Package ‘mlexperiments’

January 16, 2026

Title Machine Learning Experiments

Version 1.0.0

Description Provides 'R6' objects to perform parallelized hyperparameter optimization and cross-validation. Hyperparameter optimization can be performed with Bayesian optimization (via 'rBayesianOptimization' <https://cran.r-project.org/package=rBayesianOptimization>) and grid search. The optimized hyperparameters can be validated using k-fold cross-validation. Alternatively, hyperparameter optimization and validation can be performed with nested cross-validation. While 'mlexperiments' focuses on core wrappers for machine learning experiments, additional learner algorithms can be supplemented by inheriting from the provided learner base class.

License GPL (>= 3)

URL https://github.com/kapsner/mlexperiments

BugReports https://github.com/kapsner/mlexperiments/issues

Depends R (>= 4.1.0)

Imports data.table, kdry, progress, R6, splitTools, stats

Suggests class, datasets, lintr, measures, mlbench, parallel, quarto, rBayesianOptimization, rpart, testthat (>= 3.0.1)

VignetteBuilder quarto

Config/testthat/edition 3

Config/testthat/parallel false

Date/Publication 2026-01-16 17:00:15 UTC

Encoding UTF-8

SystemRequirements Quarto command line tools (https://github.com/quarto-dev/quarto-cli).

RoxygenNote 7.3.3

NeedsCompilation no

Author Lorenz A. Kapsner [cre, aut, cph] (ORCID: <https://orcid.org/0000-0003-1866-860X>)

Maintainer Lorenz A. Kapsner <lorenz.kapsner@gmail.com>

Repository CRAN

Contents

handle_cat_vars 2
LearnerGlm 3
LearnerKnn 4
LearnerLm 6
LearnerRpart 7
metric 9
metric_types_helper 10
MLBase 11
MLCrossValidation 12
MLExperimentsBase 18
MLLearnerBase 19
MLNestedCV 24
MLTuneParameters 31
performance 36
predictions 38
validate_fold_equality 40

Index **43**

`handle_cat_vars` *handle_cat_vars*

Description

Helper function to handle categorical variables

Usage

```
handle_cat_vars(kwargs)
```

Arguments

`kwargs`	A list containing keyword arguments.

Details

This function is a utility function to separate the list element with the names of the categorical variables from the key word arguments list to be passed further on to `kdry::dtr_matrix2df()`.

Value

Returns a list with two elements:

- `params` The keyword arguments without `cat_vars`.
- `cat_vars` The vector `cat_vars`.

See Also

`kdry::dtr_matrix2df()`

Examples

```
handle_cat_vars(list(cat_vars = c("a", "b", "c"), arg1 = 1, arg2 = 2))
```

`LearnerGlm` *LearnerGlm R6 class*

Description

This learner is a wrapper around `stats::glm()` in order to perform a generalized linear regression. There is no implementation for tuning parameters.

Details

Can be used with

- MLCrossValidation

Implemented methods:

- `$fit` To fit the model.
- `$predict` To predict new data with the model.

Super class

`mlexperiments::MLLearnerBase` -> `LearnerGlm`

Methods

Public methods:

- `LearnerGlm$new()`
- `LearnerGlm$clone()`

Method `new()`**:** Create a new `LearnerGlm` object.

Usage:

```
LearnerGlm$new()
```

Details: This learner is a wrapper around `stats::glm()` in order to perform a generalized linear regression. There is no implementation for tuning parameters, thus the only experiment to use `LearnerGlm` for is MLCrossValidation.

Returns: A new `LearnerGlm` R6 object.

Examples:

```
LearnerGlm$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
LearnerGlm$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`stats::glm()`

`stats::glm()`

Examples

```
LearnerGlm$new()


## ------------------------------------------------
## Method `LearnerGlm$new`
## ------------------------------------------------

LearnerGlm$new()
```

`LearnerKnn` *LearnerKnn R6 class*

Description

This learner is a wrapper around `class::knn()` in order to perform a k-nearest neighbor classification.

Details

Optimization metric: classification error rate Can be used with

- MLTuneParameters
- MLCrossValidation
- MLNestedCV

Implemented methods:

- `$fit` To fit the model.
- `$predict` To predict new data with the model.
- `$cross_validation` To perform a grid search (hyperparameter optimization).
- `$bayesian_scoring_function` To perform a Bayesian hyperparameter optimization.

For the two hyperparameter optimization strategies ("grid" and "bayesian"), the parameter `metric_optimization_higher_b` of the learner is set to `FALSE` by default as the mean misclassification error (`measures::MMCE()`) is used as the optimization metric.

Super class

`mlexperiments::MLLearnerBase` -> `LearnerKnn`

Methods

Public methods:

- `LearnerKnn$new()`
- `LearnerKnn$clone()`

Method `new()`**:** Create a new `LearnerKnn` object.

Usage:

```
LearnerKnn$new()
```

Details: This learner is a wrapper around `class::knn()` in order to perform a k-nearest neighbor classification. The following experiments are implemented:

- MLTuneParameters
- MLCrossValidation
- MLNestedCV For the two hyperparameter optimization strategies ("grid" and "bayesian"), the parameter `metric_optimization_higher_better` of the learner is set to `FALSE` by default as the mean misclassification error (`measures::MMCE()`) is used as the optimization metric.

Examples:

```
if (requireNamespace("class", quietly = TRUE)) {
  LearnerKnn$new()
}
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
LearnerKnn$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`class::knn()`, `measures::MMCE()`

`class::knn()`, `measures::MMCE()`

Examples

```
if (requireNamespace("class", quietly = TRUE)) {
  LearnerKnn$new()
}


## ------------------------------------------------
## Method `LearnerKnn$new`
## ------------------------------------------------

if (requireNamespace("class", quietly = TRUE)) {
  LearnerKnn$new()
}
```

`LearnerLm` *LearnerLm R6 class*

Description

This learner is a wrapper around `stats::lm()` in order to perform a linear regression. There is no implementation for tuning parameters.

Details

Can be used with

- mlexperiments::MLCrossValidation

Implemented methods:

- `$fit` To fit the model.
- `$predict` To predict new data with the model.

Super class

`mlexperiments::MLLearnerBase` -> `LearnerLm`

Methods

Public methods:

- `LearnerLm$new()`
- `LearnerLm$clone()`

Method `new()`**:** Create a new `LearnerLm` object.

Usage:

`LearnerLm$new()`

Details: This learner is a wrapper around `stats::lm()` in order to perform a linear regression. There is no implementation for tuning parameters, thus the only experiment to use `LearnerLm` for is MLCrossValidation

Returns: A new `LearnerLm` R6 object.

