Package 'dvmisc'

February 5, 2018

Type Package

Title Faster Computation of Common Statistics and Miscellaneous Functions

Version 1.1.2

Date 2018-02-05

Author Dane R. Van Domelen

Maintainer Dane R. Van Domelen <vandomed@gmail.com>

Description Faster versions of base R functions (e.g. mean, standard deviation, covariance, weighted mean), mostly written in C++, along with miscellaneous functions for various purposes (e.g. create histogram with fitted probability density function or probability mass function curve, create body mass index groups, assess linearity assumption in logistic regression).

License GPL-2

Depends rbenchmark

Imports graphics, MASS, Rcpp (>= 0.12.15), stats, utils

LinkingTo Rcpp

RoxygenNote 6.0.1

NeedsCompilation yes

Repository CRAN

Date/Publication 2018-02-05 22:19:18 UTC

R topics documented:

bmi3 *Convert Continuous BMI Values into 3-Level Factor*

Description

Converts a continuous BMI variable into a 3-level factor variable: Normal weight if `[-Inf, 25)`, Overweight if `[25, 30)`, and Obese if `[30, Inf)`.

Usage

```
bmi3(x, labels = TRUE)
```

Arguments

`x`	Numeric vector of BMI values.
`labels`	If `TRUE`, factor levels are labeled `"Normal weight"`, `"Overweight"`, and `"Obese"`; if `FALSE`, factor levels are `[-Inf, 25)`, `[25, 30)`, and `[30, Inf)`.

Value

Factor variable with 3 levels.

bmi4 *Convert Continuous BMI Values into 4-Level Factor*

Description

Converts a continuous BMI variable into a 4-level factor variable: Underweight if `[-Inf, 18.5)`, Normal weight if `[18.5, 25)`, Overweight if `[25, 30)`, and Obese if `[30, Inf)`.

Usage

```
bmi4(x, labels = TRUE)
```

Arguments

`x`	Numeric vector of BMI values.
`labels`	If `TRUE`, factor levels are labeled `"Underweight"`, `"Normal weight"`, `"Overweight"`, and `"Obese"`; if `FALSE`, factor levels are `[-Inf, 18.5)`, `[18.5, 25)`, `[25, 30)`, and `[30, Inf)`.

Value

Factor variable with 4 levels.

cov_i *Sample Covariance for Integer Vectors*

Description

Written in C++, this function should always run faster than cov for integer vectors. Will give incorrect result for non-integer vectors, and does not check that x and y are same length or that they contain no missing values.

Usage

```
cov_i(x, y)
```

Arguments

x, y	Integer vector.

Value

Numeric value.

Examples

```
# For integer vectors, cov_i is typically much faster than cov.
x <- rpois(1000, lambda = 5)
y <- rpois(1000, lambda = 5)
all.equal(cov(x, y), cov_i(x, y))
benchmark(cov(x, y), cov_i(x, y), replications = 5000)
```

cov_n *Sample Covariance for Numeric Vectors*

Description

Written in C++, this function should always run faster than cov for numeric vectors. For integer vectors, cov_i should run even faster.

Usage

```
cov_n(x, y)
```

Arguments

x, y	Numeric vector.

Value

Numeric value.

Examples

```
# In general, cov_n is much faster than cov
x <- rnorm(1000)
y <- rnorm(1000)
all.equal(cov(x, y), cov_n(x, y))
benchmark(cov(x, y), cov_n(x, y), replications = 5000)

# For integer vectors, cov_i should be even faster.
x <- rpois(1000, lambda = 5)
y <- rpois(1000, lambda = 5)
all.equal(cov(x, y), cov_i(x, y))
benchmark(cov(x, y), cov_n(x, y), cov_i(x, y), replications = 5000)
```

---

diff1_i *1-Unit Lagged Differences for Integer Values*

Description

Written in C++, this function should always run faster than `diff` for calculating differences between adjacent values of an integer vector.

Usage

```
diff1_i(x)
```

Arguments

x	Integer vector.

Value

Integer vector.

Examples

```
# diff1_i is typically much faster than diff
x <- rpois(1000, lambda = 5)
all.equal(diff(x), diff1_i(x))
benchmark(diff(x), diff1_i(x), replications = 2000)
```

diff1_n *1-Unit Lagged Differences for Numeric Values*

Description

Written in C++, this function should always run faster than `diff` for calculating differences between adjacent values of a numeric vector. For integer vectors, `diff1_i` should run even faster.

Usage

```
diff1_n(x)
```

Arguments

x	Numeric vector.

Value

Numeric vector.

Examples

```
# In general, diff1_n is much faster than diff
x <- rnorm(1000)
all.equal(diff(x), diff1_n(x))
benchmark(diff(x), diff1_n(x), replications = 3000)

# For integer vectors, diff1_i should be even faster
x <- rpois(1000, lambda = 5)
all.equal(diff(x), diff1_i(x))
benchmark(diff(x), diff1_n(x), diff1_i(x), replications = 3000)
```

diff_i *Lagged Differences for Integer Values*

Description

Written in C++, this function should always run faster than `diff` for calculating lagged differences for an integer vector.

Usage

```
diff_i(x, lag = 1L)
```

Arguments

x	Integer vector.
lag	Integer value.

Value

Integer vector.

Examples

```
# diff_i is typically much faster than diff
x <- rpois(1000, lambda = 5)
all.equal(diff(x, 2), diff_i(x, 2))
benchmark(diff(x, 2), diff_i(x, 2), replications = 2000)
```

---

diff_n *Lagged Differences for Numeric Values*

Description

Written in C++, this function should always run faster than `diff` for calculating lagged differences for a numeric vector. For integer vectors, `diff_i` should run even faster. even faster.

Usage

```
diff_n(x, lag = 1L)
```

Arguments

x	Numeric vector.
lag	Integer value.

Value

Numeric vector.

