{SLmetrics}: Machine Learning Performance Evaluation on Steroids

Version 0.3-1

Serkan Korkmaz

Table of contents

Preface

{SLmetrics} started as a personal project to learn C++, and was never *really* meant to be published nor, infact, named {SLmetrics}. But as time went by, and the committed time and commits grew the name stayed, and the goal to publish a functioning data science R package seemed like the natural next step.

The primary goal of {SLmetrics} is to be a *fast*, *memory efficient* and *reliable* low-level successor to {MLmetrics}; and the current benchmarks in Chapter 3 suggets that this goal, in fact, have been achieved.

⚠ Warning

{SLmetrics} and the documentation is currently under development

Mock Knuth (1984)

1 Introduction

Note

The disucssion in this section is academic, I have the outmost respect for all the developers, contributors and users of the {pkgs}. We are, afterall, united in our love for programming, data-science and `R`

There are currently three {pkgs} that are developed with machine leaning performance evaluation in mind: {MLmetrics}, {yardstick}, {mlr3measures}. These {pkgs} have historically bridged the gap between `R` and `Python` in terms of machine learning and data science.

1.1 The status-quo of {pkgs}

{MLmetrics} can be considered *the* legacy code when it comes to performance evaluation, and it served as a backend in {yardstick} up to version 0.0.2. It is built entirely on base R, and has been stable since its inception almost 10 years ago.

However, it appears that the development has reached it's peak and is currently stale - see, for example, this stale PR related to this issue. Micro- and macro-averages have been implented in {scikit-learn} for many years, and {MLmetrics} simply didn't keep up with the development.

{yardstick}, on the other hand, carried the torch forward and implemented these modern features. {yardstick} closely follows the syntax, naming and functionality of {scikit-learn} but is built with {tidyverse} tools; although the source code is nice to look at, it does introduce some serious overhead and carries the risk of deprecations.

Furthermore, it complicates a simple application by its verbose function naming, see for example `metric()`-function for `<tbl>` and `metric_vec()`-function for `<numeric>` - the output is the same, but the call is different. {yardstick} can't handle more than one positive class at a time, so the end-user is forced to run the same function more than once to get performance metrics for the adjacent classes.

1.1.1 Summary

In short, the existing {pkgs} are outdated, inefficient and insufficient for modern large-scale machine learning applications.

Table 1.1: Calculating RMSE on 1e7 vectors

```r
set.seed(1903)
actual <- rnorm(1e7)
predicted <- actual + rnorm(1e7)

bench::mark(
    `{SLmetrics}` = SLmetrics::rmse(actual, predicted),
    `{MLmetrics}` = MLmetrics::RMSE(predicted, actual),
    iterations    = 100
)
```

```
# A tibble: 2 x 6
  expression       min   median `itr/sec` mem_alloc `gc/sec`
  <bch:expr>  <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
1 {SLmetrics}   30.1ms   30.7ms      32.5    6.13MB      0
2 {MLmetrics}   60.9ms   62.4ms      15.8   76.39MB     77.1
```

1.2 Why {SLmetrics}?

As the name suggests, {SLmetrics} closely resembles {MLmetrics} in it's *simplistic* and *low-level* implementation of machine learning metrics. The resemblance ends there, however.

{SLmetrics} are developed with three things in mind: *speed*, *efficiency* and *scalability*. And therefore addresses the shortcomings of the status-quo by construction - the {pkg} is built on `c++` and {Rcpp} from the ground up. See Table 1.1 where

This shows that well-written `R`-code is hard to beat speed-wise. {MLmetrics} is roughly 20% faster - but uses 30,000 times more memory. How about constructing a confusion matrix

{SLmetrics} uses 1/50th of the time {MLmetrics} and the memory usage is equivalent as the previous example but uses significantly less memory than {MLmetrics}.

1.2.1 Summary

{SLmetrics} is, in the worst-case scenario, on par with low-level `R` implementations of equivalent metrics and is a multitude more memory-efficient than *any* of the {pkgs}. A detailed benchmark can be found here.

Table 1.2: Computing a 3x3 confusion matrix on 1e7 vectors

```r
set.seed(1903)
actual <- factor(sample(letters[1:3], size = 1e7, replace = TRUE))
predicted <- factor(sample(letters[1:3], size = 1e7, replace = TRUE))

bench::mark(
    `{SLmetrics}` = SLmetrics::cmatrix(actual, predicted),
    `{MLmetrics}` = MLmetrics::ConfusionMatrix(actual, predicted),
    check         = FALSE,
    iterations    = 100
)
```

```
# A tibble: 2 x 6
  expression       min   median `itr/sec` mem_alloc `gc/sec`
  <bch:expr>  <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
1 {SLmetrics}   8.56ms   8.59ms    115.      4.12KB     0
2 {MLmetrics} 247.52ms 252.72ms      3.66   381.6MB     7.55
```

1.3 Key takeaways

2 Summary

2.1 Basic Usage

2.2 Installation

2.2.1 Stable version

```
## install stable release
devtools::install_github(
  repo = 'https://github.com/serkor1/SLmetrics@*release',
  ref  = 'main'
)
```

2.2.2 Development version

```
## install development version
devtools::install_github(
  repo = 'https://github.com/serkor1/SLmetrics',
  ref  = 'development'
)
```

3 Benchmarking

In this section a detailed benchmark of {SLmetrics} is conducted. The benchmarks will be conducted on randomly selected functions, and then compared to {pkg} discussed in Chapter 1. The benchmarks are conducted on three parameters: median execution time, memory usage and `gc()` calls.

This section strucutred as follows, Section 3.1 sets up the infrastructure needed to conduct the benchmark in an unbiased way, in Section 3.2 the benchmarks are conducted and discussed and summarized in Section 3.3 and Section 3.4 respectively.

3.1 The setup

To conduct the benchmarking two functions are defined. `create_regression()` and `create_factor()`, both functions returns a vector of `actual` and `predicted` values with a `length` of 10,000,000 rows.

3.1.1 Regression problems

The benchmarks on regression metrics is conducted on correlated absolute value `<numeric>`-vectors, with uniformly distributed weights. `create_regression()` returns a named `list`, and is defined below:

```r
# regression function
create_regression <- function(
    n = 1e7) {

  # 1) actual
  # values
  actual <- abs(rnorm(n = n))

  # 2) predicted
  # values
  predicted <- actual + abs(rnorm(n = n))
  # 3) generate
  # weights
  w <- runif(n)

  list(
    actual    = actual,
    predicted = predicted,
    w         = w
  )
}
```

3.1.2 Classification problems

The benchmarks on classification metrics is conducted on the randomly sampled letters `c("a", "b", "c")`. `create_regression()` returns a vector of `<factor>`, and is defined below:

```
# classification function
create_factor <- function(
    k = 3,
    balanced = TRUE,
    n = 1e7) {

  probs <- NULL

  if (!balanced) {

    probs <- rbeta(
      n = k,
      shape1 = 10,
      shape2 = 2
    )

    probs[which.min(probs)] <- 0

    probs <- probs / sum(probs)

  }

  factor(
    x = sample(
      1:k,
      size = n,
      replace = TRUE,
      prob = probs
    ),
    labels = letters[1:k],
    levels = 1:k
  )
}
```

3.1.3 Staging the testing ground

The vectors used in the benchmarks are created with the seed `1903` for reproducibility, see below:

```
# 1) set seed for reproducibility
set.seed(1903)

# 2) create classification
# problem
fct_actual <- create_factor()
fct_predicted <- create_factor()

# 3) create regression
# problem

# 3.1) store results
# in regression
lst_regression <- create_regression()

# 3.2) assign the values
# accordingly
num_actual <- lst_regression$actual
num_predicted <- lst_regression$predicted
num_weights <- lst_regression$w
```

3.2 Benchmarking

To conduct the benchmark {bench} is used. Before the benchmarks are conducted, a `benchmark()`-wrapper is created.

This wrapper conducts `m` (Default: 10) benchmarks, with 10 iterations for each benchmarked function passed into `benchmark()` - to allow for warm-up the first iteration is discarded. The wrapper is defined as follows:

```r
benchmark <- function(
  ...,
  m = 10) {
  library(magrittr)
  # 1) create list
  # for storing values
  performance <- list()

  for (i in 1:m) {

     # 1) run the benchmarks
    results <- bench::mark(
      ...,
      iterations = 10,
      check = FALSE
    )

    # 2) extract values
    # and calculate medians
    performance$time[[i]]  <- setNames(lapply(results$time, mean), results$expression)
    performance$memory[[i]] <- setNames(lapply(results$memory, function(x) { sum(x$bytes,
    performance$n_gc[[i]] <- setNames(lapply(results$n_gc, sum), results$expression)

  }

  purrr::pmap_dfr(
  list(performance$time, performance$memory, performance$n_gc),
  ~{
    tibble::tibble(
      expression = names(..1),
      time = unlist(..1),
      memory = unlist(..2),
      n_gc = unlist(..3)
```

Table 3.1: Benchmarking selected regression metrics

```
benchmark(
    `{RMSE}`  = SLmetrics::rmse(num_actual, num_predicted),
    `{Pinball Loss}` = SLmetrics::pinball(num_actual, num_predicted),
    `{Huber Loss}` = SLmetrics::huberloss(num_actual, num_predicted)
)
```

```
#> # A tibble: 3 x 4
#>   expression     execution_time memory_usage gc_calls
#>   <fct>                <bch:tm>    <bch:byt>    <dbl>
#> 1 {RMSE}                 31.1ms           0B        0
#> 2 {Pinball Loss}         31.2ms           0B        0
#> 3 {Huber Loss}           76.1ms           0B        0
```

```
      )
    }
  ) %>%
    dplyr::mutate(expression = factor(expression, levels = unique(expression))) %>%
    dplyr::group_by(expression) %>%
    dplyr::filter(dplyr::row_number() > 1) %>%
    dplyr::summarize(
      execution_time = bench::as_bench_time(median(time)),
      memory_usage = bench::as_bench_bytes(median(memory)),
      gc_calls = median(n_gc),
      .groups = "drop"
    )

}
```

3.2.1 Regression metrics

3.2.2 Classification metrics

Table 3.2: Benchmarking RMSE across

```r
benchmark(
    `{SLmetrics}` = SLmetrics::rmse(num_actual, num_predicted),
    `{MLmetrics}` = MLmetrics::RMSE(num_actual, num_predicted),
    `{yardstick}` = yardstick::rmse_vec(num_actual, num_predicted),
    `{mlr3measures}` = mlr3measures::rmse(num_actual, num_predicted)
)
```

```
#> # A tibble: 4 x 4
#>   expression     execution_time memory_usage gc_calls
#>   <fct>                <bch:tm>    <bch:byt>    <dbl>
#> 1 {SLmetrics}            31.2ms           0B        0
#> 2 {MLmetrics}            62.5ms       76.3MB        1
#> 3 {yardstick}           174.6ms      419.6MB        9
#> 4 {mlr3measures}         88.2ms       76.3MB        1
```

Table 3.3: Benchmarking selected classification metrics

```r
benchmark(
    `{Confusion Matrix}`  = SLmetrics::cmatrix(fct_actual, fct_predicted),
    `{Accuracy}` = SLmetrics::accuracy(fct_actual, fct_predicted),

    `{F-beta}` = SLmetrics::fbeta(fct_actual, fct_predicted)
)
```

```
#> # A tibble: 3 x 4
#>   expression         execution_time memory_usage gc_calls
#>   <fct>                    <bch:tm>    <bch:byt>    <dbl>
#> 1 {Confusion Matrix}         8.61ms           0B        0
#> 2 {Accuracy}                  8.6ms           0B        0
#> 3 {F-beta}                   8.61ms           0B        0
```

Table 3.4: Benchmarking a 3x3 confusion matrix across

```r
benchmark(
    `{SLmetrics}`    = SLmetrics::cmatrix(fct_actual, fct_predicted),
    `{MLmetrics}`    = MLmetrics::ConfusionMatrix(fct_predicted, fct_actual),

    `{yardstick}`    = yardstick::conf_mat(table(fct_actual, fct_predicted))
)
```

```
#> # A tibble: 3 x 4
#>   expression  execution_time memory_usage gc_calls
#>   <fct>             <bch:tm>    <bch:byt>    <dbl>
#> 1 {SLmetrics}         8.61ms           0B        0
#> 2 {MLmetrics}        249.1ms        381MB        7
#> 3 {yardstick}       249.79ms        381MB        7
```

3.3 Discussion

Does speed *really* matter at the milliseconds level, and justify the raîson d'être for {SLmetrics} - the answer is inevitably **no**. A reduction of a few milliseconds may marginally improve performance, perhaps shaving off minutes or hours in large-scale grid searches or multi-model experiments. While this might slightly reduce cloud expenses, the overall impact is often negligible unless you're operating at an enormous scale or in latency-critical environments.

