Package: precrec (via r-universe)

October 14, 2026

Type Package

Title Calculate Accurate Precision-Recall and ROC (Receiver Operator Characteristics) Curves

Version 0.24.0

Description Accurate calculations and visualization of precision-recall and ROC (Receiver Operator Characteristics) curves. Saito and Rehmsmeier (2015) <doi:10.1371/journal.pone.0118432>.

URL https://github.com/evalclass/precrec, https://evalclass.github.io/precrec/

BugReports https://github.com/evalclass/precrec/issues

Depends R (>= 4.1)

License GPL-3

Language en-US

LazyData TRUE

Suggests testthat (>= 3.0.0), knitr (>= 1.11), rmarkdown (>= 2.0), patchwork (>= 1.1.2), spelling (>= 2.2)

LinkingTo Rcpp

Imports Rcpp (>= 1.0.0), ggplot2 (>= 3.0.0), checkmate (>= 2.0.0), cli (>= 3.4.0), grid, gridExtra (>= 2.0.0), methods, data.table (>= 1.10.4), withr (>= 2.3.0), graphics (>= 4.0.0), rlang (>= 1.0.0), stats, utils

Encoding UTF-8

VignetteBuilder knitr

Config/testthat/edition 3

Config/testthat/parallel true

Config/roxygen2/version 8.0.0

Roxygen list(markdown = TRUE)

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

Date/Publication 2026-09-14 19:01:51 UTC

RemoteUrl https://github.com/evalclass/precrec

RemoteRef HEAD

RemoteSha bbe270215ffd66ca9f0e53467c02ce946736cecc

Contents

as.data.frame . . . 3
as.data.table . . . 8
auc . . . 10
auc_boot . . . 14
auc_ci . . . 15
auc_delong . . . 17
auc_diff . . . 20
autoplot . . . 23
average_precision . . . 30
B1000 . . . 31
B500 . . . 32
best_cutoff . . . 33
C3N150 . . . 36
classification_report . . . 36
create_sim_samples . . . 38
evalmod . . . 40
format_nfold . . . 48
format_points . . . 49
fortify . . . 52
IB1000 . . . 58
IB500 . . . 58
join_labels . . . 59
join_scores . . . 60
M2N50F5 . . . 62
metric_curve . . . 62
metric_table . . . 65
mmdata . . . 68
P10N10 . . . 72
part . . . 73
pauc . . . 76
plot . . . 80
prbe . . . 85
precrec . . . 87
print . . . 89
prob_metrics . . . 91
prob_metrics_ci . . . 93

Index **96**

`as.data.frame` *Convert a curves and points object to a data frame*

Description

The `as.data.frame` function converts an `S3` object generated by `evalmod()` to a data frame.

Usage

```
## S3 method for class 'sscurves'
as.data.frame(x, row.names = NULL, optional = FALSE, raw_curves = NULL, ...)

## S3 method for class 'mscurves'
as.data.frame(x, row.names = NULL, optional = FALSE, raw_curves = NULL, ...)

## S3 method for class 'smcurves'
as.data.frame(x, row.names = NULL, optional = FALSE, raw_curves = NULL, ...)

## S3 method for class 'mmcurves'
as.data.frame(x, row.names = NULL, optional = FALSE, raw_curves = NULL, ...)

## S3 method for class 'sspoints'
as.data.frame(x, row.names = NULL, optional = FALSE, raw_curves = NULL, ...)

## S3 method for class 'mspoints'
as.data.frame(x, row.names = NULL, optional = FALSE, raw_curves = NULL, ...)

## S3 method for class 'smpoints'
as.data.frame(x, row.names = NULL, optional = FALSE, raw_curves = NULL, ...)

## S3 method for class 'mmpoints'
as.data.frame(x, row.names = NULL, optional = FALSE, raw_curves = NULL, ...)

## S3 method for class 'aucroc'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

## S3 method for class 'ssxycurves'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

## S3 method for class 'msxycurves'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

## S3 method for class 'smxycurves'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

## S3 method for class 'mmxycurves'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

`x` An `S3` object generated by `evalmod()`. The `as.data.frame` function takes one of the following `S3` objects.

1. ROC and Precision-Recall curves (mode = "rocprc")

`S3` object	# of models	# of test datasets
sscurves	single	single
mscurves	multiple	single
smcurves	single	multiple
mmcurves	multiple	multiple

2. Basic evaluation metrics (mode = "basic")

`S3` object	# of models	# of test datasets
sspoints	single	single
mspoints	multiple	single
smpoints	single	multiple
mmpoints	multiple	multiple

3. Fast AUC (ROC) calculation with the U statistic (mode = "aucroc")

`S3` object	# of models	# of test datasets
aucroc	-	-

See the **Value** section of `evalmod()` for more details.

`row.names` Not used by this method.

`optional` Not used by this method.

`raw_curves` A Boolean value to specify whether raw curves are shown instead of the average curve. It is effective only when `raw_curves` is set to `TRUE` of the `evalmod()` function.

`...` Not used by this method.

Value

The `as.data.frame` function returns a data frame.

See Also

`evalmod()` for generating `S3` objects with performance evaluation metrics.

Examples

```
## Not run:
##################################################
### Single model & single test dataset
###

## Load a dataset with 10 positives and 10 negatives
data(P10N10)

## Generate an sscurve object that contains ROC and Precision-Recall curves
sscurves <- evalmod(scores = P10N10$scores, labels = P10N10$labels)

## Convert sscurves to a data frame
sscurves.df <- as.data.frame(sscurves)

## Show data frame
head(sscurves.df)

## Generate an sspoints object that contains basic evaluation metrics
sspoints <- evalmod(
  mode = "basic", scores = P10N10$scores,
  labels = P10N10$labels
)
## Convert sspoints to a data frame
sspoints.df <- as.data.frame(sspoints)

## Show data frame
head(sspoints.df)


##################################################
### Multiple models & single test dataset
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(1, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mscurves <- evalmod(mdat)

## Convert mscurves to a data frame
mscurves.df <- as.data.frame(mscurves)

## Show data frame
head(mscurves.df)

## Generate an mspoints object that contains basic evaluation metrics
mspoints <- evalmod(mdat, mode = "basic")
```

```
## Convert mspoints to a data frame
mspoints.df <- as.data.frame(mspoints)

## Show data frame
head(mspoints.df)


##################################################
### Single model & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(10, 100, 100, "good_er")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an smcurve object that contains ROC and Precision-Recall curves
smcurves <- evalmod(mdat, raw_curves = TRUE)

## Convert smcurves to a data frame
smcurves.df <- as.data.frame(smcurves)

## Show data frame
head(smcurves.df)

## Generate an smpoints object that contains basic evaluation metrics
smpoints <- evalmod(mdat, mode = "basic")

## Convert smpoints to a data frame
smpoints.df <- as.data.frame(smpoints)

## Show data frame
head(smpoints.df)


##################################################
### Multiple models & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(10, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mmcurves <- evalmod(mdat, raw_curves = TRUE)

## Convert mmcurves to a data frame
mmcurves.df <- as.data.frame(mmcurves)
```

```
## Show data frame
head(mmcurves.df)

## Generate an mmpoints object that contains basic evaluation metrics
mmpoints <- evalmod(mdat, mode = "basic")

## Convert mmpoints to a data frame
mmpoints.df <- as.data.frame(mmpoints)

## Show data frame
head(mmpoints.df)


##################################################
### N-fold cross validation datasets
###

## Load test data
data(M2N50F5)

## Speficy nessesary columns to create mdat
cvdat <- mmdata(
  nfold_df = M2N50F5, score_cols = c(1, 2),
  lab_col = 3, fold_col = 4,
  modnames = c("m1", "m2"), dsids = 1:5
)

## Generate an mmcurve object that contains ROC and Precision-Recall curves
cvcurves <- evalmod(cvdat)

## Convert mmcurves to a data frame
cvcurves.df <- as.data.frame(cvcurves)

## Show data frame
head(cvcurves.df)

## Generate an mmpoints object that contains basic evaluation metrics
cvpoints <- evalmod(cvdat, mode = "basic")

## Convert mmpoints to a data frame
cvpoints.df <- as.data.frame(cvpoints)

## Show data frame
head(cvpoints.df)


##################################################
### AUC with the U statistic
###

## mode = "aucroc"
data(P10N10)
```

```
uauc1 <- evalmod(
  scores = P10N10$scores, labels = P10N10$labels,
  mode = "aucroc"
)

# as.data.frame 'aucroc'
as.data.frame(uauc1)

## mode = "aucroc"
samps <- create_sim_samples(10, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)
uauc2 <- evalmod(mdat, mode = "aucroc")

# as.data.frame 'aucroc'
head(as.data.frame(uauc2))

## End(Not run)
```

`as.data.table` *Convert a curves and points object to a data.table*

Description

The `as.data.table` function converts an `S3` object generated by `evalmod()` to a `data.table`. It is the counterpart of `as.data.frame()`, for callers who would rather have the `data.table` that `precrec` builds internally than a copy of it as a plain data frame.

Usage

```
## S3 method for class 'sscurves'
as.data.table(x, keep.rownames = FALSE, raw_curves = NULL, ...)

## S3 method for class 'mscurves'
as.data.table(x, keep.rownames = FALSE, raw_curves = NULL, ...)

## S3 method for class 'smcurves'
as.data.table(x, keep.rownames = FALSE, raw_curves = NULL, ...)

## S3 method for class 'mmcurves'
as.data.table(x, keep.rownames = FALSE, raw_curves = NULL, ...)

## S3 method for class 'sspoints'
as.data.table(x, keep.rownames = FALSE, raw_curves = NULL, ...)

## S3 method for class 'mspoints'
```

```
as.data.table(x, keep.rownames = FALSE, raw_curves = NULL, ...)

## S3 method for class 'smpoints'
as.data.table(x, keep.rownames = FALSE, raw_curves = NULL, ...)

## S3 method for class 'mmpoints'
as.data.table(x, keep.rownames = FALSE, raw_curves = NULL, ...)

## S3 method for class 'aucroc'
as.data.table(x, keep.rownames = FALSE, ...)
```

Arguments

x	An S3 object generated by evalmod(). The as.data.table function accepts the same objects as as.data.frame().
keep.rownames	Ignored. Present because data.table::as.data.table has it in its signature; the objects this method converts carry no row names.
raw_curves	A Boolean value to specify whether raw curves are shown instead of the average curve. It is effective only when raw_curves is set to TRUE of the evalmod() function.
...	Not used by this method.

Value

The as.data.table function returns a data.table with the same columns as as.data.frame() returns for the same object.

See Also

evalmod() for generating S3 objects with performance evaluation metrics. as.data.frame() for the plain data frame equivalent. fortify() for converting the same objects for ggplot2.

Examples

```
## Load a dataset with 10 positives and 10 negatives
data(P10N10)

## Generate an sscurve object that contains ROC and Precision-Recall curves
sscurves <- evalmod(scores = P10N10$scores, labels = P10N10$labels)

## Convert sscurves to a data.table
if (requireNamespace("data.table", quietly = TRUE)) {
  head(data.table::as.data.table(sscurves))
}
```

auc *Retrieve a data frame of AUC scores*

Description

The auc function takes an S3 object generated by evalmod() and retrieves a data frame with the Area Under the Curve (AUC) scores of ROC and Precision-Recall curves.

Usage

```
auc(curves, macro = TRUE, macro_weight = c("uniform", "prevalence"))

## S3 method for class 'aucs'
auc(curves, macro = TRUE, macro_weight = c("uniform", "prevalence"))
```

Arguments

curves

An S3 object generated by evalmod(). The auc function accepts the following S3 objects.

S3 object	# of models	# of test datasets
sscurves	single	single
mscurves	multiple	single
smcurves	single	multiple
mmcurves	multiple	multiple

See the **Value** section of evalmod() for more details.

macro

A Boolean value to specify whether the macro-average of the per-class AUCs is added. It is effective only for a multiclass evaluation - see the multiclass argument of mmdata() - and the added rows carry macro-average as their model name. Classes that could not be evaluated are left out of the average.

macro_weight

How the per-class AUCs are weighted in that average. "uniform", the default, gives every class the same weight, so a rare class counts as much as a common one. "prevalence" weights each class by the number of observations it has, so the average follows the class distribution of the data. The weighted rows are named macro-average-weighted to keep the two apart. The two agree on a balanced dataset.

For a ROC evaluation the two are the metrics other packages call roc_aunu and roc_aunp respectively.

Value

The auc function returns a data frame with one row per curve per model per test dataset, and the following columns.

`modnames`	Model name
`dsids`	Test dataset ID
`curvetypes`	`ROC` or `PRC`
`aucs`	The area under that curve
`baselines`	What that area would be by chance, see below

Reading an area against its baseline

A ROC curve's chance level is `0.5` whatever the data, so a ROC AUC can be read on its own. A precision-recall curve's chance level is the proportion of positives, so a PRC AUC cannot: the same number means different things on different data, and the difference is not small.

The `baselines` column carries that value, `0.5` on every `ROC` row and the prevalence on every `PRC` row, so that the area and what it is worth arrive together. On the same generator at three class balances:

Positives	ROC AUC	PRC AUC	PRC baseline
50%	0.807	0.801	0.50
10%	0.834	0.367	0.10
2%	0.850	0.129	0.02

The classifier is about as good in all three rows and the ROC AUC says so. The PRC AUC falls to `0.129`, which reads as failure and is in fact six times chance. Quote the two numbers together.

The baseline is looked up per model and per test dataset, because a fold need not hold the classes in the proportions the whole dataset does. On a `macro-average` row it is averaged over the classes exactly as the AUCs are, with the same weights, so the row is still read against the chance level of the mixture that produced it.

The baseline is an asymptote

The prevalence is what a precision-recall area is worth by chance in the limit. An area measured on a finite sample scatters around its chance level rather than sitting on it, and the fewer the positives the wider it scatters and the further above it the average sits. Dividing the two numbers is therefore not a test.

With twenty positives at two percent prevalence, a classifier with no signal at all averages `1.15` times its baseline over the whole curve and `1.91` times over the first tenth of recall, where it clears twice the baseline `22%` of the time. Two hundred positives brings the first figure to `1.01`. What drives it is the number of positives, not the balance, and `average_precision()` shows the same thing, so it is not an artifact of the interpolation.

The example above stands: `0.129` against a baseline of `0.02` is a real result. But when the margin is small and the positives are few, `auc_boot()` puts an interval around the area from a single test set, and `auc_ci()` does it from several. Both now report the baseline beside the interval, which is the comparison worth making.

See Also

evalmod() for generating S3 objects with performance evaluation metrics. pauc() for retrieving a dataset of pAUCs. average_precision() for the step estimator of the area under the precision-recall curve. auc_boot() and auc_ci() for an interval around an area, which is what the gap between an area and its baseline has to be read against.

Examples

```
##################################################
### Single model & single test dataset
###

## Load a dataset with 10 positives and 10 negatives
data(P10N10)

## Generate an sscurve object that contains ROC and Precision-Recall curves
sscurves <- evalmod(scores = P10N10$scores, labels = P10N10$labels)

## Shows AUCs
auc(sscurves)


##################################################
### Multiple models & single test dataset
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(1, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mscurves <- evalmod(mdat)

## Shows AUCs
auc(mscurves)


##################################################
### Single model & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(4, 100, 100, "good_er")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an smcurve object that contains ROC and Precision-Recall curves
smcurves <- evalmod(mdat, raw_curves = TRUE)
```

```
## Get AUCs
sm_aucs <- auc(smcurves)

## Shows AUCs
sm_aucs

## Get AUCs of Precision-Recall
sm_aucs_prc <- subset(sm_aucs, curvetypes == "PRC")

## Shows AUCs
sm_aucs_prc

##################################################
### Multiple models & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(4, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mmcurves <- evalmod(mdat, raw_curves = TRUE)

## Get AUCs
mm_aucs <- auc(mmcurves)

## Shows AUCs
mm_aucs

## Get AUCs of Precision-Recall
mm_aucs_prc <- subset(mm_aucs, curvetypes == "PRC")

## Shows AUCs
mm_aucs_prc


##################################################
### Multiclass evaluation
###

## Load a 3-class dataset with one score column per class
data(C3N150)

## One-vs-rest curves
mccurves <- evalmod(scores = C3N150$scores, labels = C3N150$labels)

## Per-class AUCs, plus their macro-average
auc(mccurves)
```

```
## Per-class AUCs only
auc(mccurves, macro = FALSE)

## Weighted by the class distribution instead
auc(mccurves, macro_weight = "prevalence")
```

---

auc_boot *Bootstrap AUCs from one test set*

Description

auc_boot resamples a single test set and recalculates the areas under the ROC and precision-recall curves on each resample. The result carries the whole bootstrap distribution, which auc_ci() turns into confidence intervals and auc_diff() into comparisons between models.

Usage