Examples

```
# In general, diff_n is much faster than diff
x <- rnorm(1000)
all.equal(diff(x, 2), diff_n(x, 2))
benchmark(diff(x, 2), diff_n(x, 2), replications = 2000)

# For integer vectors, diff_i should be even faster
x <- rpois(1000, lambda = 5)
all.equal(diff(x, 2), diff_i(x, 2))
benchmark(diff(x, 2), diff_n(x, 2), diff_i(x, 2), replications = 2000)
```

dots_bars *Plot Points +/- Error Bars*

Description

Creates plot showing user-specified points (e.g. means, medians, regression coefficients) along with user-specified error bars (e.g. standard deviations, min/max, 95% confidence intervals).

Usage

```
dots_bars(y = NULL, bars = NULL, bars.lower = y - bars, bars.upper = y +
  bars, group.labels = NULL, subgroup.labels = NULL, subgroup.pch = NULL,
  subgroup.col = NULL, points.list = NULL, arrows.list = NULL,
  axis.list = NULL, legend.list = NULL, ...)
```

Arguments

y	Numeric vector of y-values for different groups, or numeric matrix where each column contains y-values for clustered subgroups within a group.
bars	Numeric vector or matrix (matching whichever type y is) specifying the length of the error bar for each group/subgroup (i.e. distance from point to one end of error bar).
bars.lower	Numeric vector or matrix (matching whichever type y is) specifying the position of the lower end of the error bar for each group/subgroup.
bars.upper	Numeric vector or matrix (matching whichever type y is) specifying the position of the upper end of the error bar for each group/subgroup.
group.labels	Character vector giving labels for the groups.
subgroup.labels	Character vector giving labels for the subgroups.
subgroup.pch	Plotting symbol for different subgroups within each group.
subgroup.col	Plotting color for different subgroups within each group.
points.list	Optional list of inputs to pass to points function.
arrows.list	Optional list of inputs to pass to arrows function.
axis.list	Optional list of inputs to pass to axis function.
legend.list	Optional list of inputs to pass to legend function.
...	Additional arguments to pass to plot function.

Value

Plot showing points +/- error bars across groups/subgroups.

Examples

```
# Generate 100 values from normal distributions with different means, and
# graph mean +/- standard deviation across groups
dat <- cbind(rnorm(100, 2), rnorm(100, 2.5), rnorm(100, 1.75))
means <- apply(dat, 2, mean)
sds <- apply(dat, 2, sd)
fig1 <- dots_bars(y = means, bars = sds, main = "Mean +/- SD by Group",
                  ylab = "Mean +/- SD")

# Simulate BMI values for males and females in 3 different age groups, and
# graph mean +/- 95\% CI
sex <- as.factor(c(rep("Male", 300), rep("Female", 300)))
age <- as.factor(rep(c("Young", "Middle", "Old"), 2))
bmi <- c(rnorm(100, 25, 4), rnorm(100, 26, 4.25), rnorm(100, 27, 4.5),
         rnorm(100, 26.5, 4.5), rnorm(100, 27.25, 4.75), rnorm(100, 28, 5))
dat <- data.frame(sex = sex, age = age, bmi = bmi)
means <- tapply(dat$bmi, dat[, c("sex", "age")], mean)
ci.lower <- tapply(dat$bmi, dat[, c("sex", "age")],
                   function(x) t.test(x)$conf.int[1])
ci.upper <- tapply(dat$bmi, dat[, c("sex", "age")],
                   function(x) t.test(x)$conf.int[2])
fig2 <- dots_bars(y = means, bars.lower = ci.lower, bars.upper = ci.upper,
                  main = "BMI by Sex and Age",
                  ylab = "BMI (mean +/- CI)",
                  xlab = "Age group")
```

`dvmisc` *Faster Computation of Common Statistics and Miscellaneous Functions*

Description

Faster versions of base R functions (e.g. mean, standard deviation, covariance, weighted mean), mostly written in C++, along with miscellaneous functions for various purposes (e.g. create histogram with fitted probability density function or probability mass function curve, create body mass index groups, assess linearity assumption in logistic regression).

Details

Package:	dvmisc
Type:	Package
Version:	1.1.2
Date:	2018-02-05
License:	GPL-2

See CRAN documentation for full list of functions.

Author(s)

Dane R. Van Domelen
<vandomed@gmail.com>

References

Acknowledgment: This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-0940903.

`get_mse` *Extract Mean Squared Error (MSE) from Fitted Regression Model*

Description

The MSE, defined as the sum of the squared residuals divided by `n-p` (`n` = number of observations, `p` = number of regression coefficients), is an unbiased estimator for the error variance in a linear regression model. This is a convenience function that extracts the MSE from a fitted `lm` or `glm` object. The code is `rev(anova(model.fit)$"Mean Sq")[1]` if `model.fit` is a `lm` object and `sum(model.fit$residuals^2) / model.fit$df.residual` if `model.fit` is a `glm` object.

Usage

```
get_mse(model.fit, var.estimate = FALSE)
```

Arguments

`model.fit`	Fitted regression model returned from `lm` or `glm`.
`var.estimate`	If `TRUE`, function returns a variance estimate for the error variance, defined as `2 * MSE^2 / (n - p)`.

Value

If `var.estimate = FALSE`, numeric value indicating the MSE; if `var.estimate = TRUE`, named numeric vector indicating both the MSE and a variance estimate for the error variance.

Examples

```
# Generate 100 values: Y = 0.5 + 1.25 X + e, e ~ N(0, 1)
set.seed(123)
x <- rnorm(100)
y <- 0.5 + 1.25 * x + rnorm(100, sd = 1)

# Fit regression model using lm and using glm
lm.fit <- lm(y ~ x)
glm.fit <- glm(y ~ x)

# Extract MSE from lm.fit and glm.fit
get_mse(lm.fit)
```

```
get_mse(glm.fit)
```

`headtail` *Return the First and Last Part of an Object*

Description

Simply `head` and `tail` combined into one function.

