Package 'mlr3'

October 29, 2019

Title Machine Learning in R - Next Generation

Version 0.1.4

Description Efficient, object-oriented programming on the building blocks of machine learning. Provides 'R6' objects for tasks, learners, resamplings, and measures. The package is geared towards scalability and larger datasets by supporting parallelization and out-of-memory data-backends like databases. While 'mlr3' focuses on the core computational operations, add-on packages provide additional functionality.

License LGPL-3

URL https://mlr3.mlr-org.com, https://github.com/mlr-org/mlr3

BugReports https://github.com/mlr-org/mlr3/issues

Depends R (>= 3.1.0)

Imports backports, checkmate (>= 1.9.3), data.table, digest, lgr (>= 0.3.0), Metrics, mlbench, mlr3misc (>= 0.1.5), paradox, uuid, R6

Suggests bibtex, callr, datasets, evaluate, future (>= 1.9.0), future.apply (>= 1.1.0), future.callr, Matrix, rpart, testthat, titanic

RdMacros mlr3misc

Encoding UTF-8

LazyData true

NeedsCompilation no

RoxygenNote 6.1.1

Collate 'mlr_reflections.R' 'BenchmarkResult.R' 'DataBackend.R' 'DataBackendCbind.R' 'DataBackendDataTable.R' 'DataBackendMatrix.R' 'DataBackendRbind.R' 'DataBackendRename.R' 'Learner.R' 'LearnerClassif.R' 'mlr_learners.R' 'LearnerClassifDebug.R' 'LearnerClassifFeatureless.R' 'LearnerClassifRpart.R' 'LearnerRegr.R' 'LearnerRegrFeatureless.R' 'LearnerRegrRpart.R' 'Measure.R' 'MeasureClassif.R' 'mlr_measures.R'

'MeasureClassifACC.R' 'MeasureClassifAUC.R'
'MeasureClassifCE.R' 'MeasureClassifConfusion.R'
'MeasureClassifCosts.R' 'MeasureClassifFScore.R'
'MeasureDebug.R' 'MeasureElapsedTime.R' 'MeasureOOBError.R'
'MeasureRegr.R' 'MeasureRegrMAE.R' 'MeasureRegrMSE.R'
'MeasureRegrRMSE.R' 'MeasureSelectedFeatures.R' 'Prediction.R'
'PredictionClassif.R' 'PredictionRegr.R' 'ResampleResult.R'
'Resampling.R' 'mlr_resamplings.R' 'ResamplingBootstrap.R'
'ResamplingCV.R' 'ResamplingCustom.R' 'ResamplingHoldout.R'
'ResamplingRepeatedCV.R' 'ResamplingSubsampling.R' 'Task.R'
'TaskSupervised.R' 'TaskClassif.R' 'mlr_tasks.R'
'TaskClassif_german_credit.R' 'TaskClassif_iris.R'
'TaskClassif_pima.R' 'TaskClassif_sonar.R' 'TaskClassif_spam.R'
'TaskClassif_wine.R' 'TaskClassif_zoo.R' 'TaskGenerator.R'
'mlr_task_generators.R' 'TaskGenerator2DNormals.R'
'TaskGeneratorFriedman1.R' 'TaskGeneratorSmiley.R'
'TaskGeneratorXor.R' 'TaskRegr.R' 'TaskRegr_boston_housing.R'
'TaskRegr_mtcars.R' 'Task_mutators.R' 'as_data_backend.R'
'assertions.R' 'benchmark.R' 'benchmark_grid.R'
'default_measures.R' 'deprecated.R' 'helper.R'
'mlr_coercions.R' 'mlr_sugar.R' 'predict.R' 'reexports.R'
'resample.R' 'worker.R' 'zzz.R'

Author Michel Lang [cre, aut] (<https://orcid.org/0000-0001-9754-0393>),
Bernd Bischl [aut] (<https://orcid.org/0000-0001-6002-6980>),
Jakob Richter [aut] (<https://orcid.org/0000-0003-4481-5554>),
Patrick Schratz [aut] (<https://orcid.org/0000-0003-0748-6624>),
Giuseppe Casalicchio [ctb] (<https://orcid.org/0000-0001-5324-5966>),
Stefan Coors [ctb] (<https://orcid.org/0000-0002-7465-2146>),
Quay Au [ctb] (<https://orcid.org/0000-0002-5252-8902>),
Martin Binder [aut]

Maintainer Michel Lang <michellang@gmail.com>

Repository CRAN

Date/Publication 2019-10-28 23:40:06 UTC

R topics documented:

`mlr3-package` *mlr3: Machine Learning in R - Next Generation*

Description

Efficient, object-oriented programming on the building blocks of machine learning. Provides 'R6' objects for tasks, learners, resamplings, and measures. The package is geared towards scalability and larger datasets by supporting parallelization and out-of-memory data-backends like databases. While 'mlr3' focuses on the core computational operations, add-on packages provide additional functionality.

Author(s)

Maintainer: Michel Lang <michellang@gmail.com> (0000-0001-9754-0393)

Authors:

- Bernd Bischl <bernd_bischl@gmx.net> (0000-0001-6002-6980)
- Jakob Richter <jakob1richter@gmail.com> (0000-0003-4481-5554)
- Patrick Schratz <patrick.schratz@gmail.com> (0000-0003-0748-6624)
- Martin Binder <mlr.developer@mb706.com>

Other contributors:

- Giuseppe Casalicchio <giuseppe.casalicchio@stat.uni-muenchen.de> (0000-0001-5324-5966) [contributor]
- Stefan Coors <mail@stefancoors.de> (0000-0002-7465-2146) [contributor]
- Quay Au <quayau@gmail.com> (0000-0002-5252-8902) [contributor]

See Also

Useful links:

- https://mlr3.mlr-org.com
- https://github.com/mlr-org/mlr3
- Report bugs at https://github.com/mlr-org/mlr3/issues

---

as_benchmark_result *Convert to BenchmarkResult*

Description

Simple S3 method to convert objects to a BenchmarkResult.

Usage

```
as_benchmark_result(x, ...)

## S3 method for class 'ResampleResult'
as_benchmark_result(x, ...)
```

Arguments

x	:: any Object to dispatch on, e.g. a ResampleResult.
...	:: any Currently not used.

Value

(BenchmarkResult).

---

as_data_backend *Create a Data Backend*

Description

Wraps a DataBackend around data.

Usage

```
as_data_backend(data, ...)
```

Arguments

data	:: any Data to create a DataBackend for. For a data.frame() (this includes tibble() from **tibble** and data.table::data.table()) this function creates a DataBackendDataTable. See methods("as_data_backend") for possible input formats. Note that third-party packages may extend this functionality.
...	:: any Additional arguments passed to the respective DataBackend method.

Value

DataBackend.

See Also

Other DataBackend: DataBackendDataTable, DataBackendMatrix, DataBackend

Examples

```
# create a new backend using the iris data:
as_data_backend(iris)
```

---

as_task.character *Object Coercion*

Description

S3 generics and methods to coerce to (lists of) Task, Learner, Resampling, and Measure.

Usage

```
## S3 method for class 'character'
as_task(x, clone = FALSE)

## S3 method for class 'character'
as_tasks(x, clone = FALSE)

## S3 method for class 'character'
as_learner(x, clone = FALSE)

## S3 method for class 'character'
as_learners(x, clone = FALSE)

## S3 method for class 'character'
as_resampling(x, clone = FALSE)

## S3 method for class 'character'
```

```
as_resamplings(x, clone = FALSE)

## S3 method for class 'character'
as_measure(x, task_type = NULL, clone = FALSE)

## S3 method for class 'character'
as_measures(x, task_type = NULL, clone = FALSE)

as_task(x, clone = FALSE)

## S3 method for class 'Task'
as_task(x, clone = FALSE)

as_tasks(x, clone = FALSE)

## S3 method for class 'list'
as_tasks(x, clone = FALSE)

## S3 method for class 'Task'
as_tasks(x, clone = FALSE)

as_learner(x, clone = FALSE)

## S3 method for class 'Learner'
as_learner(x, clone = FALSE)

as_learners(x, clone = FALSE)

## S3 method for class 'list'
as_learners(x, clone = FALSE)

## S3 method for class 'Learner'
as_learners(x, clone = FALSE)

as_resampling(x, clone = FALSE)

## S3 method for class 'Resampling'
as_resampling(x, clone = FALSE)

as_resamplings(x, clone = FALSE)

## S3 method for class 'list'
as_resamplings(x, clone = FALSE)

## S3 method for class 'Resampling'
as_resamplings(x, clone = FALSE)

as_measure(x, task_type = NULL, clone = FALSE)
```

```
## S3 method for class 'NULL'
as_measure(x, task_type = NULL, clone = FALSE)

## S3 method for class 'Measure'
as_measure(x, task_type = NULL, clone = FALSE)

as_measures(x, task_type = NULL, clone = FALSE)

## S3 method for class 'NULL'
as_measures(x, task_type = NULL, clone = FALSE)

## S3 method for class 'list'
as_measures(x, task_type = NULL, clone = FALSE)

## S3 method for class 'Measure'
as_measures(x, task_type = NULL, clone = FALSE)
```

Arguments

x	:: any Object to coerce.
clone	:: logical(1) If TRUE, ensures that the returned object is not the same as the input x, e.g. by cloning it or constructing it from a mlr3misc::Dictionary.
task_type	:: character(1) Used if x is NULL to construct a default measure for the respective task type. The default measures are stored in mlr_reflections$default_measures.

Value

Coerced object. The default method will return the object as-is. Failed coercions have to be handled by on of the assertions in mlr_assertions.

Examples

```
# convert single measure to list of measures
measure = msr("classif.ce")
as_measures(measure)
```

---

benchmark *Benchmark Multiple Learners on Multiple Tasks*

Description

Runs a benchmark on arbitrary combinations of tasks (Task), learners (Learner), and resampling strategies (Resampling), possibly in parallel.

Usage

```
benchmark(design, store_models = FALSE)
```

Arguments

design	:: data.frame() Data frame (or data.table::data.table()) with three columns: "task", "learner", and "resampling". Each row defines a resampling by providing a Task, Learner and an instantiated Resampling strategy. The helper function benchmark_grid() can assist in generating an exhaustive design (see examples) and instantiate the Resamplings per Task.
store_models	:: logical(1) Keep the fitted model after the test set has been predicted? Set to TRUE if you want to further analyse the models or want to extract information like variable importance.

Value

BenchmarkResult.

Parallelization

This function can be parallelized with the **future** package. One job is one resampling iteration, and all jobs are send to an apply function from **future.apply** in a single batch. To select a parallel backend, use future::plan().

Logging

The **mlr3** uses the **lgr** package for logging. **lgr** supports multiple log levels which can be queried with getOption("lgr.log_levels").

To suppress output and reduce verbosity, you can lower the log from the default level "info" to "warn":

```
lgr::get_logger("mlr3")$set_threshold("warn")
```

To get additional log output for debugging, increase the log level to "debug" or "trace":

```
lgr::get_logger("mlr3")$set_threshold("debug")
```

To log to a file or a data base, see the documentation of lgr::lgr-package.

Note

The fitted models are discarded after the predictions have been scored in order to reduce memory consumption. If you need access to the models for later analysis, set store_models to TRUE.

Examples

```
# benchmarking with benchmark_grid()
tasks = lapply(c("iris", "sonar"), tsk)
learners = lapply(c("classif.featureless", "classif.rpart"), lrn)
resamplings = rsmp("cv", folds = 3)

design = benchmark_grid(tasks, learners, resamplings)
print(design)

set.seed(123)
bmr = benchmark(design)

## data of all resamplings
head(as.data.table(bmr))

## aggregated performance values
aggr = bmr$aggregate()
print(aggr)

## Extract predictions of first resampling result
rr = aggr$resample_result[[1]]
as.data.table(rr$prediction())

# benchmarking with a custom design:
# - fit classif.featureless on iris with a 3-fold CV
# - fit classif.rpart on sonar using a holdout
tasks = list(tsk("iris"), tsk("sonar"))
learners = list(lrn("classif.featureless"), lrn("classif.rpart"))
resamplings = list(rsmp("cv", folds = 3), rsmp("holdout"))

design = data.table::data.table(
  task = tasks,
  learner = learners,
  resampling = resamplings
)

## instantiate resamplings
design$resampling = Map(
  function(task, resampling) resampling$clone()$instantiate(task),
  task = design$task, resampling = design$resampling
)

## run benchmark
bmr = benchmark(design)
print(bmr)

## get the training set of the 2nd iteration of the featureless learner on iris
rr = bmr$aggregate()[learner_id == "classif.featureless"]$resample_result[[1]]
rr$resampling$train_set(2)
```

BenchmarkResult *Container for Results of* benchmark()

Description

This is the result container object returned by benchmark(). A BenchmarkResult consists of the data row-binded data of multiple ResampleResults, which can easily be re-constructed.

Note that all stored objects are accessed by reference. Do not modify any object without cloning it first.

Format

R6::R6Class object.

Construction

```
bmr = BenchmarkResult$new(data = data.table())
```

- data :: data.table::data.table()
  Table with data for one resampling iteration per row: Task, Learner, Resampling, iteration (integer(1)), Prediction, and the unique hash uhash (character(1)) of the corresponding ResampleResult. Additional columns are kept in the resulting object.