However, the memory efficiency of {SLmetrics} is where its real value lies. Its near-zero RAM usage allows more memory to be allocated for valuable tasks, such as feeding larger datasets into models. This can directly lead to higher-performing models, as more data generally improves learning outcomes. Furthermore, by optimizing memory usage, {SLmetrics} can reduce infrastructure costs significantly, as less powerful machines or fewer cloud resources may be required to achieve the same — or better — results.

In short, while speed optimization may seem like a more visible metric, it's the memory efficiency of {SLmetrics} that has a broader, more transformative impact on machine learning workflows, from enabling better model performance to substantial cost reductions.

3.4 Conclusion

The benchmarks conducted in Section 3.2 suggests that {SLmetrics} *is* the memory-efficient and fast alternative to {MLmetrics}, {yardstick} and {mlr3measures}.

In the worst performing benchmarks {SLmetrics} is on par with low-level implementations of equivalent metrics and is consistently more memory-efficient in all benchmarks.

Part I

Regression functions

In this section all available regression metrics and related documentation is described. Common for all regression functions is that they use the class `numeric`.

Examples

```r
## actual
actual <- c(1.3, 2.4, 0.7, 0.1)

## predicted
predicted <- c(0.7, 2.9, 0.76, 0.07)
```

```r
SLmetrics::rmse(
    actual,
    predicted
)
```

```
[1] 0.3919503
```

concordance correlation coefficient

ccc.numeric

R Documentation

Description

The `ccc()`-function computes the simple and weighted concordance correlation coefficient between the two vectors of predicted and observed `<numeric>` values. The `weighted.ccc()` function computes the weighted Concordance Correlation Coefficient. If `correction` is TRUE σ^2 is adjusted by $\frac{1-n}{n}$ in the intermediate steps.

Usage

```
## S3 method for class 'numeric'
ccc(actual, predicted, correction = FALSE, ...)

## S3 method for class 'numeric'
weighted.ccc(actual, predicted, w, correction = FALSE, ...)

ccc(...)

weighted.ccc(...)
```

Arguments

actual

A `<numeric>`-vector of length n. The observed (continuous) response variable.

predicted

A `<numeric>`-vector of length n. The estimated (continuous) response variable.

correction

A `<logical>` vector of length 1 (default: FALSE). If TRUE the variance and covariance will be adjusted with $\frac{1-n}{n}$

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. The weight assigned to each observation in the data.

Value

A `<numeric>` vector of length 1.

Calculation

The metric is calculated as follows,

$$\rho_c = \frac{2\rho\sigma_x\sigma_y}{\sigma_x^2 + \sigma_y^2 + (\mu_x - \mu_y)^2}$$

Where ρ is the pearson correlation coefficient, σ is the standard deviation and μ is the simple mean of `actual` and `predicted`.

Examples

```
# 1) fit a linear
# regression
model <- lm(
  mpg ~ .,
  data = mtcars
)

# 1.1) define actual
# and predicted values
# to measure performance
actual    <- mtcars$mpg
predicted <- fitted(model)

# 2) evaluate in-sample model
```

```r
# performance
cat(
  "Concordance Correlation Coefficient", ccc(
    actual     = actual,
    predicted  = predicted,
    correction = FALSE
  ),
  "Concordance Correlation Coefficient (corrected)", ccc(
    actual     = actual,
    predicted  = predicted,
    correction = TRUE
  ),
  "Concordance Correlation Coefficient (weigthed)", weighted.ccc(
    actual     = actual,
    predicted  = predicted,
    w          = mtcars$mpg/mean(mtcars$mpg),
    correction = FALSE
  ),
  sep = "\n"
)
```

huber loss

huberloss.numeric

R Documentation

Description

The `huberloss()`-function computes the simple and weighted huber loss between the predicted and observed `<numeric>` vectors. The `weighted.huberloss()` function computes the weighted Huber Loss.

Usage

```
## S3 method for class 'numeric'
huberloss(actual, predicted, delta = 1, ...)

## S3 method for class 'numeric'
weighted.huberloss(actual, predicted, w, delta = 1, ...)

huberloss(...)

weighted.huberloss(...)
```

Arguments

actual

A `<numeric>`-vector of length n. The observed (continuous) response variable.

predicted

A `<numeric>`-vector of length n. The estimated (continuous) response variable.

delta

A `<numeric>`-vector of length 1 (default: 1). The threshold value for switch between functions (see calculation).

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. The weight assigned to each observation in the data.

Value

A `<numeric>` vector of length 1.

Calculation

The metric is calculated as follows,

$$\frac{1}{2}(y - \upsilon)^2 \ for \ |y - \upsilon| \leq \delta$$

and

$$\delta |y - \upsilon| - \frac{1}{2}\delta^2 \ for \text{ otherwise}$$

where y and υ are the `actual` and `predicted` values respectively. If `w` is not NULL, then all values are aggregated using the weights.

Examples

```r
# 1) fit a linear
# regression
model <- lm(
  mpg ~ .,
  data = mtcars
)

# 1.1) define actual
# and predicted values
# to measure performance
```

```r
actual    <- mtcars$mpg
predicted <- fitted(model)


# 2) calculate the metric
# with delta 0.5
huberloss(
  actual = actual,
  predicted = predicted,
  delta = 0.5
)

# 3) caclulate weighted
# metric using arbitrary weights
w <- rbeta(
  n = 1e3,
  shape1 = 10,
  shape2 = 2
)

huberloss(
  actual = actual,
  predicted = predicted,
  delta = 0.5,
  w     = w
)
```

mean absolute error

mae.numeric

R Documentation

Description

The `mae()`-function computes the mean absolute error between the observed and predicted `<numeric>` vectors. The `weighted.mae()` function computes the weighted mean absolute error.

Usage

```
## S3 method for class 'numeric'
mae(actual, predicted, ...)

## S3 method for class 'numeric'
weighted.mae(actual, predicted, w, ...)

mae(...)

weighted.mae(...)
```

Arguments

actual

A `<numeric>`-vector of length n. The observed (continuous) response variable.

predicted

A `<numeric>`-vector of length n. The estimated (continuous) response variable.

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. The weight assigned to each observation in the data.

Value

A `<numeric>` vector of length 1.

Calculation

The metric is calulated as follows,

$$\frac{\sum_i^n |y_i - \upsilon_i|}{n}$$

Examples

```r
# 1) fit a linear
# regression
model <- lm(
  mpg ~ .,
  data = mtcars
)

# 1.1) define actual
# and predicted values
# to measure performance
actual    <- mtcars$mpg
predicted <- fitted(model)

# 2) evaluate in-sample model
# performance using Mean Absolute Error (MAE)
cat(
  "Mean Absolute Error", mae(
    actual    = actual,
    predicted = predicted,
  ),
  "Mean Absolute Error (weighted)", weighted.mae(
    actual    = actual,
    predicted = predicted,
    w          = mtcars$mpg/mean(mtcars$mpg)
  ),
  sep = "\n"
)
```

mean absolute percentage error

mape.numeric

R Documentation

Description

The `mape()`-function computes the mean absolute percentage error between the observed and predicted `<numeric>` vectors. The `weighted.mape()` function computes the weighted mean absolute percentage error.

Usage

```
## S3 method for class 'numeric'
mape(actual, predicted, ...)

## S3 method for class 'numeric'
weighted.mape(actual, predicted, w, ...)

mape(...)

weighted.mape(...)
```

Arguments

actual

A `<numeric>`-vector of length n. The observed (continuous) response variable.

predicted

A `<numeric>`-vector of length n. The estimated (continuous) response variable.

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. The weight assigned to each observation in the data.

Value

A `<numeric>` vector of length 1.

Calculation

The metric is calculated as,

$$\frac{1}{n} \sum_{i}^{n} \frac{|y_i - \upsilon_i|}{|y_i|}$$

Examples

```
# 1) fit a linear
# regression
model <- lm(
  mpg ~ .,
  data = mtcars
)

# 1.1) define actual
# and predicted values
# to measure performance
actual    <- mtcars$mpg
predicted <- fitted(model)

# 2) evaluate in-sample model
# performance using Mean Absolute Percentage Error (MAPE)
cat(
  "Mean Absolute Percentage Error", mape(
    actual    = actual,
    predicted = predicted,
  ),
  "Mean Absolute Percentage Error (weighted)", weighted.mape(
    actual    = actual,
    predicted = predicted,
    w         = mtcars$mpg/mean(mtcars$mpg)
  ),
  sep = "\n"
)
```

Matthews correlation coefficient

mcc.factor

R Documentation

Description

The `mcc()`-function computes the Matthews Correlation Coefficient (MCC), also known as the ϕ-coefficient, between two vectors of predicted and observed `factor()` values. The `weighted.mcc()` function computes the weighted Matthews Correlation Coefficient.

Usage

```
## S3 method for class 'factor'
mcc(actual, predicted, ...)

## S3 method for class 'factor'
weighted.mcc(actual, predicted, w, ...)

## S3 method for class 'cmatrix'
mcc(x, ...)

## S3 method for class 'factor'
phi(actual, predicted, ...)

## S3 method for class 'factor'
weighted.phi(actual, predicted, w, ...)

## S3 method for class 'cmatrix'
phi(x, ...)

mcc(...)

weighted.mcc(...)
```

```
phi(...)

weighted.phi(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels

predicted

A vector of `<factor>`-vector of length n, and k levels

...

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default

x

A confusion matrix created `cmatrix()`

Value

A `<numeric>`-vector of length 1

Calculation

The metric is calculated as follows,

$$\frac{\#TP \times \#TN - \#FP \times \#FN}{\sqrt{(\#TP + \#FP)(\#TP + \#FN)(\#TN + \#FP)(\#TN + \#FN)}}$$

Examples

```r
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate performance
# using Matthews Correlation Coefficient
cat(
  "Matthews Correlation Coefficient", mcc(
    actual    = actual,
    predicted = predicted
  ),
  "Matthews Correlation Coefficient (weighted)", weighted.mcc(
    actual    = actual,
    predicted = predicted,
    w         = iris$Petal.Length/mean(iris$Petal.Length)
  ),
  sep = "\n"
)
```

mean squared error

mse.numeric

R Documentation

Description

The `mse()`-function computes the mean squared error between the observed and predicted `<numeric>` vectors. The `weighted.mse()` function computes the weighted mean squared error.

Usage

```
## S3 method for class 'numeric'
mse(actual, predicted, ...)

## S3 method for class 'numeric'
weighted.mse(actual, predicted, w, ...)

mse(...)

weighted.mse(...)
```

Arguments

actual

A `<numeric>`-vector of length n. The observed (continuous) response variable.

predicted

A `<numeric>`-vector of length n. The estimated (continuous) response variable.

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. The weight assigned to each observation in the data.

Value

A `<numeric>` vector of length 1.

Calculation

The metric is calculated as,

$$\frac{1}{n} \sum_{i}^{n} (y_i - \upsilon_i)^2$$

Where y_i and υ_i are the `actual` and `predicted` values respectively.

Examples

```r
# 1) fit a linear
# regression
model <- lm(
  mpg ~ .,
  data = mtcars
)

# 1.1) define actual
# and predicted values
# to measure performance
actual    <- mtcars$mpg
predicted <- fitted(model)

# 2) evaluate in-sample model
# performance using Mean Squared Error (MSE)
cat(
  "Mean Squared Error", mse(
    actual    = actual,
    predicted = predicted,
  ),
  "Mean Squared Error (weighted)", weighted.mse(
    actual    = actual,
    predicted = predicted,
    w         = mtcars$mpg/mean(mtcars$mpg)
  ),
  sep = "\n"
)
```

pinball loss

pinball.numeric

R Documentation

Description

The `pinball()`-function computes the pinball loss between the observed and predicted `<numeric>` vectors. The `weighted.pinball()` function computes the weighted Pinball Loss.

Usage

```
## S3 method for class 'numeric'
pinball(actual, predicted, alpha = 0.5, deviance = FALSE, ...)