Usage

```
headtail(x, ...)
```

Arguments

`x`	Input object.
`...`	Additional arguments to pass to `head` and `tail` functions.

Value

Same class as `x`.

Examples

```
# Generate data from N(0, 1), sort, and look at smallest and largest 3 values
x <- rnorm(1000)
x.sorted <- sort(x)
headtail(x.sorted, 3)
```

`histo` *Histogram with Added Options*

Description

Similar to base R function `hist`, but with two added features: (1) Can overlay a fitted probability density/mass function (PDF/PMF) for any univariate distribution supported in R (see `Distributions`); and (2) Can generate more of a barplot type histogram, where each possible value gets its own bin centered over its value (useful for discrete variables with not too many possible values).

Usage

```
histo(x, dis = "none", dis.shift = NULL, integer.breaks = NULL,
  points.list = NULL, axis.list = NULL, ...)
```

Arguments

x	Numeric vector of values.
dis	Character vector indicating which distribution should be used to add fitted PDF/PMF to the histogram. Possible values are "none", "beta", "binom" (must specify size), "cauchy", "chisq", "exp", "f", "gamma", "geom", "hyper" (must specify total number of balls in urn, N, and number of balls drawn each time, k), "lnorm", "nbinom" (must specify size), "norm", "pois", "t", "unif", and "weibull".
dis.shift	Numeric value for shifting the fitted PDF/PMF along the x-axis of the histogram.
integer.breaks	If TRUE, integers covering the range of x are used for breaks, so there is one bin for each integer. Useful for discrete distributions that don't take on too many unique values.
points.list	Optional list of inputs to pass to points function, which is used to add the fitted PDF/PMF.
axis.list	Optional list of inputs to pass to axis function.
...	May include arguments to pass to hist and/or parameter values needed for certain distributions (size if dis = "binom" or dis = "nbinom", N and k if dis = "hyper").

Details

When x takes on whole numbers, you typically want to set dis.shift = -0.5 if right = TRUE (hist's default) and dis.shift = 0.5 if right = FALSE. The function will do this internally by default.

To illustrate, suppose a particular bin represents (7, 10]. Its midpoint will be at x = 8.5 on the graph. But if input values are whole numbers, this bin really only includes values of 8, 9, and 10, which have a mean of 9. So you really want f(9) to appear at x = 8.5. This requires shifting the curve to the left 0.5 units, i.e. setting dis.shift = -0.5.

When x takes on whole numbers with not too many unique values, you may want the histogram to show one bin for each integer. You can do this by setting integer.breaks = TRUE. By default, the function sets integer.breaks = TRUE if x contains whole numbers with 10 or fewer unique values.

Value

Histogram with fitted PDF/PMF if requested.

Examples

```
# Generate 10,000 Poisson(2) values. Compare default histograms from hist vs.
# histo.
set.seed(123)
x <- rpois(n = 10000, lambda = 2)
par(mfrow = c(1, 2))
hist(x)
histo(x)
```

```
# Generate 10,000 lognormal(0, 0.35) values. Create histograms with curves
# showing fitted log-normal and normal PDFs.
set.seed(123)
x <- rlnorm(n = 10000, meanlog = 0, sdlog = 0.35)
par(mfrow = c(1, 2))
histo(x, "lnorm", main = "Log-normal curve")
histo(x, "norm", main = "Normal curve")

# Generate 10,000 Binomial(8, 0.25) values. Create histogram, specifying
# size = 5, with blue line/points showing fitted PMF.
set.seed(123)
x <- rbinom(n = 10000, size = 5, prob = 0.25)
par(mfrow = c(1, 1))
histo(x, "binom", size = 5, points.list = list(type = "b", col = "blue"))
```

`inside` *Check Whether Numeric Value Falls Inside Two Other Numeric Values*

Description

Returns `TRUE` if `x` falls inside range defined by `ends` and `FALSE` otherwise. Also works for multiple sets of values and/or endpoints.

Usage

```
inside(x, ends, inclusive = TRUE)
```

Arguments

`x`	Numeric value or vector of numeric values.
`ends`	Numeric vector of length 2 specifying the endpoints for the interval, or a 2-column numeric matrix where each row specifies a pair of endpoints.
`inclusive`	Logical value indicating whether endpoints should be included.

Value

Logical value or vector.

Examples

```
# Check whether 2 is inside [0, 2.5]
inside(1, c(0, 2.5))

# Check whether 2 and 3 are inside (0, 3)
inside(c(2, 3), c(0, 3), inclusive = FALSE)

# Check whether 1 is inside [1, 2] and [3, 4]
inside(1, rbind(c(1, 2), c(3, 4)))
```

`interval_groups` *Split Continuous Variable into Equal-Width Groups*

Description

Splits a continuous variable into equal-width groups. Useful for assessing linearity in regression models.

Usage

```
interval_groups(x, groups = 5, ...)
```

Arguments

`x`	Numeric vector.
`groups`	Numeric value indicating how many groups should be created.
`...`	Further arguments to pass to `cut`.

Value

Factor variable.

See Also

`cut`

Examples

```
# Convert values from N(0, 1) into 6 equal-width groups
x <- rnorm(1000)
groups <- interval_groups(x, 6)
table(groups)

# Use interval_groups to detect non-linearity
set.seed(123)
x <- rnorm(1000)
y <- 1.5 + 1.25 * x + 0.25 * x^2 + rnorm(1000)
plot(tapply(y, interval_groups(x), mean))
```

`list_override` *Add Elements of Second List to First List, Replacing Elements with Same Name*

Description

Adds each element of `list2` to `list1`, overriding any elements of the same name. Similar to `modifyList` function in **utils** package, but either list can be `NULL`. Useful for `do.call` statements, when you want to combine a list of default inputs with a list of user-specified inputs.