Fields

- data :: data.table::data.table()
  Internal data storage with one row per resampling iteration. Can be joined with $rr_data by joining on column "hash". We discourage users to directly work with this table.
  Package develops on the other hand may opt to add additional columns here. These columns are preserved in all mutators.
- rr_data :: data.table::data.table()
  Internal data storage with one row per ResampleResult. Can be joined with $data by joining on column "hash". Not used in mlr3 directly, but can be exploited by add-on packages.
  Package develops may opt to add additional columns here. These columns are preserved in all mutators.
- task_type :: character(1)
  Task type of objects in the BenchmarkResult. All stored objects (Task, Learner, Prediction) in a single BenchmarkResult are required to have the same task type, e.g., "classif" or "regr".
- tasks :: data.table::data.table()
  Table of used tasks with three columns: "task_hash" (character(1)), "task_id" (character(1)) and "task" (Task).
- learners :: data.table::data.table()
  Table of used learners with three columns: "learner_hash" (character(1)), "learner_id" (character(1)) and "learner" (Learner).

- resamplings :: data.table::data.table()
  Table of used resamplings with three columns: "resampling_hash" (character(1)), "resampling_id" (character(1)) and "resampling" (Resampling).
- n_resample_results :: integer(1)
  Returns the number of stored ResampleResults.
- uhashes :: character()
  Vector of unique hashes of all included ResampleResults.

Methods

- aggregate(measures = NULL, ids = TRUE, uhashes = FALSE, params = FALSE, conditions = FALSE)
  (list of Measure, logical(1), logical(1), logical(1), logical(1)) -> data.table::data.table()
  Returns a result table where resampling iterations are combined into ResampleResults. A column with the aggregated performance score is added for each Measure, named with the id of the respective measure.

  For convenience, the following parameters can be set to extract more information from the returned ResampleResult:

  - uhashes :: logical(1)
    Adds the uhash values of the ResampleResult as extra character column "uhash".
  - ids :: logical(1)
    Adds object ids ("task_id", "learner_id", "resampling_id") as extra character columns.
  - params :: logical(1)
    Adds the hyperparameter values as extra list column "params". You can unnest them with mlr3misc::unnest().
  - conditions :: logical(1)
    Adds the number of resampling iterations with at least one warning as extra integer column "warnings", and the number of resampling iterations with errors as extra integer column "errors".
- score(measures = NULL, ids = TRUE)
  (list of Measure, logical(1)) -> data.table::data.table()
  Returns a table with one row for each resampling iteration, including all involved objects: Task, Learner, Resampling, iteration number (integer(1)), and Prediction. If ids is set to TRUE, character column of extracted ids are added to the table for convenient filtering: "task_id", "learner_id", and "resampling_id". Additionally calculates the provided performance measures and binds the performance as extra columns. These columns are named using the id of the respective Measure.
- resample_result(i = NULL, uhash = NULL)
  (integer(1), character(1)) -> ResampleResult
  Retrieve the i-th ResampleResult, by position or by unique hash uhash. i and uhash are mutually exclusive.
- combine(bmr)
  (BenchmarkResult | NULL) -> self
  Fuses a second BenchmarkResult into itself, mutating the BenchmarkResult in-place. If bmr is NULL, simply returns self.

- `help()`
  () -> NULL
  Opens the help page for this object.

S3 Methods

- `as.data.table(bmr)`
  BenchmarkResult -> `data.table::data.table()`
  Returns a copy of the internal data.

Examples

```
set.seed(123)
learners = list(
  lrn("classif.featureless", predict_type = "prob"),
  lrn("classif.rpart", predict_type = "prob")
)

design = benchmark_grid(
  tasks = list(tsk("sonar"), tsk("spam")),
  learners = learners,
  resamplings = rsmp("cv", folds = 3)
)
print(design)

bmr = benchmark(design)
print(bmr)

bmr$tasks
bmr$learners

# first 5 individual resamplings
head(as.data.table(bmr, measures = c("classif.acc", "classif.auc")), 5)

# aggregate results
bmr$aggregate()

# aggregate results with hyperparameters as separate columns
mlr3misc::unnest(bmr$aggregate(params = TRUE), "params")

# extract resample result for classif.rpart
rr = bmr$aggregate()[learner_id == "classif.rpart", resample_result][[1]]
print(rr)

# access the confusion matrix of the first resampling iteration
rr$predictions()[[1]]$confusion
```

---

benchmark_grid *Generate a Benchmark Grid Design*

Description

Takes a lists of Task, a list of Learner and a list of Resampling to generate a design in an `expand.grid()` fashion (a.k.a. cross join or Cartesian product).

Resampling strategies are not allowed to be instantiated when passing the argument, and instead will be instantiated per task internally.

Usage

```
benchmark_grid(tasks, learners, resamplings)
```

Arguments

`tasks`	:: list of Task.
`learners`	:: list of Learner.
`resamplings`	:: list of Resampling.

Value

(`data.table::data.table()`) with the cross product of the input vectors.

Examples

```
tasks = list(tsk("iris"), tsk("sonar"))
learners = list(lrn("classif.featureless"), lrn("classif.rpart"))
resamplings = list(rsmp("cv"), rsmp("subsampling"))
benchmark_grid(tasks, learners, resamplings)
```

`confusion_measures` *Calculate Confusion Measures*

Description

Based on a 2x2 confusion matrix for binary classification problems, allows to calculate various performance measures. Implemented are the following measures based on `https://en.wikipedia.org/wiki/Template:DiagnosticTesting_Diagram`:

- `"tp"`: True Positives.
- `"fn"`: False Negatives.
- `"fp"`: False Positives.
- `"tn"`: True Negatives.
- `"tpr"`: True Positive Rate.
- `"fnr"`: False Negative Rate.
- `"fpr"`: False Positive Rate.
- `"tnr"`: True Negative Rate.

- "ppv": Positive Predictive Value.
- "fdr": False Discovery Rate.
- "for": False Omission Rate.
- "npv": Negative Predictive Value.
- "dor": Diagnostic Odds Ratio.
- "f1": F1 Measure.
- "precision": Alias for "ppv".
- "recall": Alias for "tpr".
- "sensitivity": Alias for "tpr".
- "specificity": Alias for "tnr".

If the denominator is 0, the returned score is NA.

Usage

```
confusion_measures(m, type = NULL)
```

Arguments

m	:: matrix() Confusion matrix, e.g. as returned by field confusion of PredictionClassif. Truth is in columns, predicted response is in rows.
type	:: character() Selects the measure to use. See description for possible values.

Value

(named numeric()) of confusion measures.

Examples

```
task = tsk("german_credit")
learner = lrn("classif.rpart")
p = learner$train(task)$predict(task)
round(confusion_measures(p$confusion), 2)
```

DataBackend *DataBackend*

Description

This is the abstract base class for data backends.

Data Backends provide a layer of abstraction for various data storage systems. It is not recommended to work directly with the DataBackend. Instead, all data access is handled transparently via the Task.

To connect to out-of-memory database management systems such as SQL servers, see **mlr3db**.

The required set of fields and methods to implement a custom DataBackend is listed in the respective sections. See DataBackendDataTable or DataBackendMatrix for exemplary implementations of the interface.

Format

R6::R6Class object.

Construction

Note: This object is typically constructed via a derived classes, e.g. DataBackendDataTable or DataBackendMatrix, or via the S3 method `as_data_backend()`.

```
DataBackend$new(data, primary_key = NULL, data_formats = "data.table")
```

- `data` :: `any`
  The format of the input data depends on the specialization. E.g., DataBackendDataTable expects a `data.table::data.table()` and DataBackendMatrix expects a `Matrix::Matrix()` constructed with the **Matrix** package.
- `primary_key` :: `character(1)`
  Each DataBackend needs a way to address rows, which is done via a column of unique values, referenced here by `primary_key`. The use of this variable may differ between backends.
- data_formats (`character()`)
  Set of supported formats, e.g. `"data.table"` or `"Matrix"`.

Fields

- `nrow` :: `integer(1)`
  Number of rows (observations).
- `ncol` :: `integer(1)`
  Number of columns (variables), including the primary key column.
- `colnames` :: `character()`
  Returns vector of all column names, including the primary key column.
- `rownames` :: (`integer()` | `character()`)
  Returns vector of all distinct row identifiers, i.e. the primary key column.

- hash :: character(1)
  Returns a unique hash for this backend. This hash is cached.
- data_formats :: character()
  Vector of supported data formats. A specific format can be chosen in the $data() method.

Methods

- data(rows = NULL, cols = NULL, format = "data.table")
  (integer() | character(), character()) -> any
  Returns a slice of the data in the specified format. Currently, the only supported formats are "data.table" and "Matrix". The rows must be addressed as vector of primary key values, columns must be referred to via column names. Queries for rows with no matching row id and queries for columns with no matching column name are silently ignored. Rows are guaranteed to be returned in the same order as rows, columns may be returned in an arbitrary order. Duplicated row ids result in duplicated rows, duplicated column names lead to an exception.
- distinct(rows, cols, na_rm = TRUE)
  (integer() | character(), character(), logical(1)) -> named list()
  Returns a named list of vectors of distinct values for each column specified. If na_rm is TRUE, missing values are removed from the returned vectors of distinct values. Non-existing rows and columns are silently ignored.

  If rows is NULL, all possible distinct values will be returned, even if the value is not present in the data. This affects factor-like variables with empty levels, if supported by the backend.
- head(n = 6)
  integer(1) -> data.table::data.table()
  Returns the first up-to n rows of the data as data.table::data.table().
- missings(rows, cols)
  (integer() | character(), character()) -> named integer()
  Returns the number of missing values per column in the specified slice of data. Non-existing rows and columns are silently ignored.

See Also

Extension Packages: **mlr3db**

Other DataBackend: DataBackendDataTable, DataBackendMatrix, as_data_backend

Examples

```
data = data.table::data.table(id = 1:5, x = runif(5), y = sample(letters[1:3], 5, replace = TRUE))

b = DataBackendDataTable$new(data, primary_key = "id")
print(b)
b$head(2)
b$data(rows = 1:2, cols = "x")
b$distinct(rows = b$rownames, "y")
b$missings(rows = b$rownames, cols = names(data))
```

DataBackendDataTable *DataBackend for data.table*

Description

DataBackend for **data.table** as an in-memory data base.

Format

R6::R6Class object inheriting from DataBackend.

Construction

```
DataBackendDataTable$new(data, primary_key = NULL)
as_data_backend(data, primary_key = NULL, ...)
```

- `data` :: `data.table::data.table()`
  The input `data.table::data.table()`.
- `primary_key` :: `character(1)`
  Name of the primary key column.

`DataBackendDataTable` does not copy the input data, while `as_data_backend` calls `data.table::copy()`. `as_data_backend` creates a primary key column as integer column if `primary_key` is `NULL`.

Fields

See DataBackend.

Methods

See DataBackend.

See Also

Other DataBackend: `DataBackendMatrix`, `DataBackend`, `as_data_backend`

Examples

```
data = as.data.table(iris)
data$id = seq_len(nrow(iris))
b = DataBackendDataTable$new(data = data, primary_key = "id")
print(b)
b$head()
b$data(rows = 100:101, cols = "Species")

b$nrow
head(b$rownames)

b$ncol
```

```
b$colnames

# alternative construction
as_data_backend(iris)
```

`DataBackendMatrix` *DataBackend for Matrix*

Description

DataBackend for **Matrix**. Data is stored as (sparse) matrix.

Format

R6::R6Class object inheriting from DataBackend.

Construction

```
DataBackendMatrix$new(data, primary_key = NULL)
as_data_backend(data, primary_key = NULL, ...)
```

- `data` :: `Matrix::Matrix()`.
- `primary_key` :: `character(1)`
  Not supported by this backend. Rows are addresses by their `rownames()`. If the matrix does not have row names, integer row indices are used.

Fields

See DataBackend.

Methods

See DataBackend.

See Also

Other DataBackend: `DataBackendDataTable`, `DataBackend`, `as_data_backend`

Examples

```
requireNamespace("Matrix")
data = Matrix::Matrix(sample(0:1, 20, replace = TRUE), ncol = 2)
colnames(data) = c("x1", "x2")
rownames(data) = paste0("row_", 1:10)

b = as_data_backend(data)
b$head()
b$data(b$rownames[1:3], b$colnames, data_format = "Matrix")
```

default_measures *Get a Default Measure*

Description

Gets the default measures using the information in mlr_reflections$default_measures:

- "classif.ce" for classification ("classif").
- "regr.mse" for regression ("regr").
- Add-on package may register additional default measures for their own task types.

Usage

```
default_measures(task_type)
```

Arguments

task_type	:: character(1) Get the default measure for the task type task_type, e.g., "classif" or "regr". If task_type is NULL, an empty list is returned.

Value

list of Measure.

Examples

```
default_measures("classif")
default_measures("regr")
```

Learner *Learner Class*

Description

This is the abstract base class for learner objects like LearnerClassif and LearnerRegr.

Learners consist of the following parts:

- Methods train() and predict() which call train_internal() and predict_internal().
- The fitted model, after calling train().
- A paradox::ParamSet which stores meta-information about available hyperparameters, and also stores hyperparameter settings.
- Meta-information about the requirements and capabilities of the learner.

Predefined learners are stored in the mlr3misc::Dictionary mlr_learners, e.g. classif.rpart or regr.rpart. A guide on how to extend **mlr3** with custom learners can be found in the mlr3book.

Format

R6::R6Class object.

