Package 'pdqr'

December 15, 2019

Title Work with Custom Distribution Functions

Version 0.2.1

Description Create, transform, and summarize custom random variables with distribution functions (analogues of 'p*()', 'd*()', 'q*()', and 'r*()' functions from base R). Two types of distributions are supported: ``discrete'' (random variable has finite number of output values) and ``continuous'' (infinite number of values in the form of continuous random variable). Functions for distribution transformations and summaries are available. Implemented approaches often emphasize approximate and numerical solutions: all distributions assume finite support and finite values of density function; some methods implemented with simulation techniques.

License MIT + file LICENSE

URL https://github.com/echasnovski/pdqr, https://echasnovski.github.io/pdqr

BugReports https://github.com/echasnovski/pdqr/issues

Depends R (>= 3.3)

Imports graphics, stats

Suggests covr, grDevices, knitr, pillar, rmarkdown, spelling, testthat, vdiffr

VignetteBuilder knitr

Encoding UTF-8

Language en-US

LazyData true

RoxygenNote 7.0.1

Collate 'as_p.R' 'as_d.R' 'as_q.R' 'as_r.R' 'assertions.R' 'form-compare.R' 'form-other.R' 'form_regrid.R' 'form_resupport.R' 'form_retype.R' 'form_tails.R' 'form_trans.R' 'group-generics.R' 'meta.R' 'print.R' 'new_p.R' 'new_d.R' 'new_q.R' 'new_r.R' 'pdqr-package.R' 'plot.R' 'region.R' 'summ-other.R' 'summ_center.R' 'summ_classmetric.R'

'summ_distance.R' 'summ_entropy.R' 'summ_hdr.R'
'summ_interval.R' 'summ_moment.R' 'summ_order.R' 'summ_pval.R'
'summ_roc.R' 'summ_separation.R' 'summ_spread.R' 'utils-as.R'
'utils-form.R' 'utils-new.R' 'utils-summ.R' 'utils.R' 'x_tbl.R'
'zzz.R'

NeedsCompilation no

Author Evgeni Chasnovski [aut, cre]

Maintainer Evgeni Chasnovski <evgeni.chasnovski@gmail.com>

Repository CRAN

Date/Publication 2019-12-15 16:50:02 UTC

R topics documented:

---

pdqr-package *pdqr: Work with Custom Distribution Functions*

Description

Create, transform, and summarize custom random variables with distribution functions (analogues of 'p*()', 'd*()', 'q*()', and 'r*()' functions from base R). Two types of distributions are supported: "discrete" (random variable has finite number of output values) and "continuous" (infinite number of values in the form of continuous random variable). Functions for distribution transformations and summaries are available. Implemented approaches often emphasize approximate and numerical solutions: all distributions assume finite support and finite values of density function; some methods implemented with simulation techniques.

Details

Excerpt of important documentation:

- README and vignettes provide overview of package functionality.
- Documentation of meta_*() functions describes implementation details of pdqr-functions.
  - Documentation of print() and plot() methods describes how you can interactively explore properties of pdqr-functions.
- Documentation of new_*() functions describes the process of creating pdqr-functions.
- Documentation of as_*() functions describes the process of updating class of pdqr-functions.
- Documentation of form_*() functions describes how different transformation functions work. Important pages are for `form_trans()` and Pdqr methods for S3 group generic functions.
- Documentation of summ_*() functions describes how different summary functions work. A good place to start is `summ_center()`.
- Documentation of `region_*()` functions describes functionality to work with regions: data frames defining subset of one dimensional real line.

This package has the following options (should be set by options()):

- *"pdqr.group_gen.args_new"*, *"pdqr.group_gen.n_sample"*, *"pdqr.group_gen.repair_supp_method"*. They may be used to customize behavior of methods for S3 group generic functions. See their help page for more information.
- *"pdqr.assert_args"*. This boolean option (default to `TRUE`) may be used to turn off sanity checks of function arguments (set it to `FALSE`), which will somewhat increase general execution speed. **Use this option at your own risk in case you are confident that input arguments have correct type and structure**.

Author(s)

Maintainer: Evgeni Chasnovski <evgeni.chasnovski@gmail.com>

See Also

Useful links:

- https://github.com/echasnovski/pdqr
- https://echasnovski.github.io/pdqr
- Report bugs at https://github.com/echasnovski/pdqr/issues

---

as_p *Convert to pdqr-function*

Description

Convert some function to be a proper pdqr-function of specific class, i.e. a function describing distribution with finite support and finite values of probability/density.

Usage

```
as_p(f, ...)

## Default S3 method:
as_p(f, support = NULL, ..., n_grid = 10001)

## S3 method for class 'pdqr'
as_p(f, ...)

as_d(f, ...)

## Default S3 method:
as_d(f, support = NULL, ..., n_grid = 10001)

## S3 method for class 'pdqr'
as_d(f, ...)

as_q(f, ...)

## Default S3 method:
as_q(f, support = NULL, ..., n_grid = 10001)

## S3 method for class 'pdqr'
as_q(f, ...)

as_r(f, ...)

## Default S3 method:
as_r(f, support = NULL, ..., n_grid = 10001,
  n_sample = 10000, args_new = list())
```

```
## S3 method for class 'pdqr'
as_r(f, ...)
```

Arguments

f	Appropriate function to be converted (see Details).
...	Extra arguments to f.
support	Numeric vector with two increasing elements describing desired support of output. If NULL or any its value is NA, detection is done using specific algorithms (see Details).
n_grid	Number of grid points at which f will be evaluated (see Details). Bigger values lead to better approximation precision, but worse memory usage and evaluation speed (direct and in summ_*() functions).
n_sample	Number of points to sample from f inside as_r().
args_new	List of extra arguments for new_d() to control density() inside as_r().

Details

General purpose of as_*() functions is to create a proper pdqr-function of desired class from input which doesn't satisfy these conditions. Here is described sequence of steps which are taken to achieve that goal.

If f **is already a pdqr-function**, as_*() functions properly update it to have specific class. They take input's "x_tbl" metadata and type to use with corresponding new_*() function. For example, as_p(f) in case of pdqr-function f is essentially the same as new_p(x = meta_x_tbl(f), type = meta_type(f)).

If f **is a function describing "honored" distribution**, it is detected and output is created in predefined way taking into account extra arguments in .... For more details see "Honored distributions" section.

If f **is some other unknown function**, as_*() functions use heuristics for approximating input distribution with a "proper" pdqr-function. Outputs of as_*() can be only pdqr-functions of type "continuous" (because of issues with support detection). It is assumed that f returns values appropriate for desired output class of as_*() function and output type "continuous". For example, input for as_p() should return values of some continuous cumulative distribution function (monotonically non-increasing values from 0 to 1). To manually create function of type "discrete", supply data frame input describing it to appropriate new_*() function.

General algorithm of how as_*() functions work for unknown function is as follows:

- **Detect support**. See "Support detection" section for more details.
- **Create data frame input for** new_*(). The exact process differs:
  - In as_p() equidistant grid of n_grid points is created inside detected support. After that, input's values at the grid is taken as reference points of cumulative distribution function used to *approximate* density at that same grid. This method showed to work more reliably in case density goes to infinity. That grid and density values are used as "x" and "y" columns of data frame input for new_p().

  - In `as_d()` "x" column of data frame is the same equidistant grid is taken as in `as_p()`. "y" column is taken as input's values at this grid after possibly imputing infinity values. This imputation is done by taking maximum from left and right linear extrapolations on mentioned grid.
  - In `as_q()`, at first inverse of input `f` function is computed on [0; 1] interval. It is done by approximating it with piecewise-linear function on [0; 1] equidistant grid with `n_grid` points, imputing infinity values (which ensures finite support), and computing inverse of approximation. This inverse of `f` is used to create data frame input with `as_p()`.
  - In `as_r()` at first d-function with `new_d()` is created based on the same sample used for support detection and extra arguments supplied as list in `args_new` argument. In other words, density estimation is done based on sample, generated from input `f`. After that, its values are used to create data frame with `as_d()`.

- **Use appropriate** new_*() **function** with data frame from previous step and `type = "continuous"`. This step implies that all tails outside detected support are trimmed and data frame is normalized to represent proper piecewise-linear density.

Value

A pdqr-function of corresponding class.

Honored distributions

For efficient workflow, some commonly used distributions are recognized as special ("honored"). Those receive different treatment in as_*() functions.

Basically, there is a manually selected list of "honored" distributions with all their information enough to detect them. Currently that list has all common univariate distributions from 'stats' package, i.e. all except multinomial and "less common distributions of test statistics".

"Honored" distribution is **recognized only if** `f` **is one of** p*()**,** d*()**,** q*()**, or** r*() **function describing honored distribution and is supplied as variable with original name**. For example, `as_d(dunif)` will be treated as "honored" distribution but `as_d(function(x) {dunif(x)})` will not.

After it is recognized that input `f` represents "honored" distribution, **its support is computed based on predefined rules**. Those take into account special features of distribution (like infinite support or infinite density values) and supplied extra arguments in `...`. Usually output support "loses" only around `1e-6` probability on each infinite tail.

After that, for "discrete" type output `new_d()` is used for appropriate data frame input and for "continuous" - `as_d()` with appropriate d*() function and support. D-function is then converted to desired class with as_*().

Support detection

In case input is a function without any extra information, as_*() functions must know which finite support its output should have. User can supply desired support directly with `support` argument, which can also be `NULL` (mean automatic detection of both edges) or have `NA` to detect only those edges.

Support is detected in order to preserve as much information as practically reasonable. Exact methods differ:

- In as_p() support is detected as values at which input function is equal to 1e-6 (left edge detection) and 1 -1e-6 (right edge), which means "losing" 1e-6 probability on each tail. **Note** that those values are searched inside [-10^100; 10^100] interval.
- In as_d(), at first an attempt at finding one point of non-zero density is made by probing 10000 points spread across wide range of real line (approximately from -1e7 to 1e7). If input's value at all of them is zero, error is thrown. After finding such point, cumulative distribution function is made by integrating input with integrate() using found point as reference (without this there will be poor accuracy of integrate()). Created CDF function is used to find 1e-6 and 1 -1e-6 quantiles as in as_p(), which serve as detected support.
- In as_q() quantiles for 0 and 1 are probed for being infinite. If they are, 1e-6 and 1 -1e-6 quantiles are used respectively instead of infinite values to form detected support.
- In as_r() sample of size n_sample is generated and detected support is its range stretched by mean difference of sorted points (to account for possible tails at which points were not generated). **Note** that this means that original input f "demonstrates its randomness" only once inside as_r(), with output then used for approximation of "original randomness".

See Also

pdqr_approx_error() for computing approximation errors compared to some reference function (usually input to as_*() family).

Examples

```
# Convert existing "proper" pdqr-function
set.seed(101)
x <- rnorm(10)
my_d <- new_d(x, "continuous")

my_p <- as_p(my_d)

# Convert "honored" function to be a proper pdqr-function. To use this
# option, supply originally named function.
p_unif <- as_p(punif)
r_beta <- as_r(rbeta, shape1 = 2, shape2 = 2)
d_pois <- as_d(dpois, lambda = 5)

  # `pdqr_approx_error()` computes pdqr approximation error
summary(pdqr_approx_error(as_d(dnorm), dnorm))

  # This will work as if input is unkonw function because of unsupported
  # variable name
my_runif <- function(n) {runif(n)}
r_unif_2 <- as_r(my_runif)
plot(as_d(r_unif_2))

# Convert some other function to be a "proper" pdqr-function
my_d_quadr <- as_d(function(x) {0.75*(1 - x^2)}, support = c(-1, 1))

# Support detection
unknown <- function(x) {dnorm(x, mean = 1)}
```

```
  # Completely automatic support detection
as_d(unknown)
  # Semi-automatic support detection
as_d(unknown, support = c(-4, NA))
as_d(unknown, support = c(NA, 5))

  # If support is very small and very distant from zero, it probably won't
  # get detected in `as_d()` (throwing a relevant error)
## Not run:
as_d(function(x) {dnorm(x, mean = 10000, sd = 0.1)})

## End(Not run)

# Using different level of granularity
as_d(unknown, n_grid = 1001)
```

---

enpoint *Represent pdqr-function as a set of points*

Description

Function `enpoint()` suggests a reasonable default ways of converting pdqr-function into a data frame of numerical values (points) with desirable number of rows. Representation of pdqr-function as a set of numbers helps to conduct analysis using approaches outside of 'pdqr' package. For example, one can visually display pdqr-function with some other plotting functionality.

Usage

```
enpoint(f, n_points = 1001)
```

Arguments

`f`	A pdqr-function.
`n_points`	Desired number of points in the output. Not used in case of "discrete" type p-, d-, and q-function `f`.

Details

Structure of output depends on class and type of input pdqr-function `f`:

- **P-functions** are represented with "x" (for "x" values) and "p" (for cumulative probability at "x" points) columns:
  - For "continuous" type, "x" is taken as an equidistant grid (with `n_points` elements) on input's support.
  - For "discrete" type, "x" is taken directly from "x_tbl" metadata without using `n_points` argument.

- **D-functions** are represented with "x" column and one more (for values of d-function at "x" points):
  - For "continuous" type, second column is named "y" and is computed as values of `f` at elements of "x" column (which is the same grid as in p-function case).
  - For "discrete" it is named "prob". Both "x" and "prob" columns are taken from "x_tbl" metadata.
- **Q-functions** are represented almost as p-functions but in inverse fashion. Output data frame has "p" (probabilities) and "x" (values of q-function `f` at "p" elements) columns.
  - For "continuous" type, "p" is computed as equidistant grid (with `n_points` elements) between 0 and 1.
  - For "discrete" type, "p" is taken from "cumprob" column of "x_tbl" metadata.
- **R-functions** are represented by generating `n_points` elements from distribution. Output data frame has columns "n" (consecutive point number, basically a row number) and "x" (generated elements).

Note that the other way to produce points for p-, d-, and q-functions is to manually construct them with `form_regrid()` and `meta_x_tbl()`. However, this method may slightly change function values due to possible renormalization inside `form_regrid()`.

Value

A data frame with `n_points` (or less, for "discrete" type p-, d-, or q-function `f`) rows and two columns with names depending on `f`'s class and type.

See Also

`pdqr_approx_error()` for diagnostics of pdqr-function approximation accuracy.

Pdqr methods for plot() for a direct plotting of pdqr-functions.

`form_regrid()` to change underlying grid of pdqr-function.