Usage

```
list_override(list1, list2)
```

Arguments

`list1`	Initial list that has some number of named elements. Can be `NULL` or an empty list.
`list2`	List with named elements that will be added to `list1`, replacing any elements with the same name. Can be `NULL` or an empty list.

Value

List containing the named elements initially in `list1` and not in `list2`, any additional named elements in `list2`, and any named elements in `list1` that were replaced by elements of the same name in `list2`.

Examples

```
# Create list that has default inputs to the plot function
list.defaults <- list(x = 1: 5, y = 1: 5, type = "l", lty = 1)

# Create list of user-specified inputs to the plot function
list.user <- list(main = "A Straight Line", lty = 2, lwd = 1.25)

# Combine the two lists into one, giving priority to list.user
list.combined <- list_override(list.defaults, list.user)

# Plot data using do.call
do.call(plot, list.combined)
```

logit_prob *Convert Logit to Probability*

Description

Defined as: `exp_x <- exp(x); out <- exp_x / (1 + exp_x)`. This 2-step approach is faster than `exp(x) / (1 + exp(x))` because the exponentials only have to be calculated once.

Usage

```
logit_prob(x)
```

Arguments

x	Numeric vector.

Value

Numeric vector.

logodds_graph *Graph Log-Odds of Binary Variable Across A Grouping Variable*

Description

Creates plot showing sample log-odds of binary Y variable across levels of a grouping variable, with customizable error bars. Observations with missing values for `y` and/or `group` are dropped.

Usage

```
logodds_graph(y, group, error.bars = "none", alpha = 0.05,
  p.legend = "chi", plot.list = NULL, lines.list = NULL,
  axis.list = NULL, legend.list = NULL, ...)
```

Arguments

y	Vector of values for binary response variable. Must take on 2 values, but can be any type (e.g. numeric, character, factor, logical). Function plots log-odds of second value returned by `table(y)`.
group	Vector of values indicating what group each `y` observation belongs to. Function plots group levels across x-axis in same order as `table(group)`.
error.bars	Character string indicating what the error bars should represent. Possible values are `"exact.ci"` for exact 95% confidence interval based on binomial distribution, `"z.ci"` for approximate 95% confidence interval based on Z distribution, and `"none"` for no error bars.

alpha	Numeric value indicating what alpha should be set to for confidence intervals. Only used if error.bars is "exact.ci" or "z.ci".
p.legend	Character string controlling what p-value is printed in a legend. Possible values are "chi" for Chi-square test of association, "fisher" for Fisher's exact test, and "none" for no legend at all.
plot.list	Optional list of inputs to pass to plot function.
lines.list	Optional list of inputs to pass to lines function.
axis.list	Optional list of inputs to pass to axis function.
legend.list	Optional list of inputs to pass to legend function.
...	Additional arguments to pass to chisq.test or fisher.test functions.

Value

Plot showing log-odds of y across levels of group.

---

max_n *Maximum of Numeric Values*

Description

Written in C++, this function tends to run faster than max for large numeric vectors/matrices.

Usage

```
max_n(x)
```

Arguments

x	Numeric vector.

Value

Numeric value.

Examples

```
# For large objects, max_n is faster than max
x <- rnorm(100000)
max(x) == max_n(x)
benchmark(max(x), max_n(x), replications = 1000)

# For smaller objects, max_n is slower than max
x <- rnorm(100)
max(x) == max_n(x)
benchmark(max(x), max_n(x), replications = 1000)
```

means_graph *Graph Means Across a Grouping Variable*

Description

Creates plot showing mean of Y variable across levels of a grouping variable, with customizable error bars. Observations with missing values for y and/or group are dropped.

Usage

```
means_graph(y, group, error.bars = "t.ci", alpha = 0.05, p.legend = TRUE,
  plot.list = NULL, lines.list = NULL, axis.list = NULL,
  legend.list = NULL, ...)
```

Arguments

y	Numeric vector of values for the continuous variable.
group	Vector of values indicating what group each y observation belongs to. Function plots group levels across x-axis in same order as table(group).
error.bars	Character string indicating what the error bars should represent. Possible values are "sd" for +/- one standard deviation, "se" for +/- one standard error, "t.ci" for 95% confidence interval based on t distribution, "z.ci" for 95% confidence interval based on Z distribution, and "none" for no error bars.
alpha	Numeric value indicating what alpha should be set to for confidence intervals. Only used if error.bars is "t.ci" or "z.ci".
p.legend	If TRUE, p-value (from t.test function if group has 2 levels, otherwise aov function) is printed in a legend.
plot.list	Optional list of inputs to pass to plot function.
lines.list	Optional list of inputs to pass to lines function.
axis.list	Optional list of inputs to pass to axis function.
legend.list	Optional list of inputs to pass to legend function.
...	Additional arguments to pass to t.test or aov.

Value

Plot showing mean of y across levels of group.

mean_i *Mean of Integer Values*

Description

Written in C++, this function should always run faster than `mean` for integer vectors/matrices. Not valid for non-integer objects.

Usage

```
mean_i(x)
```

Arguments

x	Integer vector or matrix.

Value

Numeric value.

Examples

```
# For integer objects, mean_i is typically much faster than mean.
x <- rpois(100, lambda = 5)
mean(x) == mean_i(x)
benchmark(mean(x), mean_i(x), replications = 10000)
```

mean_n *Mean of Numeric Values*

Description

Defined simply as `sum(x) / length(x)`, this function seems to always run faster than `mean` for numeric, non-integer vectors/matrices. For integer objects, `mean_i` should run even faster.

Usage

```
mean_n(x)
```

Arguments

x	Numeric vector or matrix.

Value

Numeric value.