Construction

Note: This object is typically constructed via a derived classes, e.g. LearnerClassif or LearnerRegr.

```
l = Learner$new(id, task_type, param_set = ParamSet$new(), predict_types = character(),
    feature_types = character(), properties = character(), packages = character())
```

- `id` :: `character(1)`
  Identifier for the learner.
- `task_type` :: `character(1)`
  Type of the task the learner can operator on. E.g., `"classif"` or `"regr"`.
- `param_set` :: paradox::ParamSet
  Set of hyperparameters.
- `predict_types` :: `character()`
  Supported predict types. Must be a subset of `mlr_reflections$learner_predict_types`.
- `predict_sets` :: `character()`
  Sets to predict on during `resample()`/`benchmark()`. Creates and stores a separate Prediction object for each set. The individual sets can be combined via getters in ResampleResult/BenchmarkResult, or Measures can be set to operate on subsets of the calculated prediction sets. Must be a non-empty subset of `("train","test")`. Default is `"test"`.
- `feature_types` :: `character()`
  Feature types the learner operates on. Must be a subset of `mlr_reflections$task_feature_types`.
- `properties` :: `character()`
  Set of properties of the learner. Must be a subset of `mlr_reflections$learner_properties`.
  The following properties are currently standardized and understood by learners in **mlr3**:
  - `"missings"`: The learner can handle missing values in the data.
  - `"weights"`: The learner supports observation weights.
  - `"importance"`: The learner supports extraction of importance scores, i.e. comes with a `importance()` extractor function (see section on optional extractors).
  - `"selected_features"`: The learner supports extraction of the set of selected features, i.e. comes with a `selected_features()` extractor function (see section on optional extractors).
  - `"oob_error"`: The learner supports extraction of estimated out of bag error, i.e. comes with a `oob_error()` extractor function (see section on optional extractors).
- `data_formats` :: `character()`
  Vector of supported data formats which can be processed during `$train()` and `$predict()`. Defaults to `"data.table"`.
- `packages` :: `character()`
  Set of required packages. Note that these packages will be loaded via `requireNamespace()`, and are not attached.
- `man` :: `character(1)`
  String in the format `[pkg]::[topic]` pointing to a manual page for this object.

Fields

- `id` :: `character(1)`
  Identifier of the learner.
- `task_type` :: `character(1)`
  Stores the type of class this learner can operate on, e.g. `"classif"` or `"regr"`. A complete list of task types is stored in `mlr_reflections$task_types`.
- `param_set` :: paradox::ParamSet
  Description of available hyperparameters and hyperparameter settings.
- `predict_types` :: `character()`
  Stores the possible predict types the learner is capable of. A complete list of candidate predict types, grouped by task type, is stored in `mlr_reflections$learner_predict_types`.
- `predict_type` :: `character(1)`
  Stores the currently selected predict type. Must be an element of `l$predict_types`.
- `feature_types` :: `character()`
  Stores the feature types the learner can handle, e.g. `"logical"`, `"numeric"`, or `"factor"`. A complete list of candidate feature types, grouped by task type, is stored in `mlr_reflections$task_feature_types`.
- `properties` :: `character()`
  Stores a set of properties/capabilities the learner has. A complete list of candidate properties, grouped by task type, is stored in `mlr_reflections$learner_properties`.
- `packages` :: `character()`
  Stores the names of required packages.
- `state` :: `NULL` | named `list()`
  Current (internal) state of the learner. Contains all information learnt during `train()` and `predict()`. Do not access elements from here directly.
- `encapsulate` (named `character()`)
  How to call the code in `train_internal()` and `predict_internal()`. Must be a named character vector with names `"train"` and `"predict"`. Possible values are `"none"`, `"evaluate"` and `"callr"`. See `mlr3misc::encapsulate()` for more details.
- `fallback` (Learner)
  Learner which is fitted to impute predictions in case that either the model fitting or the prediction of the top learner is not successful. Requires you to enable encapsulation, otherwise errors are not caught and the execution is terminated before the fallback learner kicks in.
- `hash` :: `character(1)`
  Hash (unique identifier) for this object.
- `model` :: `any`
  The fitted model. Only available after `$train()` has been called.
- `timings` :: `numeric(2)`
  Elapsed time in seconds for the steps `"train"` and `"predict"`. Measured via `mlr3misc::encapsulate()`.
- `log` :: `data.table::data.table()`
  Returns the output (including warning and errors) as table with columns `"stage"` (train or predict), `"class"` (output, warning, error) and `"msg"` (`character()`).
- `warnings` :: `character()`
  Returns the logged warnings as vector.
- `errors` :: `character()`
  Returns the logged errors as vector.

Methods

- `train(task,row_ids = NULL))`
  (Task, `integer() | character()`) -> `self`
  Train the learner on the row ids of the provided Task. Mutates the learner by reference, i.e. stores the model alongside other objects in field `$state`.
- `predict(task,row_ids = NULL)`
  (Task, `integer() | character()`) -> Prediction
  Uses the data stored during `$train()` in `$state` to create a new Prediction based on the provided `row_ids` of the `task`.
- `predict_newdata(newdata,task = NULL)`
  (`data.frame()`, Task) -> Prediction
  Uses the model fitted during `$train()` in to create a new Prediction based on the new data in `newdata`. Object `task` is the task used during `$train()` and required for conversions of `newdata`. If the learner's `$train()` method has been called, there is a (size reduced) version of the training task stored in the learner. If the learner has been fitted via `resample()` or `benchmark()`, you need to pass the corresponding task stored in the ResampleResult or BenchmarkResult, respectively.
- `help()`
  () -> NULL
  Opens the corresponding help page referenced by `$man`.

Optional Extractors

Specific learner implementations are free to implement additional getters to ease the access of certain parts of the model in the inherited subclasses.

For the following operations, extractors are standardized:

- `importance(...)`: Returns the feature importance score as numeric vector. The higher the score, the more important the variable. The returned vector is named with feature names and sorted in decreasing order. Note that the model might omit features it has not used at all. The learner must be tagged with property `"importance"`. To filter variables using the importance scores, use package **mlr3filters**.
- `selected_features(...)`: Returns a subset of selected features as `character()`. The learner must be tagged with property `"selected_features"`.
- `oob_error(...)`: Returns the out-of-bag error of the model as `numeric(1)`. The learner must be tagged with property `"oob_error"`.

Setting Hyperparameters

All information about hyperparameters is stored in the slot `param_set` which is a paradox::ParamSet. The printer gives an overview about the ids of available hyperparameters, their storage type, lower and upper bounds, possible levels (for factors), default values and assigned values. To set hyperparameters, assign a named list to the subslot `values`:

```
lrn = lrn("classif.rpart")
lrn$param_set$values = list(minsplit = 3, cp = 0.01)
```

Note that this operation replaces all previously set hyperparameter values. If you only intend to change one specific hyperparameter value and leave the others as-is, you can use the helper function `mlr3misc::insert_named()`:

```
lrn$param_set$values = mlr3misc::insert_named(lrn$param_set$values, list(cp = 0.001))
```

If the learner has additional hyperparameters which are not encoded in the ParamSet, you can easily extend the learner. Here, we add a hyperparameter with id `"foo"` possible levels `"a"` and `"b"`:

```
lrn$param_set$add(paradox::ParamFct$new("foo", levels = c("a", "b")))
```

See Also

Other Learner: `LearnerClassif`, `LearnerRegr`, `mlr_learners`

`LearnerClassif` *Classification Learner*

Description

This Learner specializes Learner for classification problems.

Many predefined learners can be found in the mlr3misc::Dictionary mlr_learners after loading the **mlr3learners** package.

Format

R6::R6Class object inheriting from Learner.

Construction

```
l = LearnerClassif$new(id, param_set = ParamSet$new(), predict_types = character(), feature_types = cha
  properties = character(), data_formats = "data.table", packages = character(), man = NA_character_)
```

For a description of the arguments, see Learner. `task_type` is set to `"classif"`.

Possible values for `predict_types` are passed to and converted by PredictionClassif:

- `"response"`: Predicts a class label for each observation in the test set.
- `"prob"`: Predicts the posterior probability for each class for each observation in the test set.

Additional learner properties include:

- `"twoclass"`: The learner works on binary classification problems.
- `"multiclass"`: The learner works on multiclass classification problems.

Fields

See Learner.

Methods

See Learner.

See Also

Example classification learners: `classif.rpart`

Other Learner: `LearnerRegr`, `Learner`, `mlr_learners`

Examples

```
# get all classification learners from mlr_learners:
lrns = mlr_learners$mget(mlr_learners$keys("^classif"))
names(lrns)

# get a specific learner from mlr_learners:
lrn = lrn("classif.rpart")
print(lrn)

# train the learner:
task = tsk("iris")
lrn$train(task, 1:120)

# predict on new observations:
lrn$predict(task, 121:150)$confusion
```

LearnerClassifDebug *Classification Learner for Debugging*

Description

A simple LearnerClassif used primarily in the unit tests and for debugging purposes. If no hyperparameter is set, it simply constantly predicts a randomly selected label. The following hyperparameters trigger the following actions:

message_train: Outputs a message during train if the parameter value exceeds `runif(1)`.

message_predict: Outputs a message during predict if the parameter value exceeds `runif(1)`.

warning_train: Signals a warning during train if the parameter value exceeds `runif(1)`.

warning_predict: Signals a warning during predict if the parameter value exceeds `runif(1)`.

error_train: Raises an exception during train if the parameter value exceeds `runif(1)`.

error_predict: Raises an exception during predict if the parameter value exceeds `runif(1)`.

segfault_train: Provokes a segfault during train if the parameter value exceeds `runif(1)`.

segfault_predict: Provokes a segfault during predict if the parameter value exceeds `runif(1)`.

predict_missing Ratio of predictions which will be NA.

save_tasks: Saves input task in `model` slot during training and prediction.

x: Numeric parameter. Has no effect.

Note that segfaults may not work on your operating system. Also note that if they work, they will tear down your R session immediately!

Format

R6::R6Class inheriting from LearnerClassif.

Construction

```
LearnerClassifDebug$new()
mlr_learners$get("classif.debug")
lrn("classif.debug")
```

See Also

Dictionary of Learners: mlr_learners

`as.data.table(mlr_learners)` for a complete table of all (also dynamically created) Learner implementations.

Examples

```
learner = lrn("classif.debug")
learner$param_set$values = list(message_train = 1, save_tasks = TRUE)

# this should signal a message
task = tsk("iris")
learner$train(task)
learner$predict(task)

# task_train and task_predict are the input tasks for train() and predict()
names(learner$model)
```

LearnerClassifFeatureless

Featureless Classification Learner

Description

A simple LearnerClassif which only analyses the labels during train, ignoring all features. Hyperparameter `method` determines the mode of operation during prediction:

mode: Predicts the most frequent label. If there are two or more labels tied, randomly selects one per prediction.

sample: Randomly predict a label uniformly.

weighed.sample: Randomly predict a label, with probability estimated from the training distribution.

Format

R6::R6Class inheriting from LearnerClassif.

Construction

```
LearnerClassifFeatureless$new()
mlr_learners$get("classif.featureless")
lrn("classif.featureless")
```

See Also

Dictionary of Learners: mlr_learners

`as.data.table(mlr_learners)` for a complete table of all (also dynamically created) Learner implementations.

LearnerClassifRpart *Classification Tree Learner*

Description

A LearnerClassif for a classification tree implemented in `rpart::rpart()` in package **rpart**. Parameter `xval` is set to 0 in order to save some computation time.

Format

R6::R6Class inheriting from LearnerClassif.

Construction

```
LearnerClassifRpart$new()
mlr_learners$get("classif.rpart")
lrn("classif.rpart")
```

References

Breiman L, Friedman JH, Olshen RA, Stone CJ (2017). *Classification And Regression Trees*. Routledge. doi: 10.1201/9781315139470.

See Also

Dictionary of Learners: mlr_learners

`as.data.table(mlr_learners)` for a complete table of all (also dynamically created) Learner implementations.

LearnerRegr *Regression Learner*

Description

This Learner specializes Learner for regression problems.

Many predefined learners can be found in the mlr3misc::Dictionary mlr_learners after loading the **mlr3learners** package.

Format

R6::R6Class object inheriting from Learner.

Construction

```
l = LearnerRegr$new(id, param_set = ParamSet$new(), predict_types = character(), feature_types = charac
  properties = character(), data_formats = "data.table", packages = character(), man = NA_character_)
```

For a description of the arguments, see Learner. `task_type` is set to `"regr"`.

Possible values for `predict_types` are passed to and converted by PredictionRegr:

- `"response"`: Predicts a numeric response for each observation in the test set.
- `"se"`: Predicts the standard error for each value of response for each observation in the test set.

Fields

See Learner.

Methods

See Learner.

See Also

Example regression learners: `regr.rpart`

Other Learner: `LearnerClassif`, `Learner`, `mlr_learners`

Examples

```
# get all regression learners from mlr_learners:
lrns = mlr_learners$mget(mlr_learners$keys("^regr"))
names(lrns)

# get a specific learner from mlr_learners:
mlr_learners$get("regr.rpart")
lrn("classif.featureless")
```

LearnerRegrFeatureless *Featureless Regression Learner*

Description

A simple LearnerRegr which only analyses the response during train, ignoring all features. If hyperparameter `robust` is `FALSE` (default), constantly predicts `mean(y)` as response and `sd(y)` as standard error. If `robust` is `TRUE`, `median()` and `madn()` are used instead of `mean()` and `sd()`, respectively.

Format

R6::R6Class inheriting from LearnerRegr.

Construction

```
LearnerRegrFeatureless$new()
mlr_learners$get("regr.featureless")
lrn("regr.featureless")
```

See Also

Dictionary of Learners: mlr_learners

`as.data.table(mlr_learners)` for a complete table of all (also dynamically created) Learner implementations.

LearnerRegrRpart *Regression Tree Learner*

Description

A LearnerRegr for a regression tree implemented in `rpart::rpart()` in package **rpart**. Parameter `xval` is set to 0 in order to save some computation time.

Format

R6::R6Class inheriting from LearnerRegr.

Construction

```
LearnerRegrRpart$new()
mlr_learners$get("regr.rpart")
lrn("regr.rpart")
```

References

Breiman L, Friedman JH, Olshen RA, Stone CJ (2017). *Classification And Regression Trees*. Routledge. doi: 10.1201/9781315139470.

See Also

Dictionary of Learners: mlr_learners

`as.data.table(mlr_learners)` for a complete table of all (also dynamically created) Learner implementations.

`Measure` *Measure Class*

Description

This is the abstract base class for measures like MeasureClassif and MeasureRegr.