Examples

```
d_norm <- as_d(dnorm)
head(enpoint(d_norm))

# Control number of points with `n_points` argument
enpoint(d_norm, n_points = 5)

# Different pdqr classes and types produce different column names in output
colnames(enpoint(new_p(1:2, "discrete")))
colnames(enpoint(new_d(1:2, "discrete")))
colnames(enpoint(new_d(1:2, "continuous")))
colnames(enpoint(new_q(1:2, "continuous")))
colnames(enpoint(new_r(1:2, "continuous")))

# Manual way with different output structure
df <- meta_x_tbl(form_regrid(d_norm, 5))
  # Difference in values due to `form_regrid()` renormalization
plot(enpoint(d_norm, 5), type = "l")
```

```
lines(df[["x"]], df[["y"]], col = "blue")
```

`form_estimate` *Create a pdqr-function for distribution of sample estimate*

Description

Based on pdqr-function, statistic function, and sample size describe the distribution of sample estimate. This might be useful for statistical inference.

Usage

```
form_estimate(f, stat, sample_size, ..., n_sample = 10000, args_new = list())
```

Arguments

`f`	A pdqr-function.
`stat`	Statistic function. Should be able to accept numeric vector of size `sample_size` and return single numeric or logical output.
`sample_size`	Size of sample for which distribution of sample estimate is needed.
`...`	Other arguments for `stat`.
`n_sample`	Number of elements to generate from distribution of sample estimate.
`args_new`	List of extra arguments for new_*() function to control `density()`.

Details

General idea is to create a sample from target distribution by generating `n_sample` samples of size `sample_size` and compute for each of them its estimate by calling input `stat` function. If created sample is logical, **boolean** pdqr-function (type "discrete" with elements being exactly 0 and 1) is created with probability of being true estimated as share of `TRUE` values (after removing possible `NA`). If sample is numeric, it is used as input to new_*() of appropriate class with `type` equal to type of `f` (if not forced otherwise in `args_new`).

Notes:

- This function may be very time consuming for large values of `n_sample` and `sample_size`, as total of `n_sample*sample_size` numbers are generated and `stat` function is called `n_sample` times.
- Output distribution might have a bias compared to true distribution of sample estimate. One useful technique for bias correction: compute mean value of estimate using big `sample_size` (with `mean(as_r(f)(sample_size))`) and then recenter distribution to actually have that as a mean.

Value

A pdqr-function of the same class and type (if not forced otherwise in `args_new`) as `f`.

See Also

Other form functions: form_mix(), form_regrid(), form_resupport(), form_retype(), form_smooth(), form_tails(), form_trans()

Examples

```
# These examples take some time to run, so be cautious

set.seed(101)

# Type "discrete"
d_dis <- new_d(data.frame(x = 1:4, prob = 1:4/10), "discrete")
  # Estimate of distribution of mean
form_estimate(d_dis, stat = mean, sample_size = 10)
  # To change type of output, supply it in `args_new`
form_estimate(
  d_dis, stat = mean, sample_size = 10,
  args_new = list(type = "continuous")
)

# Type "continuous"
d_unif <- as_d(dunif)
  # Supply extra named arguments for `stat` in `...`
plot(form_estimate(d_unif, stat = mean, sample_size = 10, trim = 0.1))
lines(
  form_estimate(d_unif, stat = mean, sample_size = 10, trim = 0.3),
  col = "red"
)
lines(
  form_estimate(d_unif, stat = median, sample_size = 10),
  col = "blue"
)

# Statistic can return single logical value
d_norm <- as_d(dnorm)
all_positive <- function(x) {all(x > 0)}
  # Probability of being true should be around 0.5^5
form_estimate(d_norm, stat = all_positive, sample_size = 5)
```

---

form_mix *Form mixture of distributions*

Description

Based on a list of pdqr-functions and vector of weights form a pdqr-function for corresponding mixture distribution.

Usage

```
form_mix(f_list, weights = NULL)
```

Arguments

`f_list`	List of pdqr-functions. Can have different classes and types (see Details).
`weights`	Numeric vector of weights or `NULL` (default; corresponds to equal weights). Should be non-negative numbers with positive sum.

Details

Type of output mixture is determined by the following algorithm:

- If `f_list` consists only from pdqr-functions of "discrete" type, then output will have "discrete" type.
- If `f_list` has at least one pdqr-function of type "continuous", then output will have "continuous" type. In this case all "discrete" pdqr-functions in `f_list` are approximated with corresponding dirac-like "continuous" functions (with form_retype(*, method = "dirac")). **Note** that this approximation has consequences during computation of comparisons. For example, if original "discrete" function `f` is for distribution with one element `x`, then probability of `f >= x` being true is 1. After retyping to dirac-like function, this probability will be 0.5, because of symmetrical dirac-like approximation. Using a little nudge to `x` of `1e-7` magnitude in the correct direction (`f >= x -1e-7` in this case) will have expected output.

Class of output mixture is determined by the class of the first element of `f_list`. To change output class, use one of as_*() functions to change class of first element in `f_list` or class of output.

Note that if output "continuous" pdqr-function for mixture distribution (in theory) should have discontinuous density, it is approximated continuously: discontinuities are represented as intervals in "x_tbl" with extreme slopes (see Examples).

Value

A pdqr-function for mixture distribution of certain type and class (see Details).

See Also

Other form functions: `form_estimate()`, `form_regrid()`, `form_resupport()`, `form_retype()`, `form_smooth()`, `form_tails()`, `form_trans()`

Examples

```
# All "discrete"
d_binom <- as_d(dbinom, size = 10, prob = 0.5)
r_pois <- as_r(rpois, lambda = 1)
dis_mix <- form_mix(list(d_binom, r_pois))
plot(dis_mix)

# All "continuous"
p_norm <- as_p(pnorm)
```

```
d_unif <- as_d(dunif)

con_mix <- form_mix(list(p_norm, d_unif), weights = c(0.7, 0.3))
  # Output is a p-function, as is first element of `f_list`
con_mix
plot(con_mix)

  # Use `as_*()` functions to change class
d_con_mix <- as_d(con_mix)

  # Theoretical output density should be discontinuous, but here it is
  # approximated with continuous function
con_x_tbl <- meta_x_tbl(con_mix)
con_x_tbl[(con_x_tbl$x >= -1e-4) & (con_x_tbl$x <= 1e-4), ]

# Some "discrete", some "continuous"
all_mix <- form_mix(list(d_binom, d_unif))
plot(all_mix)
all_x_tbl <- meta_x_tbl(all_mix)

  # What dirac-like approximation looks like
all_x_tbl[(all_x_tbl$x >= 1.5) & (all_x_tbl$x <= 2.5), ]
```

`form_recenter` *Change center and spread of distribution*

Description

These functions implement linear transformations, output distribution of which has desired center and spread. These functions are useful for creating distributions with some input center and spread value based on present distribution, which is a common task during hypothesis testing.

Usage

```
form_recenter(f, to, method = "mean")

form_respread(f, to, method = "sd", center_method = "mean")
```

Arguments

`f`	A pdqr-function.
`to`	A desired value of summary.
`method`	Method of computing center for `form_recenter()` and spread for `form_respread()`. Values should be one of possible `method` values from `summ_center()` and `summ_spread()` respectively.
`center_method`	Method of computing center for `form_respread()` in order to preserve it in output.

Details

`form_recenter(f,to,method)` is basically a `f -summ_center(f,method) + to`: it moves distribution without affecting its shape so that output distribution has center at `to`.

`form_respread(f,to,method,center_method)` is a following linear transformation: `coef * (f -center) + center`, where `center` is `summ_center(f,center_method)` and `coef` is computed so as to guarantee output distribution to have spread equal to `to`. In other words, this linear transformation stretches distribution around its center until the result has spread equal to `to` (center remains the same as in input `f`).

Value

A pdqr-function describing distribution with desired center or spread.

Examples

```
my_beta <- as_d(dbeta, shape1 = 1, shape2 = 3)

my_beta2 <- form_recenter(my_beta, to = 2)
summ_center(my_beta2)

my_beta3 <- form_respread(my_beta2, to = 10, method = "range")
summ_spread(my_beta3, method = "range")
  # Center remains unchainged
summ_center(my_beta3)
```

`form_regrid` *Change grid of pdqr-function*

Description

Modify grid of pdqr-function (rows of "x_tbl" metadata) to increase (upgrid) or decrease (downgrid) granularity using method of choice. Upgridding might be useful in order to obtain more information during certain type of transformations. Downgridding might be useful for decreasing amount of used memory for storing pdqr-function without losing much information.

Usage

```
form_regrid(f, n_grid, method = "x")
```

Arguments

`f`	A pdqr-function.
`n_grid`	A desired number of grid elements in output.
`method`	Regrid method. Should be one of "x" or "q".

Details

The goal here is to create pdqr-function which is reasonably similar to `f` and has `n_grid` rows in "x_tbl" metadata.

General algorithm of regridding is as follows:

- **Compute reference grid**. For method "x" it is a sequence of equidistant points between edges of `f`'s support. For method "q" - sequence of quantiles for equidistant probabilities from 0 to 1. Lengths of reference grids for both methods are `n_grid`.
- **Adjust `f`'s grid to reference one**. This is done depending on `f`'s type and which kind or regridding is done (upgridding is the case when `n_grid` is strictly more than number of rows in "x_tbl" metadata, downgridding - when it is strictly less):
  - Type "discrete":
    * UPgridding "discrete" functions is not possible as it is assumed that input "discrete" functions can't have any "x" values other then present ones. In this case input is returned, the only case when output doesn't have desired `n_grid` rows in "x_tbl" metadata.
    * DOWNgridding "discrete" functions is done by computing nearest match of reference grid to `f`'s one and collapsing (by summing probabilities) all "x" values from input to the nearest matched ones. Here "computing nearest match" means that every element of reference grid is one-one matched with subset of unique values from `f`'s "x" elements. Matching is done in greedy iterative fashion in order to minimize total distance between reference grid and matched subset. **Note** that this can result in not optimal (with not minimum total distance) match and can take a while to compute in some cases.
  - Type "continuous":
    * UPgridding "continuous" functions is done by adding rows to "x_tbl" metadata with "x" values equal to those elements of reference grid which are the furthest away from input "x" grid as a set. Distance from point to set is meant as minimum of distances between point and all points of set. Values of "y" and "cumprob" columns are taken as values of corresponding to `f` d- and p-functions.
    * DOWNgridding "continuous" functions is done by computing nearest match of reference grid to `f`'s one (as for "discrete" type) and removing all unmatched rows from "x_tbl" metadata.

Special cases of `n_grid`:

- If `n_grid` is the same as number of rows in "x_tbl" metadata, then input `f` is returned.
- If `n_grid` is 1, appropriate new_*() function is used with single numeric input equal to distribution's median.

Value

A pdqr-function with modified grid.

See Also

`form_resupport()` for changing support of pdqr-function.

form_retype() for changing type of pdqr-function.

Other form functions: form_estimate(), form_mix(), form_resupport(), form_retype(), form_smooth(), form_tails(), form_trans()

Examples

```
# Type "discrete"
d_dis <- new_d(data.frame(x = 1:10, prob = 1:10/55), type = "discrete")

  # Downgridding
meta_x_tbl(form_regrid(d_dis, n_grid = 4))
meta_x_tbl(form_regrid(d_dis, n_grid = 4, method = "q"))

  # Upgridding for "discrete" type isn't possible. Input is returned
identical(d_dis, form_regrid(d_dis, n_grid = 100))

# Type "continuous"
  # Downgridding
d_norm <- as_d(dnorm)
plot(d_norm)
lines(form_regrid(d_norm, n_grid = 10), col = "blue")
lines(form_regrid(d_norm, n_grid = 10, method = "q"), col = "green")

  # Upgridding
d_con <- new_d(data.frame(x = 1:3, y = rep(0.5, 3)), type = "continuous")
meta_x_tbl(form_regrid(d_con, n_grid = 6))

# Pdqr-function with center at median is returned in case `n_grid` is 1
form_regrid(d_dis, n_grid = 1)
  # Dirac-like function is returned
form_regrid(d_con, n_grid = 1)
```

---

form_resupport *Change support of pdqr-function*

Description

Modify support of pdqr-function using method of choice.

Usage

```
form_resupport(f, support, method = "reflect")
```

Arguments

f	A pdqr-function.
support	Numeric vector with two increasing (or non-decreasing, see Details) elements describing support of the output. Values can be NA, in which case corresponding edge(s) are taken from f's support.

`method` Resupport method. One of "reflect", "trim", "winsor", "linear".

Details

Method "reflect" takes a density "tails" to the left of `support[1]` and to the right of `support[2]` and reflects them inside `support`. It means that values of density inside and outside of supplied `support` are added together in "symmetric fashion": `d(x) = d_f(x) + d_f(l -(x-l)) + d_f(r + (r-x))`, where `d_f` is density of input, `d` is density of output, `l` and `r` are left and right edges of input `support`. This option is useful for repairing support of new_*()'s output, as by default kernel density estimation in `density()` adds tails to the range of input `x` values. For example, if there is a need to ensure that distribution has only positive values, one can do `form_resupport(f,c(0,NA),method = "reflect")`. **Notes**:

- For "discrete" pdqr-functions that might result into creating new "x" values of distribution.
- Reflection over `support[1]` is done only if it is strictly greater than `f`'s left edge of support. Reflection over `support[2]` - if `f`'s right edge is strictly smaller.

Method "trim" removes density "tails" outside of `support`, normalizes the rest and creates appropriate pdqr-function.

Method "winsor" makes all density "tails" outside of input `support` "squashed" inside it in "dirac-like" fashion. It means that probability from both tails is moved inside `support` and becomes concentrated in `1e-8` neighborhood of nearest edge. This models a singular dirac distributions at the edges of `support`. **Note** that `support` can represent single point, in which case output has single element if `f`'s type is "discrete" or is a dirac-like distribution in case of "continuous" type.

Method "linear" transforms `f`'s support linearly to be input `support`. For example, if `f`'s support is [0; 1] and `support` is `c(-1,1)`, linear resupport is equivalent to `2*f -1`. **Note** that `support` can represent single point with the same effect as in "winsor" method.

Value

A pdqr-function with modified support and the same class and type as `f`.

See Also

`form_regrid()` for changing grid (rows of "x_tbl" metadata) of pdqr-function.

`form_retype()` for changing type of pdqr-function.

