Package 'mlr3'

September 14, 2025

Title Machine Learning in R - Next Generation

Version 1.2.0

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 (>= 3.3.0)

Imports R6 (>= 2.4.1), backports (>= 1.5.0), checkmate (>= 2.0.0), cli, data.table (>= 1.15.0), evaluate (>= 1.0.4), future, future.apply (>= 1.5.0), lgr (>= 0.3.4), mirai (>= 2.4.1), methods, mlbench, mlr3measures (>= 1.0.0), mlr3misc (>= 0.19.0), parallelly, palmerpenguins, paradox (>= 1.0.1), uuid

Suggests callr, codetools, datasets, future.callr, mlr3data, progressr, remotes, RhpcBLASctl, rpart, testthat (>= 3.2.0)

Encoding UTF-8

Config/testthat/edition 3

Config/testthat/parallel false

NeedsCompilation no

RoxygenNote 7.3.3

Collate 'mlr_reflections.R' 'BenchmarkResult.R' 'CallbackResample.R' 'ContextResample.R' 'warn_deprecated.R' 'DataBackend.R' 'DataBackendCbind.R' 'DataBackendDataTable.R' 'DataBackendRbind.R' 'DataBackendRename.R' 'HotstartStack.R' 'Learner.R' 'LearnerClassif.R' 'mlr_learners.R' 'LearnerClassifDebug.R' 'LearnerClassifFeatureless.R'

'LearnerClassifRpart.R' 'LearnerRegr.R' 'LearnerRegrDebug.R'
'LearnerRegrFeatureless.R' 'LearnerRegrRpart.R' 'Measure.R'
'mlr_measures.R' 'MeasureAIC.R' 'MeasureBIC.R'
'MeasureClassif.R' 'MeasureClassifCosts.R' 'MeasureDebug.R'
'MeasureElapsedTime.R' 'MeasureInternalValidScore.R'
'MeasureOOBError.R' 'MeasureRegr.R' 'MeasureRegrPinball.R'
'MeasureRegrRQR.R' 'MeasureRegrRSQ.R'
'MeasureSelectedFeatures.R' 'MeasureSimilarity.R'
'MeasureSimple.R' 'Prediction.R' 'PredictionClassif.R'
'PredictionData.R' 'PredictionDataClassif.R'
'PredictionDataRegr.R' 'PredictionRegr.R' 'ResampleResult.R'
'Resampling.R' 'mlr_resamplings.R' 'ResamplingBootstrap.R'
'ResamplingCV.R' 'ResamplingCustom.R' 'ResamplingCustomCV.R'
'ResamplingHoldout.R' 'ResamplingInsample.R' 'ResamplingLOO.R'
'ResamplingRepeatedCV.R' 'ResamplingSubsampling.R'
'ResultData.R' 'Task.R' 'TaskSupervised.R' 'TaskClassif.R'
'mlr_tasks.R' 'TaskClassif_breast_cancer.R'
'TaskClassif_german_credit.R' 'TaskClassif_iris.R'
'TaskClassif_penguins.R' 'TaskClassif_pima.R'
'TaskClassif_sonar.R' 'TaskClassif_spam.R' 'TaskClassif_wine.R'
'TaskClassif_zoo.R' 'TaskGenerator.R' 'mlr_task_generators.R'
'TaskGenerator2DNormals.R' 'TaskGeneratorCassini.R'
'TaskGeneratorCircle.R' 'TaskGeneratorFriedman1.R'
'TaskGeneratorMoons.R' 'TaskGeneratorPeak.R'
'TaskGeneratorSimplex.R' 'TaskGeneratorSmiley.R'
'TaskGeneratorSpirals.R' 'TaskGeneratorXor.R' 'TaskRegr.R'
'TaskRegr_california_housing.R' 'TaskRegr_mtcars.R'
'TaskUnsupervised.R' 'as_benchmark_result.R'
'as_data_backend.R' 'as_learner.R' 'as_measure.R'
'as_prediction.R' 'as_prediction_classif.R'
'as_prediction_data.R' 'as_prediction_regr.R'
'as_resample_result.R' 'as_resampling.R' 'as_result_data.R'
'as_task.R' 'as_task_classif.R' 'as_task_regr.R'
'as_task_unsupervised.R' 'assertions.R' 'auto_convert.R'
'benchmark.R' 'benchmark_grid.R' 'bibentries.R'
'default_fallback.R' 'default_measures.R' 'fix_factor_levels.R'
'helper.R' 'helper_data_table.R' 'helper_exec.R'
'helper_hashes.R' 'install_pkgs.R' 'marshal.R'
'mlr_callbacks.R' 'mlr_sugar.R' 'mlr_test_helpers.R'
'partition.R' 'predict.R' 'reexports.R' 'resample.R'
'score_roc_measures.R' 'set_threads.R' 'set_validate.R'
'task_converters.R' 'worker.R' 'zzz.R'

Author Michel Lang [aut] (ORCID: <https://orcid.org/0000-0001-9754-0393>),
Bernd Bischl [aut] (ORCID: <https://orcid.org/0000-0001-6002-6980>),
Jakob Richter [aut] (ORCID: <https://orcid.org/0000-0003-4481-5554>),
Patrick Schratz [aut] (ORCID: <https://orcid.org/0000-0003-0748-6624>),
Giuseppe Casalicchio [ctb] (ORCID:
<https://orcid.org/0000-0001-5324-5966>),

Stefan Coors [ctb] (ORCID: <https://orcid.org/0000-0002-7465-2146>),
Quay Au [ctb] (ORCID: <https://orcid.org/0000-0002-5252-8902>),
Martin Binder [aut],
Florian Pfisterer [aut] (ORCID:
<https://orcid.org/0000-0001-8867-762X>),
Raphael Sonabend [aut] (ORCID: <https://orcid.org/0000-0001-9225-4654>),
Lennart Schneider [ctb] (ORCID:
<https://orcid.org/0000-0003-4152-5308>),
Marc Becker [cre, aut] (ORCID: <https://orcid.org/0000-0002-8115-0400>),
Sebastian Fischer [aut] (ORCID:
<https://orcid.org/0000-0002-9609-3197>),
Lona Koers [ctb],
John Zobolas [ctb] (ORCID: <https://orcid.org/0000-0002-3609-8674>),
Maximilian Mücke [ctb] (ORCID: <https://orcid.org/0009-0000-9432-9795>)

Maintainer Marc Becker <marcbecker@posteo.de>

Repository CRAN

Date/Publication 2025-09-13 23:30:02 UTC

Contents

`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.

Learn mlr3

- Book on mlr3: `https://mlr3book.mlr-org.com`
- Use cases and examples gallery: `https://mlr3gallery.mlr-org.com`
- Cheat Sheets: `https://github.com/mlr-org/mlr3cheatsheets`

mlr3 extensions

- Preprocessing and machine learning pipelines: **mlr3pipelines**
- Analysis of benchmark experiments: **mlr3benchmark**
- More classification and regression tasks: **mlr3data**
- Connector to OpenML: **mlr3oml**
- Solid selection of good classification and regression learners: **mlr3learners**
- Even more learners: `https://github.com/mlr-org/mlr3extralearners`
- Tuning of hyperparameters: **mlr3tuning**
- Hyperband tuner: **mlr3hyperband**
- Visualizations for many **mlr3** objects: **mlr3viz**
- Survival analysis and probabilistic regression: **mlr3proba**
- Cluster analysis: **mlr3cluster**
- Feature selection filters: **mlr3filters**
- Feature selection wrappers: **mlr3fselect**
- Interface to real (out-of-memory) data bases: **mlr3db**
- Performance measures as plain functions: **mlr3measures**
- Resampling methods for spatiotemporal data: **mlr3spatiotempcv**
- Data storage and prediction support for spatial objects: **mlr3spatial**

Suggested packages

- Parallelization framework: **future**
- Progress bars: **progressr**
- Encapsulated evaluation: **evaluate**, **callr** (external process)

Package Options

- "mlr3.exec_random": Randomize the order of execution in resample() and benchmark() during parallelization with **future**. Defaults to TRUE. Note that this does not affect the order of results.
- "mlr3.exec_chunk_size": Number of iterations to perform in a single future::future() during parallelization with **future**. Defaults to 1.
- "mlr3.exec_chunk_bins": Number of bins to split the iterations into. If set, "mlr3.exec_chunk_size" is ignored.
- "mlr3.debug": If set to TRUE, parallelization via **future** is disabled to simplify debugging and provide more concise tracebacks. Note that results computed in debug mode use a different seeding mechanism and are **not reproducible**.
- "mlr3.warn_version_mismatch": Set to FALSE to silence warnings raised during predict if a learner has been trained with a different version version of mlr3.
- "mlr3.prob_as_default": Set to TRUE to set the predict type of classification learners to "prob" by default (if they support it).
- "mlr3.mirai_parallelization": Compute profile to use for parallelization with **mirai**. Defaults to "mlr3_parallelization".
- "mlr3.mirai_encapsulation": Compute profile to use for encapsulation with **mirai**. Defaults to "mlr3_encapsulation".

Author(s)

Maintainer: Marc Becker <marcbecker@posteo.de> (ORCID)

Authors:

- Michel Lang <michellang@gmail.com> (ORCID)
- Bernd Bischl <bernd_bischl@gmx.net> (ORCID)
- Jakob Richter <jakob1richter@gmail.com> (ORCID)
- Patrick Schratz <patrick.schratz@gmail.com> (ORCID)
- Martin Binder <mlr.developer@mb706.com>
- Florian Pfisterer <pfistererf@googlemail.com> (ORCID)
- Raphael Sonabend <raphaelsonabend@gmail.com> (ORCID)
- Sebastian Fischer <sebf.fischer@gmail.com> (ORCID)

Other contributors:

- Giuseppe Casalicchio <giuseppe.casalicchio@stat.uni-muenchen.de> (ORCID) [contributor]
- Stefan Coors <mail@stefancoors.de> (ORCID) [contributor]
- Quay Au <quayau@gmail.com> (ORCID) [contributor]
- Lennart Schneider <lennart.sch@web.de> (ORCID) [contributor]
- Lona Koers <lona.koers@gmail.com> [contributor]
- John Zobolas <bblodfon@gmail.com> (ORCID) [contributor]
- Maximilian Mücke <muecke.maximilian@gmail.com> (ORCID) [contributor]

References

Lang M, Binder M, Richter J, Schratz P, Pfisterer F, Coors S, Au Q, Casalicchio G, Kotthoff L, Bischl B (2019). "mlr3: A modern object-oriented machine learning framework in R." *Journal of Open Source Software*. doi:10.21105/joss.01903, https://joss.theoj.org/papers/10.21105/joss.01903.

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

`assert_resample_callback` *Assertions for Callbacks*

Description

Assertions for CallbackResample class.

Usage

```
assert_resample_callback(callback, null_ok = FALSE)

assert_resample_callbacks(callbacks, null_ok = FALSE)
```

Arguments

`callback`	(CallbackResample).
`null_ok`	(`logical(1)`) If `TRUE`, `NULL` is allowed.
`callbacks`	(list of CallbackResample).

Value

CallbackResample | List of CallbackResamples.

as_benchmark_result *Convert to BenchmarkResult*

Description

Convert object to a BenchmarkResult.

Usage

```
as_benchmark_result(x, ...)

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

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

Arguments

x	(any) Object to convert.
...	(any) Additional arguments.

Value

(BenchmarkResult).

as_data_backend *Create a Data Backend*

Description

Wraps a DataBackend around data. **mlr3** ships with methods for `data.frame` (converted to a DataBackendDataTable.

Additional methods are implemented in the package **mlr3db**, e.g. to connect to real DBMS like PostgreSQL (via **dbplyr**) or DuckDB (via **DBI**/**duckdb**).

Usage

```
as_data_backend(data, primary_key = NULL, ...)

## S3 method for class 'data.frame'
as_data_backend(data, primary_key = NULL, keep_rownames = FALSE, ...)
```

Arguments

data	(data.frame()) The input data.frame(). Automatically converted to a data.table::data.table().
primary_key	(character(1) | integer()) Name of the primary key column, or integer vector of row ids.
...	(any) Additional arguments passed to the respective DataBackend method.
keep_rownames	(logical(1) | character(1)) If TRUE or a single string, keeps the row names of data as a new column. The column is named like the provided string, defaulting to "..rownames" for keep_rownames == TRUE. Note that the created column will be used as a regular feature by the task unless you manually change the column role. Also see data.table::as.data.table().

Value

DataBackend.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter10/advanced_technical_aspects_of_mlr3.html#sec-backends
- Package **mlr3db** to interface out-of-memory data, e.g. SQL servers or **duckdb**.

Other DataBackend: DataBackend, DataBackendDataTable

Examples

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

---

as_learner *Convert to a Learner*

Description

Convert object to a Learner or a list of Learner.

Usage

```
as_learner(x, ...)

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

as_learners(x, ...)
```

```
## Default S3 method:
as_learners(x, ...)

## S3 method for class 'list'
as_learners(x, ...)
```

Arguments

x	(any) Object to convert.
...	(any) Additional arguments.
clone	(logical(1)) If TRUE, ensures that the returned object is not the same as the input x.
discard_state	(logical(1)) Whether to discard the state.

Value

Learner.

---

as_measure *Convert to a Measure*

Description

Convert object to a Measure or a list of Measure.

Usage

```
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, ...)

## Default S3 method:
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, ...)
```

Arguments

x	(any) Object to convert.
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.
clone	(logical(1)) If TRUE, ensures that the returned object is not the same as the input x.
...	(any) Additional arguments.

Value

Measure.

---

as_prediction *Convert to a Prediction*

Description

Convert object to a Prediction or a list of Prediction.

Usage

```
as_prediction(x, check = FALSE, ...)

## S3 method for class 'Prediction'
as_prediction(x, check = FALSE, ...)

## S3 method for class 'PredictionDataClassif'
as_prediction(x, check = FALSE, ...)

## S3 method for class 'PredictionDataRegr'
as_prediction(x, check = FALSE, ...)

as_predictions(x, predict_sets = "test", ...)

## S3 method for class 'list'
as_predictions(x, predict_sets = "test", ...)
```

Arguments

x	(any) Object to convert.
check	(logical(1)) Perform argument checks and type conversions?

`...` (any)
Additional arguments.

`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 `{"train", "test", "internal_valid"}`. If multiple sets are provided, these are first combined to a single prediction object. Default is `"test"`.

Value

Prediction.

`as_prediction_classif` *Convert to a Classification Prediction*

Description

Convert object to a PredictionClassif.

Usage

```
as_prediction_classif(x, ...)

## S3 method for class 'PredictionClassif'
as_prediction_classif(x, ...)

## S3 method for class 'data.frame'
as_prediction_classif(x, ...)
```

Arguments

`x` (any)
Object to convert.

`...` (any)
Additional arguments.

Value

PredictionClassif.

Examples

```
# create a prediction object
task = tsk("penguins")
learner = lrn("classif.rpart", predict_type = "prob")
learner$train(task)
p = learner$predict(task)

# convert to a data.table
tab = as.data.table(p)

# convert back to a Prediction
as_prediction_classif(tab)

# split data.table into a list of data.tables
tabs = split(tab, tab$truth)

# convert back to list of predictions
preds = lapply(tabs, as_prediction_classif)

# calculate performance in each group
sapply(preds, function(p) p$score())
```

---

as_prediction_data *PredictionData*

Description

Convert object to a PredictionData or a list of PredictionData.

Usage

```
as_prediction_data(x, task, row_ids = task$row_ids, check = TRUE, ...)

## S3 method for class 'Prediction'
as_prediction_data(x, task, row_ids = task$row_ids, check = TRUE, ...)

## S3 method for class 'PredictionData'
as_prediction_data(x, task, row_ids = task$row_ids, check = TRUE, ...)

## S3 method for class 'list'
as_prediction_data(
  x,
  task,
  row_ids = task$row_ids,
  check = TRUE,
  ...,
  train_task
)
```

Arguments

x	(any) Object to convert.
task	(Task).
row_ids	integer() Row indices.
check	(logical(1)) Perform argument checks and type conversions?
...	(any) Additional arguments.
train_task	(Task) Task used for training the learner.

Value

PredictionData.

---

as_prediction_regr *Convert to a Regression Prediction*

Description

Convert object to a PredictionRegr.

Usage

```
as_prediction_regr(x, ...)

## S3 method for class 'PredictionRegr'
as_prediction_regr(x, ...)

## S3 method for class 'data.frame'
as_prediction_regr(x, ...)
```

Arguments

x	(any) Object to convert.
...	(any) Additional arguments.

Value

PredictionRegr.

Examples

```
# create a prediction object
task = tsk("mtcars")
learner = lrn("regr.rpart")
learner$train(task)
p = learner$predict(task)

# convert to a data.table
tab = as.data.table(p)

# convert back to a Prediction
as_prediction_regr(tab)

# split data.table into a list of data.tables
tabs = split(tab, cut(tab$truth, 3))

# convert back to list of predictions
preds = lapply(tabs, as_prediction_regr)

# calculate performance in each group
sapply(preds, function(p) p$score())
```

---

as_resample_result *Convert to ResampleResult*

Description

Convert object to a ResampleResult.

The S3 method for `list` expects argument `x` to be a list of Prediction objects and all other relevant objects (Task, Learners, and instantiated Resampling) must be provided, too. A more flexible way to manually create a ResampleResult is implemented in `as_result_data()`.

Usage

```
as_resample_result(x, ...)

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

## S3 method for class 'ResultData'
as_resample_result(x, view = NULL, ...)

## S3 method for class 'list'
as_resample_result(x, task, learners, resampling, store_backends = TRUE, ...)
```

Arguments

x	(any) Object to convert.
...	(any) Currently not used.
view	(character()) See construction argument view of ResampleResult.
task	(Task).
learners	(list of trained Learners).
resampling	(Resampling).
store_backends	(logical(1)) If set to FALSE, the backends of the Tasks provided in data are removed.

Value

(ResampleResult).

---

as_resampling *Convert to a Resampling*

Description

Convert object to a Resampling or a list of Resampling. This method e.g. allows to convert an OMLTask of **mlr3oml** to a Resampling.

Usage

```
as_resampling(x, ...)

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

as_resamplings(x, ...)

## Default S3 method:
as_resamplings(x, ...)

## S3 method for class 'list'
as_resamplings(x, ...)
```

Arguments

x	(any) Object to convert.
...	(any) Additional arguments.
clone	(logical(1)) If TRUE, ensures that the returned object is not the same as the input x.

---

as_result_data *Convert to ResultData*

Description

This function allows to construct or convert to a ResultData object, the result container used by ResampleResult and BenchmarkResult. A ResampleResult or BenchmarkResult can be initialized with the returned object. Note that ResampleResults can be converted to a BenchmarkResult with as_benchmark_result() and multiple BenchmarkResults can be combined to a larger BenchmarkResult with the $combine() method of BenchmarkResult.

Usage

```
as_result_data(
  task,
  learners,
  resampling,
  iterations,
  predictions,
  learner_states = NULL,
  data_extra = NULL,
  store_backends = TRUE
)
```

Arguments

task	(Task).
learners	(list of trained Learners).
resampling	(Resampling).
iterations	(integer()).
predictions	(list of list of Predictions).
learner_states	(list()) Learner states. If not provided, the states of learners are automatically extracted.
data_extra	(list()) Additional data for each iteration.
store_backends	(logical(1)) If set to FALSE, the backends of the Tasks provided in data are removed.

Value

`ResultData` object which can be passed to the constructor of ResampleResult.

Examples

```
task = tsk("penguins")
learner = lrn("classif.rpart")
resampling = rsmp("cv", folds = 2)$instantiate(task)
iterations = seq_len(resampling$iters)

# manually train two learners.
# store learners and predictions
learners = list()
predictions = list()
for (i in iterations) {
  l = learner$clone(deep = TRUE)
  learners[[i]] = l$train(task, row_ids = resampling$train_set(i))
  predictions[[i]] = list(test = l$predict(task, row_ids = resampling$test_set(i)))
}

rdata = as_result_data(task, learners, resampling, iterations, predictions)
ResampleResult$new(rdata)
```

---

as_task *Convert to a Task*

Description

Convert object to a Task or a list of Task.

The function supports:

- Converting existing Task objects (with optional cloning)
- Converting objects from other packages (e.g., `OMLTask` from **mlr3oml**)
- Converting lists of objects to lists of tasks

For constructing tasks from data frames, use the dedicated converters:

- `as_task_classif()` for classification tasks
- `as_task_regr()` for regression tasks
- `as_task_unsupervised()` for unsupervised tasks

Usage

```
as_task(x, ...)

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

```
as_tasks(x, ...)

## Default S3 method:
as_tasks(x, ...)

## S3 method for class 'list'
as_tasks(x, ...)
```

Arguments

x	(any) Object to convert.
...	(any) Additional arguments.
clone	(logical(1)) If TRUE, ensures that the returned object is not the same as the input x.

---

as_task_classif *Convert to a Classification Task*

Description

Convert object to a TaskClassif. This is a S3 generic. mlr3 ships with methods for the following objects:

1. TaskClassif: returns the object as-is, possibly cloned.
2. formula, data.frame(), matrix(), and DataBackend: provides an alternative to the constructor of TaskClassif.
3. TaskRegr: Calls convert_task().

Note that the target column will be converted to a factor(), if possible.

Usage

```
as_task_classif(x, ...)

## S3 method for class 'TaskClassif'
as_task_classif(x, clone = FALSE, ...)

## S3 method for class 'data.frame'
as_task_classif(
  x,
  target,
  id = deparse1(substitute(x)),
  positive = NULL,
  label = NA_character_,
  ...
```

```
)

## S3 method for class 'matrix'
as_task_classif(
  x,
  target,
  id = deparse1(substitute(x)),
  label = NA_character_,
  ...
)

## S3 method for class 'DataBackend'
as_task_classif(
  x,
  target,
  id = deparse1(substitute(x)),
  positive = NULL,
  label = NA_character_,
  ...
)

## S3 method for class 'TaskRegr'
as_task_classif(
  x,
  target,
  drop_original_target = FALSE,
  drop_levels = TRUE,
  ...
)

## S3 method for class 'formula'
as_task_classif(
  x,
  data,
  id = deparse1(substitute(data)),
  positive = NULL,
  label = NA_character_,
  ...
)
```

Arguments

x	(any) Object to convert.
...	(any) Additional arguments.
clone	(logical(1)) If TRUE, ensures that the returned object is not the same as the input x.

target (character(1))
Name of the target column.

id (character(1))
Id for the new task. Defaults to the (deparsed and substituted) name of the data argument.

positive (character(1))
Level of the positive class. See TaskClassif.

label (character(1))
Label for the new instance.

drop_original_target
(logical(1))
If FALSE (default), the original target is added as a feature. Otherwise the original target is dropped.

drop_levels (logical(1))
If TRUE (default), unused levels of the new target variable are dropped.

data (data.frame())
Data frame containing all columns referenced in formula x.

Value

TaskClassif.

Examples

```
as_task_classif(palmerpenguins::penguins, target = "species")
```

---

as_task_regr *Convert to a Regression Task*

Description

Convert object to a TaskRegr. This is a S3 generic. mlr3 ships with methods for the following objects:

1. TaskRegr: returns the object as-is, possibly cloned.
2. formula, data.frame(), matrix(), and DataBackend: provides an alternative to the constructor of TaskRegr.
3. TaskClassif: Calls convert_task().

Usage

```
as_task_regr(x, ...)

## S3 method for class 'TaskRegr'
as_task_regr(x, clone = FALSE, ...)

## S3 method for class 'data.frame'
as_task_regr(
  x,
  target,
  id = deparse1(substitute(x)),
  label = NA_character_,
  ...
)

## S3 method for class 'matrix'
as_task_regr(
  x,
  target,
  id = deparse1(substitute(x)),
  label = NA_character_,
  ...
)

## S3 method for class 'DataBackend'
as_task_regr(
  x,
  target,
  id = deparse1(substitute(x)),
  label = NA_character_,
  ...
)

## S3 method for class 'TaskClassif'
as_task_regr(x, target, drop_original_target = FALSE, drop_levels = TRUE, ...)

## S3 method for class 'formula'
as_task_regr(
  x,
  data,
  id = deparse1(substitute(data)),
  label = NA_character_,
  ...
)
```

Arguments

x	(any) Object to convert.
...	(any) Additional arguments.
clone	(logical(1)) If TRUE, ensures that the returned object is not the same as the input x.
target	(character(1)) Name of the target column.
id	(character(1)) Id for the new task. Defaults to the (deparsed and substituted) name of the data argument.
label	(character(1)) Label for the new instance.
drop_original_target	(logical(1)) If FALSE (default), the original target is added as a feature. Otherwise the original target is dropped.
drop_levels	(logical(1)) If TRUE (default), unused levels of the new target variable are dropped.
data	(data.frame()) Data frame containing all columns referenced in formula x.

Value

TaskRegr.

Examples

```
as_task_regr(datasets::mtcars, target = "mpg")
```

---

as_task_unsupervised *Convert to an Unsupervised Task*

Description

Convert object to a TaskUnsupervised or a list of TaskUnsupervised.