Measures are classes around tailored around two functions:

1. A function `score` which quantifies the performance by comparing true and predicted response.
2. A function `aggregator` which combines multiple performance scores returned by `calculate` to a single numeric value.

In addition to these two functions, meta-information about the performance measure is stored.

Predefined measures are stored in the mlr3misc::Dictionary mlr_measures, e.g. `classif.auc` or `time_train`. A guide on how to extend **mlr3** with custom measures can be found in the mlr3book.

Format

R6::R6Class object.

Construction

Note: This object is typically constructed via a derived classes, e.g. MeasureClassif or MeasureRegr.

```
m = Measure$new(id, task_type = NA, range = c(-Inf, Inf), minimize = NA, aggregator = NULL, properties =
  predict_sets = "test", task_properties = character(), packages = character(), man = NA_character_)
```

- `id` :: `character(1)`
  Identifier for the measure.
- `task_type` :: `character(1)`
  Type of the task the measure can operator on. E.g., `"classif"` or `"regr"`.
- `range` :: `numeric(2)`
  Feasible range for this measure as `c(lower_bound,upper_bound)`. Both bounds may be infinite.

- minimize :: logical(1)
  Set to TRUE if good predictions correspond to small values, and to FALSE if good predictions correspond to large values. If set to NA (default), tuning this measure is not possible.
- aggregator :: function(x)
  Function to aggregate individual performance scores x where x is a numeric vector. If NULL, defaults to mean().
- properties :: character()
  Properties of the measure. Must be a subset of mlr_reflections$measure_properties. Supported by mlr3:
  - "requires_task" (requires the complete Task),
  - "requires_learner" (requires the trained Learner),
  - "requires_train_set" (requires the training indices from the Resampling), and
  - "na_score" (the measure is expected to occasionally return NA).
- predict_type :: character(1)
  Required predict type of the Learner. Possible values are stored in mlr_reflections$learner_predict_types.
- predict_sets :: character()
  Prediction sets to operate on, used in aggregate() to extract the matching predict_sets from the ResampleResult. Multiple predict sets are calculated by the respective Learner during resample()/benchmark(). Must be a non-empty subset of c("train","test"). If multiple sets are provided, these are first combined to a single prediction object. Default is "test".
- task_properties :: character()
  Required task properties, see Task.
- packages :: character()
  Set of required packages. Note that these packages will be loaded via requireNamespace(), and are not attached.
- man :: character(1)
  String in the format [pkg]::[topic] pointing to a manual page for this object.

Fields

All variables passed to the constructor.

Methods

- aggregate(rr)
  ResampleResult -> numeric(1)
  Aggregates multiple performance scores into a single score using the aggregator function of the measure. Operates on the Predictions of ResampleResult with matching predict_sets.
- score(prediction,task = NULL,learner = NULL,train_set = NULL)
  ((named list of) Prediction, Task, Learner, integer() | character()) -> numeric(1)
  Takes a Prediction (or a list of Prediction objects named with valid predict_sets) and calculates a numeric score. If the measure if flagged with the properties "requires_task", "requires_learner" or "requires_train_set", you must additionally pass the respective Task, the trained Learner or the training set indices. This is handled internally during resample()/benchmark().

- `help()`
  () -> NULL
  Opens the corresponding help page referenced by `$man`.

See Also

Other Measure: `MeasureClassif`, `MeasureRegr`, `mlr_measures`

`MeasureClassif` *Classification Measure*

Description

This measure specializes Measure for classification problems. Predefined measures can be found in the mlr3misc::Dictionary mlr_measures.

Format

R6::R6Class object inheriting from Measure.

Construction

```
m = MeasureClassif$new(id, range, minimize = NA, aggregator = NULL, properties = character(), predict_ty
  predict_sets = "test", task_properties = character(), packages = character(), man = NA_character_)
```

For a description of the arguments, see Measure. The `task_type` is set to `"classif"`. Possible values for `predict_type` are `"response"` and `"prob"`.

Fields

See Measure.

Methods

See Measure.

See Also

Example classification measures: `classif.ce`

Other Measure: `MeasureRegr`, `Measure`, `mlr_measures`

MeasureClassifACC *Accuracy Classification Measure*

Description

Calls `Metrics::accuracy()`.

Format

`R6::R6Class()` inheriting from MeasureClassif.

Construction

```
MeasureClassifACC$new()
mlr_measures$get("classif.acc")
msr("classif.acc")
```

See Also

Dictionary of Measures: mlr_measures

`as.data.table(mlr_measures)` for a complete table of all (also dynamically created) Measure implementations.

MeasureClassifAUC *Area Under the Curve Classification Measure*

Description

Calls `Metrics::auc()`.

Format

`R6::R6Class()` inheriting from MeasureClassif.

Construction

```
MeasureClassifAUC$new()
mlr_measures$get("classif.auc")
msr("classif.auc")
```

See Also

Dictionary of Measures: mlr_measures

`as.data.table(mlr_measures)` for a complete table of all (also dynamically created) Measure implementations.

MeasureClassifCE *Classification Error Measure*

Description

Calls `Metrics::ce()`.

Format

`R6::R6Class()` inheriting from MeasureClassif.

Construction

```
MeasureClassifCE$new()
mlr_measures$get("classif.ce")
msr("classif.ce")
```

See Also

Dictionary of Measures: mlr_measures

`as.data.table(mlr_measures)` for a complete table of all (also dynamically created) Measure implementations.

MeasureClassifConfusion *Binary Classification Measures Derived from a Confusion Matrix*

Description

All implemented Measures call `confusion_measures()` with the respective `type` internally. For the F1 measure, use MeasureClassifFScore.

Format

`R6::R6Class()` inheriting from MeasureClassif.

Construction

```
MeasureClassifConfusion$new(id = type, type)

mlr_measures("classif.tp")
mlr_measures("classif.fn")
mlr_measures("classif.fp")
mlr_measures("classif.tn")
mlr_measures("classif.tpr")
```

```
mlr_measures("classif.fnr")
mlr_measures("classif.fpr")
mlr_measures("classif.tnr")
mlr_measures("classif.ppv")
mlr_measures("classif.fdr")
mlr_measures("classif.for")
mlr_measures("classif.npv")
mlr_measures("classif.dor")
mlr_measures("classif.precision")
mlr_measures("classif.recall")
mlr_measures("classif.sensitivity")
mlr_measures("classif.specificity")

msr("classif.tp")
msr("classif.fn")
msr("classif.fp")
msr("classif.tn")
msr("classif.tpr")
msr("classif.fnr")
msr("classif.fpr")
msr("classif.tnr")
msr("classif.ppv")
msr("classif.fdr")
msr("classif.for")
msr("classif.npv")
msr("classif.dor")
msr("classif.precision")
msr("classif.recall")
msr("classif.sensitivity")
msr("classif.specificity")
```

- type :: `character(1)`
  See `confusion_measures()`.

See Also

Dictionary of Measures: mlr_measures

`as.data.table(mlr_measures)` for a complete table of all (also dynamically created) Measure implementations.

Examples

```
task = tsk("german_credit")
learner = lrn("classif.rpart")
p = learner$train(task)$predict(task)
measures = list(msr("classif.sensitivity"), msr("classif.specificity"))
round(p$score(measures), 2)
```

MeasureClassifCosts *Cost-sensitive Classification Measure*

Description

Uses a cost matrix to create a classification measure. True labels must be arranged in columns, predicted labels must be arranged in rows. The cost matrix is stored as slot `$costs`. Costs are aggregated with the mean.

Format

`R6::R6Class()` inheriting from MeasureClassif.

Construction

```
MeasureClassifCosts$new(costs = NULL, normalize = TRUE)
mlr_measures$get("classif.costs")
msr("classif.costs")
```

- `costs` :: `matrix()`
  Numeric matrix of costs (truth in columns, predicted response in rows).
- `normalize` :: `logical(1)`
  If `TRUE`, calculate the mean costs instead of the total costs.

See Also

Dictionary of Measures: mlr_measures

`as.data.table(mlr_measures)` for a complete table of all (also dynamically created) Measure implementations.

Examples

```
# get a cost sensitive task
task = tsk("german_credit")

# cost matrix as given on the UCI page of the german credit data set
# https://archive.ics.uci.edu/ml/datasets/statlog+(german+credit+data)
costs = matrix(c(0, 5, 1, 0), nrow = 2)
dimnames(costs) = list(truth = task$class_names, predicted = task$class_names)
print(costs)

# mlr3 needs truth in columns, predictions in rows
costs = t(costs)

# create measure which calculates the absolute costs
m = msr("classif.costs", id = "german_credit_costs", costs = costs, normalize = FALSE)

# fit models and calculate costs
```

```
learner = lrn("classif.rpart")
rr = resample(task, learner, rsmp("cv", folds = 3))
rr$aggregate(m)
```

MeasureClassifFScore *F-score Classification Measure*

Description

Calls `Metrics::fbeta_score()`. Argument `beta` defaults to 1.

Format

`R6::R6Class()` inheriting from MeasureClassif.

Construction

```
MeasureClassifFScore$new(beta = 1)
mlr_measures$get("classif.f_score")
msr("classif.f_score")
```

- `beta` :: `numeric(1)`
  Non-negative parameter balancing precision and recall. Passed down to `Metrics::fbeta_score()`

References

Sasaki Y, others (2007). "The truth of the F-measure." *Teach Tutor mater*, **1**(5), 1–5. https://www.cs.odu.edu/~mukka/cs795sum10dm/Lecturenotes/Day3/F-measure-YS-26Oct07.pdf.

See Also

Dictionary of Measures: mlr_measures

`as.data.table(mlr_measures)` for a complete table of all (also dynamically created) Measure implementations.

MeasureDebug *Debug Measure*

Description

This measure returns the number of observations in the Prediction object. Its main purpose is debugging.

Format

`R6::R6Class()` inheriting from Measure.

Construction

```
MeasureDebug$new(na_ratio = 0)
mlr_measures$get("debug")
msr("debug")
```

- `na_ratio` :: `numeric(1)`
  Ratio of scores which should be `NA`. Default is 0.

Fields

- `na_ratio` :: `numeric(1)`.

See Also

Dictionary of Measures: mlr_measures

`as.data.table(mlr_measures)` for a complete table of all (also dynamically created) Measure implementations.

Examples

```
task = tsk("wine")
learner = lrn("classif.featureless", predict_sets = "test")
measure = msr("debug")
rr = resample(task, learner, rsmp("cv", folds = 3))
rr$score(measure)
```

`MeasureElapsedTime` *Elapsed Time Measure*

Description

Measures the elapsed time during train ("time_train"), predict ("time_predict"), or both ("time_both").

Format

`R6::R6Class()` inheriting from Measure.

Construction

```
MeasureElapsedTime$new(id, stages)

mlr_measures$get("time_train")
mlr_measures$get("time_predict")
mlr_measures$get("time_both")

msr$get("time_train")
msr$get("time_predict")
msr$get("time_both")
```

- id :: character(1)
  Id for the created measure.
- stages :: character()
  Subset of ("train","predict"). The runtime of all stages will be summed.

See Also

Dictionary of Measures: mlr_measures

as.data.table(mlr_measures) for a complete table of all (also dynamically created) Measure implementations.

MeasureOOBError *Out-of-bag Error Measure*

Description

Returns the out-of-bag error of the Learner for learners that support it (learners with property "oob_error"). Returns NA for unsupported learners.

Format

R6::R6Class() inheriting from Measure.

Construction

```
MeasureOOBError$new()
mlr_measures$get("oob_error")
msr("oob_error")
```

See Also

Dictionary of Measures: mlr_measures

as.data.table(mlr_measures) for a complete table of all (also dynamically created) Measure implementations.

MeasureRegr *Regression Measure*

Description

This measure specializes Measure for regression problems. Predefined measures can be found in the mlr3misc::Dictionary mlr_measures.

Format

R6::R6Class object inheriting from Measure.

Construction

```
m = MeasureRegr$new(id, range, minimize = NA, aggregator = NULL, properties = character(), predict_type
    predict_sets = "test", task_properties = character(), packages = character())
```

For a description of the arguments, see Measure. The `task_type` is set to `"regr"`. Possible values for `predict_type` are `"response"` and `"se"`.

Fields

See Measure.

Methods

See Measure.

See Also

Example regression measures: `regr.mse`

Other Measure: `MeasureClassif`, `Measure`, `mlr_measures`

MeasureRegrMAE *Absolute Errors Regression Measure*

Description

Calls `Metrics::mae()`.

Format

R6::R6Class inheriting from MeasureRegr.

Construction

```
MeasureRegrMAE$new()
mlr_measures$get("regr.mae")
msr("regr.mae")
```

See Also

Dictionary of Measures: mlr_measures

`as.data.table(mlr_measures)` for a complete table of all (also dynamically created) Measure implementations.

`MeasureRegrMSE` *Mean Squared Error Regression Measure*

Description

Calls `Metrics::mse()`.

Format

R6::R6Class inheriting from MeasureRegr.

Construction

```
MeasureRegrMSE$new()
mlr_measures$get("regr.mse")
msr("regr.mse")
```

See Also

Dictionary of Measures: mlr_measures

`as.data.table(mlr_measures)` for a complete table of all (also dynamically created) Measure implementations.

MeasureRegrRMSE *Root Mean Squared Error Regression Measure*

Description

Calls Metrics::rmse().

Format

R6::R6Class inheriting from MeasureRegr.

Construction

```
MeasureRegrRMSE$new()
mlr_measures$get("regr.rmse")
msr("regr.rmse")
```

See Also

Dictionary of Measures: mlr_measures

as.data.table(mlr_measures) for a complete table of all (also dynamically created) Measure implementations.

MeasureSelectedFeatures *Selected Features Measure*

Description

Measures the number of selected features.