Examples:

```
LearnerLm$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
LearnerLm$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`stats::lm()`

`stats::lm()`

Examples

```
LearnerLm$new()


## ------------------------------------------------
## Method `LearnerLm$new`
## ------------------------------------------------

LearnerLm$new()
```

`LearnerRpart` *LearnerRpart R6 class*

Description

This learner is a wrapper around `rpart::rpart()` in order to fit recursive partitioning and regression trees.

Details

Optimization metric:

- classification (`method = "class"`): classification error rate
- regression (`method = "anova"`): mean squared error

Can be used with

- MLTuneParameters
- MLCrossValidation
- MLNestedCV

Implemented methods:

- `$fit` To fit the model.
- `$predict` To predict new data with the model.
- `$cross_validation` To perform a grid search (hyperparameter optimization).
- `$bayesian_scoring_function` To perform a Bayesian hyperparameter optimization.

Parameters that are specified with `parameter_grid` and / or `learner_args` are forwarded to `rpart`'s argument `control` (see `rpart::rpart.control()` for further details).

For the two hyperparameter optimization strategies ("grid" and "bayesian"), the parameter `metric_optimization_higher_b` of the learner is set to `FALSE` by default as the mean misclassification error rate (`measures::MMCE()`) is used as the optimization metric for classification tasks and the mean squared error (`measures::MSE()`) is used for regression tasks.

Super class

`mlexperiments::MLLearnerBase` -> `LearnerRpart`

Methods

Public methods:

- `LearnerRpart$new()`
- `LearnerRpart$clone()`

Method `new()`**:** Create a new `LearnerRpart` object.

Usage:

```
LearnerRpart$new()
```

Details: This learner is a wrapper around `rpart::rpart()` in order to fit recursive partitioning and regression trees. The following experiments are implemented:

- MLTuneParameters
- MLCrossValidation
- MLNestedCV

For the two hyperparameter optimization strategies ("grid" and "bayesian"), the parameter `metric_optimization_highe` of the learner is set to `FALSE` by default as the mean misclassification error (`measures::MMCE()`) is used as the optimization metric for classification tasks and the mean squared error (`measures::MSE()`) is used for regression tasks.

Examples:

```
if (requireNamespace("rpart", quietly = TRUE)) {
  LearnerRpart$new()
}
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`LearnerRpart$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

`rpart::rpart()`, `measures::MMCE()`, [measures::MSE)], [rpart::rpart.control()]

[measures::MSE)]: R:measures::MSE) [rpart::rpart.control()]: R:rpart::rpart.control()

`rpart::rpart()`, `measures::MMCE()`, `measures::MSE()`

Examples

```
if (requireNamespace("rpart", quietly = TRUE)) {
  LearnerRpart$new()
}


## ------------------------------------------------
## Method `LearnerRpart$new`
## ------------------------------------------------

if (requireNamespace("rpart", quietly = TRUE)) {
  LearnerRpart$new()
}
```

`metric` *metric*

Description

Returns a metric function which can be used for the experiments (especially the cross-validation experiments) to compute the performance.

Usage

```
metric(name)
```

Arguments

`name`	A metric name. Accepted names are the names of the metric function exported from the `measures` R package.

Details

This function is a utility function to select performance metrics from the measures R package and to reformat them into a form that is required by the mlexperiments R package. For mlexperiments it is required that a metric function takes the two arguments ground_truth, and predictions, as well as additional names arguments that are necessary to compute the performance, which are provided via the ellipsis argument (...). When using the performance metric with an experiment of class "MLCrossValidation", such arguments can be defined as a list provided to the field performance_metric_args of the R6 class. The main purpose of mlexperiments::metric() is convenience and to re-use already existing implementations of the metrics. However, custom functions can be provided easily to compute the performance of the experiments, simply by providing a function that takes the above mentioned arguments and returns one performance metric value.

Value

Returns a function that can be used as function to calculate the performance metric throughout the experiments.

Examples

```
if (requireNamespace("measures", quietly = TRUE)) {
  metric("AUC")
}
```

---

metric_types_helper *metric_types_helper*

Description

Prepares the data to be conform with the requirements of the metrics from measures.

Usage

```
metric_types_helper(FUN, y, perf_args)
```

Arguments

FUN	A metric function, created with metric().
y	The outcome vector.
perf_args	A list. The arguments to call the metric function with.

Details

The measures R package makes some restrictions on the data type of the ground truth and the predictions, depending on the metric, i.e. the type of the task (regression or classification). Thus, it is necessary to convert the inputs to the metric function accordingly, which is done with this helper function.

Value

Returns the calculated performance measure.

Examples

```
if (requireNamespace("measures", quietly = TRUE)) {
  set.seed(123)
  ground_truth <- sample(0:1, 100, replace = TRUE)
  predictions <- sample(0:1, 100, replace = TRUE)
  FUN <- metric("ACC")

  perf_args <- list(
    ground_truth = ground_truth,
    predictions = predictions
  )

  metric_types_helper(
    FUN = FUN,
    y = ground_truth,
    perf_args = perf_args
  )
}
```

MLBase *Basic R6 Class for the mlexperiments package*

Description

Basic R6 Class for the mlexperiments package

Basic R6 Class for the mlexperiments package

Public fields

`results` A list. This field is used to store the final results of the respective methods.

Methods

Public methods:

- `MLBase$new()`
- `MLBase$clone()`

Method `new()`**:** Create a new `MLBase` object.

Usage:

`MLBase$new(seed, ncores = -1L)`

Arguments:

`seed` An integer. Needs to be set for reproducibility purposes.

`ncores` An integer to specify the number of cores used for parallelization (default: `-1L`).

Returns: A new `MLBase` R6 object.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MLBase$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`MLCrossValidation` *R6 Class to perform cross-validation experiments*

Description

The `MLCrossValidation` class is used to construct a cross validation object and to perform a k-fold cross validation for a specified machine learning algorithm using one distinct hyperparameter setting.

Details

The `MLCrossValidation` class requires to provide a named list of predefined row indices for the cross validation folds, e.g., created with the function `splitTools::create_folds()`. This list also defines the `k` of the k-fold cross-validation. When wanting to perform a repeated k-fold cross validations, just provide a list with all repeated fold definitions, e.g., when specifying the argument `m_rep` of `splitTools::create_folds()`.

Super classes

`mlexperiments::MLBase` -> `mlexperiments::MLExperimentsBase` -> `MLCrossValidation`

Public fields

`fold_list` A named list of predefined row indices for the cross validation folds, e.g., created with the function `splitTools::create_folds()`.

`return_models` A logical. If the fitted models should be returned with the results (default: `FALSE`).

`performance_metric` Either a named list with metric functions, a single metric function, or a character vector with metric names from the `measures` package. The provided functions must take two named arguments: `ground_truth` and `predictions`. For metrics from the `measures` package, the wrapper function `metric()` exists in order to prepare them for use with the `mlexperiments` package.

`performance_metric_args` A list. Further arguments required to compute the performance metric.

`predict_args` A list. Further arguments required to compute the predictions.

Methods

Public methods:

- `MLCrossValidation$new()`
- `MLCrossValidation$execute()`
- `MLCrossValidation$clone()`

Method `new()`: Create a new `MLCrossValidation` object.