Usage

```
as_task_unsupervised(x, ...)

## S3 method for class 'Task'
as_task_unsupervised(x, clone = FALSE, ...)
```

```
## S3 method for class 'data.frame'
as_task_unsupervised(
  x,
  id = deparse1(substitute(x)),
  label = NA_character_,
  ...
)

## S3 method for class 'DataBackend'
as_task_unsupervised(
  x,
  id = deparse1(substitute(x)),
  label = NA_character_,
  ...
)

as_tasks_unsupervised(x, ...)

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

## S3 method for class 'Task'
as_tasks_unsupervised(x, clone = FALSE, ...)
```

Arguments

x	(any) Object to convert.
...	(any) Additional arguments.
clone	(logical(1)) If TRUE, ensures that the returned object is not the same as the input x.
id	(character(1)) Id for the new task. Defaults to the (deparsed and substituted) name of the data argument.
label	(character(1)) Label for the new instance.

---

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.

For large-scale benchmarking we recommend to use the **mlr3batchmark** package. This package runs benchmark experiments on high-performance computing clusters and handles failed experiments.

Usage

```
benchmark(
  design,
  store_models = FALSE,
  store_backends = TRUE,
  encapsulate = NA_character_,
  allow_hotstart = FALSE,
  clone = c("task", "learner", "resampling"),
  unmarshal = TRUE,
  callbacks = NULL
)
```

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. Additionally, you can set the additional column 'param_values', see benchmark_grid().
store_models	(logical(1)) Store the fitted model in the resulting object= Set to TRUE if you want to further analyse the models or want to extract information like variable importance.
store_backends	(logical(1)) Keep the DataBackend of the Task in the ResampleResult? Set to TRUE if your performance measures require a Task, or to analyse results more conveniently. Set to FALSE to reduce the file size and memory footprint after serialization. The current default is TRUE, but this eventually will be changed in a future release.
encapsulate	(character(1)) If not NA, enables encapsulation by setting the field Learner$encapsulate to one of the supported values: "none" (disable encapsulation), "try" (captures errors but output is printed to the console and not logged), "evaluate" (execute via **evaluate**) and "callr" (start in external session via **callr**). If NA, encapsulation is not changed, i.e. the settings of the individual learner are active. Additionally, if encapsulation is set to "evaluate" or "callr", the fallback learner is set to the featureless learner if the learner does not already have a fallback configured.
allow_hotstart	(logical(1)) Determines if learner(s) are hot started with trained models in $hotstart_stack. See also HotstartStack.
clone	(character()) Select the input objects to be cloned before proceeding by providing a set with

possible values "task", "learner" and "resampling" for Task, Learner and Resampling, respectively. Per default, all input objects are cloned.

unmarshal — Learner
Whether to unmarshal learners that were marshaled during the execution. If TRUE all models are stored in unmarshaled form. If FALSE, all learners (that need marshaling) are stored in marshaled form.

callbacks — (List of mlr3misc::Callback)
Callbacks to be executed during the resampling process. See CallbackResample and ContextResample for details.

Value

BenchmarkResult.

Stochasticity

Note that uninstantiated Resamplings are instantiated on the task, making the function stochastic even in case of deterministic learners.

Predict Sets

If you want to compare the performance of a learner on the training with the performance on the test set, you have to configure the Learner to predict on multiple sets by setting the field predict_sets to c("train", "test") (default is "test"). Each set yields a separate Prediction object during resampling. In the next step, you have to configure the measures to operate on the respective Prediction object:

```
m1 = msr("classif.ce", id = "ce.train", predict_sets = "train")
m2 = msr("classif.ce", id = "ce.test", predict_sets = "test")
```

The (list of) created measures can finally be passed to $aggregate() or $score().

Parallelization

This function can be parallelized with the **future** or **mirai** package. One job is one resampling iteration. All jobs are send to an apply function from **future.apply** or mirai::mirai_map() in a single batch. To select a parallel backend, use future::plan(). To use mirai, call mirai::daemons(.compute = "mlr3_parallelization") before calling this function. The future package guarantees reproducible results independent of the parallel backend. The results of mirai will not be the same but can be made reproducible by setting a seed when calling mirai::daemons(). More on parallelization can be found in the book: https://mlr3book.mlr-org.com/chapters/chapter10/advanced_technical_aspects_of_mlr3.html

Progress Bars

This function supports progress bars via the package **progressr**. Simply wrap the function call in progressr::with_progress() to enable them. Alternatively, call progressr::handlers() with global = TRUE to enable progress bars globally. We recommend the **progress** package as backend which can be enabled with progressr::handlers("progress").

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`.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-benchmarking
- Package **mlr3viz** for some generic visualizations.
- **mlr3benchmark** for post-hoc analysis of benchmark results.

Other benchmark: `BenchmarkResult`, `benchmark_grid()`

Examples

```
# benchmarking with benchmark_grid()
tasks = lapply(c("penguins", "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 penguins with a 3-fold CV
# - fit classif.rpart on sonar using a holdout
tasks = list(tsk("penguins"), 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 penguins
rr = bmr$aggregate()[learner_id == "classif.featureless"]$resample_result[[1]]
rr$resampling$train_set(2)
```

`BenchmarkResult` *Container for Benchmarking Results*

Description

This is the result container object returned by `benchmark()`. A BenchmarkResult consists of the data of multiple ResampleResults. The contents of a `BenchmarkResult` and ResampleResult are almost identical and the stored ResampleResults can be extracted via the `$resample_result(i)` method, where i is the index of the performed resample experiment. This allows us to investigate the extracted ResampleResult and individual resampling iterations, as well as the predictions and models from each fold.

BenchmarkResults can be visualized via **mlr3viz**'s `autoplot()` function.

For statistical analysis of benchmark results and more advanced plots, see **mlr3benchmark**.

S3 Methods

- `as.data.table(rr, ..., reassemble_learners = TRUE, convert_predictions = TRUE, predict_sets = "test", task_characteristics = FALSE)`
  BenchmarkResult -> `data.table::data.table()`
  Returns a tabular view of the internal data.

- `c(...)`
  (BenchmarkResult, ...) -> BenchmarkResult
  Combines multiple objects convertible to BenchmarkResult into a new BenchmarkResult.

Active bindings

`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"`. This is `NA` for empty BenchmarkResults.

`tasks` (`data.table::data.table()`)
: Table of included Tasks with three columns:
  - `"task_hash"` (`character(1)`),
  - `"task_id"` (`character(1)`), and
  - `"task"` (Task).

`learners` (`data.table::data.table()`)
: Table of included Learners with three columns:
  - `"learner_hash"` (`character(1)`),
  - `"learner_id"` (`character(1)`), and
  - `"learner"` (Learner).

  Note that it is not feasible to access learned models via this field, as the training task would be ambiguous. For this reason the returned learner are reset before they are returned. Instead, select a row from the table returned by `$score()`.

`resamplings` (`data.table::data.table()`)
: Table of included Resamplings with three columns:
  - `"resampling_hash"` (`character(1)`),
  - `"resampling_id"` (`character(1)`), and
  - `"resampling"` (Resampling).

`resample_results` (`data.table::data.table()`)
: Returns a table with three columns:
  - `uhash` (`character()`).
  - `resample_result` (ResampleResult).

`n_resample_results` (`integer(1)`)
: Returns the total number of stored ResampleResults.

`uhashes` (`character()`)
: Set of (unique) hashes of all included ResampleResults.

`uhash_table` (data.table::data.table)
: Table with columns `uhash`, `learner_id`, `task_id` and `resampling_id`.

Methods

Public methods:

- `BenchmarkResult$new()`
- `BenchmarkResult$help()`

- BenchmarkResult$format()
- BenchmarkResult$print()
- BenchmarkResult$combine()
- BenchmarkResult$marshal()
- BenchmarkResult$unmarshal()
- BenchmarkResult$score()
- BenchmarkResult$obs_loss()
- BenchmarkResult$aggregate()
- BenchmarkResult$filter()
- BenchmarkResult$resample_result()
- BenchmarkResult$discard()
- BenchmarkResult$set_threshold()
- BenchmarkResult$clone()

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
BenchmarkResult$new(data = NULL)
```

Arguments:

`data` (`ResultData`)
An object of type `ResultData`, either extracted from another ResampleResult, another BenchmarkResult, or manually constructed with `as_result_data()`.

Method `help()`**:** Opens the help page for this object.

Usage:

```
BenchmarkResult$help()
```

Method `format()`**:** Helper for print outputs.

Usage:

```
BenchmarkResult$format(...)
```

Arguments:

`...` (ignored).

Method `print()`**:** Printer.

Usage:

```
BenchmarkResult$print()
```

Method `combine()`**:** Fuses a second BenchmarkResult into itself, mutating the BenchmarkResult in-place. If the second BenchmarkResult `bmr` is `NULL`, simply returns `self`. Note that you can alternatively use the combine function `c()` which calls this method internally.

Usage:

```
BenchmarkResult$combine(bmr)
```

Arguments:

`bmr` (BenchmarkResult)
A second BenchmarkResult object.

Returns: Returns the object itself, but modified **by reference**. You need to explicitly `$clone()` the object beforehand if you want to keep the object in its previous state.

Method `marshal()`**:** Marshals all stored models.

Usage:

```
BenchmarkResult$marshal(...)
```

Arguments:

`...` (any)
Additional arguments passed to `marshal_model()`.

Examples:

```
bmr$marshal()
```

Method `unmarshal()`**:** Unmarshals all stored models.

Usage:

```
BenchmarkResult$unmarshal(...)
```

Arguments:

`...` (any)
Additional arguments passed to `unmarshal_model()`.

Examples:

```
bmr$unmarshal()
```

Method `score()`**:** 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 scores as extra columns. These columns are named using the id of the respective Measure.

Usage:

```
BenchmarkResult$score(
  measures = NULL,
  ids = TRUE,
  conditions = FALSE,
  predictions = TRUE
)
```

Arguments:

`measures` (Measure | list of Measure)
Measure(s) to calculate.

`ids` (`logical(1)`)
Adds object ids (`"task_id"`, `"learner_id"`, `"resampling_id"`) as extra character columns to the returned table.

`conditions` (`logical(1)`)
Adds condition messages (`"warnings"`, `"errors"`) as extra list columns of character vectors to the returned table

`predictions (logical(1))`
Additionally return prediction objects, one column for each `predict_set` of all learners combined. Columns are named `"prediction_train"`, `"prediction_test"` and `"prediction_internal_valid"`, if present.

Returns: `data.table::data.table()`.

Examples:

```
bmr$score(msr("classif.acc"))
```

Method `obs_loss()`**:** Calculates the observation-wise loss via the loss function set in the Measure's field `obs_loss`. Returns a `data.table()` with the columns `row_ids`, `truth`, `response` and one additional numeric column for each measure, named with the respective measure id. If there is no observation-wise loss function for the measure, the column is filled with `NA` values. Note that some measures such as RMSE, do have an `$obs_loss`, but they require an additional transformation after aggregation, in this example taking the square-root.

Usage:

```
BenchmarkResult$obs_loss(measures = NULL, predict_sets = "test")
```

Arguments:

`measures` (Measure | list of Measure)
Measure(s) to calculate.

`predict_sets (character())`
The predict sets.

Examples:

```
bmr$obs_loss(msr("classif.acc"))
```

Method `aggregate()`**:** 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.

The method for aggregation is controlled by the Measure, e.g. micro aggregation, macro aggregation or custom aggregation. Most measures default to macro aggregation.

Note that the aggregated performances just give a quick impression which approaches work well and which approaches are probably underperforming. However, the aggregates do not account for variance and cannot replace a statistical test. See **mlr3viz** to get a better impression via boxplots or **mlr3benchmark** for critical difference plots and significance tests.

For convenience, different flags can be set to extract more information from the returned ResampleResult.

Usage:

```
BenchmarkResult$aggregate(
  measures = NULL,
  ids = TRUE,
  uhashes = FALSE,
  params = FALSE,
  conditions = FALSE
)
```

Arguments:

`measures` (Measure | list of Measure)
Measure(s) to calculate.

`ids` (`logical(1)`)
Adds object ids (`"task_id"`, `"learner_id"`, `"resampling_id"`) as extra character columns for convenient subsetting.

`uhashes` (`logical(1)`)
Adds the uhash values of the ResampleResult as extra character column `"uhash"`.

`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"`.

Returns: `data.table::data.table()`.

Examples:

```
bmr$aggregate()
```

Method `filter()`**:** Subsets the benchmark result. You can either directly provide the row IDs or the uhashes of the resample results to keep, or use the `learner_ids`, `task_ids` and `resampling_ids` arguments to filter for learner, task and resampling IDs. The three options are mutually exclusive.

Usage:

```
BenchmarkResult$filter(
  i = NULL,
  uhashes = NULL,
  learner_ids = NULL,
  task_ids = NULL,
  resampling_ids = NULL
)
```

Arguments:

`i` (`integer()` | `NULL`)
The iteration values to filter for.

`uhashes` (`character()` | `NULL`)
The uhashes of the resample results to filter for.

`learner_ids` (`character()` | `NULL`)
The learner IDs to filter for.

`task_ids` (`character()` | `NULL`)
The task IDs to filter for.

`resampling_ids` (`character()` | `NULL`)
The resampling IDs to filter for.

Returns: Returns the object itself, but modified **by reference**. You need to explicitly `$clone()` the object beforehand if you want to keeps the object in its previous state.

Examples:

```
design = benchmark_grid(
  tsks(c("iris", "sonar")),
  lrns(c("classif.debug", "classif.featureless")),
  rsmp("holdout")
)
bmr = benchmark(design)
bmr
bmr2 = bmr$clone(deep = TRUE)
bmr2$filter(learner_ids = "classif.featureless")
bmr2
```

Method `resample_result()`**:** Retrieve the i-th ResampleResult, by position, by unique hash `uhash` or by learner, task and resampling IDs. All three options are mutually exclusive.

Usage:

```
BenchmarkResult$resample_result(
  i = NULL,
  uhash = NULL,
  task_id = NULL,
  learner_id = NULL,
  resampling_id = NULL
)
```

Arguments:

`i (integer(1) | NULL)`
: The iteration value to filter for.

`uhash (character(1) | NULL)`
: The unique identifier to filter for.

`task_id (character(1) | NULL)`
: The task ID to filter for.

`learner_id (character(1) | NULL)`
: The learner ID to filter for.

`resampling_id (character(1) | NULL)`
: The resampling ID to filter for.

Returns: ResampleResult.

Examples:

```
design = benchmark_grid(
  tsk("iris"),
  lrns(c("classif.debug", "classif.featureless")),
  rsmp("holdout")
)
bmr = benchmark(design)
bmr$resample_result(learner_id = "classif.featureless")
bmr$resample_result(i = 1)
bmr$resample_result(uhash = uhashes(bmr, learner_id = "classif.debug"))
```

Method `discard()`**:** Shrinks the BenchmarkResult by discarding parts of the internally stored data. Note that certain operations might stop work, e.g. extracting importance values from learners or calculating measures requiring the task's data.

Usage:

```
BenchmarkResult$discard(backends = FALSE, models = FALSE)
```

Arguments:

`backends (logical(1))`
If `TRUE`, the DataBackend is removed from all stored Tasks.

`models (logical(1))`
If `TRUE`, the stored model is removed from all Learners.

Returns: Returns the object itself, but modified **by reference**. You need to explicitly `$clone()` the object beforehand if you want to keeps the object in its previous state.

Examples:

```
bmr$discard(models = TRUE)
```

Method `set_threshold()`**:** Sets the threshold for the response prediction of classification learners, given they have output a probability prediction for a binary classification task.
The resample results for which to change the threshold can either be specified directly via `uhashes`, by selecting the specific iterations (`i`) or by filtering according to learner, task and resampling IDs.
If none of the three options is specified, the threshold is set for all resample results.

Usage:

```
BenchmarkResult$set_threshold(
  threshold,
  i = NULL,
  uhashes = NULL,
  learner_ids = NULL,
  task_ids = NULL,
  resampling_ids = NULL,
  ties_method = "random"
)
```

Arguments:

`threshold (numeric(1))`
Threshold value.

`i (integer() | NULL)`
The iteration values to filter for.

`uhashes (character() | NULL)`
The unique identifiers of the ResampleResults for which the threshold should be set.

`learner_ids (character() | NULL)`
The learner IDs for which the threshold should be set.

`task_ids (character() | NULL)`
The task IDs for which the threshold should be set.

`resampling_ids (character() | NULL)`
The resampling IDs for which the threshold should be set.

`ties_method (character(1))`
Method to handle ties in probabilities when selecting a class label. Must be one of `"random"`, `"first"` or `"last"` (corresponding to the same options in `max.col()`).

- `"random"`: Randomly select one of the tied class labels (default).

- `"first"`: Select the first class label among tied values.
- `"last"`: Select the last class label among tied values.

Examples:

```
design = benchmark_grid(
  tsk("sonar"),
  lrns(c("classif.debug", "classif.featureless"), predict_type = "prob"),
  rsmp("holdout")
)
bmr = benchmark(design)
bmr$set_threshold(0.8, learner_ids = "classif.featureless")
bmr$set_threshold(0.3, i = 2)
bmr$set_threshold(0.7, uhashes = uhashes(bmr, learner_ids = "classif.featureless"))
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Note

All stored objects are accessed by reference. Do not modify any extracted object without cloning it first.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-benchmarking
- Package **mlr3viz** for some generic visualizations.
- **mlr3benchmark** for post-hoc analysis of benchmark results.

Other benchmark: `benchmark()`, `benchmark_grid()`

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("penguins")),
  learners = learners,
  resamplings = rsmp("cv", folds = 3)
)
print(design)

bmr = benchmark(design)
```

```
print(bmr)

bmr$tasks
bmr$learners

# first 5 resampling iterations
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

# reduce to subset with task id "sonar"
bmr$filter(task_ids = "sonar")
print(bmr)

## ------------------------------------------------
## Method `BenchmarkResult$marshal`
## ------------------------------------------------

bmr$marshal()

## ------------------------------------------------
## Method `BenchmarkResult$unmarshal`
## ------------------------------------------------

bmr$unmarshal()

## ------------------------------------------------
## Method `BenchmarkResult$score`
## ------------------------------------------------

bmr$score(msr("classif.acc"))

## ------------------------------------------------
## Method `BenchmarkResult$obs_loss`
## ------------------------------------------------

bmr$obs_loss(msr("classif.acc"))

## ------------------------------------------------
## Method `BenchmarkResult$aggregate`
## ------------------------------------------------
```

```
bmr$aggregate()

## ------------------------------------------------
## Method `BenchmarkResult$filter`
## ------------------------------------------------

design = benchmark_grid(
  tsks(c("iris", "sonar")),
  lrns(c("classif.debug", "classif.featureless")),
  rsmp("holdout")
)
bmr = benchmark(design)
bmr
bmr2 = bmr$clone(deep = TRUE)
bmr2$filter(learner_ids = "classif.featureless")
bmr2

## ------------------------------------------------
## Method `BenchmarkResult$resample_result`
## ------------------------------------------------

design = benchmark_grid(
  tsk("iris"),
  lrns(c("classif.debug", "classif.featureless")),
  rsmp("holdout")
)
bmr = benchmark(design)
bmr$resample_result(learner_id = "classif.featureless")
bmr$resample_result(i = 1)
bmr$resample_result(uhash = uhashes(bmr, learner_id = "classif.debug"))

## ------------------------------------------------
## Method `BenchmarkResult$discard`
## ------------------------------------------------

bmr$discard(models = TRUE)

## ------------------------------------------------
## Method `BenchmarkResult$set_threshold`
## ------------------------------------------------

design = benchmark_grid(
  tsk("sonar"),
  lrns(c("classif.debug", "classif.featureless"), predict_type = "prob"),
  rsmp("holdout")
)
bmr = benchmark(design)
bmr$set_threshold(0.8, learner_ids = "classif.featureless")
bmr$set_threshold(0.3, i = 2)
bmr$set_threshold(0.7, uhashes = uhashes(bmr, learner_ids = "classif.featureless"))
```

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).

There are two modes of operation, depending on the flag `paired`.

- With `paired` set to `FALSE` (default), resampling strategies are not allowed to be instantiated, and instead will be instantiated per task internally. The only exception to this rule applies if all tasks have exactly the same row ids, and the resamplings are all instantiated for such tasks. The grid will be generated based on the Cartesian product of tasks, learners, and resamplings. Because the resamplings are instantiated on the tasks, reproducibility requires a seed to be set **before** calling this function, as this process is stochastic.
- With `paired` set to `TRUE`, tasks and resamplings are treated as pairs. This means that you must provide as many tasks as corresponding instantiated resamplings. The grid will be generated based on the Cartesian product of learners and pairs.

Usage

```
benchmark_grid(
  tasks,
  learners,
  resamplings,
  param_values = NULL,
  paired = FALSE
)
```

Arguments

`tasks`	(list of Task).
`learners`	(list of Learner).
`resamplings`	(list of Resampling).
`param_values`	(`list()`) If you want to try many parameter settings for learners, you can pass them through the design which is optimized to be faster than creating learners for each setting. A list of lists of named lists, from outer to inner: 1. One list element for each Learner. 2. One list element for each hyperparameter configuration to try. 3. Named list of hyperparameter settings to set in the Learner, possibly overwriting already set set hyperparameters in the Learner.
`paired`	(`logical(1)`) Set this to `TRUE` if the resamplings are instantiated on the tasks, i.e., the tasks and resamplings are paired. You need to provide the same number of tasks and instantiated resamplings.

Value

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

Errors and Warnings

- Mlr3WarningVaryingPredictTypes: This warning will be thrown if the learners have different predict_types.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-benchmarking
- Package **mlr3viz** for some generic visualizations.
- **mlr3benchmark** for post-hoc analysis of benchmark results.

Other benchmark: BenchmarkResult, benchmark()

Examples

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

# Set a seed to ensure reproducibility of the resampling instantiation
set.seed(123)
grid = benchmark_grid(tasks, learners, resamplings)
# the resamplings are now instantiated
head(grid$resampling[[1]]$instance)
print(grid)
## Not run:
benchmark(grid)

## End(Not run)

# paired
learner = lrn("classif.rpart")
task1 = tsk("penguins")
task2 = tsk("german_credit")
res1 = rsmp("holdout")
res2 = rsmp("holdout")
res1$instantiate(task1)
res2$instantiate(task2)
design = benchmark_grid(list(task1, task2), learner, list(res1, res2), paired = TRUE)
print(design)

# manual construction of the grid with data.table::CJ()
grid = data.table::CJ(
 task = tasks,
  learner = learners,
  resampling = resamplings,
  sorted = FALSE
```

```
)

# manual instantiation (not suited for a fair comparison of learners!)
Map(function(task, resampling) {
  resampling$instantiate(task)
}, task = grid$task, resampling = grid$resampling)
## Not run:
benchmark(grid)

## End(Not run)
```

`california_housing` *Median House Value in California*

Description

A regression task to predict the median house value in California.

Contains 9 features and 20640 observations. Target column is `"median_house_value"`.

Format

R6::R6Class inheriting from TaskRegr.

Construction

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

Meta Information

- Task type: “regr”
- Dimensions: 20640x10
- Properties: -
- Has Missings: `TRUE`
- Target: “median_house_value”
- Features: “households”, “housing_median_age”, “latitude”, “longitude”, “median_income”, “ocean_proximity”, “population”, “total_bedrooms”, “total_rooms”

Source

https://www.kaggle.com/datasets/camnugent/california-housing-prices

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3data** for more toy tasks.
- Package **mlr3oml** for downloading tasks from https://www.openml.org.
- Package **mlr3viz** for some generic visualizations.
- Dictionary of Tasks: mlr_tasks
- `as.data.table(mlr_tasks)` for a table of available Tasks in the running session (depending on the loaded packages).
- **mlr3fselect** and **mlr3filters** for feature selection and feature filtering.
- Extension packages for additional task types:
  - Unsupervised clustering: **mlr3cluster**
  - Probabilistic supervised regression and survival analysis: https://mlr3proba.mlr-org.com/.

Other Task: `Task`, `TaskClassif`, `TaskRegr`, `TaskSupervised`, `TaskUnsupervised`, `mlr_tasks`, `mlr_tasks_breast_cancer`, `mlr_tasks_german_credit`, `mlr_tasks_iris`, `mlr_tasks_mtcars`, `mlr_tasks_penguins`, `mlr_tasks_pima`, `mlr_tasks_sonar`, `mlr_tasks_spam`, `mlr_tasks_wine`, `mlr_tasks_zoo`

CallbackResample *Resample Callback*

Description

Specialized mlr3misc::Callback to customize the behavior of `resample()` and `benchmark()` in mlr3. For example, callbacks can be used to extract information from models on the worker or to store intermediate results to disk. The `callback_resample()` function is used to create instances of this class. Predefined callbacks are stored in the dictionary mlr_callbacks and can be retrieved with `clbk()`. For more information on callbacks, see the `callback_resample()` documentation.

Super class

`mlr3misc::Callback` -> `CallbackResample`

Public fields

`on_resample_begin (function())`
: Stage called at the beginning of the resampling iteration. Called in `workhorse()` (internal).

`on_resample_before_train (function())`
: Stage called before training the learner. Called in `workhorse()` (internal).

`on_resample_before_predict (function())`
: Stage called before predicting. Called in `workhorse()` (internal).

`on_resample_end (function())`
: Stage called at the end of the resample iteration. Called in `workhorse()` (internal).

Methods

Public methods:

- `CallbackResample$clone()`

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`callback_resample` *Create Evaluation Callback*

Description

Function to create a CallbackResample. Predefined callbacks are stored in the dictionary mlr_callbacks and can be retrieved with `clbk()`.