Other form functions: `form_estimate()`, `form_mix()`, `form_regrid()`, `form_retype()`, `form_smooth()`, `form_tails()`, `form_trans()`

Examples

```
set.seed(101)
d_norm <- as_d(dnorm)
d_dis <- new_d(data.frame(x = 1:4, prob = 1:4/10), "discrete")

# Method "reflect"
plot(d_norm)
lines(form_resupport(d_norm, c(-2, 1.5), "reflect"), col = "blue")
```

```
  # For "discrete" functions it might create new values
meta_x_tbl(form_resupport(d_dis, c(NA, 2.25), "reflect"))

  # This is often useful to ensure constraints after `new_()`
x <- runif(1e4)
d_x <- new_d(x, "continuous")
plot(d_x)
lines(form_resupport(d_x, c(0, NA), "reflect"), col = "red")
lines(form_resupport(d_x, c(0, 1), "reflect"), col = "blue")

# Method "trim"
plot(d_norm)
lines(form_resupport(d_norm, c(-2, 1.5), "trim"), col = "blue")

# Method "winsor"
plot(d_norm)
lines(form_resupport(d_norm, c(-2, 1.5), "winsor"), col = "blue")

# Method "linear"
plot(d_norm)
lines(form_resupport(d_norm, c(-2, 1.5), "linear"), col = "blue")
```

`form_retype` *Change type of pdqr-function*

Description

Modify type of pdqr-function using method of choice.

Usage

```
form_retype(f, type = NULL, method = "value")
```

Arguments

`f`	A pdqr-function.
`type`	A desired type of output. Should be one of "discrete" or "continuous". If `NULL` (default), it is chosen as an opposite of `f`'s type.
`method`	Retyping method. Should be one of "value", "piecelin", "dirac".

Details

If type of `f` is equal to input `type` then `f` is returned.

Method "value" uses renormalized columns of `f`'s "x_tbl" metadata as values for output's "x_tbl" metadata. In other words, it preserves ratios between values of d-function at certain "x" points. Its main advantages are that this method can work well with any pdqr type and that two consecutive conversions return the same function. Conversion algorithm is as follows:

- Retyping from "continuous" to `type` "discrete" is done by creating pdqr-function of corresponding class with the following "x_tbl" metadata: "x" column is the same as in `f`; "prob" column is equal to `f`'s "y" column after renormalization (so that their sum is 1).
- Retyping from "discrete" to `type` "continuous" is done in the same fashion: "x" column is the same; "y" column is equal to `f`'s "prob" column after renormalization (so that total integral of piecewise-linear density is equal to 1).

Method "piecelin" (default) should be used mostly for converting from "continuous" to "discrete" type. It uses the fact that 'pdqr' densities are piecewise-linear (linear in intervals between values of "x" column of "x_tbl" metadata) on their support:

- Retyping from "continuous" to `type` "discrete" is done by computing "x" values as centers of interval masses with probabilities equal to interval total probabilities.
- Retyping from "discrete" to `type` "continuous" is made approximately by trying to compute "x" grid, for which "x" values of input distribution are going to be centers of mass. Algorithm is approximate and might result into a big errors in case of small number of "x" values or if they are not "suitable" for this kind of transformation.

Method "dirac" is used mostly for converting from "discrete" to "continuous" type (for example, in `form_mix()` in case different types of input pdqr-functions). It works in the following way:

- Retyping from "continuous" to `type` "discrete" works only if "x_tbl" metadata represents a mixture of dirac-like distributions. In that case it is transformed to have "x" values from centers of those dirac-like distributions with corresponding probabilities.
- Retyping from "discrete" to `type` "continuous" works by transforming each "x" value from "x_tbl" metadata into dirac-like distribution with total probability taken from corresponding value of "prob" column. Output essentially represents a mixture of dirac-like distributions.

Value

A pdqr-function with type equal to input `type`.

See Also

`form_regrid()` for changing grid (rows of "x_tbl" metadata) of pdqr-function.

`form_resupport()` for changing support of pdqr-function.

Other form functions: `form_estimate()`, `form_mix()`, `form_regrid()`, `form_resupport()`, `form_smooth()`, `form_tails()`, `form_trans()`

Examples

```
my_con <- new_d(data.frame(x = 1:5, y = c(1, 2, 3, 2, 1)/9), "continuous")
meta_x_tbl(my_con)

  # By default, conversion is done to the opposite type
my_dis <- form_retype(my_con)
meta_x_tbl(my_dis)

# Default retyping (with method "value") is accurate when doing consecutive
# retyping
```

```
my_con_2 <- form_retype(my_dis, "continuous")
meta_x_tbl(my_con_2)

# Method "dirac"
my_dirac <- form_retype(my_dis, "continuous", method = "dirac")
meta_x_tbl(my_dirac)

# Method "piecelin"
  # From "continuous" to "discrete" (preferred direction)
my_dis_piece <- form_retype(my_con, "discrete", method = "piecelin")
meta_x_tbl(my_dis_piece)
  # Conversion from "discrete" to "continuous" is very approximate
my_con_piece <- form_retype(my_dis_piece, "continuous", method = "piecelin")
meta_x_tbl(my_con_piece)

plot(my_con, main = 'Approximate nature of method "piecelin"')
lines(my_con_piece, col = "blue")
```

`form_smooth` *Smooth pdqr-function*

Description

Smooth pdqr-function using random sampling and corresponding new_*() function.

Usage

```
form_smooth(f, n_sample = 10000, args_new = list())
```

Arguments

`f`	A pdqr-function.
`n_sample`	Number of elements to sample.
`args_new`	List of extra arguments for new_*() to control `density()`.

Details

General idea of smoothing is to preserve "sampling randomness" as much as reasonably possible while creating more "smooth" probability mass or density function.

At first step, sample of size `n_sample` is generated from distribution represented by `f`. Then, based on the sample, "continuous" d-function is created with `new_d()` and arguments from `args_new` list. To account for density()'s default behavior of "stretching range" by adding small tails, support of d-function is forced to be equal to `f`'s support (this is done with `form_resupport()` and method "reflect"). Output represents a "smooth" version of `f` as d-function.

Final output is computed by modifying "y" or "prob" column of `f`'s "x_tbl" metadata to be proportional to values of "smooth" output at corresponding points from "x" column. This way output distribution has exactly the same "x" grid as `f` but "more smooth" nature.

Value

A smoothed version of `f` with the same class and type.

See Also

Other form functions: `form_estimate()`, `form_mix()`, `form_regrid()`, `form_resupport()`, `form_retype()`, `form_tails()`, `form_trans()`

Examples

```
set.seed(101)

# Type "discrete"
bad_dis <- new_d(
  data.frame(x = sort(runif(100)), prob = runif(100)),
  type = "discrete"
)
smoothed_dis <- form_smooth(bad_dis)
plot(bad_dis)
lines(smoothed_dis, col = "blue")

# Type "continuous"
bad_con <- new_d(
  data.frame(x = sort(runif(100)), y = runif(100)),
  type = "continuous"
)
smoothed_con <- form_smooth(bad_con)
plot(bad_con)
lines(smoothed_con, col = "blue")
```

`form_tails` *Transform tails of distribution*

Description

Modify tail(s) of distribution defined by certain cutoff level using method of choice. This function is useful for doing robust analysis in presence of possible outliers.

Usage

```
form_tails(f, level, method = "trim", direction = "both")
```

Arguments

`f`	A pdqr-function.
`level`	Cutoff level. For direction "both" should be between 0 and 0.5; for "left" and "right" - between 0 and 1.
`method`	Modification method. One of "trim" or "winsor".
`direction`	Information about which tail(s) to modify. One of "both", "left", "right".

Details

Edges for left and right tails are computed as `level` and `1 - level` quantiles respectively. The left tail is interval to the left of left edge, and right tail - to the right of right edge.

Method "trim" removes tail(s) while normalizing "center part". Method "winsor" "squashes" tails inside center of distribution in dirac-like fashion, i.e. probability of tail(s) is moved inside and becomes concentrated in `1e-8` neighborhood of nearest edge.

Direction "both" affect both tails. Directions "left" and "right" affect only left and right tail respectively.

Value

A pdqr-function with transformed tail(s).

See Also

`form_resupport()` for changing support to some known interval.

`summ_center()` and `summ_spread()` for computing summaries of distributions.

Other form functions: `form_estimate()`, `form_mix()`, `form_regrid()`, `form_resupport()`, `form_retype()`, `form_smooth()`, `form_trans()`

Examples

```
# Type "discrete"
my_dis <- new_d(data.frame(x = 1:4, prob = (1:4)/10), type = "discrete")
meta_x_tbl(form_tails(my_dis, level = 0.1))
meta_x_tbl(
  form_tails(my_dis, level = 0.35, method = "winsor", direction = "left")
)

# Type "continuous"
d_norm <- as_d(dnorm)
plot(d_norm)
lines(form_tails(d_norm, level = 0.1), col = "blue")
lines(
  form_tails(d_norm, level = 0.1, method = "winsor", direction = "right"),
  col = "green"
)

# Use `form_resupport()` and `as_q()` to remove different levels from both
# directions. Here 0.1 level tail from left is removed, and 0.05 level from
# right
new_supp <- as_q(d_norm)(c(0.1, 1-0.05))
form_resupport(d_norm, support = new_supp)

# Examples of robust mean
set.seed(101)
x <- rcauchy(1000)
d_x <- new_d(x, "continuous")
summ_mean(d_x)
  # Trimmed mean
```

```
summ_mean(form_tails(d_x, level = 0.1, method = "trim"))
  # Winsorized mean
summ_mean(form_tails(d_x, level = 0.1, method = "winsor"))
```

`form_trans` *Transform pdqr-function*

Description

Perform a transformation of pdqr-function(s) (which assumed to be independent).

Usage

```
form_trans(f_list, trans, ..., method = "random", n_sample = 10000,
  args_new = list())

form_trans_self(f, trans, ..., method = "random", args_new = list())
```

Arguments

`f_list`	A list consisting from pdqr-function(s) and/or single number(s). Should have at least one pdqr-function (see Details).
`trans`	Transformation function. Should take as many (vectorized) arguments as there are elements in `f_list` or a single argument for `form_trans_self()`. Should return numeric or logical values.
`...`	Extra arguments to `trans`.
`method`	Transformation method. One of "random" or "bruteforce".
`n_sample`	Number of elements to sample.
`args_new`	List of extra arguments for new_*() to control `density()`.
`f`	A pdqr-function.

Details

`form_trans_self()` is a thin wrapper for `form_trans()` that accepts a single pdqr-function instead of a list of them.

Class of output is chosen as class of first pdqr-function in `f_list`. Type of output is chosen to be "discrete" in case all input pdqr-functions have "discrete" type, and "continuous" otherwise.

Method "random" performs transformation using random generation of samples:

- **Generates a sample of size** `n_sample` **from every element of** `f_list` (if element is single number, it is repeated `n_sample` times).
- **Calls** `trans` with all generated samples (in order aligned with `f_list`). **Note** that output should be either numeric or logical and have `n_sample` elements (one for each combination of input values in "vectorized" fashion). So, for example, using `sum` directly is not possible as it returns only single number.

- **Creates output pdqr-function**. If output is logical, probability of being true is estimated as share of TRUE in output, and boolean pdqr-function is created (type "discrete" with "x" values equal to 0 and 1, and probabilities of being false and true respectively). If output is numeric, one of new_*() (suitable for output class) is called with arguments from `args_new` list.

Method "bruteforce":

- **Retypes input** pdqr-function to "discrete" type (using "piecelin" method).
- **Computes output for every combination of "x" values** (probability of which will be a product of corresponding probabilities).
- **Creates pdqr-function of type "discrete"** with suitable new_*() function.
- **Possibly retypes to "continuous" type** if output should have it (also with "piecelin" method).

Notes about "bruteforce" method:

- Its main advantage is that it is not random.
- It may start to be very memory consuming very quickly.
- It is usually useful when type of output function is "discrete". In case of "continuous" type, retyping from "discrete" to "continuous" might introduce big errors.
- Used "discrete" probabilities shouldn't be very small because they will be directly multiplied, which might cause numerical accuracy issues.

Value

A pdqr-function for transformed random variable.

See Also

Pdqr methods for S3 group generic functions for more accurate implementations of most commonly used functions. Some of them are direct (without randomness) and some of them use `form_trans()` with "random" method.

`form_regrid()` to increase/decrease granularity of pdqr-functions for method "bruteforce".

Other form functions: `form_estimate()`, `form_mix()`, `form_regrid()`, `form_resupport()`, `form_retype()`, `form_smooth()`, `form_tails()`

Examples

```
# Default "random" transformation
d_norm <- as_d(dnorm)
  # More accurate result would give use of `+` directly with: d_norm + d_norm
d_norm_2 <- form_trans(list(d_norm, d_norm), trans = `+`)
plot(d_norm_2)
lines(as_d(dnorm, sd = sqrt(2)), col = "red")

  # Input list can have single numbers
form_trans(list(d_norm, 100), trans = `+`)

  # Output of `trans` can be logical. Next example is random version of
  # `d_norm >= 0`.
```

```
form_trans(list(d_norm, 0), trans = `>=`)

# Transformation with "bruteforce" method
power <- function(x, n = 1) {x^n}
p_dis <- new_p(
  data.frame(x = 1:3, prob = c(0.1, 0.2, 0.7)),
  type = "discrete"
)

p_dis_sq <- form_trans_self(
  p_dis, trans = power, n = 2, method = "bruteforce"
)
meta_x_tbl(p_dis_sq)
  # Compare with "random" method
p_dis_sq_rand <- form_trans_self(p_dis, trans = power, n = 2)
meta_x_tbl(p_dis_sq_rand)

# `form_trans_self()` is a wrapper for `form_trans()`
form_trans_self(d_norm, trans = function(x) {2*x})
```

`meta` *Get metadata of pdqr-function*

Description

Tools for getting metadata of **pdqr-function**: a function which represents distribution with finite support and finite values of probability/density. The key metadata which defines underline distribution is **"x_tbl"**. If two pdqr-functions have the same "x_tbl" metadata, they represent the same distribution and can be converted to one another with as_*() family of functions.

Usage

```
meta_all(f)

meta_class(f)

meta_type(f)

meta_support(f)

meta_x_tbl(f)
```

Arguments

`f`	A pdqr-function.

Details

Internally storage of metadata is implemented as follows:

- Pdqr class is a first "appropriate" ("p", "d", "q", or "r") S3 class of pdqr-function. All "proper" pdqr-functions have full S3 class of the form: `c(cl, "pdqr", "function")`, where `cl` is pdqr class.
- Pdqr type, support, and "x_tbl" are stored into function's environment.

Value

`meta_all()` returns a list of all metadata. `meta_class()`, `meta_type()`, `meta_support`, and `meta_x_tbl()` return corresponding metadata.