## S3 method for class 'numeric'
weighted.pinball(actual, predicted, w, alpha = 0.5, deviance = FALSE, ...)

pinball(...)

weighted.pinball(...)
```

Arguments

actual

A `<numeric>`-vector of length n. The observed (continuous) response variable.

predicted

A `<numeric>`-vector of length n. The estimated (continuous) response variable.

alpha

A `<numeric>`-value of length 1 (default: 0.5). The slope of the pinball loss function.

deviance

A `<logical>`-value of length 1 (default: FALSE). If TRUE the function returns the D^2 loss.

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. The weight assigned to each observation in the data.

Value

A `<numeric>` vector of length 1.

Calculation

The metric is calculated as,

$$\text{PinballLoss}_{\text{unweighted}} = \frac{1}{n} \sum_{i=1}^{n} [\alpha \cdot \max(0, y_i - \hat{y}_i) - (1 - \alpha) \cdot \max(0, \hat{y}_i - y_i)]$$

where y_i is the actual value, $\hat{y}_i$ is the predicted value and α is the quantile level.

Examples

```r
# 1) fit a linear
# regression
model <- lm(
  mpg ~ .,
  data = mtcars
)

# 1.1) define actual
# and predicted values
# to measure performance
actual    <- mtcars$mpg
predicted <- fitted(model)

# 2) evaluate in-sample model
# performance using Pinball Loss
```

```r
cat(
  "Pinball Loss", pinball(
    actual    = actual,
    predicted = predicted,
  ),
  "Pinball Loss (weighted)", weighted.pinball(
    actual    = actual,
    predicted = predicted,
    w         = mtcars$mpg/mean(mtcars$mpg)
  ),
  sep = "\n"
)
```

relative absolute error

rae.numeric

R Documentation

Description

The `rae()`-function calculates the normalized relative absolute error between the predicted and observed `<numeric>` vectors. The `weighted.rae()` function computes the weigthed relative absolute error.

Usage

```
## S3 method for class 'numeric'
rae(actual, predicted, ...)

## S3 method for class 'numeric'
weighted.rae(actual, predicted, w, ...)

rae(...)

weighted.rae(...)
```

Arguments

actual

A `<numeric>`-vector of length n. The observed (continuous) response variable.

predicted

A `<numeric>`-vector of length n. The estimated (continuous) response variable.

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. The weight assigned to each observation in the data.

Value

A `<numeric>` vector of length 1.

Calculation

The Relative Absolute Error (RAE) is calculated as:

$$\text{RAE} = \frac{\sum_{i=1}^{n} |y_i - \upsilon_i|}{\sum_{i=1}^{n} |y_i - \bar{y}|}$$

Where y_i are the `actual` values, υ_i are the `predicted` values, and $\bar{y}$ is the mean of the `actual` values.

Examples

```r
# 1) fit a linear
# regression
model <- lm(
  mpg ~ .,
  data = mtcars
)

# 1.1) define actual
# and predicted values
# to measure performance
actual    <- mtcars$mpg
predicted <- fitted(model)

# 2) evaluate in-sample model
# performance using Relative Absolute Error (RAE)
cat(
  "Relative Absolute Error", rae(
    actual    = actual,
    predicted = predicted,
  ),
  "Relative Absolute Error (weighted)", weighted.rae(
    actual    = actual,
    predicted = predicted,
    w         = mtcars$mpg/mean(mtcars$mpg)
  ),
  sep = "\n"
)
```

root mean squared error

rmse.numeric

R Documentation

Description

The `rmse()`-function computes the root mean squared error between the observed and predicted `<numeric>` vectors. The `weighted.rmse()` function computes the weighted root mean squared error.

Usage

```
## S3 method for class 'numeric'
rmse(actual, predicted, ...)

## S3 method for class 'numeric'
weighted.rmse(actual, predicted, w, ...)

rmse(...)

weighted.rmse(...)
```

Arguments

actual

A `<numeric>`-vector of length n. The observed (continuous) response variable.

predicted

A `<numeric>`-vector of length n. The estimated (continuous) response variable.

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. The weight assigned to each observation in the data.

Value

A `<numeric>` vector of length 1.

Calculation

The metric is calculated as,

$$\sqrt{\frac{1}{n} \sum_{i}^{n} (y_i - \upsilon_i)^2}$$

Where y_i and υ_i are the `actual` and `predicted` values respectively.

Examples

```
# 1) fit a linear
# regression
model <- lm(
  mpg ~ .,
  data = mtcars
)

# 1.1) define actual
# and predicted values
# to measure performance
actual    <- mtcars$mpg
predicted <- fitted(model)

# 2) evaluate in-sample model
# performance using Root Mean Squared Error (RMSE)
cat(
  "Root Mean Squared Error", rmse(
    actual    = actual,
    predicted = predicted,
  ),
  "Root Mean Squared Error (weighted)", weighted.rmse(
```

```r
    actual    = actual,
    predicted = predicted,
    w         = mtcars$mpg/mean(mtcars$mpg)
  ),
  sep = "\n"
)
```

root mean squared logarithmic error

rmsle.numeric

R Documentation

Description

The `rmsle()`-function computes the root mean squared logarithmic error between the observed and predicted `<numeric>` vectors. The `weighted.rmsle()` function computes the weighted root mean squared logarithmic error.

Usage

```
## S3 method for class 'numeric'
rmsle(actual, predicted, ...)

## S3 method for class 'numeric'
weighted.rmsle(actual, predicted, w, ...)

rmsle(...)

weighted.rmsle(...)
```

Arguments

actual

A `<numeric>`-vector of length n. The observed (continuous) response variable.

predicted

A `<numeric>`-vector of length n. The estimated (continuous) response variable.

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. The weight assigned to each observation in the data.

Value

A `<numeric>` vector of length 1.

Calculation

The metric is calculated as,

$$\sqrt{\frac{1}{n} \sum_{i}^{n} (\log(1 + y_i) - \log(1 + \upsilon_i))^2}$$

Where y_i and υ_i are the `actual` and `predicted` values respectively.

Examples

```
# 1) fit a linear
# regression
model <- lm(
  mpg ~ .,
  data = mtcars
)

# 1.1) define actual
# and predicted values
# to measure performance
actual    <- mtcars$mpg
predicted <- fitted(model)


# 2) evaluate in-sample model
# performance using Root Mean Squared Logarithmic Error (RMSLE)
cat(
  "Root Mean Squared Logarithmic Error", rmsle(
    actual    = actual,
    predicted = predicted,
  ),
  "Root Mean Squared Logarithmic Error (weighted)", weighted.rmsle(
    actual    = actual,
    predicted = predicted,
    w         = mtcars$mpg/mean(mtcars$mpg)
  ),
  sep = "\n"
)
```

relative root mean squared error

rrmse.numeric

R Documentation

Description

The `rrmse()`-function computes the Relative Root Mean Squared Error between the observed and predicted `<numeric>` vectors. The `weighted.rrmse()` function computes the weighted Relative Root Mean Squared Error.

Usage

```
## S3 method for class 'numeric'
rrmse(actual, predicted, normalization = 1L, ...)

## S3 method for class 'numeric'
weighted.rrmse(actual, predicted, w, normalization = 1L, ...)

rrmse(...)

weighted.rrmse(...)
```

Arguments

actual

A `<numeric>`-vector of length n. The observed (continuous) response variable.

predicted

A `<numeric>`-vector of length n. The estimated (continuous) response variable.

normalization

A `<numeric>`-value of length 1 (default: 1). 0: mean-normalization, 1: range-normalization, 2: IQR-normalization.

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. The weight assigned to each observation in the data.

Value

A `<numeric>` vector of length 1.

Calculation

The metric is calculated as,

$$\frac{RMSE}{\gamma}$$

Where γ is the normalization factor.

Examples

```
# 1) fit a linear
# regression
model <- lm(
  mpg ~ .,
  data = mtcars
)

# 1.1) define actual
# and predicted values
# to measure performance
actual    <- mtcars$mpg
predicted <- fitted(model)

# 2) evaluate in-sample model
# performance using Relative Root Mean Squared Error (RRMSE)
cat(
```

```r
  "IQR Relative Root Mean Squared Error", rrmse(
    actual        = actual,
    predicted     = predicted,
    normalization = 2
  ),
  "IQR Relative Root Mean Squared Error (weighted)", weighted.rrmse(
    actual        = actual,
    predicted     = predicted,
    w             = mtcars$mpg/mean(mtcars$mpg),
    normalization = 2
  ),
  sep = "\n"
)
```

root relative squared error

rrse.numeric

R Documentation

Description

The `rrse()`-function calculates the root relative squared error between the predicted and observed `<numeric>` vectors. The `weighted.rrse()` function computes the weighed root relative squared errorr.

Usage

```
## S3 method for class 'numeric'
rrse(actual, predicted, ...)

## S3 method for class 'numeric'
weighted.rrse(actual, predicted, w, ...)

rrse(...)

weighted.rrse(...)
```

Arguments

actual

A `<numeric>`-vector of length n. The observed (continuous) response variable.

predicted

A `<numeric>`-vector of length n. The estimated (continuous) response variable.

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. The weight assigned to each observation in the data.

Value

A `<numeric>` vector of length 1.

Calculation

The metric is calculated as,

$$\text{RRSE} = \sqrt{\frac{\sum_{i=1}^n (y_i - \upsilon_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

Where y_i are the `actual` values, υ_i are the `predicted` values, and $\bar{y}$ is the mean of the `actual` values.

Examples

```r
# 1) fit a linear
# regression
model <- lm(
  mpg ~ .,
  data = mtcars
)

# 1.1) define actual
# and predicted values
# to measure performance
actual    <- mtcars$mpg
predicted <- fitted(model)

# 2) evaluate in-sample model
# performance using Relative Root Squared Errror (RRSE)
cat(
  "Relative Root Squared Errror", rrse(
    actual    = actual,
    predicted = predicted,
  ),
  "Relative Root Squared Errror (weighted)", weighted.rrse(
    actual    = actual,
    predicted = predicted,
    w         = mtcars$mpg/mean(mtcars$mpg)
  ),
  sep = "\n"
)
```

symmetric mean absolute percentage error

smape.numeric

R Documentation

Description

The `smape()`-function computes the symmetric mean absolute percentage error between the observed and predicted `<numeric>` vectors. The `weighted.smape()` function computes the weighted symmetric mean absolute percentage error.

Usage

```
## S3 method for class 'numeric'
smape(actual, predicted, ...)

