampleTTest( x = likert, mu = 4 )
oneSampleTTest( x = likert, mu = 4, one.sided = "greater" )
oneSampleTTest( x = likert, mu = 4, conf.level=.99 )

likert <- c(3,NA,4,NA,4,6,7,NA,6,6,7)
oneSampleTTest( x = likert, mu = 4 )
```

---

pairedSamplesTTest *Paired samples t-test*

Description

Convenience function that runs a paired samples t-test. This is a wrapper function intended to be used for pedagogical purposes only.

Usage

```
pairedSamplesTTest(
  formula,
  data = NULL,
  id = NULL,
```

```
  one.sided = FALSE,
  conf.level = 0.95
)
```

Arguments

formula	Formula specifying the outcome and the groups (required).
data	Optional data frame containing the variables.
id	The name of the id variable (must be a character string).
one.sided	One sided or two sided hypothesis test (default = FALSE)
conf.level	The confidence level for the confidence interval (default = .95).

Details

The pairedSamplesTTest function runs a paired-sample t-test, and prints the results in a format that is easier for novices to handle than the output of t.test. All the actual calculations are done by the t.test and cohensD functions.

There are two different ways of specifying the formula, depending on whether the data are in wide form or long form. If the data are in wide form, then the input should be a one-sided formula of the form ~ variable1 + variable2. The id variable is not required: the first element of variable1 is paired with the first element of variable2 and so on. Both variable1 and variable2 must be numeric.

If the data are in long form, a two sided formula is required. The simplest way to specify the test is to input a formula of the form outcome ~ group + (id). The term in parentheses is assumed to be the id variable, and must be a factor. The group variable must be a factor with two levels (if there are more than two levels but only two are used in the data, a warning is given). The outcome variable must be numeric.

The reason for using the outcome ~ group + (id) format is that it is broadly consistent with the way repeated measures analyses are specified in the lme4 package. However, this format may not appeal to some people for teaching purposes. Given this, the pairedSamplesTTest also supports a simpler formula of the form outcome ~ group, so long as the user specifies the id argument: this must be a character vector specifying the name of the id variable

As with the t.test function, the default test is two sided, corresponding to a default value of one.sided = FALSE. To specify a one sided test, the one.sided argument must specify the name of the factor level (long form data) or variable (wide form data) that is hypothesised (under the alternative) to have the larger mean. For instance, if the outcome at "time2" is expected to be higher than at "time1", then the corresponding one sided test is specified by one.sided = "time2".

Value

An object of class 'TTest'. When printed, the output is organised into five short sections. The first section lists the name of the test and the variables included. The second provides means and standard deviations. The third states explicitly what the null and alternative hypotheses were. The fourth contains the test results: t-statistic, degrees of freedom and p-value. The final section includes the relevant confidence interval and an estimate of the effect size (i.e., Cohen's d)

See Also

t.test, oneSampleTTest, independentSamplesTTest, cohensD

Examples

```
# long form data frame
df <- data.frame(
  id = factor( x=c(1, 1, 2, 2, 3, 3, 4, 4),
               labels=c("alice","bob","chris","diana") ),
  time = factor( x=c(1,2,1,2,1,2,1,2),
                 labels=c("time1","time2")),
  wm = c(3, 4, 6, 6, 9, 12,7,9)
)

# wide form
df2 <- longToWide( df, wm ~ time )

# basic test, run from long form or wide form data
pairedSamplesTTest( formula= wm ~ time, data=df, id="id" )
pairedSamplesTTest( formula= wm ~ time + (id), data=df )
pairedSamplesTTest( formula= ~wm_time1 + wm_time2, data=df2 )

# one sided test
pairedSamplesTTest( formula= wm~time, data=df, id="id", one.sided="time2" )

# missing data because of NA values
df$wm[1] <- NA
pairedSamplesTTest( formula= wm~time, data=df, id="id" )

# missing data because of missing cases from the long form data frame
df <- df[-1,]
pairedSamplesTTest( formula= wm~time, data=df, id="id" )
```

`permuteLevels` *Permute the levels of a factor*

Description

Apply an arbitrary permutation to the ordering of levels within a factor

Usage