Evaluation callbacks are called at different stages of the resampling process. Each stage is called once per resampling iteration. The stages are prefixed with `on_resample_*`. The text in brackets indicates what happens between the stages in the internal `workhorse()` function and which accesses to the ContextResample (`ctx`) are typical for the stage.

```
Start Resampling Iteration on Worker
 - on_resample_begin
  (Split `ctx$task` into training and test set with `ctx$resampling` and `ctx$iteration`)
 - on_resample_before_train
   (Train the learner `ctx$learner` on training data)
 - on_resample_before_predict
   (Predict on predict sets and store prediction data `ctx$pdatas`)
 - on_resample_end
   (Erase model `ctx$learner$model` if requested and return results)
End Resampling Iteration on Worker
```

The callback can store data in `ctx$learner$state` or `ctx$data_extra`. The data in `ctx$data_extra` is stored in the ResampleResult or BenchmarkResult. See also the section on parameters for more information on the stages.

Usage

```
callback_resample(
  id,
  label = NA_character_,
  man = NA_character_,
  on_resample_begin = NULL,
  on_resample_before_train = NULL,
  on_resample_before_predict = NULL,
  on_resample_end = NULL
)
```

Arguments

`id` (`character(1)`)
Identifier for the new instance.

`label` (`character(1)`)
Label for the new instance.

`man` (`character(1)`)
String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

`on_resample_begin` (`function()`)
Stage called at the beginning of an evaluation. Called in `workhorse()` (internal).

`on_resample_before_train` (`function()`)
Stage called before training the learner. Called in `workhorse()` (internal).

`on_resample_before_predict` (`function()`)
Stage called before predicting. Called in `workhorse()` (internal).

`on_resample_end` (`function()`)
Stage called at the end of an evaluation. Called in `workhorse()` (internal).

Details

When implementing a callback, each function must have two arguments named `callback` and `context`. A callback can write data to the state (`$state`), e.g. settings that affect the callback itself. We highly discourage changing the task, learner and resampling objects via the callback.

Examples

```
learner = lrn("classif.rpart")
task = tsk("pima")
resampling = rsmp("cv", folds = 3)

# save selected features callback
callback = callback_resample("selected_features",
 on_resample_end = function(callback, context) {
    context$learner$state$selected_features = context$learner$selected_features()
  }
)

rr = resample(task, learner, resampling, callbacks = callback)
rr$learners[[1]]$state$selected_features

# holdout task callback
callback = callback_resample("holdout_task",
  on_resample_before_predict = function(callback, context) {
    pred = context$learner$predict(callback$state$task)
    context$data_extra = list(prediction_holdout = pred)
  }
```

```
)

task_holdout = tsk("pima")
splits = partition(task, 0.7)
task$filter(splits$train)
task_holdout$filter(splits$test)

callback$state$task = task_holdout

rr = resample(task, learner, resampling, callbacks = callback)
rr$data_extra
```

`ContextResample` *Resample Context*

Description

A CallbackResample accesses and modifies data during `resample()` and `benchmark()` via the `ContextResample`. See the section on fields for a list of modifiable objects. See `callback_resample()` for a list of stages that access `ContextResample`.

Super class

`mlr3misc::Context` -> `ContextResample`

Active bindings

`task` (Task)
: The task to be evaluated. The task is unchanged during the evaluation. The task is read-only.

`learner` (Learner)
: The learner to be evaluated. The learner contains the models after stage `on_resample_before_train`.

`resampling` Resampling
: The resampling strategy to be used. The resampling is unchanged during the evaluation. The resampling is read-only.

`iteration` (`integer()`)
: The current iteration. The iteration is read-only.

`pdatas` (List of PredictionData)
: The prediction data. The data is available on stage `on_resample_end`.

`data_extra` (list())
: Data saved in the ResampleResult or BenchmarkResult. Use this field to save results. Must be a `list()`.

Methods

Public methods:

- `ContextResample$new()`
- `ContextResample$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
ContextResample$new(task, learner, resampling, iteration)
```

Arguments:

`task` (Task)
The task to be evaluated.

`learner` (Learner)
The learner to be evaluated.

`resampling` (Resampling)
The resampling strategy to be used.

`iteration` (`integer()`)
The current iteration.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`convert_task` *Convert a Task from One Type to Another*

Description

The task's target is replaced by a different column from the data.

Usage

```
convert_task(
  intask,
  target = NULL,
  new_type = NULL,
  drop_original_target = FALSE,
  drop_levels = TRUE
)
```

Arguments

`intask`	(Task) A Task to be converted.
`target`	(`character(1)`) New target to be set, must be a column in the `intask` data. If `NULL`, no new target is set, and task is converted as-is.
`new_type`	(`character(1)`) The new task type. Must be in `mlr_reflections$task_types`]. If `NULL` (default), a new task with the same task_type is created.
`drop_original_target`	(`logical(1)`) If `FALSE` (default), the original target is added as a feature. Otherwise the original target is dropped.
`drop_levels`	(`logical(1)`) If `TRUE` (default), unused levels of the new target variable are dropped.

Value

Task of requested type.

`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.

This package currently shups with one implementation for backends:

- DataBackendDataTable which stores the data as `data.table::data.table()`.

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

Details

The required set of fields and methods to implement a custom `DataBackend` is listed in the respective sections (see DataBackendDataTable).

Public fields

`primary_key (character(1))`
Column name of the primary key column of positive and unique integer row ids.

Active bindings

`hash` (`character(1)`)
: Hash (unique identifier) for this object.

`col_hashes` (named `character`)
: Hash (unique identifier) for all columns except the `primary_key`: A `character` vector, named by the columns that each element refers to.
Columns of different `Task`s or `DataBackend`s that have agreeing `col_hashes` always represent the same data, given that the same `row`s are selected. The reverse is not necessarily true: There can be columns with the same content that have different `col_hashes`.

Methods

Public methods:

- `DataBackend$new()`
- `DataBackend$format()`
- `DataBackend$print()`

Method `new()`**:** Creates a new instance of this R6 class.
Note: This object is typically constructed via a derived classes, e.g. DataBackendDataTable, or via the S3 method `as_data_backend()`.

Usage:

```
DataBackend$new(data, primary_key)
```

Arguments:

`data` (any)
: The format of the input data depends on the specialization. E.g., DataBackendDataTable expects a `data.table::data.table()`.

`primary_key` (`character(1)`)
: Each DataBackend needs a way to address rows, which is done via a column of unique integer values, referenced here by `primary_key`. The use of this variable may differ between backends.

Method `format()`**:** Helper for print outputs.

Usage:

```
DataBackend$format(...)
```

Arguments:

`...` (ignored).

Method `print()`**:** Printer.

Usage:

```
DataBackend$print()
```

See Also

- Chapter in the mlr3book: `https://mlr3book.mlr-org.com/chapters/chapter10/advanced_technical_aspects_of_mlr3.html#sec-backends`
- Package **mlr3db** to interface out-of-memory data, e.g. SQL servers or **duckdb**.

Other DataBackend: `DataBackendDataTable`, `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** which serves as an efficient in-memory data base.

Super class

`mlr3::DataBackend` -> `DataBackendDataTable`

Public fields

`compact_seq logical(1)`
: If `TRUE`, row ids are a natural sequence from 1 to `nrow(data)` (determined internally). In this case, row lookup uses faster positional indices instead of equi joins.

Active bindings

`rownames (integer())`
: Returns vector of all distinct row identifiers, i.e. the contents of the primary key column.

`colnames (character())`
: Returns vector of all column names, including the primary key column.

`nrow (integer(1))`
: Number of rows (observations).

`ncol (integer(1))`
: Number of columns (variables), including the primary key column.

Methods

Public methods:

- `DataBackendDataTable$new()`
- `DataBackendDataTable$data()`
- `DataBackendDataTable$head()`
- `DataBackendDataTable$distinct()`

- DataBackendDataTable$missings()

Method new(): Creates a new instance of this R6 class.

Note that DataBackendDataTable does not copy the input data, while as_data_backend() calls data.table::copy(). as_data_backend() also takes care about casting to a data.table() and adds a primary key column if necessary.

Usage:

```
DataBackendDataTable$new(data, primary_key)
```

Arguments:

data (data.table::data.table())
: The input data.table().

primary_key (character(1) | integer())
: Name of the primary key column, or integer vector of row ids.

Method data(): Returns a slice of the data. 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.

Usage:

```
DataBackendDataTable$data(rows, cols)
```

Arguments:

rows (positive integer())
: Vector or row indices. Always refers to the complete data set, even after filtering.

cols (character())
: Vector of column names.

Method head(): Retrieve the first n rows.

Usage:

```
DataBackendDataTable$head(n = 6L)
```

Arguments:

n (integer(1))
: Number of rows.

Returns: data.table::data.table() of the first n rows.

Method distinct(): 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.

Usage:

```
DataBackendDataTable$distinct(rows, cols, na_rm = TRUE)
```

Arguments:

rows (positive integer())
: Vector or row indices. Always refers to the complete data set, even after filtering.

`cols` (`character()`)
Vector of column names.

`na_rm` `logical(1)`
Whether to remove NAs or not.

Returns: Named `list()` of distinct values.

Method `missings()`**:** Returns the number of missing values per column in the specified slice of data. Non-existing rows and columns are silently ignored.

Usage:

```
DataBackendDataTable$missings(rows, cols)
```

Arguments:

`rows` (positive `integer()`)
Vector or row indices. Always refers to the complete data set, even after filtering.

`cols` (`character()`)
Vector of column names.

Returns: Total of missing values per column (named `numeric()`).

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter10/advanced_technical_aspects_of_mlr3.html#sec-backends
- Package **mlr3db** to interface out-of-memory data, e.g. SQL servers or **duckdb**.

Other DataBackend: `DataBackend`, `as_data_backend()`

Examples

```
data = as.data.table(palmerpenguins::penguins)
data$id = seq_len(nrow(palmerpenguins::penguins))
b = DataBackendDataTable$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(palmerpenguins::penguins)
```

default_fallback *Create a Fallback Learner*

Description

Create a fallback learner for a given learner. The function searches for a suitable fallback learner based on the task type. Additional checks are performed to ensure that the fallback learner supports the predict type.

Usage

```
default_fallback(learner, ...)

## S3 method for class 'Learner'
default_fallback(learner, ...)

## S3 method for class 'LearnerClassif'
default_fallback(learner, ...)

## S3 method for class 'LearnerRegr'
default_fallback(learner, ...)
```

Arguments

`learner`	Learner The learner for which a fallback learner should be created.
`...`	any ignored.

Value

Learner

default_measures *Get the 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")
```

HotstartStack *Stack for Hot Start Learners*

Description

This class stores learners for hot starting training, i.e. resuming or continuing from an already fitted model. We assume that hot starting is only possible if a single hyperparameter (also called the fidelity parameter, usually controlling the complexity or expensiveness) is altered and all other hyperparameters are identical.

The HotstartStack stores trained learners which can be potentially used to hot start a learner. Learner automatically hot start while training if a stack is attached to the $hotstart_stack field and the stack contains a suitable learner.

For example, if you want to train a random forest learner with 1000 trees but already have a random forest learner with 500 trees (hot start learner), you can add the hot start learner to the HotstartStack of the expensive learner with 1000 trees. If you now call the train() method (or resample() or benchmark()), a random forest with 500 trees will be fitted and combined with the 500 trees of the hotstart learner, effectively saving you to fit 500 trees.

Hot starting is only supported by learners which have the property "hotstart_forward" or "hotstart_backward". For example, an xgboost model (in **mlr3learners**) can hot start forward by adding more boosting iterations, and a random forest can go backwards by removing trees. The fidelity parameters are tagged with "hotstart" in learner's parameter set.

Public fields

stack data.table::data.table()
: Stores hot start learners.

hotstart_threshold (named numeric(1))
: Threshold for storing learners in the stack. If the value of the hotstart parameter is below this threshold, the learner is not added to the stack.

Methods

Public methods:

- `HotstartStack$new()`
- `HotstartStack$add()`
- `HotstartStack$start_cost()`
- `HotstartStack$format()`
- `HotstartStack$print()`
- `HotstartStack$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
HotstartStack$new(learners = NULL, hotstart_threshold = NULL)
```

Arguments:

`learners` (List of Learners)
Learners are added to the hotstart stack. If `NULL` (default), empty stack is created.

`hotstart_threshold` (named `numeric(1)`)
Threshold for storing learners in the stack.

Method `add()`**:** Add learners to hot start stack.

Usage:

```
HotstartStack$add(learners)
```

Arguments:

`learners` (List of Learners). Learners are added to the hotstart stack.

Returns: self (invisibly).

Method `start_cost()`**:** Calculates the cost for each learner of the stack to hot start the target `learner`.

The following cost values can be returned:

- `NA_real_`: Learner is unsuitable to hot start target `learner`.
- `-1`: Hotstart learner in the stack and target `learner` are identical.
- `0` Cost for hot starting backwards is always 0.
- `> 0` Cost for hot starting forward.

Usage:

```
HotstartStack$start_cost(learner, task_hash)
```

Arguments:

`learner` Learner
Target learner.

`task_hash` Task
Hash of the task on which the target learner is trained.

Method `format()`**:** Helper for print outputs.

Usage:

`HotstartStack$format(...)`

Arguments:

`...` (ignored).

Method `print()`**:** Printer.

Usage:

`HotstartStack$print(...)`

Arguments:

`...` (ignored).

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

Usage:

`HotstartStack$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# train learner on pima task
task = tsk("pima")
learner = lrn("classif.debug", iter = 1)
learner$train(task)

# initialize stack with previously fitted learner
hot = HotstartStack$new(list(learner))

# retrieve learner with increased fidelity parameter
learner = lrn("classif.debug", iter = 2)

# calculate cost of hot starting
hot$start_cost(learner, task$hash)

# add stack with hot start learner
learner$hotstart_stack = hot

# train automatically uses hot start learner while fitting the model
learner$train(task)
```

`install_pkgs` *Install (Missing) Packages*

Description

`extract_pkgs()` extracts required package from various objects, including TaskGenerator, Learner, Measure and objects from extension packages such as **mlr3pipelines** or **mlr3filters**. If applied on a list, the function is called recursively on all elements.

`install_pkgs()` calls `extract_pkgs()` internally and proceeds with the installation of extracted packages.

Usage

```
install_pkgs(x, ...)

extract_pkgs(x)

## S3 method for class 'character'
extract_pkgs(x)

## S3 method for class 'R6'
extract_pkgs(x)

## S3 method for class 'list'
extract_pkgs(x)

## S3 method for class 'ResampleResult'
extract_pkgs(x)

## S3 method for class 'BenchmarkResult'
extract_pkgs(x)
```

Arguments

x	(any) Object with package information (or a list of such objects).
...	(any) Additional arguments passed down to `remotes::install_cran()` or `remotes::install_github()`. Arguments `force` and `upgrade` are often important in this context.

Details

If a package contains a forward slash ('/'), it is assumed to be a package hosted on GitHub in `"<user>/<repo>"` format, and the string will be passed to `remotes::install_github()`. Otherwise, the package name will be passed to `remotes::install_cran()`.

Value

`extract_pkgs()` returns a `character()` of package strings, `install_pkgs()` returns the names of extracted packages invisibly.

Examples

```
extract_pkgs(lrns(c("regr.rpart", "regr.featureless")))
```

`Learner` *Learner Class*

Description

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

Learners are build around the three following key parts:

- Methods `$train()` and `$predict()` which call internal methods or private methods `$.train()`/`$.predict()`).
- 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.
- The fitted model stored in field `$model`, available after calling `$train()`.

Predefined learners are stored in the dictionary mlr_learners, e.g. `classif.rpart` or `regr.rpart`.

More classification and regression learners are implemented in the add-on package **mlr3learners**. Learners for survival analysis (or more general, for probabilistic regression) can be found in **mlr3proba**. Unsupervised cluster algorithms are implemented in **mlr3cluster**. The dictionary mlr_learners gets automatically populated with the new learners as soon as the respective packages are loaded.

More (experimental) learners can be found in the GitHub repository: `https://github.com/mlr-org/mlr3extralearners`. A guide on how to extend **mlr3** with custom learners can be found in the mlr3book.

To combine the learner with preprocessing operations like factor encoding, **mlr3pipelines** is recommended. Hyperparameters stored in the `param_set` can be tuned with **mlr3tuning**.

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, see 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"`.
- `internal_valid_scores`: Returns the internal validation score(s) of the model as a named `list()`. Only available for `Learner`s with the `"validation"` property. If the learner is not trained yet, this returns `NULL`.
- `internal_tuned_values`: Returns the internally tuned hyperparameters of the model as a named `list()`. Only available for `Learner`s with the `"internal_tuning"` property. If the learner is not trained yet, this returns `NULL`.

Weights

Many learners support observation weights, indicated by their property `"weights"`. The weights are stored in the Task where the column role `weights_learner` needs to be assigned to a single numeric column. If a task has weights and the learner supports them, they are used automatically. If a task has weights but the learner does not support them, an error is thrown by default. Both of these behaviors can be disabled by setting the `use_weights` field to `"ignore"`. See the description of `use_weights` for more information.

If the learner is set-up to use weights but the task does not have a designated weight column, samples are considered to have equal weight. When weights are being used, they are passed down to the learner directly; the effect of weights depends on the specific learner. Generally, weights do not need to sum up to 1.

When implementing a Learner that uses weights, the `"weights"` property should be set. The `$.train()`-method should then call the `$.get_weights()`-method to retrieve the weights from the task. `$.get_weights()` will automatically discard weights when `use_weights` is set to `"ignore"`;

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, call the `set_values()` method on the `param_set`:

```
lrn = lrn("classif.rpart")
lrn$param_set$set_values(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 factor hyperparameter with id `"foo"` and possible levels `"a"` and `"b"`:

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

Implementing Validation

Some Learners, such as `XGBoost`, other boosting algorithms, or deep learning models (`mlr3torch`), utilize validation data during the training to prevent overfitting or to log the validation performance. It is possible to configure learners to be able to receive such an independent validation set during training. To do so, one must:

- annotate the learner with the `"validation"` property
- implement the active binding `$internal_valid_scores` (see section *Optional Extractors*), as well as the private method `$.extract_internal_valid_scores()` which returns the (final) internal validation scores from the model of the `Learner` and returns them as a named `list()` of `numeric(1)`. If the model is not trained yet, this method should return `NULL`.

- Add the `validate` parameter, which can be either `NULL`, a ratio in $(0, 1)$, `"test"`, or `"predefined"`:
  - `NULL`: no validation
  - `ratio`: only proportion `1 - ratio` of the task is used for training and `ratio` is used for validation.
  - `"test"` means that the `"test"` task is used. **Warning**: This can lead to biased performance estimation. This option is only available if the learner is being trained via `resample()`, `benchmark()` or functions that internally use them, e.g. `tune()` of **mlr3tuning** or `batchmark()` of **mlr3batchmark**. This is especially useful for hyperparameter tuning, where one might e.g. want to use the same validation data for early stopping and model evaluation.
  - `"predefined"` means that the task's (manually set) `$internal_valid_task` is used. See the `Task` documentation for more information.

For an example how to do this, see `LearnerClassifDebug`. Note that in `.train()`, the `$internal_valid_task` will only be present if the `$validate` field of the `Learner` is set to a non-`NULL` value.

Implementing Internal Tuning

Some learners such as `XGBoost` or `cv.glmnet` can internally tune hyperparameters. XGBoost, for example, can tune the number of boosting rounds based on the validation performance. CV Glmnet, on the other hand, can tune the regularization parameter based on an internal cross-validation. Internal tuning *can* therefore rely on the internal validation data, but does not necessarily do so.

In order to be able to combine this internal hyperparamer tuning with the standard hyperparameter optimization implemented via **mlr3tuning**, one most:

- annotate the learner with the `"internal_tuning"` property
- implement the active binding `$internal_tuned_values` (see section *Optional Extractors*) as well as the private method `$.extract_internal_tuned_values()` which extracts the internally tuned values from the `Learner`'s model and returns them as a named `list()`. If the model is not trained yet, this method should return `NULL`.
- Have at least one parameter tagged with `"internal_tuning"`, which requires to also provide a `in_tune_fn` and `disable_tune_fn`, and *should* also include a default aggregation function.

For an example how to do this, see `LearnerClassifDebug`.

Implementing Marshaling

Some `Learner`s have models that cannot be serialized as they e.g. contain external pointers. In order to still be able to save them, use them with parallelization or callr encapsulation it is necessary to implement how they should be (un)-marshaled. See `marshaling` for how to do this.

Implementing Out-of-Bag Error

Some `Learner`s can compute the out-of-bag error during training. In order to do this, the learner must:

- annotate the learner with the `"oob_error"` property
- implement the private method `$.extract_oob_error()` which extracts the out-of-bag error from the `Learner`'s model and returns it as a `numeric(1)`.

Public fields

`id` (`character(1)`)
: Identifier of the object. Used in tables, plot and text output.

`label` (`character(1)`)
: Label for this object. Can be used in tables, plot and text output instead of the ID.

`state` (`NULL` | named `list()`)
: Current (internal) state of the learner. Contains all information gathered during `train()` and `predict()`. It is not recommended to access elements from `state` directly. This is an internal data structure which may change in the future.

`task_type` (`character(1)`)
: Task type, e.g. `"classif"` or `"regr"`.

  For a complete list of possible task types (depending on the loaded packages), see `mlr_reflections$task_types$type`

`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(1)`)
: Set of required packages. These packages are loaded, but not attached.

`predict_sets` (`character()`)
: During `resample()`/`benchmark()`, a Learner can predict on multiple sets. Per default, a learner only predicts observations in the test set (`predict_sets == "test"`). To change this behavior, set `predict_sets` to a non-empty subset of `{"train", "test", "internal_valid"}`. The `"train"` predict set contains the train ids from the resampling. This means that if a learner does validation and sets `$validate` to a ratio (creating the validation data from the training data), the train predictions will include the predictions for the validation data. Each set yields a separate Prediction object. Those can be combined via getters in ResampleResult/BenchmarkResult, or Measures can be configured to operate on specific subsets of the calculated prediction sets.

`parallel_predict` (`logical(1)`)
: If set to `TRUE`, use **future** to calculate predictions in parallel (default: `FALSE`). The row ids of the `task` will be split into `future::nbrOfWorkers()` chunks, and predictions are evaluated according to the active `future::plan()`. This currently only works for methods `Learner$predict()` and `Learner$predict_newdata()`, and has no effect during `resample()` or `benchmark()` where you have other means to parallelize.

  Note that the recorded time required for prediction reports the time required to predict is not properly defined and depends on the parallelization backend.

`timeout` (named `numeric(2)`)
: Timeout for the learner's train and predict steps, in seconds. This works differently for different encapsulation methods, see `mlr3misc::encapsulate()`. Default is `c(train = Inf, predict = Inf)`. Also see the section on error handling the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter10/advanced_technical_aspects_of_mlr3.html#sec-error-handling

`man` (`character(1)`)
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. Defaults to `NA`, but can be set by child classes.

Active bindings

`use_weights` (`character(1)`)
: How weights should be handled. Settings are `"use"` `"ignore"`, and `"error"`.

  - `"use"`: use weights, as supported by the underlying `Learner`. Only available for `Learners` with the property `"weights"`.
  - `"ignore"`: do not use weights.
  - `"error"`: throw an error if weights are present in the training `Task`.

  For `Learners` with the property `"weights"`, this is initialized as `"use"`. For `Learners` that do not support weights, i.e. without the `"weights"` property, this is initialized as `"error"`. The latter behavior is to avoid cases where a user erroneously assumes that a `Learner` supports weights when it does not. For `Learners` that do not support weights, `use_weights` needs to be set to `"ignore"` if tasks with weights should be handled (by dropping the weights). See Section 'weights' for more details.

`model` (any)
: The fitted model. Only available after `$train()` has been called.

`timings` (named `numeric(2)`)
: Elapsed time in seconds for the steps `"train"` and `"predict"`.

  When predictions for multiple predict sets were made during `resample()` or `benchmark()`, the predict time shows the cumulative duration of all predictions. If `learner$predict()` is called manually, the last predict time gets overwritten.

  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", or "error"), and
  - `"msg"` (`character()`).

`warnings` (`character()`)
: Logged warnings as vector.

`errors` (`character()`)
: Logged errors as vector.

`hash` (`character(1)`)
: Hash (unique identifier) for this object. The hash is calculated based on the learner id, the parameter settings, the predict type, the fallback hash, the parallel predict setting, the validate setting, and the predict sets.

`phash` (`character(1)`)
: Hash (unique identifier) for this partial object, excluding some components which are varied systematically during tuning (parameter values).

`predict_type` (`character(1)`)
: Stores the currently active predict type, e.g. `"response"`. Must be an element of `$predict_types`. A few learners already use the predict type during training. So there is no guarantee that changing the predict type after training will have any effect or does not lead to errors.

`param_set` (paradox::ParamSet)
: Set of hyperparameters.

`fallback` (Learner)
: Returns the fallback learner set with `$encapsulate()`.

`encapsulation` (`character(2)`)
: Returns the encapsulation settings set with `$encapsulate()`.

`hotstart_stack` (HotstartStack)
: . Stores `HotstartStack`.

`selected_features_impute` (`character(1)`)
: Controls the behavior if the learner does not support feature selection. If set to `"error"`, an error is thrown. If set to `"all"` the complete feature set is returned.

`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`. This field is read-only.

Methods

Public methods:

- `Learner$new()`
- `Learner$format()`
- `Learner$print()`
- `Learner$help()`
- `Learner$train()`
- `Learner$predict()`
- `Learner$predict_newdata()`
- `Learner$reset()`
- `Learner$base_learner()`
- `Learner$encapsulate()`
- `Learner$configure()`
- `Learner$selected_features()`
- `Learner$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

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

Usage:

```
Learner$new(
  id,
  task_type,
  param_set = ps(),
  predict_types = character(),
  feature_types = character(),
  properties = character(),
  packages = character(),
  label = NA_character_,
  man = NA_character_
)
```

Arguments:

`id (character(1))`
Identifier for the new instance.

`task_type (character(1))`
Type of task, e.g. `"regr"` or `"classif"`. Must be an element of mlr_reflections$task_types$type.

`param_set (paradox::ParamSet)`
Set of hyperparameters.

`predict_types (character())`
Supported predict types. Must be a subset of `mlr_reflections$learner_predict_types`.

`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.
- `"offset"`: The learner can incorporate offset values to adjust predictions.
- `"importance"`: The learner supports extraction of importance scores, i.e. comes with an `$importance()` extractor function (see section on optional extractors in Learner).
- `"selected_features"`: The learner supports extraction of the set of selected features, i.e. comes with a `$selected_features()` extractor function (see section on optional extractors in Learner).
- `"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 in Learner).
- `"validation"`: The learner can use a validation task during training.
- `"internal_tuning"`: The learner is able to internally optimize hyperparameters (those are also tagged with `"internal_tuning"`).
- `"marshal"`: To save learners with this property, you need to call `$marshal()` first. If a learner is in a marshaled state, you call first need to call `$unmarshal()` to use its model, e.g. for prediction.
- `"hotstart_forward"`: The learner supports to hotstart a model forward.
- `"hotstart_backward"`: The learner supports hotstarting a model backward.
- ‘"featureless": The learner does not use features.

`packages (character())`
Set of required packages. A warning is signaled by the constructor if at least one of the packages is not installed, but loaded (not attached) later on-demand via `requireNamespace()`.

`label (character(1))`
Label for the new instance.

`man (character(1))`
String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

Method `format()`**:** Helper for print outputs.

Usage:

```
Learner$format(...)
```

Arguments:

`...` (ignored).

Method `print()`**:** Printer.

Usage:

```
Learner$print(...)
```

Arguments:

`...` (ignored).

Method `help()`**:** Opens the corresponding help page referenced by field `$man`.

Usage:

```
Learner$help()
```

Method `train()`**:** Train the learner on a set of observations of the provided `task`. Mutates the learner by reference, i.e. stores the model alongside other information in field `$state`.

Usage:

```
Learner$train(task, row_ids = NULL)
```

Arguments:

`task` (Task).

`row_ids (integer())`
: Vector of training indices as subset of `task$row_ids`. For a simple split into training and test set, see `partition()`.

Returns: Returns the object itself, but modified **by reference**. You need to explicitly `$clone()` the object beforehand if you want to keeps the object in its previous state.

Examples:

```
task    = tsk("penguins")
learner = lrn("classif.rpart")
learner$train(task)
```

Method `predict()`**:** Uses the fitted model stored in `$state` to generate predictions for a set of observations from the provided `task`. This method requires that the learner has been previously trained using `$train()`.

Usage:

```
Learner$predict(task, row_ids = NULL)
```

Arguments:

`task` (Task)
: The task containing the observations to predict on. Must be compatible with the learner's task type and feature types. Unlike `$predict_newdata()`, no type conversion is done.

`row_ids (integer())`
: Vector of row indices from `task$row_ids` to predict on. If `NULL` (default), predictions are made for all rows in the task. For a simple train-test split, see `partition()`.

Returns: Prediction object containing the predictions for the specified observations.

Examples:

```
task = tsk("penguins")
learner = lrn("classif.rpart")$train(task)
learner$predict(task)
```

Method `predict_newdata()`**:** Uses the model fitted during `$train()` to create a new Prediction based on the new data in `newdata`. Object `task` is the task used during `$train()` and required for conversion of `newdata`. If the learner's `$train()` method has been called, there is a (size reduced) version of the training task stored in the learner. If the learner has been fitted via `resample()` or `benchmark()`, you need to pass the corresponding task stored in the ResampleResult or BenchmarkResult, respectively. Further, `auto_convert` is used for type-conversions to ensure compatability of features between `$train()` and `$predict()`.

If the stored training task has a `weights_measure` column, *and* if `newdata` contains a column with the same name, that column must be numeric with no missing values and is used as measure weights column. Otherwise, no measure weights are used.

Usage:

```
Learner$predict_newdata(newdata, task = NULL)
```

Arguments:

`newdata` (any object supported by `as_data_backend()`)
: New data to predict on. All data formats convertible by `as_data_backend()` are supported, e.g. `data.frame()` or DataBackend. If a DataBackend is provided as `newdata`, the row ids are preserved, otherwise they are set to to the sequence `1:nrow(newdata)`.

`task` (Task).

Returns: Prediction.

Examples:

```
task = tsk("penguins")
learner = lrn("classif.rpart")$train(task)
learner$predict_newdata(task$data(rows = 1:5))
```

Method `reset()`**:** Reset the learner, i.e. un-train by resetting the `state`.

Usage:

```
Learner$reset()
```

Returns: Returns the object itself, but modified **by reference**. You need to explicitly `$clone()` the object beforehand if you want to keeps the object in its previous state.

Examples:

```
task = tsk("penguins")
learner = lrn("classif.rpart")$train(task)
learner$reset()
```

Method `base_learner()`**:** Extracts the base learner from nested learner objects like `GraphLearner` in **mlr3pipelines** or `AutoTuner` in **mlr3tuning**. Returns the Learner itself for regular learners.

Usage:

```
Learner$base_learner(recursive = Inf)
```

Arguments:

`recursive (integer(1))`
: Depth of recursion for multiple nested objects.

Returns: Learner

Method `encapsulate()`**:** Sets the encapsulation method and fallback learner for the train and predict steps. There are currently four different methods implemented:

- `"none"`: Just runs the learner in the current session and measures the elapsed time. Does not keep a log, output is printed directly to the console. Works well together with `traceback()`.
- `"try"`: Similar to `"none"`, but catches error. Output is printed to the console and not logged.
- `"evaluate"`: Uses the package **evaluate** to call the learner, measure time and do the logging.
- `"callr"`: Uses the package **callr** to call the learner, measure time and do the logging. This encapsulation spawns a separate R session in which the learner is called. While this comes with a considerable overhead, it also guards your session from being teared down by segfaults.
- `"mirai"`: Uses the package **mirai** to call the learner, measure time and do the logging. This encapsulation calls the function in a `mirai` on a `daemon`. The `daemon` can be pre-started via `daemons(1, .compute = "mlr3_encapsulation")`, otherwise a new R session will be created for each encapsulated call. If a `deamon` is already running with compute profile `"mlr3_encapsulation"`, it will be used to executed all calls. Using `mirai"` is similarly safe as `callr` but much faster if several learners are encapsulated one after the other on the same daemon.

The fallback learner is fitted to create valid predictions in case that either the model fitting or the prediction of the original learner fails. If the training step or the predict step of the original learner fails, the fallback is used to make the predictions. If the original learner only partially fails during predict step (usually in the form of missing to predict some observations or producing some `NA` predictions), these missing predictions are imputed by the fallback. Note that the fallback is always trained, as we do not know in advance whether prediction will fail. If the training step fails, the `$model` field of the original learner is `NULL`. The results are reproducible across the different encapsulation methods.

Note that for errors of class `Mlr3ErrorConfig`, the function always errs and no fallback learner is trained.

Also see the section on error handling the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter10/advanced_technical_aspects_of_mlr3.html#sec-error-handling

Usage:

```
Learner$encapsulate(method, fallback = NULL, when = NULL)
```

Arguments:

`method character(1)`
: One of `"none"`, `"try"`, `"evaluate"` or `"callr"`. See the description for details.

`fallback` Learner
: The fallback learner for failed predictions.

`when (function(cond, stage))`
: Function that takes in the condition (`cond`) and the stage (`"train"` or `"predict"`) and returns `logical(1)` indicating whether to run the fallback learner.
If `NULL` (default), the fallback is always used, except for errors of class `Mlr3ErrorConfig`.

Returns: `self` (invisibly).

Examples:

```
learner = lrn("classif.rpart")
fallback = lrn("classif.featureless")
learner$encapsulate("try", fallback = fallback)
```

Method `configure()`**:** Sets parameter values and fields of the learner. All arguments whose names match the name of a parameter of the paradox::ParamSet are set as parameters. All remaining arguments are assumed to be regular fields.

Usage:

```
Learner$configure(..., .values = list())
```

Arguments:

`...` (named any)
Named arguments to set parameter values and fields.

`.values` (named any)
Named list of parameter values and fields.

Examples:

```
learner = lrn("classif.rpart")
learner$configure(minsplit = 3, parallel_predict = FALSE)
learner$configure(.values = list(cp = 0.005))
```

Method `selected_features()`**:** Returns the features selected by the model. The field `selected_features_impute` controls the behavior if the learner does not support feature selection. If set to `"error"`, an error is thrown, otherwise all features are returned.

Usage:

```
Learner$selected_features()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-learners
- Package **mlr3learners** for a solid collection of essential learners.
- Package mlr3extralearners for more learners.
- Dictionary of Learners: mlr_learners
- `as.data.table(mlr_learners)` for a table of available Learners in the running session (depending on the loaded packages).
- **mlr3pipelines** to combine learners with pre- and postprocessing steps.
- Package **mlr3viz** for some generic visualizations.
- Extension packages for additional task types:

  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.
- **mlr3tuning** for tuning of hyperparameters, **mlr3tuningspaces** for established default tuning spaces.

Other Learner: `LearnerClassif`, `LearnerRegr`, `mlr_learners`, `mlr_learners_classif.debug`, `mlr_learners_classif.featureless`, `mlr_learners_classif.rpart`, `mlr_learners_regr.debug`, `mlr_learners_regr.featureless`, `mlr_learners_regr.rpart`

Examples

```
## ------------------------------------------------
## Method `Learner$train`
## ------------------------------------------------

task   = tsk("penguins")
learner = lrn("classif.rpart")
learner$train(task)

## ------------------------------------------------
## Method `Learner$predict`
## ------------------------------------------------

task = tsk("penguins")
learner = lrn("classif.rpart")$train(task)
learner$predict(task)

## ------------------------------------------------
## Method `Learner$predict_newdata`
## ------------------------------------------------

task = tsk("penguins")
learner = lrn("classif.rpart")$train(task)
learner$predict_newdata(task$data(rows = 1:5))

## ------------------------------------------------
## Method `Learner$reset`
## ------------------------------------------------

task = tsk("penguins")
learner = lrn("classif.rpart")$train(task)
learner$reset()

## ------------------------------------------------
## Method `Learner$encapsulate`
## ------------------------------------------------

learner = lrn("classif.rpart")
fallback = lrn("classif.featureless")
learner$encapsulate("try", fallback = fallback)

## ------------------------------------------------
```

```
## Method `Learner$configure`
## ------------------------------------------------

learner = lrn("classif.rpart")
learner$configure(minsplit = 3, parallel_predict = FALSE)
learner$configure(.values = list(cp = 0.005))
```

`LearnerClassif` *Classification Learner*

Description

This Learner specializes Learner for classification problems:

- `task_type` is set to `"classif"`.
- Creates Predictions of class PredictionClassif.
- Possible values for `predict_types` are:
  - `"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.

Predefined learners can be found in the dictionary mlr_learners. Essential classification learners can be found in this dictionary after loading **mlr3learners**. Additional learners are implement in the Github package `https://github.com/mlr-org/mlr3extralearners`.

Super class

`mlr3::Learner` -> `LearnerClassif`

Methods

Public methods:

- `LearnerClassif$new()`
- `LearnerClassif$predict_newdata_fast()`
- `LearnerClassif$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
LearnerClassif$new(
  id,
  param_set = ps(),
  predict_types = "response",
  feature_types = character(),
  properties = character(),
  packages = character(),
  label = NA_character_,
  man = NA_character_
)
```

Arguments:

`id (character(1))`
Identifier for the new instance.

`param_set (paradox::ParamSet)`
Set of hyperparameters.

`predict_types (character())`
Supported predict types. Must be a subset of `mlr_reflections$learner_predict_types`.

`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.
- `"offset"`: The learner can incorporate offset values to adjust predictions.
- `"importance"`: The learner supports extraction of importance scores, i.e. comes with an `$importance()` extractor function (see section on optional extractors in Learner).
- `"selected_features"`: The learner supports extraction of the set of selected features, i.e. comes with a `$selected_features()` extractor function (see section on optional extractors in Learner).
- `"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 in Learner).
- `"validation"`: The learner can use a validation task during training.
- `"internal_tuning"`: The learner is able to internally optimize hyperparameters (those are also tagged with `"internal_tuning"`).
- `"marshal"`: To save learners with this property, you need to call `$marshal()` first. If a learner is in a marshaled state, you call first need to call `$unmarshal()` to use its model, e.g. for prediction.
- `"hotstart_forward"`: The learner supports to hotstart a model forward.
- `"hotstart_backward"`: The learner supports hotstarting a model backward.
- ‘"featureless": The learner does not use features.

`packages (character())`
Set of required packages. A warning is signaled by the constructor if at least one of the packages is not installed, but loaded (not attached) later on-demand via `requireNamespace()`.

`label (character(1))`
Label for the new instance.

man (character(1))
String in the format [pkg]::[topic] pointing to a manual page for this object. The referenced help package can be opened via method $help().

Method predict_newdata_fast(): Predicts outcomes for new data in newdata using the model fitted during $train(). This method is faster than $predict_newdata() as it skips assertions, type conversions, encapsulation, and logging.

Unlike $predict_newdata(), this method does not return a Prediction object. Instead, it returns a list with either a "response" or "prob" element, depending on the prediction type.

Note that state$predict_time and state$log will remain empty after using this method. Some learners may not support this method and may fail when it is called. Prefer $predict_newdata() unless performance is critical.

If the model was trained via resample() or benchmark(), you must pass the associated task object stored in the corresponding ResampleResult or BenchmarkResult.

Usage:

```
LearnerClassif$predict_newdata_fast(newdata, task = NULL)
```

Arguments:

newdata data.table::data.table()
New data to predict on.

task (Task).

Returns: list() with elements "response" or "prob" depending on the predict type.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-learners
- Package **mlr3learners** for a solid collection of essential learners.
- Package mlr3extralearners for more learners.
- Dictionary of Learners: mlr_learners
- as.data.table(mlr_learners) for a table of available Learners in the running session (depending on the loaded packages).
- **mlr3pipelines** to combine learners with pre- and postprocessing steps.
- Package **mlr3viz** for some generic visualizations.
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

- **mlr3tuning** for tuning of hyperparameters, **mlr3tuningspaces** for established default tuning spaces.

Other Learner: Learner, LearnerRegr, mlr_learners, mlr_learners_classif.debug, mlr_learners_classif.featur
mlr_learners_classif.rpart, mlr_learners_regr.debug, mlr_learners_regr.featureless,
mlr_learners_regr.rpart

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("penguins")
lrn$train(task, 1:200)

# predict on new observations:
lrn$predict(task, 201:344)$confusion
```

`LearnerRegr` *Regression Learner*

Description

This Learner specializes Learner for regression problems:

- `task_type` is set to `"regr"`.
- Creates Predictions of class PredictionRegr.
- Possible values for `predict_types` are:
  - `"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.
  - `"distr"`: Probability distribution as `VectorDistribution` object (requires package `distr6`, available via repository `https://raphaels1.r-universe.dev`).
- `"quantiles"`: Predicts quantile estimates for each observation in the test set. Set `$quantiles` to specify the quantiles to predict and `$quantile_response` to specify the response quantile. See mlr3book section on quantile regression for more details.

Predefined learners can be found in the dictionary mlr_learners. Essential regression learners can be found in this dictionary after loading **mlr3learners**. Additional learners are implement in the Github package `https://github.com/mlr-org/mlr3extralearners`.

Super class

`mlr3::Learner` -> `LearnerRegr`

Active bindings

`quantiles (numeric())`
: Numeric vector of probabilities to be used while predicting quantiles. Elements must be between 0 and 1, not missing and provided in ascending order. If only one quantile is provided, it is used as response. Otherwise, set `$quantile_response` to specify the response quantile.

`quantile_response (numeric(1))`
: The quantile to be used as response.

Methods

Public methods:

- `LearnerRegr$new()`
- `LearnerRegr$predict_newdata_fast()`
- `LearnerRegr$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
LearnerRegr$new(
  id,
  task_type = "regr",
  param_set = ps(),
  predict_types = "response",
  feature_types = character(),
  properties = character(),
  packages = character(),
  label = NA_character_,
  man = NA_character_
)
```

Arguments:

`id (character(1))`
: Identifier for the new instance.

`task_type (character(1))`
: Type of task, e.g. `"regr"` or `"classif"`. Must be an element of mlr_reflections$task_types$type.

`param_set (paradox::ParamSet)`
: Set of hyperparameters.

`predict_types (character())`
: Supported predict types. Must be a subset of `mlr_reflections$learner_predict_types`.

`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.
- "offset": The learner can incorporate offset values to adjust predictions.
- "importance": The learner supports extraction of importance scores, i.e. comes with an $importance() extractor function (see section on optional extractors in Learner).
- "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 in Learner).
- "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 in Learner).
- "validation": The learner can use a validation task during training.
- "internal_tuning": The learner is able to internally optimize hyperparameters (those are also tagged with "internal_tuning").
- "marshal": To save learners with this property, you need to call $marshal() first. If a learner is in a marshaled state, you call first need to call $unmarshal() to use its model, e.g. for prediction.
- "hotstart_forward": The learner supports to hotstart a model forward.
- "hotstart_backward": The learner supports hotstarting a model backward.
- ‘"featureless": The learner does not use features.

packages (character())
: Set of required packages. A warning is signaled by the constructor if at least one of the packages is not installed, but loaded (not attached) later on-demand via requireNamespace().

label (character(1))
: Label for the new instance.

man (character(1))
: String in the format [pkg]::[topic] pointing to a manual page for this object. The referenced help package can be opened via method $help().

Method predict_newdata_fast()**:** Predicts outcomes for new data in newdata using the model fitted during $train(). This method is faster than $predict_newdata() as it skips assertions, type conversions, encapsulation, and logging.

Unlike $predict_newdata(), this method does not return a Prediction object. Instead, it returns a list with either a "response" or "prob" element, depending on the prediction type.

Note that state$predict_time and state$log will remain empty after using this method. Some learners may not support this method and may fail when it is called. Prefer $predict_newdata() unless performance is critical.

If the model was trained via resample() or benchmark(), you must pass the associated task object stored in the corresponding ResampleResult or BenchmarkResult.

Usage:

```
LearnerRegr$predict_newdata_fast(newdata, task = NULL)
```

Arguments:

newdata data.table::data.table()
: New data to predict on.

task (Task).

Returns: `list()` with elements `"response"`, `"se"` or `"quantiles"` depending on the predict type.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-learners
- Package **mlr3learners** for a solid collection of essential learners.
- Package mlr3extralearners for more learners.
- Dictionary of Learners: mlr_learners
- `as.data.table(mlr_learners)` for a table of available Learners in the running session (depending on the loaded packages).
- **mlr3pipelines** to combine learners with pre- and postprocessing steps.
- Package **mlr3viz** for some generic visualizations.
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.
- **mlr3tuning** for tuning of hyperparameters, **mlr3tuningspaces** for established default tuning spaces.

Other Learner: `Learner`, `LearnerClassif`, `mlr_learners`, `mlr_learners_classif.debug`, `mlr_learners_classif.fea`
`mlr_learners_classif.rpart`, `mlr_learners_regr.debug`, `mlr_learners_regr.featureless`,
`mlr_learners_regr.rpart`

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")
```

marshaling *(Un)marshal a Learner*

Description

Marshaling is the process of processing the model of a trained `Learner` so it an be successfully serialized and deserialized. The naming is inspired by the marshal package and we plan to fully migrate to this package once it is on CRAN. The current implementation should therfore be considered as a temporary solution and is likely to change in the future.

The central functions (and the only methods that are used by `mlr3` internally) are:

- the S3 generic `marshal_model(model, inplace, ...)`. Which takes in a model and returns it in marshaled form. This means, that the resulting object can be serialized and de-serialzed without loss of information. If a model is serializable anyway, nothing has to be implemented and the generic will fall back to the default implementation of `marshal_model`, which is to return the object as-is. Otherwise, the marshaled object should be a list with named elements `marshaled` and `packages`, where the former contains the marshaled object, and the latter the package that contains the packages required to unmarshal. Most importantly, this list should contain the package that contains the `unmarshal_model` method. The returned object should have the classes of the original object with the suffix `"_marshaled"` appended and the root class should be set to `"marshaled"`.
- the S3 generic `unmarshal_model(model, inplace ...)`. Which takes in the marshaled model and returns it in unmarshaled form. The generic takes care that the packages specified during `"marshal"` are loaded, and errs if they are not availabe. Calling this function on a marshaled model should reconstruct the original model, i.e. `unmarshal_model(marshal_model(x))` should return `x`. The default implementation of this generic returns `x` as-is.
- the function `is_marshaled_model(model)`. This (helper) function returns `TRUE` if the model inherits from class `"marshaled"` and `FALSE` otherwise. Note that it is not guarateed that `is_marshaled_model(marshal_model(x))` returns `TRUE`. This is because the default `marshal_model(x)` returns `x` as-is.

For both `marshal_model` and `unmarshal_model`, the `inplace` argument determines whether inplace marshaling should be performed. This is especially relevant in the context of references semantics. If `inplace` is `FALSE`, the original input should not be modified, otherwise this is allowed. Note that the input and output can still share references, even when `inplace` is `FALSE`.

Usage

```
learner_unmarshal(.learner, ...)

learner_marshal(.learner, ...)

learner_marshaled(.learner)

marshal_model(model, inplace = FALSE, ...)
```

```
unmarshal_model(model, inplace = FALSE, ...)

is_marshaled_model(model)
```

Arguments

`.learner`	`Learner` The learner.
`...`	(any) Additional parameters, currently unused.
`model`	(any) Model to marshal.
`inplace`	(`logical(1)`) Whether to marshal in-place.

Implementing Marshaling

In order to implement marshaling for a Learner, you need to overload the `marshal_model` and `unmarshal_model` methods for the class of the learner's model and tag the learner with the `"marshal"` property. To make marshaling accessible in an R6-manner, you should also add the public methods `$marshal()`, `$unmarshal()` and the active binding `$marshaled`. To make this as convenient as possible, the functions `learner_marshal(.learner, ...)`, `learner_unmarshal(.learner, ...)` and `learner_marshaled(.learner)` are provided and can be called from the public methods.

You can verify whether you have correctly implemented marshaling by using the internal test helper `expect_marshalable_learner(learner, task)`. This is also run by `expect_learner()` if a task is provided.

For a concrete example on how to implement marshaling, see `LearnerClassifDebug`.

`Measure` *Measure Class*

Description

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

Measures are classes tailored around two functions doing the work:

1. A function `$score()` which quantifies the performance on a Prediction object, so a set of predicted observation via a scalar number – usually an aggregate of losses on the contained observations, by comparing the truth and prediction columns in the prediction object.
2. A function `$aggregator()` which combines multiple performance scores returned by `$score()` obtained in different resampling iterations to a scalar performance value associated with the complete resampling – usually by averaging or summing.

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

Predefined measures are stored in the dictionary mlr_measures, e.g. `classif.auc` or `time_train`. Many of the measures in **mlr3** are implemented in **mlr3measures** as ordinary functions.

A guide on how to extend **mlr3** with custom measures can be found in the mlr3book.

Inheriting

For some measures (such as confidence intervals from `mlr3inferr`) it is necessary that a measure returns more than one value. In such cases it is necessary to overwrite the public methods `$aggregate()` and/or `$score()` to return a named `numeric()` where at least one of its names corresponds to the `id` of the measure itself.

Weights

Many measures support observation weights, indicated by their property `"weights"`. The weights are stored in the Task where the column role `weights_measure` needs to be assigned to a single numeric column. The weights are automatically used if found in the task, this can be disabled by setting the field `use_weights` to `"ignore"`. See the description of `use_weights` for more information.

If the measure is set-up to use weights but the task does not have a designated `weights_measure` column, an unweighted version is calculated instead. The weights do not necessarily need to sum up to 1, they are normalized by the measure if necessary.

Most measures are so-called decomposable loss functions where a point-wise loss is computed and then either mean-aggregated or summed over the test set. For measures that do mean-aggregation, weights are typically used to compute the weighted mean, which normalizes weights to sum to 1. Measures that use sum-aggregation do not normalize weights and instead multiply individual losses with the given weights. See the documentation of specific measures for more details.

Missing Values during Scoring

Many measurements cannot be calculated if the test set or predictions are unfortunate, for example because a denominator is 0. This typically occurs during (binary) classification if some entries of the confusion matrix are 0. For this reason, many measures which origin in **mlr3measures** allow to change the default missing value (`NaN`) via the field `na_value`.

If you encounter missing values in a compound object like a ResampleResult or BenchmarkResult during scoring or aggregating, simply removing iterations with missing values is statistically arguable (but technically possible by prividng a custom aggregation function which handles missing values, e.g. `function(x) mean(x, na.rm = TRUE)`). Instead, consider stratification on the target of the Task to work around missing values. Switching to micro averaging in the Resampling can also be a solution here.