## S3 method for class 'numeric'
weighted.smape(actual, predicted, w, ...)

smape(...)

weighted.smape(...)
```

Arguments

actual

A `<numeric>`-vector of length n. The observed (continuous) response variable.

predicted

A `<numeric>`-vector of length n. The estimated (continuous) response variable.

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. The weight assigned to each observation in the data.

Value

A `<numeric>` vector of length 1.

Calculation

The metric is calculated as follows,

$$\sum_i^n \frac{1}{n} \frac{|y_i - \upsilon_i|}{\frac{|y_i|+|\upsilon_i|}{2}}$$

where y_i and υ_i is the `actual` and `predicted` values respectively.

Examples

```
# 1) fit a linear
# regression
model <- lm(
  mpg ~ .,
  data = mtcars
)

# 1.1) define actual
# and predicted values
# to measure performance
actual    <- mtcars$mpg
predicted <- fitted(model)

# 2) evaluate in-sample model
# performance using Symmetric Mean Absolute Percentage Error (MAPE)
cat(
  "Symmetric Mean Absolute Percentage Error", mape(
    actual    = actual,
    predicted = predicted,
  ),
  "Symmetric Mean Absolute Percentage Error (weighted)", weighted.mape(
```

```r
    actual    = actual,
    predicted = predicted,
    w         = mtcars$mpg/mean(mtcars$mpg)
  ),
  sep = "\n"
)
```

specificity

specificity.factor

R Documentation

Description

The `specificity()`-function computes the specificity, also known as the True Negative Rate (TNR) or selectivity, between two vectors of predicted and observed `factor()` values. The `weighted.specificity()` function computes the weighted specificity.

Usage

```
## S3 method for class 'factor'
specificity(actual, predicted, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
weighted.specificity(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
specificity(x, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
tnr(actual, predicted, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
weighted.tnr(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
tnr(x, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
selectivity(actual, predicted, micro = NULL, na.rm = TRUE, ...)
```

```r
## S3 method for class 'factor'
weighted.selectivity(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
selectivity(x, micro = NULL, na.rm = TRUE, ...)

specificity(...)

tnr(...)

selectivity(...)

weighted.specificity(...)

weighted.tnr(...)

weighted.selectivity(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels.

predicted

A vector of `<factor>`-vector of length n, and k levels.

micro

A `<logical>`-value of length 1 (default: NULL). If TRUE it returns the micro average across all k classes, if FALSE it returns the macro average.

na.rm

A `<logical>` value of length 1 (default: TRUE). If TRUE, NA values are removed from the computation. This argument is only relevant when `micro != NULL`. When `na.rm = TRUE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(na.omit(c(1, 2, NA)))`. When `na.rm = FALSE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(c(1, 2, NA))`.

...

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default.

x

A confusion matrix created `cmatrix()`.

Value

If `micro` is NULL (the default), a named `<numeric>`-vector of length k

If `micro` is TRUE or FALSE, a `<numeric>`-vector of length 1

Examples

```
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
```

```r
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate class-wise performance
# using Specificity

# 4.1) unweighted Specificity
specificity(
  actual    = actual,
  predicted = predicted
)

# 4.2) weighted Specificity
weighted.specificity(
  actual    = actual,
  predicted = predicted,
  w         = iris$Petal.Length/mean(iris$Petal.Length)
)

# 5) evaluate overall performance
# using micro-averaged Specificity
cat(
  "Micro-averaged Specificity", specificity(
    actual    = actual,
    predicted = predicted,
    micro     = TRUE
  ),
  "Micro-averaged Specificity (weighted)", weighted.specificity(
    actual    = actual,
    predicted = predicted,
    w         = iris$Petal.Length/mean(iris$Petal.Length),
    micro     = TRUE
  ),
  sep = "\n"
)
```

Part II

Classification functions

In this section all available classification metrics and related documentation is described. Common for all classifcation functions is that they use the method `foo.factor` or `foo.cmatrix`.

A primer on factors

Consider a classification problem with three classes: `A`, `B`, and `C`. The actual vector of `factor` values is defined as follows:

```r
## set seed
set.seed(1903)

## actual
actual <- factor(
    x = sample(x = 1:3, size = 10, replace = TRUE),
    levels = c(1, 2, 3),
    labels = c("A", "B", "C")
)

## print values
print(actual)
```

```
#>  [1] B A B B A C B C C A
#> Levels: A B C
```

Here, the values 1, 2, and 3 are mapped to `A`, `B`, and `C`, respectively. Now, suppose your model does not predict any `B`'s. The predicted vector of `factor` values would be defined as follows:

```r
## set seed
set.seed(1903)

## predicted
predicted <- factor(
    x = sample(x = c(1, 3), size = 10, replace = TRUE),
    levels = c(1, 2, 3),
    labels = c("A", "B", "C")
)

## print values
print(predicted)
```

```
#>  [1] C A C C C C C C A C
#> Levels: A B C
```

In both cases, $k = 3$, determined indirectly by the `levels` argument.

Examples

In this section a brief introduction to the two methods are given.

factor method

```
## factor method
SLmetrics::accuracy(
  actual,
  predicted
)
```

```
#> [1] 0.3
```

cmatrix method

```
## 1) generate confusion
## matrix (cmatrix class)
confusion_matrix <- SLmetrics::cmatrix(
  actual,
  predicted
)

## 2) check class
class(confusion_matrix)
```

```
#> [1] "cmatrix"
```

```
## 3) summarise
summary(confusion_matrix)
```

```
#> Confusion Matrix (3 x 3)
#> ================================================================================
#>   A B C
#> A 1 0 2
#> B 0 0 4
#> C 1 0 2
#> ================================================================================
#> Overall Statistics (micro average)
#>  - Accuracy:          0.30
#>  - Balanced Accuracy: 0.33
#>  - Sensitivity:       0.30
#>  - Specificity:       0.65
#>  - Precision:         0.30
```

The `confusion_matrix` can be passed into `accuracy()` as follows:

```r
SLmetrics::accuracy(
  confusion_matrix
)
```

```
#> [1] 0.3
```

Using the `cmatrix`-method is more efficient if more than one classification metric is going to be calculated, as the metrics are calculated directly from the `cmatrix`-object, instead of looping though all the values in `actual` and `predicted` values for each metrics. See below:

```r
cat(
  sep = "\n",
  paste("Accuracy:", SLmetrics::accuracy(
  confusion_matrix)),
  paste("Balanced Accuracy:", SLmetrics::baccuracy(
  confusion_matrix))
)
```

```
#> Accuracy: 0.3
#> Balanced Accuracy: 0.333333333333333
```

receiver operator characteristics

ROC.factor

R Documentation

Description

The `ROC()`-function computes the `tpr()` and `fpr()` at thresholds provided by the *response*- or *thresholds*-vector. The function constructs a `data.frame()` grouped by k-classes where each class is treated as a binary classification problem.

Usage

```
## S3 method for class 'factor'
ROC(actual, response, thresholds = NULL, ...)

## S3 method for class 'factor'
weighted.ROC(actual, response, w, thresholds = NULL, ...)

ROC(...)

weighted.ROC(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels.

response

A `<numeric>`-vector of length n. The estimated response probabilities.

thresholds

An optional `<numeric>`-vector of non-zero length (default: NULL).

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. NULL by default.

Value

A data.frame on the following form,

`threshold`

`<numeric>` Thresholds used to determine `tpr()` and `fpr()`

`level`

`<character>` The level of the actual `<factor>`

`label`

`<character>` The levels of the actual `<factor>`

`fpr`

`<numeric>` The false positive rate

`tpr`

`<numeric>` The true positve rate

Examples

```
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
response <-predict(model, type = "response")

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) generate reciever
# operator characteristics
roc <- ROC(
  actual   = actual,
  response = response
)

# 5) plot by species
plot(roc)

# 5.1) summarise
summary(roc)

# 6) provide custom
# threholds
roc <- ROC(
  actual     = actual,
  response   = response,
  thresholds = seq(0, 1, length.out = 4)
)


# 5) plot by species
plot(roc)
```

accuracy

accuracy.factor

R Documentation

Description

The `accuracy()` function computes the accuracy between two vectors of predicted and observed `factor()` values. The `weighted.accuracy()` function computes the weighted accuracy.

Usage

```
## S3 method for class 'factor'
accuracy(actual, predicted, ...)

## S3 method for class 'factor'
weighted.accuracy(actual, predicted, w, ...)

## S3 method for class 'cmatrix'
accuracy(x, ...)

accuracy(...)

weighted.accuracy(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels

predicted

A vector of `<factor>`-vector of length n, and k levels

…

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default

x

A confusion matrix created `cmatrix()`

Value

A `<numeric>`-vector of length 1

Calculation

The metric is calculated as follows,

$$\frac{\#TP + \#TN}{\#TP + \#TN + \#FP + \#FN}$$

Where $\#TP$, $\#TN$, $\#FP$, and $\#FN$ is the number of true positives, true negatives, false positives, and false negatives, respectively.

Examples

```
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate model
# performance
cat(
  "Accuracy", accuracy(
    actual    = actual,
    predicted = predicted
  ),

  "Accuracy (weigthed)", weighted.accuracy(
    actual    = actual,
    predicted = predicted,
    w         = iris$Petal.Length/mean(iris$Petal.Length)
  ),
  sep = "\n"
)
```

balanced accuracy

baccuracy.factor

R Documentation

Description

The `baccuracy()`-function computes the balanced accuracy between two vectors of predicted and observed `factor()` values. The `weighted.baccuracy()` function computes the weighted balanced accuracy.

Usage

```r
## S3 method for class 'factor'
baccuracy(actual, predicted, adjust = FALSE, na.rm = TRUE, ...)

## S3 method for class 'factor'
weighted.baccuracy(actual, predicted, w, adjust = FALSE, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
baccuracy(x, adjust = FALSE, na.rm = TRUE, ...)

baccuracy(...)

weighted.baccuracy(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels

predicted

A vector of `<factor>`-vector of length n, and k levels

adjust

A logical value (default: FALSE). If TRUE the metric is adjusted for random chance $\frac{1}{k}$.

na.rm

A logical values (default: TRUE). If TRUE calculation of the metric is based on valid classes.

...

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default

x

A confusion matrix created `cmatrix()`

Value

A numeric-vector of length 1

Calculation

The metric is calculated as follows,

$$\frac{\text{sensitivity} + \text{specificty}}{2}$$

See the `sensitivity()`- and/or `specificity()`-function for more details.

Examples

```
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
```

```r
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate the
# model
cat(
  "Balanced accuracy", baccuracy(
    actual    = actual,
    predicted = predicted
  ),

  "Balanced accuracy (weigthed)", weighted.baccuracy(
    actual    = actual,
    predicted = predicted,
    w         = iris$Petal.Length/mean(iris$Petal.Length)
  ),
  sep = "\n"
)
```

Cohen's kappa

ckappa.factor

R Documentation

Description

The `kappa()`-function computes Cohen's κ, a statistic that measures inter-rater agreement for categorical items between two vectors of predicted and observed `factor()` values. The `weighted.ckappa()` function computes the weighted κ-statistic.

If $\beta \neq 0$ the off-diagonals of the confusion matrix are penalized with a factor of $(y_+ - y_{i,-})^\beta$. See below for further details.

Usage

```
## S3 method for class 'factor'
ckappa(actual, predicted, beta = 0, ...)

## S3 method for class 'factor'
weighted.ckappa(actual, predicted, w, beta = 0, ...)

## S3 method for class 'cmatrix'
ckappa(x, beta = 0, ...)

ckappa(...)

weighted.ckappa(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels.

predicted

A vector of `<factor>`-vector of length n, and k levels.

beta

A `<numeric>` value of length 1 (default: 0). If set to a value different from zero, the off-diagonal confusion matrix will be penalized.

...

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default.

x

A confusion matrix created `cmatrix()`.

Value

If `micro` is NULL (the default), a named `<numeric>`-vector of length k

If `micro` is TRUE or FALSE, a `<numeric>`-vector of length 1

Calculation

$$\frac{\rho_p - \rho_e}{1 - \rho_e}$$

where ρ_p is the empirical probability of agreement between predicted and actual values, and ρ_e is the expected probability of agreement under random chance.

Examples

```r
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate model performance with
# Cohens Kappa statistic
cat(
  "Kappa", ckappa(
    actual    = actual,
    predicted = predicted
  ),
  "Kappa (penalized)", ckappa(
    actual    = actual,
    predicted = predicted,
    beta      = 2
  ),
  "Kappa (weigthed)", weighted.ckappa(
    actual    = actual,
    predicted = predicted,
    w         = iris$Petal.Length/mean(iris$Petal.Length)
  ),
  sep = "\n"
)
```

confusion matrix

cmatrix.factor

R Documentation

Description

The `cmatrix()`-function uses cross-classifying factors to build a confusion matrix of the counts at each combination of the factor levels. Each row of the matrix represents the actual factor levels, while each column represents the predicted factor levels.

Usage

```
## S3 method for class 'factor'
cmatrix(actual, predicted, ...)

## S3 method for class 'factor'
weighted.cmatrix(actual, predicted, w, ...)

cmatrix(...)

weighted.cmatrix(...)
```

Arguments

actual

A `<factor>`-vector of length n, and k levels.

predicted

A `<factor>`-vector of length n, and k levels.

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n (default: NULL) If passed it will return a weighted confusion matrix.