Format

R6::R6Class() inheriting from Measure.

Construction

```
MeasureSelectedFeatures$new(normalize = FALSE)
mlr_measures$get("selected_features")
msr("selected_features")
```

- normalize :: logical(1)
  If normalize is set to TRUE, divides the number of features by the total number of features.

See Also

Dictionary of Measures: mlr_measures

`as.data.table(mlr_measures)` for a complete table of all (also dynamically created) Measure implementations.

`mlr_learners` *Dictionary of Learners*

Description

A simple mlr3misc::Dictionary storing objects of class Learner. Each learner has an associated help page, see `mlr_learners_[id]`.

This dictionary can get populated with additional learners by add-on packages. For more classification and regression learners, load the **mlr3learners** package.

For a more convenient way to retrieve and construct learners, see `lrn()`.

Format

R6::R6Class object inheriting from mlr3misc::Dictionary.

Methods

See mlr3misc::Dictionary.

S3 methods

- `as.data.table(dict)`
  mlr3misc::Dictionary -> `data.table::data.table()`
  Returns a `data.table::data.table()` with fields "key", "feature_types", "packages", "properties" and "predict_types" as columns.

See Also

Example learners: `classif.rpart`, `regr.rpart`, `classif.featureless`, `regr.featureless`, `classif.debug`

Sugar function: `lrn()`

Extension Packages: **mlr3learners**

Other Dictionary: `mlr_measures`, `mlr_resamplings`, `mlr_task_generators`, `mlr_tasks`

Other Learner: `LearnerClassif`, `LearnerRegr`, `Learner`

Examples

```
as.data.table(mlr_learners)
mlr_learners$get("classif.featureless")
lrn("classif.rpart")
```

`mlr_measures` *Dictionary of Performance Measures*

Description

A simple mlr3misc::Dictionary storing objects of class Measure. Each measure has an associated help page, see `mlr_measures_[id]`.

This dictionary can get populated with additional measures by add-on packages.

For a more convenient way to retrieve and construct measures, see `msr()`.

Format

R6::R6Class object inheriting from mlr3misc::Dictionary.

Methods

See mlr3misc::Dictionary.

S3 methods

- `as.data.table(dict)`
  mlr3misc::Dictionary -> `data.table::data.table()`
  Returns a `data.table::data.table()` with fields "key", "task_type", "predict_type", and "packages" as columns.

See Also

Example measures: `classif.auc`, `time_train` Sugar function: `msr()`

Other Dictionary: `mlr_learners`, `mlr_resamplings`, `mlr_task_generators`, `mlr_tasks`

Other Measure: `MeasureClassif`, `MeasureRegr`, `Measure`

Examples

```
as.data.table(mlr_measures)
mlr_measures$get("classif.ce")
msr("regr.mse")
```

`mlr_resamplings` *Dictionary of Resampling Strategies*

Description

A simple mlr3misc::Dictionary storing objects of class Resampling. Each resampling has an associated help page, see `mlr_resamplings_[id]`.

This dictionary can get populated with additional resampling strategies by add-on packages.

For a more convenient way to retrieve and construct resampling strategies, see `rsmp()`.

Format

R6::R6Class object inheriting from mlr3misc::Dictionary.

Methods

See mlr3misc::Dictionary.

S3 methods

- `as.data.table(dict)`
  mlr3misc::Dictionary -> `data.table::data.table()`
  Returns a `data.table::data.table()` with columns `"key"`, `"params"`, and `"iters"`.

See Also

Example resamplings: `cv`, `bootstrap`

Sugar function: `rsmp()`

Other Dictionary: `mlr_learners`, `mlr_measures`, `mlr_task_generators`, `mlr_tasks`

Other Resampling: `Resampling`

Examples

```
as.data.table(mlr_resamplings)
mlr_resamplings$get("cv")
rsmp("subsampling")
```

`mlr_sugar` *Syntactic Sugar for Object Construction*

Description

Functions to retrieve objects, set hyperparameters and assign to fields in one go. Relies on `mlr3misc::dictionary_sugar_g` to extract objects from the respective mlr3misc::Dictionary:

- `tsk()` for a Task from mlr_tasks.
- `tsks()` for a list of Tasks from mlr_tasks.
- `tgen()` for a TaskGenerator from mlr_task_generators.
- `tgens()` for a list of TaskGenerators from mlr_task_generators.
- `lrn()` for a Learner from mlr_learners.
- `lrns()` for a list of Learners from mlr_learners.
- `rsmp()` for a Resampling from mlr_resamplings.
- `rsmps()` for a list of Resamplings from mlr_resamplings.
- `msr()` for a Measure from mlr_measures.
- `msrs()` for a list of Measures from mlr_measures.

Usage

```
tsk(.key, ...)

tsks(.keys, ...)

tgen(.key, ...)

tgens(.keys, ...)

lrn(.key, ...)

lrns(.keys, ...)

rsmp(.key, ...)

rsmps(.keys, ...)

msr(.key, ...)

msrs(.keys, ...)
```

Arguments

`.key`	:: `character(1)` Key passed to the respective mlr3misc::Dictionary to retrieve the object.
`...`	:: named `list()` Named arguments passed to the constructor, to be set as parameters in the paradox::ParamSet, or to be set as public field. See `mlr3misc::dictionary_sugar_get()` for more details.
`.keys`	:: `character()` Keys passed to the respective mlr3misc::Dictionary to retrieve multiple objects.

Value

R6::R6Class object of the respective type, or a list of R6::R6Class objects for the plural versions.

Examples

```
# iris task with new id
tsk("iris", id = "iris2")

# classification tree with different hyperparameters
# and predict type set to predict probabilities
lrn("classif.rpart", cp = 0.1, predict_type = "prob")

# multiple learners with predict type 'prob'
lapply(c("classif.featureless", "classif.rpart"), lrn, predict_type = "prob")
```

`mlr_tasks` *Dictionary of Tasks*

Description

A simple mlr3misc::Dictionary storing objects of class Task. Each task has an associated help page, see `mlr_tasks_[id]`.

This dictionary can get populated with additional tasks by add-on packages.

For a more convenient way to retrieve and construct tasks, see `tsk()`.

Format

R6::R6Class object inheriting from mlr3misc::Dictionary.

Methods

See mlr3misc::Dictionary.

S3 methods

- `as.data.table(dict)`
  mlr3misc::Dictionary -> `data.table::data.table()`
  Returns a `data.table::data.table()` with columns `"key"`, `"task_type"`, `"measures"`, `"nrow"`, `"ncol"` and the number of features of type `"lgl"`, `"int"`, `"dbl"`, `"chr"`, `"fct"` and `"ord"` as columns.

See Also

Example tasks: `iris` (multi-class classification), `spam` (binary classification), `boston_housing` (regression)

Sugar function: `tsk()`

Other Dictionary: `mlr_learners`, `mlr_measures`, `mlr_resamplings`, `mlr_task_generators`

Other Task: `TaskClassif`, `TaskRegr`, `TaskSupervised`, `Task`

Examples

```
as.data.table(mlr_tasks)
task = mlr_tasks$get("iris") # same as tsk("iris")
head(task$data())

# Add a new task, based on a subset of iris:
data = iris
data$Species = factor(ifelse(data$Species == "setosa", "1", "0"))
task = TaskClassif$new("iris.binary", data, target = "Species", positive = "1")

# add to dictionary
mlr_tasks$add("iris.binary", task)

# list available tasks
mlr_tasks$keys()

# retrieve from dictionary
mlr_tasks$get("iris.binary")

# remove task again
mlr_tasks$remove("iris.binary")
```

---

mlr_tasks_boston_housing

Boston Housing Regression Task

Description

A regression task for the mlbench::BostonHousing2 data set.

Format

R6::R6Class inheriting from TaskRegr.

Construction

```
mlr_tasks$get("boston_housing")
tsk("boston_housing")
```

See Also

Dictionary of Tasks: mlr_tasks

`as.data.table(mlr_tasks)` for a complete table of all (also dynamically created) Tasks.

---

mlr_tasks_german_credit

German Credit Classification Task

Description

A classification task for the German credit data set. The aim is to predict creditworthiness, labeled as "good" and "bad". Positive class is set to label "good".

See example for the creation of a MeasureClassifCosts as described misclassification costs.

Format

R6::R6Class inheriting from TaskClassif.

Construction

```
mlr_tasks$get("german_credit")
tsk("german_credit")
```

Source

Data set originally published on UCI. This is the preprocessed version taken from package **evtree**.

Donor: Professor Dr. Hans Hofmann
Institut für Statistik und Ökonometrie
Universität Hamburg
FB Wirtschaftswissenschaften
Von-Melle-Park 5
2000 Hamburg 13

See Also

Dictionary of Tasks: mlr_tasks

`as.data.table(mlr_tasks)` for a complete table of all (also dynamically created) Tasks.

Examples

```
task = tsk("german_credit")
costs = matrix(c(0, 1, 5, 0), nrow = 2)
dimnames(costs) = list(predicted = task$class_names, truth = task$class_names)
measure = msr("classif.costs", id = "german_credit_costs", costs = costs)
print(measure)
```

---

mlr_tasks_iris *Iris Classification Task*

Description

A classification task for the popular datasets::iris data set.

Format

R6::R6Class inheriting from TaskClassif.

Construction

```
mlr_tasks$get("iris")
tsk("iris")
```

See Also

Dictionary of Tasks: mlr_tasks

`as.data.table(mlr_tasks)` for a complete table of all (also dynamically created) Tasks.

---

mlr_tasks_mtcars *"Motor Trend" Car Road Tests Task*

Description

A regression task for the datasets::mtcars data set. Target variable is `mpg` (Miles/(US) gallon).

Format

R6::R6Class inheriting from TaskRegr.

Construction

```
mlr_tasks$get("mtcars")
tsk("mtcars")
```

See Also

Dictionary of Tasks: mlr_tasks

`as.data.table(mlr_tasks)` for a complete table of all (also dynamically created) Tasks.

mlr_tasks_pima *Pima Indian Diabetes Classification Task*

Description

A classification task for the mlbench::PimaIndiansDiabetes2 data set. Positive class is set to "pos".

Format

R6::R6Class inheriting from TaskClassif.

Construction

```
mlr_tasks$get("pima")
tsk("pima")
```

See Also

Dictionary of Tasks: mlr_tasks

as.data.table(mlr_tasks) for a complete table of all (also dynamically created) Tasks.

mlr_tasks_sonar *Sonar Classification Task*

Description

A classification task for the mlbench::Sonar data set. Positive class is set to "M" (Mine).

Format

R6::R6Class inheriting from TaskClassif.

Construction

```
mlr_tasks$get("sonar")
tsk("sonar")
```

See Also

Dictionary of Tasks: mlr_tasks

as.data.table(mlr_tasks) for a complete table of all (also dynamically created) Tasks.

mlr_tasks_spam *Spam Classification Task*

Description

Spam data set from the UCI machine learning repository (http://archive.ics.uci.edu/ml/datasets/spambase). Data set collected at Hewlett-Packard Labs to classify emails as spam or non-spam. 57 variables indicate the frequency of certain words and characters in the e-mail. The positive class is set to "spam".

Format

R6::R6Class inheriting from TaskClassif.

Construction

```
mlr_tasks$get("spam")
tsk("spam")
```

Source

Creators: Mark Hopkins, Erik Reeber, George Forman, Jaap Suermondt. Hewlett-Packard Labs, 1501 Page Mill Rd., Palo Alto, CA 94304

Donor: George Forman (gforman at nospam hpl.hp.com) 650-857-7835

Preprocessing: Columns have been renamed. Preprocessed data taken from the **kernlab** package.

References

Dua D, Graff C (2017). "UCI Machine Learning Repository." http://archive.ics.uci.edu/ml.

See Also

Dictionary of Tasks: mlr_tasks

`as.data.table(mlr_tasks)` for a complete table of all (also dynamically created) Tasks.

mlr_tasks_wine *Wine Classification Task*

Description

Wine data set from the UCI machine learning repository (https://archive.ics.uci.edu/ml/datasets/wine). Results of a chemical analysis of three types of wines grown in the same region in Italy but derived from three different cultivars.

Format

R6::R6Class inheriting from TaskClassif.

Construction

```
mlr_tasks$get("wine")
tsk("wine")
```

Source

Original owners: Forina, M. et al, PARVUS - An Extendible Package for Data Exploration, Classification and Correlation. Institute of Pharmaceutical and Food Analysis and Technologies, Via Brigata Salerno, 16147 Genoa, Italy.

Donor: Stefan Aeberhard, email: stefan@coral.cs.jcu.edu.au

References

Dua D, Graff C (2017). "UCI Machine Learning Repository." http://archive.ics.uci.edu/ml.

See Also

Dictionary of Tasks: mlr_tasks

`as.data.table(mlr_tasks)` for a complete table of all (also dynamically created) Tasks.

`mlr_tasks_zoo` *Zoo Classification Task*

Description

A classification task for the mlbench::Zoo data set.

Format

R6::R6Class inheriting from TaskClassif.

Construction

```
mlr_tasks$get("zoo")
tsk("zoo")
```

See Also

Dictionary of Tasks: mlr_tasks

`as.data.table(mlr_tasks)` for a complete table of all (also dynamically created) Tasks.

`mlr_task_generators` *Dictionary of Task Generators*

Description

A simple mlr3misc::Dictionary storing objects of class TaskGenerator. Each task generator has an associated help page, see `mlr_task_generators_[id]`.

This dictionary can get populated with additional task generators by add-on packages.

For a more convenient way to retrieve and construct task generators, see `tgen()`.

Format

R6::R6Class object inheriting from mlr3misc::Dictionary.

Methods

See mlr3misc::Dictionary.

S3 methods

- `as.data.table(dict)`
  mlr3misc::Dictionary -> `data.table::data.table()`
  Returns a `data.table::data.table()` with fields `"key"` and `"packages"` as columns.