Public fields

`id (character(1))`
: Identifier of the object. Used in tables, plot and text output.

`label (character(1))`
: Label for this object. Can be used in tables, plot and text output instead of the ID.

`task_type (character(1))`
: Task type, e.g. `"classif"` or `"regr"`.

  For a complete list of possible task types (depending on the loaded packages), see `mlr_reflections$task_types$type`

`param_set (paradox::ParamSet)`
: Set of hyperparameters.

`obs_loss` (`function()` | `NULL`) Function to calculate the observation-wise loss.

`trafo` (`list()` | `NULL`) `NULL` or a list with two elements:

- `trafo`: the transformation function applied after aggregating observation-wise losses (e.g. `sqrt` for RMSE)
- `deriv`: The derivative of the `trafo`.

`predict_type` (`character(1)`)
Required predict type of the Learner.

`check_prerequisites` (`character(1)`)
How to proceed if one of the following prerequisites is not met:

- wrong predict type (e.g., probabilities required, but only labels available).
- wrong predict set (e.g., learner predicted on training set, but predictions of test set required).
- task properties not satisfied (e.g., binary classification measure on multiclass task).

Possible values are `"ignore"` (just return `NaN`) and `"warn"` (default, raise a warning before returning `NaN`).

`task_properties` (`character()`)
Required properties of the Task.

`range` (`numeric(2)`)
Lower and upper bound of possible performance scores.

`minimize` (`logical(1)`)
If `TRUE`, good predictions correspond to small values of performance scores.

`packages` (`character(1)`)
Set of required packages. These packages are loaded, but not attached.

`man` (`character(1)`)
String in the format `[pkg]::[topic]` pointing to a manual page for this object. Defaults to `NA`, but can be set by child classes.

Active bindings

`predict_sets` (`character()`)
During `resample()`/`benchmark()`, a Learner can predict on multiple sets. Per default, a learner only predicts observations in the test set (`predict_sets == "test"`). To change this behavior, set `predict_sets` to a non-empty subset of `{"train", "test", "internal_valid"}`. The `"train"` predict set contains the train ids from the resampling. This means that if a learner does validation and sets `$validate` to a ratio (creating the validation data from the training data), the train predictions will include the predictions for the validation data. Each set yields a separate Prediction object. Those can be combined via getters in ResampleResult/BenchmarkResult, or Measures can be configured to operate on specific subsets of the calculated prediction sets.

`hash` (`character(1)`)
Hash (unique identifier) for this object. The hash is calculated based on the id, the parameter settings, predict sets and the `$score`, `$average`, `$aggregator`, `$obs_loss`, `$trafo` method. Measure can define additional fields to be included in the hash by setting the field `$.extra_hash`.

`properties` (`character()`)
Properties of this measure.

`average` (`character(1)`)
Method for aggregation:

- `"micro"`: All predictions from multiple resampling iterations are first combined into a single Prediction object. Next, the scoring function of the measure is applied on this combined object, yielding a single numeric score.
- `"macro"`: The scoring function is applied on the Prediction object of each resampling iterations, each yielding a single numeric score. Next, the scores are combined with the `aggregator` function to a single numerical score.
- `"macro_weighted"`: The scoring function is applied on the Prediction object of each resampling iterations, each yielding a single numeric score. Next, the scores are combined with the `aggregator` function to a single numerical score. The scores are weighted by the total sample weights (if present, and if `$use_weights` is set to `"use"`), or the number of samples in each resampling iteration.
- `"custom"`: The measure comes with a custom aggregation method which directly operates on a ResampleResult.

`aggregator` (`function()`)
Function to aggregate scores computed on different resampling iterations.

`use_weights` (`character(1)`)
How to handle weights. Settings are `"use"`, `"ignore"`, and `"error"`.

- `"use"`: Weights are used automatically if found in the task, as supported by the measure.
- `"ignore"`: Weights are ignored.
- `"error"`: throw an error if weights are present in the training `Task`.

For measures with the property `"weights"`, this is initialized as `"use"`. For measures with the property `"requires_no_prediction"`, this is initialized as `"ignore"`. For measures that have neither of the properties, this is initialized as `"error"`. The latter behavior is to avoid cases where a user erroneously assumes that a measure supports weights when it does not. For measures that do not support weights, `use_weights` needs to be set to `"ignore"` if tasks with weights should be handled (by dropping the weights).

Methods

Public methods:

- `Measure$new()`
- `Measure$format()`
- `Measure$print()`
- `Measure$help()`
- `Measure$score()`
- `Measure$aggregate()`
- `Measure$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

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

Usage:

```
Measure$new(
  id,
  task_type = NA,
  param_set = ps(),
  range = c(-Inf, Inf),
  minimize = NA,
  average = "macro",
  aggregator = NULL,
  obs_loss = NULL,
  properties = character(),
  predict_type = "response",
  predict_sets = "test",
  task_properties = character(),
  packages = character(),
  label = NA_character_,
  man = NA_character_,
  trafo = NULL
)
```

Arguments:

`id (character(1))`
: Identifier for the new instance.

`task_type (character(1))`
: Type of task, e.g. `"regr"` or `"classif"`. Must be an element of mlr_reflections$task_types$type.

`param_set (paradox::ParamSet)`
: Set of hyperparameters.

`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.

`average (character(1))`
: How to average multiple Predictions from a ResampleResult.
The default, `"macro"`, calculates the individual performances scores for each Prediction and then uses the function defined in `$aggregator` to average them to a single number.
`"macro_weighted"` is similar to `"macro"`, but uses weighted averages. Weights are taken from the `weights_measure` column of the resampled Task if present. Note that `"macro_weighted"` can differ from `"macro"` even if no weights are present or if `$use_weights` is set to `"ignore"`, since then aggregation is done using *uniform sample weights*, which result in non-uniform weights for Predictions if they contain different numbers of samples.
If set to `"micro"`, the individual Prediction objects are first combined into a single new Prediction object which is then used to assess the performance. The function in `$aggregator` is not used in this case.

`aggregator (function())`
: Function to aggregate over multiple iterations. The role of this function depends on the value of field `"average"`:

- "macro": A numeric vector of scores (one per iteration) is passed. The aggregate function defaults to mean() in this case.
- "micro": The aggregator function is not used. Instead, predictions from multiple iterations are first combined and then scored in one go.
- "custom": A ResampleResult is passed to the aggregate function.

obs_loss (function or NULL)
: The observation-wise loss function, e.g. zero-one for classification error.

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_model" (requires the trained Learner, including the fitted model),
  - "requires_train_set" (requires the training indices from the Resampling),
  - "na_score" (the measure is expected to occasionally return NA or NaN),
  - "weights" (support weighted scoring using sample weights from task, column role weights_measure), and
  - "primary_iters" (the measure explictly handles resamplings that only use a subset of their iterations for the point estimate)
  - "requires_no_prediction" (No prediction is required; This usually means that the measure extracts some information from the learner state.).

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 {"train", "test", "internal_valid"}. 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. A warning is signaled by the constructor if at least one of the packages is not installed, but loaded (not attached) later on-demand via requireNamespace().

label (character(1))
: Label for the new instance.

man (character(1))
: String in the format [pkg]::[topic] pointing to a manual page for this object. The referenced help package can be opened via method $help().

trafo (list() or NULL)
: An optional list with two elements, containing the transformation "fn" and its derivative "deriv". The transformation function is the function that is applied after aggregating the pointwise losses, i.e. this requires an $obs_loss to be present. An example is sqrt for RMSE.

Method format()**:** Helper for print outputs.

Usage:

```
Measure$format(...)
```

Arguments:

`...` (ignored).

Method `print()`**:** Printer.

Usage:

```
Measure$print(...)
```

Arguments:

`...` (ignored).

Method `help()`**:** Opens the corresponding help page referenced by field `$man`.

Usage:

```
Measure$help()
```

Method `score()`**:** 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"`, `"requires_model"` 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()`.

Usage:

```
Measure$score(prediction, task = NULL, learner = NULL, train_set = NULL)
```

Arguments:

`prediction` (Prediction | named list of Prediction).

`task` (Task).

`learner` (Learner).

`train_set` (`integer()`).

Returns: `numeric(1)`.

Examples:

```
task = tsk("penguins")
learner = lrn("classif.rpart")$train(task)
prediction = learner$predict(task)
msr("classif.ce")$score(prediction)
```

Method `aggregate()`**:** Aggregates multiple performance scores into a single score, e.g. by using the `aggregator` function of the measure.

Usage:

```
Measure$aggregate(rr)
```

Arguments:

`rr` ResampleResult.

Returns: `numeric(1)`.

Examples:

```
task = tsk("penguins")
learner = lrn("classif.rpart")
rr = resample(task, learner, rsmp("holdout"))
msr("classif.ce")$aggregate(rr)
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-eval
- Package **mlr3measures** for the scoring functions. Dictionary of Measures: mlr_measures `as.data.table(mlr_measures)` for a table of available Measures in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other Measure: `MeasureClassif`, `MeasureRegr`, `MeasureSimilarity`, `mlr_measures`, `mlr_measures_aic`, `mlr_measures_bic`, `mlr_measures_classif.costs`, `mlr_measures_debug_classif`, `mlr_measures_elapsed_time`, `mlr_measures_internal_valid_score`, `mlr_measures_oob_error`, `mlr_measures_regr.pinball`, `mlr_measures_regr.rqr`, `mlr_measures_regr.rsq`, `mlr_measures_selected_features`

Examples

```
## ------------------------------------------------
## Method `Measure$score`
## ------------------------------------------------

task = tsk("penguins")
learner = lrn("classif.rpart")$train(task)
prediction = learner$predict(task)
msr("classif.ce")$score(prediction)

## ------------------------------------------------
## Method `Measure$aggregate`
## ------------------------------------------------

task = tsk("penguins")
learner = lrn("classif.rpart")
rr = resample(task, learner, rsmp("holdout"))
msr("classif.ce")$aggregate(rr)
```

`MeasureClassif` *Classification Measure*

Description

This measure specializes Measure for classification problems:

- `task_type` is set to `"classif"`.
- Possible values for `predict_type` are `"response"` and `"prob"`.

Predefined measures can be found in the dictionary mlr_measures. The default measure for classification is `classif.ce`.

Super class

`mlr3::Measure` -> `MeasureClassif`

Methods

Public methods:

- `MeasureClassif$new()`
- `MeasureClassif$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
MeasureClassif$new(
  id,
  param_set = ps(),
  range,
  minimize = NA,
  average = "macro",
  aggregator = NULL,
  properties = character(),
  predict_type = "response",
  predict_sets = "test",
  task_properties = character(),
  packages = character(),
  label = NA_character_,
  man = NA_character_
)
```

Arguments:

`id` (`character(1)`)
Identifier for the new instance.

`param_set` (paradox::ParamSet)
Set of hyperparameters.

`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.

`average (character(1))`
How to average multiple Predictions from a ResampleResult.
The default, `"macro"`, calculates the individual performances scores for each Prediction and then uses the function defined in `$aggregator` to average them to a single number.
`"macro_weighted"` is similar to `"macro"`, but uses weighted averages. Weights are taken from the `weights_measure` column of the resampled Task if present. Note that `"macro_weighted"` can differ from `"macro"` even if no weights are present or if `$use_weights` is set to `"ignore"`, since then aggregation is done using *uniform sample weights*, which result in non-uniform weights for Predictions if they contain different numbers of samples.
If set to `"micro"`, the individual Prediction objects are first combined into a single new Prediction object which is then used to assess the performance. The function in `$aggregator` is not used in this case.

`aggregator (function())`
Function to aggregate over multiple iterations. The role of this function depends on the value of field `"average"`:

- `"macro"`: A numeric vector of scores (one per iteration) is passed. The aggregate function defaults to `mean()` in this case.
- `"micro"`: The `aggregator` function is not used. Instead, predictions from multiple iterations are first combined and then scored in one go.
- `"custom"`: A ResampleResult is passed to the aggregate function.

`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_model"` (requires the trained Learner, including the fitted model),
- `"requires_train_set"` (requires the training indices from the Resampling),
- `"na_score"` (the measure is expected to occasionally return `NA` or `NaN`),
- `"weights"` (support weighted scoring using sample weights from task, column role `weights_measure`), and
- `"primary_iters"` (the measure explictly handles resamplings that only use a subset of their iterations for the point estimate)
- `"requires_no_prediction"` (No prediction is required; This usually means that the measure extracts some information from the learner state.).

`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 `{"train", "test", "internal_valid"}`.

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. A warning is signaled by the constructor if at least one of the packages is not installed, but loaded (not attached) later on-demand via requireNamespace().

label (character(1))
: Label for the new instance.

man (character(1))
: String in the format [pkg]::[topic] pointing to a manual page for this object. The referenced help package can be opened via method $help().

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-eval
- Package **mlr3measures** for the scoring functions. Dictionary of Measures: mlr_measures as.data.table(mlr_measures) for a table of available Measures in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other Measure: Measure, MeasureRegr, MeasureSimilarity, mlr_measures, mlr_measures_aic, mlr_measures_bic, mlr_measures_classif.costs, mlr_measures_debug_classif, mlr_measures_elapsed_time, mlr_measures_internal_valid_score, mlr_measures_oob_error, mlr_measures_regr.pinball, mlr_measures_regr.rqr, mlr_measures_regr.rsq, mlr_measures_selected_features

---

MeasureRegr *Regression Measure*

Description

This measure specializes Measure for regression problems:

- task_type is set to "regr".
- Possible values for predict_type are "response", "se" and "distr".

Predefined measures can be found in the dictionary mlr_measures. The default measure for regression is regr.mse.

Super class

`mlr3::Measure` -> `MeasureRegr`

Methods

Public methods:

- `MeasureRegr$new()`
- `MeasureRegr$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
MeasureRegr$new(
  id,
  param_set = ps(),
  range,
  minimize = NA,
  average = "macro",
  aggregator = NULL,
  properties = character(),
  predict_type = "response",
  predict_sets = "test",
  task_properties = character(),
  packages = character(),
  label = NA_character_,
  man = NA_character_
)
```

Arguments:

`id (character(1))`
: Identifier for the new instance.

`param_set (paradox::ParamSet)`
: Set of hyperparameters.

`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.

`average (character(1))`
: How to average multiple Predictions from a ResampleResult.
The default, `"macro"`, calculates the individual performances scores for each Prediction and then uses the function defined in `$aggregator` to average them to a single number.
`"macro_weighted"` is similar to `"macro"`, but uses weighted averages. Weights are taken from the `weights_measure` column of the resampled Task if present. Note that `"macro_weighted"` can differ from `"macro"` even if no weights are present or if `$use_weights` is set to `"ignore"`, since then aggregation is done using *uniform sample weights*, which result in non-uniform weights for Predictions if they contain different numbers of samples.

If set to "micro", the individual Prediction objects are first combined into a single new Prediction object which is then used to assess the performance. The function in $aggregator is not used in this case.

aggregator (function())
Function to aggregate over multiple iterations. The role of this function depends on the value of field "average":
- "macro": A numeric vector of scores (one per iteration) is passed. The aggregate function defaults to mean() in this case.
- "micro": The aggregator function is not used. Instead, predictions from multiple iterations are first combined and then scored in one go.
- "custom": A ResampleResult is passed to the aggregate function.

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_model" (requires the trained Learner, including the fitted model),
- "requires_train_set" (requires the training indices from the Resampling),
- "na_score" (the measure is expected to occasionally return NA or NaN),
- "weights" (support weighted scoring using sample weights from task, column role weights_measure), and
- "primary_iters" (the measure explictly handles resamplings that only use a subset of their iterations for the point estimate)
- "requires_no_prediction" (No prediction is required; This usually means that the measure extracts some information from the learner state.).

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 {"train", "test", "internal_valid"}. 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. A warning is signaled by the constructor if at least one of the packages is not installed, but loaded (not attached) later on-demand via requireNamespace().

label (character(1))
Label for the new instance.

man (character(1))
String in the format [pkg]::[topic] pointing to a manual page for this object. The referenced help package can be opened via method $help().

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-eval
- Package **mlr3measures** for the scoring functions. Dictionary of Measures: mlr_measures `as.data.table(mlr_measures)` for a table of available Measures in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other Measure: Measure, MeasureClassif, MeasureSimilarity, mlr_measures, mlr_measures_aic, mlr_measures_bic, mlr_measures_classif.costs, mlr_measures_debug_classif, mlr_measures_elapsed_time, mlr_measures_internal_valid_score, mlr_measures_oob_error, mlr_measures_regr.pinball, mlr_measures_regr.rqr, mlr_measures_regr.rsq, mlr_measures_selected_features

---

MeasureSimilarity *Similarity Measure*

Description

This measure specializes Measure for measures quantifying the similarity of sets of selected features. To calculate similarity measures, the Learner must have the property `"selected_features"`.

- `task_type` is set to `NA_character_`.
- `average` is set to `"custom"`.

Predefined measures can be found in the dictionary mlr_measures, prefixed with `"sim."`.

Super class

`mlr3::Measure` -> `MeasureSimilarity`

Methods

Public methods:

- `MeasureSimilarity$new()`
- `MeasureSimilarity$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
MeasureSimilarity$new(
  id,
  param_set = ps(),
  range,
  minimize = NA,
  average = "macro",
  aggregator = NULL,
  properties = character(),
  predict_type = NA_character_,
  task_properties = character(),
  packages = character(),
  label = NA_character_,
  man = NA_character_
)
```

Arguments:

`id (character(1))`
: Identifier for the new instance.

`param_set (paradox::ParamSet)`
: Set of hyperparameters.

`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.

`average (character(1))`
: How to average multiple Predictions from a ResampleResult.
The default, `"macro"`, calculates the individual performances scores for each Prediction and then uses the function defined in `$aggregator` to average them to a single number.
`"macro_weighted"` is similar to `"macro"`, but uses weighted averages. Weights are taken from the `weights_measure` column of the resampled Task if present. Note that `"macro_weighted"` can differ from `"macro"` even if no weights are present or if `$use_weights` is set to `"ignore"`, since then aggregation is done using *uniform sample weights*, which result in non-uniform weights for Predictions if they contain different numbers of samples.
If set to `"micro"`, the individual Prediction objects are first combined into a single new Prediction object which is then used to assess the performance. The function in `$aggregator` is not used in this case.

`aggregator (function())`
: Function to aggregate over multiple iterations. The role of this function depends on the value of field `"average"`:
  - `"macro"`: A numeric vector of scores (one per iteration) is passed. The aggregate function defaults to `mean()` in this case.
  - `"micro"`: The `aggregator` function is not used. Instead, predictions from multiple iterations are first combined and then scored in one go.
  - `"custom"`: A ResampleResult is passed to the aggregate function.

`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_model" (requires the trained Learner, including the fitted model),
- "requires_train_set" (requires the training indices from the Resampling),
- "na_score" (the measure is expected to occasionally return NA or NaN),
- "weights" (support weighted scoring using sample weights from task, column role weights_measure), and
- "primary_iters" (the measure explictly handles resamplings that only use a subset of their iterations for the point estimate)
- "requires_no_prediction" (No prediction is required; This usually means that the measure extracts some information from the learner state.).

`predict_type (character(1))`
Required predict type of the Learner. Possible values are stored in mlr_reflections$learner_predict_types.

`task_properties (character())`
Required task properties, see Task.

`packages (character())`
Set of required packages. A warning is signaled by the constructor if at least one of the packages is not installed, but loaded (not attached) later on-demand via `requireNamespace()`.

`label (character(1))`
Label for the new instance.

`man (character(1))`
String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-eval
- Package **mlr3measures** for the scoring functions. Dictionary of Measures: mlr_measures `as.data.table(mlr_measures)` for a table of available Measures in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other Measure: Measure, MeasureClassif, MeasureRegr, mlr_measures, mlr_measures_aic, mlr_measures_bic, mlr_measures_classif.costs, mlr_measures_debug_classif, mlr_measures_elapsed_time, mlr_measures_internal_valid_score, mlr_measures_oob_error, mlr_measures_regr.pinball, mlr_measures_regr.rqr, mlr_measures_regr.rsq, mlr_measures_selected_features

Examples

```
task = tsk("penguins")
learners = list(
  lrn("classif.rpart", maxdepth = 1, id = "r1"),
  lrn("classif.rpart", maxdepth = 2, id = "r2")
)
resampling = rsmp("cv", folds = 3)
grid = benchmark_grid(task, learners, resampling)
bmr = benchmark(grid, store_models = TRUE)
bmr$aggregate(msrs(c("classif.ce", "sim.jaccard")))
```

---

mlr3.holdout_task *Callback Holdout Task*

Description

This CallbackResample predicts on an additional holdout task after training.

Arguments

task	(Task) The holdout task.

Examples

```
task = tsk("pima")
task_holdout = task$clone()
learner = lrn("classif.rpart")
resampling = rsmp("cv", folds = 3)
splits = partition(task, 0.7)

task$filter(splits$train)
task_holdout$filter(splits$test)

callback = clbk("mlr3.holdout_task", task = task_holdout)

rr = resample(task, learner, resampling = resampling, callbacks = callback)

rr$data_extra
```

mlr3.model_extractor *Model Extractor Callback*

Description

This CallbackResample extracts information from the model after training with a user-defined function. This way information can be extracted from the model without saving the model (`store_models = FALSE`). The `fun` must be a function that takes a learner as input and returns the extracted information as named list (see example). The callback is very helpful to call `$selected_features()`, `$importance()`, `$oob_error()` on the learner.

Arguments

`fun`	`(function(learner))` Function to extract information from the learner. The function must have the argument `learner`. The function must return a named list.

Examples

```
task = tsk("pima")
learner = lrn("classif.rpart")
resampling = rsmp("cv", folds = 3)

# define function to extract selected features
selected_features = function(learner) list(selected_features = learner$selected_features())

# create callback
callback = clbk("mlr3.model_extractor", fun = selected_features)

rr = resample(task, learner, resampling = resampling, store_models = FALSE, callbacks = callback)

rr$data_extra
```

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 an opinionated set of solid classification and regression learners, install and load the **mlr3learners** package. More learners are connected via https://github.com/mlr-org/mlr3extralearners.

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

Format

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

Methods

See mlr3misc::Dictionary.

S3 methods

- `as.data.table(dict, ..., objects = FALSE)`
  mlr3misc::Dictionary -> `data.table::data.table()`
  Returns a `data.table::data.table()` with fields "key", "label", "task_type", "feature_types", "packages", "properties", and "predict_types" as columns. If `objects` is set to `TRUE`, the constructed objects are returned in the list column named `object`.

See Also

Sugar functions: `lrn()`, `lrns()`

Extension Packages: **mlr3learners**

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

Other Learner: `Learner`, `LearnerClassif`, `LearnerRegr`, `mlr_learners_classif.debug`, `mlr_learners_classif.featu`
`mlr_learners_classif.rpart`, `mlr_learners_regr.debug`, `mlr_learners_regr.featureless`,
`mlr_learners_regr.rpart`

Examples

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

---

mlr_learners_classif.debug

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:

error_predict: Probability to raise an exception during predict.

error_train: Probability to raises an exception during train.

message_predict: Probability to output a message during predict.

message_train: Probability to output a message during train.

predict_missing: Ratio of predictions which will be NA.

predict_missing_type: To to encode missingness. "na" will insert NA values, "omit" will just return fewer predictions than requested.

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

segfault_predict: Probability to provokes a segfault during predict.

segfault_train: Probability to provokes a segfault during train.

sleep_train: Function returning a single number determining how many seconds to sleep during `$train()`.

sleep_predict: Function returning a single number determining how many seconds to sleep during `$predict()`.

threads: Number of threads to use. Has no effect.

warning_predict: Probability to signal a warning during predict.

warning_train: Probability to signal a warning during train.

x: Numeric tuning parameter. Has no effect.

iter: Integer parameter for testing hotstarting.

count_marshaling: If `TRUE`, `marshal_model` will increase the `marshal_count` by 1 each time it is called. The default is `FALSE`.

check_pid: If `TRUE`, the `$predict()` function will throw an error if the model was not unmarshaled in the same session that is used for prediction.)

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

Dictionary

This Learner can be instantiated via the dictionary mlr_learners or with the associated sugar function `lrn()`:

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

Meta Information

- Task type: "classif"
- Predict Types: "response", "prob"
- Feature Types: "logical", "integer", "numeric", "character", "factor", "ordered"
- Required Packages: **mlr3**

Parameters

Id	Type	Default	Levels	Range
error_predict	numeric	0		$[0, 1]$
error_train	numeric	0		$[0, 1]$
message_predict	numeric	0		$[0, 1]$
message_train	numeric	0		$[0, 1]$

predict_missing	numeric	0		$[0, 1]$
predict_missing_type	character	na	na, omit	-
save_tasks	logical	FALSE	TRUE, FALSE	-
segfault_predict	numeric	0		$[0, 1]$
segfault_train	numeric	0		$[0, 1]$
sleep_train	untyped	-		-
sleep_predict	untyped	-		-
threads	integer	-		$[1, \infty)$
warning_predict	numeric	0		$[0, 1]$
warning_train	numeric	0		$[0, 1]$
x	numeric	-		$[0, 1]$
iter	integer	1		$[1, \infty)$
early_stopping	logical	FALSE	TRUE, FALSE	-
count_marshaling	logical	FALSE	TRUE, FALSE	-
check_pid	logical	TRUE	TRUE, FALSE	-
config_error	logical	FALSE	TRUE, FALSE	-

Super classes

`mlr3::Learner` -> `mlr3::LearnerClassif` -> `LearnerClassifDebug`

Active bindings

`marshaled (logical(1))`
: Whether the learner has been marshaled.