```
permuteLevels(x, perm, ordered = is.ordered(x), invert = FALSE)
```

Arguments

x	The factor to be permuted
perm	A vector specifying the permutation

ordered	Should the output be an ordered factor?
invert	Use the inverse of perm to specify the permutation

Details

This is a convenience function used to shuffle the order in which the levels of a factor are specified. It is similar in spirit to the relevel function, but more general. The relevel function only changes the first level of the factor, whereas permuteLevels can apply an arbitrary permutation. This can be useful for plotting data, because some plotting functions will display the factor levels in the same order that they appear within the factor.

The perm argument is a vector of the same length as levels(x), such that perm[k] is an integer that indicates which of the old levels should be moved to position k. However, if invert=TRUE, the inverse permutation is applied: that is, perm[k] is an integer specifying where to move the k-th level of the original factor. See the examples for more details.

Value

Returns a factor with identical values, but with the ordering of the factor levels shuffled.

See Also

factor, order, relevel

Examples

```
# original factor specifies the levels in order: a,b,c,d,e,f
x <- factor( c(1,4,2,2,3,3,5,5,6,6), labels=letters[1:6] )
print(x)

# apply permutation (5 3 2 1 4 6)... i.e., move 5th factor level (e)
# into position 1, move 3rd factor level (c) into position 2, etc
permuteLevels(x,perm = c(5,3,2,1,4,6))

# apply the inverse of permutation (5 3 2 1 4 6)... i.e., move 1st
# level (a) into position 5, move 2nd level (b) into position 3, etc
permuteLevels(x,perm = c(5,3,2,1,4,6),invert=TRUE)
```

---

posthocPairwiseT *Post-hoc pairwise t-tests for ANOVA*

Description

Performs pairwise t-tests for an analysis of variance, making corrections for multiple comparisons.

Usage

```
posthocPairwiseT(x, ...)
```

Arguments

x	An aov object
...	Arguments to be passed to pairwise.t.test

Details

The intention behind this function is to allow users to use simple tools for multiple corrections (e.g., Bonferroni, Holm) as post hoc corrections in an ANOVA context, using the fitted model object (i.e., an aov object) as the input. The reason for including this function is that Tukey / Scheffe methods for constructing simultaneous confidence intervals (as per TukeyHSD) are not often discussed in the context of an introductory class, and the more powerful tools provided by the multcomp package are not appropriate for students just beginning to learn statistics.

This function is currently just a wrapper function for pairwise.t.test, and it only works for one-way ANOVA, but this may change in future versions.

Value

As per pairwise.t.test

See Also

pairwise.t.test, TukeyHSD

Examples

```
# create the data set to analyse:
dataset <- data.frame(
  outcome = c( 1,2,3, 2,3,4, 5,6,7 ),
  group = factor(c( "a","a","a", "b","b","b","c","c","c"))
)

# run the ANOVA and print out the ANOVA table:
anova1 <- aov( outcome ~ group, data = dataset )
summary(anova1)

# Currently, the following two commands are equivalent:
posthocPairwiseT( anova1 )
pairwise.t.test( dataset$outcome, dataset$group )
```

---

print.assocTest *Print method for lsr chi-square tests*

Description

Print method for lsr chi-square tests

Usage

```
## S3 method for class 'assocTest'
print(x, ...)
```

Arguments

x	An object of class 'assocTest'
...	For consistency with the generic (unused)

Value

Invisibly returns the original object

---

print.correlate *Print method for correlate objects*

Description

Print method for correlate objects

Usage

```
## S3 method for class 'correlate'
print(x, ...)
```

Arguments

x	An object of class 'correlate'
...	For consistency with the generic (unused)

Value

Invisibly returns the original object

print.gofTest *Print method for lsr goodness-of-fit tests*

Description

Print method for lsr goodness-of-fit tests

Usage

```
## S3 method for class 'gofTest'
print(x, ...)
```

Arguments

x	An object of class 'gofTest'
...	For consistency with the generic (unused)

Value

Invisibly returns the original object

print.TTest *Print method for lsr t-tests*

Description

Print method for lsr t-tests

Usage

```
## S3 method for class 'TTest'
print(x, ...)
```

Arguments

x	An object of class 'TTest'
...	For consistency with the generic (unused)

Value

Invisibly returns the original object

print.whoList *Print method for whoList objects*

Description

Print method for whoList objects

Usage

```
## S3 method for class 'whoList'
print(x, ...)
```

Arguments

x	An object of class 'whoList'
...	For consistency with the generic (unused)

Value

Invisibly returns the original object

quantileCut *Cut by quantiles*

Description

Cuts a variable into equal sized categories

It is sometimes convenient (though not always wise) to split a continuous numeric variable x into a set of n discrete categories that contain an approximately equal number of cases. The quantileCut function does exactly this. The actual categorisation is done by the cut function. However, instead of selecting ranges of equal sizes (the default behaviour in cut), the quantileCut function uses the quantile function to select unequal sized ranges so as to ensure that each of the categories contains the same number of observations. The intended purpose of the function is to assist in exploratory data analysis; it is not generally a good idea to use the output of quantileCut function as a factor in an analysis of variance, for instance, since the factor levels are not interpretable and will almost certainly violate homogeneity of variance.

Usage