Examples

```
# In general, mean_n is much faster than mean.
x <- rnorm(10000)
mean(x) == mean_n(x)
benchmark(mean(x), mean_n(x), replications = 1000)

# For very large integer objects, mean may be faster than mean_n. But then
# mean_i should be even faster.
x <- rpois(100000, lambda = 5)
mean(x) == mean_n(x)
mean(x) == mean_i(x)
benchmark(mean(x), mean_n(x), mean_i(x), replications = 1000)
```

---

min_n *Minimum of Numeric Values*

Description

Written in C++, this function tends to run faster than `min` for large numeric vectors/matrices.

Usage

```
min_n(x)
```

Arguments

x	Numeric vector.

Value

Numeric value.

Examples

```
# For large objects, min_n is faster than min
x <- rnorm(100000)
min(x) == min_n(x)
benchmark(min(x), min_n(x), replications = 1000)

# For smaller objects, min_n is slower than min
x <- rnorm(100)
min(x) == min_n(x)
benchmark(min(x), min_n(x), replications = 20000)
```

odds_prob *Convert Odds to Probability*

Description

Defined simply as `log(x / (x + 1))`.

Usage

```
odds_prob(x)
```

Arguments

x	Numeric vector.

Value

Numeric vector.

pooled_var_i *Pooled Sample Variance for Integer Vectors*

Description

Calculates pooled sample variance used in equal variance two-sample t-test. Runs faster than `pooled_var_n` when x and y are integer vectors, but not valid if x or y are non-integer vectors.

Usage

```
pooled_var_i(x, y)
```

Arguments

x	Integer vector.
y	Integer vector.

Value

Numeric value.

pooled_var_n *Pooled Sample Variance for Numeric Vectors*

Description

Calculates pooled sample variance used in equal variance two-sample t-test. For integer vectors, `pooled_var_i` will run faster.

Usage

```
pooled_var_n(x, y)
```

Arguments

x	Numeric vector.
y	Numeric vector.

Value

Numeric value.

prob_logit *Convert Probability to Logit*

Description

Defined simply as `log(x / (1 - x))`.

Usage

```
prob_logit(x)
```

Arguments

x	Numeric vector.

Value

Numeric vector.

prob_odds *Convert Probability to Odds*

Description

Defined simply as `x / (1 - x)`.

Usage

```
prob_odds(x)
```

Arguments

x	Numeric vector.

Value

Numeric vector.

quant_groups *Split Continuous Variable into Quantile Groups*

Description

Splits a continuous variable into quantiles groups. Basically combines `quantile` and `cut` into a single function.

Usage

```
quant_groups(x, groups = 5, ...)
```

Arguments

x	Numeric vector.
groups	Numeric value indicating how many quantile groups should be created.
...	Further arguments to pass to `quantile` or `cut`.

Value

Factor variable.

Examples

```
# Convert values from N(0, 1) into quintiles (i.e. 5 groups)
x <- rnorm(1000)
groups <- quant_groups(x, 5)
table(groups)
```

range_i *Range (Actually Minimum and Maximum) of Integer Values*

Description

Written in C++, this function should always run faster than `range` for integer vectors/matrices. Not valid for non-integer objects.

Usage

```
range_i(x)
```

Arguments

x	Integer vector or matrix.

Value

Integer vector.

Examples

```
# In general, range_i is much faster than range
x <- rpois(1000, lambda = 5)
all.equal(range(x), range_i(x))
benchmark(range(x), range_i(x), replications = 10000)
```

range_n *Range (Actually Minimum and Maximum) of Numeric Values*

Description

Written in C++, this function should always run faster than `range` for numeric vectors/matrices. For integer objects, `range_i` should run even faster.

Usage

```
range_n(x)
```

Arguments

x	Numeric vector or matrix.

Value

Numeric vector.

Examples

```
# In general, range_n is much faster than range
x <- rnorm(1000)
all.equal(range(x), range_n(x))
benchmark(range(x), range_n(x), replications = 5000)

# For integer vectors, range_i should be even faster
x <- rpois(1000, lambda = 5)
all.equal(range(x), range_i(x))
benchmark(range(x), range_n(x), range_i(x), replications = 10000)
```

---

sd_i *Sample Standard Deviation for Integer Values*

Description

Written in C++, this function should always run faster than sd for integer vectors. Not valid for non-integer vectors.

Usage

```
sd_i(x)
```

Arguments

x	Integer vector.

Value

Numeric value.

Examples

```
# For integer vectors, var_i is typically much faster than var.
x <- rpois(1000, lambda = 5)
all.equal(sd(x), sd_i(x))
benchmark(sd(x), sd_i(x), replications = 2000)
```

sd_n *Sample Standard Deviation for Numeric Values*

Description

Written in C++, this function should always run faster than sd for numeric vectors. For integer vectors, sd_i should run even faster.

Usage

```
sd_n(x)
```

Arguments

x	Numeric vector.

Value

Numeric value.

Examples

```
# In general, sd_n is much faster than sd.
x <- rnorm(1000)
all.equal(sd(x), sd_n(x))
benchmark(sd(x), sd_n(x), replications = 2000)

# For integer vectors, sd_i should be even faster.
x <- rpois(1000, lambda = 5)
all.equal(sd(x), sd_i(x))
benchmark(sd(x), sd_n(x), sd_i(x), replications = 2000)
```

sumsim *Summarize Simulation Results*

Description

Creates table summarizing results of statistical simulations, providing common metrics of performance like mean bias, standard deviation, mean standard error, mean squared error, and confidence interval coverage.