`internal_valid_scores` Retrieves the internal validation scores as a named `list()`. Returns `NULL` if learner is not trained yet.

`internal_tuned_values` Retrieves the internally tuned values as a named `list()`. Returns `NULL` if learner is not trained yet.

`validate` How to construct the internal validation data. This parameter can be either `NULL`, a ratio in $(0, 1)$, `"test"`, or `"predefined"`.

Methods

Public methods:

- `LearnerClassifDebug$new()`
- `LearnerClassifDebug$marshal()`
- `LearnerClassifDebug$unmarshal()`
- `LearnerClassifDebug$importance()`
- `LearnerClassifDebug$selected_features()`
- `LearnerClassifDebug$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
LearnerClassifDebug$new()
```

Method `marshal()`**:** Marshal the learner's model.

Usage:

```
LearnerClassifDebug$marshal(...)
```

Arguments:

`...` (any)
Additional arguments passed to `marshal_model()`.

Method `unmarshal()`**:** Unmarshal the learner's model.

Usage:

```
LearnerClassifDebug$unmarshal(...)
```

Arguments:

`...` (any)
Additional arguments passed to `unmarshal_model()`.

Method `importance()`**:** Returns 0 for each feature seen in training.

Usage:

```
LearnerClassifDebug$importance()
```

Returns: Named `numeric()`.

Method `selected_features()`**:** Always returns character(0).

Usage:

```
LearnerClassifDebug$selected_features()
```

Returns: `character()`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: `https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-learners`
- Package **mlr3learners** for a solid collection of essential learners.
- Package mlr3extralearners for more learners.
- Dictionary of Learners: mlr_learners
- `as.data.table(mlr_learners)` for a table of available Learners in the running session (depending on the loaded packages).
- **mlr3pipelines** to combine learners with pre- and postprocessing steps.
- Package **mlr3viz** for some generic visualizations.

- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.
- **mlr3tuning** for tuning of hyperparameters, **mlr3tuningspaces** for established default tuning spaces.

Other Learner: Learner, LearnerClassif, LearnerRegr, mlr_learners, mlr_learners_classif.featureless, mlr_learners_classif.rpart, mlr_learners_regr.debug, mlr_learners_regr.featureless, mlr_learners_regr.rpart

Examples

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

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

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

---

mlr_learners_classif.featureless

Featureless Classification Learner

Description

A simple LearnerClassif which only analyzes 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. Probabilities correspond to the relative frequency of the class labels in the training set. For weighted data, the label(s) with the highest weighted frequency are selected.

sample: Randomly predict a label uniformly. Probabilities correspond to a uniform distribution of class labels, i.e. 1 divided by the number of classes. Weights are ignored, if present.

weighted.sample: Randomly predict a label, with probability estimated from the training distribution. For consistency, probabilities are 1 for the sampled label and 0 for all other labels. For weighted data, sample weights are used to weight the class labels.

Dictionary

This Learner can be instantiated via the dictionary mlr_learners or with the associated sugar function `lrn()`:

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

Meta Information

- Task type: "classif"
- Predict Types: "response", "prob"
- Feature Types: "logical", "integer", "numeric", "character", "factor", "ordered", "POSIXct", "Date"
- Required Packages: **mlr3**

Parameters

Id	Type	Default	Levels
method	character	mode	mode, sample, weighted.sample

Super classes

`mlr3::Learner` -> `mlr3::LearnerClassif` -> `LearnerClassifFeatureless`

Methods

Public methods:

- `LearnerClassifFeatureless$new()`
- `LearnerClassifFeatureless$importance()`
- `LearnerClassifFeatureless$selected_features()`
- `LearnerClassifFeatureless$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
LearnerClassifFeatureless$new()
```

Method `importance()`**:** All features have a score of `0` for this learner.

Usage:

```
LearnerClassifFeatureless$importance()
```

Returns: Named `numeric()`.

Method `selected_features()`**:** Selected features are always the empty set for this learner.

Usage:

```
LearnerClassifFeatureless$selected_features()
```

Returns: `character(0)`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-learners
- Package **mlr3learners** for a solid collection of essential learners.
- Package mlr3extralearners for more learners.
- Dictionary of Learners: mlr_learners
- `as.data.table(mlr_learners)` for a table of available Learners in the running session (depending on the loaded packages).
- **mlr3pipelines** to combine learners with pre- and postprocessing steps.
- Package **mlr3viz** for some generic visualizations.
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.
- **mlr3tuning** for tuning of hyperparameters, **mlr3tuningspaces** for established default tuning spaces.

Other Learner: `Learner`, `LearnerClassif`, `LearnerRegr`, `mlr_learners`, `mlr_learners_classif.debug`, `mlr_learners_classif.rpart`, `mlr_learners_regr.debug`, `mlr_learners_regr.featureless`, `mlr_learners_regr.rpart`

`mlr_learners_classif.rpart`

Classification Tree Learner

Description

A LearnerClassif for a classification tree implemented in `rpart::rpart()` in package **rpart**.

Initial parameter values

- Parameter `xval` is initialized to 0 in order to save some computation time.

Custom mlr3 parameters

- Parameter `model` has been renamed to `keep_model`.

Dictionary

This Learner can be instantiated via the dictionary mlr_learners or with the associated sugar function `lrn()`:

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

Meta Information

- Task type: "classif"
- Predict Types: "response", "prob"
- Feature Types: "logical", "integer", "numeric", "factor", "ordered"
- Required Packages: **mlr3**, **rpart**

Parameters

Id	Type	Default	Levels	Range
cp	numeric	0.01		$[0, 1]$
keep_model	logical	FALSE	TRUE, FALSE	-
maxcompete	integer	4		$[0, \infty)$
maxdepth	integer	30		$[1, 30]$
maxsurrogate	integer	5		$[0, \infty)$
minbucket	integer	-		$[1, \infty)$
minsplit	integer	20		$[1, \infty)$
surrogatestyle	integer	0		$[0, 1]$
usesurrogate	integer	2		$[0, 2]$
xval	integer	10		$[0, \infty)$

Super classes

`mlr3::Learner` -> `mlr3::LearnerClassif` -> `LearnerClassifRpart`

Methods

Public methods:

- `LearnerClassifRpart$new()`
- `LearnerClassifRpart$importance()`
- `LearnerClassifRpart$selected_features()`
- `LearnerClassifRpart$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
LearnerClassifRpart$new()
```

Method `importance()`**:** The importance scores are extracted from the model slot `variable.importance`.

Usage:

```
LearnerClassifRpart$importance()
```

Returns: Named `numeric()`.

Method `selected_features()`**:** Selected features are extracted from the model slot `frame$var`.

Usage:

```
LearnerClassifRpart$selected_features()
```

Returns: `character()`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Breiman L, Friedman JH, Olshen RA, Stone CJ (1984). *Classification And Regression Trees*. Routledge. doi:10.1201/9781315139470.

See Also

- Chapter in the mlr3book: `https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-learners`
- Package **mlr3learners** for a solid collection of essential learners.
- Package mlr3extralearners for more learners.
- Dictionary of Learners: mlr_learners
- `as.data.table(mlr_learners)` for a table of available Learners in the running session (depending on the loaded packages).
- **mlr3pipelines** to combine learners with pre- and postprocessing steps.
- Package **mlr3viz** for some generic visualizations.
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.
- **mlr3tuning** for tuning of hyperparameters, **mlr3tuningspaces** for established default tuning spaces.

Other Learner: `Learner`, `LearnerClassif`, `LearnerRegr`, `mlr_learners`, `mlr_learners_classif.debug`, `mlr_learners_classif.featureless`, `mlr_learners_regr.debug`, `mlr_learners_regr.featureless`, `mlr_learners_regr.rpart`

`mlr_learners_regr.debug`

Regression Learner for Debugging

Description

A simple LearnerRegr used primarily in the unit tests and for debugging purposes. If no hyperparameter is set, it simply constantly predicts the mean value of the training data. The following hyperparameters trigger the following actions:

predict_missing: Ratio of predictions which will be NA.

predict_missing_type: To to encode missingness. "na" will insert NA values, "omit" will just return fewer predictions than requested.

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

threads: Number of threads to use. Has no effect.

x: Numeric tuning parameter. Has no effect.

Dictionary

This Learner can be instantiated via the dictionary mlr_learners or with the associated sugar function `lrn()`:

```
mlr_learners$get("regr.debug")
lrn("regr.debug")
```

Meta Information

- Task type: "regr"
- Predict Types: "response", "se", "quantiles"
- Feature Types: "logical", "integer", "numeric", "character", "factor", "ordered"
- Required Packages: **mlr3**, 'stats'

Parameters

Id	Type	Default	Levels	Range
error_predict	numeric	0		$[0, 1]$
error_train	numeric	0		$[0, 1]$
message_predict	numeric	0		$[0, 1]$
message_train	numeric	0		$[0, 1]$
predict_missing	numeric	0		$[0, 1]$
predict_missing_type	character	na	na, omit	-
save_tasks	logical	FALSE	TRUE, FALSE	-
segfault_predict	numeric	0		$[0, 1]$
segfault_train	numeric	0		$[0, 1]$

threads	integer	-	$[1, \infty)$
warning_predict	numeric	0	$[0, 1]$
warning_train	numeric	0	$[0, 1]$
x	numeric	-	$[0, 1]$

Super classes

`mlr3::Learner` -> `mlr3::LearnerRegr` -> `LearnerRegrDebug`

Methods

Public methods:

- `LearnerRegrDebug$new()`
- `LearnerRegrDebug$importance()`
- `LearnerRegrDebug$selected_features()`
- `LearnerRegrDebug$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
LearnerRegrDebug$new()
```

Method `importance()`**:** Returns 0 for each feature seen in training.

Usage:

```
LearnerRegrDebug$importance()
```

Returns: Named `numeric()`.

Method `selected_features()`**:** Always returns character(0).

Usage:

```
LearnerRegrDebug$selected_features()
```

Returns: `character()`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-learners
- Package **mlr3learners** for a solid collection of essential learners.
- Package mlr3extralearners for more learners.
- Dictionary of Learners: mlr_learners
- `as.data.table(mlr_learners)` for a table of available Learners in the running session (depending on the loaded packages).
- **mlr3pipelines** to combine learners with pre- and postprocessing steps.
- Package **mlr3viz** for some generic visualizations.
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.
- **mlr3tuning** for tuning of hyperparameters, **mlr3tuningspaces** for established default tuning spaces.

Other Learner: `Learner`, `LearnerClassif`, `LearnerRegr`, `mlr_learners`, `mlr_learners_classif.debug`, `mlr_learners_classif.featureless`, `mlr_learners_classif.rpart`, `mlr_learners_regr.featureless`, `mlr_learners_regr.rpart`

Examples

```
task = tsk("mtcars")
learner = lrn("regr.debug", save_tasks = TRUE)
learner$train(task, row_ids = 1:20)
prediction = learner$predict(task, row_ids = 21:32)

learner$model$task_train
learner$model$task_predict
```

`mlr_learners_regr.featureless`

Featureless Regression Learner

Description

A simple LearnerRegr which only analyzes 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 `mad()` are used instead of `mean()` and `sd()`, respectively.

For weighted data, the response is the weighted mean (weighted median for robust regression). The predicted standard error is the square root of the weighted variance estimator with bias correction based on effective degrees of freedom:

```
sd(y, weights) = sqrt(
  sum(weights * (y - weighted.mean(y, weights))^2) /
    (sum(weights) - sum(weights ^2) / sum(weights))
)
```

If `robust` is `TRUE`, the weighted median absolute deviation is used, adjusted by a factor of 1.4826 for consistency with `mad()`.

Dictionary

This Learner can be instantiated via the dictionary mlr_learners or with the associated sugar function `lrn()`:

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

Meta Information

- Task type: "regr"
- Predict Types: "response", "se", "quantiles"
- Feature Types: "logical", "integer", "numeric", "character", "factor", "ordered", "POSIXct", "Date"
- Required Packages: **mlr3**, 'stats'

Parameters

Id	Type	Default	Levels
robust	logical	TRUE	TRUE, FALSE

Super classes

`mlr3::Learner` -> `mlr3::LearnerRegr` -> `LearnerRegrFeatureless`

Methods

Public methods:

- `LearnerRegrFeatureless$new()`
- `LearnerRegrFeatureless$importance()`
- `LearnerRegrFeatureless$selected_features()`
- `LearnerRegrFeatureless$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
LearnerRegrFeatureless$new()
```

Method `importance()`**:** All features have a score of `0` for this learner.

Usage:

```
LearnerRegrFeatureless$importance()
```

Returns: Named `numeric()`.

Method `selected_features()`**:** Selected features are always the empty set for this learner.

Usage:

```
LearnerRegrFeatureless$selected_features()
```

Returns: `character(0)`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-learners
- Package **mlr3learners** for a solid collection of essential learners.
- Package mlr3extralearners for more learners.
- Dictionary of Learners: mlr_learners
- `as.data.table(mlr_learners)` for a table of available Learners in the running session (depending on the loaded packages).
- **mlr3pipelines** to combine learners with pre- and postprocessing steps.
- Package **mlr3viz** for some generic visualizations.
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.
- **mlr3tuning** for tuning of hyperparameters, **mlr3tuningspaces** for established default tuning spaces.

Other Learner: `Learner`, `LearnerClassif`, `LearnerRegr`, `mlr_learners`, `mlr_learners_classif.debug`, `mlr_learners_classif.featureless`, `mlr_learners_classif.rpart`, `mlr_learners_regr.debug`, `mlr_learners_regr.rpart`

`mlr_learners_regr.rpart`
Regression Tree Learner

Description

A LearnerRegr for a regression tree implemented in `rpart::rpart()` in package **rpart**.

Initial parameter values

- Parameter `xval` is initialized to 0 in order to save some computation time.

Custom mlr3 parameters

- Parameter `model` has been renamed to `keep_model`.

Dictionary

This Learner can be instantiated via the dictionary mlr_learners or with the associated sugar function `lrn()`:

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

Meta Information

- Task type: "regr"
- Predict Types: "response"
- Feature Types: "logical", "integer", "numeric", "factor", "ordered"
- Required Packages: **mlr3**, **rpart**

Parameters

Id	Type	Default	Levels	Range
cp	numeric	0.01		$[0, 1]$
keep_model	logical	FALSE	TRUE, FALSE	-
maxcompete	integer	4		$[0, \infty)$
maxdepth	integer	30		$[1, 30]$
maxsurrogate	integer	5		$[0, \infty)$
minbucket	integer	-		$[1, \infty)$
minsplit	integer	20		$[1, \infty)$
surrogatestyle	integer	0		$[0, 1]$
usesurrogate	integer	2		$[0, 2]$
xval	integer	10		$[0, \infty)$

Super classes

`mlr3::Learner` -> `mlr3::LearnerRegr` -> `LearnerRegrRpart`

Methods

Public methods:

- `LearnerRegrRpart$new()`
- `LearnerRegrRpart$importance()`
- `LearnerRegrRpart$selected_features()`
- `LearnerRegrRpart$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
LearnerRegrRpart$new()
```

Method `importance()`**:** The importance scores are extracted from the model slot `variable.importance`.

Usage:

```
LearnerRegrRpart$importance()
```

Returns: Named `numeric()`.

Method `selected_features()`**:** Selected features are extracted from the model slot `frame$var`.

Usage:

```
LearnerRegrRpart$selected_features()
```

Returns: `character()`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Breiman L, Friedman JH, Olshen RA, Stone CJ (1984). *Classification And Regression Trees*. Routledge. doi:10.1201/9781315139470.

See Also

- Chapter in the mlr3book: `https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-learners`
- Package **mlr3learners** for a solid collection of essential learners.
- Package mlr3extralearners for more learners.
- Dictionary of Learners: mlr_learners
- `as.data.table(mlr_learners)` for a table of available Learners in the running session (depending on the loaded packages).

- **mlr3pipelines** to combine learners with pre- and postprocessing steps.
- Package **mlr3viz** for some generic visualizations.
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.
- **mlr3tuning** for tuning of hyperparameters, **mlr3tuningspaces** for established default tuning spaces.

Other Learner: Learner, LearnerClassif, LearnerRegr, mlr_learners, mlr_learners_classif.debug, mlr_learners_classif.featureless, mlr_learners_classif.rpart, mlr_learners_regr.debug, mlr_learners_regr.featureless

---

`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. E.g., **mlr3proba** adds survival measures and **mlr3cluster** adds cluster analysis measures.

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

Format

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

Methods

See mlr3misc::Dictionary.

S3 methods

- `as.data.table(dict, ..., objects = FALSE)`
  mlr3misc::Dictionary -> `data.table::data.table()`
  Returns a `data.table::data.table()` with fields "key", "label", "task_type", "packages", "predict_type", and "task_properties" as columns. If `objects` is set to `TRUE`, the constructed objects are returned in the list column named `object`.

See Also

Sugar functions: `msr()`, `msrs()`

Implementation of most measures: **mlr3measures**

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

Other Measure: Measure, MeasureClassif, MeasureRegr, MeasureSimilarity, mlr_measures_aic, mlr_measures_bic, mlr_measures_classif.costs, mlr_measures_debug_classif, mlr_measures_elapsed_time, mlr_measures_internal_valid_score, mlr_measures_oob_error, mlr_measures_regr.pinball, mlr_measures_regr.rqr, mlr_measures_regr.rsq, mlr_measures_selected_features

Examples

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

`mlr_measures_aic` *Akaike Information Criterion Measure*

Description

Calculates the Akaike Information Criterion (AIC) which is a trade-off between goodness of fit (measured in terms of log-likelihood) and model complexity (measured in terms of number of included features). Internally, `stats::AIC()` is called with parameter `k` (defaulting to 2). Requires the learner property `"loglik"`, `NA` is returned for unsupported learners.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("aic")
msr("aic")
```

Meta Information

- Task type: "NA"
- Range: $(-\infty, \infty)$
- Minimize: TRUE
- Average: macro
- Required Prediction: "NA"
- Required Packages: **mlr3**

Parameters

Id	Type	Default	Range
k	integer	-	$[0, \infty)$

Super class

`mlr3::Measure` -> `MeasureAIC`

Methods

Public methods:

- `MeasureAIC$new()`
- `MeasureAIC$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
MeasureAIC$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-eval
- Package **mlr3measures** for the scoring functions. Dictionary of Measures: mlr_measures `as.data.table(mlr_measures)` for a table of available Measures in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other Measure: `Measure`, `MeasureClassif`, `MeasureRegr`, `MeasureSimilarity`, `mlr_measures`, `mlr_measures_bic`, `mlr_measures_classif.costs`, `mlr_measures_debug_classif`, `mlr_measures_elapsed_time`, `mlr_measures_internal_valid_score`, `mlr_measures_oob_error`, `mlr_measures_regr.pinball`, `mlr_measures_regr.rqr`, `mlr_measures_regr.rsq`, `mlr_measures_selected_features`

`mlr_measures_bic` *Bayesian Information Criterion Measure*

Description

Calculates the Bayesian Information Criterion (BIC) which is a trade-off between goodness of fit (measured in terms of log-likelihood) and model complexity (measured in terms of number of included features). Internally, `stats::BIC()` is called. Requires the learner property `"loglik"`, `NA` is returned for unsupported learners.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("bic")
msr("bic")
```

Meta Information

- Task type: "NA"
- Range: $(-\infty, \infty)$
- Minimize: TRUE
- Average: macro
- Required Prediction: "NA"
- Required Packages: **mlr3**

Parameters

Empty ParamSet

Super class

`mlr3::Measure` -> `MeasureBIC`

Methods

Public methods:

- `MeasureBIC$new()`
- `MeasureBIC$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
MeasureBIC$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-eval
- Package **mlr3measures** for the scoring functions. Dictionary of Measures: mlr_measures as.data.table(mlr_measures) for a table of available Measures in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other Measure: Measure, MeasureClassif, MeasureRegr, MeasureSimilarity, mlr_measures, mlr_measures_aic, mlr_measures_classif.costs, mlr_measures_debug_classif, mlr_measures_elapsed_time, mlr_measures_internal_valid_score, mlr_measures_oob_error, mlr_measures_regr.pinball, mlr_measures_regr.rqr, mlr_measures_regr.rsq, mlr_measures_selected_features

---

`mlr_measures_classif.acc`

Classification Accuracy

Description

Measure to compare true observed labels with predicted labels in multiclass classification tasks.

Details

The Classification Accuracy is defined as

$$\sum_{i=1}^{n} w_i \mathbf{1}\left(t_i = r_i\right),$$

where w_i are weights normalized to sum to 1 for all observations x_i.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

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

Parameters

Empty ParamSet

Meta Information

- Type: `"classif"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::acc()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbr`
`mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbe`
`mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`,
`mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.n`
`mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`,
`mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`,
`mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prauc`, `mlr_measures_classif.pr`
`mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`,
`mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other multiclass classification measures: `mlr_measures_classif.bacc`, `mlr_measures_classif.ce`,
`mlr_measures_classif.costs`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`,
`mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`,
`mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`

`mlr_measures_classif.auc`

Area Under the ROC Curve

Description

Measure to compare true observed labels with predicted probabilities in binary classification tasks.

Details

Computes the area under the Receiver Operator Characteristic (ROC) curve. The AUC can be interpreted as the probability that a randomly chosen positive observation has a higher predicted probability than a randomly chosen negative observation.

This measure is undefined if the true values are either all positive or all negative.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

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

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `prob`

Note

The score function calls `mlr3measures::auc()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbr`
`mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbe`
`mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`,
`mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.m`
`mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`,
`mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`,
`mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prauc`, `mlr_measures_classif.pr`
`mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`,
`mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other binary classification measures: `mlr_measures_classif.bbrier`, `mlr_measures_classif.dor`,
`mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`
`mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.npv`,
`mlr_measures_classif.ppv`, `mlr_measures_classif.prauc`, `mlr_measures_classif.precision`,
`mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`,
`mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

`mlr_measures_classif.bacc` *Balanced Accuracy*

Description

Measure to compare true observed labels with predicted labels in multiclass classification tasks.

Details

The Balanced Accuracy computes the weighted balanced accuracy, suitable for imbalanced data sets. It is defined analogously to the definition in sklearn.

First, all sample weights w_i are normalized per class so that each class has the same influence:

$$\hat{w}_i = \frac{w_i}{\sum_{j=1}^{n} w_j \cdot \mathbf{1}(t_j = t_i)}.$$

The Balanced Accuracy is then calculated as

$$\frac{1}{\sum_{i=1}^{n} \hat{w}_i} \sum_{i=1}^{n} \hat{w}_i \cdot \mathbf{1}(r_i = t_i).$$

This definition is equivalent to `acc()` with class-balanced sample weights.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.bacc")
msr("classif.bacc")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"classif"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::bacc()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: mlr_measures_classif.acc, mlr_measures_classif.auc, mlr_measures_classif.bbri
mlr_measures_classif.ce, mlr_measures_classif.costs, mlr_measures_classif.dor, mlr_measures_classif.fbe
mlr_measures_classif.fdr, mlr_measures_classif.fn, mlr_measures_classif.fnr, mlr_measures_classif.fomr,
mlr_measures_classif.fp, mlr_measures_classif.fpr, mlr_measures_classif.logloss, mlr_measures_classif.r
mlr_measures_classif.mauc_au1u, mlr_measures_classif.mauc_aunp, mlr_measures_classif.mauc_aunu,
mlr_measures_classif.mauc_mu, mlr_measures_classif.mbrier, mlr_measures_classif.mcc,
mlr_measures_classif.npv, mlr_measures_classif.ppv, mlr_measures_classif.prauc, mlr_measures_classif.pr
mlr_measures_classif.recall, mlr_measures_classif.sensitivity, mlr_measures_classif.specificity,
mlr_measures_classif.tn, mlr_measures_classif.tnr, mlr_measures_classif.tp, mlr_measures_classif.tpr

Other multiclass classification measures: mlr_measures_classif.acc, mlr_measures_classif.ce,
mlr_measures_classif.costs, mlr_measures_classif.logloss, mlr_measures_classif.mauc_au1p,
mlr_measures_classif.mauc_au1u, mlr_measures_classif.mauc_aunp, mlr_measures_classif.mauc_aunu,
mlr_measures_classif.mauc_mu, mlr_measures_classif.mbrier, mlr_measures_classif.mcc

---

mlr_measures_classif.bbrier

Binary Brier Score

Description

Measure to compare true observed labels with predicted probabilities in binary classification tasks.

Details

The Binary Brier Score is defined as

$$\frac{1}{n} \sum_{i=1}^{n} w_i (I_i - p_i)^2,$$

where w_i are the sample weights, and I_i is 1 if observation x_i belongs to the positive class, and 0 otherwise.

Note that this (more common) definition of the Brier score is equivalent to the original definition of the multi-class Brier score (see mbrier()) divided by 2.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function msr():

```
mlr_measures$get("classif.bbrier")
msr("classif.bbrier")
```

Parameters

Empty ParamSet

Meta Information

- Type: "binary"
- Range: $[0, 1]$
- Minimize: TRUE
- Required prediction: prob

Note

The score function calls mlr3measures::bbrier() from package **mlr3measures**.