Pdqr class

Pdqr class is returned by `meta_class()`. This can be one of "p", "d", "q", "r". Represents **how pdqr-function describes underlying distribution**:

- P-function (i.e. of class "p") returns value of cumulative distribution function (probability of random variable being not more than certain value) at points `q` (its numeric vector input). Internally it is implemented as direct integration of corresponding (with the same "x_tbl" metadata) d-function.
- D-function returns value of probability mass or density function (depending on pdqr type) at points `x` (its numeric vector input). Internally it is implemented by directly using "x_tbl" metadata (see section '"x_tbl" metadata' for more details).
- Q-function returns value of quantile function at points `p` (its numeric vector input). Internally it is implemented as inverse of corresponding p-function (returns the smallest "x" value which has cumulative probability not less than input).
- R-function generates random sample of size `n` (its single number input) from distribution. Internally it is implemented using inverse transform sampling: certain amount of points from standard uniform distribution is generated, and the output is values of corresponding q-function at generated points.

These names are chosen so as to follow base R convention of naming distribution functions. All pdqr-functions take only one argument with the same meaning as the first ones in base R. It has no other arguments specific to some parameters of distribution family. To emulate their other common arguments, use the following transformations (here `d_f` means a function of class "d", etc.):

- For `d_f(x, log = TRUE)` use `log(d_f(x))`.
- For `p_f(q, lower.tail = FALSE)` use `1 - p_f(q)`.
- For `p_f(q, log.p = TRUE)` use `log(p_f(q))`.
- For `q_f(p, lower.tail = FALSE)` use `q_f(1 - p)`.
- For `q_f(p, log.p = TRUE)` use `q_f(exp(p))`.

Pdqr type

Pdqr type is returned by `meta_type()`. This can be one of "discrete" or "continuous". Represents **type of underlying distribution**:

- Type "discrete" is used for distributions with finite number of outcomes. Functions with "discrete" type has a fixed set of "x" values ("x" column in "x_tbl" metadata) on which d-function returns possibly non-zero output (values from "prob" column in "x_tbl" metadata).
- Type "continuous" is used to represent continuous distributions with piecewise-linear density with finite values and on finite support. Density goes through points defined by "x" and "y" columns in "x_tbl" metadata.

Pdqr support

Pdqr support is returned by `meta_support()`. This is a numeric vector with two finite values. Represents **support of underlying distribution**: closed interval, outside of which d-function is equal to zero. **Note** that inside of support d-function can also be zero, which especially true for "discrete" functions.

Technically, pdqr support is range of values from "x" column of "x_tbl" metadata.

"x_tbl" metadata

Metadata "x_tbl" is returned by `meta_x_tbl()`. This is a key metadata which **completely defines distribution**. It is a data frame with three numeric columns, content of which partially depends on pdqr type.

Type "discrete" functions have "x_tbl" with columns "x", "prob", "cumprob". D-functions return a value from "prob" column for input which is very near (should be equal up to ten digits, defined by round(*, digits = 10)) to corresponding value of "x" column. Rounding is done to account for issues with representation of numerical values (see Note section of ==’s help page). For any other input, d-functions return zero.

Type "continuous" functions have "x_tbl" with columns "x", "y", "cumprob". D-functions return a value of piecewise-linear function passing through points that have "x" and "y" coordinates. For any value outside support (i.e. strictly less than minimum "x" and strictly more than maximum "x") output is zero.

Column "cumprob" always represents the probability of underlying random variable being not more than corresponding value in "x" column.

Change of metadata

All metadata of pdqr-functions are not meant to be changed directly. Also change of pdqr type, support, and "x_tbl" metadata will lead to a complete change of underlying distribution.

To change **pdqr class**, for example to convert p-function to d-function, use as_*() family of functions: `as_p()`, `as_d()`, `as_q()`, `as_r()`.

To change **pdqr type**, use `form_retype()`. It changes underlying distribution in the most suitable for user way.

To change **pdqr support**, use `form_resupport()` or `form_tails()`.

Change of **"x_tbl" metadata** is not possible, because basically it means creating completely new pdqr-function. To do that, supply data frame with "x_tbl" format suitable for desired "type" to appropriate new_*() function: `new_p()`, `new_d()`, `new_q()`, `new_r()`. Also, there is a `form_regrid()` function which will increase or decrease granularity of pdqr-function.

Examples

```
d_unif <- as_d(dunif)

str(meta_all(d_unif))

meta_class(d_unif)
meta_type(d_unif)
meta_support(d_unif)
head(meta_x_tbl(d_unif))
```

`methods-group-generic` *Pdqr methods for S3 group generic functions*

Description

There are custom methods implemented for three out of four S3 group generic functions: `Math`, `Ops`, `Summary`. **Note** that many of them have random nature with an idea of generating samples from input pdqr-functions, performing certain operation on them (results in one generated sample from desired random variable), and creating new pdqr-function with appropriate new_*() function. This is done with `form_trans()`, so all rules for determining class and type of output is taken from it.

Usage

```
## S3 method for class 'pdqr'
Math(x, ...)

## S3 method for class 'pdqr'
Ops(e1, e2)

## S3 method for class 'pdqr'
Summary(..., na.rm = FALSE)
```

Arguments

`x, e1, e2`	Objects.
`...`	Further arguments passed to methods.
`na.rm`	Logical: should missing values be removed?

Details

Customization of method behavior may be done using mechanism of options(). These are the possible options:

- `pdqr.group_gen.args_new`. This will be used as `args_new` argument for `form_trans()` in methods with random nature. Default is `list()`.
- `pdqr.group_gen.n_sample`. This will be used as `n_sample` argument for `form_trans()` in methods with random nature. Default is 10000.
- `pdqr.group_gen.repair_supp_method`. All methods that have random nature take care of output support by trying to "repair" it, because default use of new_*() functions returns a slightly bigger support than range of input sample (see Examples). Repairing is done with `form_resupport()` where target support is computed separately and `method` argument is controlled by this option (preferred ones are `"reflect"`, default, and `"trim"`). In most cases output support is computed directly based on special features of generic function. But for some difficult cases, like `gamma()`, `digamma()`, `lgamma()`, `psigamma()`, `^`, and `%%` it is a result of simulation (i.e. slightly random, which slightly increases random nature of those methods).

Value

All methods return pdqr-function which represents the result of applying certain function to random variable(s) described with input pdqr-function(s). **Note** that independence of input random variables is assumed, i.e. `f + f` is not the same as `2*f` (see Examples).

Math

This family of S3 generics represents mathematical functions. Most of the methods have **random nature**, except `abs()` and `sign()` which are computed directly. Output of `sign()` has "discrete" type with 3 "x" values: -1, 0, 1.

Note that `cumsum()`, `cumprod()`, `cummmax()`, and `cummin()` functions don't make much sense in these implementations: their outputs represent random variable, sample of which is computed by applying cum*() function to a sample, generated from input pdqr-function.

Ops

This family of S3 generics represents common operators. For all functions (except `&` and `|`) input can be a pdqr-function or single number.

A list of methods with **non-random nature**:

- `!`, `+`, `-` in case of single input, i.e. `!f` or `-f`.
- Functions representing linear transformation, i.e. adding, subtracting, multiplying, and dividing by a single number. For example, all `f + 1`, `2 - f` (which is actually `(-f) + 2`), `3*f` and `f/2` are linear transformations, but `1 / f`, `f + g` are not.
- Functions for comparing: `==`, `!=`, `<`, `<=`, `>=`, `>`. Their output is **boolean pdqr-function**: "discrete" type function with elements being exactly 0 and 1. Probability of 0 represents probability of operator output being false, and 1 - being true. Probability of being true is computed directly as **limit of empirical estimation from simulations** (as size of samples grows to infinity). In other words, output is an exact number which might be approximated by simulating

two big samples of same size from input `e1` and `e2` (one of which can be a single number), and estimating probability as share of those pairs from samples for which comparison is true. **Note** that if at least one input has "continuous" type, then:

  - `==` will always have probability 0 of being true because probability of generating a certain exact one or two numbers from continuous random variable is zero.
  - `!=` will always have probability 1 of being true for the same reason as above.
  - Pairs `>=` and `>`, `<=` and `<` will return the same input because probability of being equal is always zero.

- Logical functions `&` and `|`. Their input can be only pdqr-functions (because single number input doesn't make much sense). They are most useful for applying to boolean pdqr-functions (see description of functions for comparing), and warning is thrown in case any input is not a boolean pdqr-function. `&`'s probability of being true is a product of those probabilities from input `e1` and `e2`. `|`'s probability of being false is a product of those probabilities from input `e1` and `e2`. **Note** that probability of being false is a probability of being equal to 0; of being true - complementary to that.

All other methods are **random**. For example, `f + f`, `f^g` are random.

Summary

Methods for `all()` and `any()` have **non-random nature**. Their input can be only pdqr-functions, and if any of them is not boolean, a warning is thrown (because otherwise output doesn't make much sense). They return a boolean pdqr-function with the following probability of being true:

- In `all()` - probability of *all* input function being true, i.e. product of probabilities of being true (implemented as complementary to probability of being equal to 0).
- In `any()` - probability of *any* input function being true, i.e. complementary probability to product of all functions being false (implemented as probability of being equal to 0).

Methods for `sum()`, `prod()`, `min()`, `max()` have **random nature**. They are implemented to use vectorized version of certain generic, because transformation function for `form_trans()` should be vectorized: for input samples which all have size n it should also return sample of size n (where each element is a transformation output for corresponding elements from input samples). This way `min(f,g)` can be read as "random variable representing minimum of `f` and `g`", etc.

Notes:

- `range()` function doesn't make sense here because it returns 2 numbers per input and therefore can't be made vectorized. Error is thrown if it is applied to pdqr-function.
- Although all `sum()`, `prod()`, `min()`, `max()` accept pdqr-functions or single numbers, using numbers and "continuous" functions simultaneously is not a great idea. This is because output will be automatically smoothed (as `form_trans()` will use some new_*() function) which will give a misleading picture. For a more realistic output:
  - Instead of `min(f,num)` use `form_resupport(f,c(num,NA),method = "winsor")` (see `form_resupport()`).
  - Instead of `max(f,num)` use `form_resupport(f,c(NA,num),method = "winsor")`.
  - Instead of `sum(f,num)` use `f + num`.
  - Instead of `prod(f,num)` use `f * num`.

See Also

summm_prob_true() and summ_prob_false() for extracting probability from boolean pdqr-functions.

Other pdqr methods for generic functions: methods-plot, methods-print

Examples

```
d_norm <- as_d(dnorm)
d_unif <- as_d(dunif)
d_dis <- new_d(data.frame(x = 1:4, prob = 1:4 / 10), "discrete")

set.seed(101)

# Math
plot(d_norm, main = "Math methods")
  # `abs()` and `sign()` are not random
lines(abs(d_norm), col = "red")
  # All others are random
lines(cos(d_norm), col = "green")
lines(cos(d_norm), col = "blue")

  # Although here distribution shouldn't change, it changes slightly due to
  # random implementation
meta_x_tbl(d_dis)
meta_x_tbl(floor(d_dis))

# Ops
  # Single input, linear transformations, and logical are not random
d_dis > 1
!(d_dis > 1)
d_norm >= (2*d_norm+1)
  # All others are random
plot(d_norm + d_norm)
  # This is an exact reference curve
lines(as_d(dnorm, sd = sqrt(2)), col = "red")

plot(d_dis + d_norm)

plot(d_unif^d_unif)

# Summary
  # `all()` and `any()` are non-random
all(d_dis > 1, d_dis > 1)
  # Others are random
plot(max(d_norm, d_norm, d_norm))

plot(d_norm + d_norm + d_norm)
lines(sum(d_norm, d_norm, d_norm), col = "red")

  # Using single numbers is allowed, but gives misleading output in case of
  # "continuous" functions. Use other functions instead (see documentation).
plot(min(d_unif, 0.5))
lines(form_resupport(d_unif, c(NA, 0.5), method = "winsor"), col = "blue")
```

```
# Use `options()` to control methods
plot(d_unif + d_unif)
op <- options(
  pdqr.group_gen.n_sample = 100,
  pdqr.group_gen.args_new = list(adjust = 0.5)
)
lines(d_unif + d_unif, col = "red")
  # `f + f` is different from `2*f` due to independency assumption. Also the
  # latter implemented non-randomly.
lines(2 * d_unif, col = "blue")

# Methods for generics attempt to repair support, so they are more reasonable
# to use than direct use of `form_trans()`
d_unif + d_unif
form_trans(list(d_unif, d_unif), `+`)
```

`methods-plot` *Pdqr methods for base plotting functions*

Description

Pdqr-functions have their own methods for `plot()` and `lines()` (except r-functions, see Details).

Usage

```
## S3 method for class 'p'
plot(x, y = NULL, n_extra_grid = 1001, ...)

## S3 method for class 'd'
plot(x, y = NULL, n_extra_grid = 1001, ...)

## S3 method for class 'q'
plot(x, y = NULL, n_extra_grid = 1001, ...)

## S3 method for class 'r'
plot(x, y = NULL, n_sample = 1000, ...)

## S3 method for class 'p'
lines(x, n_extra_grid = 1001, ...)

## S3 method for class 'd'
lines(x, n_extra_grid = 1001, ...)

## S3 method for class 'q'
lines(x, n_extra_grid = 1001, ...)
```

Arguments

x	Pdqr-function to plot.
y	Argument for compatibility with plot() signature. Doesn't used.
n_extra_grid	Number of extra grid points at which to evaluate pdqr-function (see Details). Supply NULL or 0 to not use extra grid.
...	Other arguments for plot() or hist() (in case of plotting r-function).
n_sample	Size of a sample to be generated for plotting histogram in case of an r-function.

Details

Main idea of plotting pdqr-functions is to use plotting mechanisms for appropriate numerical data.

Plotting of type **discrete** functions:

- P-functions are plotted as step-line with jumps at points of "x" column of "x_tbl" metadata.
- D-functions are plotted with vertical lines at points of "x" column of "x_tbl" with height equal to values from "prob" column.
- Q-functions are plotted as step-line with jumps at points of "cumprob" column of "x_tbl".
- R-functions are plotted by generating sample of size n_sample and calling hist() function.

Plotting of type **continuous** functions:

- P-functions are plotted in piecewise-linear fashion at their values on compound grid: sorted union of "x" column from "x_tbl" metadata and sequence of length n_extra_grid consisting from equidistant points between edges of support. Here extra grid is needed to show curvature of lines between "x" points from "x_tbl" (see Examples).
- D-functions are plotted in the same way as p-functions.
- Q-functions are plotted similarly as p- and d-functions but grid consists from union of "cumprob" column of "x_tbl" metadata and equidistant grid of length n_extra_grid from 0 to 1.
- R-functions are plotted the same way as type "discrete" ones: as histogram of generated sample of size n_sample.

Value

Output of invisible() without arguments, i.e. NULL without printing.