Usage

```
sumsim(estimates, ses = NULL, truth = NULL, statistics = c("mean_bias",
  "sd", "mean_se", "mse", "coverage"), alpha = 0.05, digits = 3)
```

Arguments

estimates	Numeric matrix where each column gives the point estimates for a particular method across multiple trials.
ses	Numeric matrix where each column gives the standard errors for a particular method across multiple trials.
truth	Numeric value specifying the true value of the parameter being estimated.
statistics	Numeric vector specifying which performance metrics should be calculated. Possible values are "mean", "median", "mean_bias", "median_bias", "sd", "iqr", "mean_se" (for mean standard error), "mse" (for mean squared error), and "coverage" (for confidence interval coverage).
alpha	Numeric value specifying alpha for confidence interval. Set to 0.05 for the usual 95% CI, 0.1 for a 90% CI, and so forth.
digits	Numeric value or vector specifying the number of decimal places to include.

Value

Numeric matrix.

Examples

```
# For X ~ N(mu, sigma^2), the MLE for sigma^2 is the sample variance with n
# in the denominator, but the unbiased version with (n - 1) is typically used
# for its unbiasedness. Compare these estimators in 1,000 trials with n = 25.
MLE <- c()
Unbiased <- c()
for (ii in 1: 1000) {
   x <- rnorm(n = 25)
   MLE[ii] <- sum((x - mean(x))^2) / 25
   Unbiased[ii] <- sum((x - mean(x))^2) / 24
 }
sumsim(estimates = cbind(MLE, Unbiased), truth = 1)
```

---

sum_i *Sum of Integer Values*

Description

Written in C++, this function runs faster than sum for large integer vectors/matrices.

Usage

```
sum_i(x)
```

Arguments

x	Integer vector or matrix.

Value

Numeric value.

Examples

```
# For very large integer objects, sum_i is faster than sum
x <- rpois(100000, lambda = 5)
sum(x) == sum_i(x)
benchmark(sum(x), sum_i(x), replications = 1000)

# For smaller integer objects, sum_i is slower than sum
x <- rpois(1000, lambda = 5)
sum(x) == sum_i(x)
benchmark(sum(x), sum_i(x), replications = 1000)
```

---

trim *Trim Tail Values off of a Vector*

Description

Returns input vector with tail values trimmed off of it. User can specify tail probability to trim or lower and upper cutpoints for values to retain.

Usage

```
trim(x, p = NULL, tails = "both", cutpoints = NULL, keep.edge = TRUE)
```

Arguments

x	Numeric vector.
p	Numeric value giving tail probability to trim from x. Can leave as NULL if you specify cutpoints.
tails	Numeric value indicating which tail should be trimmed. Possible values are "both", "lower", and "upper".
cutpoints	Numeric vector indicating what range of values should be retained. For example, set to c(0, 1) to trim all values below 0 or greater than 1. Can leave as NULL if you specify p.
keep.edge	Logical value indicating whether values in x that are on the edge of being trimmed (i.e. equal to one of the endpoints) should be retained.

Value

Numeric vector.

See Also

inside

Examples

```
# Generate data from N(0, 1) and then trim the lower and upper 1\%
x <- rnorm(1000)
y <- trim(x, p = 0.01)

# Generate data from N(0, 1) and then trim values outside of (-1.5, 1.5)
x <- rnorm(100000)
y <- trim(x, cutpoints = c(-1.5, 1.5))
```

---

true_range_i *True Range of Integer Values*

Description

Defined as the difference between the maximum and the minimum. Equivalent to base R code `diff(range(x))`, but much faster.

Usage

```
true_range_i(x)
```

Arguments

x	Integer vector or matrix.

Value

Integer value.

Examples

```
# In general, true_range_i is much faster than diff(range(x))
x <- rpois(1000, lambda = 5)
all.equal(diff(range(x)), true_range_i(x))
benchmark(diff(range(x)), true_range_i(x), replications = 5000)
```

true_range_n *True Range of Numeric Values*

Description

Defined as the difference between the maximum and the minimum. Equivalent to base R code `diff(range(x))`, but much faster. For integer objects, `true_range_i` should run even faster.

Usage

```
true_range_n(x)
```

Arguments

x	Numeric vector or matrix.

Value

Numeric value.

Examples

```
# In general, true_range_n is much faster than diff(range(x))
x <- rnorm(1000)
all.equal(diff(range(x)), true_range_n(x))
benchmark(diff(range(x)), true_range_n(x), replications = 5000)

# For integer vectors, true_range_i should be even faster
x <- rpois(1000, lambda = 5)
all.equal(diff(range(x)), true_range_i(x))
benchmark(diff(range(x)), true_range_n(x), true_range_i(x),
          replications = 5000)
```

var_i *Sample Variance for Integer Values*

Description

Written in C++, this function should always run faster than `var` for integer vectors. Not valid for non-integer input vectors.

Usage

```
var_i(x)
```

Arguments

x	Integer vector.

Value

Numeric value.

Examples

```
# For integer vectors, var_i is typically much faster than var.
x <- rpois(1000, lambda = 5)
all.equal(var(x), var_i(x))
benchmark(var(x), var_i(x), replications = 5000)
```

---

var_n *Sample Variance for Numeric Values*

Description

Written in C++, this function should always run faster than var for numeric vectors. For integer vectors, var_i should run even faster.

Usage

```
var_n(x)
```

Arguments

x	Numeric vector.

Value

Numeric value.

Examples

```
# In general, var_n is much faster than var.
x <- rnorm(1000)
all.equal(var(x), var_n(x))
benchmark(var(x), var_n(x), replications = 1000)

# For integer vectors, var_i should be even faster.
x <- rpois(1000, lambda = 5)
all.equal(var(x), var_i(x))
benchmark(var(x), var_n(x), var_i(x), replications = 1000)
```

`weighted_mean_ii` *Weighted Arithmetic Mean for Integer Values and Integer Weights*

Description

Written in C++, this function should always run faster than `weighted.mean`.

Usage

```
weighted_mean_ii(x, w)
```

Arguments

`x`	Integer vector of values.
`w`	Integer vector of weights.