```
auc_boot(mdat, scores = NULL, labels = NULL, boot_n = 1000, seed = NULL, ...)
```

Arguments

mdat	An mdata object created by mmdata() holding exactly one dataset. It can be omitted when scores and labels are given.
scores	A numeric vector, matrix, array, data frame, or list of scores. See mmdata() for the accepted shapes.
labels	A numeric, character, logical, or factor vector of observed labels, or a list of such vectors.
boot_n	The number of bootstrap resamples. The default of 1000 is the usual compromise; the tails of a percentile interval are estimated from alpha / 2 of these, so 25 resamples at each end of a 95% interval, and 2000 or more is worth the wait when the interval itself is the result being reported.
seed	A seed for the resampling, so that a reported interval can be reproduced. The global random number state is restored afterwards, so a seeded call does not disturb the stream the caller is drawing from. NULL, the default, resamples from the current state.
...	Further arguments passed to mmdata() when scores and labels are given instead of mdat.

Details

The rest of the package builds its intervals from the variation between several test sets, so auc_ci() on an evalmod result needs several and says so. This is the answer for the ordinary case of one test set, where the only sample there is has to stand in for the population.

Value

An object of class `aucboot`: a data frame of one row per model, curve type and resample, with the columns `modnames`, `curvetypes`, `boot_id`, `aucs` and `baselines`. The AUCs of the original data are attached as the `observed` attribute, with a `baselines` column of their own.

`baselines` is the same on every resample, because the resampling is stratified and so holds the class balance fixed - see below. It is carried on the resamples anyway so that an area and what it is worth by chance never have to be joined back together by hand.

Resampling

The resampling is **stratified**: positives are drawn from the positives and negatives from the negatives, so every resample holds exactly the class balance the original does. An unstratified bootstrap of an imbalanced dataset produces resamples with a different balance, and sometimes with no positives at all, which moves the precision-recall baseline underneath the quantity being estimated.

Every model is resampled on the **same** draws. Comparisons between models are then paired, which is what makes `auc_diff()` a statement about the difference rather than about two independent intervals.

See Also

`auc_ci()` for the intervals, `auc_diff()` for comparing two models, `auc_delong()` for an exact standard error of the ROC AUC, and `auc()` for the point estimates.

Examples

```
## One test set, two models
samps <- create_sim_samples(1, 100, 100, c("poor_er", "good_er"))
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]]
)

booted <- auc_boot(mdat, boot_n = 200, seed = 42)
auc_ci(booted)
auc_diff(booted)
```

---

auc_ci — *Calculate CIs of ROC and precision-recall AUCs*

Description

The `auc_ci` function takes an `S3` object generated by `evalmod()` and calculates CIs of AUCs when multiple data sets are specified.

Usage

```
auc_ci(curves, alpha = NULL, dtype = NULL)

## S3 method for class 'aucs'
auc_ci(curves, alpha = 0.05, dtype = "normal")

## S3 method for class 'aucboot'
auc_ci(curves, alpha = 0.05, dtype = NULL)

## S3 method for class 'aucdelong'
auc_ci(curves, alpha = 0.05, dtype = NULL)
```

Arguments

`curves` An `S3` object generated by `evalmod()`. The `auc_ci` function accepts the following S3 objects.

S3 object	# of models	# of test datasets
smcurves	single	multiple
mmcurves	multiple	multiple

See the **Value** section of `evalmod()` for more details.

It also accepts the two objects that describe the uncertainty of a single test set: an `aucboot` object from `auc_boot()`, which gives a percentile interval, and an `aucdelong` object from `auc_delong()`, which gives a normal interval around DeLong's analytic standard error.

`alpha` A numeric value of the significant level (default: 0.05)

`dtype` A string to specify the distribution used for CI calculation.

dtype	distribution
normal (default)	Normal distribution
z	Normal distribution
t	t-distribution

Value

The `auc_ci` function returns a dataframe of AUC CIs, with a `baselines` column beside the area giving what that area would be by chance - `0.5` for a ROC curve and the proportion of positives for a precision-recall curve. See `Reading an area against its baseline` in `auc()`.

Over several test datasets the baseline is averaged over the same datasets the mean area is, so a fold that could not be evaluated is left out of both. From `auc_boot()` or `auc_delong()` it is the balance of the single test set.

An interval is the point of this function: the prevalence is what an area is worth by chance in the limit, and an area from a finite sample scatters around it, so an area sitting above its baseline means little on its own. Read the baseline against the interval instead - see `The baseline is an asymptote` in `auc()`.

See Also

evalmod() for generating S3 objects with performance evaluation metrics. auc() for retrieving a dataset of AUCs. auc_boot() and auc_delong() for a single test set.

Examples

```
##################################################
### Single model & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(4, 100, 100, "good_er")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an smcurve object that contains ROC and Precision-Recall curves
smcurves <- evalmod(mdat)

## Calculate CI of AUCs
sm_auc_cis <- auc_ci(smcurves)

## Shows the result
sm_auc_cis

##################################################
### Multiple models & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(4, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mmcurves <- evalmod(mdat)

## Calculate CI of AUCs
mm_auc_ci <- auc_ci(mmcurves)

## Shows the result
mm_auc_ci
```

---

auc_delong *DeLong's standard error for the ROC AUC*

Description

auc_delong calculates the ROC AUC of every model on one test set together with the variance and covariance of those AUCs, exactly rather than by resampling. auc_ci() turns the result into confidence intervals and auc_diff() into comparisons between models.

Usage

```
auc_delong(mdat, scores = NULL, labels = NULL, ...)
```

Arguments

mdat	An mdata object created by mmdata() holding exactly one dataset. It can be omitted when scores and labels are given.
scores	A numeric vector, matrix, array, data frame, or list of scores. See mmdata() for the accepted shapes.
labels	A numeric, character, logical, or factor vector of observed labels, or a list of such vectors.
...	Further arguments passed to mmdata() when scores and labels are given instead of mdat.

Details

The ROC AUC is a Mann-Whitney U statistic, so its variance follows from the structural components of that statistic and needs no bootstrap. That makes this the exact counterpart of auc_boot(): same input, same two functions reading the result, no boot_n, no seed, nothing that changes between two runs, and no floor under the p-value.

Value

An object of class aucdelong: a data frame of one row per model, with the columns modnames, curvetypes, aucs and error, the last being the standard error of the AUC. The covariance matrix of the AUCs is attached as the cov attribute, and is what makes the comparison in auc_diff() a paired one.

The ROC AUC only

There is no precision-recall counterpart of this. The ROC AUC is a U statistic - the probability that a random positive outranks a random negative - and everything below rests on that. The precision-recall AUC is not one, and the interpolated area auc() reports is further from being one still, so it has no analytic variance to report and does not appear in the result.

auc_boot() remains the answer for the precision-recall AUC, and the one to reach for whenever the precision-recall curve is the point. This function is here because the ROC AUC is what a great many readers and reviewers ask about by name, and because it is exact where the bootstrap is approximate.

How the variance is calculated

Write `m` for the positives and `n` for the negatives. Each positive `i` and each negative `j` contributes a structural component - the share of the other class it beats:

```
V10[i] = (1 / n) * sum over j of psi(x[i], y[j])
V01[j] = (1 / m) * sum over i of psi(x[i], y[j])
```

where `psi` is 1 when the positive outranks the negative, 0.5 when they are tied and 0 otherwise - the same half credit for a tie that the AUC itself gives. The variance is then

```
var(auc) = var(V10) / m + var(V01) / n
```

and for two models on the same test set the covariance is the same expression with the covariance of their components. Both are computed from midranks rather than from all `m * n` comparisons, so the cost is a sort rather than a product.

The components are taken from the ranks `precrec` already assigns, so ties and `NA` scores are handled exactly as they are everywhere else in the package and the AUC reported here is the one `auc()` reports.

What it assumes

The variance is an **asymptotic** one. It describes how the AUC would move over repeated samples of the same size, and the interval built from it is a normal one, so both get better as the test set grows. On a small or badly imbalanced test set the percentile interval of `auc_boot()` is the safer reading, and comparing the two is a cheap way to find out whether the sample is large enough for this one.

See Also

`auc_boot()` for the resampling counterpart, which also covers the precision-recall AUC, `auc_ci()` for the intervals and `auc_diff()` for comparing models.

Examples

```
## One test set, two models
samps <- create_sim_samples(1, 100, 100, c("poor_er", "good_er"))
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]]
)

delong <- auc_delong(mdat)
delong

auc_ci(delong)
auc_diff(delong)
```

auc_diff *Compare AUCs between models*

Description

auc_diff reports, for every pair of models, the difference between their AUCs with a confidence interval around it. It reads either of the two estimates of uncertainty the package offers for a single test set: the resamples of auc_boot(), which cover both curve types, or the analytic covariance of auc_delong(), which covers the ROC AUC exactly.

Usage

```
auc_diff(x, alpha = NULL, alternative = NULL)

## S3 method for class 'aucboot'
auc_diff(x, alpha = 0.05, alternative = "two.sided")

## S3 method for class 'aucdelong'
auc_diff(x, alpha = 0.05, alternative = "two.sided")
```

Arguments

x	An aucboot object from auc_boot() or an aucdelong object from auc_delong().
alpha	The interval covers 1 - alpha of the resampled differences, so the default of 0.05 gives a 95% interval.
alternative	The side of zero the alternative hypothesis is on, one of "two.sided", "greater" and "less". "greater" tests whether the first model of the pair has the larger AUC. It selects the tail of every p-value in the result and leaves z_values alone; the interval stays two-sided whatever it is set to.

Details

The comparison is paired either way. With a bootstrap both models are scored on the same resample, so the difference is calculated within a resample and the spread of those differences is what the interval describes; with DeLong the pairing is carried by the covariance between the two AUCs. Two separate intervals from auc_ci() cannot be read this way - they can overlap while the difference is still clearly on one side of zero, because they say nothing about how the two models move together.

Value

A data frame with one row per curve type and pair of models.

From an aucboot object:

curvetypes	ROC or PRC
modnames1, modnames2	The pair, in the order compared

`diffs`	Observed AUC of the first minus that of the second
`lower_bound`, `upper_bound`	Percentile interval of the resampled differences
`p_values`	Percentile p-value, see below
`z_values`	Wald statistic, `diffs / sd(d)`, see below
`p_values_wald`	Wald p-value, see below
`n`	Resamples the row is based on

From an `aucdelong` object, where there is one p-value rather than two and only the ROC AUC to report:

`curvetypes`	ROC
`modnames1`, `modnames2`	The pair, in the order compared
`diffs`	AUC of the first minus that of the second
`lower_bound`, `upper_bound`	Normal interval of the difference, clipped to `[-1, 1]`
`z_values`	`diffs` over its standard error
`p_values`	DeLong's p-value, see below
`n`	Instances in the test set

DeLong's comparison

Given an `aucdelong` object the standard error of the difference comes out of the covariance matrix,

```
se = sqrt(var(auc1) + var(auc2) - 2 * cov(auc1, auc2))
z_values = diffs / se
p_values = 2 * pnorm(-abs(z_values))
```

and the covariance is what makes it a paired test: two models scored on the same test set rise and fall together, and dropping the last term would overstate how uncertain their difference is.

This is a Wald test too, so `alternative` selects its tail the same way. What it does not need is `boot_n`: the standard error is exact rather than resampled, so the p-value has no floor and does not move between two runs. What it does assume is that the variance is asymptotic - see `auc_delong()` - and it has nothing to say about the precision-recall AUC.

`z_values` is `NA` when two models rank every instance the same way and there is no standard error to divide by, and `p_values` is `NA` with it.

The interval and the two p-values

This section and the two that follow describe an `aucboot` object.

The interval is the primary result, and is the `alpha / 2` and `1 - alpha / 2` quantiles of the resampled differences.

The two p-values are the two ways of getting one out of a bootstrap, and each column is named for the method that produced it. Write `d` for the vector of `n` resampled differences and `diffs` for the observed one.

`p_values` is the **percentile** p-value - the share of `d` on the other side of zero from `diffs`, doubled for a two-sided test:

```
p_values = 2 * min(1 + sum(d <= 0), 1 + sum(d >= 0)) / (n + 1)
```

The added ones keep it away from exactly zero. It cannot go below `2 / (n + 1)`, so `boot_n` sets a floor under it.

`z_values` and `p_values_wald` are the **Wald** test - an estimate over an estimate of its standard error, referred to a standard normal:

```
z_values      = diffs / sd(d)
p_values_wald = 2 * pnorm(-abs(z_values))
```

`sd(d)` is the standard error, because the spread of the bootstrap distribution is what the bootstrap has to say about how far the difference moves from sample to sample. Having a scale underneath it rather than a count of resamples, the Wald p-value has no floor.

`alternative` replaces the doubled minimum with the matching one-sided count, and `-abs()` with `-` or `+`, in the two formulas above.

Why two p-values

Because they fail in opposite ways, and neither one on its own tells you that it is failing.

The percentile p-value assumes nothing about the shape of `d`, and pays for that with the floor. Once it reaches `2 / (n + 1)` it has stopped measuring the models and started reporting `boot_n`: a difference that is merely clear and one that is overwhelming both come out at 0.002 at the default thousand resamples, and the number gives no sign of which it is looking at.

The Wald p-value has no floor, and pays for that with an assumption. It takes `d` to be roughly normal, and locates the null by reflecting the sampling distribution rather than by enforcing it. Where `d` is skewed - few positives, or either model near the ceiling of the precision-recall AUC - it is confidently wrong, and again the number carries no warning.

Side by side they cover each other. When they agree, the normality the Wald test assumes is doing no harm at this sample size and you can quote its resolution. When they disagree sharply, `d` is not the shape the Wald test needs, and the percentile p-value - floor and all - is the one to trust. The comparison is the diagnostic; neither column is one on its own.

There is a second, exact sense in which they are different answers. A Wald test is the counterpart of the bootstrap **normal** interval, `diffs` plus and minus `z` standard errors, while `lower_bound` and `upper_bound` are the **percentile** interval. So `p_values_wald` is not the dual of the bounds reported beside it, and need not agree with them.

Read the interval first. Both p-values are companions to it rather than exact tests.

Not a t statistic

`z_values` divides by a standard error, not by a standard error of a mean, and is read off the normal rather than off a t. `auc_ci(dtype = "t")` is the genuine t in this package, and the contrast is exact: there the spread is taken over a handful of real test sets and divided by the square root of how many there were, so `n - 1` degrees of freedom mean something. Here the spread is taken over resamples and divided by nothing - `boot_n` is a setting rather than a sample size, and a t on `boot_n - 1` degrees of freedom would be a p-value that shrinks when the caller resamples harder.

Nor is this the bootstrap-*t*, which forms a statistic of its own inside every resample and takes its reference distribution from those rather than from the normal.

z_values is NA when the resamples have no spread to divide by, which happens when two models are given the same scores, and p_values_wald is NA with it.

See Also

auc_boot() for the resampling, auc_delong() for the analytic alternative, and auc_ci() for one model at a time.

Examples

```
samps <- create_sim_samples(1, 100, 100, c("poor_er", "good_er"))
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]]
)

booted <- auc_boot(mdat, boot_n = 200, seed = 42)
auc_diff(booted)

## Is the second model the better one?
auc_diff(booted, alternative = "less")

## The same comparison without resampling, for the ROC AUC
auc_diff(auc_delong(mdat))
```

---

autoplot *Plot performance evaluation metrics with ggplot2*

Description

The autoplot function plots performance evaluation metrics by using **ggplot2** instead of the general R plot.

Usage

```
## S3 method for class 'sscurves'
autoplot(object, curvetype = c("ROC", "PRC"), ...)

## S3 method for class 'mscurves'
autoplot(object, curvetype = c("ROC", "PRC"), ...)

## S3 method for class 'smcurves'
autoplot(object, curvetype = c("ROC", "PRC"), ...)

## S3 method for class 'mmcurves'
autoplot(object, curvetype = c("ROC", "PRC"), ...)

## S3 method for class 'sspoints'
```

```
autoplot(object, curvetype = .get_obj_metrics(object), ...)