Value

A named k x k `<matrix>` of class

Dimensions

There is no robust defensive measure against misspecififying the confusion matrix. If the arguments are correctly specified, the resulting confusion matrix is on the form:

A (Predicted)

B (Predicted)

A (Actual)

Value

Value

B (Actual)

Value

Value

Examples

```
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) summarise performance
# in a confusion matrix

# 4.1) unweighted matrix
confusion_matrix <- cmatrix(
  actual    = actual,
  predicted = predicted
)

# 4.1.1) summarise matrix
summary(
  confusion_matrix
)

# 4.1.2) plot confusion
# matrix
plot(
  confusion_matrix
```

```
)

# 4.2) weighted matrix
confusion_matrix <- weighted.cmatrix(
  actual    = actual,
  predicted = predicted,
  w         = iris$Petal.Length/mean(iris$Petal.Length)
)

# 4.2.1) summarise matrix
summary(
  confusion_matrix
)

# 4.2.1) plot confusion
# matrix
plot(
  confusion_matrix
)
```

diagnostic odds ratio

dor.factor

R Documentation

Description

The `dor()`-function computes the Diagnostic Odds Ratio (DOR), a single indicator of test performance, between two vectors of predicted and observed `factor()` values. The `weighted.dor()` function computes the weighted diagnostic odds ratio.

When `aggregate = TRUE`, the function returns the micro-average DOR across all classes k. By default, it returns the class-wise DOR.

Usage

```r
## S3 method for class 'factor'
dor(actual, predicted, ...)

## S3 method for class 'factor'
weighted.dor(actual, predicted, w, ...)

## S3 method for class 'cmatrix'
dor(x, ...)

dor(...)

weighted.dor(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels.

predicted

A vector of `<factor>`-vector of length n, and k levels.

...

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default.

x

A confusion matrix created `cmatrix()`.

Value

If `micro` is NULL (the default), a named `<numeric>`-vector of length k

If `micro` is TRUE or FALSE, a `<numeric>`-vector of length 1

Calculation

The metric is calculated for each class k as follows,

$$\text{DOR}_k = \frac{\text{PLR}_k}{\text{NLR}_k}$$

Where PLR_k and NLR_k is the positive and negative likelihood ratio for class k, respectively. See `plr()` and `nlr()` for more details.

When `aggregate = TRUE`, the `micro`-average is calculated as,

$$\overline{\text{DOR}} = \frac{\overline{\text{PLR}_k}}{\overline{\text{NLR}_k}}$$

Where $\overline{\text{PLR}}$ and $\overline{\text{NLR}}$ is the micro-averaged is the positive and negative likelihood ratio, respectively.

Examples

```r
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)


# 4) evaluate model performance
# with Diagnostic Odds Ratio
cat("Diagnostic Odds Ratio", sep = "\n")
```

```r
dor(
  actual    = actual,
  predicted = predicted
)

cat("Diagnostic Odds Ratio (weighted)", sep = "\n")
weighted.dor(
  actual    = actual,
  predicted = predicted,
  w         = iris$Petal.Length/mean(iris$Petal.Length)
)
```

entropy

entropy.matrix

R Documentation

Description

The `entropy()` function calculates the Entropy of given probability distributions.

Usage

```
## S3 method for class 'matrix'
entropy(pk, dim = 0L, base = -1, ...)

## S3 method for class 'matrix'
relative.entropy(pk, qk, dim = 0L, base = -1, ...)

## S3 method for class 'matrix'
cross.entropy(pk, qk, dim = 0L, base = -1, ...)

entropy(...)

relative.entropy(...)

cross.entropy(...)
```

Arguments

pk

A $n \times k$ `<numeric>`-matrix of observed probabilities. The i-th row should sum to 1 (i.e., a valid probability distribution over the k classes). The first column corresponds to the first factor level in `actual`, the second column to the second factor level, and so on.

dim

An `<integer>` value of length 1 (Default: 0). Defines the dimensions of to calculate the entropy. 0: Total entropy, 1: row-wise, 2: column-wise

base

A `<numeric>` value of length 1 (Default: -1). The logarithmic base to use. Default value specifies natural logarithms.

...

Arguments passed into other methods

qk

A $n \times k$ `<numeric>`-matrix of predicted probabilities. The i-th row should sum to 1 (i.e., a valid probability distribution over the k classes). The first column corresponds to the first factor level in `actual`, the second column to the second factor level, and so on.

Value

A `<numeric>` value or vector:

A single `<numeric>` value (length 1) if `dim == 0`.

A `<numeric>` vector with length equal to the length of rows if `dim == 1`.

A `<numeric>` vector with length equal to the length of columns if `dim == 2`.

Calculation

Entropy:

$$H(pk) = -\sum_i pk_i \log(pk_i)$$

Cross Entropy:

$$H(pk, qk) = -\sum_i pk_i \log(qk_i)$$

Relative Entropy

$$D_{KL}(pk \parallel qk) = \sum_i pk_i \log\left(\frac{pk_i}{qk_i}\right)$$

Examples

```r
# 1) Define actual
# and observed probabilities

# 1.1) actual probabilies
pk <- matrix(
  cbind(1/2, 1/2),
  ncol = 2
)

# 1.2) observed (estimated) probabilites
qk <- matrix(
  cbind(9/10, 1/10),
  ncol = 2
)

# 2) calculate
# Entropy
cat(
  "Entropy", entropy(
    pk
  ),
  "Relative Entropy", relative.entropy(
    pk,
    qk
  ),
  "Cross Entropy", cross.entropy(
    pk,
    qk
  ),
  sep = "\n"
)
```

F-beta score

fbeta.factor

R Documentation

Description

The `fbeta()`-function computes the F_β score, the weighted harmonic mean of `precision()` and `recall()`, between two vectors of predicted and observed `factor()` values. The parameter β determines the weight of precision and recall in the combined score. The `weighted.fbeta()` function computes the weighted F_β score.

Usage

```
## S3 method for class 'factor'
fbeta(actual, predicted, beta = 1, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
weighted.fbeta(actual, predicted, w, beta = 1, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
fbeta(x, beta = 1, micro = NULL, na.rm = TRUE, ...)

fbeta(...)

weighted.fbeta(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels.

predicted

A vector of `<factor>`-vector of length n, and k levels.

beta

A `<numeric>` vector of length 1 (default: 1).

micro

A `<logical>`-value of length 1 (default: NULL). If TRUE it returns the micro average across all k classes, if FALSE it returns the macro average.

na.rm

A `<logical>` value of length 1 (default: TRUE). If TRUE, NA values are removed from the computation. This argument is only relevant when `micro != NULL`. When `na.rm = TRUE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(na.omit(c(1, 2, NA)))`. When `na.rm = FALSE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(c(1, 2, NA))`.

...

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default.

x

A confusion matrix created `cmatrix()`.

Value

If `micro` is NULL (the default), a named `<numeric>`-vector of length k

If `micro` is TRUE or FALSE, a `<numeric>`-vector of length 1

Calculation

The metric is calculated for each class k as follows,

$$(1 + \beta^2) \frac{\text{Precision}_k \cdot \text{Recall}_k}{(\beta^2 \cdot \text{Precision}_k) + \text{Recall}_k}$$

Where precision is $\frac{\#TP_k}{\#TP_k + \#FP_k}$ and recall (sensitivity) is $\frac{\#TP_k}{\#TP_k + \#FN_k}$, and β determines the weight of precision relative to recall.

Examples

```r
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate class-wise performance
# using F1-score

# 4.1) unweighted F1-score
```

```r
fbeta(
  actual    = actual,
  predicted = predicted,
  beta      = 1
)

# 4.2) weighted F1-score
weighted.fbeta(
  actual    = actual,
  predicted = predicted,
  w         = iris$Petal.Length/mean(iris$Petal.Length),
  beta      = 1
)

# 5) evaluate overall performance
# using micro-averaged F1-score
cat(
  "Micro-averaged F1-score", fbeta(
    actual    = actual,
    predicted = predicted,
    beta      = 1,
    micro     = TRUE
  ),
  "Micro-averaged F1-score (weighted)", weighted.fbeta(
    actual    = actual,
    predicted = predicted,
    w         = iris$Petal.Length/mean(iris$Petal.Length),
    beta      = 1,
    micro     = TRUE
  ),
  sep = "\n"
)
```

false discovery rate

fdr.factor

R Documentation

Description

The `fdr()`-function computes the false discovery rate (FDR), the proportion of false positives among the predicted positives, between two vectors of predicted and observed `factor()` values. The `weighted.fdr()` function computes the weighted false discovery rate.

Usage

```
## S3 method for class 'factor'
fdr(actual, predicted, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
weighted.fdr(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
fdr(x, micro = NULL, na.rm = TRUE, ...)

fdr(...)

weighted.fdr(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels.

predicted

A vector of `<factor>`-vector of length n, and k levels.

micro

A `<logical>`-value of length 1 (default: NULL). If TRUE it returns the micro average across all k classes, if FALSE it returns the macro average.

na.rm

A `<logical>` value of length 1 (default: TRUE). If TRUE, NA values are removed from the computation. This argument is only relevant when `micro != NULL`. When `na.rm = TRUE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(na.omit(c(1, 2, NA)))`. When `na.rm = FALSE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(c(1, 2, NA))`.

...

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default.

x

A confusion matrix created `cmatrix()`.

Value

If `micro` is NULL (the default), a named `<numeric>`-vector of length k

If `micro` is TRUE or FALSE, a `<numeric>`-vector of length 1

Calculation

The metric is calculated for each class k as follows,

$$\frac{\#FP_k}{\#TP_k + \#FP_k}$$

Where $\#TP_k$ and $\#FP_k$ is the number of true psotives and false positives, respectively, for each class k.

Examples

```
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate class-wise performance
# using False Discovery Rate

# 4.1) unweighted False Discovery Rate
```

```r
fdr(
  actual    = actual,
  predicted = predicted
)

# 4.2) weighted False Discovery Rate
weighted.fdr(
  actual    = actual,
  predicted = predicted,
  w         = iris$Petal.Length/mean(iris$Petal.Length)
)

# 5) evaluate overall performance
# using micro-averaged False Discovery Rate
cat(
  "Micro-averaged False Discovery Rate", fdr(
    actual    = actual,
    predicted = predicted,
    micro     = TRUE
  ),
  "Micro-averaged False Discovery Rate (weighted)", weighted.fdr(
    actual    = actual,
    predicted = predicted,
    w         = iris$Petal.Length/mean(iris$Petal.Length),
    micro     = TRUE
  ),
  sep = "\n"
)
```

false omission rate

fer.factor

R Documentation

Description

The `fer()`-function computes the false omission rate (FOR), the proportion of false negatives among the predicted negatives, between two vectors of predicted and observed `factor()` values. The `weighted.fer()` function computes the weighted false omission rate.

Usage

```
## S3 method for class 'factor'
fer(actual, predicted, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
weighted.fer(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
fer(x, micro = NULL, na.rm = TRUE, ...)

fer(...)

weighted.fer(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels.

predicted

A vector of `<factor>`-vector of length n, and k levels.

micro

A `<logical>`-value of length 1 (default: NULL). If TRUE it returns the micro average across all k classes, if FALSE it returns the macro average.

na.rm

A `<logical>` value of length 1 (default: TRUE). If TRUE, NA values are removed from the computation. This argument is only relevant when `micro != NULL`. When `na.rm = TRUE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(na.omit(c(1, 2, NA)))`. When `na.rm = FALSE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(c(1, 2, NA))`.

...

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default.

x

A confusion matrix created `cmatrix()`.

Value

If `micro` is NULL (the default), a named `<numeric>`-vector of length k

If `micro` is TRUE or FALSE, a `<numeric>`-vector of length 1

Calculation

The metric is calculated for each class k as follows,

$$\frac{\#FN_k}{\#FN_k + \#TN_k}$$

Where $\#FN_k$ and $\#TN_k$ are the number of false negatives and true negatives, respectively, for each class k.