Details

For optimal speed, choose the version of this function that matches the class of your `x` and `w`:
`weighted_mean_nn` for numeric `x`, numeric `w`
`weighted_mean_ni` for numeric `x`, integer `w`
`weighted_mean_in` for integer `x`, numeric `w`
`weighted_mean_ii` for integer `x`, integer `w`

These functions typically execute several times faster than the base R function `weighted.mean` and weighted average functions in other packages (e.g. `wtd.mean` in **Hmisc** and `wt.mean` in **SDMTools**).

Value

Numeric value.

Examples

```
# weighted_mean_ii is typically much faster than weighted.mean
x <- rpois(1000, lambda = 5)
w <- rpois(1000, lambda = 5)
all.equal(weighted.mean(x, w), weighted_mean_ii(x, w))
benchmark(weighted.mean(x, w), weighted_mean_ii(x, w), replications = 2000)
```

`weighted_mean_in` *Weighted Arithmetic Mean for Integer Values and Numeric Weights*

Description

Written in C++, this function should always run faster than `weighted.mean`.

Usage

```
weighted_mean_in(x, w)
```

Arguments

`x`	Integer vector of values.
`w`	Numeric vector of weights.

Details

For optimal speed, choose the version of this function that matches the class of your `x` and `w`:
`weighted_mean_nn` for numeric `x`, numeric `w`
`weighted_mean_ni` for numeric `x`, integer `w`
`weighted_mean_in` for integer `x`, numeric `w`
`weighted_mean_ii` for integer `x`, integer `w`

These functions typically execute several times faster than the base R function `weighted.mean` and weighted average functions in other packages (e.g. `wtd.mean` in **Hmisc** and `wt.mean` in **SDMTools**).

Value

Numeric value.

Examples

```
# weighted_mean_in is typically much faster than weighted.mean
x <- rpois(1000, lambda = 5)
w <- runif(1000)
all.equal(weighted.mean(x, w), weighted_mean_in(x, w))
benchmark(weighted.mean(x, w), weighted_mean_in(x, w), replications = 2000)
```

`weighted_mean_ni` *Weighted Arithmetic Mean for Numeric Values and Integer Weights*

Description

Written in C++, this function should always run faster than `weighted.mean`.

Usage

```
weighted_mean_ni(x, w)
```

Arguments

`x`	Numeric vector of values.
`w`	Integer vector of weights.

Details

For optimal speed, choose the version of this function that matches the class of your `x` and `w`:
`weighted_mean_nn` for numeric `x`, numeric `w`
`weighted_mean_ni` for numeric `x`, integer `w`
`weighted_mean_in` for integer `x`, numeric `w`
`weighted_mean_ii` for integer `x`, integer `w`

These functions typically execute several times faster than the base R function `weighted.mean` and weighted average functions in other packages (e.g. `wtd.mean` in **Hmisc** and `wt.mean` in **SDMTools**).

Value

Numeric value.

Examples

```
# weighted_mean_ni is typically much faster than weighted.mean
x <- rnorm(1000)
w <- rpois(1000, lambda = 5)
all.equal(weighted.mean(x, w), weighted_mean_ni(x, w))
benchmark(weighted.mean(x, w), weighted_mean_ni(x, w), replications = 2000)
```

weighted_mean_nn *Weighted Arithmetic Mean for Numeric Values and Numeric Weights*

Description

Written in C++, this function should always run faster than weighted.mean.

Usage

```
weighted_mean_nn(x, w)
```

Arguments

x	Numeric vector of values.
w	Numeric vector of weights.

Details

For optimal speed, choose the version of this function that matches the class of your x and w:
weighted_mean_nn for numeric x, numeric w
weighted_mean_ni for numeric x, integer w
weighted_mean_in for integer x, numeric w
weighted_mean_ii for integer x, integer w

These functions typically execute several times faster than the base R function weighted.mean and weighted average functions in other packages (e.g. wtd.mean in **Hmisc** and wt.mean in **SDMTools**).

Value

Numeric value.

Examples

```
# weighted_mean_nn is typically much faster than weighted.mean
x <- rnorm(1000)
w <- runif(1000)
all.equal(weighted.mean(x, w), weighted_mean_nn(x, w))
benchmark(weighted.mean(x, w), weighted_mean_nn(x, w), replications = 2000)
```

`which_max_im` *Return (Row, Column) Index of (First) Maximum of an Integer Matrix*

Description

Written in C++, this function tends to run much faster than the equivalent (if maximum is unique) base R solution `which(x == max(x), arr.ind = TRUE)`.

Usage

```
which_max_im(x)
```

Arguments

`x`	Integer matrix.

Details

For optimal speed, choose the version of this function that matches the class of your `x`:

`which_max_nv` for numeric vector.
`which_max_iv` for integer vector.
`which_max_nm` for numeric matrix.
`which_max_im` for integer matrix.

Value

Integer vector.

Examples

```
# which_max_im is typically much faster than
# which(x == max(x), arr.ind = TRUE)
x <- matrix(rpois(100, lambda = 15), ncol = 10)
all(which(x == max(x), arr.ind = TRUE) == which_max_im(x))
benchmark(which(x == max(x), arr.ind = TRUE), which_max_im(x),
          replications = 5000)
```

`which_max_iv` *Return Index of (First) Maximum of an Integer Vector*

Description

Written in C++, this function tends to run faster than `which.max` for large integer vectors.

Usage

```
which_max_iv(x)
```

Arguments

`x`	Integer vector.

Details

For optimal speed, choose the version of this function that matches the class of your `x`:

`which_max_nv` for numeric vector.
`which_max_iv` for integer vector.
`which_max_nm` for numeric matrix.
`which_max_im` for integer matrix.

Value

Integer value.