## S3 method for class 'mspoints'
autoplot(object, curvetype = .get_obj_metrics(object), ...)

## S3 method for class 'smpoints'
autoplot(object, curvetype = .get_obj_metrics(object), ...)

## S3 method for class 'mmpoints'
autoplot(object, curvetype = .get_obj_metrics(object), ...)

## S3 method for class 'ssxycurves'
autoplot(object, ...)

## S3 method for class 'msxycurves'
autoplot(object, ...)

## S3 method for class 'smxycurves'
autoplot(object, ...)

## S3 method for class 'mmxycurves'
autoplot(object, ...)
```

Arguments

`object` An `S3` object generated by `evalmod()`. The `autoplot` function accepts the following `S3` objects for two different modes, "rocprc" and "basic".

1. ROC and Precision-Recall curves (`mode = "rocprc"`)

`S3` **object**	**# of models**	**# of test datasets**
sscurves	single	single
mscurves	multiple	single
smcurves	single	multiple
mmcurves	multiple	multiple

2. Basic evaluation metrics (`mode = "basic"`)

`S3` **object**	**# of models**	**# of test datasets**
sspoints	single	single
mspoints	multiple	single
smpoints	single	multiple
mmpoints	multiple	multiple

See the **Value** section of `evalmod()` for more details.

`curvetype` A character vector with the following curve types.

1. ROC and Precision-Recall curves (mode = "rocprc")

curvetype	description
ROC	ROC curve
PRC	Precision-Recall curve

Multiple `curvetype` can be combined, such as `c("ROC", "PRC")`.

2. Basic evaluation metrics (mode = "basic")

curvetype	description
error	Normalized ranks vs. error rate
accuracy	Normalized ranks vs. accuracy
specificity	Normalized ranks vs. specificity
sensitivity	Normalized ranks vs. sensitivity
precision	Normalized ranks vs. precision
mcc	Normalized ranks vs. Matthews correlation coefficient
fscore	Normalized ranks vs. F-score
balanced_accuracy	Normalized ranks vs. balanced accuracy
npv	Normalized ranks vs. negative predictive value
informedness	Normalized ranks vs. informedness (Youden's J)
markedness	Normalized ranks vs. markedness
kappa	Normalized ranks vs. Cohen's kappa

Multiple `curvetype` can be combined, such as `c("precision", "sensitivity")`.

... Following additional arguments can be specified.

type A character to specify the line type as follows.

- **"l"** lines
- **"p"** points
- **"b"** both lines and points

show_cb A Boolean value to specify whether point-wise confidence bounds are drawn. It is effective only when `calc_avg` of the `evalmod()` function is set to `TRUE` .

raw_curves A Boolean value to specify whether raw curves are shown instead of the average curve. It is effective only when `raw_curves` of the `evalmod()` function is set to `TRUE`.

show_legend A Boolean value to specify whether the legend is shown.

ret_grob A logical value to indicate whether `autoplot` returns a `grob` object. The `grob` object is internally generated by `gridExtra::arrangeGrob()`. The `grid::grid.draw()` function takes a `grob` object and shows a plot. It is effective only when a multiple-panel plot is generated, for example, when `curvetype` is `c("ROC", "PRC")`.

reduce_points A Boolean value to decide whether the points should be reduced. The number kept is `x_bins` of the `evalmod()` function, per curve for `mode = "rocprc"` and per metric for `mode = "basic"`. The default is `TRUE` for the curves and `FALSE` for the basic metrics, whose points are the data rather than an interpolation. Reduction changes only what is drawn, never how it was calculated.

Value

The `autoplot` function returns a `ggplot` object for a single-panel plot and a frame-grob object for a multiple-panel plot.

See Also

`evalmod()` for generating an `S3` object. `fortify()` for converting a curves and points object to a data frame. `plot()` for plotting the equivalent curves with the general R plot.

Examples

```
## Not run:

## Load libraries
library(ggplot2)
library(grid)

##################################################
### Single model & single test dataset
###

## Load a dataset with 10 positives and 10 negatives
data(P10N10)

## Generate an sscurve object that contains ROC and Precision-Recall curves
sscurves <- evalmod(scores = P10N10$scores, labels = P10N10$labels)

## Plot both ROC and Precision-Recall curves
autoplot(sscurves)

## Reduced/Full supporting points
sampss <- create_sim_samples(1, 50000, 50000)
evalss <- evalmod(scores = sampss$scores, labels = sampss$labels)

# Reduced supporting point
system.time(autoplot(evalss))

# Full supporting points
system.time(autoplot(evalss, reduce_points = FALSE))

## Get a grob object for multiple plots
pp1 <- autoplot(sscurves, ret_grob = TRUE)
plot.new()
grid.draw(pp1)

## A ROC curve
autoplot(sscurves, curvetype = "ROC")

## A Precision-Recall curve
autoplot(sscurves, curvetype = "PRC")

## Generate an sspoints object that contains basic evaluation metrics
```

```
sspoints <- evalmod(
  mode = "basic", scores = P10N10$scores,
  labels = P10N10$labels
)

## Normalized ranks vs. basic evaluation metrics
autoplot(sspoints)

## Normalized ranks vs. precision
autoplot(sspoints, curvetype = "precision")


##################################################
### Multiple models & single test dataset
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(1, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mscurves <- evalmod(mdat)

## ROC and Precision-Recall curves
autoplot(mscurves)

## Reduced/Full supporting points
sampms <- create_sim_samples(5, 50000, 50000)
evalms <- evalmod(scores = sampms$scores, labels = sampms$labels)

# Reduced supporting point
system.time(autoplot(evalms))

# Full supporting points
system.time(autoplot(evalms, reduce_points = FALSE))

## Hide the legend
autoplot(mscurves, show_legend = FALSE)

## Generate an mspoints object that contains basic evaluation metrics
mspoints <- evalmod(mdat, mode = "basic")

## Normalized ranks vs. basic evaluation metrics
autoplot(mspoints)

## Hide the legend
autoplot(mspoints, show_legend = FALSE)


##################################################
### Single model & multiple test datasets
```

```
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(10, 100, 100, "good_er")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an smcurve object that contains ROC and Precision-Recall curves
smcurves <- evalmod(mdat, raw_curves = TRUE)

## Average ROC and Precision-Recall curves
autoplot(smcurves, raw_curves = FALSE)

## Hide confidence bounds
autoplot(smcurves, raw_curves = FALSE, show_cb = FALSE)

## Raw ROC and Precision-Recall curves
autoplot(smcurves, raw_curves = TRUE, show_cb = FALSE)

## Reduced/Full supporting points
sampsm <- create_sim_samples(4, 5000, 5000)
mdatsm <- mmdata(sampsm$scores, sampsm$labels, expd_first = "dsids")
evalsm <- evalmod(mdatsm, raw_curves = TRUE)

# Reduced supporting point
system.time(autoplot(evalsm, raw_curves = TRUE))

# Full supporting points
system.time(autoplot(evalsm, raw_curves = TRUE, reduce_points = FALSE))

## Generate an smpoints object that contains basic evaluation metrics
smpoints <- evalmod(mdat, mode = "basic")

## Normalized ranks vs. average basic evaluation metrics
autoplot(smpoints)


##################################################
### Multiple models & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(10, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mmcurves <- evalmod(mdat, raw_curves = TRUE)
```

```
## Average ROC and Precision-Recall curves
autoplot(mmcurves, raw_curves = FALSE)

## Show confidence bounds
autoplot(mmcurves, raw_curves = FALSE, show_cb = TRUE)

## Raw ROC and Precision-Recall curves
autoplot(mmcurves, raw_curves = TRUE)

## Reduced/Full supporting points
sampmm <- create_sim_samples(4, 5000, 5000)
mdatmm <- mmdata(sampmm$scores, sampmm$labels,
  modnames = c("m1", "m2"),
  dsids = c(1, 2), expd_first = "modnames"
)
evalmm <- evalmod(mdatmm, raw_curves = TRUE)

# Reduced supporting point
system.time(autoplot(evalmm, raw_curves = TRUE))

# Full supporting points
system.time(autoplot(evalmm, raw_curves = TRUE, reduce_points = FALSE))

## Generate an mmpoints object that contains basic evaluation metrics
mmpoints <- evalmod(mdat, mode = "basic")

## Normalized ranks vs. average basic evaluation metrics
autoplot(mmpoints)


##################################################
### N-fold cross validation datasets
###

## Load test data
data(M2N50F5)

## Speficy nessesary columns to create mdat
cvdat <- mmdata(
  nfold_df = M2N50F5, score_cols = c(1, 2),
  lab_col = 3, fold_col = 4,
  modnames = c("m1", "m2"), dsids = 1:5
)

## Generate an mmcurve object that contains ROC and Precision-Recall curves
cvcurves <- evalmod(cvdat)

## Average ROC and Precision-Recall curves
autoplot(cvcurves)

## Show confidence bounds
autoplot(cvcurves, show_cb = TRUE)
```

```
## Generate an mmpoints object that contains basic evaluation metrics
cvpoints <- evalmod(cvdat, mode = "basic")

## Normalized ranks vs. average basic evaluation metrics
autoplot(cvpoints)

## End(Not run)
```

`average_precision` *Calculate the average precision*

Description

The `average_precision` function returns the average precision of every precision-recall curve of an `S3` object generated by `evalmod()`. Average precision summarizes a precision-recall curve as the precision at each cutoff weighted by the recall it gains over the cutoff before it.

Usage

```
average_precision(curves)

## S3 method for class 'aucs'
average_precision(curves)
```

Arguments

`curves`	An `S3` object generated by `evalmod()`.

Value

The `average_precision` function returns a data frame with the columns `modnames`, `dsids`, `aps` and `baselines`. Unlike `auc()` it has one row per model and dataset rather than one per curve, because average precision is defined on the precision-recall curve only.

`baselines` is the proportion of positives, which is what average precision is worth by chance - see `Reading an area against its baseline` in `auc()`. It moves with the class balance, so an average precision quoted without it does not say how good the classifier is.

How this differs from the area under the curve

Average precision is $AP = \sum_i (r_i - r_{i-1}) p_i$, summed over the cutoffs from the second one on. Joining the raw precision-recall points with horizontal steps is not the same as measuring the area under the curve through them, and it is not the same number: the step estimator reads high wherever the two disagree, because linear or stepwise movement between precision-recall points overstates what a classifier can actually achieve there.

`auc()` reports the area under the curve `evalmod()` has interpolated the correct way, which is the reason this package exists, and is the number to prefer. Average precision is reported because

several other packages report it under this name, and because the size of the gap between the two is itself worth seeing.

The sum starts at the second cutoff, so the precision of the empty prediction set - `0/0`, which every tool defines differently - never enters it.

See Also

`auc()` for the areas under the ROC and precision-recall curves. `prbe()` for the precision-recall break-even point. `evalmod()` for generating `S3` objects with performance evaluation metrics.

Examples

```
##################################################
### Single model & single test dataset
###
samps <- create_sim_samples(1, 50, 50, "good_er")
sscurves <- evalmod(scores = samps[["scores"]], labels = samps[["labels"]])
average_precision(sscurves)

## The area under the properly interpolated curve, for comparison
auc(sscurves)

##################################################
### Multiple models & multiple test datasets
###
samps <- create_sim_samples(4, 50, 50, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]], dsids = samps[["dsids"]]
)
mmcurves <- evalmod(mdat)
average_precision(mmcurves)
```

`B1000` *Balanced data with 1000 positives and 1000 negatives.*

Description

A list contains labels and scores of five different performance levels. All scores were randomly generated.

Usage

```
data(B1000)
```

Format

A list with 8 items.

np number of positives: 1000

nn number of negatives: 1000

labels labels of observed data

random_scores scores of a random performance level

poor_er_scores scores of a poor early retrieval level

good_er_scores scores of a good early retrieval level

excel_scores scores of an excellent level

perf_scores scores of the perfect level

B500 *Balanced data with 500 positives and 500 negatives.*

Description

A list contains labels and scores of five different performance levels. All scores were randomly generated.

Usage

```
data(B500)
```

Format

A list with 8 items.

np number of positives: 500

nn number of negatives: 500

labels labels of observed data

random_scores scores of a random performance level

poor_er_scores scores of a poor early retrieval level

good_er_scores scores of a good early retrieval level

excel_scores scores of an excellent level

perf_scores scores of the perfect level

best_cutoff *Choose an operating point*

Description

The best_cutoff function picks the cutoff that optimizes one evaluation metric, in the manner of pROC::coords(x, "best"). It returns one row per model per test dataset - a row of metric_table(), together with the criterion that chose it.

Usage

```
best_cutoff(x, scores = NULL, labels = NULL, metric = "youden", ...)
```

Arguments

x — An S3 object created by the mmdata() function, or a basic-metric object created by evalmod(mode = "basic"). The best_cutoff function ignores scores and labels when x is specified. These arguments are internally passed to the mmdata() function when x is unspecified. In that case, both scores and labels must be at least specified.

scores — A numeric dataset of predicted scores. It can be a vector, a matrix, an array, a data frame, or a list.

labels — A numeric, character, logical, or factor dataset of observed labels. It can be a vector, a matrix, an array, a data frame, or a list.

metric — A string naming the metric to optimize. It can be any of the metrics evalmod() calculates, together with two names that come from the cutpoint literature rather than from the metric table.

Name	Metric	Optimum
"youden"	informedness	Maximum
"topleft"	roc_dist	Minimum
"fscore"	fscore	Maximum
"mcc"	mcc	Maximum
"cost"	cost	Minimum

The direction is a property of the metric, so it is not an argument - error, cost, roc_dist, the two error rates and the negative likelihood ratio are minimized, and the rest are maximized. score, label, predicted_positive_rate and predicted_negative_rate describe a cutoff rather than score it, and are refused.

... — These additional arguments are passed to evalmod(), and through it to mmdata(), when this function builds the metrics itself. beta for "fscore", cost_fp and cost_fn for "cost", and basic_ties, modnames, dsids and posclass are the useful ones here.

Value

The `best_cutoff` function returns a data frame with one row per model per test dataset, and the following columns.

`modname`	Model name
`dsid`	Test dataset ID
`metric`	The metric that was optimized
`value`	Its value at the chosen cutoff
`rank`	Number of instances called positive, `1` to `n`
`normalized_rank`	`rank / n`
`score`	The cutoff itself, the rule being `score >=  it`
`label`	`1` if the instance at this rank is positive, `-1` if it is negative
...	One column per metric, as `metric_table()` returns them

The row is a row of `metric_table()`, so every other metric is there to be read at the same cutoff - which is the point of returning the whole row rather than the threshold alone.

The criterion matters more than the threshold

Youden's J and the closest point to the top left corner are the two criteria most often reached for, and both are computed from sensitivity and specificity alone. Both of those are conditioned on the true class, so neither knows the prevalence, and on imbalanced data the cutoff they choose can sit at a precision no one would deploy. That is the argument this package makes about the ROC curve, one step further down the pipeline.

Prevalence-blind	`"youden"`, `"topleft"`
Prevalence-aware	`"fscore"`, `"mcc"`, `"cost"`

The default is `"youden"` because that is what a caller arriving from another package expects. On imbalanced data it is the wrong default, and `"mcc"` or a `"cost"` weighted by what the two mistakes actually cost is the better choice. The *Balanced and imbalanced data* article shows the two disagreeing on the same dataset: https://evalclass.github.io/precrec/articles/howto-imbalanced-data.html.

Ties, and what is not here

Several cutoffs can share the optimum. The one with the smallest `rank` is returned - the threshold that calls the fewest instances positive - so the result is one row per model per test dataset whatever the data does.

Rows that share a `score` are an exception, because they are one threshold seen several times rather than several cutoffs: `score >=  ` that value calls all of the tied instances positive, so the `rank` reported is the last of the run and not the first. That is what `evalmod(basic_ties = "hold")` produces throughout, since it gives every cutoff in a run of tied scores the counts of the whole run.

The rank `0` row of `metric_table()` - the rule that calls nothing positive - is not a candidate. There is no threshold that predicts nothing, so its `score` is `NA` and it is not an operating point, which is

what this function returns. It would otherwise win outright wherever the empty rule is optimal, and win on the tie-break wherever it merely ties: precision at rank `0` is the limit from above, so it is `1` whenever the top-ranked instance is a positive, and any classifier that ranks one first would come back with no threshold at all.

A metric that has no interior optimum is still optimized at an end of the range, and no warning says so: `sensitivity` is largest when everything is called positive, and `specificity` when as little as possible is. Those are the correct answers to the question asked, and rarely the question meant. `accuracy` and `error` have an interior optimum on balanced data and lose it as positives get rare, which is the same trap arriving by a different route: at a few percent positives, calling almost nothing positive is close to the most accurate thing a classifier can do.

A cutoff chosen on the same data the model is evaluated on is optimistic, by however much the criterion was free to chase. Choosing it on held-out data is the fix, and it is the caller's to make: nothing here resamples or cross-validates the choice.

See Also

`metric_table()` for every metric at every cutoff, which this function takes one row of, `evalmod()` for calculating the metrics, and `prbe()` for the precision-recall break-even point, which is an operating point defined by a crossing rather than by an optimum.

Examples

```
##################################################
### Single model & single test dataset
###

## Load a dataset with 10 positives and 10 negatives
data(P10N10)

## The cutoff that maximizes Youden's J
best_cutoff(scores = P10N10$scores, labels = P10N10$labels)


##################################################
### A criterion that knows the prevalence
###

best_cutoff(scores = P10N10$scores, labels = P10N10$labels, metric = "mcc")

## Weighted by what the two mistakes cost
best_cutoff(
  scores = P10N10$scores, labels = P10N10$labels,
  metric = "cost", cost_fp = 1, cost_fn = 5
)


##################################################
### Multiple models & multiple test datasets
###

samps <- create_sim_samples(2, 50, 50, c("poor_er", "good_er"))
```

```
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]], dsids = samps[["dsids"]]
)
best_cutoff(mdat, metric = "fscore")[, 1:7]
```

C3N150 *Multiclass sample with three classes.*

Description

A list contains labels and one score column per class for a 3-class dataset. The three classes are separated to different degrees, so that a one-vs-rest evaluation of them shows three clearly different curves.

Usage

```
data(C3N150)
```

Format

A list with 2 items.

scores a 150 by 3 matrix of scores, one column per class, with the column names 'c1', 'c2' and 'c3'

labels 150 labels as 'c1', 'c2' or 'c3', 50 of each

---

classification_report *Classification report*

Description

`classification_report` builds the per-class table of precision, recall and F-score that `scikit-learn`'s `classification_report` prints, together with the summary rows that go under it.

Usage