Usage:

```
MLCrossValidation$new(
  learner,
  fold_list,
  seed,
  ncores = -1L,
  return_models = FALSE
)
```

Arguments:

`learner` An initialized learner object that inherits from class "MLLearnerBase".

`fold_list` A named list of predefined row indices for the cross validation folds, e.g., created with the function `splitTools::create_folds()`.

`seed` An integer. Needs to be set for reproducibility purposes.

`ncores` An integer to specify the number of cores used for parallelization (default: `-1L`).

`return_models` A logical. If the fitted models should be returned with the results (default: `FALSE`).

Details: The `MLCrossValidation` class requires to provide a named list of predefined row indices for the cross validation folds, e.g., created with the function `splitTools::create_folds()`. This list also defines the `k` of the k-fold cross-validation. When wanting to perform a repeated k-fold cross validations, just provide a list with all repeated fold definitions, e.g., when specifing the argument `m_rep` of `splitTools::create_folds()`.

Examples:

```
if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {
  dataset <- do.call(
    cbind,
    c(sapply(paste0("col", 1:6), function(x) {
      rnorm(n = 500)
      },
      USE.NAMES = TRUE,
      simplify = FALSE
     ),
     list(target = sample(0:1, 500, TRUE))
  ))
  fold_list <- splitTools::create_folds(
    y = dataset[, 7],
    k = 3,
    type = "stratified",
```

```
    seed = 123
  )
  cv <- MLCrossValidation$new(
    learner = LearnerKnn$new(),
    fold_list = fold_list,
    seed = 123,
    ncores = 2
  )
}
```

Method `execute()`**:** Execute the cross validation.

Usage:

`MLCrossValidation$execute()`

Details: All results of the cross validation are saved in the field `$results` of the `MLCrossValidation` class. After successful execution of the cross validation, `$results` contains a list with the items:

- "fold" A list of folds containing the following items for each cross validation fold:
  - "fold_ids" A vector with the utilized in-sample row indices.
  - "ground_truth" A vector with the ground truth.
  - "predictions" A vector with the predictions.
  - "learner.args" A list with the arguments provided to the learner.
  - "model" If `return_models = TRUE`, the fitted model.
- "summary" A data.table with the summarized results (same as the returned value of the `execute` method).
- "performance" A list with the value of the performance metric calculated for each of the cross validation folds.

Returns: The function returns a data.table with the results of the cross validation. More results are accessible from the field `$results` of the `MLCrossValidation` class.

Examples:

```
if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {
  dataset <- do.call(
    cbind,
    c(sapply(paste0("col", 1:6), function(x) {
      rnorm(n = 500)
      },
      USE.NAMES = TRUE,
      simplify = FALSE
     ),
     list(target = sample(0:1, 500, TRUE))
  ))
  fold_list <- splitTools::create_folds(
    y = dataset[, 7],
    k = 3,
    type = "stratified",
    seed = 123
```

```
    )
    cv <- MLCrossValidation$new(
      learner = LearnerKnn$new(),
      fold_list = fold_list,
      seed = 123,
      ncores = 2
    )
    cv$learner_args <- list(
      k = 20,
      l = 0,
      test = parse(text = "fold_test$x")
    )
    cv$predict_args <- list(type = "response")
    cv$performance_metric_args <- list(
      positive = "1",
      negative = "0"
    )
    cv$performance_metric <- metric("MMCE")

    # set data
    cv$set_data(
      x = data.matrix(dataset[, -7]),
      y = dataset[, 7]
    )

    cv$execute()
  }
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MLCrossValidation$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`splitTools::create_folds()`

`splitTools::create_folds()`, `metric()`

Examples

```
if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {

  dataset <- do.call(
    cbind,
    c(sapply(paste0("col", 1:6), function(x) {
```

```
      rnorm(n = 500)
      },
      USE.NAMES = TRUE,
      simplify = FALSE
     ),
     list(target = sample(0:1, 500, TRUE))
  ))

  fold_list <- splitTools::create_folds(
    y = dataset[, 7],
    k = 3,
    type = "stratified",
    seed = 123
  )

  cv <- MLCrossValidation$new(
    learner = LearnerKnn$new(),
    fold_list = fold_list,
    seed = 123,
    ncores = 2
  )

  # learner parameters
  cv$learner_args <- list(
    k = 20,
    l = 0,
    test = parse(text = "fold_test$x")
  )

  # performance parameters
  cv$predict_args <- list(type = "response")
  cv$performance_metric_args <- list(
    positive = "1",
    negative = "0"
  )
  cv$performance_metric <- metric("MMCE")

  # set data
  cv$set_data(
    x = data.matrix(dataset[, -7]),
    y = dataset[, 7]
  )

  cv$execute()
}


## ------------------------------------------------
## Method `MLCrossValidation$new`
## ------------------------------------------------

if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {
```

```
  dataset <- do.call(
    cbind,
    c(sapply(paste0("col", 1:6), function(x) {
      rnorm(n = 500)
      },
      USE.NAMES = TRUE,
      simplify = FALSE
     ),
     list(target = sample(0:1, 500, TRUE))
  ))
  fold_list <- splitTools::create_folds(
    y = dataset[, 7],
    k = 3,
    type = "stratified",
    seed = 123
  )
  cv <- MLCrossValidation$new(
    learner = LearnerKnn$new(),
    fold_list = fold_list,
    seed = 123,
    ncores = 2
  )
}


## ------------------------------------------------
## Method `MLCrossValidation$execute`
## ------------------------------------------------

if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {
  dataset <- do.call(
    cbind,
    c(sapply(paste0("col", 1:6), function(x) {
      rnorm(n = 500)
      },
      USE.NAMES = TRUE,
      simplify = FALSE
     ),
     list(target = sample(0:1, 500, TRUE))
  ))
  fold_list <- splitTools::create_folds(
    y = dataset[, 7],
    k = 3,
    type = "stratified",
    seed = 123
  )
  cv <- MLCrossValidation$new(
    learner = LearnerKnn$new(),
    fold_list = fold_list,
    seed = 123,
    ncores = 2
  )
```

```
  cv$learner_args <- list(
    k = 20,
    l = 0,
    test = parse(text = "fold_test$x")
  )
  cv$predict_args <- list(type = "response")
  cv$performance_metric_args <- list(
    positive = "1",
    negative = "0"
  )
  cv$performance_metric <- metric("MMCE")

  # set data
  cv$set_data(
    x = data.matrix(dataset[, -7]),
    y = dataset[, 7]
  )

  cv$execute()
}
```

`MLExperimentsBase` *R6 Class on which the experiment classes are built on*

Description

R6 Class on which the experiment classes are built on

R6 Class on which the experiment classes are built on

Super class

`mlexperiments::MLBase` -> `MLExperimentsBase`

Public fields

`learner_args` A list containing the parameter settings of the learner algorithm.

`learner` An initialized learner object that inherits from class `"MLLearnerBase"`.

Methods

Public methods:

- `MLExperimentsBase$new()`
- `MLExperimentsBase$set_data()`
- `MLExperimentsBase$clone()`

Method `new()`**:** Create a new `MLExperimentsBase` object.

Usage:

```
MLExperimentsBase$new(learner, seed, ncores = -1L)
```

Arguments:

`learner` An initialized learner object that inherits from class "MLLearnerBase".

`seed` An integer. Needs to be set for reproducibility purposes.

`ncores` An integer to specify the number of cores used for parallelization (default: `-1L`).

Returns: A new `MLExperimentsBase` R6 object.

Method `set_data()`**:** Set the data for the experiment.

Usage:

```
MLExperimentsBase$set_data(x, y, cat_vars = NULL)
```

Arguments:

`x` A matrix with the training data.

`y` A vector with the target.

`cat_vars` A character vector with the column names of variables that should be treated as categorical features (if applicable / supported by the respective algorithm).

Returns: The function has no return value. It internally performs quality checks on the provided data and, if passed, defines private fields of the R6 class.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MLExperimentsBase$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`MLLearnerBase` *R6 Class to construct learners*

Description

The `MLLearnerBase` class is used to construct a learner object that can be used with the experiment classes from the `mlexperiments` package. It is thought to serve as a class to inherit from when creating new learners.

Details

The learner class exposes 4 methods that can be defined:

- `$fit` A wrapper around the private function `fun_fit`, which needs to be defined for every learner. The return value of this function is the fitted model.
- `$predict` A wrapper around the private function `fun_predict`, which needs to be defined for every learner. The function must accept the three arguments `model`, `newdata`, and `ncores` and is a wrapper around the respective learner's predict-function. In order to allow the passing of further arguments, the ellipsis (`...`) can be used. The function should return the prediction results.

- `$cross_validation` A wrapper around the private function `fun_optim_cv`, which needs to be defined when hyperparameters should be optimized with a grid search (required for use with MLTuneParameters, and MLNestedCV).
- `$bayesian_scoring_function` A wrapper around the private function `fun_bayesian_scoring_function`, which needs to be defined when hyperparameters should be optimized with a Bayesian process (required for use with MLTuneParameters, and MLNestedCV).

For further details please refer to the package's vignette.

Public fields

`cluster_export` A character vector defining the (internal) functions that need to be exported to the parallelization cluster. This is only required when performing a Bayesian hyperparameter optimization. See also `parallel::clusterExport()`.

`metric_optimization_higher_better` A logical. Defines the direction of the optimization metric used throughout the hyperparameter optimization. This field is set automatically during the initialization of the `MLLearnerBase` object. Its purpose is to make it accessible by the evaluation functions from MLTuneParameters.

`environment` The environment in which to search for the functions of the learner (default: `-1L`).

`seed` Seed for reproducible results.

Methods

Public methods:

- `MLLearnerBase$new()`
- `MLLearnerBase$cross_validation()`
- `MLLearnerBase$fit()`
- `MLLearnerBase$predict()`
- `MLLearnerBase$bayesian_scoring_function()`
- `MLLearnerBase$clone()`

Method `new()`**:** Create a new `MLLearnerBase` object.

Usage:

```
MLLearnerBase$new(metric_optimization_higher_better)
```

Arguments:

`metric_optimization_higher_better` A logical. Defines the direction of the optimization metric used throughout the hyperparameter optimization.

Returns: A new `MLLearnerBase` R6 object.

Examples:

```
MLLearnerBase$new(metric_optimization_higher_better = FALSE)
```

Method `cross_validation()`**:** Perform a cross-validation with an `MLLearnerBase`.

Usage:

```
MLLearnerBase$cross_validation(...)
```

Arguments:

... Arguments to be passed to the learner's cross-validation function.

Details: A wrapper around the private function `fun_optim_cv`, which needs to be defined when hyperparameters should be optimized with a grid search (required for use with MLTuneParameters, and MLNestedCV. However, the function should be never executed directly but by the respective experiment wrappers MLTuneParameters, and MLNestedCV. For further details please refer to the package's vignette.

Returns: The fitted model.

Examples:

```
learner <- MLLearnerBase$new(metric_optimization_higher_better = FALSE)
\dontrun{
# This example cannot be run without further adaptions.
# The method `$cross_validation()` needs to be overwritten when
# inheriting from this class.
learner$cross_validation()
}
```

Method `fit()`**:** Fit a `MLLearnerBase` object.

Usage:

```
MLLearnerBase$fit(...)
```

Arguments:

... Arguments to be passed to the learner's fitting function.

Details: A wrapper around the private function `fun_fit`, which needs to be defined for every learner. The return value of this function is the fitted model. However, the function should be never executed directly but by the respective experiment wrappers MLTuneParameters, MLCrossValidation, and MLNestedCV. For further details please refer to the package's vignette.

Returns: The fitted model.

Examples:

```
learner <- MLLearnerBase$new(metric_optimization_higher_better = FALSE)
\dontrun{
# This example cannot be run without further adaptions.
# The method `$fit()` needs to be overwritten when
# inheriting from this class.
learner$fit()
}
```

Method `predict()`**:** Make predictions from a fitted `MLLearnerBase` object.

Usage:

```
MLLearnerBase$predict(model, newdata, ncores = -1L, ...)
```

Arguments:

`model` A fitted model of the learner (as returned by `MLLearnerBase$fit()`).

`newdata` The new data for which predictions should be made using the `model`.

`ncores` An integer to specify the number of cores used for parallelization (default: `-1L`).

`...` Further arguments to be passed to the learner's predict function.

Details: A wrapper around the private function `fun_predict`, which needs to be defined for every learner. The function must accept the three arguments `model`, `newdata`, and `ncores` and is a wrapper around the respective learner's predict-function. In order to allow the passing of further arguments, the ellipsis (`...`) can be used. The function should return the prediction results. However, the function should be never executed directly but by the respective experiment wrappers MLTuneParameters, MLCrossValidation, and MLNestedCV. For further details please refer to the package's vignette.

Returns: The predictions for `newdata`.

Examples:

```
learner <- MLLearnerBase$new(metric_optimization_higher_better = FALSE)
\dontrun{
# This example cannot be run without further adaptions.
# The method `$predict()` needs to be overwritten when
# inheriting from this class.
learner$fit()
learner$predict()
}
```

Method `bayesian_scoring_function()`: Perform a Bayesian hyperparameter optimization with an `MLLearnerBase`.

Usage:

```
MLLearnerBase$bayesian_scoring_function(...)
```

Arguments:

`...` Arguments to be passed to the learner's Bayesian scoring function.

Details: A wrapper around the private function `fun_bayesian_scoring_function`, which needs to be defined when hyperparameters should be optimized with a Bayesian process (required for use with MLTuneParameters, and MLNestedCV. However, the function should be never executed directly but by the respective experiment wrappers MLTuneParameters, and MLNestedCV. For further details please refer to the package's vignette.

Returns: The results of the Bayesian scoring.

Examples:

```
learner <- MLLearnerBase$new(metric_optimization_higher_better = FALSE)
\dontrun{
# This example cannot be run without further adaptions.
# The method `$fun_bayesian_scoring_function()` needs to be overwritten when
# inheriting from this class.
learner$bayesian_scoring_function()
}
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
MLLearnerBase$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

MLTuneParameters, MLCrossValidation, and MLNestedCV

MLTuneParameters, MLCrossValidation, and MLNestedCV

MLTuneParameters, MLCrossValidation, and MLNestedCV

`rBayesianOptimization::BayesianOptimization()`, MLTuneParameters, and MLNestedCV

Examples

```
MLLearnerBase$new(metric_optimization_higher_better = FALSE)


## ------------------------------------------------
## Method `MLLearnerBase$new`
## ------------------------------------------------

MLLearnerBase$new(metric_optimization_higher_better = FALSE)


## ------------------------------------------------
## Method `MLLearnerBase$cross_validation`
## ------------------------------------------------

learner <- MLLearnerBase$new(metric_optimization_higher_better = FALSE)
## Not run:
# This example cannot be run without further adaptions.
# The method `$cross_validation()` needs to be overwritten when
# inheriting from this class.
learner$cross_validation()

## End(Not run)


## ------------------------------------------------
## Method `MLLearnerBase$fit`
## ------------------------------------------------

learner <- MLLearnerBase$new(metric_optimization_higher_better = FALSE)
## Not run:
# This example cannot be run without further adaptions.
# The method `$fit()` needs to be overwritten when
# inheriting from this class.
learner$fit()

## End(Not run)
```

```
## ------------------------------------------------
## Method `MLLearnerBase$predict`
## ------------------------------------------------

learner <- MLLearnerBase$new(metric_optimization_higher_better = FALSE)
## Not run:
# This example cannot be run without further adaptions.
# The method `$predict()` needs to be overwritten when
# inheriting from this class.
learner$fit()
learner$predict()

## End(Not run)


## ------------------------------------------------
## Method `MLLearnerBase$bayesian_scoring_function`
## ------------------------------------------------

learner <- MLLearnerBase$new(metric_optimization_higher_better = FALSE)
## Not run:
# This example cannot be run without further adaptions.
# The method `$fun_bayesian_scoring_function()` needs to be overwritten when
# inheriting from this class.
learner$bayesian_scoring_function()

## End(Not run)
```

MLNestedCV *R6 Class to perform nested cross-validation experiments*

Description

The MLNestedCV class is used to construct a nested cross validation object and to perform a nested cross validation for a specified machine learning algorithm by performing a hyperparameter optimization with the in-sample observations of each of the k outer folds and validate them directly on the out-of-sample observations of the respective fold.

Details

The MLNestedCV class requires to provide a named list of predefined row indices for the outer cross validation folds, e.g., created with the function splitTools::create_folds(). This list also defines the k of the k-fold cross-validation. Furthermore, a strategy needs to be chosen ("grid" or "bayesian") for the hyperparameter optimization as well as the parameter k_tuning to define the number of inner cross validation folds.

Super classes

mlexperiments::MLBase -> mlexperiments::MLExperimentsBase -> mlexperiments::MLCrossValidation -> MLNestedCV

Public fields

`strategy` A character. The strategy to optimize the hyperparameters (either `"grid"` or `"bayesian"`).

`parameter_bounds` A named list of tuples to define the parameter bounds of the Bayesian hyperparameter optimization. For further details please see the documentation of the `rBayesianOptimization` package.

`parameter_grid` A matrix with named columns in which each column represents a parameter that should be optimized and each row represents a specific hyperparameter setting that should be tested throughout the procedure. For `strategy = "grid"`, each row of the `parameter_grid` is considered as a setting that is evaluated. For `strategy = "bayesian"`, the `parameter_grid` is passed further on to the `initGrid` argument of the function `rBayesianOptimization::BayesianOptimization()` in order to initialize the Bayesian process. The maximum rows considered for initializing the Bayesian process can be specified with the R option `option("mlexperiments.bayesian.max_init")`, which is set to `4L` by default.

`optim_args` A named list of tuples to define the parameter bounds of the Bayesian hyperparameter optimization. For further details please see the documentation of the `rBayesianOptimization` package.

`split_type` A character. The splitting strategy to construct the k cross-validation folds. This parameter is passed further on to the function `splitTools::create_folds()` and defaults to `"stratified"`.

`split_vector` A vector If another criteria than the provided `y` should be considered for generating the cross-validation folds, it can be defined here. It is important, that a vector of the same length as `x` is provided here.

`k_tuning` An integer to define the number of cross-validation folds used to tune the hyperparameters.

Methods

Public methods:

- `MLNestedCV$new()`
- `MLNestedCV$execute()`
- `MLNestedCV$clone()`

Method `new()`: Create a new `MLNestedCV` object.

Usage:

```
MLNestedCV$new(
  learner,
  strategy = c("grid", "bayesian"),
  k_tuning,
  fold_list,
  seed,
  ncores = -1L,
  return_models = FALSE
)
```

Arguments:

`learner` An initialized learner object that inherits from class `"MLLearnerBase"`.

`strategy` A character. The strategy to optimize the hyperparameters (either "grid" or "bayesian").

`k_tuning` An integer to define the number of cross-validation folds used to tune the hyperparameters.

`fold_list` A named list of predefined row indices for the cross validation folds, e.g., created with the function `splitTools::create_folds()`.

`seed` An integer. Needs to be set for reproducibility purposes.

`ncores` An integer to specify the number of cores used for parallelization (default: `-1L`).

`return_models` A logical. If the fitted models should be returned with the results (default: `FALSE`).

Details: The `MLNestedCV` class requires to provide a named list of predefined row indices for the outer cross validation folds, e.g., created with the function `splitTools::create_folds()`. This list also defines the `k` of the k-fold cross-validation. Furthermore, a strategy needs to be chosen ("grid" or "bayesian") for the hyperparameter optimization as well as the parameter `k_tuning` to define the number of inner cross validation folds.

Examples:

```
if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {
  dataset <- do.call(
    cbind,
    c(sapply(paste0("col", 1:6), function(x) {
      rnorm(n = 500)
      },
      USE.NAMES = TRUE,
      simplify = FALSE
     ),
     list(target = sample(0:1, 500, TRUE))
  ))

  fold_list <- splitTools::create_folds(
    y = dataset[, 7],
    k = 3,
    type = "stratified",
    seed = 123
  )

  cv <- MLNestedCV$new(
    learner = LearnerKnn$new(),
    strategy = "grid",
    fold_list = fold_list,
    k_tuning = 3L,
    seed = 123,
    ncores = 2
  )
}
```

Method `execute()`**:** Execute the nested cross validation.