If the measure is undefined for the input, NaN is returned. This can be customized by setting the field na_value.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: mlr_measures_classif.acc, mlr_measures_classif.auc, mlr_measures_classif.bacc
mlr_measures_classif.ce, mlr_measures_classif.costs, mlr_measures_classif.dor, mlr_measures_classif.fbe
mlr_measures_classif.fdr, mlr_measures_classif.fn, mlr_measures_classif.fnr, mlr_measures_classif.fomr,
mlr_measures_classif.fp, mlr_measures_classif.fpr, mlr_measures_classif.logloss, mlr_measures_classif.m
mlr_measures_classif.mauc_au1u, mlr_measures_classif.mauc_aunp, mlr_measures_classif.mauc_aunu,
mlr_measures_classif.mauc_mu, mlr_measures_classif.mbrier, mlr_measures_classif.mcc,
mlr_measures_classif.npv, mlr_measures_classif.ppv, mlr_measures_classif.prauc, mlr_measures_classif.pr
mlr_measures_classif.recall, mlr_measures_classif.sensitivity, mlr_measures_classif.specificity,
mlr_measures_classif.tn, mlr_measures_classif.tnr, mlr_measures_classif.tp, mlr_measures_classif.tpr

Other binary classification measures: mlr_measures_classif.auc, mlr_measures_classif.dor,
mlr_measures_classif.fbeta, mlr_measures_classif.fdr, mlr_measures_classif.fn, mlr_measures_classif.fnr
mlr_measures_classif.fomr, mlr_measures_classif.fp, mlr_measures_classif.fpr, mlr_measures_classif.npv,
mlr_measures_classif.ppv, mlr_measures_classif.prauc, mlr_measures_classif.precision,
mlr_measures_classif.recall, mlr_measures_classif.sensitivity, mlr_measures_classif.specificity,
mlr_measures_classif.tn, mlr_measures_classif.tnr, mlr_measures_classif.tp, mlr_measures_classif.tpr

---

mlr_measures_classif.ce

Classification Error

Description

Measure to compare true observed labels with predicted labels in multiclass classification tasks.

Details

The Classification Error is defined as

$$\frac{1}{n}\sum_{i=1}^{n} w_i \mathbf{1}\left(t_i \neq r_i\right),$$

where w_i are normalized weights for each observation x_i.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

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

Parameters

Empty ParamSet

Meta Information

- Type: `"classif"`
- Range: $[0, 1]$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::ce()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prauc`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other multiclass classification measures: mlr_measures_classif.acc, mlr_measures_classif.bacc, mlr_measures_classif.costs, mlr_measures_classif.logloss, mlr_measures_classif.mauc_au1p, mlr_measures_classif.mauc_au1u, mlr_measures_classif.mauc_aunp, mlr_measures_classif.mauc_aunu, mlr_measures_classif.mauc_mu, mlr_measures_classif.mbrier, mlr_measures_classif.mcc

---

`mlr_measures_classif.costs`

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`.

For calculation of the score, the confusion matrix is multiplied element-wise with the cost matrix. The costs are then summed up (and potentially divided by the number of observations if `normalize` is set to `TRUE` (default)).

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.costs")
msr("classif.costs")
```

Meta Information

- Task type: "classif"
- Range: $(-\infty, \infty)$
- Minimize: TRUE
- Average: macro
- Required Prediction: "response"
- Required Packages: **mlr3**

Parameters

Id	Type	Default	Levels
normalize	logical	-	TRUE, FALSE

Super classes

`mlr3::Measure` -> `mlr3::MeasureClassif` -> `MeasureClassifCosts`

Active bindings

`costs` (numeric `matrix()`)
Matrix of costs (truth in columns, predicted response in rows).

Methods

Public methods:

- `MeasureClassifCosts$new()`
- `MeasureClassifCosts$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
MeasureClassifCosts$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-eval
- Package **mlr3measures** for the scoring functions. Dictionary of Measures: mlr_measures `as.data.table(mlr_measures)` for a table of available Measures in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other Measure: `Measure`, `MeasureClassif`, `MeasureRegr`, `MeasureSimilarity`, `mlr_measures`, `mlr_measures_aic`, `mlr_measures_bic`, `mlr_measures_debug_classif`, `mlr_measures_elapsed_time`, `mlr_measures_internal_valid_score`, `mlr_measures_oob_error`, `mlr_measures_regr.pinball`, `mlr_measures_regr.rqr`, `mlr_measures_regr.rsq`, `mlr_measures_selected_features`

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prauc`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other multiclass classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.ce`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`

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

# fit models and evaluate with the cost measure
learner = lrn("classif.rpart")
rr = resample(task, learner, rsmp("cv", folds = 3))
rr$aggregate(m)
```

---

mlr_measures_classif.dor

Diagnostic Odds Ratio

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

The Diagnostic Odds Ratio is defined as

$$\frac{\mathrm{TP}/\mathrm{FP}}{\mathrm{FN}/\mathrm{TN}}.$$

This measure is undefined if FP = 0 or FN = 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.dor")
msr("classif.dor")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, \infty)$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::dor()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logl`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prau`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other binary classification measures: `mlr_measures_classif.auc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prauc`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

`mlr_measures_classif.fbeta`

F-beta Score

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

With P as `precision()` and R as `recall()`, the F-beta Score is defined as

$$(1 + \beta^2) \frac{P \cdot R}{(\beta^2 P) + R}.$$

It measures the effectiveness of retrieval with respect to a user who attaches β times as much importance to recall as precision. For $\beta = 1$, this measure is called "F1" score.

This measure is undefined if precision or recall is undefined, i.e. TP + FP = 0 or TP + FN = 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.fbeta")
msr("classif.fbeta")
```

Parameters

Id	Type	Default	Range
beta	integer	-	$[0, \infty)$

Meta Information

- Type: `"binary"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::fbeta()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`
`mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`,

mlr_measures_classif.dor, mlr_measures_classif.fdr, mlr_measures_classif.fn, mlr_measures_classif.fnr, mlr_measures_classif.fomr, mlr_measures_classif.fp, mlr_measures_classif.fpr, mlr_measures_classif.logl mlr_measures_classif.mauc_au1p, mlr_measures_classif.mauc_au1u, mlr_measures_classif.mauc_aunp, mlr_measures_classif.mauc_aunu, mlr_measures_classif.mauc_mu, mlr_measures_classif.mbrier, mlr_measures_classif.mcc, mlr_measures_classif.npv, mlr_measures_classif.ppv, mlr_measures_classif.prau mlr_measures_classif.precision, mlr_measures_classif.recall, mlr_measures_classif.sensitivity, mlr_measures_classif.specificity, mlr_measures_classif.tn, mlr_measures_classif.tnr, mlr_measures_classif.tp, mlr_measures_classif.tpr

Other binary classification measures: mlr_measures_classif.auc, mlr_measures_classif.bbrier, mlr_measures_classif.dor, mlr_measures_classif.fdr, mlr_measures_classif.fn, mlr_measures_classif.fnr, mlr_measures_classif.fomr, mlr_measures_classif.fp, mlr_measures_classif.fpr, mlr_measures_classif.npv, mlr_measures_classif.ppv, mlr_measures_classif.prauc, mlr_measures_classif.precision, mlr_measures_classif.recall, mlr_measures_classif.sensitivity, mlr_measures_classif.specificity, mlr_measures_classif.tn, mlr_measures_classif.tnr, mlr_measures_classif.tp, mlr_measures_classif.tpr

---

`mlr_measures_classif.fdr`

False Discovery Rate

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

The False Discovery Rate is defined as

$$\frac{\mathrm{FP}}{\mathrm{TP} + \mathrm{FP}}.$$

This measure is undefined if TP + FP = 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.fdr")
msr("classif.fdr")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, 1]$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::fdr()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logl`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prau`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other binary classification measures: `mlr_measures_classif.auc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prauc`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

---

mlr_measures_classif.fn

False Negatives

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

This measure counts the false negatives (type 2 error), i.e. the number of predictions indicating a negative class label while in fact it is positive. This is sometimes also called a "miss" or an "underestimation".

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.fn")
msr("classif.fn")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, \infty)$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::fn()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prauc`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other binary classification measures: `mlr_measures_classif.auc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prauc`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

`mlr_measures_classif.fnr` *False Negative Rate*

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

The False Negative Rate is defined as

$$\frac{\mathrm{FN}}{\mathrm{TP} + \mathrm{FN}}.$$

Also know as "miss rate".

This measure is undefined if TP + FN = 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.fnr")
msr("classif.fnr")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, 1]$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::fnr()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: mlr_measures_classif.acc, mlr_measures_classif.auc, mlr_measures_classif.bacc, mlr_measures_classif.bbrier, mlr_measures_classif.ce, mlr_measures_classif.costs, mlr_measures_classif.dor, mlr_measures_classif.fbeta, mlr_measures_classif.fdr, mlr_measures_classif.fn, mlr_measures_classif.fomr, mlr_measures_classif.fp, mlr_measures_classif.fpr, mlr_measures_classif.logl, mlr_measures_classif.mauc_au1p, mlr_measures_classif.mauc_au1u, mlr_measures_classif.mauc_aunp, mlr_measures_classif.mauc_aunu, mlr_measures_classif.mauc_mu, mlr_measures_classif.mbrier, mlr_measures_classif.mcc, mlr_measures_classif.npv, mlr_measures_classif.ppv, mlr_measures_classif.prau, mlr_measures_classif.precision, mlr_measures_classif.recall, mlr_measures_classif.sensitivity, mlr_measures_classif.specificity, mlr_measures_classif.tn, mlr_measures_classif.tnr, mlr_measures_classif.tp, mlr_measures_classif.tpr

Other binary classification measures: mlr_measures_classif.auc, mlr_measures_classif.bbrier, mlr_measures_classif.dor, mlr_measures_classif.fbeta, mlr_measures_classif.fdr, mlr_measures_classif.fn, mlr_measures_classif.fomr, mlr_measures_classif.fp, mlr_measures_classif.fpr, mlr_measures_classif.npv, mlr_measures_classif.ppv, mlr_measures_classif.prauc, mlr_measures_classif.precision, mlr_measures_classif.recall, mlr_measures_classif.sensitivity, mlr_measures_classif.specificity, mlr_measures_classif.tn, mlr_measures_classif.tnr, mlr_measures_classif.tp, mlr_measures_classif.tpr

---

mlr_measures_classif.fomr

False Omission Rate

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

The False Omission Rate is defined as

$$\frac{\mathrm{FN}}{\mathrm{FN} + \mathrm{TN}}.$$

This measure is undefined if FN + TN = 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function msr():

```
mlr_measures$get("classif.fomr")
msr("classif.fomr")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, 1]$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::fomr()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`
`mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`,
`mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fr`
`mlr_measures_classif.fnr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.loglo`
`mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`,
`mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`,
`mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prau`
`mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`,
`mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`,
`mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other binary classification measures: `mlr_measures_classif.auc`, `mlr_measures_classif.bbrier`,
`mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fr`
`mlr_measures_classif.fnr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.npv`,
`mlr_measures_classif.ppv`, `mlr_measures_classif.prauc`, `mlr_measures_classif.precision`,
`mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`,
`mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

`mlr_measures_classif.fp`

False Positives

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

This measure counts the false positives (type 1 error), i.e. the number of predictions indicating a positive class label while in fact it is negative. This is sometimes also called a "false alarm".

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.fp")
msr("classif.fp")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, \infty)$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::fp()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fpr`, `mlr_measures_classif.log`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prau`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other binary classification measures: `mlr_measures_classif.auc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fpr`, `mlr_measures_classif.npv`

mlr_measures_classif.ppv, mlr_measures_classif.prauc, mlr_measures_classif.precision, mlr_measures_classif.recall, mlr_measures_classif.sensitivity, mlr_measures_classif.specificity, mlr_measures_classif.tn, mlr_measures_classif.tnr, mlr_measures_classif.tp, mlr_measures_classif.tpr

---

`mlr_measures_classif.fpr`

False Positive Rate

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

The False Positive Rate is defined as

$$\frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}.$$

Also know as fall out or probability of false alarm.

This measure is undefined if FP + TN = 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.fpr")
msr("classif.fpr")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, 1]$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::fpr()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: mlr_measures_classif.acc, mlr_measures_classif.auc, mlr_measures_classif.bacc
mlr_measures_classif.bbrier, mlr_measures_classif.ce, mlr_measures_classif.costs,
mlr_measures_classif.dor, mlr_measures_classif.fbeta, mlr_measures_classif.fdr, mlr_measures_classif.fr
mlr_measures_classif.fnr, mlr_measures_classif.fomr, mlr_measures_classif.fp, mlr_measures_classif.logl
mlr_measures_classif.mauc_au1p, mlr_measures_classif.mauc_au1u, mlr_measures_classif.mauc_aunp,
mlr_measures_classif.mauc_aunu, mlr_measures_classif.mauc_mu, mlr_measures_classif.mbrier,
mlr_measures_classif.mcc, mlr_measures_classif.npv, mlr_measures_classif.ppv, mlr_measures_classif.prau
mlr_measures_classif.precision, mlr_measures_classif.recall, mlr_measures_classif.sensitivity,
mlr_measures_classif.specificity, mlr_measures_classif.tn, mlr_measures_classif.tnr,
mlr_measures_classif.tp, mlr_measures_classif.tpr

Other binary classification measures: mlr_measures_classif.auc, mlr_measures_classif.bbrier,
mlr_measures_classif.dor, mlr_measures_classif.fbeta, mlr_measures_classif.fdr, mlr_measures_classif.fr
mlr_measures_classif.fnr, mlr_measures_classif.fomr, mlr_measures_classif.fp, mlr_measures_classif.npv,
mlr_measures_classif.ppv, mlr_measures_classif.prauc, mlr_measures_classif.precision,
mlr_measures_classif.recall, mlr_measures_classif.sensitivity, mlr_measures_classif.specificity,
mlr_measures_classif.tn, mlr_measures_classif.tnr, mlr_measures_classif.tp, mlr_measures_classif.tpr

---

mlr_measures_classif.logloss

Log Loss

Description

Measure to compare true observed labels with predicted probabilities in multiclass classification tasks.

Details

The Log Loss (a.k.a Benoulli Loss, Logistic Loss, Cross-Entropy Loss) is defined as

$$-\frac{1}{n}\sum_{i=1}^{n} w_i \log\left(p_i\right)$$

where p_i is the probability for the true class of observation i and w_i are normalized weights for each observation x_i.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function msr():

```
mlr_measures$get("classif.logloss")
msr("classif.logloss")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"classif"`
- Range: $[0, \infty)$
- Minimize: `TRUE`
- Required prediction: `prob`

Note

The score function calls `mlr3measures::logloss()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prau`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other multiclass classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`

---

mlr_measures_classif.mauc_au1p

Multiclass AUC Scores

Description

Measure to compare true observed labels with predicted probabilities in multiclass classification tasks.

Details

Multiclass AUC measures.

- *AUNU*: AUC of each class against the rest, using the uniform class distribution. Computes the AUC treating a c-dimensional classifier as c two-dimensional 1-vs-rest classifiers, where classes are assumed to have uniform distribution, in order to have a measure which is independent of class distribution change (Fawcett 2001).
- *AUNP*: AUC of each class against the rest, using the a-priori class distribution. Computes the AUC treating a c-dimensional classifier as c two-dimensional 1-vs-rest classifiers, taking into account the prior probability of each class (Fawcett 2001).
- *AU1U*: AUC of each class against each other, using the uniform class distribution. Computes something like the AUC of `c(c - 1)` binary classifiers (all possible pairwise combinations). See Hand (2001) for details.
- *AU1P*: AUC of each class against each other, using the a-priori class distribution. Computes something like AUC of `c(c - 1)` binary classifiers while considering the a-priori distribution of the classes as suggested in Ferri (2009). Note we deviate from the definition in Ferri (2009) by a factor of c.
- *MU*: Multiclass AUC as defined in Kleinman and Page (2019). This measure is an average of the pairwise AUCs between all classes. The measure was tested against the Python implementation by Ross Kleinman.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.mauc_au1p")
msr("classif.mauc_au1p")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"classif"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `prob`

Note

The score function calls `mlr3measures::mauc_au1p()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prau`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other multiclass classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`

`mlr_measures_classif.mauc_au1u`

Multiclass AUC Scores

Description

Measure to compare true observed labels with predicted probabilities in multiclass classification tasks.

Details

Multiclass AUC measures.

- *AUNU*: AUC of each class against the rest, using the uniform class distribution. Computes the AUC treating a `c`-dimensional classifier as `c` two-dimensional 1-vs-rest classifiers, where classes are assumed to have uniform distribution, in order to have a measure which is independent of class distribution change (Fawcett 2001).
- *AUNP*: AUC of each class against the rest, using the a-priori class distribution. Computes the AUC treating a `c`-dimensional classifier as `c` two-dimensional 1-vs-rest classifiers, taking into account the prior probability of each class (Fawcett 2001).
- *AU1U*: AUC of each class against each other, using the uniform class distribution. Computes something like the AUC of `c(c - 1)` binary classifiers (all possible pairwise combinations). See Hand (2001) for details.
- *AU1P*: AUC of each class against each other, using the a-priori class distribution. Computes something like AUC of `c(c - 1)` binary classifiers while considering the a-priori distribution of the classes as suggested in Ferri (2009). Note we deviate from the definition in Ferri (2009) by a factor of `c`.

- *MU*: Multiclass AUC as defined in Kleinman and Page (2019). This measure is an average of the pairwise AUCs between all classes. The measure was tested against the Python implementation by Ross Kleinman.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.mauc_au1u")
msr("classif.mauc_au1u")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"classif"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `prob`

Note

The score function calls `mlr3measures::mauc_au1u()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prau`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other multiclass classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`

`mlr_measures_classif.mauc_aunp` *Multiclass AUC Scores*

Description

Measure to compare true observed labels with predicted probabilities in multiclass classification tasks.

Details

Multiclass AUC measures.

- *AUNU*: AUC of each class against the rest, using the uniform class distribution. Computes the AUC treating a `c`-dimensional classifier as `c` two-dimensional 1-vs-rest classifiers, where classes are assumed to have uniform distribution, in order to have a measure which is independent of class distribution change (Fawcett 2001).
- *AUNP*: AUC of each class against the rest, using the a-priori class distribution. Computes the AUC treating a `c`-dimensional classifier as `c` two-dimensional 1-vs-rest classifiers, taking into account the prior probability of each class (Fawcett 2001).
- *AU1U*: AUC of each class against each other, using the uniform class distribution. Computes something like the AUC of `c(c - 1)` binary classifiers (all possible pairwise combinations). See Hand (2001) for details.
- *AU1P*: AUC of each class against each other, using the a-priori class distribution. Computes something like AUC of `c(c - 1)` binary classifiers while considering the a-priori distribution of the classes as suggested in Ferri (2009). Note we deviate from the definition in Ferri (2009) by a factor of `c`.
- *MU*: Multiclass AUC as defined in Kleinman and Page (2019). This measure is an average of the pairwise AUCs between all classes. The measure was tested against the Python implementation by Ross Kleinman.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.mauc_aunp")
msr("classif.mauc_aunp")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"classif"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `prob`

Note

The score function calls `mlr3measures::mauc_aunp()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prau`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other multiclass classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`

---

mlr_measures_classif.mauc_aunu

Multiclass AUC Scores

Description

Measure to compare true observed labels with predicted probabilities in multiclass classification tasks.

Details

Multiclass AUC measures.

- *AUNU*: AUC of each class against the rest, using the uniform class distribution. Computes the AUC treating a c-dimensional classifier as c two-dimensional 1-vs-rest classifiers, where classes are assumed to have uniform distribution, in order to have a measure which is independent of class distribution change (Fawcett 2001).
- *AUNP*: AUC of each class against the rest, using the a-priori class distribution. Computes the AUC treating a c-dimensional classifier as c two-dimensional 1-vs-rest classifiers, taking into account the prior probability of each class (Fawcett 2001).
- *AU1U*: AUC of each class against each other, using the uniform class distribution. Computes something like the AUC of `c(c - 1)` binary classifiers (all possible pairwise combinations). See Hand (2001) for details.
- *AU1P*: AUC of each class against each other, using the a-priori class distribution. Computes something like AUC of `c(c - 1)` binary classifiers while considering the a-priori distribution of the classes as suggested in Ferri (2009). Note we deviate from the definition in Ferri (2009) by a factor of c.
- *MU*: Multiclass AUC as defined in Kleinman and Page (2019). This measure is an average of the pairwise AUCs between all classes. The measure was tested against the Python implementation by Ross Kleinman.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.mauc_aunu")
msr("classif.mauc_aunu")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"classif"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `prob`

Note

The score function calls `mlr3measures::mauc_aunu()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prau`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other multiclass classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`

`mlr_measures_classif.mauc_mu`

Multiclass AUC Scores

Description

Measure to compare true observed labels with predicted probabilities in multiclass classification tasks.

Details

Multiclass AUC measures.

- *AUNU*: AUC of each class against the rest, using the uniform class distribution. Computes the AUC treating a `c`-dimensional classifier as `c` two-dimensional 1-vs-rest classifiers, where classes are assumed to have uniform distribution, in order to have a measure which is independent of class distribution change (Fawcett 2001).
- *AUNP*: AUC of each class against the rest, using the a-priori class distribution. Computes the AUC treating a `c`-dimensional classifier as `c` two-dimensional 1-vs-rest classifiers, taking into account the prior probability of each class (Fawcett 2001).
- *AU1U*: AUC of each class against each other, using the uniform class distribution. Computes something like the AUC of `c(c - 1)` binary classifiers (all possible pairwise combinations). See Hand (2001) for details.
- *AU1P*: AUC of each class against each other, using the a-priori class distribution. Computes something like AUC of `c(c - 1)` binary classifiers while considering the a-priori distribution of the classes as suggested in Ferri (2009). Note we deviate from the definition in Ferri (2009) by a factor of `c`.

- *MU*: Multiclass AUC as defined in Kleinman and Page (2019). This measure is an average of the pairwise AUCs between all classes. The measure was tested against the Python implementation by Ross Kleinman.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.mauc_mu")
msr("classif.mauc_mu")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"classif"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `prob`

Note

The score function calls `mlr3measures::mauc_mu()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prau`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other multiclass classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`

`mlr_measures_classif.mbrier` *Multiclass Brier Score*

Description

Measure to compare true observed labels with predicted probabilities in multiclass classification tasks.

Details

Brier score for multi-class classification problems with k labels defined as

$$\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k}(I_{ij} - p_{ij})^2.$$

I_{ij} is 1 if observation x_i has true label j, and 0 otherwise. p_{ij} is the probability that observation x_i belongs to class j.

Note that there also is the more common definition of the Brier score for binary classification problems in `bbrier()`.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.mbrier")
msr("classif.mbrier")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"classif"`
- Range: $[0, 2]$
- Minimize: `TRUE`
- Required prediction: `prob`

Note

The score function calls `mlr3measures::mbrier()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prau`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other multiclass classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mcc`

`mlr_measures_classif.mcc`

Matthews Correlation Coefficient

Description

Measure to compare true observed labels with predicted labels in multiclass classification tasks.

Details

In the binary case, the Matthews Correlation Coefficient is defined as

$$\frac{\mathrm{TP} \cdot \mathrm{TN} - \mathrm{FP} \cdot \mathrm{FN}}{\sqrt{(\mathrm{TP} + \mathrm{FP})(\mathrm{TP} + \mathrm{FN})(\mathrm{TN} + \mathrm{FP})(\mathrm{TN} + \mathrm{FN})}},$$

where TP, FP, TN, TP are the number of true positives, false positives, true negatives, and false negatives respectively.

In the multi-class case, the Matthews Correlation Coefficient is defined for a multi-class confusion matrix C with K classes:

$$\frac{c \cdot s - \sum_k^K p_k \cdot t_k}{\sqrt{(s^2 - \sum_k^K p_k^2) \cdot (s^2 - \sum_k^K t_k^2)}},$$

where

- $s = \sum_i^K \sum_j^K C_{ij}$: total number of samples
- $c = \sum_k^K C_{kk}$: total number of correctly predicted samples

- $t_k = \sum_i^K C_{ik}$: number of predictions for each class k
- $p_k = \sum_j^K C_{kj}$: number of true occurrences for each class k.

The above formula is undefined if any of the four sums in the denominator is 0 in the binary case and more generally if either $s^2 - \sum_k^K p_k^2$ or $s^2 - \sum_k^K t_k^2$) is equal to 0. The denominator is then set to 1.

When there are more than two classes, the MCC will no longer range between -1 and +1. Instead, the minimum value will be between -1 and 0 depending on the true distribution. The maximum value is always +1.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.mcc")
msr("classif.mcc")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"classif"`
- Range: $[-1, 1]$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::mcc()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`
`mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`,
`mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`
`mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`,
`mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`,
`mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`,
`mlr_measures_classif.mbrier`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.p`

mlr_measures_classif.precision, mlr_measures_classif.recall, mlr_measures_classif.sensitivity, mlr_measures_classif.specificity, mlr_measures_classif.tn, mlr_measures_classif.tnr, mlr_measures_classif.tp, mlr_measures_classif.tpr

Other multiclass classification measures: mlr_measures_classif.acc, mlr_measures_classif.bacc, mlr_measures_classif.ce, mlr_measures_classif.costs, mlr_measures_classif.logloss, mlr_measures_classif.mauc_au1p, mlr_measures_classif.mauc_au1u, mlr_measures_classif.mauc_aunp, mlr_measures_classif.mauc_aunu, mlr_measures_classif.mauc_mu, mlr_measures_classif.mbrier

---

`mlr_measures_classif.npv`

Negative Predictive Value

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

The Negative Predictive Value is defined as

$$\frac{\mathrm{TN}}{\mathrm{FN} + \mathrm{TN}}.$$

This measure is undefined if FN + TN = 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.npv")
msr("classif.npv")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::npv()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.ppv`, `mlr_measures_classif.p`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other binary classification measures: `mlr_measures_classif.auc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prauc`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

`mlr_measures_classif.ppv`

Positive Predictive Value

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

The Positive Predictive Value is defined as

$$\frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}.$$

Also know as "precision".

This measure is undefined if TP + FP = 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.ppv")
msr("classif.ppv")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::ppv()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.p`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other binary classification measures: `mlr_measures_classif.auc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.npv`, `mlr_measures_classif.prauc`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

`mlr_measures_classif.prauc`
Area Under the Precision-Recall Curve

Description

Measure to compare true observed labels with predicted probabilities in binary classification tasks.

Details

Computes the area under the Precision-Recall curve (PRC). The PRC can be interpreted as the relationship between precision and recall (sensitivity), and is considered to be a more appropriate measure for unbalanced datasets than the ROC curve. The AUC-PRC is computed by integration of the piecewise function.

This measure is undefined if the true values are either all positive or all negative.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.prauc")
msr("classif.prauc")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `prob`

Note

The score function calls `mlr3measures::prauc()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.p`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other binary classification measures: `mlr_measures_classif.auc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

---

mlr_measures_classif.precision

Positive Predictive Value

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

The Positive Predictive Value is defined as

$$\frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}.$$

Also know as "precision".

This measure is undefined if TP + FP = 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.precision")
msr("classif.precision")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::precision()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`
`mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`,
`mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fr`
`mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`,
`mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`,
`mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`,
`mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.p`
`mlr_measures_classif.prauc`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`,
`mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`,
`mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other binary classification measures: `mlr_measures_classif.auc`, `mlr_measures_classif.bbrier`,
`mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fr`
`mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`,
`mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prauc`, `mlr_measures_classif.re`
`mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`,
`mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

`mlr_measures_classif.recall` *True Positive Rate*

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

The True Positive Rate is defined as

$$\frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}.$$

This is also know as "recall", "sensitivity", or "probability of detection".

This measure is undefined if TP + FN = 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.recall")
msr("classif.recall")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::recall()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`
`mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`,
`mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`
`mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`,
`mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`,
`mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`,
`mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.p`
`mlr_measures_classif.prauc`, `mlr_measures_classif.precision`, `mlr_measures_classif.sensitivity`,
`mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`,
`mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other binary classification measures: mlr_measures_classif.auc, mlr_measures_classif.bbrier, mlr_measures_classif.dor, mlr_measures_classif.fbeta, mlr_measures_classif.fdr, mlr_measures_classif.fr mlr_measures_classif.fnr, mlr_measures_classif.fomr, mlr_measures_classif.fp, mlr_measures_classif.fpr, mlr_measures_classif.npv, mlr_measures_classif.ppv, mlr_measures_classif.prauc, mlr_measures_classif.pr mlr_measures_classif.sensitivity, mlr_measures_classif.specificity, mlr_measures_classif.tn, mlr_measures_classif.tnr, mlr_measures_classif.tp, mlr_measures_classif.tpr

---

`mlr_measures_classif.sensitivity`
True Positive Rate

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

The True Positive Rate is defined as

$$\frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}.$$

This is also know as "recall", "sensitivity", or "probability of detection".

This measure is undefined if TP + FN = 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.sensitivity")
msr("classif.sensitivity")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::sensitivity()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: mlr_measures_classif.acc, mlr_measures_classif.auc, mlr_measures_classif.bacc
mlr_measures_classif.bbrier, mlr_measures_classif.ce, mlr_measures_classif.costs,
mlr_measures_classif.dor, mlr_measures_classif.fbeta, mlr_measures_classif.fdr, mlr_measures_classif.fr
mlr_measures_classif.fnr, mlr_measures_classif.fomr, mlr_measures_classif.fp, mlr_measures_classif.fpr,
mlr_measures_classif.logloss, mlr_measures_classif.mauc_au1p, mlr_measures_classif.mauc_au1u,
mlr_measures_classif.mauc_aunp, mlr_measures_classif.mauc_aunu, mlr_measures_classif.mauc_mu,
mlr_measures_classif.mbrier, mlr_measures_classif.mcc, mlr_measures_classif.npv, mlr_measures_classif.p
mlr_measures_classif.prauc, mlr_measures_classif.precision, mlr_measures_classif.recall,
mlr_measures_classif.specificity, mlr_measures_classif.tn, mlr_measures_classif.tnr,
mlr_measures_classif.tp, mlr_measures_classif.tpr

Other binary classification measures: mlr_measures_classif.auc, mlr_measures_classif.bbrier,
mlr_measures_classif.dor, mlr_measures_classif.fbeta, mlr_measures_classif.fdr, mlr_measures_classif.fr
mlr_measures_classif.fnr, mlr_measures_classif.fomr, mlr_measures_classif.fp, mlr_measures_classif.fpr,
mlr_measures_classif.npv, mlr_measures_classif.ppv, mlr_measures_classif.prauc, mlr_measures_classif.pr
mlr_measures_classif.recall, mlr_measures_classif.specificity, mlr_measures_classif.tn,
mlr_measures_classif.tnr, mlr_measures_classif.tp, mlr_measures_classif.tpr

---

mlr_measures_classif.specificity

True Negative Rate

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

The True Negative Rate is defined as

$$\frac{\mathrm{TN}}{\mathrm{FP} + \mathrm{TN}}.$$

Also know as "specificity" or "selectivity".

This measure is undefined if FP + TN = 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function msr():