Examples

```
# For long vectors, which_max_iv is faster than which.max
x <- rpois(10000, lambda = 15)
which.max(x) == which_max_iv(x)
benchmark(which.max(x), which_max_iv(x), replications = 5000)

# For shorter vectors, which_max_iv is slower than which.max
x <- rpois(100, lambda = 15)
which.max(x) == which_max_iv(x)
benchmark(which.max(x), which_max_iv(x), replications = 20000)
```

`which_max_nm` *Return (Row, Column) Index of (First) Maximum of a Numeric Matrix*

Description

Written in C++, this function tends to run much faster than the equivalent (if maximum is unique) base R solution `which(x == max(x), arr.ind = TRUE)`.

Usage

```
which_max_nm(x)
```

Arguments

`x`	Numeric matrix.

Details

For optimal speed, choose the version of this function that matches the class of your `x`:

`which_max_nv` for numeric vector.
`which_max_iv` for integer vector.
`which_max_nm` for numeric matrix.
`which_max_im` for integer matrix.

Value

Integer vector.

Examples

```
# which_max_nm is typically much faster than
# which(x == max(x), arr.ind = TRUE)
x <- matrix(rnorm(100), ncol = 10)
all(which(x == max(x), arr.ind = TRUE) == which_max_nm(x))
benchmark(which(x == max(x), arr.ind = TRUE), which_max_nm(x),
          replications = 5000)
```

which_max_nv *Return Index of (First) Maximum of a Numeric Vector*

Description

Written in C++, this function tends to run faster than `which.max` for large numeric vectors.

Usage

```
which_max_nv(x)
```

Arguments

x	Numeric vector.

Details

For optimal speed, choose the version of this function that matches the class of your `x`:

`which_max_nv` for numeric vector.
`which_max_iv` for integer vector.
`which_max_nm` for numeric matrix.
`which_max_im` for integer matrix.

Value

Integer value.

Examples

```
# For long vectors, which_max_nv is faster than which.max
x <- rnorm(100000)
which.max(x) == which_max_nv(x)
benchmark(which.max(x), which_max_nv(x), replications = 500)

# For shorter vectors, which_max_nv is slower than which.max
x <- rnorm(100)
which.max(x) == which_max_nv(x)
benchmark(which.max(x), which_max_nv(x), replications = 10000)
```

`which_min_im` *Return (Row, Column) Index of (First) Minimum of an Integer Matrix*

Description

Written in C++, this function tends to run much faster than the equivalent (if minimum is unique) base R solution `which(x == min(x), arr.ind = TRUE)`.

Usage

```
which_min_im(x)
```

Arguments

`x`	Integer matrix.

Details

For optimal speed, choose the version of this function that matches the class of your `x`:

`which_min_nv` for numeric vector.
`which_min_iv` for integer vector.
`which_min_nm` for numeric matrix.
`which_min_im` for integer matrix.

Value

Integer vector.

Examples

```
# which_min_im is typically much faster than
# which(x == min(x), arr.ind = TRUE)
x <- matrix(rpois(100, lambda = 10), ncol = 10)
all(which(x == min(x), arr.ind = TRUE) == which_min_im(x))
benchmark(which(x == min(x), arr.ind = TRUE), which_min_im(x),
          replications = 5000)
```

which_min_iv *Return Index of (First) Minimum of an Integer Vector*

Description

Written in C++, this function tends to run faster than `which.min` for large integer vectors.

Usage

```
which_min_iv(x)
```

Arguments

x	Integer vector.

Details

For optimal speed, choose the version of this function that matches the class of your x:

`which_min_nv` for numeric vector.
`which_min_iv` for integer vector.
`which_min_nm` for numeric matrix.
`which_min_im` for integer matrix.

Value

Integer value.

Examples

```
# For long vectors, which_min_iv is faster than which.min
x <- rpois(10000, lambda = 15)
which.min(x) == which_min_iv(x)
benchmark(which.min(x), which_min_iv(x), replications = 5000)

# For shorter vectors, which_min_iv is slower than which.min
x <- rpois(100, lambda = 15)
which.min(x) == which_min_iv(x)
benchmark(which.min(x), which_min_iv(x), replications = 20000)
```

`which_min_nm` *Return (Row, Column) Index of (First) Minimum of a Numeric Matrix*

Description

Written in C++, this function tends to run much faster than the equivalent (if minimum is unique) base R solution `which(x == min(x), arr.ind = TRUE)`.

Usage

```
which_min_nm(x)
```

Arguments

`x`	Numeric matrix.

Details

For optimal speed, choose the version of this function that matches the class of your `x`:

`which_min_nv` for numeric vector.
`which_min_iv` for integer vector.
`which_min_nm` for numeric matrix.
`which_min_im` for integer matrix.

Value

Integer vector.

Examples

```
# which_min_nm is typically much faster than
# which(x == min(x), arr.ind = TRUE)
x <- matrix(rnorm(100), ncol = 10)
all(which(x == min(x), arr.ind = TRUE) == which_min_nm(x))
benchmark(which(x == min(x), arr.ind = TRUE), which_min_nm(x),
          replications = 5000)
```

`which_min_nv` *Return Index of (First) Minimum of a Numeric Vector*

Description

Written in C++, this function tends to run faster than `which.min` for large numeric vectors.

Usage

```
which_min_nv(x)
```

Arguments

`x`	Numeric vector.

Details

For optimal speed, choose the version of this function that matches the class of your `x`:

`which_min_nv` for numeric vector.
`which_min_iv` for integer vector.
`which_min_nm` for numeric matrix.
`which_min_im` for integer matrix.

Value

Integer value.

Examples

```
# For long vectors, which_min_nv is faster than which.min
x <- rnorm(100000)
which.min(x) == which_min_nv(x)
benchmark(which.min(x), which_min_nv(x), replications = 1000)

# For shorter vectors, which_min_nv is slower than which.min
x <- rnorm(100)
which.min(x) == which_min_nv(x)
benchmark(which.min(x), which_min_nv(x), replications = 10000)
```

Index