```
classification_report(
  mdat,
  scores = NULL,
  labels = NULL,
  at = NULL,
  zero_division = 0,
  ...
)
```

Arguments

`mdat`	An `mdata` object created by `mmdata()`. It can be omitted when `scores` and `labels` are given.
`scores`	A numeric vector, matrix, array, data frame, or list of scores. See `mmdata()` for the accepted shapes.
`labels`	A numeric, character, logical, or factor vector of observed labels, or a list of such vectors.
`at`	The operating point, as a score threshold. An observation is predicted positive for a class when its score for that class is greater than or equal to the threshold. It has no default and must be given; see the note below on why. Either • a single number, used for every class, or • one number per class, named after the classes or given in their order. A score of `NA` is never predicted positive, matching the default `na_worst = TRUE` of the rest of the package.
`zero_division`	The value reported when a precision, recall or F-score divides by zero - a class nothing was predicted into, or a class with no observations. `0` by default, as in `scikit-learn`. Use `NA` for the convention the per-cutoff metrics of `evalmod()` follow.
`...`	Further arguments passed to `mmdata()` when `scores` and `labels` are given instead of `mdat`.

Value

A data frame with one row per class and one row per summary, and the columns

`modnames`	Model name
`dsids`	Dataset ID
`class`	Class name, or the name of a summary row
`precision`	Predicted positives that are positive
`recall`	Positives that are predicted positive
`fscore`	Harmonic mean of the two
`support`	Observations of the class

The object also has the class `classification_report`, which only affects how it prints; it is a data frame in every other respect.

The summary rows

`macro avg` is the unweighted mean over the classes and `weighted avg` the mean weighted by support. The third row depends on whether the predictions assign each observation to exactly one class, which is the rule `scikit-learn` documents:

- A binary problem at a threshold predicts each observation into exactly one of the two classes, so the row is `accuracy`.

- A multi-class problem is evaluated by one-vs-rest, and each class is thresholded on its own, so an observation can fall into no class or into several. There is then no single-label accuracy, and the row is `micro avg`: precision, recall and F-score recomputed from the true positives, false positives and false negatives pooled over the classes.

Why `at` has no default

`scikit-learn` reports on `y_pred`, so the caller has already chosen an operating point before the function is called. `precrec` holds scores and evaluates every cutoff, so the report has to be told which one to use, and the answer changes the table: on the three-class `C3N150` the F-score of one class moves from 0.37 to 0.51 between two reasonable choices. Scores in `precrec` are on whatever scale the classifier produced, so there is no threshold that is meaningful for all of them - `0.5` says nothing about a log-odds or an SVM margin. The threshold is therefore always the caller's.

See Also

`evalmod()` for the same metrics at every cutoff, `auc()` for the threshold-free summaries and their macro averages, and `prob_metrics()` for the probability-based losses.

Examples

```
## Multi-class: one row per class, then micro, macro and weighted averages
data(C3N150)
mdat <- mmdata(C3N150$scores, C3N150$labels)
classification_report(mdat, at = 0.5)

## A threshold per class, named or in class order
classification_report(mdat, at = c(c1 = 0.4, c2 = 0.5, c3 = 0.6))

## Binary: both classes, and an accuracy row rather than a micro average
data(P10N10)
classification_report(
  scores = P10N10$scores, labels = P10N10$labels,
  at = 12
)

## It is a data frame, so the usual accessors work
report <- classification_report(mdat, at = 0.5)
report[report$class == "macro avg", ]
```

`create_sim_samples` *Create random samples for simulations*

Description

The `create_sim_samples` function generates random samples with different performance levels.

Usage

```
create_sim_samples(n_repeat, np, nn, score_names = "random")
```

Arguments

`n_repeat`	The number of iterations to make samples.
`np`	The number of positives in a sample.
`nn`	The number of negatives in a sample.
`score_names`	A character vector for the names of the following performance levels. **"random"** Random **"poor_er"** Poor early retrieval **"good_er"** Good early retrieval **"excel"** Excellent **"perf"** Perfect **"all"** All of the above

Value

The `create_sim_samples` function returns a list with the following items.

- scores: a list of numeric vectors
- labels: an integer vector
- modnames: a character vector of the model names
- dsids: a character vector of the dataset IDs

See Also

`mmdata()` for formatting input data. `evalmod()` for calculation evaluation metrics.

Examples

```
##################################################
### Create a set of samples with 10 positives and 10 negatives
### for the random performance level
###
samps1 <- create_sim_samples(1, 10, 10, "random")

## Show the list structure
str(samps1)


##################################################
### Create two sets of samples with 10 positives and 20 negatives
### for the random and the poor early retrieval performance levels
###
samps2 <- create_sim_samples(2, 10, 20, c("random", "poor_er"))

## Show the list structure
```

```
str(samps2)


##################################################
### Create 3 sets of samples with 5 positives and 5 negatives
### for all 5 levels
###
samps3 <- create_sim_samples(3, 5, 5, "all")

## Show the list structure
str(samps3)
```

---

evalmod *Evaluate models and calculate performance evaluation metrics*

Description

The evalmod function calculates ROC and Precision-Recall curves for specified prediction scores and binary labels. It also calculate several basic performance evaluation metrics, such as accuracy, error rate, and precision, by specifying mode as "basic".

Usage

```
evalmod(
  mdat,
  mode = NULL,
  scores = NULL,
  labels = NULL,
  modnames = NULL,
  dsids = NULL,
  posclass = NULL,
  na_worst = TRUE,
  ties_method = "equiv",
  calc_avg = TRUE,
  cb_alpha = 0.05,
  raw_curves = FALSE,
  x_bins = 1000,
  interpolate = TRUE,
  beta = 1,
  on_single_class = "error",
  metrics = NULL,
  cost_fp = 1,
  cost_fn = 1,
  basic_ties = "split",
  ...
)
```

Arguments

`mdat`

An `S3` object created by the `mmdata()` function. It contains formatted scores and labels. The `evalmod` function ignores the following arguments when `mdat` is specified.

- `scores`
- `labels`
- `modnames`
- `dsids`
- `posclass`
- `na_worst`
- `ties_method`

These arguments are internally passed to the `mmdata()` function when `mdat` is unspecified. In that case, both `scores` and `labels` must be at least specified.

`mode`

A string that specifies the types of evaluation metrics that the `evalmod` function calculates.

"rocprc" ROC and Precision-Recall curves

"prcroc" Same as above

"basic" Normalized ranks vs. accuracy, error rate, specificity, sensitivity, precision, Matthews correlation coefficient, F-score, balanced accuracy, negative predictive value, informedness, markedness, and Cohen's kappa.

"aucroc" Fast AUC(ROC) calculation with the U statistic

`scores`

A numeric dataset of predicted scores. It can be a vector, a matrix, an array, a data frame, or a list. The `join_scores()` function can be useful to make scores with multiple datasets.

`labels`

A numeric, character, logical, or factor dataset of observed labels. It can be a vector, a matrix, an array, a data frame, or a list. The `join_labels()` function can be useful to make labels with multiple datasets.

`modnames`

A character vector for the names of the models. The `evalmod` function automatically generates default names as "m1", "m2", "m3", and so on when it is `NULL`.

`dsids`

A numeric vector for test dataset IDs. The `evalmod` function automatically generates the default ID as `1` when it is `NULL`.

`posclass`

A scalar value to specify the label of positives in `labels`. It must be the same data type as `labels`. For example, `posclass = -1` changes the positive label from `1` to `-1` when `labels` contains `1` and `-1`. The positive label will be automatically detected when `posclass` is `NULL`.

`na_worst`

A Boolean value for controlling the treatment of NAs in `scores`.

TRUE All NAs are treated as the worst scores

FALSE All NAs are treated as the best scores

`ties_method`

A string for controlling ties in `scores`.

"equiv" Ties are equivalently ranked

"first" Ties are ranked in an increasing order as appeared

"random" Ties are ranked in random order

`calc_avg` A logical value to specify whether average curves should be calculated. It is effective only when `dsids` contains multiple dataset IDs. For instance, the function calculates the average for the model "m1" when `modnames` is `c("m1", "m1", "m1")` and `dsids` is `c(1, 2, 3)`. The calculation points are defined by `x_bins`.

`cb_alpha` A numeric value with range [0, 1] to specify the alpha value of the point-wise confidence bounds calculation. It is effective only when `calc_avg` is set to `TRUE`. For example, it should be `0.05` for the 95% confidence level. The calculation points are defined by `x_bins`.

`raw_curves` A logical value to specify whether all raw curves should be discarded after the average curves are calculated. It is effective only when `calc_avg` is set to `TRUE`.

`x_bins` An integer value to specify the number of minimum bins on the x-axis. It is then used to define supporting points For instance, the x-values of the supporting points will be `c(0, 0.5, 1)` and `c(0, 0.25, 0.5, 0.75, 1)` when `x_bins = 2` and `x_bins = 4`, respectively. All corresponding y-values of the supporting points are calculated. `x_bins` places supporting points only when `mode` is set to `rocprc` or `prcroc`; with `mode = "basic"` there is no interpolation to place them on, and the value is instead the number of points kept per metric when a plot or a data frame is asked for with `reduce_points = TRUE`. It must be `1e6` or smaller; every stage sized by it allocates a vector of that length per curve, and a million supporting points is already finer than a plot resolves.

`interpolate` A Boolean value to specify whether or not interpolation of ROC and precision-recall curves are performed. `x_bins` and `calc_avg` are ignored and when `x_bins` is set to `FALSE`. `interpolate` is effective only when `mode` is set to `rocprc` or `prcroc`.

`beta` A numeric value to specify the beta of the F-beta score, which weights recall `beta` times as heavily as precision. The default `1` gives the F1 score. `beta` is effective only when `mode` is set to `basic`.

`on_single_class`
A string that specifies what the `evalmod` function does with a dataset in which every label belongs to the same class.

"error" Raise an error (default)

"na" Warn, and return `NA` for the metrics that are undefined

ROC and precision-recall curves are undefined for such a dataset, so `on_single_class` is effective only when `mode` is set to `rocprc`, `prcroc`, or `aucroc`. `mode = "basic"` always warns and calculates what it can, because accuracy and error rate are still defined.

`metrics` A character vector that names the basic evaluation metrics to calculate in addition to the default set, or the string `"all"` for every metric `precrec` knows. The default `NULL` is the fourteen metrics `evalmod` has always returned: `score`, `label`, `error`, `accuracy`, `specificity`, `sensitivity`, `precision`, `mcc`, `fscore`, `balanced_accuracy`, `npv`, `informedness`, `markedness` and `kappa`.

The metrics that can be added are `fpr`, `fnr`, `false_discovery_rate`, `false_omission_rate`, `predicted_positive_rate`, `predicted_negative_rate`, `lift`, `odds`, `mi`, `chisq`,

cost and sar. They are the metrics ROCR provides that precrec did not, and each of them also answers to the identifier ROCR uses for it - fall, miss, pcfall, pcmiss, rpp, rnp and mutual_information - and to its standard abbreviation where it has one.

roc_dist and sedi can be added on the same footing. roc_dist is the distance from (1 - specificity, sensitivity) to the perfect corner of ROC space, and is the one metric here that is better when it is smaller. sedi is the symmetric extremal dependence index, a skill score built to stay informative when the positive class is rare.

jaccard, positive_likelihood_ratio and negative_likelihood_ratio come from scikit-learn. jaccard is the Jaccard index, also called the critical success index: TP / (TP + FP + FN), the confusion matrix with its true negative corner left out, which is the same omission precision and sensitivity make. The two likelihood ratios are sensitivity / fpr and fnr / specificity; odds, the diagnostic odds ratio, is their quotient.

They are not calculated by default because each is another vector the size of the dataset, and because plot and autoplot draw one panel per metric the object holds. A metric that was not asked for cannot be plotted; metrics is effective only when mode is set to basic.

cost_fp — A numeric value for the cost of a false positive, used by the cost metric. cost is not normalized, following ROCR: it is cost_fp * FP / n + cost_fn * FN / n, which with the default weights of 1 is the error rate. cost_fp is effective only when mode is set to basic and metrics asks for cost.

cost_fn — A numeric value for the cost of a false negative.

basic_ties — A string that specifies what the basic evaluation metrics report at the cutoffs inside a run of tied scores. basic_ties is effective only when mode is set to basic.

"split" Spread the true and false positives of the run evenly over its cutoffs (default). This is the interpolation the ROC and precision-recall curves need, and is what precrec has always done.

"hold" Give every cutoff in the run the counts it has once the whole run is taken, so tied instances share one value of every metric.

A cutoff inside a tied run splits instances that share a score, so no threshold produces it. With "split" a perfect classifier whose scores are all 0 or 1 reports sensitivity climbing from 0 to 1 across the positives rather than reaching 1 at once. "hold" makes each metric a step function that changes only where the score does. The two agree whenever the scores are all distinct, and "split" is kept as the default because it is what every published precrec result was computed with.

... — These additional arguments are passed to mmdata() for data preparation. multiclass = "ovr" asks for a one-vs-rest evaluation of a dataset with more than two classes; see mmdata().

Value

The evalmod function returns an S3 object that contains performance evaluation metrics. The number of models and the number of datasets can be controlled by modnames and dsids. For example,

the number of models is "single" and the number of test datasets is "multiple" when `modnames = c("m1", "m1", "m1")` and `dsids = c(1, 2, 3)` are specified.

Different `S3` objects have different default behaviors of `S3` generics, such as `plot()`, `autoplot()`, and `fortify()`.

1. The `evalmod` function returns one of the following `S3` objects when `mode` is "prcroc". The objects contain ROC and Precision-Recall curves.

`S3` object	# of models	# of test datasets
sscurves	single	single
mscurves	multiple	single
smcurves	single	multiple
mmcurves	multiple	multiple

2. The `evalmod` function returns one of the following `S3` objects when `mode` is "basic". They contain the per-rank basic evaluation metrics; error rate, accuracy, specificity, sensitivity, precision, Matthews correlation coefficient, F-score, balanced accuracy, negative predictive value, informedness, markedness, and Cohen's kappa.

`S3` object	# of models	# of test datasets
sspoints	single	single
mspoints	multiple	single
smpoints	single	multiple
mmpoints	multiple	multiple

3. The `evalmod` function returns the `aucroc` S3 object when `mode` is "aucroc", which can be used with 'print' and 'as.data.frame'.

See Also

`plot()` for plotting curves with the general R plot. `autoplot()` and `fortify()` for plotting curves with **ggplot2**. `mmdata()` for formatting input data. `join_scores()` and `join_labels()` for formatting scores and labels with multiple datasets. `format_nfold()` for creating n-fold cross validation dataset from data frame. `create_sim_samples()` for generating random samples for simulations.

Examples

```
##################################################
### Single model & single test dataset
###

## Load a dataset with 10 positives and 10 negatives
data(P10N10)

## Generate an sscurve object that contains ROC and Precision-Recall curves
sscurves <- evalmod(scores = P10N10$scores, labels = P10N10$labels)
sscurves
```

```
## Generate an sspoints object that contains basic evaluation metrics
sspoints <- evalmod(
  mode = "basic", scores = P10N10$scores,
  labels = P10N10$labels
)
sspoints

## Let tied scores share one value of every basic metric
tiedpoints <- evalmod(
  mode = "basic", scores = round(P10N10$scores, 1),
  labels = P10N10$labels, basic_ties = "hold"
)
tiedpoints


##################################################
### Multiple models & single test dataset
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(1, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mscurves <- evalmod(mdat)
mscurves

## Generate an mspoints object that contains basic evaluation metrics
mspoints <- evalmod(mdat, mode = "basic")
mspoints


##################################################
### Single model & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(4, 100, 100, "good_er")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an smcurve object that contains ROC and Precision-Recall curves
smcurves <- evalmod(mdat)
smcurves

## Generate an smpoints object that contains basic evaluation metrics
smpoints <- evalmod(mdat, mode = "basic")
smpoints
```

```
##################################################
### Multiple models & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(4, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an mmcurve object that contains ROC and Precision-Recall curves
mmcurves <- evalmod(mdat)
mmcurves

## Generate an mmpoints object that contains basic evaluation metrics
mmpoints <- evalmod(mdat, mode = "basic")
mmpoints


##################################################
### N-fold cross validation datasets
###

## Load test data
data(M2N50F5)

## Speficy nessesary columns to create mdat
cvdat <- mmdata(
  nfold_df = M2N50F5, score_cols = c(1, 2),
  lab_col = 3, fold_col = 4,
  modnames = c("m1", "m2"), dsids = 1:5
)

## Generate an mmcurve object that contains ROC and Precision-Recall curves
cvcurves <- evalmod(cvdat)
cvcurves

## Generate an mmpoints object that contains basic evaluation metrics
cvpoints <- evalmod(cvdat, mode = "basic")
cvpoints

## Specify mmdata arguments from evalmod
cvcurves2 <- evalmod(
  nfold_df = M2N50F5, score_cols = c(1, 2),
  lab_col = 3, fold_col = 4,
  modnames = c("m1", "m2"), dsids = 1:5
)
cvcurves2
```

```
##################################################
### AUC with the U statistic
###

## mode = "aucroc" returns 'aucroc' S3 object
data(P10N10)

# 'aucroc' S3 object
uauc1 <- evalmod(
  scores = P10N10$scores, labels = P10N10$labels,
  mode = "aucroc"
)

# print 'aucroc'
uauc1

# as.data.frame 'aucroc'
as.data.frame(uauc1)

## It is 2-3 times faster than mode = "rocprc"
# A sample of 100,000
samp1 <- create_sim_samples(1, 50000, 50000)

# a function to test mode = "rocprc"
func_evalmod_rocprc <- function(samp) {
  curves <- evalmod(scores = samp$scores, labels = samp$labels)
  aucs <- auc(curves)
}

# a function to test mode = "aucroc"
func_evalmod_aucroc <- function(samp) {
  uaucs <- evalmod(
    scores = samp$scores, labels = samp$labels,
    mode = "aucroc"
  )
  as.data.frame(uaucs)
}

# Process time
system.time(res1 <- func_evalmod_rocprc(samp1))
system.time(res2 <- func_evalmod_aucroc(samp1))

# AUCs
res1
res2


##################################################
### Multiclass evaluation
###

## Load a 3-class dataset with one score column per class
data(C3N150)
```

```
## Each class is evaluated against the rest
mccurves <- evalmod(scores = C3N150$scores, labels = C3N150$labels)
mccurves

## Per-class AUCs, plus their macro-average
auc(mccurves)
```

---

format_nfold *Create n-fold cross validation dataset from data frame*

Description

The format_nfold function takes a data frame with scores, label, and n-fold columns and convert it to a list for evalmod() and mmdata().

Usage