Examples

```r
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate class-wise performance
# using False Omission Rate

# 4.1) unweighted False Omission Rate
```

```
fer(
  actual    = actual,
  predicted = predicted
)

# 4.2) weighted False Omission Rate
weighted.fer(
  actual    = actual,
  predicted = predicted,
  w         = iris$Petal.Length/mean(iris$Petal.Length)
)

# 5) evaluate overall performance
# using micro-averaged False Omission Rate
cat(
  "Micro-averaged False Omission Rate", fer(
    actual    = actual,
    predicted = predicted,
    micro     = TRUE
  ),
  "Micro-averaged False Omission Rate (weighted)", weighted.fer(
    actual    = actual,
    predicted = predicted,
    w         = iris$Petal.Length/mean(iris$Petal.Length),
    micro     = TRUE
  ),
  sep = "\n"
)
```

Fowlkes-Mallows index

fmi.factor

R Documentation

Description

The `fmi()`-function computes the Fowlkes-Mallows Index (FMI), a measure of the similarity between two sets of clusterings, between two vectors of predicted and observed `factor()` values.

Usage

```
## S3 method for class 'factor'
fmi(actual, predicted, ...)

## S3 method for class 'cmatrix'
fmi(x, ...)

fmi(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels

predicted

A vector of `<factor>`-vector of length n, and k levels

...

Arguments passed into other methods

x

A confusion matrix created `cmatrix()`

Value

A `<numeric>`-vector of length 1

Calculation

The metric is calculated for each class k as follows,

$$\sqrt{\frac{\#TP_k}{\#TP_k + \#FP_k} \times \frac{\#TP_k}{\#TP_k + \#FN_k}}$$

Where $\#TP_k$, $\#FP_k$, and $\#FN_k$ represent the number of true positives, false positives, and false negatives for each class k, respectively.

Examples

```
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
```

```r
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate model performance
# using Fowlkes Mallows Index
cat(
  "Fowlkes Mallows Index", fmi(
  actual    = actual,
  predicted = predicted
  ),
  sep = "\n"
)
```

false positive rate

fpr.factor

R Documentation

Description

The `fpr()`-function computes the False Positive Rate (FPR), also known as the fall-out (`fallout()`), between two vectors of predicted and observed `factor()` values. The `weighted.fpr()` function computes the weighted false positive rate.

Usage

```
## S3 method for class 'factor'
fpr(actual, predicted, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
weighted.fpr(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
fpr(x, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
fallout(actual, predicted, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
weighted.fallout(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
fallout(x, micro = NULL, na.rm = TRUE, ...)

fpr(...)

fallout(...)
```

```
weighted.fpr(...)

weighted.fallout(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels.

predicted

A vector of `<factor>`-vector of length n, and k levels.

micro

A `<logical>`-value of length 1 (default: NULL). If TRUE it returns the micro average across all k classes, if FALSE it returns the macro average.

na.rm

A `<logical>` value of length 1 (default: TRUE). If TRUE, NA values are removed from the computation. This argument is only relevant when `micro != NULL`. When `na.rm = TRUE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(na.omit(c(1, 2, NA)))`. When `na.rm = FALSE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(c(1, 2, NA))`.

…

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default.

x

A confusion matrix created `cmatrix()`.

Value

If `micro` is NULL (the default), a named `<numeric>`-vector of length k

If `micro` is TRUE or FALSE, a `<numeric>`-vector of length 1

Calculation

The metric is calculated for each class k as follows,

$$\frac{\#FP_k}{\#FP_k + \#TN_k}$$

Where $\#FP_k$ and $\#TN_k$ represent the number of false positives and true negatives, respectively, for each class k.

Examples

```r
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
```

```r
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate class-wise performance
# using False Positive Rate

# 4.1) unweighted False Positive Rate
fpr(
  actual    = actual,
  predicted = predicted
)

# 4.2) weighted False Positive Rate
weighted.fpr(
  actual    = actual,
  predicted = predicted,
  w         = iris$Petal.Length/mean(iris$Petal.Length)
)

# 5) evaluate overall performance
# using micro-averaged False Positive Rate
cat(
  "Micro-averaged False Positive Rate", fpr(
    actual    = actual,
    predicted = predicted,
    micro     = TRUE
  ),
  "Micro-averaged False Positive Rate (weighted)", weighted.fpr(
    actual    = actual,
    predicted = predicted,
    w         = iris$Petal.Length/mean(iris$Petal.Length),
    micro     = TRUE
  ),
  sep = "\n"
)
```

Jaccard score

jaccard.factor

R Documentation

Description

The `jaccard()`-function computes the Jaccard Index, also known as the Intersection over Union, between two vectors of predicted and observed `factor()` values. The `weighted.jaccard()` function computes the weighted Jaccard Index.

Usage

```
## S3 method for class 'factor'
jaccard(actual, predicted, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
weighted.jaccard(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
jaccard(x, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
csi(actual, predicted, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
weighted.csi(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
csi(x, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
tscore(actual, predicted, micro = NULL, na.rm = TRUE, ...)
```

```
## S3 method for class 'factor'
weighted.tscore(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
tscore(x, micro = NULL, na.rm = TRUE, ...)

jaccard(...)

csi(...)

tscore(...)

weighted.jaccard(...)

weighted.csi(...)

weighted.tscore(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels.

predicted

A vector of `<factor>`-vector of length n, and k levels.

micro

A `<logical>`-value of length 1 (default: NULL). If TRUE it returns the micro average across all k classes, if FALSE it returns the macro average.

na.rm

A `<logical>` value of length 1 (default: TRUE). If TRUE, NA values are removed from the computation. This argument is only relevant when `micro != NULL`. When `na.rm = TRUE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(na.omit(c(1, 2, NA)))`. When `na.rm = FALSE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(c(1, 2, NA))`.

...

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default.

x

A confusion matrix created `cmatrix()`.

Value

If `micro` is NULL (the default), a named `<numeric>`-vector of length k

If `micro` is TRUE or FALSE, a `<numeric>`-vector of length 1

Calculation

The metric is calculated for each class k as follows,

$$\frac{\#TP_k}{\#TP_k + \#FP_k + \#FN_k}$$

Where $\#TP_k$, $\#FP_k$, and $\#FN_k$ represent the number of true positives, false positives, and false negatives for each class k, respectively.

Examples

```
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)
```

```
# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate class-wise performance
# using Jaccard Index

# 4.1) unweighted Jaccard Index
jaccard(
  actual    = actual,
  predicted = predicted
)

# 4.2) weighted Jaccard Index
weighted.jaccard(
  actual    = actual,
  predicted = predicted,
  w         = iris$Petal.Length/mean(iris$Petal.Length)
)

# 5) evaluate overall performance
# using micro-averaged Jaccard Index
cat(
  "Micro-averaged Jaccard Index", jaccard(
    actual    = actual,
    predicted = predicted,
    micro     = TRUE
  ),
  "Micro-averaged Jaccard Index (weighted)", weighted.jaccard(
    actual    = actual,
    predicted = predicted,
    w         = iris$Petal.Length/mean(iris$Petal.Length),
    micro     = TRUE
  ),
  sep = "\n"
)
```

log loss

logloss.factor

R Documentation

Description

The `logloss()` function computes the Log Loss between observed classes (as a `<factor>`) and their predicted probability distributions (a `<numeric>` matrix). The `weighted.logloss()` function is the weighted version, applying observation-specific weights.

Usage

```
## S3 method for class 'factor'
logloss(actual, qk, normalize = TRUE, ...)

## S3 method for class 'factor'
weighted.logloss(actual, qk, w, normalize = TRUE, ...)

logloss(...)

weighted.logloss(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels

qk

A $n \times k$ `<numeric>`-matrix of predicted probabilities. The i-th row should sum to 1 (i.e., a valid probability distribution over the k classes). The first column corresponds to the first factor level in `actual`, the second column to the second factor level, and so on.

normalize

A `<logical>`-value (default: TRUE). If TRUE, the mean cross-entropy across all observations is returned; otherwise, the sum of cross-entropies is returned.

…

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default

Value

A `<numeric>`-vector of length 1

Calculation

$$H(p, qk) = -\sum_i \sum_j y_{ij} \log_2(qk_{ij})$$

where:

y_{ij} is the `actual`-values, where $y_{ij} = 1$ if the `i`-th sample belongs to class `j`, and 0 otherwise.

qk_{ij} is the estimated probability for the `i`-th sample belonging to class `j`.

Examples

```
# 1) Recode the iris data set to a binary classification problem
#    Here, the positive class ("Virginica") is coded as 1,
#    and the rest ("Others") is coded as 0.
iris$species_num <- as.numeric(iris$Species == "virginica")

# 2) Fit a logistic regression model predicting species_num from Sepal.Length &amp; Sepal.
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(link = "logit")
)

# 3) Generate predicted classes: "Virginica" vs. "Others"
```

```r
predicted <- factor(
  as.numeric(predict(model, type = "response") >` 0.5),
  levels = c(1, 0),
  labels = c("Virginica", "Others")
)

# 3.1) Generate actual classes
actual <- factor(
  x      = iris$species_num,
  levels = c(1, 0),
  labels = c("Virginica", "Others")
)

# For Log Loss, we need predicted probabilities for each class.
# Since it's a binary model, we create a 2-column matrix:
#   1st column = P("Virginica")
#   2nd column = P("Others") = 1 - P("Virginica")
predicted_probs <- predict(model, type = "response")
qk_matrix <- cbind(predicted_probs, 1 - predicted_probs)

# 4) Evaluate unweighted Log Loss
#    'logloss' takes (actual, qk_matrix, normalize=TRUE/FALSE).
#    The factor 'actual' must have the positive class (Virginica) as its first level.
unweighted_LogLoss <- logloss(
  actual    = actual,           # factor
  qk        = qk_matrix,        # numeric matrix of probabilities
  normalize = TRUE              # normalize = TRUE
)

# 5) Evaluate weighted Log Loss
#    We introduce a weight vector, for example:
weights <- iris$Petal.Length / mean(iris$Petal.Length)
weighted_LogLoss <- weighted.logloss(
  actual    = actual,
  qk  = qk_matrix,
  w         = weights,
  normalize = TRUE
)

# 6) Print Results
cat(
  "Unweighted Log Loss:", unweighted_LogLoss,
  "Weighted Log Loss:", weighted_LogLoss,
  sep = "\n"
)
```

mean percentage error

mpe.numeric

R Documentation

Description

The `mpe()`-function computes the mean percentage error between the observed and predicted `<numeric>` vectors. The `weighted.mpe()` function computes the weighted mean percentage error.

Usage

```
## S3 method for class 'numeric'
mpe(actual, predicted, ...)

## S3 method for class 'numeric'
weighted.mpe(actual, predicted, w, ...)

mpe(...)

weighted.mpe(...)
```

Arguments

actual

A `<numeric>`-vector of length n. The observed (continuous) response variable.

predicted

A `<numeric>`-vector of length n. The estimated (continuous) response variable.

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. The weight assigned to each observation in the data.

Value

A `<numeric>` vector of length 1.

Calculation

The metric is calculated as,

$$\frac{1}{n} \sum_{i}^{n} \frac{y_i - \upsilon_i}{y_i}$$

Where y_i and υ_i are the `actual` and `predicted` values respectively.

Examples

```
# 1) fit a linear
# regression
model <- lm(
  mpg ~ .,
  data = mtcars
)

# 1.1) define actual
# and predicted values
# to measure performance
actual    <- mtcars$mpg
predicted <- fitted(model)

# 2) evaluate in-sample model
# performance using Mean Percentage Error (MPE)
cat(
  "Mean Percentage Error", mpe(
    actual    = actual,
    predicted = predicted,
  ),
  "Mean Percentage Error (weighted)", weighted.mpe(
    actual    = actual,
    predicted = predicted,
    w         = mtcars$mpg/mean(mtcars$mpg)
  ),
  sep = "\n"
)
```

negative likelihood ratio

nlr.factor

R Documentation

Description

The `nlr()`-function computes the negative likelihood ratio, also known as the likelihood ratio for negative results, between two vectors of predicted and observed `factor()` values. The `weighted.nlr()` function computes the weighted negative likelihood ratio.