See Also

Other pdqr methods for generic functions: methods-group-generic, methods-print

Examples

```
d_norm_1 <- as_d(dnorm)
d_norm_2 <- as_d(dnorm, mean = 1)

plot(d_norm_1)
lines(d_norm_2, col = "red")
```

```
# Usage of `n_extra_grid` is important in case of "continuous" p- and
# q-functions
simple_p <- new_p(data.frame(x = c(0, 1), y = c(0, 1)), "continuous")
plot(simple_p, main = "Case study of n_extra_grid argument")
lines(simple_p, n_extra_grid = 0, col = "red")

# R-functions are plotted with histogram
plot(as_r(d_norm_1))
```

`methods-print` *Pdqr methods for print function*

Description

Pdqr-functions have their own methods for `print()` which displays function's metadata in readable and concise form.

Usage

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

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

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

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

Arguments

`x`	Pdqr-function to print.
`...`	Further arguments passed to or from other methods.

Details

Print output of pdqr-function describes the following information:

- Full name of function class:
  - P-function is "Cumulative distribution function".
  - D-function is "Probability mass function" for "discrete" type and "Probability density function" for "continuous".
  - Q-function is "Quantile function".
  - R-function is "Random generation function".

- Type of function in the form "of * type" where "*" is "discrete" or "continuous" depending on actual type.
- Support of function.
- Number of elements in distribution for "discrete" type or number of intervals of piecewise-linear density for "continuous" type.
- If pdqr-function has "discrete" type and exactly two possible values 0 and 1, it is treated as "boolean" pdqr-function and probability of 1 is shown. This is done to simplify interactive work with output of comparing functions like >=, etc. (see description of methods for S3 group generic functions). To extract probabilities from "boolean" pdqr-function, use `summ_prob_true()` and `summ_prob_false()`.

Symbol "~" in `print()` output indicates that printed value or support is an approximation to a true one (for readability purpose).

See Also

Other pdqr methods for generic functions: `methods-group-generic`, `methods-plot`

Examples

```
print(new_d(1:10, "discrete"))

r_unif <- as_r(runif, n_grid = 251)
print(r_unif)

# Printing of boolean pdqr-function
print(r_unif >= 0.3)
```

`new_p` *Create new pdqr-function*

Description

Functions for creating new pdqr-functions based on numeric sample or data frame describing distribution. They construct appropriate "x_tbl" metadata based on the input and then create pdqr-function (of corresponding pdqr class) defined by that "x_tbl".

Usage

```
new_p(x, type, ...)

new_d(x, type, ...)

new_q(x, type, ...)

new_r(x, type, ...)
```

Arguments

x	Numeric vector or data frame with appropriate columns (see "Data frame input" section).
type	Type of pdqr-function. Should be one of "discrete" or "continuous".
...	Extra arguments for density().

Details

Data frame input x is treated as having enough information for creating (including normalization of "y" column) an "x_tbl" metadata. For more details see "Data frame input" section.

Numeric input is transformed into data frame which is then used as "x_tbl" metadata (for more details see "Numeric input" section):

- If type is "discrete" then x is viewed as sample from distribution that can produce only values from x. Input is tabulated and normalized to form "x_tbl" metadata.
- If type is "continuous" then:
  - If x has 1 element, output distribution represents a **dirac-like** distribution which is an approximation to singular dirac distribution.
  - If x has more than 1 element, output distribution represents a **density estimation** with density() treating x as sample.

Value

A pdqr-function of corresponding class ("p" for new_p(), etc.) and type.

Numeric input

If x is a numeric vector, it is transformed into a data frame which is then used as "x_tbl" metadata to create pdqr-function of corresponding class.

First, all NaN, NA, and infinite values are removed with warnings. If there are no elements left, error is thrown. Then data frame is created in the way which depends on the type argument.

For "discrete" type elements of filtered x are:

- Rounded to 10th digit to avoid numerical representation issues (see Note in =='s help page).
- Tabulated (all unique values are counted). Output data frame has three columns: "x" with unique values, "prob" with normalized (divided by sum) counts, "cumprob" with cumulative sum of "prob" column.

For "continuous" type output data frame has columns "x", "y", "cumprob". Choice of algorithm depends on the number of x elements:

- If x has 1 element, an "x_tbl" metadata describes **dirac-like** "continuous" pdqr-function. It is implemented as triangular peak with center at x's value and width of 2e-8 (see Examples). This is an approximation of singular dirac distribution. Data frame has columns "x" with value c(x-1e-8,x,x+1e-8), "y" with value c(0,1e8,0) normalized to have total integral of "x"-"y" points of 1, "cumprob" c(0,0.5,1).

- If x has more than 1 element, it serves as input to density(x, ...) for density estimation (here arguments in ... of new_*() serve as extra arguments to density()). The output's "x" element is used as "x" column in output data frame. Column "y" is taken as "y" element of density() output, normalized so that piecewise-linear function passing through "x"-"y" points has total integral of 1. Column "cumprob" has cumulative probability of piecewise-linear d-function.

Data frame input

If x is a data frame, it should have numeric columns appropriate for "x_tbl" metadata of input type: "x", "prob" for "discrete" type and "x", "y" for "continuous" type ("cumprob" column will be computed inside new_*()). To become an appropriate "x_tbl" metadata, input data frame is ordered in increasing order of "x" column and then **imputed** in the way which depends on the type argument.

For "discrete" type:

- Values in column "x" are rounded to 10th digit to avoid numerical representation issues (see Note in =='s help page).
- If there are duplicate values in "x" column, they are "squashed" into one having sum of their probability in "prob" column.
- Column "prob" is normalized by its sum to have total sum of 1.
- Column "cumprob" is computed as cumulative sum of "prob" column.

For "continuous" type column "y" is normalized so that piecewise-linear function passing through "x"-"y" points has total integral of 1. Column "cumprob" has cumulative probability of piecewise-linear d-function.

Examples

```
set.seed(101)
x <- rnorm(10)

# Type "discrete": `x` values are directly tabulated
my_d_dis <- new_d(x, "discrete")
meta_x_tbl(my_d_dis)
plot(my_d_dis)

# Type "continuous": `x` serves as input to `density()`
my_d_con <- new_d(x, "continuous")
head(meta_x_tbl(my_d_con))
plot(my_d_con)

# Data frame input
  # Values in "prob" column will be normalized automatically
my_p_dis <- new_p(data.frame(x = 1:4, prob = 1:4), "discrete")
  # As are values in "y" column
my_p_con <- new_p(data.frame(x = 1:3, y = c(0, 10, 0)), "continuous")

# Using bigger bandwidth in `density()`
my_d_con_2 <- new_d(x, "continuous", adjust = 2)
plot(my_d_con, main = "Comparison of density bandwidths")
```

```
lines(my_d_con_2, col = "red")

# Dirac-like "continuous" pdqr-function is created if `x` is a single number
meta_x_tbl(new_d(1, "continuous"))
```

---

pdqr_approx_error *Diagnose pdqr approximation*

Description

pdqr_approx_error() computes errors that are results of 'pdqr' approximation, which occurs because of possible tail trimming and assuming piecewise linearity of density function in case of "continuous" type. For an easy view summary, use summary().

Usage

```
pdqr_approx_error(f, ref_f, ..., gran = 10, remove_infinity = TRUE)
```

Arguments

f	A p-, d-, or q-function to diagnose. Usually the output of one of as_p(), as_d(), or as_q() default methods.
ref_f	A "true" distribution function of the same class as f. Usually the input to the aforementioned as_*() function.
...	Other arguments to ref_f. If they were supplied to as_*() function, then the exact same values must be supplied here.
gran	Degree of grid "granularity" in case of "continuous" type: number of subintervals to be produced inside every interval of density linearity. Should be not less than 1 (indicator that original column from "x_tbl" will be used, see details).
remove_infinity	Whether to remove rows corresponding to infinite error.

Details

Errors are computed as difference between "true" value (output of ref_f) and output of pdqr-function f. They are computed at "granulated" gran times grid (which is an "x" column of "x_tbl" in case f is p- or d-function and "cumprob" column if q-function). They are usually negative because of possible tail trimming of reference distribution.

Notes:

- gran argument for "discrete" type is always 1.
- Quantile pdqr approximation of "discrete" distribution with infinite tale(s) can result into "all one" summary of error. This is expected output and is because test grid is chosen to be quantiles of pdqr-distribution which due to renormalization can differ by one from reference ones. For example: summary(pdqr_approx_error(as_p(ppois,lambda = 10),ppois,lambda = 10)).

Value

A data frame with the following columns:

- **grid** <dbl> : A grid at which errors are computed.
- **error** <dbl> : Errors which are computed as `ref_f(grid, ...) - f(grid)`.
- **abserror** <dbl> : Absolute value of "error" column.

See Also

`enpoint()` for representing pdqr-function as a set of points with desirable number of rows.

Examples

```
d_norm <- as_d(dnorm)
error_norm <- pdqr_approx_error(d_norm, dnorm)
summary(error_norm)

# Setting `gran` results into different number of rows in output
error_norm_2 <- pdqr_approx_error(d_norm, dnorm, gran = 1)
nrow(meta_x_tbl(d_norm)) == nrow(error_norm_2)

# By default infinity errors are removed
d_beta <- as_d(dbeta, shape1 = 0.3, shape2 = 0.7)
error_beta_1 <- pdqr_approx_error(d_beta, dbeta, shape1 = 0.3, shape2 = 0.7)
summary(error_beta_1)

# To not remove them, set `remove_infinity` to `FALSE`
error_beta_2 <- pdqr_approx_error(
  d_beta, dbeta, shape1 = 0.3, shape2 = 0.7, remove_infinity = FALSE
)
summary(error_beta_2)
```

---

region *Work with regions*

Description

These functions provide ways of working with a **region**: a data frame with numeric "left" and "right" columns, each row of which represents a unique finite interval (open, either type of half-open, or closed). Values of "left" and "right" columns should create an "ordered" set of intervals: left[1] <= right[1] <= left[2] <= right[2] <= ... (intervals with zero width are accepted). Originally, region_*() functions were designed to work with output of `summ_hdr()` and `summ_interval()`, but can be used for any data frame which satisfies the definition of a region.

Usage

```
region_is_in(region, x, left_closed = TRUE, right_closed = TRUE)

region_prob(region, f, left_closed = TRUE, right_closed = TRUE)

region_height(region, f, left_closed = TRUE, right_closed = TRUE)

region_width(region)

region_draw(region, col = "blue", alpha = 0.2)
```

Arguments

region	A data frame representing region.
x	Numeric vector to be tested for being inside region.
left_closed	A single logical value representing whether to treat left ends of intervals as their parts.
right_closed	A single logical value representing whether to treat right ends of intervals as their parts.
f	A pdqr-function.
col	Single color of rectangles to be used. Should be appropriate for col argument of col2rgb().
alpha	Single number representing factor modifying the opacity alpha; typically in [0; 1].

Details

region_is_in() tests each value of x for being inside interval. In other words, if there is a row for which element of x is between "left" and "right" value (respecting left_closed and right_closed options), output for that element will be TRUE. **Note** that for zero-width intervals one of left_closed or right_closed being TRUE is enough to accept that point as "in region".

region_prob() computes total probability of region according to pdqr-function f. If f has "discrete" type, output is computed as sum of probabilities for all "x" values from "x_tbl" metadata which lie inside a region (respecting left_closed and right_closed options while using region_is_in()). If f has "continuous" type, output is computed as integral of density over a region (*_closed options having any effect).

region_height() computes "height" of a region (with respect to f): minimum value of corresponding to f d-function can return based on relevant points inside a region. If f has "discrete" type, those relevant points are computed as "x" values from "x_tbl" metadata which lie inside a region (if there are no such points, output is 0). If f has "continuous" type, the whole intervals are used as relevant points. The notion of "height" comes from summ_hdr() function: if region is summ_hdr(f,level) for some level, then region_height(region,f) is what is called in summ_hdr()'s docs as "target height" of HDR. That is, a maximum value of d-function for which a set consisting from points at which d-function has values not less than target height and total probability of the set being not less than level.

`region_width()` computes total width of a region, i.e. sum of differences between "right" and "left" columns.

`region_draw()` draws (on current plot) intervals stored in `region` as colored rectangles vertically starting from zero and ending in the top of the plot (technically, at "y" value of `2e8`).

Value

`region_is_in()` returns a logical vector (with length equal to length of `x`) representing whether certain element of `x` is inside a region.

`region_prob()` returns a single number between 0 and 1 representing total probability of region.

`region_height()` returns a single number representing a height of a region with respect to `f`, i.e. minimum value that corresponding d-function can return based on relevant points inside a region.

`region_width()` returns a single number representing total width of a region.

`region_draw()` draws colored rectangles filling `region` intervals.

See Also

`summ_hdr()` for computing of Highest Density Region.

`summ_interval()` for computing of single interval summary of distribution.

Examples

```
# Type "discrete"
d_binom <- as_d(dbinom, size = 10, prob = 0.7)
hdr_dis <- summ_hdr(d_binom, level = 0.6)
region_is_in(hdr_dis, 0:10)
  # This should be not less than 0.6
region_prob(hdr_dis, d_binom)
region_height(hdr_dis, d_binom)
region_width(hdr_dis)

# Type "continuous"
d_norm <- as_d(dnorm)
hdr_con <- summ_hdr(d_norm, level = 0.95)
region_is_in(hdr_con, c(-Inf, -2, 0, 2, Inf))
  # This should be approximately equal to 0.95
region_prob(hdr_con, d_norm)
  # This should be equal to `d_norm(hdr_con[["left"]][1])`
region_height(hdr_con, d_norm)
region_width(hdr_con)

# Usage of `*_closed` options
region <- data.frame(left = 1, right = 3)
  # Closed intervals
region_is_in(region, 1:3)
  # Open from left, closed from right
region_is_in(region, 1:3, left_closed = FALSE)
  # Closed from left, open from right
region_is_in(region, 1:3, right_closed = FALSE)
  # Open intervals
```

```
region_is_in(region, 1:3, left_closed = FALSE, right_closed = FALSE)

# Handling of intervals with zero width
region <- data.frame(left = 1, right = 1)
  # If at least one of `*_closed` options is `TRUE`, 1 will be considered as
  # "in a region"
region_is_in(region, 1)
region_is_in(region, 1, left_closed = FALSE)
region_is_in(region, 1, right_closed = FALSE)
  # Only this will return `FALSE`
region_is_in(region, 1, left_closed = FALSE, right_closed = FALSE)

# Drawing
d_mix <- form_mix(list(as_d(dnorm), as_d(dnorm, mean = 5)))
plot(d_mix)
region_draw(summ_hdr(d_mix, 0.95))
```

`summ_center` *Summarize distribution with center*

Description

Functions to compute center of distribution. `summ_center()` is a wrapper for respective summ_*() functions (from this page) with default arguments.