Usage:

```
MLNestedCV$execute()
```

Details: All results of the cross validation are saved in the field `$results` of the `MLNestedCV` class. After successful execution of the nested cross validation, `$results` contains a list with the items:

- "results.optimization" A list with the results of the hyperparameter optimization.
- "fold" A list of folds containing the following items for each cross validation fold:
  - "fold_ids" A vector with the utilized in-sample row indices.
  - "ground_truth" A vector with the ground truth.
  - "predictions" A vector with the predictions.
  - "learner.args" A list with the arguments provided to the learner.
  - "model" If `return_models = TRUE`, the fitted model.
- "summary" A data.table with the summarized results (same as the returned value of the `execute` method).
- "performance" A list with the value of the performance metric calculated for each of the cross validation folds.

Returns: The function returns a data.table with the results of the nested cross validation. More results are accessible from the field `$results` of the `MLNestedCV` class.

Examples:

```
if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {
  dataset <- do.call(
    cbind,
    c(sapply(paste0("col", 1:6), function(x) {
      rnorm(n = 500)
      },
      USE.NAMES = TRUE,
      simplify = FALSE
     ),
     list(target = sample(0:1, 500, TRUE))
  ))

  fold_list <- splitTools::create_folds(
    y = dataset[, 7],
    k = 3,
    type = "stratified",
    seed = 123
  )

  cv <- MLNestedCV$new(
    learner = LearnerKnn$new(),
    strategy = "grid",
    fold_list = fold_list,
    k_tuning = 3L,
    seed = 123,
    ncores = 2
```

```
  )

  # learner args (not optimized)
  cv$learner_args <- list(
    l = 0,
    test = parse(text = "fold_test$x")
  )

  # parameters for hyperparameter tuning
  cv$parameter_grid <- expand.grid(
    k = seq(4, 68, 8)
  )
  cv$split_type <- "stratified"

  # performance parameters
  cv$predict_args <- list(type = "response")
  cv$performance_metric_args <- list(
    positive = "1",
    negative = "0"
  )
  cv$performance_metric <- metric("MMCE")

  # set data
  cv$set_data(
    x = data.matrix(dataset[, -7]),
    y = dataset[, 7]
  )

  cv$execute()
}
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MLNestedCV$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

splitTools::create_folds()

splitTools::create_folds()

Examples

```
if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {
  dataset <- do.call(
```

```
  cbind,
  c(sapply(paste0("col", 1:6), function(x) {
    rnorm(n = 500)
    },
    USE.NAMES = TRUE,
    simplify = FALSE
   ),
   list(target = sample(0:1, 500, TRUE))
))

fold_list <- splitTools::create_folds(
  y = dataset[, 7],
  k = 3,
  type = "stratified",
  seed = 123
)

cv <- MLNestedCV$new(
  learner = LearnerKnn$new(),
  strategy = "grid",
  fold_list = fold_list,
  k_tuning = 3L,
  seed = 123,
  ncores = 2
)

# learner args (not optimized)
cv$learner_args <- list(
  l = 0,
  test = parse(text = "fold_test$x")
)

# parameters for hyperparameter tuning
cv$parameter_grid <- expand.grid(
  k = seq(4, 16, 8)
)
cv$split_type <- "stratified"

# performance parameters
cv$predict_args <- list(type = "response")
cv$performance_metric_args <- list(
  positive = "1",
  negative = "0"
)
cv$performance_metric <- metric("MMCE")

# set data
cv$set_data(
  x = data.matrix(dataset[, -7]),
  y = dataset[, 7]
)

cv$execute()
```

```
}


## ------------------------------------------------
## Method `MLNestedCV$new`
## ------------------------------------------------

if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {
  dataset <- do.call(
    cbind,
    c(sapply(paste0("col", 1:6), function(x) {
      rnorm(n = 500)
      },
      USE.NAMES = TRUE,
      simplify = FALSE
     ),
     list(target = sample(0:1, 500, TRUE))
  ))

  fold_list <- splitTools::create_folds(
    y = dataset[, 7],
    k = 3,
    type = "stratified",
    seed = 123
  )

  cv <- MLNestedCV$new(
    learner = LearnerKnn$new(),
    strategy = "grid",
    fold_list = fold_list,
    k_tuning = 3L,
    seed = 123,
    ncores = 2
  )
}


## ------------------------------------------------
## Method `MLNestedCV$execute`
## ------------------------------------------------

if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {
  dataset <- do.call(
    cbind,
    c(sapply(paste0("col", 1:6), function(x) {
      rnorm(n = 500)
      },
      USE.NAMES = TRUE,
      simplify = FALSE
     ),
     list(target = sample(0:1, 500, TRUE))
```

```
  ))

  fold_list <- splitTools::create_folds(
    y = dataset[, 7],
    k = 3,
    type = "stratified",
    seed = 123
  )

  cv <- MLNestedCV$new(
    learner = LearnerKnn$new(),
    strategy = "grid",
    fold_list = fold_list,
    k_tuning = 3L,
    seed = 123,
    ncores = 2
  )

  # learner args (not optimized)
  cv$learner_args <- list(
    l = 0,
    test = parse(text = "fold_test$x")
  )

  # parameters for hyperparameter tuning
  cv$parameter_grid <- expand.grid(
    k = seq(4, 68, 8)
  )
  cv$split_type <- "stratified"

  # performance parameters
  cv$predict_args <- list(type = "response")
  cv$performance_metric_args <- list(
    positive = "1",
    negative = "0"
  )
  cv$performance_metric <- metric("MMCE")

  # set data
  cv$set_data(
    x = data.matrix(dataset[, -7]),
    y = dataset[, 7]
  )

  cv$execute()
}
```

`MLTuneParameters` *R6 Class to perform hyperparameter tuning experiments*

Description

The MLTuneParameters class is used to construct a parameter tuner object and to perform the tuning of a set of hyperparameters for a specified machine learning algorithm using either a grid search or a Bayesian optimization.

Details

The hyperparameter tuning can be performed with a grid search or a Bayesian optimization. In both cases, each hyperparameter setting is evaluated in a k-fold cross-validation on the dataset specified.

Super classes

mlexperiments::MLBase -> mlexperiments::MLExperimentsBase -> MLTuneParameters

Public fields

parameter_bounds A named list of tuples to define the parameter bounds of the Bayesian hyperparameter optimization. For further details please see the documentation of the rBayesianOptimization package.

parameter_grid A matrix with named columns in which each column represents a parameter that should be optimized and each row represents a specific hyperparameter setting that should be tested throughout the procedure. For strategy = "grid", each row of the parameter_grid is considered as a setting that is evaluated. For strategy = "bayesian", the parameter_grid is passed further on to the initGrid argument of the function rBayesianOptimization::BayesianOptimization() in order to initialize the Bayesian process. The maximum rows considered for initializing the Bayesian process can be specified with the R option option("mlexperiments.bayesian.max_init"), which is set to 4L by default.

optim_args A named list of tuples to define the parameter bounds of the Bayesian hyperparameter optimization. For further details please see the documentation of the rBayesianOptimization package.

split_type A character. The splitting strategy to construct the k cross-validation folds. This parameter is passed further on to the function splitTools::create_folds() and defaults to "stratified".

split_vector A vector If another criteria than the provided y should be considered for generating the cross-validation folds, it can be defined here. It is important, that a vector of the same length as x is provided here.

Methods

Public methods:

- MLTuneParameters$new()
- MLTuneParameters$execute()
- MLTuneParameters$clone()

Method new()**:** Create a new MLTuneParameters object.

Usage:

```
MLTuneParameters$new(
  learner,
  seed,
  strategy = c("grid", "bayesian"),
  ncores = -1L
)
```

Arguments:

`learner` An initialized learner object that inherits from class "MLLearnerBase".

`seed` An integer. Needs to be set for reproducibility purposes.

`strategy` A character. The strategy to optimize the hyperparameters (either "grid" or "bayesian").

`ncores` An integer to specify the number of cores used for parallelization (default: -1L).

Details: For `strategy = "bayesian"`, the number of starting iterations can be set using the R option "mlexperiments.bayesian.max_init", which defaults to 4L. This option reduces the provided initialization grid to contain at most the specified number of rows. This initialization grid is then further passed on to the `initGrid` argument of rBayesianOptimization::BayesianOptimization.

Returns: A new `MLTuneParameters` R6 object.

Examples:

```
if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {
  MLTuneParameters$new(
    learner = LearnerKnn$new(),
    seed = 123,
    strategy = "grid",
    ncores = 2
  )
}
```

Method `execute()`**:** Execute the hyperparameter tuning.

Usage:

```
MLTuneParameters$execute(k)
```

Arguments:

`k` An integer to define the number of cross-validation folds used to tune the hyperparameters.

Details: All results of the hyperparameter tuning are saved in the field `$results` of the `MLTuneParameters` class. After successful execution of the parameter tuning, `$results` contains a list with the items

"summary" A data.table with the summarized results (same as the returned value of the `execute` method).

"best.setting" The best setting (according to the learner's parameter `metric_optimization_higher_better`) identified during the hyperparameter tuning.

"bayesOpt" The returned value of `rBayesianOptimization::BayesianOptimization()` (only for `strategy = "bayesian"`).

Returns: A `data.table` with the results of the hyperparameter optimization. The optimized metric, i.e. the cross-validated evaluation metric is given in the column `metric_optim_mean`. More results are accessible from the field `$results` of the `MLTuneParameters` class.

Examples:

```
if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {
  dataset <- do.call(
    cbind,
    c(sapply(paste0("col", 1:6), function(x) {
      rnorm(n = 500)
      },
      USE.NAMES = TRUE,
      simplify = FALSE
     ),
     list(target = sample(0:1, 500, TRUE))
  ))
  tuner <- MLTuneParameters$new(
    learner = LearnerKnn$new(),
    seed = 123,
    strategy = "grid",
    ncores = 2
  )
  tuner$parameter_bounds <- list(k = c(2L, 80L))
  tuner$parameter_grid <- expand.grid(
    k = seq(4, 68, 8),
    l = 0,
    test = parse(text = "fold_test$x")
  )
  tuner$split_type <- "stratified"
  tuner$optim_args <- list(
    n_iter = 4,
    kappa = 3.5,
    acq = "ucb"
  )

  # set data
  tuner$set_data(
    x = data.matrix(dataset[, -7]),
    y = dataset[, 7]
  )

  tuner$execute(k = 3)
}
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MLTuneParameters$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

rBayesianOptimization::BayesianOptimization(), splitTools::create_folds()

Examples

```
if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {
  knn_tuner <- MLTuneParameters$new(
    learner = LearnerKnn$new(),
    seed = 123,
    strategy = "grid",
    ncores = 2
  )
}


## ------------------------------------------------
## Method `MLTuneParameters$new`
## ------------------------------------------------

if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {
  MLTuneParameters$new(
    learner = LearnerKnn$new(),
    seed = 123,
    strategy = "grid",
    ncores = 2
  )
}


## ------------------------------------------------
## Method `MLTuneParameters$execute`
## ------------------------------------------------

if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {
  dataset <- do.call(
    cbind,
    c(sapply(paste0("col", 1:6), function(x) {
      rnorm(n = 500)
      },
      USE.NAMES = TRUE,
      simplify = FALSE
     ),
     list(target = sample(0:1, 500, TRUE))
  ))
  tuner <- MLTuneParameters$new(
```

```
    learner = LearnerKnn$new(),
    seed = 123,
    strategy = "grid",
    ncores = 2
  )
  tuner$parameter_bounds <- list(k = c(2L, 80L))
  tuner$parameter_grid <- expand.grid(
    k = seq(4, 68, 8),
    l = 0,
    test = parse(text = "fold_test$x")
  )
  tuner$split_type <- "stratified"
  tuner$optim_args <- list(
    n_iter = 4,
    kappa = 3.5,
    acq = "ucb"
  )

  # set data
  tuner$set_data(
    x = data.matrix(dataset[, -7]),
    y = dataset[, 7]
  )

  tuner$execute(k = 3)
}
```

`performance` *performance*

Description

Calculate performance measures from the predictions results.

Usage

```
performance(object, prediction_results, y_ground_truth, type = NULL, ...)
```

Arguments

`object`	An R6 object of class "`MLCrossValidation`" for which the performance should be computed.
`prediction_results`	An object of class "`mlexPredictions`" (the output of the function `predictions()`).
`y_ground_truth`	A vector with the ground truth of the predicted data.
`type`	A character to select a pre-defined set of metrics for "binary" and "regression" tasks. If not specified (default: `NULL`), the metrics that were specified during fitting the `object` are used.
`...`	A list. Further arguments required to compute the performance metrics.

Details

The performance metric has to be specified in the `object` that is used to carry out the experiment, i.e., MLCrossValidation or MLNestedCV. Please note that the option `return_models = TRUE` must be set in the experiment class in order to be able to compute the predictions, which are required to conduct the calculation of the performance.

Value

The function returns a data.table with the computed performance metric of each fold.

Examples

```
if (requireNamespace("measures", quietly = TRUE)) {
  dataset <- do.call(
    cbind,
    c(sapply(paste0("col", 1:6), function(x) {
      rnorm(n = 500)
      },
      USE.NAMES = TRUE,
      simplify = FALSE
     ),
     list(target = sample(0:1, 500, TRUE))
  ))

  fold_list <- splitTools::create_folds(
    y = dataset[, 7],
    k = 3,
    type = "stratified",
    seed = 123
  )

  glm_optimization <- mlexperiments::MLCrossValidation$new(
    learner = LearnerGlm$new(),
    fold_list = fold_list,
    seed = 123
  )

  glm_optimization$learner_args <- list(family = binomial(link = "logit"))
  glm_optimization$predict_args <- list(type = "response")
  glm_optimization$performance_metric_args <- list(
    positive = "1",
    negative = "0"
  )
  glm_optimization$performance_metric <- list(
    auc = metric("AUC"), sensitivity = metric("TPR"),
    specificity = metric("TNR")
  )
  glm_optimization$return_models <- TRUE

  # set data
  glm_optimization$set_data(
    x = data.matrix(dataset[, -7]),
```

```
    y = dataset[, 7]
  )

  cv_results <- glm_optimization$execute()

  # predictions
  preds <- mlexperiments::predictions(
    object = glm_optimization,
    newdata = data.matrix(dataset[, -7]),
    na.rm = FALSE,
    ncores = 2L,
    type = "response"
  )

  # performance
  mlexperiments::performance(
    object = glm_optimization,
    prediction_results = preds,
    y_ground_truth = dataset[, 7],
    positive = "1"
  )

  # performance - binary
  mlexperiments::performance(
    object = glm_optimization,
    prediction_results = preds,
    y_ground_truth = dataset[, 7],
    type = "binary",
    positive = "1"
  )
}
```

`predictions` *predictions*

Description

Apply an R6 object of class "`MLCrossValidation`" to new data to compute predictions.