```
quantileCut(x, n, ...)
```

Arguments

x	A vector containing the observations.
n	Number of categories
...	Additional arguments to cut

Value

A factor containing n levels. The factor levels are determined in the same way as for the cut function, and can be specified manually using the labels argument, which is passed to the cut function.

See Also

cut, quantile

Examples

```
# An example illustrating why care is needed

dataset <- c( 0,1,2, 3,4,5, 7,10,15 )       # note the uneven spread of data
x <- quantileCut( dataset, 3 )              # cut into 3 equally frequent bins
table(x)                                    # tabulate

# For comparison purposes, here is the behaviour of the more standard cut
# function when applied to the same data:
y <- cut( dataset, 3 )
table(y)
```

---

rmAll *Remove all objects*

Description

Removes all objects from the workspace

Usage

```
rmAll(ask = TRUE)
```

Arguments

ask	Logical value indicating whether to ask user to confirm deletions. Default is TRUE

Details

The rmAll function provides a simple way of deleting all objects from the workspace. It is almost equivalent to the usual rm(list = objects()) command. The only difference that it requires the user to confirm the deletions first if ask = TRUE, after displaying a list of the current objects in the worspace. This can occasionally be useful for teaching purposes.

Value

Invisibly returns 0 if no deletions are made, 1 if at least one deletion is made.

See Also

rm

---

sortFrame *Sort a data frame*

Description

Sorts a data frame using one or more variables.

Usage

```
sortFrame(x, ..., alphabetical = TRUE)
```

Arguments

x	Data frame to be sorted
...	A list of sort terms (see below)
alphabetical	Should character vectors be sorted alphabetically?

Details

The simplest use of this function is to sort a data frame x in terms of one or more of the variables it contains. If for instance, the data frame x contains two variables a and b, then the command sortFrame(x,a,b) sorts by variable a, breaking ties using variable b. Numeric variables are sorted in ascending order: to sort in descending order of a and then ascending order of b, use the command sortFrame(x,-a,b). Factors are treated as numeric variables, and are sorted by the internal codes (i.e., the first factor level equals 1, the second factor levels equals 2 and so on). Character vectors are sorted in alphabetical order, which differs from the ordering used by the sort function; to use the default 'ascii' ordering, specify alphabetical=FALSE. Minus signs can be used in conjunction with character vectors in order to sort in reverse alphabetical order. If c represents a character variable, then sortFrame(x,c) sorts in alphabetical order, whereas sortFrame(x,-c) sorts in reverse alphabetical order.

It is also possible to specify more complicated sort terms by including expressions using multiple variables within a single term, but care is required. For instance, it is possible to sort the data frame by the sum of two variables, using the command sortFrame(x, a+b). For numeric variables expressions of this kind should work in the expected manner, but this is not always the case for non-numeric variables: sortFrame uses the xtfrm function to provide, for every variable referred to in the list of sort terms (...) a numeric vector that sorts in the same order as the original variable. This reliance is what makes reverse alphabetical order (e.g., sortFrame(x,-c)) work. However, it also means that it is possible to specify somewhat nonsensical sort terms for character vectors by abusing the numerical coding (e.g. sortFrame(x,(c-3)^2); see the examples section). It also means that sorting in terms of string operation functions (e.g., nchar) do not work as expected. See examples section. Future versions of sortFrame will (hopefully) address this, possibly by allowing the user to "switch off" the internal use of xtfrm, or else by allowing AsIs expressions to be used in sort terms.

Value

The sorted data frame

See Also

sort, order, xtfrm

Examples

```
txt <- c("bob","Clare","clare","bob","eve","eve")
num1 <- c(3,1,2,0,0,2)
num2 <- c(1,1,3,0,3,2)
etc <- c("not","used","as","a","sort","term")
dataset <- data.frame( txt, num1, num2, etc, stringsAsFactors=FALSE )

sortFrame( dataset, num1 )
sortFrame( dataset, num1, num2 )
sortFrame( dataset, txt )
```

---

standardCoefs *Standardised regression coefficients*

Description

Calculates the standardised regression coefficients for a linear model.

Usage