See Also

Example generators: `xor`

Sugar function: `tgen()`

Other Dictionary: `mlr_learners`, `mlr_measures`, `mlr_resamplings`, `mlr_tasks`

Other TaskGenerator: `TaskGenerator`

Examples

```
mlr_task_generators$get("smiley")
tgen("2dnormals")
```

`predict.Learner` *Predict Method for Learners*

Description

Extends the generic `stats::predict()` with a method for Learner. Note that this function is intended as glue code to be used in third party packages. We recommend to work with the Learner directly, i.e. calling `learner$predict()` or `learner$predict_newdata()` directly.

Performs the following steps:

- Sets additional hyperparameters passed to this function.
- Creates a Prediction object by calling `learner$predict_newdata()`.
- Returns (subset of) Prediction.

Usage

```
## S3 method for class 'Learner'
predict(object, newdata, predict_type = NULL, ...)
```

Arguments

`object`	:: Learner Any Learner.
`newdata`	:: `data.frame()` New data to predict on.
`predict_type`	:: `character(1)` The predict type to return. Set to `<Prediction>` to retrieve the complete Prediction object. If set to `NULL` (default), the first predict type for the respective class of the Learner as stored in mlr_reflections is used.
`...`	:: any Hyperparameters to pass down to the Learner.

Examples

```
task = tsk("spam")

learner = lrn("classif.rpart", predict_type = "prob")
learner$train(task)
predict(learner, task$data(1:3), predict_type = "response")
predict(learner, task$data(1:3), predict_type = "prob")
predict(learner, task$data(1:3), predict_type = "<Prediction>")
```

`Prediction` *Abstract Prediction Object*

Description

This is the abstract base class for task objects like PredictionClassif or PredictionRegr.

Prediction objects store the following information:

1. The row ids of the test set
2. The corresponding true (observed) response.
3. The corresponding predicted response.
4. Additional predictions based on the class and `predict_type`. E.g., the class probabilities for classification or the estimated standard error for regression.

Format

R6::R6Class object.

Construction

This object is constructed via a derived classes, e.g. PredictionClassif or PredictionRegr.

Fields

- `row_ids` :: (`integer()` | `character()`)
  Vector of row ids for which predictions are stored.
- `truth` :: `any`
  True (observed) outcome.
- `task_type` :: `character(1)`
  Stores the type of the Task.
- `predict_types` :: `character()`
  Vector of predict types this object stores.
- `missing` :: `logical()`
  Returns `row_ids` for which the predictions are missing or incomplete.

Methods

- `score(measures = NULL, task = NULL, learner = NULL)`
  (list of Measure, Task, Learner) -> Prediction
  Calculates the performance for all provided measures Task and Learner may be `NULL` for most measures, but some measures need to extract information from these objects.
- `help()`
  () -> `NULL`
  Opens the help page for this object.

S3 Methods

- `as.data.table(rr)`
  Prediction -> `data.table::data.table()`
  Converts the data to a `data.table::data.table()`.
- `c(...,keep_duplicates = TRUE)`
  (Prediction, Prediction, ...) -> Prediction
  Combines multiple `Predictions` to a single `Prediction`. If `keep_duplicates` is `FALSE` and there are duplicated row ids, the data of the former passed objects get overwritten by the data of the later passed objects.

See Also

Other Prediction: `PredictionClassif`, `PredictionRegr`

`PredictionClassif` *Prediction Object for Classification*

Description

This object wraps the predictions returned by a learner of class LearnerClassif, i.e. the predicted response and class probabilities.

If the response is not provided during construction, but class probabilities are, the response is calculated from the probabilities: the class label with the highest probability is chosen. In case of ties, a label is selected randomly.

Format

R6::R6Class object inheriting from Prediction.

Construction

```
p = PredictionClassif$new(task = NULL, row_ids = task$row_ids, truth = task$truth(), response = NULL, pr
```

- `task` :: TaskClassif
  Task, used to extract defaults for `row_ids` and `truth`.
- `row_ids` :: `(integer() | character())`
  Row ids of the observations in the test set.
- `truth` :: `factor()`
  True (observed) labels. See the note on manual construction.
- `response` :: `(character() | factor())`
  Vector of predicted class labels. One element for each observation in the test set. Character vectors are automatically converted to factors. See the note on manual construction.
- `prob` :: `matrix()`
  Numeric matrix of posterior class probabilities with one column for each class and one row for each observation in the test set. Columns must be named with class labels, row names are automatically removed. If `prob` is provided, but `response` is not, the class labels are calculated from the probabilities using `mlr3misc::which_max()` with `ties_method` set to `"random"`.

Fields

All fields from Prediction, and additionally:

- `response` :: `factor()`
  Access to the stored predicted class labels.
- `prob` :: `matrix()`
  Access to the stored probabilities.
- `confusion` :: `matrix()`
  Confusion matrix resulting from the comparison of truth and response. Truth is in columns, predicted response is in rows.

The field `task_type` is set to `"classif"`.

Methods

- `set_threshold(th)`
  `numeric()` -> `self`
  Sets the prediction response based on the provided threshold. See the section on thresholding for more information.

Thresholding

If probabilities are stored, it is possible to change the threshold which determines the predicted class label. Usually, the label of the class with the highest predicted probability is selected. For binary classification problems, such an threshold defaults to 0.5. For cost-sensitive or imbalanced classification problems, manually adjusting the threshold can increase the predictive performance.

- For binary problems only a single threshold value can be set. If the probability exceeds the threshold, the positive class is predicted. If the probability equals the threshold, the label is selected randomly.
- For binary and multi-class problems, a named numeric vector of thresholds can be set. The length and names must correspond to the number of classes and class names, respectively. To determine the class label, the probabilities are divided by the threshold. This results in a ratio > 1 if the probability exceeds the threshold, and a ratio < 1 otherwise. Note that it is possible that either none or multiple ratios are greater than 1 at the same time. Anyway, the class label with maximum ratio is selected. In case of ties in the ratio, one of the tied class labels is selected randomly.

Note

If this object is constructed manually, make sure that the factor levels for `truth` have the same levels as the task, in the same order. In case of binary classification tasks, the positive class label must be the first level.

See Also

Other Prediction: `PredictionRegr`, `Prediction`

Examples

```
task = tsk("iris")
learner = lrn("classif.rpart", predict_type = "prob")
learner$train(task)
p = learner$predict(task)
p$predict_types
head(as.data.table(p))

# confusion matrix
p$confusion

# change threshold
th = c(0.05, 0.9, 0.05)
names(th) = task$class_names

# new predictions
p$set_threshold(th)$response
p$score(measures = msr("classif.ce"))
```

PredictionRegr *Prediction Object for Regression*

Description

This object wraps the predictions returned by a learner of class LearnerRegr, i.e. the predicted response and standard error.

Format

R6::R6Class object inheriting from Prediction.

Construction

```
p = PredictionRegr$new(task = NULL, row_ids = task$row_ids, truth = task$truth(), response = NULL, se = N
```

- `task` :: TaskRegr
  Task, used to extract defaults for `row_ids` and `truth`.
- `row_ids` :: (`integer()` | `character()`)
  Row ids of the observations in the test set.
- `truth` :: `numeric()`
  True (observed) response.
- `response` :: `numeric()`
  Vector of numeric response values. One element for each observation in the test set.
- `se` :: `numeric()`
  Numeric vector of predicted standard errors. One element for each observation in the test set.

Fields

All fields from Prediction, and additionally:

- `response` :: `numeric()`
  Access to the stored predicted response.
- `se` :: `numeric()`
  Access to the stored standard error.

The field `task_type` is set to `"regr"`.

See Also

Other Prediction: `PredictionClassif`, `Prediction`

Examples

```
task = tsk("boston_housing")
learner = lrn("regr.featureless", predict_type = "se")
p = learner$train(task)$predict(task)
p$predict_types
head(as.data.table(p))
```

`resample` *Resample a Learner on a Task*

Description

Runs a resampling (possibly in parallel).

Usage

```
resample(task, learner, resampling, store_models = FALSE)
```

Arguments

`task`	:: Task.
`learner`	:: Learner.
`resampling`	:: Resampling.
`store_models`	:: `logical(1)` Keep the fitted model after the test set has been predicted? Set to `TRUE` if you want to further analyse the models or want to extract information like variable importance.

Value

ResampleResult.

Parallelization

This function can be parallelized with the **future** package. One job is one resampling iteration, and all jobs are send to an apply function from **future.apply** in a single batch. To select a parallel backend, use `future::plan()`.

Logging

The **mlr3** uses the **lgr** package for logging. **lgr** supports multiple log levels which can be queried with `getOption("lgr.log_levels")`.

To suppress output and reduce verbosity, you can lower the log from the default level `"info"` to `"warn"`:

```
lgr::get_logger("mlr3")$set_threshold("warn")
```

To get additional log output for debugging, increase the log level to `"debug"` or `"trace"`:

```
lgr::get_logger("mlr3")$set_threshold("debug")
```

To log to a file or a data base, see the documentation of lgr::lgr-package.

Note

The fitted models are discarded after the predictions have been scored in order to reduce memory consumption. If you need access to the models for later analysis, set `store_models` to `TRUE`.

Examples

```
task = tsk("iris")
learner = lrn("classif.rpart")
resampling = rsmp("cv")

# explicitly instantiate the resampling for this task for reproduciblity
set.seed(123)
resampling$instantiate(task)

rr = resample(task, learner, resampling)
print(rr)

# retrieve performance
rr$score(msr("classif.ce"))
rr$aggregate(msr("classif.ce"))

# merged prediction objects of all resampling iterations
pred = rr$prediction()
pred$confusion

# Repeat resampling with featureless learner
rr_featureless = resample(task, lrn("classif.featureless"), resampling)

# Convert results to BenchmarkResult, then combine them
```

```
bmr1 = as_benchmark_result(rr)
bmr2 = as_benchmark_result(rr_featureless)
print(bmr1$combine(bmr2))
```

`ResampleResult` *Container for Results of* `resample()`

Description

This is the result container object returned by `resample()`.

Note that all stored objects are accessed by reference. Do not modify any object without cloning it first.

Format

R6::R6Class object.

Construction

```
rr = ResampleResult$new(data, uhash = NULL)
```

- `data` :: `data.table::data.table()`
  Table with data for one resampling iteration per row: Task, Learner, Resampling, iteration (`integer(1)`), and Prediction.
- `uhash` :: `character(1)`
  Unique hash for this `ResampleResult`. If `NULL`, a new unique hash is generated. This unique hash is primarily needed to group information in BenchmarkResults.

Fields

- `data` :: `data.table::data.table()`
  Internal data storage. We discourage users to directly work with this field.
- `task` :: Task

  The task `resample()` operated on.
- `learners` :: list of Learner
  List of trained learners, sorted by resampling iteration.
- `resampling` :: Resampling
  Instantiated Resampling object which stores the splits into training and test.
- `warnings` :: `data.table::data.table()`
  Returns a table with all warning messages. Column names are `"iteration"` and `"msg"`. Note that there can be multiple rows per resampling iteration if multiple warnings have been recorded.
- `errors` :: `data.table::data.table()`
  Returns a table with all error messages. Column names are `"iteration"` and `"msg"`. Note that there can be multiple rows per resampling iteration if multiple errors have been recorded.
- `uhash` :: `character(1)`
  Unique hash for this object.

Methods

- `predictions(predict_sets = "test")`
  `character()` -> list of Prediction
  List of prediction objects, sorted by resampling iteration. If multiple sets are given, these are combined to a single one for each iteration.
- `prediction(predict_sets = "test")`
  `character()` -> Prediction
  Combined Prediction of all individual resampling iterations, and all provided predict sets. Note that performance measures do not operate on this object, but instead on each prediction object separately and then combine the performance scores with the aggregate function of the respective Measure.
- `score(measures = NULL, ids = TRUE)`
  (list of Measure, `logical(1)`) -> `data.table::data.table()`
  Returns a table with one row for each resampling iteration, including all involved objects: Task, Learner, Resampling, iteration number (`integer(1)`), and Prediction. A column with the individual (per resampling iteration) performance is added for each Measure, named with the id of the respective measure. If `ids` is `TRUE`, extra columns with the ids of objects (`"task_id"`, `"learner_id"`, `"resampling_id"`) are binded to the table to allow a more convenient subsetting. If `measures` is `NULL`, `measures` defaults to the return value of `default_measures()`.
- `aggregate(measures = NULL)`
  list of Measure -> named `numeric()`
  Calculates and aggregates performance values for all provided measures, according to the respective aggregation function in Measure. If `measures` is `NULL`, `measures` defaults to the return value of `default_measures()`.
- `help()`
  () -> `NULL`
  Opens the help page for this object.

S3 Methods

- `as.data.table(rr)`
  ResampleResult -> `data.table::data.table()`
  Returns a copy of the internal data.

Examples

```
task = tsk("iris")
learner = lrn("classif.rpart")
resampling = rsmp("cv", folds = 3)
rr = resample(task, learner, resampling)
print(rr)

rr$aggregate(msr("classif.acc"))
rr$prediction()
rr$prediction()$confusion
rr$warnings
rr$errors
```

`Resampling` *Resampling Class*

Description

This is the abstract base class for resampling objects like ResamplingCV and ResamplingBootstrap.

The objects of this class define how a task is partitioned for resampling (e.g., in `resample()` or `benchmark()`), using a set of hyperparameters such as the number of folds in cross-validation.

Resampling objects can be instantiated on a Task, which applies the strategy on the task and manifests in a fixed partition of `row_ids` of the Task.

Predefined resamplings are stored in the mlr3misc::Dictionary mlr_resamplings, e.g. `cv` or `bootstrap`.

Format

R6::R6Class object.