```
format_nfold(nfold_df, score_cols, lab_col, fold_col)
```

Arguments

nfold_df	A data frame that contains at least one score column, label and fold columns.
score_cols	A character/numeric vector that specifies score columns of nfold_df.
lab_col	A number/string that specifies the label column of nfold_df.
fold_col	A number/string that specifies the fold column of nfold_df.

Value

The format_nfold function returns a list that contains multiple scores and labels.

See Also

evalmod() for calculation evaluation metrics. mmdata() for formatting input data. join_scores() and join_labels() for formatting scores and labels with multiple datasets.

Examples

```
##################################################
### Convert dataframe with 2 models and 5-fold datasets
###

## Load test data
data(M2N50F5)
head(M2N50F5)

## Convert with format_nfold
```

```
nfold_list1 <- format_nfold(
  nfold_df = M2N50F5, score_cols = c(1, 2),
  lab_col = 3, fold_col = 4
)

## Show the list structure
str(nfold_list1)
str(nfold_list1$scores)
str(nfold_list1$labels)


##################################################
### Speficy a single score column
###

## Convert with format_nfold
nfold_list2 <- format_nfold(
  nfold_df = M2N50F5, score_cols = 1,
  lab_col = 3, fold_col = 4
)

## Show the list structure
str(nfold_list2)
str(nfold_list2$scores)
str(nfold_list2$labels)


##################################################
### Use column names
###

## Convert with format_nfold
nfold_list3 <- format_nfold(
  nfold_df = M2N50F5,
  score_cols = c("score1", "score2"),
  lab_col = "label", fold_col = "fold"
)

## Show the list structure
str(nfold_list3)
str(nfold_list3$scores)
str(nfold_list3$labels)
```

`format_points` *Reconstruct per-instance data from a table of curve points*

Description

The `format_points` function takes a data frame of performance values calculated at a set of thresholds - true and false positive rates, or recall and precision - and reconstructs per-instance scores and

labels that reproduce those points exactly. The result is a list for `evalmod()` and `mmdata()`.

Usage

```
format_points(
  points_df,
  threshold_col,
  tpr_col = NULL,
  fpr_col = NULL,
  rec_col = NULL,
  prec_col = NULL,
  np = NULL,
  nn = NULL,
  mod_col = NULL,
  dsid_col = NULL
)
```

Arguments

`points_df`	A data frame with one row per threshold, holding a threshold column, two rate columns, and optionally model and dataset columns.
`threshold_col`	A number/string that specifies the threshold column of `points_df`.
`tpr_col`	A number/string that specifies the true positive rate column of `points_df`.
`fpr_col`	A number/string that specifies the false positive rate column of `points_df`.
`rec_col`	A number/string that specifies the recall column of `points_df`. Recall and the true positive rate are the same quantity; supply one pair of columns, either `tpr_col`/`fpr_col` or `rec_col`/`prec_col`.
`prec_col`	A number/string that specifies the precision column of `points_df`.
`np`	The number of positives. A single number, a numeric vector with one element per group, or a number/string that specifies a column of `points_df`.
`nn`	The number of negatives, in the same three forms as `np`.
`mod_col`	A number/string that specifies the model column of `points_df`. `NULL` treats the whole table as one model.
`dsid_col`	A number/string that specifies the dataset column of `points_df`. `NULL` treats the whole table as one dataset.

Details

A table of rates is a set of supporting points on a curve, and the instances between two adjacent points are missing. `format_points` puts them back: the rows between two thresholds become instances that share a score, and precrec spreads the true and false positives of a tied run evenly over the cutoffs inside it. That even spread is the non-linear interpolation of Davis and Goadrich (2006), so the reconstructed curve is the one that interpolation prescribes.

The interpolation is exact only where a gap holds a single instance. Over the wider gaps a threshold table leaves, it assumes the positives and negatives inside a gap alternate at a constant rate, and

the area under the resulting curve is an estimate whose error the table gives no way to bound. Per-instance scores and labels remain the input to prefer wherever they exist.

np and nn are required. Recall and precision determine the class skew but not the totals, and true and false positive rates determine neither, so the counts cannot be recovered without them. They also place the chance-level baseline of the precision-recall curve, and supply the instances below the lowest threshold when the table stops short of predicting everything positive.

Reconstructed scores are threshold values, not per-instance predictions. Everything that reads the *ranking* is exact - the curves, the areas, the basic metrics - but prob_metrics() reads the values themselves, and a Brier score or log loss taken from reconstructed data means nothing.

The direction of the threshold column is inferred from the rates. Both conventions work - a threshold that keeps fewer instances as it rises, and one that keeps more - and the reconstructed scores are negated for the second so that a larger score always means a more likely positive.

Value

The format_points function returns a list that contains scores, labels, model names and dataset IDs.

References

Davis J, Goadrich M (2006) The relationship between precision-recall and ROC curves. *Proceedings of the 23rd International Conference on Machine Learning*, 233-240. doi:10.1145/1143844.1143874

See Also

evalmod() for calculation evaluation metrics. mmdata() for formatting input data. format_nfold() for cross validation data frames.

Examples

```
##################################################
### A single model, true and false positive rates
###

roc_df <- data.frame(
  threshold = c(0.9, 0.7, 0.5, 0.3, 0.1),
  tpr = c(0.2, 0.5, 0.7, 0.9, 1.0),
  fpr = c(0.02, 0.10, 0.25, 0.55, 1.0)
)

pts1 <- format_points(roc_df,
  threshold_col = "threshold",
  tpr_col = "tpr", fpr_col = "fpr",
  np = 50, nn = 100
)

evalmod(mmdata(pts1$scores, pts1$labels))


##################################################
```

```
### Two models, recall and precision
###

prc_df <- data.frame(
  model = rep(c("m1", "m2"), each = 4),
  threshold = rep(c(0.8, 0.6, 0.4, 0.2), 2),
  recall = c(0.3, 0.6, 0.8, 1.0, 0.2, 0.4, 0.7, 1.0),
  precision = c(0.9, 0.8, 0.6, 0.4, 0.7, 0.6, 0.5, 0.4)
)

pts2 <- format_points(prc_df,
  threshold_col = "threshold",
  rec_col = "recall", prec_col = "precision",
  mod_col = "model", np = 40, nn = 60
)

evalmod(mmdata(pts2$scores, pts2$labels, modnames = pts2$modnames))
```

`fortify` *Convert a curves and points object to a data frame for ggplot2*

Description

The `fortify` function converts an `S3` object generated by `evalmod()` to a data frame for **ggplot2**.

Usage

```
## S3 method for class 'sscurves'
fortify(model, data, raw_curves = NULL, reduce_points = FALSE, ...)

## S3 method for class 'mscurves'
fortify(model, data, raw_curves = NULL, reduce_points = FALSE, ...)

## S3 method for class 'smcurves'
fortify(model, data, raw_curves = NULL, reduce_points = FALSE, ...)

## S3 method for class 'mmcurves'
fortify(model, data, raw_curves = NULL, reduce_points = FALSE, ...)

## S3 method for class 'sspoints'
fortify(model, data, raw_curves = NULL, reduce_points = FALSE, ...)

## S3 method for class 'mspoints'
fortify(model, data, raw_curves = NULL, reduce_points = FALSE, ...)

## S3 method for class 'smpoints'
fortify(model, data, raw_curves = NULL, reduce_points = FALSE, ...)
```

```
## S3 method for class 'mmpoints'
fortify(model, data, raw_curves = NULL, reduce_points = FALSE, ...)

## S3 method for class 'ssxycurves'
fortify(model, data, ...)

## S3 method for class 'msxycurves'
fortify(model, data, ...)

## S3 method for class 'smxycurves'
fortify(model, data, ...)

## S3 method for class 'mmxycurves'
fortify(model, data, ...)
```

Arguments

`model` An `S3` object generated by `evalmod()`. The `fortify` function takes one of the following `S3` objects.

1. ROC and Precision-Recall curves (mode = "rocprc")

`S3` object	# of models	# of test datasets
sscurves	single	single
mscurves	multiple	single
smcurves	single	multiple
mmcurves	multiple	multiple

2. Basic evaluation metrics (mode = "basic")

`S3` object	# of models	# of test datasets
sspoints	single	single
mspoints	multiple	single
smpoints	single	multiple
mmpoints	multiple	multiple

See the **Value** section of `evalmod()` for more details.

`data` Not used by this method.

`raw_curves` A Boolean value to specify whether raw curves are shown instead of the average curve. It is effective only when `raw_curves` is set to `TRUE` of the `evalmod()` function. A single test dataset has no average to contrast the raw curves with, so `sscurves` and `sspoints`, and likewise `mscurves` and `mspoints`, always show the raw curves.

`reduce_points` A Boolean value to decide whether the points should be reduced. The number kept is `x_bins` of the `evalmod()` function, per curve for `mode = "rocprc"` and per metric for `mode = "basic"`. The default value is `FALSE`. Reduction changes

only which points are returned, never how they were calculated, and nothing is dropped from an object that already holds `x_bins` points or fewer.

`...` Not used by this method.

Value

The `fortify` function returns a data frame for **ggplot2**.

See Also

`evalmod()` for generating `S3` objects with performance evaluation metrics. `autoplot()` for plotting with **ggplot2**.

Examples

```
## Not run:

## Load library
library(ggplot2)

##################################################
### Single model & single test dataset
###

## Load a dataset with 10 positives and 10 negatives
data(P10N10)

## Generate an sscurve object that contains ROC and Precision-Recall curves
sscurves <- evalmod(scores = P10N10$scores, labels = P10N10$labels)

## Let ggplot internally call fortify
p_rocprc <- ggplot(sscurves, aes(x = x, y = y))
p_rocprc <- p_rocprc + geom_line()
p_rocprc <- p_rocprc + facet_wrap(~curvetype)
p_rocprc

## Explicitly fortify sscurves
ssdf <- fortify(sscurves)

## Plot a ROC curve
p_roc <- ggplot(subset(ssdf, curvetype == "ROC"), aes(x = x, y = y))
p_roc <- p_roc + geom_line()
p_roc

## Plot a Precision-Recall curve
p_prc <- ggplot(subset(ssdf, curvetype == "PRC"), aes(x = x, y = y))
p_prc <- p_prc + geom_line()
p_prc

## Generate an sspoints object that contains basic evaluation metrics
sspoints <- evalmod(
  mode = "basic", scores = P10N10$scores,
```

```
  labels = P10N10$labels
)
## Fortify sspoints
ssdf <- fortify(sspoints)

## Plot normalized ranks vs. precision
p_prec <- ggplot(subset(ssdf, curvetype == "precision"), aes(x = x, y = y))
p_prec <- p_prec + geom_point()
p_prec


##################################################
### Multiple models & single test dataset
###

## Create sample datasets with 10 positives and 10 negatives
samps <- create_sim_samples(1, 10, 10, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mscurves <- evalmod(mdat)

## Let ggplot internally call fortify
p_rocprc <- ggplot(mscurves, aes(x = x, y = y, color = modname))
p_rocprc <- p_rocprc + geom_line()
p_rocprc <- p_rocprc + facet_wrap(~curvetype)
p_rocprc

## Explicitly fortify mscurves
msdf <- fortify(mscurves)

## Plot ROC curve
df_roc <- subset(msdf, curvetype == "ROC")
p_roc <- ggplot(df_roc, aes(x = x, y = y, color = modname))
p_roc <- p_roc + geom_line()
p_roc

## Fortified data frame can be used for plotting a Precision-Recall curve
df_prc <- subset(msdf, curvetype == "PRC")
p_prc <- ggplot(df_prc, aes(x = x, y = y, color = modname))
p_prc <- p_prc + geom_line()
p_prc

## Generate an mspoints object that contains basic evaluation metrics
mspoints <- evalmod(mdat, mode = "basic")

## Fortify mspoints
msdf <- fortify(mspoints)

## Plot normalized ranks vs. precision
df_prec <- subset(msdf, curvetype == "precision")
```

```
p_prec <- ggplot(df_prec, aes(x = x, y = y, color = modname))
p_prec <- p_prec + geom_point()
p_prec


##################################################
### Single model & multiple test datasets
###

## Create sample datasets with 10 positives and 10 negatives
samps <- create_sim_samples(5, 10, 10, "good_er")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an smcurve object that contains ROC and Precision-Recall curves
smcurves <- evalmod(mdat, raw_curves = TRUE)

## Let ggplot internally call fortify
p_rocprc <- ggplot(smcurves, aes(x = x, y = y, group = dsid))
p_rocprc <- p_rocprc + geom_smooth(stat = "identity")
p_rocprc <- p_rocprc + facet_wrap(~curvetype)
p_rocprc

## Explicitly fortify smcurves
smdf <- fortify(smcurves, raw_curves = FALSE)

## Plot average ROC curve
df_roc <- subset(smdf, curvetype == "ROC")
p_roc <- ggplot(df_roc, aes(x = x, y = y, ymin = ymin, ymax = ymax))
p_roc <- p_roc + geom_smooth(stat = "identity")
p_roc

## Plot average Precision-Recall curve
df_prc <- subset(smdf, curvetype == "PRC")
p_prc <- ggplot(df_prc, aes(x = x, y = y, ymin = ymin, ymax = ymax))
p_prc <- p_prc + geom_smooth(stat = "identity")
p_prc

## Generate an smpoints object that contains basic evaluation metrics
smpoints <- evalmod(mdat, mode = "basic")

## Fortify smpoints
smdf <- fortify(smpoints)

## Plot normalized ranks vs. precision
df_prec <- subset(smdf, curvetype == "precision")
p_prec <- ggplot(df_prec, aes(x = x, y = y, ymin = ymin, ymax = ymax))
p_prec <- p_prec + geom_ribbon(aes(min = ymin, ymax = ymax),
  stat = "identity", alpha = 0.25,
  fill = "grey25"
)
```

```
p_prec <- p_prec + geom_point(aes(x = x, y = y))
p_prec


##################################################
### Multiple models & multiple test datasets
###

## Create sample datasets with 10 positives and 10 negatives
samps <- create_sim_samples(5, 10, 10, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mmcurves <- evalmod(mdat, raw_curves = TRUE)

## Let ggplot internally call fortify
p_rocprc <- ggplot(mmcurves, aes(x = x, y = y, group = dsid))
p_rocprc <- p_rocprc + geom_smooth(aes(color = modname), stat = "identity")
p_rocprc <- p_rocprc + facet_wrap(~curvetype)
p_rocprc

## Explicitly fortify mmcurves
mmdf <- fortify(mmcurves, raw_curves = FALSE)

## Plot average ROC curve
df_roc <- subset(mmdf, curvetype == "ROC")
p_roc <- ggplot(df_roc, aes(x = x, y = y, ymin = ymin, ymax = ymax))
p_roc <- p_roc + geom_smooth(aes(color = modname), stat = "identity")
p_roc

## Plot average Precision-Recall curve
df_prc <- subset(mmdf, curvetype == "PRC")
p_prc <- ggplot(df_prc, aes(x = x, y = y, ymin = ymin, ymax = ymax))
p_prc <- p_prc + geom_smooth(aes(color = modname), stat = "identity")
p_prc

## Generate an mmpoints object that contains basic evaluation metrics
mmpoints <- evalmod(mdat, mode = "basic")

## Fortify mmpoints
mmdf <- fortify(mmpoints)

## Plot normalized ranks vs. precision
df_prec <- subset(mmdf, curvetype == "precision")
p_prec <- ggplot(df_prec, aes(x = x, y = y, ymin = ymin, ymax = ymax))
p_prec <- p_prec + geom_ribbon(aes(min = ymin, ymax = ymax, group = modname),
  stat = "identity", alpha = 0.25,
  fill = "grey25"
)
p_prec <- p_prec + geom_point(aes(x = x, y = y, color = modname))
```

```
p_prec

## End(Not run)
```

IB1000 *Imbalanced data with 1000 positives and 10000 negatives.*

Description

A list contains labels and scores of five different performance levels. All scores were randomly generated.

Usage

```
data(IB1000)
```

Format

A list with 8 items.

np number of positives: 1000

nn number of negatives: 10000

labels labels of observed data

random_scores scores of a random performance level

poor_er_scores scores of a poor early retrieval level

good_er_scores scores of a good early retrieval level

excel_scores scores of an excellent level

perf_scores scores of the perfect level

IB500 *Imbalanced data with 500 positives and 5000 negatives.*

Description

A list contains labels and scores of five different performance levels. All scores were randomly generated.

Usage

```
data(IB500)
```

Format

A list with 8 items.

np number of positives: 500

nn number of negatives: 5000

labels labels of observed data

random_scores scores of a random performance level

poor_er_scores scores of a poor early retrieval level

good_er_scores scores of a good early retrieval level

excel_scores scores of an excellent level

perf_scores scores of the perfect level

---

join_labels *Join observed labels of multiple test datasets into a list*

Description

`join_labels` takes observed labels and converts them to a list.

Usage

```
join_labels(..., byrow = FALSE, chklen = TRUE)
```

Arguments

`...`	Multiple datasets. They can be vectors, arrays, matrices, data frames, and lists.
`byrow`	A Boolean value to specify whether row vectors are used for matrix, data frame, and array.
`chklen`	A Boolean value to specify whether all list items must be the same lengths.

Value

The `join_labels` function returns a list that contains all combined label data.

See Also

`evalmod()` for calculation evaluation metrics. `mmdata()` for formatting input data. `join_scores()` for formatting scores with multiple datasets.

Examples

```
##################################################
### Add three numeric vectors
###
l1 <- c(1, 0, 1, 1)
l2 <- c(1, 1, 0, 0)
l3 <- c(0, 1, 0, 1)
labels1 <- join_labels(l1, l2, l3)

## Show the list structure
str(labels1)


##################################################
### Add a matrix and a numeric vector
###
a1 <- matrix(rep(c(1, 0), 4), 4, 2)
labels2 <- join_labels(a1, l3)

