Package 'testarguments'

May 28, 2021

Title Test (Multiple) Arguments of a User-Defined Prediction Algorithm

Version 0.0.1

Description

Finding the best values for user-specified arguments of a prediction algorithm can be difficult, particularly if there is an interaction between argument levels. This package automates the testing of any user-defined prediction algorithm over an arbitrary number of arguments. It includes functions for testing the algorithm over the given arguments with respect to an arbitrary number of user-defined diagnostics, visualising the results of these tests, and finding the optimal argument combinations with respect to each diagnostic.

Maintainer Matthew Sainsbury-Dale <msainsburydale@gmail.com>

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.1.1

Imports magrittr, ggplot2, reshape2, plyr, methods, stats, dplyr

Suggests knitr, rmarkdown, markdown

VignetteBuilder knitr

NeedsCompilation no

Author Matthew Sainsbury-Dale [aut, cre]

Repository CRAN

Date/Publication 2021-05-28 07:10:02 UTC

R topics documented:

c,testargs-method *Combine objects of class 'testargs'*

Description

Combines an arbitrary number of 'testargs' objects

Usage

```
## S4 method for signature 'testargs'
c(x, ...)
```

Arguments

x	object of class 'testargs'
...	objects of class 'testargs' to be combined with x

Details

If the argument and diagnostic names are inconsistent across objects, the combined 'testargs' object is constructed by simply taking the union of all argument and diagnostic names. Then, rbind.fill() is used to combine the diagnostic data, producing intentional NA values where appropriate.

Value

An object of class 'testargs', the result of combining x and ...

optimal_arguments *Find the optimal combinations of arguments for each diagnostic*

Description

The measure of optimality is typically diagnostic dependent; for example, we wish to minimise the RMSE and run time, but we want coverage to be as close to the purported value as possible. Hence, optimal_arguments() allows one to set the optimality criteria individually for each diagnostic.

Usage

```
optimal_arguments(object, optimality_criterion = which.min)
```

Arguments

object	an object of class 'testargs'
optimality_criterion	a function (or list of functions) that defines the optimality criterion for each diagnostic. Each function should return a single positive integer indicating the index of the optimal argument combination. If a named list is provided with less elements than the number of diagnostic scores, unspecified diagnostics are assumed to be negatively oriented (i.e., assigned optimality criterion which.min)

Value

A data.frame; each row corresponds to one of the diagnostics (specified by the row names), and the columns contain the argument values that optimise the corresponding diagnostic. The diagnostics at each of these optimal argument combinations are also included

Examples

```
## See ?test_arguments
```

---

plot_diagnostics *Visualise diagnostics across the tested arguments*

Description

Using various aesthetics, plot_diagnostics() can visualise the performance of all combinations of up to 4 different arguments simultaneously.

Usage

```
plot_diagnostics(
  object,
  focused_args = NULL,
  average_out_non_focused_args = TRUE,
  plot_order = NULL
)
```

Arguments

object	an object of class 'testargs'
focused_args	the arguments we wish to plot. If NULL, all arguments are plotted (i.e., focused_args = object@arg_names)
average_out_non_focused_args	logical indicating whether we should average over the non-focused arguments
plot_order	specifies the order in which we are to assign arguments to the various aesthetics. If NULL, the arguments are assigned based on their type, in the order 'numeric', 'integer', 'factor', 'character', and 'logical'. Otherwise, plot_order should be an integer vector with the same length as focused_args

Value

a facetted 'ggplot' object, where:

- the columns of the facet are split by the diagnostics
- the y-axis corresponds to the values of the diagnostics
- the x-axis corresponds to the first argument
- the colour scale and grouping correspond to the second argument (if present)
- if a third argument is present, facet_grid() is used, whereby columns correspond to levels of the third argument, and rows correspond to diagnostics. Note that facet_grid() forces a given row to share a common y-scale, so the plot would be misleading if diagnostics were kept as columns
- the shape of the points corresponds to the fourth argument (if present)

Examples

```
## See ?test_arguments
```

---

testargs-class 'testargs' *class*

Description

This is the central class definition of the testarguments package, containing all information from a call to test_arguments

Slots

diagnostics_df a data.frame containing the diagnostics for each combination of the supplied arguments

arg_names the argument names

diagnostic_names the diagnostic names

---

test_arguments *Test (multiple) arguments of a prediction algorithm*

Description

Test the performance of a prediction algorithm over a range of argument values. Multiple arguments can be tested simultaneously.