Usage

```
summ_center(f, method = "mean")

summ_mean(f)

summ_median(f)

summ_mode(f, method = "global")
```

Arguments

`f`	A pdqr-function representing distribution.
`method`	Method of center computation. For `summ_center()` is one of "mean", "median", "mode". For `summ_mode()` is one of "global" or "local".

Details

`summ_mean()` computes distribution's mean.

`summ_median()` computes a smallest `x` value for which cumulative probability is not less than 0.5. Essentially, it is a `as_q(f)(0.5)`. This also means that for pdqr-functions with type "discrete" it always returns an entry of "x" column from `f`'s "x_tbl" metadata.

summ_mode(*, method = "global") computes a smallest x (which is an entry of "x" column from f's x_tbl) with the highest probability/density. summ_mode(*, method = "local") computes all x values which represent non-strict **local maxima** of probability mass/density function.

Value

summ_center(), summ_mean(), summ_median() and summ_mode(*, method = "global") always return a single number representing a center of distribution. summ_mode(*, method = "local") can return a numeric vector with multiple values representing local maxima.

See Also

summ_spread() for computing distribution's spread, summ_moment() for general moments.

Other summary functions: summ_classmetric(), summ_distance(), summ_entropy(), summ_hdr(), summ_interval(), summ_moment(), summ_order(), summ_prob_true(), summ_pval(), summ_quantile(), summ_roc(), summ_separation(), summ_spread()

Examples

```
# Type "continuous"
d_norm <- as_d(dnorm)
  # The same as `summ_center(d_norm, method = "mean")`
summ_mean(d_norm)
summ_median(d_norm)
summ_mode(d_norm)

# Type "discrete"
d_pois <- as_d(dpois, lambda = 10)
summ_mean(d_pois)
summ_median(d_pois)
  # Returns the smallest `x` with highest probability
summ_mode(d_pois)
  # Returns all values which are non-strict local maxima
summ_mode(d_pois, method = "local")

# Details of computing local modes
my_d <- new_d(data.frame(x = 11:15, y = c(0, 1, 0, 2, 0)/3), "continuous")
  # Several values, which are entries of `x_tbl`, are returned as local modes
summ_mode(my_d, method = "local")
```

---

summ_classmetric *Summarize pair of distributions with classification metric*

Description

Compute metric of the following one-dimensional binary classification setup: any x value not more than threshold value is classified as "negative"; if strictly greater - "positive". Classification metrics are computed based on two pdqr-functions: f, which represents the distribution of values which *should be* classified as "negative" ("true negative"), and g - the same for "positive" ("true positive").

Usage

```
summ_classmetric(f, g, threshold, method = "F1")

summ_classmetric_df(f, g, threshold, method = "F1")
```

Arguments

f	A pdqr-function of any type and class. Represents distribution of "true negative" values.
g	A pdqr-function of any type and class. Represents distribution of "true positive" values.
threshold	A numeric vector of classification threshold(s).
method	Method of classification metric (might be a vector for summ_classmetric_df()). Should be one of "TPR", "TNR", "FPR", "FNR", "PPV", "NPV", "FDR", "FOR", "LR+", "LR-", "Acc", "ER", "GM", "F1", "OP", "MCC", "YI", "MK", "Jaccard", "DOR" (with possible aliases, see Details).

Details

Binary classification setup used here to compute metrics is a simplified version of the most common one, when there is a finite set of already classified objects. Usually, there are N objects which are truly "negative" and P truly "positive" ones. Values N and P can vary, which often results in class imbalance. However, in current setup both N and P are equal to 1 (total probability of f and g).

In common setup, classification of all N + P objects results into the following values: "TP" (number of truly "positive" values classified as "positive"), "TN" (number of negatives classified as "negative"), "FP" (number of negatives falsely classified as "positive"), and "FN" (number of positives falsely classified as "negative"). In current setup all those values are equal to respective "rates" (because N and P are both equal to 1).

Both summ_classmetric() and summ_classmetric_df() allow aliases to some classification metrics (for readability purposes).

Following classification metrics are available:

- Simple metrics:
  - *True positive rate*, method "TPR" (aliases: "TP", "sensitivity", "recall"): proportion of actual positives correctly classified as such. Computed as 1 - as_p(g)(threshold).
  - *True negative rate*, method "TNR" (aliases: "TN", "specificity"): proportion of actual negatives correctly classified as such. Computed as as_p(f)(threshold).
  - *False positive rate*, method "FPR" (aliases: "FP", "fall-out"): proportion of actual negatives falsely classified as "positive". Computed as 1 - as_p(f)(threshold).
  - *False negative rate*, method "FNR" (aliases: "FN", "miss_rate"): proportion of actual positives falsely classified as "negative". Computed as as_p(g)(threshold).
  - *Positive predictive value*, method "PPV" (alias: "precision"): proportion of output positives that are actually "positive". Computed as TP / (TP + FP).
  - *Negative predictive value*, method "NPV": proportion of output negatives that are actually "negative". Computed as TN / (TN + FN).

  - *False discovery rate*, `method` "FDR": proportion of output positives that are actually "negative". Computed as `FP / (TP + FP)`.
  - *False omission rate*, `method` "FOR": proportion of output negatives that are actually "positive". Computed as `FN / (TN + FN)`.
  - *Positive likelihood*, `method` "LR+": measures how much the odds of being "positive" increase when value is classified as "positive". Computed as `TPR / (1 - TNR)`.
  - *Negative likelihood*, `method` "LR-": measures how much the odds of being "positive" decrease when value is classified as "negative". Computed as `(1 - TPR) / TNR`.

- Combined metrics (for all, except "error rate", bigger value represents better classification performance):
  - *Accuracy*, `method` "Acc" (alias: "accuracy"): proportion of total number of input values that were correctly classified. Computed as `(TP + TN) / 2` (here 2 is used because of special classification setup, `TP + TN + FP + FN = 2`).
  - *Error rate*, `method` "ER" (alias: "error_rate"): proportion of total number of input values that were incorrectly classified. Computed as `(FP + FN) / 2`.
  - *Geometric mean*, `method` "GM": geometric mean of TPR and TNR. Computed as `sqrt(TPR * TNR)`.
  - *F1 score*, `method` "F1": harmonic mean of PPV and TPR. Computed as `2*TP / (2*TP + FP + FN)`.
  - *Optimized precision*, `method` "OP": accuracy, penalized for imbalanced class performance. Computed as `Acc - abs(TPR - TNR) / (TPR + TNR)`.
  - *Matthews correlation coefficient*, `method` "MCC" (alias: "corr"): correlation between the observed and predicted classifications. Computed as `(TP*TN - FP*FN) / sqrt((TP+FP) * (TN+FN))` (here equalities `TP+FN = 1` and `TN+FP = 1` are used to simplify formula).
  - *Youden's index*, `method` "YI" (aliases: "youden", "informedness"): evaluates the discriminative power of the classification setup. Computed as `TPR + TNR - 1`.
  - *Markedness*, `method` "MK" (alias: "markedness"): evaluates the predictive power of the classification setup. Computed as `PPV + NPV - 1`.
  - *Jaccard*, `method` "Jaccard": accuracy ignoring correct classification of negatives. Computed as `TP / (TP + FP + FN)`.
  - *Diagnostic odds ratio*, `method` "DOR" (alias: "odds_ratio"): ratio between positive and negative likelihoods. Computed as `"LR+" / "LR-"`.

Value

`summ_classmetric()` returns a numeric vector, of the same length as `threshold`, representing classification metrics for different threshold values.

`summ_classmetric_df()` returns a data frame with rows corresponding to `threshold` values. First column is "threshold" (with `threshold` values), and all other represent classification metric for every input method (see Examples).

See Also

summ_separation for computing optimal separation threshold (which is symmetrical with respect to `f` and `g`).

Other summary functions: summ_center(), summ_distance(), summ_entropy(), summ_hdr(), summ_interval(), summ_moment(), summ_order(), summ_prob_true(), summ_pval(), summ_quantile(), summ_roc(), summ_separation(), summ_spread()

Examples

```
d_unif <- as_d(dunif)
d_norm <- as_d(dnorm)
t_vec <- c(0, 0.5, 0.75, 1.5)

summ_classmetric(d_unif, d_norm, threshold = t_vec, method = "F1")
summ_classmetric(d_unif, d_norm, threshold = t_vec, method = "Acc")

summ_classmetric_df(
  d_unif, d_norm, threshold = t_vec, method = c("F1", "Acc")
)

# Using method aliases
summ_classmetric_df(
  d_unif, d_norm, threshold = t_vec, method = c("TPR", "sensitivity")
)
```

---

summ_distance *Summarize pair of distributions with distance*

Description

This function computes distance between two distributions represented by pdqr-functions. Here "distance" is used in a broad sense: a single non-negative number representing how much two distributions differ from one another. Bigger values indicate bigger difference. Zero value means that input distributions are equivalent based on the method used. The notion of "distance" is useful for doing statistical inference about similarity of two groups of numbers.

Usage

```
summ_distance(f, g, method = "KS")
```

Arguments

f	A pdqr-function of any type and class.
g	A pdqr-function of any type and class.
method	Method for computing distance. Should be one of "KS", "totvar", "compare", "wass", "cramer", "align", "entropy".

Details

Methods can be separated into three categories: probability based, metric based, and entropy based.

Probability based methods return a number between 0 and 1 which is computed in the way that mostly based on probability:

- *Method "KS"* (short for Kolmogorov-Smirnov) computes the supremum of absolute difference between p-functions corresponding to `f` and `g` (|F - G|). Here "supremum" is meant to describe the fact that if input functions have different types, there can be no point at which "KS" distance is achieved. Instead, there might be a sequence of points from left to right with |F - G| values tending to the result (see Examples).
- *Method "totvar"* (short for "total variation") computes a biggest absolute difference of probabilities for any subset of real line. In other words, there is a set of points for "discrete" type and intervals for "continuous", total probability of which under `f` and `g` differs the most. **Note** that if `f` and `g` have different types, output is always 1. The set of interest consists from all "x" values of "discrete" pdqr-function: probability under "discrete" distribution is 1 and under "continuous" is 0.
- *Method "compare"* represents a value computed based on probabilities of one distribution being bigger than the other (see pdqr methods for "Ops" group generic family for more details on comparing pdqr-functions). It is computed as `2*max(P(F > G),P(F < G)) + 0.5*P(F = G) -1` (here `P(F > G)` is basically `summ_prob_true(f > g)`). This is maximum of two values (`P(F > G) + 0.5*P(F = G)` and `P(F < G) + 0.5*P(F = G)`), normalized to return values from 0 to 1. Other way to look at this measure is that it computes (before normalization) two ROC AUC values with method `"expected"` for two possible ordering (f, g, and g, f) and takes their maximum.

Metric based methods compute "how far" two distributions are apart on the real line:

- *Method "wass"* (short for "Wasserstein") computes a 1-Wasserstein distance: "minimum cost of 'moving' one density into another", or "average path density point should go while transforming from one into another". It is computed as integral of |F - G| (absolute difference between p-functions). If any of `f` and `g` has "continuous" type, `stats::integrate()` is used, so relatively small numerical errors can happen.
- *Method "cramer"* computes Cramer distance: integral of `(F -G)^2`. This somewhat relates to "wass" method as variance relates to first central absolute moment. Relatively small numerical errors can happen.
- *Method "align"* computes an absolute value of shift `d` (possibly negative) that should be added to `f` to achieve both `P(f+d >= g) >= 0.5` and `P(f+d <= g) >= 0.5` (in other words, align `f+d` and `g`) as close as reasonably possible. Solution is found numerically with `stats::uniroot()`, so relatively small numerical errors can happen. Also **note** that this method is somewhat slow (compared to all others). To increase speed, use less elements in "x_tbl" metadata. For example, with `form_retype()` or smaller `n_grid` argument in as_*() functions.

Entropy based methods compute output based on entropy characteristics:

- *Method "entropy"* computes sum of two Kullback-Leibler divergences: `KL(f,g) + KL(g,f)`, which are outputs of `summ_entropy2()` with method "relative". **Notes**:
  - If `f` and `g` don't have the same support, distance can be very high.
  - Error is thrown if `f` and `g` have different types (the same as in `summ_entropy2()`).

Value

A single non-negative number representing distance between pair of distributions. For methods "KS", "totvar", and "compare" it is not bigger than 1.

See Also

`summ_separation()` for computation of optimal threshold separating pair of distributions.

Other summary functions: `summ_center()`, `summ_classmetric()`, `summ_entropy()`, `summ_hdr()`, `summ_interval()`, `summ_moment()`, `summ_order()`, `summ_prob_true()`, `summ_pval()`, `summ_quantile()`, `summ_roc()`, `summ_separation()`, `summ_spread()`

Examples

```
d_unif <- as_d(dunif, max = 2)
d_norm <- as_d(dnorm, mean = 1)

vapply(
  c("KS", "totvar", "compare", "wass", "cramer", "align", "entropy"),
  function(meth) {summ_distance(d_unif, d_norm, method = meth)},
  numeric(1)
)

# "Supremum" quality of "KS" distance
d_dis <- new_d(2, "discrete")
  # Distance is 1, which is a limit of |F - G| at points which tend to 2 from
  # left
summ_distance(d_dis, d_unif, method = "KS")
```

---

summ_entropy *Summarize distribution with entropy*

Description

`summ_entropy()` computes entropy of single distribution while `summ_entropy2()` - for a pair of distributions. For "discrete" pdqr-functions a classic formula `-sum(p * log(p))` (in nats) is used. In "continuous" case a differential entropy is computed.

Usage

```
summ_entropy(f)

summ_entropy2(f, g, method = "relative", clip = exp(-20))
```

Arguments

f	A pdqr-function representing distribution.
g	A pdqr-function. Should be the same type as f.
method	Entropy method for pair of distributions. One of "relative" (Kullback–Leibler divergence) or "cross" (for cross-entropy).
clip	Value to be used instead of 0 during log() computation. -log(clip) represents the maximum value of output entropy.

Details

Note that due to pdqr approximation error there can be a rather big error in entropy estimation in case original density goes to infinity.

Value

A single number representing entropy. If clip is strictly positive, then it will be finite.