## Show the list structure
str(labels2)


##################################################
### Use byrow
###
a2 <- matrix(rep(c(1, 0), 4), 2, 4, byrow = TRUE)
labels3 <- join_labels(a2, l3, byrow = TRUE)

## Show the list structure
str(labels3)


##################################################
### Use chklen
###
l4 <- c(-1, 0, -1)
l5 <- c(0, -1)
labels4 <- join_labels(l4, l5, chklen = FALSE)

## Show the list structure
str(labels4)
```

`join_scores` *Join scores of multiple models into a list*

Description

The `join_scores` function takes predicted scores from multiple models and converts them to a list.

Usage

```
join_scores(..., byrow = FALSE, chklen = TRUE)
```

Arguments

...	Multiple datasets. They can be vectors, arrays, matrices, data frames, and lists.
byrow	A Boolean value to specify whether row vectors are used for matrix, data frame, and array.
chklen	A Boolean value to specify whether all list items must be the same lengths.

Value

The `join_scores` function returns a list that contains all combined score data.

See Also

`evalmod()` for calculation evaluation metrics. `mmdata()` for formatting input data. `join_labels()` for formatting labels with multiple datasets.

Examples

```
##################################################
### Add three numeric vectors
###
s1 <- c(1, 2, 3, 4)
s2 <- c(5, 6, 7, 8)
s3 <- c(2, 4, 6, 8)
scores1 <- join_scores(s1, s2, s3)

## Show the list structure
str(scores1)


##################################################
### Add a matrix and a numeric vector
###
a1 <- matrix(seq(8), 4, 2)
scores2 <- join_scores(a1, s3)

## Show the list structure
str(scores2)


##################################################
### Use byrow
###
a2 <- matrix(seq(8), 2, 4, byrow = TRUE)
scores3 <- join_scores(a2, s3, byrow = TRUE)

## Show the list structure
str(scores3)
```

```
##################################################
### Use chklen
###
s4 <- c(1, 2, 3)
s5 <- c(5, 6, 7, 8)
scores4 <- join_scores(s4, s5, chklen = FALSE)

## Show the list structure
str(scores4)
```

M2N50F5 *5-fold cross validation sample.*

Description

A data frame contains labels and scores for 5-fold test sets.

Usage

```
data(M2N50F5)
```

Format

A data frame with 4 columns.

score1 50 random scores

score2 50 random scores

label 50 labels as 'pos' or 'neg'

fold 50 fold IDs as 1:5

---

metric_curve *Draw one evaluation metric against another*

Description

The `metric_curve` function takes the name of a metric for the x axis and the name of a metric for the y axis and calculates one curve per test dataset, in the manner of `ROCR::performance`. Every metric `evalmod()` can calculate is available on both axes.

Usage

```
metric_curve(
  mdat,
  scores = NULL,
  labels = NULL,
  x_metric = "fpr",
  y_metric = "sensitivity",
  modnames = NULL,
  dsids = NULL,
  posclass = NULL,
  na_worst = TRUE,
  ties_method = "equiv",
  x_bins = 1000,
  interpolate = TRUE,
  cost_fp = 1,
  cost_fn = 1,
  ...
)
```

Arguments

mdat	An S3 object created by the mmdata() function. It contains formatted scores and labels. The metric_curve function ignores scores and labels when mdat is specified. These arguments are internally passed to the mmdata() function when mdat is unspecified. In that case, both scores and labels must be at least specified.
scores	A numeric dataset of predicted scores. It can be a vector, a matrix, an array, a data frame, or a list.
labels	A numeric, character, logical, or factor dataset of observed labels. It can be a vector, a matrix, an array, a data frame, or a list.
x_metric	A string that specifies the metric of the x axis. It accepts every name evalmod() accepts for metrics, together with the identifiers ROCR uses. The default "fpr" with the default y_metric reproduces ROCR's most common call.
y_metric	A string that specifies the metric of the y axis.
modnames	A character vector for the names of the models.
dsids	A numeric vector for the dataset IDs.
posclass	A string or a numeric value to specify the label of positives.
na_worst	A Boolean value for the ties method of NAs.
ties_method	A string for the ties method.
x_bins	An integer for the number of supporting points of a registered pair, 1e6 or smaller. It is ignored for every other pair, which has no interpolation to place supporting points on.
interpolate	A Boolean value to specify whether or not interpolation of a registered pair is performed.

cost_fp	A numeric value for the cost of a false positive, used when one of the two axes is the cost metric. See evalmod().
cost_fn	A numeric value for the cost of a false negative.
...	These additional arguments are passed to mmdata() for data preparation.

Value

The metric_curve function returns an S3 object of one of the following classes, chosen the way evalmod() chooses between its own: ssxycurves, msxycurves, smxycurves and mmxycurves. The object holds one curve per test dataset, and works with print, as.data.frame, fortify, plot and autoplot.

Which pairs are joined by a line

precrec exists because the points of a precision-recall curve must not be joined by straight lines. The metrics this function reads are raw per-cutoff values with no interpolation, so joining an arbitrary pair of them would be the very error the package was written to avoid.

Two pairs have a defined interpolation, and only those two are drawn as curves: x_metric = "fpr" with y_metric = "sensitivity", which is the ROC curve, and x_metric = "sensitivity" with y_metric = "precision", which is the precision-recall curve. For those, metric_curve hands the work to the same code evalmod(mode = "rocprc") uses, so the two cannot disagree.

Every other pair is drawn as points. Pass type = "l" to plot or autoplot to join them anyway, having decided that the straight lines mean something for the pair at hand.

What this function does not do

metric_curve draws one curve per test dataset and does not average over them. An average needs a rule for interpolating between the points of each curve, which is exactly what an unregistered pair does not have. Use evalmod(calc_avg = TRUE) for averaged ROC and precision-recall curves.

See Also

evalmod() for the metrics themselves and for averaged ROC and precision-recall curves. mmdata() for formatting input data. autoplot() and plot() for the plots.

Examples

```
##################################################
### The ROC curve, the way ROCR asks for it
###
samps <- create_sim_samples(1, 50, 50, "good_er")
xy1 <- metric_curve(
  scores = samps[["scores"]], labels = samps[["labels"]],
  x_metric = "fpr", y_metric = "sensitivity"
)
xy1

##################################################
### A pair with no interpolation, drawn as points
```

```
###
xy2 <- metric_curve(
  scores = samps[["scores"]], labels = samps[["labels"]],
  x_metric = "predicted_positive_rate", y_metric = "lift"
)
xy2

##################################################
### Multiple models and multiple test datasets
###
samps2 <- create_sim_samples(3, 50, 50, c("poor_er", "good_er"))
mdat <- mmdata(samps2[["scores"]], samps2[["labels"]],
  modnames = samps2[["modnames"]], dsids = samps2[["dsids"]]
)
xy3 <- metric_curve(mdat, x_metric = "score", y_metric = "precision")
xy3
```

`metric_table` *Get every evaluation metric at every cutoff*

Description

The `metric_table` function returns one row per cutoff and one column per metric, in the manner of `pROC::coords` and the cutoff slots of a `ROCR::performance` object. It is the table counterpart of `metric_curve()`, which projects one metric against another.

Usage

```
metric_table(x, scores = NULL, labels = NULL, metrics = NULL, at = NULL, ...)
```

Arguments

`x`	An `S3` object created by the `mmdata()` function, or a basic-metric object created by `evalmod(mode = "basic")`. The `metric_table` function ignores `scores` and `labels` when `x` is specified. These arguments are internally passed to the `mmdata()` function when `x` is unspecified. In that case, both `scores` and `labels` must be at least specified.
`scores`	A numeric dataset of predicted scores. It can be a vector, a matrix, an array, a data frame, or a list.
`labels`	A numeric, character, logical, or factor dataset of observed labels. It can be a vector, a matrix, an array, a data frame, or a list.
`metrics`	A character vector of the metrics to calculate in addition to the default set, or the string `"all"` for every metric `precrec` knows, exactly as `evalmod()` takes it. It must be unspecified when `x` is already a basic-metric object, which carries the metrics it was built with.

at	A numeric vector of thresholds to report, instead of every cutoff. Each one is looked up rather than recalculated: `score >=` the threshold calls some number of instances positive, and that count is a rank the table already has a row for. Use it to read the metrics of a threshold chosen somewhere else - on other data, from a requirement, or by hand.
...	These additional arguments are passed to `evalmod()`, and through it to `mmdata()`, when this function builds the basic metrics itself. `beta`, `cost_fp`, `cost_fn`, `basic_ties`, `modnames`, `dsids` and `posclass` are the useful ones here.

Value

The `metric_table` function returns a data frame with one row per cutoff per model per test dataset - or, when `at` is given, one row per threshold per model per test dataset - and the following columns.

modname	Model name
dsid	Test dataset ID
at	The threshold asked for, when `at` is given
rank	Number of instances called positive, `0` to `n`
normalized_rank	`rank / n`, the x axis of the basic metric plots
score	The cutoff itself, see below
label	`1` if the instance at this rank is positive, `-1` if it is negative
...	One column per metric, named as `evalmod()` names them

What a row means

Row `rank = k` is the cutoff that calls the top `k` instances positive, so `score` is the score of the instance at rank `k` and the rule the row stands for is `score >= that value`. The metrics on the row are the metrics of that rule.

The first row is `rank = 0` - call nothing positive. There is no instance at that rank, so `score` and `label` are `NA`, and it is kept because it is a real operating point.

The two end rows are missing one cell each. Precision is `TP / (TP + FP)`, so the row that calls nothing positive has no denominator for it, and NPV is `TN / (TN + FN)`, so the row that calls everything positive has none for that. Both are `NA`, together with the metrics derived from them - the false discovery rate is `1 - precision`, the false omission rate is `1 - NPV`, and `markedness` is `precision + NPV - 1`, so it is missing at both ends. Every other metric on those rows is measured.

The long form `as.data.frame(evalmod(mode = "basic"))` returns carries a value in those cells instead, taken from the neighboring row. That is what anchors the precision-recall curve at recall `0`, and it is the right value for a curve - but it is not a measurement of the rule the row stands for, and a `precision` of `1` beside a `sensitivity` of `0` reads as a perfect threshold. Plot from the long form; decide from this table.

Tied scores share one value of each metric rather than taking a value that depends on the order the ties arrived in. `basic_ties` of `evalmod()` controls that.

Reading a threshold you already have

`at` answers the question the rest of the table cannot: what are the metrics of *this* threshold. A threshold is rarely one of the observed scores, so it has no row of its own, but it always names one

of the cutoffs - `score >=` it calls a certain number of instances positive, and that count is a rank. The row returned is that rank's row, so `at` is what was asked for and `score` is the observed cutoff realizing it, the smallest score still called positive.

A threshold above every score gives the `rank = 0` row, where `score` is `NA` because nothing is called positive, and `precision` is `NA` because a rule that makes no positive predictions has none. One below every score gives the `rank = n` row, with no NPV for the same reason at the other end. Neither is an edge case here: a threshold chosen on one dataset lands outside the range of another routinely, which is most of why it is worth asking what it does there. A score of `NA` is never a positive prediction, matching the `na_worst = TRUE` default of the rest of the package.

Ranks are per test dataset, so one threshold lands on a different rank in each of them. That is the reason to ask by threshold: a threshold is what transfers between datasets, and a rank is not. It is what makes a cutoff chosen on held-out data measurable on the data it was kept from, which the *Choose an operating point* article works through: https://evalclass.github.io/precrec/articles/howto-operating-point.html.

Several test datasets

A cutoff belongs to the dataset it was read off, so the rows are per dataset and nothing is averaged across them - unlike the basic metric plots, which average by default. A basic-metric object passed as `x` therefore has to have been built with `raw_curves = TRUE` when it holds more than one dataset, since an object built without it keeps only the average.

See Also

`evalmod()` for calculating the metrics, `metric_curve()` for projecting one metric against another, and `as.data.frame()` for the long form the plots use.

Examples

```
##################################################
### Single model & single test dataset
###

## Load a dataset with 10 positives and 10 negatives
data(P10N10)

## Get every metric at every cutoff
tab <- metric_table(scores = P10N10$scores, labels = P10N10$labels)
head(tab)

## The cutoff that maximizes the F-score
tab[which.max(tab$fscore), c("rank", "score", "precision", "fscore")]


##################################################
### Add metrics beyond the default set
###

lifted <- metric_table(
  scores = P10N10$scores, labels = P10N10$labels,
```

```
  metrics = c("lift", "jaccard")
)
head(lifted[, c("rank", "score", "precision", "lift", "jaccard")])


##################################################
### Read the metrics of thresholds you already have
###

## 14 is a run of six tied scores, so the rank is the end of the run
metric_table(
  scores = P10N10$scores, labels = P10N10$labels,
  at = c(17, 14, 6)
)[, c("at", "rank", "score", "sensitivity", "precision")]


###################################################
### Reuse an object that is already calculated
###

points <- evalmod(
  scores = P10N10$scores, labels = P10N10$labels,
  mode = "basic"
)
head(metric_table(points))
```

---

mmdata *Reformat input data for performance evaluation calculation*

Description

The mmdata function takes predicted scores and labels and returns an mdat object. The evalmod() function takes an mdat object as input data to calculate evaluation metrics.

Usage

```
mmdata(
  scores,
  labels,
  modnames = NULL,
  dsids = NULL,
  posclass = NULL,
  na_worst = TRUE,
  ties_method = "equiv",
  expd_first = NULL,
  mode = "rocprc",
  multiclass = NULL,
  nfold_df = NULL,
```

```
  score_cols = NULL,
  lab_col = NULL,
  fold_col = NULL,
  ...
)
```

Arguments

`scores` A numeric dataset of predicted scores. It can be a vector, a matrix, an array, a data frame, or a list. The `join_scores()` function can be useful to make scores with multiple datasets.

`labels` A numeric, character, logical, or factor dataset of observed labels. It can be a vector, a matrix, an array, a data frame, or a list. The `join_labels()` function can be useful to make labels with multiple datasets.

`modnames` A character vector for the names of the models. The `evalmod` function automatically generates default names as "m1", "m2", "m3", and so on when it is `NULL`.

`dsids` A numeric vector for test dataset IDs. The `evalmod` function automatically generates the default ID as `1` when it is `NULL`.

`posclass` A scalar value to specify the label of positives in `labels`. It must be the same data type as `labels`. For example, `posclass = -1` changes the positive label from `1` to `-1` when `labels` contains `1` and `-1`. The positive label will be automatically detected when `posclass` is `NULL`.

`na_worst` A Boolean value for controlling the treatment of NAs in `scores`.

TRUE All NAs are treated as the worst scores

FALSE All NAs are treated as the best scores

`ties_method` A string for controlling ties in `scores`.

"equiv" Ties are equivalently ranked

"first" Ties are ranked in an increasing order as appeared

"random" Ties are ranked in random order

`expd_first` A string to indicate which of the two variables - model names or test dataset IDs should be expanded first when they are automatically generated.

"modnames" Model names are expanded first. For example, The `mmdata` function generates `modnames` as `c("m1", "m2")` and `dsids` as `c(1, 1)` when two vectors are passed as input, and `modnames` and `dsids` are unspecified.

"dsids" Test dataset IDs are expanded first. For example, The `mmdata` function generates `modnames` as `c("m1", "m1")` and `dsids` as `c(1, 2)` when two vectors are passed as input, and `modnames` and `dsids` are unspecified.

`mode` A string that specifies the types of evaluation metrics that the `evalmod` function calculates.

"rocprc" ROC and Precision-Recall curves

"prcroc" Same as above

"basic" Normalized ranks vs. accuracy, error rate, specificity, sensitivity, precision, Matthews correlation coefficient, and F-score.

"aucroc" Fast AUC(ROC) calculation with the U statistic

`multiclass` A string that specifies how a dataset with more than two classes is evaluated.

"none" Binary evaluation. Labels with more than two classes are an error, as they have always been.

"ovr" One-vs-rest. Each class becomes its own binary problem - that class against all the others - and the `K` decompositions are carried on the model axis, so that they behave like `K` models evaluated on one dataset.

`multiclass` is detected from the input when it is `NULL`: "ovr" when `labels` hold more than two classes and `scores` is a matrix or a data frame with one column per class, "none" otherwise. `scores` columns are matched to classes by their column names when those name the classes, and by position otherwise.

`posclass` is ignored in one-vs-rest mode, and `multiclass` cannot be combined with `nfold_df`.

Note that each one-vs-rest decomposition has its own class balance, so the baseline of a precision-recall curve differs from class to class. The plots leave the baseline out for that reason.

`nfold_df` A data frame that contains at least one score column, label and fold columns.

`score_cols` A character/numeric vector that specifies score columns of `nfold_df`.

`lab_col` A number/string that specifies the label column of `nfold_df`.

`fold_col` A number/string that specifies the fold column of `nfold_df`.

`...` Not used by this method.

Value

The `mmdata` function returns an `mdat` object that contains formatted labels and score ranks. The object can be used as input data for the `evalmod()` function.

See Also

`evalmod()` for calculation evaluation metrics. `join_scores()` and `join_labels()` for formatting scores and labels with multiple datasets. `format_nfold()` for creating n-fold cross validation dataset from data frame.

Examples

```
##################################################
### Single model & single test dataset
###

## Load a dataset with 10 positives and 10 negatives
data(P10N10)

## Generate mdat object
ssmdat1 <- mmdata(P10N10$scores, P10N10$labels)
ssmdat1
ssmdat2 <- mmdata(1:8, sample(c(0, 1), 8, replace = TRUE))
ssmdat2
```