```
standardCoefs(x)
```

Arguments

x	A linear model object (i.e. class lm)

Details

Calculates the standardised regression coefficients (beta-weights), namely the values of the regression coefficients that would have been observed has all regressors and the outcome variable been scaled to have mean 0 and variance 1 before fitting the regression model. Standardised coefficients are sometimes useful in some applied contexts since there is a sense in which all beta values are "on the same scale", though this is not entirely unproblematic.

Value

A matrix with the regressors as rows, and the two different regression coefficients (unstandardised and standardised) as the two columns. The columns are labeled b (unstandardised) and beta (standardised).

Examples

```
# Example 1: simple linear regression

# data
X1 <- c(0.69, 0.77, 0.92, 1.72, 1.79, 2.37, 2.64, 2.69, 2.84, 3.41)
Y  <- c(3.28, 4.23, 3.34, 3.73, 5.33, 6.02, 5.16, 6.49, 6.49, 6.05)

model1 <- lm( Y ~ X1 )  # run a simple linear regression
coefficients( model1 )  # extract the raw regression coefficients
standardCoefs( model1 ) # extract standardised coefficients


# Example 2: multiple linear regression

X2 <- c(0.19, 0.22, 0.95, 0.43, 0.51, 0.04, 0.12, 0.44, 0.38, 0.33)
model2 <- lm( Y ~ X1 + X2 )   # new model
standardCoefs( model2 )       # standardised coefficients

#Example 3: interaction terms

model3 <- lm( Y ~ X1 * X2 )
coefficients( model3 )
standardCoefs( model3 )

# Note that these beta values are equivalent to standardising all
# three regressors including the interaction term X1:X2, not merely
# standardising the two predictors X1 and X2.
```

---

tFrame *Transpose a data frame*

Description

Transposes a data frame, converting variables to cases and vice versa

Usage

```
tFrame(x)
```

Arguments

x	The data frame to be transposed.

Details

The tFrame function is a convenience function that simply transposes the input data frame and coerces the result back to a data frame. Apart from a very small amount of exception handling, it is equivalent to as.data.frame(t(x)). It exists simply because I sometimes find it convenient when

teaching statistics to discuss simple data handling before going into details regarding coercion; similarly, since I generally have students work with data frames before exposing them to matrices, it is convenient to have a transpose function that returns a data frame as output.

Naturally, the `tFrame` function should only be used when it is actually sensible to think of the cases of `x` as variables in their own right. In real life I expect that this maps almost perfectly onto those cases where `x` could be a matrix just as easily as a data frame, so I don't believe that `tFrame` is useful in real world data analysis. It is intended as a teaching tool.

Value

The transposed data frame

See Also

`t`

Examples

```
# Create a data frame that could sensibly be transposed...
Gf <- c(105, 119, 121, 98)   # fluid intelligence for 4 people
Gc <- c(110, 115, 119, 103)  # crystallised intelligence
Gs <- c(112, 102, 108, 99)   # speed of processing
dataset <- data.frame( Gf, Gc, Gs )
rownames(dataset) <- paste( "person", 1:4, sep="" )
print(dataset)

# Now transpose it...
tFrame( dataset )
```

---

unlibrary *Unload a package*

Description

A wrapper function to `detach` that removes a package from the search path, but takes a package name as input similar to `library`.

Usage

```
unlibrary(package)
```

Arguments

package	A package name, which may be specified with or without quotes.

Details

Unloads a package. This is just a wrapper for the detach function. However, the package argument is just the name of the package (rather than the longer string that is required by the detach function), and – like the library function – can be specified without quote marks. The unlibrary function does not unload dependencies, only the named package.

The name "unlibrary" is a bit of an abuse of both R terminology (in which one has a library of packages) and the English language, but I think it helps convey that the goal of the unlibrary function is to do the opposite of what the library function does.

Value

Identical to detach.

See Also

library, require, detach

---

who *Contents of workspace*

Description

Prints out a simple summary of all the objects in the workspace

Usage

```
who(expand = FALSE)
```

Arguments

expand	Should R "expand" data frames when listing variables? If expand = TRUE, variables inside a data frame are included in the output. The default is FALSE

Details

The who function prints out some basic information about all variables in the workspace. Specifically, it lists the names of all variables, what class they are, and how big they are (see below for specifics). If the expand argument is TRUE it will also print out the same information about variables within data frames. See the examples below to see what the output looks like.

The purpose for the function is to show more information than the objects function (especially as regards the names of variables inside data frames), but not to show as much detail as the ls.str function, which is generally too verbose for novice users.

The "size" of an object is only reported for some kinds of object: specifically, only those objects whose mode is either numeric, character, logical, complex or list. Nothing is printed for any other kind of object. If the object has explicit dimensions (e.g., data frames or matrices) then who prints out the dimension sizes (e.g., "2 x 3"). Otherwise the length of the object is printed.

Value

who returns an object of class whoList which is just a data frame with a dedicated print method.

See Also

objects, ls.str

Examples