See Also

Other summary functions: summ_center(), summ_classmetric(), summ_distance(), summ_hdr(), summ_interval(), summ_moment(), summ_order(), summ_prob_true(), summ_pval(), summ_quantile(), summ_roc(), summ_separation(), summ_spread()

Examples

```
d_norm <- as_d(dnorm)
d_norm_2 <- as_d(dnorm, mean = 2, sd = 0.5)

summ_entropy(d_norm)
summ_entropy2(d_norm, d_norm_2)
summ_entropy2(d_norm, d_norm_2, method = "cross")

# Increasing `clip` leads to decreasing maximum output value
d_1 <- new_d(1:10, "discrete")
d_2 <- new_d(20:21, "discrete")

  # Formally, output isn't clearly defined because functions don't have the
  # same support. Direct use of entropy formulas gives infinity output, but
  # here maximum value is `-log(clip)`.
summ_entropy2(d_1, d_2, method = "cross")
summ_entropy2(d_1, d_2, method = "cross", clip = exp(-10))
summ_entropy2(d_1, d_2, method = "cross", clip = 0)
```

summ_hdr *Summarize distribution with Highest Density Region*

Description

`summ_hdr()` computes a Highest Density Region (HDR) of some pdqr-function for a supplied `level`: a union of (closed) intervals total probability of which is not less than `level` and probability/density at any point inside it is bigger than some threshold (which should be maximum one with a property of HDR having total probability not less than `level`). This also represents a set of intervals with the lowest total width among all sets with total probability not less than a `level`.

Usage

```
summ_hdr(f, level = 0.95)
```

Arguments

`f`	A pdqr-function representing distribution.
`level`	A desired lower bound for a total probability of an output set of intervals.

Details

General algorithm of `summ_hdr()` consists from two steps:

1. **Find "target height"**. That is a value of probability/density which divides all support into two sets: the one with probability/density not less than target height (it is a desired HDR) and the other - with strictly less. The first set should also have total probability not less than `level`.
2. **Form a HDR as a set of closed intervals**.

If `f` has "discrete" type, target height is computed by looking at "x" values of "x_tbl" metadata in order of decreasing probability until their total probability is not less than `level`. After that, all "x" values with probability not less than height are considered to form a HDR. Output is formed as a set of **closed** intervals (i.e. both edges included) inside of which lie all HDR "x" elements and others - don't.

If `f` has "continuous" type, target height is estimated as `1-level` quantile of `Y = d_f(X)` distribution, where `d_f` is d-function corresponding to `f` (`as_d(f)` in other words) and `X` is a random variable represented by `f`. Essentially, `Y` has a distribution of `f`'s density values and its `1-level` quantile is a target height. After that, HDR is formed as a set of intervals **with positive width** (if `level` is more than 0, see Notes) inside which density is not less than target height.

Notes:

- If `level` is 0, output has one interval of zero width at point of global mode.
- If `level` is 1, output has one interval equal to support.
- Computation of target height in case of "continuous" type is approximate which in some extreme cases (for example, like winsorized distributions) can lead to HDR having total probability very approximate to and even slightly lower than `level`.

- If d-function has "plateaus" (consecutive values with equal probability/density) at computed target height, total probability of HDR can be considerably bigger than `level` (see examples). However, this aligns with HDR definition, as density values should be **not less** than target height and total probability should be **not less** than `level`.

Value

A data frame with one row representing one closed interval of HDR and the following columns:

- **left** <dbl> : Left end of intervals.
- **right** <dbl> : Right end of intervals.

See Also

`region_*()` family of functions for working with output HDR.

`summ_interval()` for computing of single interval summary of distribution.

Other summary functions: `summ_center()`, `summ_classmetric()`, `summ_distance()`, `summ_entropy()`, `summ_interval()`, `summ_moment()`, `summ_order()`, `summ_prob_true()`, `summ_pval()`, `summ_quantile()`, `summ_roc()`, `summ_separation()`, `summ_spread()`

Examples

```
# "discrete" functions
d_dis <- new_d(data.frame(x = 1:4, prob = c(0.4, 0.2, 0.3, 0.1)), "discrete")
summ_hdr(d_dis, 0.3)
summ_hdr(d_dis, 0.5)
summ_hdr(d_dis, 0.9)
  # Zero width interval at global mode
summ_hdr(d_dis, 0)

# "continuous" functions
d_norm <- as_d(dnorm)
summ_hdr(d_norm, 0.95)
  # Zero width interval at global mode
summ_hdr(d_norm, 0)

# Works well with mixture distributions
d_mix <- form_mix(list(as_d(dnorm), as_d(dnorm, mean = 5)))
summ_hdr(d_mix, 0.95)

# Plateaus
d_unif <- as_d(dunif)
  # Returns all support because of density "plateau"
summ_hdr(d_unif, 0.1)

# Draw HDR
plot(d_mix)
region_draw(summ_hdr(d_mix, 0.95))
```

`summ_interval` *Summarize distribution with interval*

Description

These functions summarize distribution with one interval based on method of choice.

Usage

```
summ_interval(f, level = 0.95, method = "minwidth", n_grid = 10001)
```

Arguments

`f`	A pdqr-function representing distribution.
`level`	A number between 0 and 1 representing a coverage degree of interval. Interpretation depends on `method` but the bigger is number, the wider is interval.
`method`	Method of interval computation. Should be on of "minwidth", "percentile", "sigma".
`n_grid`	Number of grid elements to be used for "minwidth" method (see Details).

Details

Method "minwidth" searches for an interval with total probability of `level` that has minimum width. This is done with grid search: `n_grid` possible intervals with `level` total probability are computed and the one with minimum width is returned (if there are several, the one with the smallest left end). Left ends of computed set of intervals are created as a grid from `0` to `1-level` quantiles with `n_grid` number of elements. Right ends are computed so that intervals have `level` total probability.

Method "percentile" returns an interval with edges being `0.5*(1-level)` and `1 -0.5*(1-level)` quantiles. Output has total probability equal to `level`.

Method "sigma" computes an interval symmetrically centered at mean of distribution. Left and right edges are distant from center by the amount of standard deviation multiplied by `level`'s critical value. Critical value is computed using normal distribution as `qnorm(1 -0.5*(1-level))`, which corresponds to a way of computing sample confidence interval with known standard deviation. The final output interval is possibly cut so that not to be out of `f`'s support.

Note that supported methods correspond to different ways of computing distribution's center. This idea is supported by the fact that when `level` is 0, "minwidth" method returns zero width interval at distribution's global mode, "percentile" method - median, "sigma" - mean.

Value

A region with one row. That is a data frame with one row and the following columns:

- **left** <dbl> : Left end of interval.
- **right** <dbl> : Right end of interval.

To return a simple numeric vector, call unlist() on `summ_interval()`'s output (see Examples).

See Also

summ_hdr() for computing of Highest Density Region, which can summarize distribution with multiple intervals.

region_*() family of functions for working with summ_interval() output.

Other summary functions: summ_center(), summ_classmetric(), summ_distance(), summ_entropy(), summ_hdr(), summ_moment(), summ_order(), summ_prob_true(), summ_pval(), summ_quantile(), summ_roc(), summ_separation(), summ_spread()

Examples

```
# Type "discrete"
d_dis <- new_d(data.frame(x = 1:6, prob = c(3:1, 0:2)/9), "discrete")
summ_interval(d_dis, level = 0.5, method = "minwidth")
summ_interval(d_dis, level = 0.5, method = "percentile")
summ_interval(d_dis, level = 0.5, method = "sigma")

  # Visual difference between methods
plot(d_dis)
region_draw(summ_interval(d_dis, 0.5, method = "minwidth"), col = "blue")
region_draw(summ_interval(d_dis, 0.5, method = "percentile"), col = "red")
region_draw(summ_interval(d_dis, 0.5, method = "sigma"), col = "green")

# Type "continuous"
d_con <- form_mix(
  list(as_d(dnorm), as_d(dnorm, mean = 5)),
  weights = c(0.25, 0.75)
)
summ_interval(d_con, level = 0.5, method = "minwidth")
summ_interval(d_con, level = 0.5, method = "percentile")
summ_interval(d_con, level = 0.5, method = "sigma")

  # Visual difference between methods
plot(d_con)
region_draw(summ_interval(d_con, 0.5, method = "minwidth"), col = "blue")
region_draw(summ_interval(d_con, 0.5, method = "percentile"), col = "red")
region_draw(summ_interval(d_con, 0.5, method = "sigma"), col = "green")

# Output interval is always inside input's support. Formally, next code
# should return interval from `-Inf` to `Inf`, but output is cut to be inside
# support.
summ_interval(d_con, level = 1, method = "sigma")

# To get vector output, use `unlist()`
unlist(summ_interval(d_con))
```

---

summ_moment *Summarize distribution with moment*

Description

summ_moment() computes a moment of distribution. It can be one of eight kinds determined by the combination of central, standard, and absolute boolean features. summ_skewness() and summ_kurtosis() are wrappers for commonly used kinds of moments: third and forth order central standard ones. **Note** that summ_kurtosis() by default computes excess kurtosis, i.e. subtracts 3 from computed forth order moment.

Usage

```
summ_moment(f, order, central = FALSE, standard = FALSE, absolute = FALSE)

summ_skewness(f)

summ_kurtosis(f, excess = TRUE)
```

Arguments

f	A pdqr-function representing distribution.
order	A single number representing order of a moment. Should be non-negative number (even fractional).
central	Whether to compute central moment (subtract mean of distribution).
standard	Whether to compute standard moment (divide by standard deviation of distribution).
absolute	Whether to compute absolute moment (take absolute value of random variable created after possible effect of central and standard).
excess	Whether to compute excess kurtosis (subtract 3 from third order central standard moment). Default is TRUE.

Value

A single number representing moment. If summ_sd(f) is zero and standard is TRUE, then it is Inf; otherwise - finite number.

See Also

summ_center() for computing distribution's center, summ_spread() for spread.

Other summary functions: summ_center(), summ_classmetric(), summ_distance(), summ_entropy(), summ_hdr(), summ_interval(), summ_order(), summ_prob_true(), summ_pval(), summ_quantile(), summ_roc(), summ_separation(), summ_spread()

Examples

```
d_beta <- as_d(dbeta, shape1 = 2, shape2 = 1)

# The same as `summ_mean(d_beta)`
summ_moment(d_beta, order = 1)

# The same as `summ_var(d_beta)`
```

```
summ_moment(d_beta, order = 2, central = TRUE)

# Return the same number
summ_moment(d_beta, order = 3, central = TRUE, standard = TRUE)
summ_skewness(d_beta)

# Return the same number representing non-excess kurtosis
summ_moment(d_beta, order = 4, central = TRUE, standard = TRUE)
summ_kurtosis(d_beta, excess = FALSE)
```

`summ_order` *Summarize list of pdqr-functions with order*

Description

Functions for ordering the set of pdqr-functions supplied in a list. This might be useful for doing comparative statistical inference for several groups of data.

Usage

```
summ_order(f_list, method = "compare", decreasing = FALSE)

summ_sort(f_list, method = "compare", decreasing = FALSE)

summ_rank(f_list, method = "compare")
```

Arguments

`f_list`	List of pdqr-functions.
`method`	Method to be used for ordering. Should be one of "compare", "mean", "median", "mode".
`decreasing`	If `TRUE` ordering is done decreasingly.

Details

Ties for all methods are handled so as to preserve the original order.

Method "compare" is using the following ordering relation: pdqr-function `f` is greater than `g` if and only if `P(f >= g) > 0.5`, or in code `summ_prob_true(f >= g) > 0.5` (see pdqr methods for "Ops" group generic family for more details on comparing pdqr-functions). This method orders input based on this relation and order() function. **Notes**:

- This relation doesn't define strictly ordering because it is not transitive: there can be pdqr-functions `f`, `g`, and `h`, for which `f` is greater than `g`, `g` is greater than `h`, and `h` is greater than `f` (but should be otherwise). If not addressed, this might result into dependence of output on order of the input. It is solved by first preordering `f_list` based on method "mean" and then calling `order()`.

- Because comparing two pdqr-functions can be time consuming, this method becomes rather slow as number of `f_list` elements grows.

Methods "mean", "median", and "mode" are based on `summ_center()`: ordering of `f_list` is defined as ordering of corresponding measures of distribution's center.

Value

`summ_order()` works essentially like order(). It returns an integer vector representing a permutation which rearranges `f_list` in desired order.

`summ_sort()` returns a sorted (in desired order) variant of `f_list`.

`summ_rank()` returns a numeric vector representing ranks of `f_list` elements: 1 for the "smallest", `length(f_list)` for the "biggest".

See Also

Other summary functions: `summ_center()`, `summ_classmetric()`, `summ_distance()`, `summ_entropy()`, `summ_hdr()`, `summ_interval()`, `summ_moment()`, `summ_prob_true()`, `summ_pval()`, `summ_quantile()`, `summ_roc()`, `summ_separation()`, `summ_spread()`

Examples

```
d_fun <- as_d(dunif)
f_list <- list(a = d_fun, b = d_fun + 1, c = d_fun - 1)
summ_order(f_list)
summ_sort(f_list)
summ_rank(f_list)

# All methods might give different results on some elaborated pdqr-functions
# Methods "compare" and "mean" are not equivalent
non_mean_list <- list(
  new_d(data.frame(x = c(0.56, 0.815), y = c(1, 1)), "continuous"),
  new_d(data.frame(x = 0:1, y = c(0, 1)), "continuous")
)
summ_order(non_mean_list, method = "compare")
summ_order(non_mean_list, method = "mean")

# Methods powered by `summ_center()` are not equivalent
m <- c(0, 0.2, 0.1)
s <- c(1.1, 1.2, 1.3)
dlnorm_list <- lapply(seq_along(m), function(i) {
  as_d(dlnorm, meanlog = m[i], sdlog = s[i])
})
summ_order(dlnorm_list, method = "mean")
summ_order(dlnorm_list, method = "median")
summ_order(dlnorm_list, method = "mode")

# Method "compare" handles inherited non-transitivity. Here third element is
# "greater" than second (`P(f >= g) > 0.5`), second - than first, and first
# is "greater" than third.
non_trans_list <- list(
```

```
  new_d(data.frame(x = c(0.39, 0.44, 0.46), y = c(17, 14, 0)), "continuous"),
  new_d(data.frame(x = c(0.05, 0.3, 0.70), y = c(4, 0, 4)), "continuous"),
  new_d(data.frame(x = c(0.03, 0.40, 0.80), y = c(1, 1, 1)), "continuous")
)
summ_sort(non_trans_list)
  # Output doesn't depend on initial order
summ_sort(non_trans_list[c(2, 3, 1)])
```

`summ_prob_true` *Summarize boolean distribution with probability*

Description

Here `summ_prob_false()` returns a probability of 0 and `summ_prob_true()` - complementary probability (one minus `summ_prob_false()` output). Both of them check if their input is a **boolean** pdqr-function: type "discrete" with `x` in `x_tbl` identical to `c(0,1)`. If it is not, warning is thrown.