```
##################################################
### Multiple models & single test dataset
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(1, 100, 100, "all")

## Multiple models & single test dataset
msmdat1 <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]]
)
msmdat1

## Use join_scores and join_labels
s1 <- c(1, 2, 3, 4)
s2 <- c(5, 6, 7, 8)
scores <- join_scores(s1, s2)

l1 <- c(1, 0, 1, 1)
l2 <- c(1, 0, 1, 1)
labels <- join_labels(l1, l2)

msmdat2 <- mmdata(scores, labels, modnames = c("ms1", "ms2"))
msmdat2


##################################################
### Single model & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(10, 100, 100, "good_er")

## Single model & multiple test datasets
smmdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)
smmdat


##################################################
### Multiple models & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(10, 100, 100, "all")

## Multiple models & multiple test datasets
mmmdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
```

```
  dsids = samps[["dsids"]]
)
mmmdat


##################################################
### N-fold cross validation datasets
###

## Load test data
data(M2N50F5)
head(M2N50F5)

## Speficy nessesary columns to create mdat
cvdat1 <- mmdata(
  nfold_df = M2N50F5, score_cols = c(1, 2),
  lab_col = 3, fold_col = 4,
  modnames = c("m1", "m2"), dsids = 1:5
)
cvdat1

## Use column names
cvdat2 <- mmdata(
  nfold_df = M2N50F5, score_cols = c("score1", "score2"),
  lab_col = "label", fold_col = "fold",
  modnames = c("m1", "m2"), dsids = 1:5
)
cvdat2


##################################################
### Multiclass dataset
###

## Load a 3-class dataset with one score column per class
data(C3N150)

## One-vs-rest decomposition, detected from the input
mcmdat <- mmdata(C3N150$scores, C3N150$labels)
mcmdat
```

P10N10 *A small example dataset with several tied scores.*

Description

A list contains labels and scores for 10 positives and 10 negatives.

Usage

```
data(P10N10)
```

Format

A list with 4 items.

np number of positives: 10

nn number of negatives: 10

labels 20 labels of observed data

scores 20 scores with some ties

---

part — *Calculate partial AUCs*

Description

The `part` function takes an `S3` object generated by `evalmod()` and calculate partial AUCs and Standardized partial AUCs of ROC and Precision-Recall curves. Standardized pAUCs are standardized to the range between 0 and 1.

Usage

```
part(curves, xlim = NULL, ylim = NULL, curvetype = NULL)

## S3 method for class 'sscurves'
part(curves, xlim = c(0, 1), ylim = c(0, 1), curvetype = c("ROC", "PRC"))

## S3 method for class 'mscurves'
part(curves, xlim = c(0, 1), ylim = c(0, 1), curvetype = c("ROC", "PRC"))

## S3 method for class 'smcurves'
part(curves, xlim = c(0, 1), ylim = c(0, 1), curvetype = c("ROC", "PRC"))

## S3 method for class 'mmcurves'
part(curves, xlim = c(0, 1), ylim = c(0, 1), curvetype = c("ROC", "PRC"))
```

Arguments

`curves` An `S3` object generated by `evalmod()`. The `part` function accepts the following S3 objects.

`S3` object	# of models	# of test datasets
sscurves	single	single
mscurves	multiple	single
smcurves	single	multiple

mmcurves	multiple	multiple

See the **Value** section of `evalmod()` for more details.

`xlim` A numeric vector of length two to specify x range between two points in [0, 1]

`ylim` A numeric vector of length two to specify y range between two points in [0, 1]

`curvetype` A character vector with the following curve types.

curvetype	description
ROC	ROC curve
PRC	Precision-Recall curve

Multiple `curvetype` can be combined, such as `c("ROC", "PRC")`.

Value

The `part` function returns the same S3 object specified as input with calculated pAUCs and standardized pAUCs.

See Also

`evalmod()` for generating `S3` objects with performance evaluation metrics. `pauc()` for retrieving a dataset of pAUCs.

Examples

```
## Not run:

## Load library
library(ggplot2)

##################################################
### Single model & single test dataset
###

## Load a dataset with 10 positives and 10 negatives
data(P10N10)

## Generate an sscurve object that contains ROC and Precision-Recall curves
sscurves <- evalmod(scores = P10N10$scores, labels = P10N10$labels)

## Calculate partial AUCs
sscurves.part <- part(sscurves, xlim = c(0.25, 0.75))

## Show AUCs
sscurves.part

## Plot partial curve
plot(sscurves.part)
```

```
## Plot partial curve with ggplot
autoplot(sscurves.part)


##################################################
### Multiple models & single test dataset
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(1, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mscurves <- evalmod(mdat)

## Calculate partial AUCs
mscurves.part <- part(mscurves, xlim = c(0, 0.75), ylim = c(0.25, 0.75))

## Show AUCs
mscurves.part

## Plot partial curves
plot(mscurves.part)

## Plot partial curves with ggplot
autoplot(mscurves.part)


#################################################
### Single model & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(4, 100, 100, "good_er")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an smcurve object that contains ROC and Precision-Recall curves
smcurves <- evalmod(mdat)

## Calculate partial AUCs
smcurves.part <- part(smcurves, xlim = c(0.25, 0.75))

## Show AUCs
smcurves.part

## Plot partial curve
plot(smcurves.part)
```

```
## Plot partial curve with ggplot
autoplot(smcurves.part)


##################################################
### Multiple models & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(4, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mmcurves <- evalmod(mdat, raw_curves = TRUE)

## Calculate partial AUCs
mmcurves.part <- part(mmcurves, xlim = c(0, 0.25))

## Show AUCs
mmcurves.part

## Plot partial curves
plot(mmcurves.part)

## Plot partial curves with ggplot
autoplot(mmcurves.part)

## End(Not run)
```

---

pauc *Retrieve a data frame of pAUC scores*

Description

The pauc function takes an S3 object generated by part() and evalmod() and retrieves a data frame with the partial AUC scores of ROC and Precision-Recall curves.

Usage

```
pauc(curves, corrected = FALSE)

## S3 method for class 'aucs'
pauc(curves, corrected = FALSE)
```

Arguments

`curves` An `S3` object generated by `part()` and `evalmod()`. The `pauc` function accepts the following S3 objects.

`S3` object	# of models	# of test datasets
sscurves	single	single
mscurves	multiple	single
smcurves	single	multiple
mmcurves	multiple	multiple

See the **Value** section of `evalmod()` for more details.

`corrected` A logical value to add the `cpaucs` column, the McClish correction of the ROC partial areas. The default is `FALSE`.

Value

The `pauc` function returns a data frame with one row per curve per model per test dataset, and the following columns. An object that holds averaged curves only has no `dsids` column, because an averaged curve has no single test dataset behind it.

`modnames`	Model name
`dsids`	Test dataset ID
`curvetypes`	`ROC` or `PRC`
`paucs`	The area under that curve over the region
`baselines`	What that area would be by chance, see below
`spaucs`	That area over the area of the region
`sbaselines`	What `spaucs` would be by chance
`cpaucs`	The McClish correction, with `corrected = TRUE`

Each area is followed by what it is worth by chance, because neither of them can be read without it. `cpaucs` has no column of its own: the correction puts chance at `0.5` by construction, which is the whole point of it.

What a partial area is worth by chance

Restricting the region changes the chance level, and not by an amount anyone guesses correctly. Over false positive rates `[x1, x2]` a coin flip covers the area under the diagonal across that span, `(x2^2 - x1^2) / 2`. A precision-recall curve is flat at the proportion of positives instead, so chance covers `prevalence * (x2 - x1)`.

`spaucs` divides by the area of the region, which divides those two by the width: chance for a standardized ROC partial area is `(x1 + x2) / 2` and **not** `0.5`. Over false positive rates up to `0.2` a coin flip covers `0.1` of the region, so a `spaucs` of `0.3` there is three times chance and not a failing grade. For a precision-recall curve the standardized chance level is the prevalence, unchanged by the region.

Both are NA when part() was given a ylim other than c(0, 1). The region is then a box the chance curve may cross, touch or miss entirely, and the area of chance inside it has no settled definition - the case cpaucs declines for the same reason.

A baseline is what an area is worth by chance in the limit, and the estimate from a finite sample scatters around it. Comparing an area to its baseline is not a test, and the smaller the region and the fewer the positives the less it is worth: over the first tenth of recall with twenty positives, a classifier with no signal averages nearly twice its baseline. auc_boot() puts an interval around the area, which is the comparison that means something.

Two ways to standardize a partial area

A partial area is not comparable across regions on its own - a wider region holds more area - so it is reported standardized, and there are two conventions for doing it. Both are called "the standardized partial AUC", and a number carried between tools looks like a disagreement when it is a choice of convention.

spaucs divides the area by the area of the region, so it says what fraction of what was available the curve covered. The McClish correction instead rescales the span between chance and perfect onto 0.5 to 1, so that a partial area reads on the same scale as a full one. Over false positive rates up to 0.2, a coin flip covers an area of 0.02: that is 0.1 of the region, and 0.5 after the correction.

Neither is the right one. spaucs is the more direct reading of how much of the corner the curve filled; the corrected one is the more comparable to a full AUC. Say which you used.

cpaucs is what pROC::auc(..., partial.auc.correct = TRUE) returns, to the last digit. The chance area is taken over the region actually asked for, (x2^2 - x1^2) / 2, and not over a region assumed to start at a false positive rate of 0 - the formula usually quoted is the special case of that one, and gets a region such as xlim = c(0.1, 0.3) wrong.

The correction is the identity over the whole curve, where chance is 0.5 and the region is 1, so a cpaucs from part(xlim = c(0, 1)) is the plain AUC. At the other end it is unstable: over a narrow region of high false positive rates, chance and perfect are close together and the rescaling divides by the small difference, so a curve a little below the diagonal there can produce a large negative number. That is the correction behaving as defined, not an error.

cpaucs is NA on every PRC row, and on every row when part() was given a ylim other than c(0, 1). A precision-recall curve's chance level is the proportion of positives rather than the diagonal, and rescaling by it would make curves at different class balances look comparable when the point of the precision-recall plot is that they are not - see auc(). A restricted ylim has no counterpart in pROC and no settled definition, so it reports nothing rather than invent one.

See Also

evalmod() for generating S3 objects with performance evaluation metrics. part() for calculation of pAUCs. auc() for retrieving a dataset of AUCs, and for the baseline a full area is read against.

Examples

```
##################################################
### Single model & single test dataset
###

## Load a dataset with 10 positives and 10 negatives
```

```
data(P10N10)

## Generate an sscurve object that contains ROC and Precision-Recall curves
sscurves <- evalmod(scores = P10N10$scores, labels = P10N10$labels)

## Calculate partial AUCs
sscurves.part <- part(sscurves, xlim = c(0.25, 0.75))

## Shows pAUCs
pauc(sscurves.part)

##################################################
### Multiple models & single test dataset
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(1, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mscurves <- evalmod(mdat)

## Calculate partial AUCs
mscurves.part <- part(mscurves, xlim = c(0, 0.75), ylim = c(0.25, 0.75))

## Shows pAUCs
pauc(mscurves.part)

##################################################
### Single model & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(4, 100, 100, "good_er")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an smcurve object that contains ROC and Precision-Recall curves
smcurves <- evalmod(mdat, raw_curves = TRUE)

## Calculate partial AUCs
smcurves.part <- part(smcurves, xlim = c(0.25, 0.75))

## Shows pAUCs
pauc(smcurves.part)

##################################################
### Multiple models & multiple test datasets
###
```

```
## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(4, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mmcurves <- evalmod(mdat, raw_curves = TRUE)

## Calculate partial AUCs
mmcurves.part <- part(mmcurves, xlim = c(0, 0.25))

## Shows pAUCs
pauc(mmcurves.part)

##################################################
### The McClish correction
###

## The pAUC rescaled between chance and perfect, as pROC reports it
pauc(sscurves.part, corrected = TRUE)
```

---

plot *Plot performance evaluation metrics*

Description

The `plot` function creates a plot of performance evaluation metrics.

Usage

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

## S3 method for class 'mscurves'
plot(x, y = NULL, ...)

## S3 method for class 'smcurves'
plot(x, y = NULL, ...)

## S3 method for class 'mmcurves'
plot(x, y = NULL, ...)

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

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

## S3 method for class 'smpoints'
plot(x, y = NULL, ...)

## S3 method for class 'mmpoints'
plot(x, y = NULL, ...)

## S3 method for class 'ssxycurves'
plot(x, y = NULL, ...)

## S3 method for class 'msxycurves'
plot(x, y = NULL, ...)

## S3 method for class 'smxycurves'
plot(x, y = NULL, ...)

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

Arguments

`x` An `S3` object generated by `evalmod()`. The `plot` function accepts the following `S3` objects.

1. ROC and Precision-Recall curves (mode = "rocprc")

`S3` object	# of models	# of test datasets
sscurves	single	single
mscurves	multiple	single
smcurves	single	multiple
mmcurves	multiple	multiple

2. Basic evaluation metrics (mode = "basic")

`S3` object	# of models	# of test datasets
sspoints	single	single
mspoints	multiple	single
smpoints	single	multiple
mmpoints	multiple	multiple

See the **Value** section of `evalmod()` for more details.

`y` Equivalent with `curvetype`.

`...` All the following arguments can be specified.

curvetype 1. ROC and Precision-Recall curves (mode = "rocprc")

curvetype	description
ROC	ROC curve
PRC	Precision-Recall curve

Multiple `curvetype` can be combined, such as `c("ROC", "PRC")`.

2. Basic evaluation metrics (mode = "basic")

curvetype	description
error	Normalized ranks vs. error rate
accuracy	Normalized ranks vs. accuracy
specificity	Normalized ranks vs. specificity
sensitivity	Normalized ranks vs. sensitivity
precision	Normalized ranks vs. precision
mcc	Normalized ranks vs. Matthews correlation coefficient
fscore	Normalized ranks vs. F-score
balanced_accuracy	Normalized ranks vs. balanced accuracy
npv	Normalized ranks vs. negative predictive value
informedness	Normalized ranks vs. informedness (Youden's J)
markedness	Normalized ranks vs. markedness
kappa	Normalized ranks vs. Cohen's kappa

Multiple `curvetype` can be combined, such as `c("precision", "sensitivity")`.

type A character to specify the line type as follows.

- **"l"** lines
- **"p"** points
- **"b"** both lines and points

show_cb A Boolean value to specify whether point-wise confidence bounds are drawn. It is effective only when `calc_avg` of the `evalmod()` function is set to `TRUE`.

raw_curves A Boolean value to specify whether raw curves are shown instead of the average curve. It is effective only when `raw_curves` of the `evalmod()` function is set to `TRUE`.

show_legend A Boolean value to specify whether the legend is shown.

Value

The `plot` function shows a plot and returns NULL.

See Also

`evalmod()` for generating an `S3` object. `autoplot()` for plotting the equivalent curves with **ggplot2**.

Examples

```
## Not run:
##################################################
```

```
### Single model & single test dataset
###

## Load a dataset with 10 positives and 10 negatives
data(P10N10)

## Generate an sscurve object that contains ROC and Precision-Recall curves
sscurves <- evalmod(scores = P10N10$scores, labels = P10N10$labels)

## Plot both ROC and Precision-Recall curves
plot(sscurves)

## Plot a ROC curve
plot(sscurves, curvetype = "ROC")

## Plot a Precision-Recall curve
plot(sscurves, curvetype = "PRC")

## Generate an sspoints object that contains basic evaluation metrics
sspoints <- evalmod(
  mode = "basic", scores = P10N10$scores,
  labels = P10N10$labels
)

## Plot normalized ranks vs. basic evaluation metrics
plot(sspoints)

## Plot normalized ranks vs. precision
plot(sspoints, curvetype = "precision")


##################################################
### Multiple models & single test dataset
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(1, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mscurves <- evalmod(mdat)

## Plot both ROC and Precision-Recall curves
plot(mscurves)

## Hide the legend
plot(mscurves, show_legend = FALSE)

## Generate an mspoints object that contains basic evaluation metrics
mspoints <- evalmod(mdat, mode = "basic")
```

```
## Plot normalized ranks vs. basic evaluation metrics
plot(mspoints)

## Hide the legend
plot(mspoints, show_legend = FALSE)


##################################################
### Single model & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(10, 100, 100, "good_er")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an smcurve object that contains ROC and Precision-Recall curves
smcurves <- evalmod(mdat, raw_curves = TRUE)

## Plot average ROC and Precision-Recall curves
plot(smcurves, raw_curves = FALSE)

## Hide confidence bounds
plot(smcurves, raw_curves = FALSE, show_cb = FALSE)

## Plot raw ROC and Precision-Recall curves
plot(smcurves, raw_curves = TRUE, show_cb = FALSE)

## Generate an smpoints object that contains basic evaluation metrics
smpoints <- evalmod(mdat, mode = "basic")

## Plot normalized ranks vs. average basic evaluation metrics
plot(smpoints)


###################################################
### Multiple models & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(10, 100, 100, "all")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Generate an mscurve object that contains ROC and Precision-Recall curves
mmcurves <- evalmod(mdat, raw_curves = TRUE)

## Plot average ROC and Precision-Recall curves
plot(mmcurves, raw_curves = FALSE)
```

```
## Show confidence bounds
plot(mmcurves, raw_curves = FALSE, show_cb = TRUE)

## Plot raw ROC and Precision-Recall curves
plot(mmcurves, raw_curves = TRUE)

## Generate an mmpoints object that contains basic evaluation metrics
mmpoints <- evalmod(mdat, mode = "basic")

## Plot normalized ranks vs. average basic evaluation metrics
plot(mmpoints)


##################################################
### N-fold cross validation datasets
###