Usage

```
predictions(object, newdata, na.rm = FALSE, ncores = -1L, ...)
```

Arguments

`object`	An R6 object of class "`MLCrossValidation`" for which the predictions should be computed.
`newdata`	The new data for which predictions should be made using the `model`.

na.rm	A logical. If missings should be removed before computing the mean and standard deviation of the performance across different folds for each observation in newdata.
ncores	An integer to specify the number of cores used for parallelization (default: -1L).
...	A list. Further arguments required to compute the predictions.

Value

The function returns a data.table of class "mlexPredictions"with one row for each observation in newdata and the columns containing the predictions for each fold, along with the mean and standard deviation across all folds.

Examples

```
if (requireNamespace("measures", quietly = TRUE)) {
  dataset <- do.call(
    cbind,
    c(sapply(paste0("col", 1:6), function(x) {
      rnorm(n = 500)
      },
      USE.NAMES = TRUE,
      simplify = FALSE
     ),
     list(target = sample(0:1, 500, TRUE))
  ))

  fold_list <- splitTools::create_folds(
    y = dataset[, 7],
    k = 3,
    type = "stratified",
    seed = 123
  )

  glm_optimization <- mlexperiments::MLCrossValidation$new(
    learner = LearnerGlm$new(),
    fold_list = fold_list,
    seed = 123
  )

  glm_optimization$learner_args <- list(family = binomial(link = "logit"))
  glm_optimization$predict_args <- list(type = "response")
  glm_optimization$performance_metric_args <- list(
     positive = 1,
     negative =0
  )
  glm_optimization$performance_metric <- metric("AUC")
  glm_optimization$return_models <- TRUE

  # set data
  glm_optimization$set_data(
    x = data.matrix(dataset[, -7]),
    y = dataset[, 7]
```

```
  )

  cv_results <- glm_optimization$execute()

  # predictions
  preds <- mlexperiments::predictions(
    object = glm_optimization,
    newdata = data.matrix(dataset[, -7]),
    na.rm = FALSE,
    ncores = 2L,
    type = "response"
  )
  head(preds)
}
```

---

validate_fold_equality *validate_fold_equality*

Description

Validate that the same folds were used in two or more independent experiments.

Usage

```
validate_fold_equality(experiments)
```

Arguments

`experiments`	A list of experiments.

Details

This function can be applied to all implemented experiments, i.e., MLTuneParameters, MLCrossValidation, and MLNestedCV. However, it is required that the list `experiments` contains only experiments of the same class.

Value

Writes messages to the console on the result of the comparison.

Examples

```
if (requireNamespace("measures", quietly = TRUE)  &&
requireNamespace("class", quietly = TRUE)) {
  dataset <- do.call(
    cbind,
    c(sapply(paste0("col", 1:6), function(x) {
```

```
      rnorm(n = 500)
      },
      USE.NAMES = TRUE,
      simplify = FALSE
    ),
    list(target = sample(0:1, 500, TRUE))
))

fold_list <- splitTools::create_folds(
  y = dataset[, 7],
  k = 3,
  type = "stratified",
  seed = 123
)

# GLM
glm_optimization <- mlexperiments::MLCrossValidation$new(
  learner = LearnerGlm$new(),
  fold_list = fold_list,
  seed = 123
)

glm_optimization$learner_args <- list(family = binomial(link = "logit"))
glm_optimization$predict_args <- list(type = "response")
glm_optimization$performance_metric_args <- list(
  positive = "1",
  negative = "0"
)
glm_optimization$performance_metric <- metric("AUC")
glm_optimization$return_models <- TRUE

# set data
glm_optimization$set_data(
  x = data.matrix(dataset[, -7]),
  y = dataset[, 7]
)

glm_cv_results <- glm_optimization$execute()

# KNN
knn_optimization <- mlexperiments::MLCrossValidation$new(
  learner = LearnerKnn$new(),
  fold_list = fold_list,
  seed = 123
)
knn_optimization$learner_args <- list(
  k = 3,
  l = 0,
  test = parse(text = "fold_test$x")
)
knn_optimization$predict_args <- list(type = "prob")
knn_optimization$performance_metric_args <- list(
  positive = "1",
```

```
    negative = "0"
  )
  knn_optimization$performance_metric <- metric("AUC")

  # set data
  knn_optimization$set_data(
    x = data.matrix(dataset[, -7]),
    y = dataset[, 7]
  )

  cv_results_knn <- knn_optimization$execute()

  # validate folds
  validate_fold_equality(
    list(glm_optimization, knn_optimization)
  )
}
```

Index

class::knn(), *4, 5*

handle_cat_vars, 2

kdry::dtr_matrix2df(), *2, 3*

LearnerGlm, 3
LearnerKnn, 4
LearnerLm, 6
LearnerRpart, 7

measures::MMCE(), *5*, *8, 9*
measures::MSE(), *8, 9*
metric, 9
metric(), *10*, *12*, *15*
metric_types_helper, 10
MLBase, 11
MLCrossValidation, *3–5*, *7, 8*, 12, *21–23*, *37*, *40*
mlexperiments::MLBase, *12*, *18*, *24*, *32*
mlexperiments::MLCrossValidation, *24*
mlexperiments::MLExperimentsBase, *12*, *24*, *32*
mlexperiments::MLLearnerBase, *3*, *5, 6*, *8*
MLExperimentsBase, 18
MLLearnerBase, 19
MLNestedCV, *4, 5*, *8*, *20–23*, 24, *37*, *40*
MLTuneParameters, *4, 5*, *8*, *20–23*, 31, *40*

parallel::clusterExport(), *20*
performance, 36
predictions, 38
predictions(), *36*

rBayesianOptimization::BayesianOptimization, *33*
rBayesianOptimization::BayesianOptimization(), *23*, *25*, *32, 33*, *35*
rpart::rpart(), *7–9*
rpart::rpart.control(), *8*

splitTools::create_folds(), *12, 13*, *15*, *24–26*, *28*, *32*, *35*
stats::glm(), *3, 4*
stats::lm(), *6, 7*

validate_fold_equality, 40