Usage

```
## S3 method for class 'factor'
nlr(actual, predicted, ...)

## S3 method for class 'factor'
weighted.nlr(actual, predicted, w, ...)

## S3 method for class 'cmatrix'
nlr(x, ...)

nlr(...)

weighted.nlr(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels.

predicted

A vector of `<factor>`-vector of length n, and k levels.

...

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default.

x

A confusion matrix created `cmatrix()`.

Value

If `micro` is NULL (the default), a named `<numeric>`-vector of length k

If `micro` is TRUE or FALSE, a `<numeric>`-vector of length 1

Calculation

The metric is calculated for each class k as follows,

$$\frac{1 - \text{Sensitivity}_k}{\text{Specificity}_k}$$

Where sensitivity (or true positive rate) is calculated as $\frac{\#TP_k}{\#TP_k + \#FN_k}$ and specificity (or true negative rate) is calculated as $\frac{\#TN_k}{\#TN_k + \#FP_k}$.

Examples

```r
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate model performance
# with class-wise negative likelihood ratios
cat("Negative Likelihood Ratio", sep = "\n")
nlr(
  actual    = actual,
  predicted = predicted
)

cat("Negative Likelihood Ratio (weighted)", sep = "\n")
weighted.nlr(
  actual    = actual,
  predicted = predicted,
  w         = iris$Petal.Length/mean(iris$Petal.Length)
)
```

negative predictive value

npv.factor

R Documentation

Description

The `npv()`-function computes the negative predictive value, also known as the True Negative Predictive Value, between two vectors of predicted and observed `factor()` values. The `weighted.npv()` function computes the weighted negative predictive value.

Usage

```
## S3 method for class 'factor'
npv(actual, predicted, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
weighted.npv(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
npv(x, micro = NULL, na.rm = TRUE, ...)

npv(...)

weighted.npv(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels.

predicted

A vector of `<factor>`-vector of length n, and k levels.

micro

A `<logical>`-value of length 1 (default: NULL). If TRUE it returns the micro average across all k classes, if FALSE it returns the macro average.

na.rm

A `<logical>` value of length 1 (default: TRUE). If TRUE, NA values are removed from the computation. This argument is only relevant when `micro != NULL`. When `na.rm = TRUE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(na.omit(c(1, 2, NA)))`. When `na.rm = FALSE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(c(1, 2, NA))`.

...

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default.

x

A confusion matrix created `cmatrix()`.

Value

If `micro` is NULL (the default), a named `<numeric>`-vector of length k

If `micro` is TRUE or FALSE, a `<numeric>`-vector of length 1

Calculation

The metric is calculated for each class k as follows,

$$\frac{\#TN_k}{\#TN_k + \#FN_k}$$

Where $\#TN_k$ and $\#FN_k$ are the number of true negatives and false negatives, respectively, for each class k.

Examples

```
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate class-wise performance
# using Negative Predictive Value

# 4.1) unweighted Negative Predictive Value
```

```r
npv(
  actual    = actual,
  predicted = predicted
)

# 4.2) weighted Negative Predictive Value
weighted.npv(
  actual    = actual,
  predicted = predicted,
  w         = iris$Petal.Length/mean(iris$Petal.Length)
)

# 5) evaluate overall performance
# using micro-averaged Negative Predictive Value
cat(
  "Micro-averaged Negative Predictive Value", npv(
    actual    = actual,
    predicted = predicted,
    micro     = TRUE
  ),
  "Micro-averaged Negative Predictive Value (weighted)", weighted.npv(
    actual    = actual,
    predicted = predicted,
    w         = iris$Petal.Length/mean(iris$Petal.Length),
    micro     = TRUE
  ),
  sep = "\n"
)
```

positive likelihood ratio

plr.factor

R Documentation

Description

The `plr()`-function computes the positive likelihood ratio, also known as the likelihood ratio for positive results, between two vectors of predicted and observed `factor()` values. The `weighted.plr()` function computes the weighted positive likelihood ratio.

Usage

```
## S3 method for class 'factor'
plr(actual, predicted, ...)

## S3 method for class 'factor'
weighted.plr(actual, predicted, w, ...)

## S3 method for class 'cmatrix'
plr(x, ...)

plr(...)

weighted.plr(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels.

predicted

A vector of `<factor>`-vector of length n, and k levels.

...

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default.

x

A confusion matrix created `cmatrix()`.

Value

If `micro` is NULL (the default), a named `<numeric>`-vector of length k

If `micro` is TRUE or FALSE, a `<numeric>`-vector of length 1

Calculation

The metric is calculated for each class k as follows,

$$\frac{\text{Sensitivity}_k}{1 - \text{Specificity}_k}$$

Where sensitivity (or true positive rate) is calculated as $\frac{\#TP_k}{\#TP_k + \#FN_k}$ and specificity (or true negative rate) is calculated as $\frac{\#TN_k}{\#TN_k + \#FP_k}$.

When `aggregate = TRUE`, the `micro`-average is calculated,

$$\frac{\sum_{k=1}^{k} \text{Sensitivity}_k}{1 - \sum_{k=1}^{k} \text{Specificity}_k}$$

Examples

```r
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)
```

```r
# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate model performance
# with class-wise positive likelihood ratios
cat("Positive Likelihood Ratio", sep = "\n")
plr(
  actual    = actual,
  predicted = predicted
)

cat("Positive Likelihood Ratio (weighted)", sep = "\n")
weighted.plr(
  actual    = actual,
  predicted = predicted,
  w         = iris$Petal.Length/mean(iris$Petal.Length)
```

```
)
```

Precision-Recall curve

prROC.factor

R Documentation

Description

The `prROC()`-function computes the `precision()` and `recall()` at thresholds provided by the *response*- or *thresholds*-vector. The function constructs a `data.frame()` grouped by k-classes where each class is treated as a binary classification problem.

Usage

```
## S3 method for class 'factor'
prROC(actual, response, thresholds = NULL, ...)

## S3 method for class 'factor'
weighted.prROC(actual, response, w, thresholds = NULL, ...)

prROC(...)

weighted.prROC(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels.

response

A `<numeric>`-vector of length n. The estimated response probabilities.

thresholds

An optional `<numeric>`-vector of non-zero length (default: NULL).

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. NULL by default.

Value

A data.frame on the following form,

`threshold`

`<numeric>` Thresholds used to determine `recall()` and `precision()`

`level`

`<character>` The level of the actual `<factor>`

`label`

`<character>` The levels of the actual `<factor>`

`recall`

`<numeric>` The recall

`precision`

`<numeric>` The precision

Examples

```r
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
response <- predict(model, type = "response")

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)



# 4) generate reciever
# operator characteristics
roc <- prROC(
  actual   = actual,
  response = response
)


# 5) plot by species
plot(roc)

# 5.1) summarise
summary(roc)

# 6) provide custom
# threholds
roc <- prROC(
  actual     = actual,
  response   = response,
  thresholds = seq(0, 1, length.out = 4)
)
```

```
# 5) plot by species
plot(roc)
```

precision

precision.factor

R Documentation

Description

The `precision()`-function computes the precision, also known as the positive predictive value (PPV), between two vectors of predicted and observed `factor()` values. The `weighted.precision()` function computes the weighted precision.

Usage

```
## S3 method for class 'factor'
precision(actual, predicted, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
weighted.precision(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
precision(x, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
ppv(actual, predicted, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
weighted.ppv(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
ppv(x, micro = NULL, na.rm = TRUE, ...)

precision(...)

weighted.precision(...)
```

```
ppv(...)

weighted.ppv(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels.

predicted

A vector of `<factor>`-vector of length n, and k levels.

micro

A `<logical>`-value of length 1 (default: NULL). If TRUE it returns the micro average across all k classes, if FALSE it returns the macro average.

na.rm

A `<logical>` value of length 1 (default: TRUE). If TRUE, NA values are removed from the computation. This argument is only relevant when `micro != NULL`. When `na.rm = TRUE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(na.omit(c(1, 2, NA)))`. When `na.rm = FALSE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(c(1, 2, NA))`.

…

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default.

x

A confusion matrix created `cmatrix()`.

Value

If `micro` is NULL (the default), a named `<numeric>`-vector of length k

If `micro` is TRUE or FALSE, a `<numeric>`-vector of length 1

Calculation

The metric is calculated for each class k as follows,

$$\frac{\#TP_k}{\#TP_k + \#FP_k}$$

Where $\#TP_k$ and $\#FP_k$ are the number of true positives and false positives, respectively, for each class k.

Examples

```r
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
```

```r
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate class-wise performance
# using Precision

# 4.1) unweighted Precision
precision(
  actual    = actual,
  predicted = predicted
)

# 4.2) weighted Precision
weighted.precision(
  actual    = actual,
  predicted = predicted,
  w         = iris$Petal.Length/mean(iris$Petal.Length)
)

# 5) evaluate overall performance
# using micro-averaged Precision
cat(
  "Micro-averaged Precision", precision(
    actual    = actual,
    predicted = predicted,
    micro     = TRUE
  ),
  "Micro-averaged Precision (weighted)", weighted.precision(
    actual    = actual,
    predicted = predicted,
    w         = iris$Petal.Length/mean(iris$Petal.Length),
    micro     = TRUE
  ),
  sep = "\n"
)
```

recall

recall.factor

R Documentation

Description

The `recall()`-function computes the recall, also known as sensitivity or the True Positive Rate (TPR), between two vectors of predicted and observed `factor()` values. The `weighted.recall()` function computes the weighted recall.

Usage

```
## S3 method for class 'factor'
recall(actual, predicted, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
weighted.recall(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
recall(x, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
sensitivity(actual, predicted, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
weighted.sensitivity(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
sensitivity(x, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'factor'
tpr(actual, predicted, micro = NULL, na.rm = TRUE, ...)
```

```r
## S3 method for class 'factor'
weighted.tpr(actual, predicted, w, micro = NULL, na.rm = TRUE, ...)

## S3 method for class 'cmatrix'
tpr(x, micro = NULL, na.rm = TRUE, ...)

recall(...)

sensitivity(...)

tpr(...)

weighted.recall(...)

weighted.sensitivity(...)

weighted.tpr(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels.

predicted

A vector of `<factor>`-vector of length n, and k levels.

micro

A `<logical>`-value of length 1 (default: NULL). If TRUE it returns the micro average across all k classes, if FALSE it returns the macro average.

na.rm

A `<logical>` value of length 1 (default: TRUE). If TRUE, NA values are removed from the computation. This argument is only relevant when `micro != NULL`. When `na.rm = TRUE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(na.omit(c(1, 2, NA)))`. When `na.rm = FALSE`, the computation corresponds to `sum(c(1, 2, NA), na.rm = TRUE) / length(c(1, 2, NA))`.

...

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default.

x

A confusion matrix created `cmatrix()`.

Value

If `micro` is NULL (the default), a named `<numeric>`-vector of length k

If `micro` is TRUE or FALSE, a `<numeric>`-vector of length 1

Calculation

The metric is calculated for each class k as follows,

$$\frac{\#TP_k}{\#TP_k + \#FN_k}$$

Where $\#TP_k$ and $\#FN_k$ is the number of true positives and false negatives, respectively, for each class k.

Examples

```r
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)
```

```
# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate class-wise performance
# using Recall

# 4.1) unweighted Recall
recall(
  actual    = actual,
  predicted = predicted
)

# 4.2) weighted Recall
weighted.recall(
  actual    = actual,
  predicted = predicted,
  w         = iris$Petal.Length/mean(iris$Petal.Length)
)

# 5) evaluate overall performance
# using micro-averaged Recall
cat(
  "Micro-averaged Recall", recall(
    actual    = actual,
    predicted = predicted,
    micro     = TRUE
  ),
  "Micro-averaged Recall (weighted)", weighted.recall(
    actual    = actual,
    predicted = predicted,
    w         = iris$Petal.Length/mean(iris$Petal.Length),
    micro     = TRUE
  ),
  sep = "\n"
)
```

coefficient of determination

rsq.numeric

R Documentation

Description

The `rsq()`-function calculates the R^2, the coefficient of determination, between the ovserved and predicted `<numeric>` vectors. By default `rsq()` returns the unadjusted R^2. For adjusted R^2 set $k = \kappa - 1$, where κ is the number of parameters.