## Load test data
data(M2N50F5)

## Speficy nessesary columns to create mdat
cvdat <- mmdata(
  nfold_df = M2N50F5, score_cols = c(1, 2),
  lab_col = 3, fold_col = 4,
  modnames = c("m1", "m2"), dsids = 1:5
)

## Generate an mmcurve object that contains ROC and Precision-Recall curves
cvcurves <- evalmod(cvdat)

## Average ROC and Precision-Recall curves
plot(cvcurves)

## Show confidence bounds
plot(cvcurves, show_cb = TRUE)

## Generate an mmpoints object that contains basic evaluation metrics
cvpoints <- evalmod(cvdat, mode = "basic")

## Normalized ranks vs. average basic evaluation metrics
plot(cvpoints)

## End(Not run)
```

---

prbe *Calculate the precision-recall break-even point*

Description

The prbe function finds the points of a precision-recall curve at which precision and recall are equal. It takes an S3 object generated by evalmod() and returns a data frame with one row per break-even point.

Usage

```
prbe(curves)

## S3 method for class 'curve_info'
prbe(curves)
```

Arguments

curves	An S3 object generated by evalmod().

Value

The prbe function returns a data frame with the columns modnames, dsids, prbe and baselines. prbe is the common value of precision and recall at the break-even point.

A curve can cross the diagonal more than once, and then the data frame holds one row per crossing, ordered by recall. A curve that never reaches equal precision and recall gets a single row of NA, which at a low proportion of positives is a common and correct answer: such a curve can leave the origin below the diagonal and never catch up.

baselines is the proportion of positives, which is where a classifier that ranks at random breaks even - at chance the curve is flat at the prevalence, so it meets the diagonal at that recall. A break-even point of 0.2 is chance on data that is 20% positive and five times chance on data that is 4% positive. See Reading an area against its baseline in auc().

The origin is not reported as a break-even point. A curve is anchored at recall 0, where precision is 1 if the top-ranked instance is a positive and 0 if it is a negative; in the second case precision and recall are equal there, but nothing has been retrieved and the equality is an artifact of where the curve starts rather than a point at which the classifier balances the two.

How this differs from ROCR

ROCR::performance(pred, "prbe") interpolates linearly between adjacent raw precision-recall points to find the crossing. Linear interpolation between precision-recall points is not correct, which is the reason this package exists, so prbe reads the crossing off the curve evalmod() has already interpolated properly. The two agree wherever the crossing falls on a point both of them hold, and differ where it falls between two of them.

See Also

evalmod() for generating S3 objects with performance evaluation metrics. auc() for the areas under the curves.

Examples

```
##################################################
### Single model & single test dataset
###
samps <- create_sim_samples(1, 50, 50, "good_er")
sscurves <- evalmod(scores = samps[["scores"]], labels = samps[["labels"]])
prbe(sscurves)

##################################################
### Multiple models & multiple test datasets
###
samps2 <- create_sim_samples(2, 50, 50, c("poor_er", "good_er"))
mmdat <- mmdata(samps2[["scores"]], samps2[["labels"]],
  modnames = samps2[["modnames"]], dsids = samps2[["dsids"]]
)
mmcurves <- evalmod(mmdat, raw_curves = TRUE)
prbe(mmcurves)
```

---

precrec — *precrec: A package for computing accurate ROC and Precision-Recall curves*

Description

The precrec package contains several functions and S3 generics to provide a robust platform for performance evaluation of binary classifiers.

Functions

The precrec package provides the following six functions.

Function	Description
evalmod()	Main function to calculate evaluation metrics
mmdata()	Reformat input data for performance evaluation calculation
join_scores()	Join scores of multiple models into a list
join_labels()	Join observed labels of multiple test datasets into a list
create_sim_samples()	Create random samples for simulations
format_nfold()	Create n-fold cross validation dataset from data frame

S3 generics

The precrec package provides nine different S3 generics for the S3 objects generated by the evalmod() function.

S3 generic	Library	Description
print	base	Print the calculation results and the summary of the test data

`as.data.frame()`	base	Convert a precrec object to a data frame
`plot()`	graphics	Plot performance evaluation metrics
`autoplot()`	ggplot2	Plot performance evaluation metrics with ggplot2
`fortify()`	ggplot2	Prepare a data frame for ggplot2
`auc()`	precrec	Make a data frame with AUC scores
`part()`	precrec	Calculate partial curves and partial AUC scores
`pauc()`	precrec	Make a data frame with pAUC scores
`auc_ci()`	precrec	Calculate confidence intervals of AUC scores

Performance metric calculations

The `evalmod()` function calculates ROC and Precision-Recall curves and returns an `S3` object. The generated `S3` object can be used with several different `S3` generics, such as `print` and `plot()`. The `evalmod()` function can also calculate basic evaluation metrics - error rate, accuracy, specificity, sensitivity, precision, Matthews correlation coefficient, F-score, balanced accuracy, negative predictive value, informedness, markedness, and Cohen's kappa.

Data preparation

The `mmdata()` function creates an input dataset for the `evalmod()` function. The generated dataset contains formatted scores and labels.

`join_scores()` and `join_labels()` are helper functions to combine multiple scores and labels.

The `create_sim_samples()` function creates test datasets with five different performance levels.

Data visualization

`plot()` takes an `S3` object generated by `evalmod()` as input and plot corresponding curves.

`autoplot()` uses `ggplot` to plot curves.

Result retrieval

`as.data.frame()` takes an `S3` object generated by `evalmod()` as input and and returns a data frame with calculated curve points.

`auc()` and `pauc()` returns a data frame with AUC scores and partial AUC scores, respectively. `auc_ci()` returns confidence intervals of AUCs for both ROC and precision-recall curves.

`average_precision()` returns the step estimator of the area under the precision-recall curve, and `prbe()` the precision-recall break-even point.

Probability-based metrics

`prob_metrics()` calculates the Brier score and the log loss of prediction scores that are probabilities, and `prob_metrics_ci()` returns their confidence intervals over multiple test datasets.

Uncertainty from one test set

`auc_boot()` resamples a single test set so that `auc_ci()` can put a percentile interval around its AUC, and `auc_diff()` can compare two models on the same resamples.

Classification report

classification_report() reports precision, recall and the F-score of every class at one operating point, in the layout scikit-learn's function of that name prints.

Author(s)

Maintainer: Takaya Saito <takaya.saito@outlook.com> (ORCID)

Authors:

- Takaya Saito <takaya.saito@outlook.com> (ORCID)
- Marc Rehmsmeier <marc.rehmsmeier@ii.uib.no> (ORCID)

See Also

Useful links:

- https://github.com/evalclass/precrec
- https://evalclass.github.io/precrec/
- Report bugs at https://github.com/evalclass/precrec/issues

---

print — *Print the summary of a precrec object*

Description

The print function prints a summary of an S3 object created by mmdata(), evalmod(), metric_curve() or classification_report(). It is called for its side effect, and is what the console shows when one of those objects is evaluated at the prompt.

Usage

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

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

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

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

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

## S3 method for class 'classification_report'
print(x, digits = 2, ...)
```

Arguments

x — An S3 object created by mmdata(), evalmod(), metric_curve() or classification_report(). The print function takes one of the following S3 objects.

S3 object	Created by
mdat	mmdata()
curve_info	evalmod()
beval_info	evalmod(mode = "basic")
aucroc	evalmod(mode = "aucroc")
xycurve_info	metric_curve()
classification_report	classification_report()

Every object but a classification_report includes a summary of the input data - the model names, the dataset IDs and the class counts. Alongside it, a curve object reports its AUCs beside the baseline each one is read against - 0.5 for a ROC curve and the proportion of positives for a precision-recall curve, see auc() - and its partial AUCs as well when it came from part(); a basic-metric object reports what each metric abbreviation means and a five-number summary of every metric; an aucroc object reports the AUCs beside the U statistics they came from; and a metric_curve() object names the metric pair and counts the points on it. A classification_report prints its own table of per-class precision, recall and F-score.

The curve and point objects carry a second class naming how many models and test datasets they hold, such as sscurves or mmpoints, but all of them print through curve_info or beval_info. See the **Value** section of evalmod().

... — Not used by these methods.

digits — The number of digits after the decimal point, between 0 and 20. Used by the classification_report method only.

Value

The print function returns x invisibly.

See Also

evalmod() and mmdata() for creating the objects, as.data.frame() for the same results as a data frame, and auc() for the AUCs alone.

Examples

```
##################################################
### Input data
###

## Load a dataset with 10 positives and 10 negatives
data(P10N10)
```

```
mdat <- mmdata(P10N10$scores, P10N10$labels)
mdat


##################################################
### ROC and Precision-Recall curves
###

curves <- evalmod(mdat)
curves

## Partial curves also report the partial AUCs
part(curves, xlim = c(0, 0.25))


##################################################
### Basic evaluation metrics
###

points <- evalmod(mdat, mode = "basic")
points


##################################################
### AUC with the U statistic
###

evalmod(mdat, mode = "aucroc")


##################################################
### One metric against another
###

metric_curve(mdat)


##################################################
### Per-class precision, recall and F-score
###

classification_report(mdat, at = 12)
```

---

prob_metrics *Calculate the Brier score, the RMSE and the log loss of predicted probabilities*

Description

The prob_metrics function calculates three probability-based evaluation metrics - the Brier score, its square root the root mean squared error, and the log loss - for prediction scores that are proba-

bilities. Unlike ROC and Precision-Recall curves, these metrics depend on the values of the scores rather than on their ranks, so the scores must lie in the range [0, 1].

Usage

```
prob_metrics(
  mdat,
  scores = NULL,
  labels = NULL,
  eps = 1e-15,
  metrics = NULL,
  ...
)
```

Arguments

`mdat`	An `S3` object created by the `mmdata()` function. It contains formatted scores and labels. The `prob_metrics` function ignores `scores` and `labels` when `mdat` is specified. These arguments are internally passed to the `mmdata()` function when `mdat` is unspecified. In that case, both `scores` and `labels` must be at least specified.
`scores`	A numeric dataset of predicted probabilities. It can be a vector, a matrix, an array, a data frame, or a list.
`labels`	A numeric, character, logical, or factor dataset of observed labels. It can be a vector, a matrix, an array, a data frame, or a list.
`eps`	A numeric value used to clamp the scores away from `0` and `1` before the log loss is calculated. A single confident and wrong prediction would otherwise make the log loss infinite.
`metrics`	A character vector of additional metrics to calculate. The three metrics above are always returned; `"d2_brier"` and `"d2_logloss"` are returned as well when they are named here, and `"all"` asks for every metric the function knows. A D2 score rescales a loss against the loss of the null model, the one that predicts the observed prevalence for every case and ignores the scores: `D2 = 1 - loss(model) / loss(null)` It is `1` for a perfect model and `0` for one that does no better than the prevalence, and it is negative for a model that does worse - which is a real result rather than an error, so it is not clipped. A dataset holding a single class has a null loss of `0` and so a D2 score of `NA`.
`...`	These additional arguments are passed to `mmdata()` for data preparation.

Value

The `prob_metrics` function returns a data frame with the columns `modnames`, `dsids`, `metrics`, and `values`. `metrics` is one of "brier", "rmse" or "logloss", so each model and dataset combination takes up three rows - plus one row for each metric named in `metrics`.

See Also

prob_metrics_ci() for the CIs of these metrics over multiple datasets. evalmod() for generating S3 objects with performance evaluation metrics. mmdata() for formatting input data.

Examples

```
##################################################
### Single model & single test dataset
###

## Predicted probabilities of 10 positives and 10 negatives
set.seed(1)
scores <- c(runif(10, 0.4, 1), runif(10, 0, 0.6))
labels <- c(rep(1, 10), rep(0, 10))

## Brier score and log loss
prob_metrics(scores = scores, labels = labels)


##################################################
### Multiple models & multiple test datasets
###

## The "poor_er" and "good_er" samples are drawn from beta distributions,
## so their scores can be read as probabilities
samps <- create_sim_samples(4, 100, 100, c("poor_er", "good_er"))
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

pm <- prob_metrics(mdat)

## Show the Brier scores only
subset(pm, metrics == "brier")

## The D2 scores, which say how much of each loss the model explains
subset(
  prob_metrics(mdat, metrics = c("d2_brier", "d2_logloss")),
  dsids == 1
)
```

---

prob_metrics_ci *Calculate CIs of the Brier score and the log loss*

Description

The prob_metrics_ci function calculates the confidence intervals of the Brier score and the log loss when multiple test datasets are specified.

Usage

```
prob_metrics_ci(
  mdat,
  scores = NULL,
  labels = NULL,
  eps = 1e-15,
  alpha = 0.05,
  dtype = "normal",
  metrics = NULL,
  ...
)
```

Arguments

`mdat` An `S3` object created by the `mmdata()` function. It contains formatted scores and labels. The `prob_metrics` function ignores `scores` and `labels` when `mdat` is specified. These arguments are internally passed to the `mmdata()` function when `mdat` is unspecified. In that case, both `scores` and `labels` must be at least specified.

`scores` A numeric dataset of predicted probabilities. It can be a vector, a matrix, an array, a data frame, or a list.

`labels` A numeric, character, logical, or factor dataset of observed labels. It can be a vector, a matrix, an array, a data frame, or a list.

`eps` A numeric value used to clamp the scores away from `0` and `1` before the log loss is calculated. A single confident and wrong prediction would otherwise make the log loss infinite.

`alpha` A numeric value of the significant level (default: 0.05)

`dtype` A string to specify the distribution used for CI calculation.

dtype	distribution
normal (default)	Normal distribution
z	Normal distribution
t	t-distribution

`metrics` A character vector of additional metrics to calculate. The three metrics above are always returned; `"d2_brier"` and `"d2_logloss"` are returned as well when they are named here, and `"all"` asks for every metric the function knows.

A D2 score rescales a loss against the loss of the null model, the one that predicts the observed prevalence for every case and ignores the scores:

`D2 = 1 - loss(model) / loss(null)`

It is `1` for a perfect model and `0` for one that does no better than the prevalence, and it is negative for a model that does worse - which is a real result rather than an error, so it is not clipped. A dataset holding a single class has a null loss of `0` and so a D2 score of `NA`.

`...` These additional arguments are passed to `mmdata()` for data preparation.

Value

The prob_metrics_ci function returns a data frame with the columns modnames, metrics, mean, error, lower_bound, upper_bound, and n.

See Also

prob_metrics() for the per-dataset metrics themselves. auc_ci() for the equivalent calculation on AUC scores.

Examples

```
##################################################
### Single model & multiple test datasets
###

## Create sample datasets with 100 positives and 100 negatives
samps <- create_sim_samples(4, 100, 100, "good_er")
mdat <- mmdata(samps[["scores"]], samps[["labels"]],
  modnames = samps[["modnames"]],
  dsids = samps[["dsids"]]
)

## Calculate the CIs
prob_metrics_ci(mdat)
```

Index

∗ **datasets**
B1000, 31
B500, 32
C3N150, 36
IB1000, 58
IB500, 58
M2N50F5, 62
P10N10, 72

as.data.frame, 3
as.data.frame(), *8, 9*, *67*, *88*, *90*
as.data.table, 8
auc, 10
auc(), *15–19*, *30, 31*, *38*, *78*, *86*, *88*, *90*
auc_boot, 14
auc_boot(), *11, 12*, *16–20*, *23*, *78*, *88*
auc_ci, 15
auc_ci(), *11, 12*, *14, 15*, *18–20*, *23*, *88*, *95*
auc_delong, 17
auc_delong(), *15–17*, *20, 21*, *23*
auc_diff, 20
auc_diff(), *14, 15*, *18, 19*, *88*
autoplot, 23
autoplot(), *44*, *54*, *64*, *82*, *88*
average_precision, 30
average_precision(), *11, 12*, *88*

B1000, 31
B500, 32
best_cutoff, 33

C3N150, 36
classification_report, 36
classification_report(), *89, 90*
create_sim_samples, 38
create_sim_samples(), *44*, *87, 88*

evalmod, 40
evalmod(), *3, 4*, *8–10*, *12*, *15–17*, *24–26*, *30, 31*, *33*, *35*, *37–39*, *48*, *50–54*, *59*, *61–68*, *70*, *73, 74*, *76–78*, *81, 82*, *86–90*, *93*

format_nfold, 48
format_nfold(), *44*, *51*, *70*, *87*
format_points, 49
fortify, 52
fortify(), *9*, *26*, *44*, *88*

grid::grid.draw(), *25*
gridExtra::arrangeGrob(), *25*

IB1000, 58
IB500, 58

join_labels, 59
join_labels(), *41*, *44*, *48*, *61*, *69, 70*, *87, 88*
join_scores, 60
join_scores(), *41*, *44*, *48*, *59*, *69, 70*, *87, 88*

M2N50F5, 62
metric_curve, 62
metric_curve(), *65*, *67*, *89, 90*
metric_table, 65
metric_table(), *33–35*
mmdata, 68
mmdata(), *10*, *14*, *18*, *33*, *37*, *39*, *41*, *43, 44*, *48*, *50, 51*, *59*, *61*, *63–66*, *87–90*, *92–94*

P10N10, 72
part, 73
part(), *76–78*, *88*, *90*
pauc, 76
pauc(), *12*, *74*, *88*
plot, 80
plot(), *26*, *44*, *64*, *88*
prbe, 85
prbe(), *31*, *35*, *88*
precrec, 87
precrec-package (precrec), 87

print, 89
prob_metrics, 91
prob_metrics(), *38*, *51*, *88*, *95*
prob_metrics_ci, 93
prob_metrics_ci(), *88*, *93*