Construction

Note: This object is typically constructed via a derived classes, e.g. ResamplingCV or ResamplingHoldout.

```
r = Resampling$new(id, param_set, duplicated_ids = FALSE, man = NA_character_)
```

- `id` :: `character(1)`
  Identifier for the resampling strategy.
- `param_set` :: paradox::ParamSet
  Set of hyperparameters.
- `duplicated_ids` :: `logical(1)`
  Set to `TRUE` if this resampling strategy may have duplicated row ids in a single training set or test set.
- `man` :: `character(1)`
  String in the format `[pkg]::[topic]` pointing to a manual page for this object.

Fields

All variables passed to the constructor, and additionally:

- `iters` :: `integer(1)`
  Return the number of resampling iterations, depending on the values stored in the `param_set`.
- `instance` :: `any`
  During `instantiate()`, the instance is stored in this slot. The instance can be in any arbitrary format.
- `is_instantiated` :: `logical(1)`
  Is `TRUE`, if the resampling has been instantiated.

- `task_hash` :: `character(1)`
  The hash of the task which was passed to `r$instantiate()`.
- `hash` :: `character(1)`
  Hash (unique identifier) for this object.

  E.g., this is `TRUE` for Bootstrap, and `FALSE` for cross validation. Only used internally.

Methods

- `instantiate(task)`
  Task -> `self`
  Materializes fixed training and test splits for a given task and stores them in `r$instance`.
- `train_set(i)`
  `integer(1)` -> `(integer() | character())`
  Returns the row ids of the i-th training set.
- `test_set(i)`
  `integer(1)` -> `(integer() | character())`
  Returns the row ids of the i-th test set.
- `help()`
  () -> `NULL`
  Opens the corresponding help page referenced by `$man`.

Stratification

All derived classes support stratified sampling. The stratification variables are assumed to be discrete and must be stored in the Task with column role `"stratum"`. In case of multiple stratification variables, each combination of the values of the stratification variables forms a strata.

First, the observations are divided into subpopulations based one or multiple stratification variables (assumed to be discrete), c.f. `task$strata`.

Second, the sampling is performed in each of the `k` subpopulations separately. Each subgroup is divided into `iter` training sets and `iter` test sets by the derived `Resampling`. These sets are merged based on their iteration number: all training sets from all subpopulations with iteration 1 are combined, then all training sets with iteration 2, and so on. Same is done for all test sets. The merged sets can be accessed via `$train_set(i)` and `$test_set(i)`, respectively.

Grouping / Blocking

All derived classes support grouping of observations. The grouping variable is assumed to be discrete and must be stored in the Task with column role `"group"`.

Observations in the same group are treated like a "block" of observations which must be kept together. These observations either all go together into the training set or together into the test set.

The sampling is performed by the derived Resampling on the grouping variable. Next, the grouping information is replaced with the respective row ids to generate training and test sets. The sets can be accessed via `$train_set(i)` and `$test_set(i)`, respectively.

See Also

Dictionary of Resamplings: mlr_resamplings

`as.data.table(mlr_resamplings)` for a complete table of all (also dynamically created) Resampling implementations.

Other Resampling: `mlr_resamplings`

Examples

```
r = rsmp("subsampling")

# Default parametrization
r$param_set$values

# Do only 3 repeats on 10% of the data
r$param_set$values = list(ratio = 0.1, repeats = 3)
r$param_set$values

# Instantiate on iris task
task = tsk("iris")
r$instantiate(task)

# Extract train/test sets
train_set = r$train_set(1)
print(train_set)
intersect(train_set, r$test_set(1))

# Another example: 10-fold CV
r = rsmp("cv")$instantiate(task)
r$train_set(1)

# Stratification
task = tsk("pima")
prop.table(table(task$truth())) # moderately unbalanced
task$col_roles$stratum = task$target_names

r = rsmp("subsampling")
r$instantiate(task)
prop.table(table(task$truth(r$train_set(1)))) # roughly same proportion
```

ResamplingBootstrap *Bootstrap Resampling*

Description

Splits data into bootstrap samples (sampling with replacement). Hyperparameters are the number of bootstrap iterations (`repeats`, default: 30) and the ratio of observations to draw per iteration (`ratio`, default: 1) for the training set.

Format

R6::R6Class inheriting from Resampling.

Construction

```
ResamplingBootstrap$new()
mlr_resamplings$get("bootstrap")
rsmp("bootstrap")
```

Fields

See Resampling.

Methods

See Resampling.

Parameters

- `repeats` :: `integer(1)`
  Number of repetitions.
- `ratio` :: `numeric(1)`
  Ratio of observations to put into the training set.

See Also

Dictionary of Resamplings: mlr_resamplings

`as.data.table(mlr_resamplings)` for a complete table of all (also dynamically created) Resampling implementations.

Examples

```
# Create a task with 10 observations
task = tsk("iris")
task$filter(1:10)

# Instantiate Resampling
rb = rsmp("bootstrap", repeats = 2, ratio = 1)
rb$instantiate(task)

# Individual sets:
rb$train_set(1)
rb$test_set(1)
intersect(rb$train_set(1), rb$test_set(1))

# Internal storage:
rb$instance$M # Matrix of counts
```

`ResamplingCustom` *Custom Resampling*

Description

Splits data into training and test sets using manually provided indices.

Format

R6::R6Class inheriting from Resampling.

Construction

```
ResamplingCustom$new()
mlr_resamplings$get("custom")
rsmp("custom")
```

Fields

See Resampling.

Methods

See Resampling.

See Also

Dictionary of Resamplings: mlr_resamplings

`as.data.table(mlr_resamplings)` for a complete table of all (also dynamically created) Resampling implementations.

Examples

```
# Create a task with 10 observations
task = tsk("iris")
task$filter(1:10)

# Instantiate Resampling
rc = rsmp("custom")
train_sets = list(1:5, 5:10)
test_sets = list(5:10, 1:5)
rc$instantiate(task, train_sets, test_sets)

rc$train_set(1)
rc$test_set(1)
```

`ResamplingCV` *Cross Validation Resampling*

Description

Splits data using a `folds`-folds (default: 10 folds) cross-validation.

Format

R6::R6Class inheriting from Resampling.

Construction

```
ResamplingCV$new()
mlr_resamplings$get("cv")
rsmp("cv")
```

Fields

See Resampling.

Methods

See Resampling.

Parameters

- `folds` :: `integer(1)`
  Number of folds.

See Also

Dictionary of Resamplings: mlr_resamplings

`as.data.table(mlr_resamplings)` for a complete table of all (also dynamically created) Resampling implementations.

Examples

```
# Create a task with 10 observations
task = tsk("iris")
task$filter(1:10)

# Instantiate Resampling
rcv = rsmp("cv", folds = 3)
rcv$instantiate(task)

# Individual sets:
rcv$train_set(1)
rcv$test_set(1)
```

```
intersect(rcv$train_set(1), rcv$test_set(1))

# Internal storage:
rcv$instance # table
```

`ResamplingHoldout` *Holdout Resampling*

Description

Splits data into a training set and a test set. Parameter `ratio` determines the ratio of observation going into the training set (default: 2/3).

Format

R6::R6Class inheriting from Resampling.

Construction

```
ResamplingHoldout$new()
mlr_resamplings$get("holdout")
rsmp("holdout")
```

Fields

See Resampling.

Methods

See Resampling.

Parameters

- `ratio` :: `numeric(1)`
  Ratio of observations to put into the training set.

See Also

Dictionary of Resamplings: mlr_resamplings

`as.data.table(mlr_resamplings)` for a complete table of all (also dynamically created) Resampling implementations.

Examples

```
# Create a task with 10 observations
task = tsk("iris")
task$filter(1:10)

# Instantiate Resampling
rho = rsmp("holdout", ratio = 0.5)
rho$instantiate(task)

# Individual sets:
rho$train_set(1)
rho$test_set(1)
intersect(rho$train_set(1), rho$test_set(1))

# Internal storage:
rho$instance # simple list
```

ResamplingRepeatedCV *Repeated Cross Validation Resampling*

Description

Splits data `repeats` (default: 10) times using a `folds`-fold (default: 10) cross-validation.

The iteration counter translates to `repeats` blocks of `folds` cross-validations, i.e., the first `folds` iterations belong to a single cross-validation.

Format

`R6::R6Class()` inheriting from Resampling.

Construction

```
ResamplingRepeatedCV$new()
mlr_resamplings$get("repeated_cv")
rsmp("repeated_cv")
```

Fields

See Resampling.

Methods

See Resampling. Additionally, the class provides two helper function to translate iteration numbers to folds / repeats:

- `folds(iters)`
  `integer()` -> `integer()`
  Translates iteration numbers to fold number.

- `repeats(iters)`
  `integer()` -> `integer()`
  Translates iteration numbers to repetition number.

Parameters

- `repeats` :: `integer(1)`
  Number of repetitions.
- `folds` :: `integer(1)`
  Number of folds.

See Also

Dictionary of Resamplings: mlr_resamplings

`as.data.table(mlr_resamplings)` for a complete table of all (also dynamically created) Resampling implementations.

Examples

```
# Create a task with 10 observations
task = tsk("iris")
task$filter(1:10)

# Instantiate Resampling
rrcv = rsmp("repeated_cv", repeats = 2, folds = 3)
rrcv$instantiate(task)
rrcv$iters
rrcv$folds(1:6)
rrcv$repeats(1:6)

# Individual sets:
rrcv$train_set(1)
rrcv$test_set(1)
intersect(rrcv$train_set(1), rrcv$test_set(1))

# Internal storage:
rrcv$instance # table
```

`ResamplingSubsampling` *Subsampling Resampling*

Description

Splits data `repeats` (default: 30) times into training and test set with a ratio of `ratio` (default: 2/3) observations going into the training set.

Format

R6::R6Class inheriting from Resampling.

Construction

```
ResamplingSubsampling$new()
mlr_resamplings$get("subsampling")
rsmp("subsampling")
```

Fields

See Resampling.

Methods

See Resampling.

Parameters

- `repeats` :: `integer(1)`
  Number of repetitions.
- `ratio` :: `numeric(1)`
  Ratio of observations to put into the training set.

See Also

Dictionary of Resamplings: mlr_resamplings

`as.data.table(mlr_resamplings)` for a complete table of all (also dynamically created) Resampling implementations.

Examples

```
# Create a task with 10 observations
task = tsk("iris")
task$filter(1:10)

# Instantiate Resampling
rss = rsmp("subsampling", repeats = 2, ratio = 0.5)
rss$instantiate(task)

# Individual sets:
rss$train_set(1)
rss$test_set(1)
intersect(rss$train_set(1), rss$test_set(1))

# Internal storage:
rss$instance$train # list of index vectors
```

Task *Task Class*

Description

This is the abstract base class for task objects like TaskClassif and TaskRegr.

Tasks serve two purposes:

1. Tasks wrap a DataBackend, an object to transparently interface different data storage types.
2. Tasks store meta-information, such as the role of the individual columns in the DataBackend. For example, for a classification task a single column must be marked as target column, and others as features.

Predefined (toy) tasks are stored in the mlr3misc::Dictionary mlr_tasks, e.g. `iris` or `boston_housing`.

Format

R6::R6Class object.

Construction

Note: This object is typically constructed via a derived classes, e.g. TaskClassif or TaskRegr.

```
t = Task$new(id, task_type, backend)
```

- `id` :: `character(1)`
  Identifier for the task.
- `task_type` :: `character(1)`
  Set in the classes which inherit from this class. Must be an element of mlr_reflections$task_types$type.
- `backend` :: DataBackend
  Either a DataBackend, or any object which is convertible to a DataBackend with `as_data_backend()`. E.g., a `data.frame()` will be converted to a DataBackendDataTable.

Fields

- `backend` :: DataBackend.
- `col_info` :: `data.table::data.table()`
  Table with with 3 columns:
  - `"id"` stores the name of the column.
  - `"type"` holds the storage type of the variable, e.g. `integer`, `numeric` or `character`.
  - `"levels"` stores a vector of distinct values (levels) for factor and character variables.
- `col_roles` :: named `list()`
  Each column (feature) can have an arbitrary number of the following roles:
  - `"feature"`: Regular feature used in the model fitting process.
  - `"target"`: Target variable.

  - `"name"`: Row names / observation labels. To be used in plots.
  - `"order"`: Data returned by `$data()` is ordered by this column (or these columns).
  - `"group"`: During resampling, observations with the same value of the variable with role "group" are marked as "belonging together". They will be exclusively assigned to be either in the training set or in the test set for each resampling iteration. Only up to one column may have this role.
  - `"stratum"`: Stratification variables. Multiple discrete columns may have this role.
  - `"weight"`: Observation weights. Only up to one column (assumed to be discrete) may have this role.

  `col_roles` keeps track of the roles with a named list, the elements are named by column role and each element is a character vector of column names. To alter the roles, just modify the list, e.g. with R's set functions (`intersect()`, `setdiff()`, `union()`, ...).

- `row_roles` :: named `list()`
  Each row (observation) can have an arbitrary number of roles in the learning task:
  - `"use"`: Use in train / predict / resampling.
  - `"validation"`: Hold the observations back unless explicitly requested. Validation sets are not yet completely integrated into the package.