Usage

```
## S3 method for class 'numeric'
rsq(actual, predicted, k = 0, ...)

## S3 method for class 'numeric'
weighted.rsq(actual, predicted, w, k = 0, ...)

rsq(...)

weighted.rsq(...)
```

Arguments

actual

A `<numeric>`-vector of length n. The observed (continuous) response variable.

predicted

A `<numeric>`-vector of length n. The estimated (continuous) response variable.

k

A `<numeric>`-vector of length 1 (default: 0). If $k >$` 0 the function returns the adjusted R^2.

...

Arguments passed into other methods.

w

A `<numeric>`-vector of length n. The weight assigned to each observation in the data.

Value

A `<numeric>` vector of length 1.

Calculation

The metric is calculated as follows,

$$R^2 = 1 - \frac{\text{SSE}}{\text{SST}} \frac{n-1}{n-(k+1)}$$

Where SSE is the sum of squared errors, SST is total sum of squared errors, n is the number of observations, and k is the number of non-constant parameters.

Examples

```r
# 1) fit a linear
# regression
model <- lm(
  mpg ~ .,
  data = mtcars
)

# 1.1) define actual
# and predicted values
# to measure in-sample performance
actual    <- mtcars$mpg
predicted <- fitted(model)

# 2) calculate performance
# using R squared adjusted and
# unadjused for features
cat(
  "Rsq", rsq(
    actual    = actual,
    predicted = fitted(model)
  ),
  "Rsq (Adjusted)", rsq(
    actual    = actual,
    predicted = fitted(model),
    k = ncol(model.matrix(model)) - 1
  ),
  sep = "\n"
)
```

zero-one loss

zerooneloss.factor

R Documentation

Description

The `zerooneloss()`-function computes the zero-one Loss, a classification loss function that calculates the proportion of misclassified instances between two vectors of predicted and observed `factor()` values. The `weighted.zerooneloss()` function computes the weighted zero-one loss.

Usage

```
## S3 method for class 'factor'
zerooneloss(actual, predicted, ...)

## S3 method for class 'factor'
weighted.zerooneloss(actual, predicted, w, ...)

## S3 method for class 'cmatrix'
zerooneloss(x, ...)

zerooneloss(...)

weighted.zerooneloss(...)
```

Arguments

actual

A vector of `<factor>`- of length n, and k levels

predicted

A vector of `<factor>`-vector of length n, and k levels

...

Arguments passed into other methods

w

A `<numeric>`-vector of length n. NULL by default

x

A confusion matrix created `cmatrix()`

Value

A `<numeric>`-vector of length 1

Calculation

The metric is calculated as follows,

$$\frac{\#FP + \#FN}{\#TP + \#TN + \#FP + \#FN}$$

Where $\#TP$, $\#TN$, $\#FP$, and $\#FN$ represent the true positives, true negatives, false positives, and false negatives, respectively.

Examples

```r
# 1) recode Iris
# to binary classification
# problem
iris$species_num <- as.numeric(
  iris$Species == "virginica"
)

# 2) fit the logistic
# regression
model <- glm(
  formula = species_num ~ Sepal.Length + Sepal.Width,
  data    = iris,
  family  = binomial(
    link = "logit"
  )
)

# 3) generate predicted
# classes
predicted <- factor(
  as.numeric(
    predict(model, type = "response") >` 0.5
  ),
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 3.1) generate actual
# classes
actual <- factor(
  x = iris$species_num,
  levels = c(1,0),
  labels = c("Virginica", "Others")
)

# 4) evaluate model
# performance using Zero-One Loss
cat(
  "Zero-One Loss", zerooneloss(
    actual    = actual,
    predicted = predicted
  ),
  "Zero-One Loss (weigthed)", weighted.zerooneloss(
    actual    = actual,
    predicted = predicted,
    w         = iris$Petal.Length/mean(iris$Petal.Length)
  ),
  sep = "\n"
)
```

4 OpenMP

{SLmetrics} supports parallelization through OpenMP. In this section this functionality is introduced.

4.1 Enabling/Disabling OpenMP

OpenMP is disabled by `default`, but can be enabled as follows:

```
SLmetrics::setUseOpenMP(TRUE)
```

```
#> OpenMP usage set to: enabled
```

And disabled as follows:

```
SLmetrics::setUseOpenMP(FALSE)
```

```
#> OpenMP usage set to: disabled
```

By `default` all cores are used. To control the amount of cores, see the following code:

```
SLmetrics::setNumberThreads(3)
```

```
#> Number of threads set to: 3
```

To use all cores:

```
SLmetrics::setNumberThreads(-1)
```

```
#> Number of threads set to: All (4 threads)
```

4.2 Benchmarking OpenMP

```r
# 1) set seed for reproducibility
set.seed(1903)

# 2) create classification
# problem
fct_actual <- create_factor()
fct_predicted <- create_factor()
```

```r
SLmetrics::setUseOpenMP(TRUE)
```

```
#> OpenMP usage set to: enabled
```

```r
benchmark(
    `With OpenMP` = SLmetrics::cmatrix(fct_actual, fct_predicted)
)
```

```
#> # A tibble: 1 x 4
#>   expression  execution_time memory_usage gc_calls
#>   <fct>             <bch:tm>    <bch:byt>    <dbl>
#> 1 With OpenMP         3.85ms           0B        0
```

```r
SLmetrics::setUseOpenMP(FALSE)
```

```
#> OpenMP usage set to: disabled
```

```r
benchmark(
    `Wihtout OpenMP` = SLmetrics::cmatrix(fct_actual, fct_predicted)
)
```

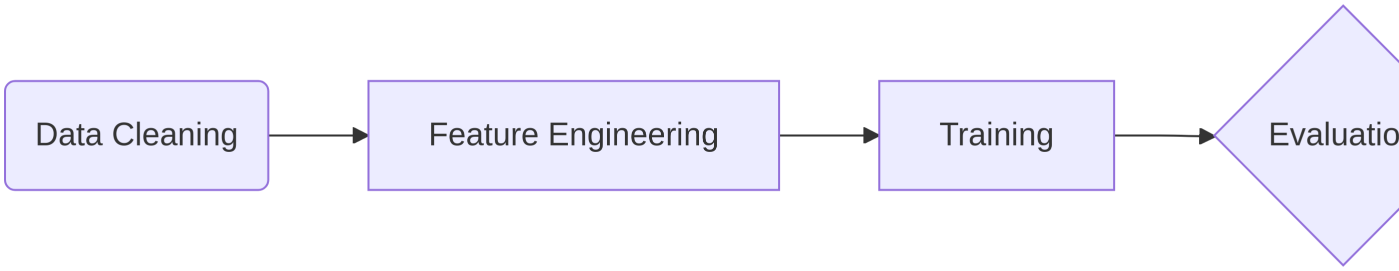
```
#> # A tibble: 1 x 4
#>   expression     execution_time memory_usage gc_calls
#>   <fct>                <bch:tm>    <bch:byt>    <dbl>
#> 1 Wihtout OpenMP         8.61ms           0B        0
```

5 Garbage in, garbage out

This section examines the underlying assumptions in {SLmetrics}, and how it may affect your pipeline if you decide adopt it.

5.1 Implicit assumptions

All evaluation functions in {SLmetrics} assumes that end-user follows the typical AI/ML workflow:



The implications of this assumption is two-fold:

- There is no handling of **missing data** in input variables
- There is no **validity check** of inputs

Hence, the implicit assumption is that the end-user has a high degree of control over the training process and an understanding of `R` beyond beginner-level. See, for example, the following code:

```r
# 1) define values
actual    <- c(-1.2, 1.3, 2.6, 3)
predicted <- rev(actual)

# 2) evaluate with RMSLE
SLmetrics::rmsle(
    actual,
    predicted
```

```
)
```

```
#> [1] NaN
```

The `actual`- and `predicted`-vector contains negative values, and is being passed to the root mean squared logarithmic error (`rmsle()`)-function. It returns `NaN` without any warnings. The same action in using base `R` would lead to verbose errors:

```
mean(log(actual))
```

```
#> Warning in log(actual): NaNs produced
```

```
#> [1] NaN
```

5.2 Undefined behavior

! Important

Do **NOT** run the chunks in this section in an `R`-session where you have important work, as your session *will* crash.

{SLmetrics} uses pointer arithmetics via `C++` which, contrary to usual practice in `R`, performs computations on memory addresses rather than the object itself. If the memory address is ill-defined, which can occur in cases where values lack valid binary representations for the operations being performed, undefined behavior[1] follows and *will* crash your `R`-session. See this code:

```
# 1) define values
actual <- factor(c(NA, "A", "B", "A"))
predicted <- rev(actual)

# 2) pass into
# cmatrix
SLmetrics::cmatrix(
    actual,
    predicted
```

[1]Undefined behavior refers to program operations that are not prescribed by the language specification, leading to unpredictable results or crashes.

```
)
#> address 0x5946ff482178, cause 'memory not mapped'
#> An irrecoverable exception occurred. R is aborting now ...
```

This is not something that can prevented with, say, `try()`, as the error is undefined. See this SO-post for details.

5.3 Edge cases

There are cases, where it can be hard to predict what will happen when passing a given set of actual and predicted classes. Especially if the input is too large, and it becomes inefficient to check these every iteration. In such cases {SLmetrics} does help. See for example the following code:

```
# 1) define values
actual <- factor(
    sample(letters[1:3],size = 1e7, replace = TRUE, prob = c(0.5, 0.5, 0)),
    levels = letters[1:3]
    )
predicted <- rev(actual)

# 2) pass into
# cmatrix
SLmetrics::fbeta(
    actual,
    predicted
)
```

```
#>         a         b         c
#> 0.5002649 0.4998854       NaN
```

One class, `c`, is never predicted, nor is it present in the actual labels - therefore, by construction, the value is `NaN` as there is division by zero. During aggregation to `micro` or `macro` averages these are being handled according to `na.rm`. See below:

```
# 1) macro average
SLmetrics::fbeta(
    actual,
    predicted,
    micro = FALSE,
    na.rm = TRUE
)
```

```
#> [1] 0.5000751
```

```
# 2) macro average
SLmetrics::fbeta(
    actual,
    predicted,
    micro = FALSE,
    na.rm = FALSE
)
```

```
#> [1] 0.3333834
```

```
# 1) define values
actual    <- c(-1.2, 1.3, 2.6, 3)
predicted <- rev(actual)

# 2) evaluate with RMSLE
try(
    RMSLE(
    actual,
    predicted
    )
)
```

```
#> Error in RMSLE(actual, predicted) : could not find function "RMSLE"
```

In these cases, there is no undefined behaviour or exploding `R` sessions as all of this is handled internally.

5.4 Staying "safe"

To avoid undefined behavior when passing ill-defined input one option is to write a wrapper function, or using existing infrastructure. Below is an example of a wrapper function:

```r
# 1) RMSLE
confusion_matrix <- function(
    actual,
    predicted) {

        if (any(is.na(actual))) {
            stop("`actual` contains missing values")
        }

        if (any(is.na(predicted))) {
            stop("`predicted` contains missing values")
        }

        SLmetrics::cmatrix(
            actual,
            predicted
        )

}
```

```r
# 1) define values
actual <- factor(c(NA, "A", "B", "A", "B"))
predicted <- rev(actual)

# 2)
try(
    confusion_matrix(
    actual,
    predicted
    )
)
```

```
#> Error in confusion_matrix(actual, predicted) :
#>   `actual` contains missing values
```

Another option is to use the existing infrastructure. {yardstick} does all kinds of safety checks before executing a function, and you can, via the `metric_vec_template()` pass a `SLmetrics::foo()` in the `foo_impl()`-function. This gives you the safety of {yardstick}, and the efficiency of {SLmetrics}.[2]

[2]An example would be appropriate. But my first attempt lead to a `decrecated`-warning, which is also one of

! Important

Be aware that using {SLmetrics} with {yardstick} will introduce some efficiency overhead - especially on large vectors.

5.5 Key take-aways

{SLmetrics} assumes that the end-user follows the typical AI/ML workflow, and has an understanding of `R` beyond beginner-level. And therefore {SLmetrics} does not check the validity of the user-input, which may lead to undefined behavior if input is ill-defined.

the main reasons I developed this {pkg}, and gave up. See the {documentation} on how to create custom metrics using {yardstick}.

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.