Usage

```
summ_prob_false(f)

summ_prob_true(f)
```

Arguments

`f`	A pdqr-function representing distribution.

Value

A single numeric value representing corresponding probability.

See Also

Other summary functions: `summ_center()`, `summ_classmetric()`, `summ_distance()`, `summ_entropy()`, `summ_hdr()`, `summ_interval()`, `summ_moment()`, `summ_order()`, `summ_pval()`, `summ_quantile()`, `summ_roc()`, `summ_separation()`, `summ_spread()`

Examples

```
d_unif <- as_d(dunif)
d_norm <- as_d(dnorm)
summ_prob_true(d_unif > d_norm)
summ_prob_false(2*d_norm > d_unif)

# When input is "continuous" function or doesn't have 0 as distribution
# element, probability of being false is returned as 0.
summ_prob_false(d_unif)
summ_prob_true(new_d(2, "discrete"))
```

summ_pval *Summarize distribution with p-value*

Description

summ_pval() computes p-value(s) based on supplied distribution and observed value(s). There are several methods of computing p-values ("both", "right", and "left") as well as several types of multiple comparison adjustments (using on stats::p.adjust()).

Usage

```
summ_pval(f, obs, method = "both", adjust = "holm")
```

Arguments

f	A pdqr-function representing distribution.
obs	Numeric vector of observed values to be used as threshold for p-value. Can have multiple values, in which case output will be adjusted for multiple comparisons with p.adjust().
method	Method representing direction of p-value computation. Should be one of "both", "right", "left".
adjust	Adjustment method as method argument to p.adjust().

Details

Method "both" for each element in obs computes two-sided p-value as min(1,2 * min(right_p_val,left_p_val)), where right_p_val and left_p_val are right and left one-sided p-values (ones which are computed with "right" and "left" methods) of obs's elements correspondingly.

Method "right" for each element x of obs computes probability of f >= x being true (more strictly, of random variable, represented by f, being not less than x). This corresponds to right one-sided p-value.

Method "left" for each element x of obs computes probability of f <= x, which is a left one-sided p-value.

Note that by default multiple p-values in output are adjusted with p.adjust(*, method = adjust). To not do any adjustment, use adjust = "none".

Value

A numeric vector with the same length as obs representing corresponding p-values after possible adjustment for multiple comparisons.

See Also

Other summary functions: summ_center(), summ_classmetric(), summ_distance(), summ_entropy(), summ_hdr(), summ_interval(), summ_moment(), summ_order(), summ_prob_true(), summ_quantile(), summ_roc(), summ_separation(), summ_spread()

Examples

```
# Type "discrete"
d_dis <- new_d(data.frame(x = 1:5, prob = c(1, 2, 3, 2, 1) / 9), "discrete")
summ_pval(d_dis, 3, method = "both")
summ_pval(d_dis, 3, method = "right")
summ_pval(d_dis, 3, method = "left")

# Type "continuous"
d_norm <- as_d(dnorm)
summ_pval(d_norm, 2, method = "both")
summ_pval(d_norm, 2, method = "right")
summ_pval(d_norm, 2, method = "left")

# Adjustment is made for multiple observed values
summ_pval(d_norm, seq(0, 2, by = 0.1))
  # Use `adjust = "none"` for to not do any adjustment
summ_pval(d_norm, seq(0, 2, by = 0.1), adjust = "none")
```

---

summ_quantile *Summarize distribution with quantiles*

Description

Essentially, this is a more strict wrapper of `as_q(f)(probs)`. If any value in `probs` is outside of segment \[0; 1\], an error is thrown.

Usage

```
summ_quantile(f, probs)
```

Arguments

`f`	A pdqr-function representing distribution.
`probs`	Vector of probabilities for which quantiles should be returned.

Value

A numeric vector of the same length as `probs` representing corresponding quantiles.

See Also

Other summary functions: `summ_center()`, `summ_classmetric()`, `summ_distance()`, `summ_entropy()`, `summ_hdr()`, `summ_interval()`, `summ_moment()`, `summ_order()`, `summ_prob_true()`, `summ_pval()`, `summ_roc()`, `summ_separation()`, `summ_spread()`

Examples

```
d_norm <- as_d(dnorm)
probs <- c(0.25, 0.5, 0.75)
all.equal(summ_quantile(d_norm, probs), as_q(d_norm)(probs))
```

`summ_roc` *Summarize distributions with ROC curve*

Description

These functions help you perform a ROC ("Receiver Operating Characteristic") analysis for one-dimensional linear classifier: values not more than some threshold are classified as "negative", and more than threshold - as "positive". Here input pair of pdqr-functions represent "true" distributions of values with "negative" (`f`) and "positive" (`g`) labels.

Usage

```
summ_roc(f, g, n_grid = 1001)

summ_rocauc(f, g, method = "expected")

roc_plot(roc, ..., add_bisector = TRUE)

roc_lines(roc, ...)
```

Arguments

`f`	A pdqr-function of any type and class. Represents "true" distribution of "negative" values.
`g`	A pdqr-function of any type and class. Represents "true" distribution of "positive" values.
`n_grid`	Number of points of ROC curve to be computed.
`method`	Method of computing ROC AUC. Should be one of "expected", "pessimistic", "optimistic" (see Details).
`roc`	A data frame representing ROC curve. Typically an output of `summ_roc()`.
`...`	Other arguments to be passed to `plot()` or `lines()`.
`add_bisector`	If `TRUE` (default), `roc_plot()` adds bisector line as reference for "random guess" classifier.

Details

ROC curve describes how well classifier performs under different thresholds. For all possible thresholds two classification metrics are computed which later form x and y coordinates of a curve:

- **False positive rate (FPR)**: proportion of "negative" distribution which was (incorrectly) classified as "positive". This is the same as one minus "specificity" (proportion of "negative" values classified as "negative").
- **True positive rate (TPR)**: proportion of "positive" distribution which was (correctly) classified as "positive". This is also called "sensitivity".

`summ_roc()` creates a uniform grid of decreasing `n_grid` values (so that output points of ROC curve are ordered from left to right) covering range of all meaningful thresholds. This range is computed as slightly extended range of `f` and `g` supports (extension is needed to achieve extreme values of "fpr" in presence of "discrete" type). Then FPR and TPR are computed for every threshold.

`summ_rocauc()` computes a common general (without any particular threshold in mind) diagnostic value of classifier, **area under ROC curve** ("ROC AUC" or "AUROC"). Numerically it is equal to a probability of random variable with distribution `g` *being strictly greater than* `f` plus *possible correction for functions being equal*, with multiple ways to account for it. Method "pessimistic" doesn't add correction, "expected" adds half of probability of `f` and `g` being equal (which is default), "optimistic" adds full probability. **Note** that this means that correction might be done only if both input pdqr-functions have "discrete" type. See pdqr methods for "Ops" group generic family for more details on comparing functions.

`roc_plot()` and `roc_lines()` perform plotting (with plot()) and adding (with lines()) ROC curves respectively.

Value

`summ_roc()` returns a data frame with `n_grid` rows and columns "threshold" (grid of classification thresholds, ordered decreasingly), "fpr", and "tpr" (corresponding false and true positive rates, ordered non-decreasingly by "fpr").

`summ_rocauc()` returns single number representing area under the ROC curve.

`roc_plot()` and `roc_lines()` create plotting side effects.

See Also

`summ_separation()` for computing optimal separation threshold.

Other summary functions: `summ_center()`, `summ_classmetric()`, `summ_distance()`, `summ_entropy()`, `summ_hdr()`, `summ_interval()`, `summ_moment()`, `summ_order()`, `summ_prob_true()`, `summ_pval()`, `summ_quantile()`, `summ_separation()`, `summ_spread()`

Examples

```
d_norm_1 <- as_d(dnorm)
d_norm_2 <- as_d(dnorm, mean = 1)
roc <- summ_roc(d_norm_1, d_norm_2)
head(roc)

# `summ_rocauc()` is equivalent to probability of `g > f`
```

```
summ_rocauc(d_norm_1, d_norm_2)
summ_prob_true(d_norm_2 > d_norm_1)

# Plotting
roc_plot(roc)
roc_lines(summ_roc(d_norm_2, d_norm_1), col = "blue")

# For "discrete" functions `summ_rocauc()` can produce different outputs
d_dis_1 <- new_d(1:2, "discrete")
d_dis_2 <- new_d(2:3, "discrete")
summ_rocauc(d_dis_1, d_dis_2)
summ_rocauc(d_dis_1, d_dis_2, method = "pessimistic")
summ_rocauc(d_dis_1, d_dis_2, method = "optimistic")
  # These methods correspond to different ways of plotting ROC curves
roc <- summ_roc(d_dis_1, d_dis_2)
  # Default line plot for "expected" method
roc_plot(roc, main = "Different type of plotting ROC curve")
  # Method "pessimistic"
roc_lines(roc, type = "s", col = "blue")
  # Method "optimistic"
roc_lines(roc, type = "S", col = "green")
```

`summ_separation` *Summarize distributions with separation threshold*

Description

Compute for pair of pdqr-functions the optimal threshold that separates distributions they represent. In other words, `summ_separation()` solves a binary classification problem with one-dimensional linear classifier: values not more than some threshold are classified as one class, and more than threshold - as another. Order of input functions doesn't matter.

Usage

```
summ_separation(f, g, method = "KS", n_grid = 10001)
```

Arguments

`f`	A pdqr-function of any type and class. Represents "true" distribution of "negative" values.
`g`	A pdqr-function of any type and class. Represents "true" distribution of "positive" values.
`method`	Separation method. Should be one of "KS" (Kolmogorov-Smirnov), "GM", "OP", "F1", "MCC" (all four are methods for computing classification metric in `summ_classmetric()`).
`n_grid`	Number of grid points to be used during optimization.

Details

All methods:

- Return middle point of nearest support edges in case of non-overlapping or "touching" supports of `f` and `g`.
- Return the smallest optimal solution in case of several candidates.

Method "KS" computes "x" value at which corresponding p-functions of `f` and `g` achieve supremum of their absolute difference (so input order of `f` and `g` doesn't matter). If input pdqr-functions have the same type, then result is a point of maximum absolute difference. If inputs have different types, then absolute difference of p-functions at the result point can be not the biggest. In that case output represents a left limit of points at which target supremum is reached (see Examples).

Methods "GM", "OP", "F1", "MCC" compute threshold which maximizes corresponding classification metric for best suited classification setup. They evaluate metrics at equidistant grid (with `n_grid` elements) for both directions (summ_classmetric(f, g, *) and summ_classmetric(g, f, *)) and return threshold which results into maximum of both setups. **Note** that other `summ_classmetric()` methods are either useless here (always return one of the edges) or are equivalent to ones already present.

Value

A single number representing optimal separation threshold.

See Also

`summ_roc()` for computing ROC curve related summaries.

`summ_classmetric()` for computing of classification metric for ordered classification setup.

Other summary functions: `summ_center()`, `summ_classmetric()`, `summ_distance()`, `summ_entropy()`, `summ_hdr()`, `summ_interval()`, `summ_moment()`, `summ_order()`, `summ_prob_true()`, `summ_pval()`, `summ_quantile()`, `summ_roc()`, `summ_spread()`

Examples

```
d_norm_1 <- as_d(dnorm)
d_unif <- as_d(dunif)
summ_separation(d_norm_1, d_unif, method = "KS")
summ_separation(d_norm_1, d_unif, method = "OP")

# Mixed types for "KS" method
p_dis <- new_p(1, "discrete")
p_unif <- as_p(punif)
thres <- summ_separation(p_dis, p_unif)
abs(p_dis(thres) - p_unif(thres))
  # Actual difference at `thres` is 0. However, supremum (equal to 1) as
  # limit value is # reached there.
x_grid <- seq(0, 1, by = 1e-3)
plot(x_grid, abs(p_dis(x_grid) - p_unif(x_grid)), type = "b")

# Handling of non-overlapping supports
```

```
summ_separation(new_d(2, "discrete"), new_d(3, "discrete"))

# The smallest "x" value is returned in case of several optimal thresholds
summ_separation(d_norm_1, d_norm_1) == meta_support(d_norm_1)[1]
```

`summ_spread` *Summarize distribution with spread*

Description

Functions to compute spread (variability, dispersion) of distribution (i.e. "how wide it is spread"). `summ_spread()` is a wrapper for respective summ_*() functions (from this page) with default arguments.

Usage

```
summ_spread(f, method = "sd")

summ_sd(f)

summ_var(f)

summ_iqr(f)

summ_mad(f)

summ_range(f)
```

Arguments

`f`	A pdqr-function representing distribution.
`method`	Method of spread computation. Should be one of "sd", "var", "iqr", "mad", "range".

Details

`summ_sd()` computes distribution's standard deviation.

`summ_var()` computes distribution's variance.

`summ_iqr()` computes distribution's interquartile range. Essentially, it is a `as_q(f)(0.75) - as_q(f)(0.25)`.

`summ_mad()` computes distribution's *median* absolute deviation around the distribution's *median*.

`summ_range()` computes range length (difference between maximum and minimum) of "x" values within region of positive probability. **Note** that this might differ from length of support because the latter might be affected by tails with zero probability (see Examples).

Value

All functions return a single number representing a spread of distribution.

See Also

`summ_center()` for computing distribution's center, `summ_moment()` for general moments.

Other summary functions: `summ_center()`, `summ_classmetric()`, `summ_distance()`, `summ_entropy()`, `summ_hdr()`, `summ_interval()`, `summ_moment()`, `summ_order()`, `summ_prob_true()`, `summ_pval()`, `summ_quantile()`, `summ_roc()`, `summ_separation()`

Examples

```
# Type "continuous"
d_norm <- as_d(dnorm)
  # The same as `summ_spread(d_norm, method = "sd")`
summ_sd(d_norm)
summ_var(d_norm)
summ_iqr(d_norm)
summ_mad(d_norm)
summ_range(d_norm)

# Type "discrete"
d_pois <- as_d(dpois, lambda = 10)
summ_sd(d_pois)
summ_var(d_pois)
summ_iqr(d_pois)
summ_mad(d_pois)
summ_range(d_pois)

# Difference of `summ_range(f)` and `diff(meta_support(f))`
zero_tails <- new_d(data.frame(x = 1:5, y = c(0, 0, 1, 0, 0)), "continuous")
  # This returns difference between 5 and 1
diff(meta_support(zero_tails))
  # This returns difference between 2 and 4 as there are zero-probability
  # tails
summ_range(zero_tails)
```

Index