Usage

```
test_arguments(pred_fun, df_train, df_test, diagnostic_fun, arguments)
```

Arguments

pred_fun	The prediction algorithm to be tested. It should be a function with formal arguments df_train and df_test, which are data used to train the model and test out-of-sample predictive performance, respectively, as well as any arguments which are to be tested. The value of pred_fun should be a matrix-like object with named columns and the same number of rows as df_test
df_train	training data
df_test	testing data
diagnostic_fun	the criteria with which the predictive performance will be assessed
arguments	named list of arguments and their values to check

Details

For each combination of the supplied argument levels, the value of pred_fun() is combined with df_test using cbind(), which is then passed into diagnostic_fun() to compute the diagnostics. Since the number of columns in the returned value of pred_fun() is arbitrary, one can test both predictions and uncertainty quantification of the predictions (e.g., by including prediction standard errors or predictive interval bounds)

Value

an object of class 'testargs' containing all information from the testing procedure

See Also

plot_diagnostics, optimal_arguments

Examples

```
library("testarguments")

## Simulate training and testing data
RNGversion("3.6.0"); set.seed(1)
n  <- 1000                                          # sample size
x  <- seq(-1, 1, length.out = n)                    # covariates
mu <- exp(3 + 2 * x * (x - 1) * (x + 1) * (x - 2))  # polynomial function in x
Z  <- rpois(n, mu)                                  # simulate data
df       <- data.frame(x = x, Z = Z, mu = mu)
train_id <- sample(1:n, n/2, replace = FALSE)
df_train <- df[train_id, ]
df_test  <- df[-train_id, ]

## Algorithm that uses df_train to predict over df_test. We use glm(), and
## test the degree of the regression polynomial and the link function.
pred_fun <- function(df_train, df_test, degree, link) {

  M <- glm(Z ~ poly(x, degree), data = df_train,
           family = poisson(link = as.character(link)))
```

```
  ## Predict over df_test
  pred <- as.data.frame(predict(M, df_test, type = "link", se.fit = TRUE))

  ## Compute response level predictions and 90% prediction interval
  inv_link <- family(M)$linkinv
  fit_Y <- pred$fit
  se_Y  <- pred$se.fit
  pred <- data.frame(fit_Z = inv_link(fit_Y),
                     upr_Z = inv_link(fit_Y + 1.645 * se_Y),
                     lwr_Z = inv_link(fit_Y - 1.645 * se_Y))

  return(pred)
}

## Define diagnostic function. Should return a named vector
diagnostic_fun <- function(df) {
  with(df, c(
    RMSE = sqrt(mean((Z - fit_Z)^2)),
    MAE = mean(abs(Z - fit_Z)),
    coverage = mean(lwr_Z < mu & mu < upr_Z)
  ))
}

## Compute the user-defined diagnostics over a range of argument levels
testargs_object <- test_arguments(
  pred_fun, df_train, df_test, diagnostic_fun,
  arguments = list(degree = 1:6, link = c("log", "sqrt"))
)

## Visualise the performance across all combinations of the supplied arguments
plot_diagnostics(testargs_object)

## Focus on a subset of the tested arguments
plot_diagnostics(testargs_object, focused_args = "degree")

## Compute the optimal arguments for each diagnostic
optimal_arguments(
  testargs_object,
  optimality_criterion = list(coverage = function(x) which.min(abs(x - 0.90)))
)
```

Index