  `row_roles` keeps track of the roles with a named list, elements are named by row role and each element is a `integer()` or `character()` vector of row ids. To alter the roles, just modify the list, e.g. with R's set functions (`intersect()`, `setdiff()`, `union()`, ...).
- `feature_names` :: `character()`
  Return all column names with `role == "feature"`.
- `feature_types` :: `data.table::data.table()`
  Returns a table with columns `id` and `type` where `id` are the column names of "active" features of the task and `type` is the storage type.
- `hash` :: `character(1)`
  Hash (unique identifier) for this object.
- `id` :: `character(1)`
  Identifier of the Task.
- `ncol` :: `integer(1)`
  Returns the total number of cols with role "target" or "feature".
- `nrow` :: `integer(1)`
  Return the total number of rows with role "use".
- `row_ids` :: (`integer()` | `character()`)
  Returns the row ids of the DataBackend for observations with with role "use".
- `target_names` :: `character()`
  Returns all column names with role "target".
- `task_type` :: `character(1)`
  Stores the type of the Task.
- `properties` :: `character()`
  Set of task properties. Possible properties are are stored in mlr_reflections$task_properties. The following properties are currently standardized and understood by tasks in **mlr3**:
  - `"strata"`: The task is resampled using one or more stratification variables (role `"stratum"`).

  - `"groups"`: The task comes with grouping/blocking information (role `"group"`).
  - `"weights"`: The task comes with observation weights (role `"weight"`). Note that above listed properties are calculated from the `$col_roles` and must not be set explicitly.
- `strata` :: `data.table::data.table()`
  If the task has columns designated with role `"stratum"`, returns a table with one subpopulation per row and two columns: `N` (`integer()`) with the number of observations in the subpopulation and `row_id` (list of `integer()` | list of `character()`) as list column with the row ids in the respective subpopulation. Returns `NULL` if there are is no stratification variable.

  See Resampling for more information on stratification.
- `groups` :: `data.table::data.table()`
  If the task has a column with designated role "group", table with two columns: `row_id` (`integer()` | `character()`) and the grouping variable `group` (`vector()`). Returns `NULL` if there are is no grouping column.

  See Resampling for more information on grouping.
- `weights` :: `data.table::data.table()`
  If the task has a column with designated role "weight", table with two columns: `row_id` (`integer()` | `character()`) and the observation weights `weight` (`numeric()`). Returns `NULL` if there are is no weight column.
- `man` :: `character(1)`
  String in the format `[pkg]::[topic]` pointing to a manual page for this object.

Methods

- `data(rows = NULL, cols = NULL, data_format = NULL)`
  `(integer() | character(), character(1), character(1)) -> any`
  Returns a slice of the data from the DataBackend in the data format specified by `data_format` (depending on the DataBackend, but usually a `data.table::data.table()`).

  Rows are additionally subsetted to only contain observations with role "use", and columns are filtered to only contain features with roles "target" and "feature". If invalid `rows` or `cols` are specified, an exception is raised.
- `formula(rhs = ".")`
  `character() -> stats::formula()`
  Constructs a `stats::formula()`, e.g. `[target] ~ [feature_1] + [feature_2] + ... + [feature_k]`, using the features provided in argument `rhs` (defaults to all columns with role `"feature"`, symbolized by `"."`).
- `levels(cols = NULL)`
  `character() -> named list()`
  Returns the distinct values for columns referenced in `cols` with storage type "character", "factor" or "ordered". Argument `cols` defaults to all such columns with role `"target"` or `"feature"`.

  Note that this function ignores the row roles, it returns all levels available in the DataBackend. To update the stored level information, e.g. after filtering a task, call `$droplevels()`.
- `droplevels(cols = NULL)`
  `character() -> self`
  Updates the cache of stored factor levels, removing all levels not present in the current set of active rows. `cols` defaults to all columns with storage type "character", "factor", or "ordered".

- `missings(cols = NULL)`
  `character()` -> named `integer()`
  Returns the number of missing observations for columns referenced in `cols`. Considers only active rows with row role `"use"`. Argument `cols` defaults to all columns with role "target" or "feature".
- `head(n = 6)`
  `integer()` -> `data.table::data.table()`
  Get the first `n` observations with role `"use"`.
- `set_row_role(rows, new_roles, exclusive = TRUE)`
  (`character()`, `character()`, `logical(1)`) -> `self`
  Adds the roles `new_roles` to rows referred to by `rows`. If `exclusive` is `TRUE`, the referenced rows will be removed from all other roles.

  This function is deprecated and will be removed in the next version in favor of directly modifying `$row_roles`.
- `set_col_role(cols, new_roles, exclusive = TRUE)`

  (`character()`, `character()`, `logical(1)`) -> `self`
  Adds the roles `new_roles` to columns referred to by `cols`. If `exclusive` is `TRUE`, the referenced columns will be removed from all other roles.

  This function is deprecated and will be removed in the next version in favor of directly modifying `$col_roles`.
- `filter(rows)`
  (`integer()` | `character()`) -> `self`
  Subsets the task, reducing it to only keep the rows specified in `rows`.

  This operation mutates the task in-place. See the section on task mutators for more information.
- `select(cols)`
  `character()` -> `self`
  Subsets the task, reducing it to only keep the features specified in `cols`. Note that you cannot deselect the target column, for obvious reasons.

  This operation mutates the task in-place. See the section on task mutators for more information.
- `cbind(data)`
  `data.frame()` -> `self`
  Adds additional columns to the DataBackend. The row ids must be provided as column in `data` (with column name matching the primary key name of the DataBackend). If this column is missing, it is assumed that the rows are exactly in the order of `t$row_ids`. In case of name clashes of column names in `data` and DataBackend, columns in `data` have higher precedence and virtually overwrite the columns in the DataBackend.

  This operation mutates the task in-place. See the section on task mutators for more information.
- `rbind(data)`
  `data.frame()` -> `self`
  Adds additional rows to the DataBackend. The new row ids must be provided as column in `data`. If this column is missing, new row ids are constructed automatically. In case of name

clashes of row ids, rows in `data` have higher precedence and virtually overwrite the rows in the DataBackend.

This operation mutates the task in-place. See the section on task mutators for more information.

- `rename(from,to)`
  (`character()`, `character()`) -> `self`
  Renames columns by mapping column names in `old` to new column names in `new`.

  This operation mutates the task in-place. See the section on task mutators for more information.
- `help()`
  () -> `NULL`
  Opens the corresponding help page referenced by `$man`.

S3 methods

- `as.data.table(t)`
  Task -> `data.table::data.table()`
  Returns the complete data as `data.table::data.table()`.

Task mutators

The following methods change the task in-place:

- Any modification to `$col_roles` and `row_roles`. This provides a different "view" on the data without altering the data itself.
- `filter()` and `select()` subset the set of active rows or features in `row_roles` or `col_roles`, respectively. This provides a different "view" on the data without altering the data itself.
- `rbind()` and `cbind()` change the task in-place by binding rows or columns to the data, but without modifying the original DataBackend. Instead, the methods first create a new DataBackendDataTable from the provided new data, and then merge both backends into an abstract DataBackend which combines the results on-demand.
- `rename()` wraps the DataBackend of the Task in an additional DataBackend which deals with the renaming. Also updates `col_roles` and `col_info`.

See Also

Other Task: `TaskClassif`, `TaskRegr`, `TaskSupervised`, `mlr_tasks`

Examples

```
# we use the inherited class TaskClassif here,
# Class Task is not intended for direct use
task = TaskClassif$new("iris", iris, target = "Species")

task$nrow
task$ncol
task$feature_names
task$formula()
```

```
# de-select "Petal.Width"
task$select(setdiff(task$feature_names, "Petal.Width"))

task$feature_names

# Add new column "foo"
task$cbind(data.frame(foo = 1:150))
task$head()
```

`TaskClassif` *Classification Task*

Description

This task specializes Task and TaskSupervised for classification problems. The target column is assumed to be a factor. The `task_type` is set to `"classif"`.

Additional task properties include:

- `"twoclass"`: The task is a binary classification problem.
- `"multiclass"`: The task is a multiclass classification problem.

Predefined tasks are stored in the mlr3misc::Dictionary mlr_tasks.

Format

R6::R6Class object inheriting from Task/TaskSupervised.

Construction

```
t = TaskClassif$new(id, backend, target, positive = NULL)
```

- `id` :: `character(1)`
  Identifier for the task.
- `backend` :: DataBackend
  Either a DataBackend, or any object which is convertible to a DataBackend with `as_data_backend()`. E.g., a `data.frame()` will be converted to a DataBackendDataTable.
- `target` :: `character(1)`
  Name of the target column.
- `positive` :: `character(1)`
  Only for binary classification: Name of the positive class. The levels of the target columns are reordered accordingly, so that the first element of `$class_names` is the positive class, and the second element is the negative class.

Fields

All methods from TaskSupervised, and additionally:

- `class_names` :: `character()`
  Returns all class labels of the target column.
- `positive` :: `character(1)`
  Stores the positive class for binary classification tasks, and `NA` for multiclass tasks. To switch the positive class, assign a level to this field.
- `negative` :: `character(1)`
  Stores the negative class for binary classification tasks, and `NA` for multiclass tasks.

Methods

See TaskSupervised.

See Also

Example classification tasks: `iris`

Other Task: `TaskRegr`, `TaskSupervised`, `Task`, `mlr_tasks`

Examples

```
data("Sonar", package = "mlbench")
task = TaskClassif$new("sonar", backend = Sonar, target = "Class", positive = "M")

task$task_type
task$formula()
task$truth()
task$class_names
task$positive

# possible properties:
mlr_reflections$task_properties$classif
```

`TaskGenerator` *TaskGenerator Class*

Description

Creates a Task of arbitrary size. Predefined task generators are stored in the mlr3misc::Dictionary mlr_task_generators, e.g. `xor`.

Format

R6::R6Class object.

Construction

```
g = TaskGenerator$new(id, task_type, packages = character(), param_set = ParamSet$new(), man = NA_chara
```

- `id` :: `character(1)`
  Identifier for the learner.
- `task_type` :: `character(1)`
  Type of the task the learner can operator on. E.g., `"classif"` or `"regr"`.
- `packages` :: `character()`
  Set of required packages. Note that these packages will be loaded via `requireNamespace()`, and are not attached.
- `param_set` :: paradox::ParamSet
  Set of hyperparameters.
- `man` :: `character(1)`
  String in the format `[pkg]::[topic]` pointing to a manual page for this object.

Fields

All variables passed to the constructor, and additionally:

- `task_type` :: `character(1)`
  Stores the type of class this learner can operate on, e.g. `"classif"` or `"regr"`. A complete list of task types is stored in `mlr_reflections$task_types$type`.

Methods

- `generate(n)`
  `integer(1)` -> Task
  Creates a task of type `task_type` with `n` observations, possibly using additional settings stored in `param_set`.
- `help()`
  () -> `NULL`
  Opens the corresponding help page referenced by `$man`.

See Also

Other TaskGenerator: `mlr_task_generators`

TaskGenerator2DNormals

2d Normals Classification Task Generator

Description

A TaskGenerator for the 2d normals task in `mlbench::mlbench.2dnormals()`.

Format

R6::R6Class inheriting from TaskGenerator.

Construction

```
TaskGenerator2DNormals$new()
mlr_task_generators$get("2dnormals")
tgen("2dnormals")
```

See Also

Dictionary of TaskGenerators: mlr_task_generators

`as.data.table(mlr_resamplings)` for a complete table of all (also dynamically created) TaskGenerator implementations.

Examples

```
tgen("2dnormals")$generate(10)$data()
```

TaskGeneratorFriedman1

Friedman1 Regression Task Generator

Description

A TaskGenerator for the friedman1 task in `mlbench::mlbench.friedman1()`.

Format

R6::R6Class inheriting from TaskGenerator.

Construction

```
TaskGeneratorFriedman1$new()
mlr_task_generators$get("friedman1")
tgen("friedman1")
```

See Also

Dictionary of TaskGenerators: mlr_task_generators

`as.data.table(mlr_resamplings)` for a complete table of all (also dynamically created) TaskGenerator implementations.

Examples

```
tgen("friedman1")$generate(10)$data()
```

TaskGeneratorSmiley *Smiley Classification Task Generator*

Description

A TaskGenerator for the smiley task in `mlbench::mlbench.smiley()`.

Format

R6::R6Class inheriting from TaskGenerator.

Construction

```
TaskGeneratorSmiley$new()
mlr_task_generators$get("smiley")
tgen("smiley")
```

See Also

Dictionary of TaskGenerators: mlr_task_generators

`as.data.table(mlr_resamplings)` for a complete table of all (also dynamically created) TaskGenerator implementations.

Examples

```
tgen("smiley")$generate(10)$data()
```

TaskGeneratorXor *XOR Classification Task Generator*

Description

A TaskGenerator for the xor task in `mlbench::mlbench.xor()`.

Format

R6::R6Class inheriting from TaskGenerator.

Construction

```
TaskGeneratorXor$new()
mlr_task_generators$get("xor")
tgen("xor")
```

See Also

Dictionary of TaskGenerators: mlr_task_generators

`as.data.table(mlr_resamplings)` for a complete table of all (also dynamically created) TaskGenerator implementations.

Examples

```
tgen("xor")$generate(10)$data()
```

`TaskRegr` *Regression Task*

Description

This task specializes Task and TaskSupervised for regression problems. The target column is assumed to be numeric. The `task_type` is set to `"classif"`.

Predefined tasks are stored in the mlr3misc::Dictionary mlr_tasks.

Format

R6::R6Class object inheriting from Task/TaskSupervised.

Construction

```
t = TaskRegr$new(id, backend, target)
```

- `id` :: `character(1)`
  Identifier for the task.
- `backend` :: (DataBackend | `data.frame()` | ...)
  Either a DataBackend, or any object which is convertible to a DataBackend with `as_data_backend()`. E.g., a `data.frame()` will be converted to a DataBackendDataTable.
- `target` :: `character(1)`
  Name of the target column.

Fields

See TaskSupervised.

Methods

See TaskSupervised.

See Also

Example regression tasks: `boston_housing`

Other Task: `TaskClassif`, `TaskSupervised`, `Task`, `mlr_tasks`

Examples

```
task = TaskRegr$new("iris", backend = iris, target = "Sepal.Length")
task$task_type
task$formula()
task$truth()

# possible properties:
mlr_reflections$task_properties$regr
```

Index