```
cats <- 4
mood <- "happy"
who()

dataset <- data.frame(
  hi = c( "hello","cruel","world" ),
  pi = c( 3,1,4 )
)

who()
who(expand = TRUE)
```

---

wideToLong *Reshape from wide to long*

Description

Reshape a data frame from wide form to long form using the variable names

Usage

```
wideToLong(data, within = "within", sep = "_", split = TRUE)
```

Arguments

data	The data frame.
within	Name to give to the long-form within-subject factor(s)
sep	Separator string used in wide-form variable names
split	Should multiple within-subject factors be split into multiple variables?

Details

The wideToLong function is the companion function to longToWide. The data argument is a "wide form" data frame, in which each row corresponds to a single experimental unit (e.g., a single subject). The output is a "long form" data frame, in which each row corresponds to a single observation.

The wideToLong function relies on the variable names to determine how the data should be reshaped. The naming scheme for these variables places the name of the measured variable first,

followed by the levels of the within-subjects variable(s), separated by the separator string `sep` (default is _) The separator string cannot appear anywhere else in the variable names: variables without the separator string are assumed to be between-subject variables.

If the experiment measured the `accuracy` of participants at some task at two different points in time, then the wide form data frame would contain variables of the form `accuracy_t1` and `accuracy_t2`. After reshaping, the long form data frame would contain one measured variable called `accuracy`, and a within-subjects factor with levels `t1` and `t2`. The name of the within-subjects factor is the `within` argument.

The function supports experimental designs with multiple within-subjects factors and multi-variable observations. For example, suppose each experimental subject is tested in two `conditions` (`cond1` and `cond2`), on each of two `days` (`day1` and `day2`), yielding an experimental design in which four observations are made for each subject. For each such observation, we record the mean response time `MRT` for and proportion of correct responses `PC` for the participant. The variable names needed for a design such as this one would be `MRT_cond1_day1`, `MRT_cond1_day2`, `PC_cond1_day1`, etc. The `within` argument should be a vector of names for the within-subject factors: in this case, `within = c("condition","day")`.

By default, if there are multiple within-subject factors implied by the existence of multiple separators, the output will keep these as distinct variables in the long form data frame (`split=FALSE`). If `split=TRUE`, the within-subject factors will be collapsed into a single variable.

Value

A data frame containing the reshaped data

See Also

longToWide, reshape

Examples

```
# Outcome measure is mean response time (MRT), measured in two conditions
# with 4 participants. All participants participate in both conditions.

wide <- data.frame( accuracy_t1 = c( .15,.50,.78,.55 ),  # accuracy at time point 1
                    accuracy_t2 = c( .55,.32,.99,.60 ),  # accuracy at time point 2
                    id = 1:4 )                           # id variable

# convert to long form
wideToLong( wide, "time" )


# A more complex design with multiple within-subject factors. Again, we have only
# four participants, but now we have two different outcome measures, mean response
# time (MRT) and the proportion of correct responses (PC). Additionally, we have two
# different repeated measures variables. As before, we have the experimental condition
# (cond1, cond2), but this time each participant does both conditions on two different
# days (day1, day2). Finally, we have multiple between-subject variables too, namely
# id and gender.

wide2 <- data.frame( id = 1:4,
```

```
                      gender = factor( c("male","male","female","female") ),
                      MRT_cond1_day1 = c( 415,500,478,550 ),
                      MRT_cond2_day1 = c( 455,532,499,602 ),
                      MRT_cond1_day2 = c( 400,490,468,502 ),
                      MRT_cond2_day2 = c( 450,518,474,588 ),
                      PC_cond1_day1 = c( 79,83,91,75 ),
                      PC_cond2_day1 = c( 82,86,90,78 ),
                      PC_cond1_day2 = c( 88,92,98,89 ),
                      PC_cond2_day2 = c( 93,97,100,95 ) )

# conversion to long form:
wideToLong( wide2 )
wideToLong( wide2, within = c("condition","day") )

# treat "condition x day" as a single repeated measures variable:
wideToLong( wide2, split = FALSE)
```

Index