```
mlr_measures$get("classif.specificity")
msr("classif.specificity")
```

Parameters

Empty ParamSet

Meta Information

- Type: "binary"
- Range: $[0, 1]$
- Minimize: FALSE
- Required prediction: response

Note

The score function calls mlr3measures::specificity() from package **mlr3measures**.

If the measure is undefined for the input, NaN is returned. This can be customized by setting the field na_value.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: mlr_measures_classif.acc, mlr_measures_classif.auc, mlr_measures_classif.bacc
mlr_measures_classif.bbrier, mlr_measures_classif.ce, mlr_measures_classif.costs,
mlr_measures_classif.dor, mlr_measures_classif.fbeta, mlr_measures_classif.fdr, mlr_measures_classif.fr
mlr_measures_classif.fnr, mlr_measures_classif.fomr, mlr_measures_classif.fp, mlr_measures_classif.fpr,
mlr_measures_classif.logloss, mlr_measures_classif.mauc_au1p, mlr_measures_classif.mauc_au1u,
mlr_measures_classif.mauc_aunp, mlr_measures_classif.mauc_aunu, mlr_measures_classif.mauc_mu,
mlr_measures_classif.mbrier, mlr_measures_classif.mcc, mlr_measures_classif.npv, mlr_measures_classif.p
mlr_measures_classif.prauc, mlr_measures_classif.precision, mlr_measures_classif.recall,
mlr_measures_classif.sensitivity, mlr_measures_classif.tn, mlr_measures_classif.tnr,
mlr_measures_classif.tp, mlr_measures_classif.tpr

Other binary classification measures: mlr_measures_classif.auc, mlr_measures_classif.bbrier,
mlr_measures_classif.dor, mlr_measures_classif.fbeta, mlr_measures_classif.fdr, mlr_measures_classif.fr
mlr_measures_classif.fnr, mlr_measures_classif.fomr, mlr_measures_classif.fp, mlr_measures_classif.fpr,
mlr_measures_classif.npv, mlr_measures_classif.ppv, mlr_measures_classif.prauc, mlr_measures_classif.pr
mlr_measures_classif.recall, mlr_measures_classif.sensitivity, mlr_measures_classif.tn,
mlr_measures_classif.tnr, mlr_measures_classif.tp, mlr_measures_classif.tpr

---

mlr_measures_classif.tn

True Negatives

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

This measure counts the true negatives, i.e. the number of predictions correctly indicating a negative class label. This is sometimes also called a "correct rejection".

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.tn")
msr("classif.tn")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, \infty)$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::tn()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.p`, `mlr_measures_classif.prauc`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other binary classification measures: `mlr_measures_classif.auc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`,

mlr_measures_classif.npv, mlr_measures_classif.ppv, mlr_measures_classif.prauc, mlr_measures_classif.pr
mlr_measures_classif.recall, mlr_measures_classif.sensitivity, mlr_measures_classif.specificity,
mlr_measures_classif.tnr, mlr_measures_classif.tp, mlr_measures_classif.tpr

---

`mlr_measures_classif.tnr`
True Negative Rate

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

The True Negative Rate is defined as

$$\frac{\mathrm{TN}}{\mathrm{FP} + \mathrm{TN}}.$$

Also know as "specificity" or "selectivity".

This measure is undefined if FP + TN = 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.tnr")
msr("classif.tnr")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::tnr()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.p`, `mlr_measures_classif.prauc`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

Other binary classification measures: `mlr_measures_classif.auc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prauc`, `mlr_measures_classif.pr`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tp`, `mlr_measures_classif.tpr`

---

mlr_measures_classif.tp

True Positives

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

This measure counts the true positives, i.e. the number of predictions correctly indicating a positive class label. This is sometimes also called a "hit".

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.tp")
msr("classif.tp")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, \infty)$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::tp()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.p`, `mlr_measures_classif.prauc`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tpr`

Other binary classification measures: `mlr_measures_classif.auc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.npv`, `mlr_measures_classif.ppv`, `mlr_measures_classif.prauc`, `mlr_measures_classif.pr`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tpr`

`mlr_measures_classif.tpr`

True Positive Rate

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.

Details

The True Positive Rate is defined as

$$\frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}.$$

This is also know as "recall", "sensitivity", or "probability of detection".

This measure is undefined if TP + FN = 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("classif.tpr")
msr("classif.tpr")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"binary"`
- Range: $[0, 1]$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::tpr()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other classification measures: `mlr_measures_classif.acc`, `mlr_measures_classif.auc`, `mlr_measures_classif.bacc`, `mlr_measures_classif.bbrier`, `mlr_measures_classif.ce`, `mlr_measures_classif.costs`, `mlr_measures_classif.dor`, `mlr_measures_classif.fbeta`, `mlr_measures_classif.fdr`, `mlr_measures_classif.fn`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_au1p`, `mlr_measures_classif.mauc_au1u`, `mlr_measures_classif.mauc_aunp`, `mlr_measures_classif.mauc_aunu`, `mlr_measures_classif.mauc_mu`, `mlr_measures_classif.mbrier`, `mlr_measures_classif.mcc`, `mlr_measures_classif.npv`, `mlr_measures_classif.p`, `mlr_measures_classif.prauc`, `mlr_measures_classif.precision`, `mlr_measures_classif.recall`, `mlr_measures_classif.sensitivity`, `mlr_measures_classif.specificity`, `mlr_measures_classif.tn`, `mlr_measures_classif.tnr`, `mlr_measures_classif.tp`

Other binary classification measures: mlr_measures_classif.auc, mlr_measures_classif.bbrier, mlr_measures_classif.dor, mlr_measures_classif.fbeta, mlr_measures_classif.fdr, mlr_measures_classif.fr mlr_measures_classif.fnr, mlr_measures_classif.fomr, mlr_measures_classif.fp, mlr_measures_classif.fpr, mlr_measures_classif.npv, mlr_measures_classif.ppv, mlr_measures_classif.prauc, mlr_measures_classif.pr mlr_measures_classif.recall, mlr_measures_classif.sensitivity, mlr_measures_classif.specificity, mlr_measures_classif.tn, mlr_measures_classif.tnr, mlr_measures_classif.tp

---

`mlr_measures_debug_classif`

Debug Measure for Classification

Description

This measure returns the number of observations in the PredictionClassif object. Its main purpose is debugging. The parameter `na_ratio` (`numeric(1)`) controls the ratio of scores which randomly are set to `NA`, between 0 (default) and 1.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("debug_classif")
msr("debug_classif")
```

Meta Information

- Task type: "NA"
- Range: $[0, \infty)$
- Minimize: NA
- Average: macro
- Required Prediction: "response"
- Required Packages: **mlr3**

Parameters

Id	Type	Default	Range
na_ratio	numeric	-	$[0, 1]$

Super class

`mlr3::Measure` -> `MeasureDebugClassif`

Methods

Public methods:

- `MeasureDebugClassif$new()`
- `MeasureDebugClassif$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
MeasureDebugClassif$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-eval
- Package **mlr3measures** for the scoring functions. Dictionary of Measures: mlr_measures `as.data.table(mlr_measures)` for a table of available Measures in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other Measure: `Measure`, `MeasureClassif`, `MeasureRegr`, `MeasureSimilarity`, `mlr_measures`, `mlr_measures_aic`, `mlr_measures_bic`, `mlr_measures_classif.costs`, `mlr_measures_elapsed_time`, `mlr_measures_internal_valid_score`, `mlr_measures_oob_error`, `mlr_measures_regr.pinball`, `mlr_measures_regr.rqr`, `mlr_measures_regr.rsq`, `mlr_measures_selected_features`

Examples

```
task = tsk("wine")
learner = lrn("classif.featureless")
measure = msr("debug_classif", na_ratio = 0.5)
rr = resample(task, learner, rsmp("cv", folds = 5))
rr$score(measure)
```

`mlr_measures_elapsed_time` *Elapsed Time Measure*

Description

Measures the elapsed time during train ("time_train"), predict ("time_predict"), or both ("time_both"). Aggregation of elapsed time defaults to mean but can be configured via the field `aggregator` of the Measure.

When predictions for multiple predict sets were made during `resample()` or `benchmark()`, the predict time shows the cumulative duration of all predictions. If `learner$predict()` is called manually, the last predict time gets overwritten. The elapsed time accounts only for the training duration of the primary learner, excluding the time required for training the fallback learner.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("time_train")
msr("time_train")
```

Meta Information

- Task type: "NA"
- Range: $[0, \infty)$
- Minimize: TRUE
- Average: macro
- Required Prediction: "NA"
- Required Packages: **mlr3**

Parameters

Empty ParamSet

Super class

`mlr3::Measure` -> `MeasureElapsedTime`

Public fields

`stages` (`character()`)
Which stages of the learner to measure? Usually set during construction.

Methods

Public methods:

- `MeasureElapsedTime$new()`
- `MeasureElapsedTime$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
MeasureElapsedTime$new(id = "elapsed_time", stages)
```

Arguments:

`id (character(1))`
Identifier for the new instance.

`stages (character())`
Subset of `("train", "predict")`. The runtime of provided stages will be summed.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-eval
- Package **mlr3measures** for the scoring functions. Dictionary of Measures: mlr_measures `as.data.table(mlr_measures)` for a table of available Measures in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other Measure: `Measure`, `MeasureClassif`, `MeasureRegr`, `MeasureSimilarity`, `mlr_measures`, `mlr_measures_aic`, `mlr_measures_bic`, `mlr_measures_classif.costs`, `mlr_measures_debug_classif`, `mlr_measures_internal_valid_score`, `mlr_measures_oob_error`, `mlr_measures_regr.pinball`, `mlr_measures_regr.rqr`, `mlr_measures_regr.rsq`, `mlr_measures_selected_features`

`mlr_measures_internal_valid_score` *Measure Internal Validation Score*

Description

Returns the selected internal validation score of the Learner for learners with property `"validation"`. Returns `NA` for unsupported learners, when no validation was done, or when the selected id was not found. The `id` of this measure is set to the value of `select` if provided.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("internal_valid_score")
msr("internal_valid_score")
```

Meta Information

- Task type: "NA"
- Range: $(-\infty, \infty)$
- Minimize: NA
- Average: macro
- Required Prediction: "NA"
- Required Packages: **mlr3**

Parameters

Empty ParamSet

Super class

`mlr3::Measure` -> `MeasureInternalValidScore`

Methods

Public methods:

- `MeasureInternalValidScore$new()`
- `MeasureInternalValidScore$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
MeasureInternalValidScore$new(select = NULL, minimize = NA)
```

Arguments:

`select (character(1))`
: Which of the internal validation scores to select. Which scores are available depends on the learner and its configuration. By default, the first score is chosen.

`minimize (logical(1))`
: Whether smaller values are better. Must be set to use for tuning.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-eval
- Package **mlr3measures** for the scoring functions. Dictionary of Measures: mlr_measures `as.data.table(mlr_measures)` for a table of available Measures in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other Measure: `Measure`, `MeasureClassif`, `MeasureRegr`, `MeasureSimilarity`, `mlr_measures`, `mlr_measures_aic`, `mlr_measures_bic`, `mlr_measures_classif.costs`, `mlr_measures_debug_classif`, `mlr_measures_elapsed_time`, `mlr_measures_oob_error`, `mlr_measures_regr.pinball`, `mlr_measures_regr.rqr`, `mlr_measures_regr.rsq`, `mlr_measures_selected_features`

Examples

```
rr = resample(tsk("iris"), lrn("classif.debug", validate = 0.3), rsmp("holdout"))
rr$score(msr("internal_valid_score", select = "acc"))
```

---

mlr_measures_oob_error

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.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

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

Meta Information

- Task type: "NA"
- Range: $(-\infty, \infty)$
- Minimize: TRUE
- Average: macro
- Required Prediction: "NA"
- Required Packages: **mlr3**

Parameters

Empty ParamSet

Super class

`mlr3::Measure` -> `MeasureOOBError`

Methods

Public methods:

- `MeasureOOBError$new()`
- `MeasureOOBError$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
MeasureOOBError$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-eval
- Package **mlr3measures** for the scoring functions. Dictionary of Measures: mlr_measures `as.data.table(mlr_measures)` for a table of available Measures in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other Measure: `Measure`, `MeasureClassif`, `MeasureRegr`, `MeasureSimilarity`, `mlr_measures`, `mlr_measures_aic`, `mlr_measures_bic`, `mlr_measures_classif.costs`, `mlr_measures_debug_classif`, `mlr_measures_elapsed_time`, `mlr_measures_internal_valid_score`, `mlr_measures_regr.pinball`, `mlr_measures_regr.rqr`, `mlr_measures_regr.rsq`, `mlr_measures_selected_features`

`mlr_measures_regr.bias` *Bias*

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

The Bias is defined as

$$\frac{1}{n} \sum_{i=1}^{n} w_i \left(r_i - t_i \right),$$

where w_i are normalized sample weights. Good predictions score close to 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.bias")
msr("regr.bias")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"regr"`
- Range: $(-\infty, \infty)$
- Minimize: `NA`
- Required prediction: `response`

Note

The score function calls `mlr3measures::bias()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: `mlr_measures_regr.ktau`, `mlr_measures_regr.mae`, `mlr_measures_regr.mape`, `mlr_measures_regr.maxae`, `mlr_measures_regr.medae`, `mlr_measures_regr.medse`, `mlr_measures_regr.mse`, `mlr_measures_regr.msle`, `mlr_measures_regr.pbias`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

---

mlr_measures_regr.ktau

Kendall's tau

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

Kendall's tau is defined as Kendall's rank correlation coefficient between truth and response. It is defined as

$$\tau = \frac{(\text{numberofconcordantpairs}) - (\text{numberofdiscordantpairs})}{(\text{numberofpairs})}$$

Calls `stats::cor()` with `method` set to `"kendall"`.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.ktau")
msr("regr.ktau")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"regr"`
- Range: $[-1, 1]$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::ktau()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: `mlr_measures_regr.bias`, `mlr_measures_regr.mae`, `mlr_measures_regr.mape`, `mlr_measures_regr.maxae`, `mlr_measures_regr.medae`, `mlr_measures_regr.medse`, `mlr_measures_regr.mse`, `mlr_measures_regr.msle`, `mlr_measures_regr.pbias`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

`mlr_measures_regr.mae` *Mean Absolute Error*

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

The Mean Absolute Error is defined as

$$\frac{1}{n} \sum_{i=1}^{n} w_i \left| t_i - r_i \right|,$$

where w_i are normalized sample weights.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

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

Parameters

Empty ParamSet

Meta Information

- Type: "regr"
- Range: $[0, \infty)$
- Minimize: TRUE
- Required prediction: response

Note

The score function calls mlr3measures::mae() from package **mlr3measures**.

If the measure is undefined for the input, NaN is returned. This can be customized by setting the field na_value.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: mlr_measures_regr.bias, mlr_measures_regr.ktau, mlr_measures_regr.mape, mlr_measures_regr.maxae, mlr_measures_regr.medae, mlr_measures_regr.medse, mlr_measures_regr.mse, mlr_measures_regr.msle, mlr_measures_regr.pbias, mlr_measures_regr.rmse, mlr_measures_regr.rmsle, mlr_measures_regr.sae, mlr_measures_regr.smape, mlr_measures_regr.srho, mlr_measures_regr.sse

---

mlr_measures_regr.mape

Mean Absolute Percent Error

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

The Mean Absolute Percent Error is defined as

$$\frac{1}{n} \sum_{i=1}^{n} w_i \left| \frac{t_i - r_i}{t_i} \right|,$$

where w_i are normalized sample weights.

This measure is undefined if any element of t is 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function msr():

```
mlr_measures$get("regr.mape")
msr("regr.mape")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"regr"`
- Range: $[0, \infty)$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::mape()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: `mlr_measures_regr.bias`, `mlr_measures_regr.ktau`, `mlr_measures_regr.mae`, `mlr_measures_regr.maxae`, `mlr_measures_regr.medae`, `mlr_measures_regr.medse`, `mlr_measures_regr.mse`, `mlr_measures_regr.msle`, `mlr_measures_regr.pbias`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

`mlr_measures_regr.maxae`

Max Absolute Error

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

The Max Absolute Error is defined as

$$\max\left(|t_i - r_i|\right).$$

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.maxae")
msr("regr.maxae")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"regr"`
- Range: $[0, \infty)$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::maxae()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: `mlr_measures_regr.bias`, `mlr_measures_regr.ktau`, `mlr_measures_regr.mae`, `mlr_measures_regr.mape`, `mlr_measures_regr.medae`, `mlr_measures_regr.medse`, `mlr_measures_regr.mse`, `mlr_measures_regr.msle`, `mlr_measures_regr.pbias`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

`mlr_measures_regr.medae`

Median Absolute Error

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

The Median Absolute Error is defined as

$$\text{median}\,|t_i - r_i|\,.$$

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.medae")
msr("regr.medae")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"regr"`
- Range: $[0, \infty)$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::medae()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: `mlr_measures_regr.bias`, `mlr_measures_regr.ktau`, `mlr_measures_regr.mae`, `mlr_measures_regr.mape`, `mlr_measures_regr.maxae`, `mlr_measures_regr.medse`, `mlr_measures_regr.mse`, `mlr_measures_regr.msle`, `mlr_measures_regr.pbias`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

`mlr_measures_regr.medse`

Median Squared Error

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

The Median Squared Error is defined as

$$\text{median}\left[(t_i - r_i)^2\right].$$

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.medse")
msr("regr.medse")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"regr"`
- Range: $[0, \infty)$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::medse()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: `mlr_measures_regr.bias`, `mlr_measures_regr.ktau`, `mlr_measures_regr.mae`, `mlr_measures_regr.mape`, `mlr_measures_regr.maxae`, `mlr_measures_regr.medae`, `mlr_measures_regr.mse`, `mlr_measures_regr.msle`, `mlr_measures_regr.pbias`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

`mlr_measures_regr.mse` *Mean Squared Error*

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

The Mean Squared Error is defined as

$$\frac{1}{n} \sum_{i=1}^{n} w_i \left(t_i - r_i\right)^2,$$

where w_i are normalized sample weights.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

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

Parameters

Empty ParamSet

Meta Information

- Type: `"regr"`
- Range: $[0, \infty)$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::mse()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: `mlr_measures_regr.bias`, `mlr_measures_regr.ktau`, `mlr_measures_regr.mae`, `mlr_measures_regr.mape`, `mlr_measures_regr.maxae`, `mlr_measures_regr.medae`, `mlr_measures_regr.medse`, `mlr_measures_regr.msle`, `mlr_measures_regr.pbias`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

`mlr_measures_regr.msle`

Mean Squared Log Error

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

The Mean Squared Log Error is defined as

$$\frac{1}{n} \sum_{i=1}^{n} w_i \left(\ln(1 + t_i) - \ln(1 + r_i) \right)^2 ,$$

where w_i are normalized sample weights. This measure is undefined if any element of t or r is less than or equal to -1.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.msle")
msr("regr.msle")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"regr"`
- Range: $[0, \infty)$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::msle()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: `mlr_measures_regr.bias`, `mlr_measures_regr.ktau`, `mlr_measures_regr.mae`, `mlr_measures_regr.mape`, `mlr_measures_regr.maxae`, `mlr_measures_regr.medae`, `mlr_measures_regr.medse`, `mlr_measures_regr.mse`, `mlr_measures_regr.pbias`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

`mlr_measures_regr.pbias` *Percent Bias*

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

The Percent Bias is defined as

$$\frac{1}{n} \sum_{i=1}^{n} w_i \frac{(r_i - t_i)}{|t_i|},$$

where w_i are normalized sample weights. Good predictions score close to 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.pbias")
msr("regr.pbias")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"regr"`
- Range: $(-\infty, \infty)$
- Minimize: `NA`
- Required prediction: `response`

Note

The score function calls `mlr3measures::pbias()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: `mlr_measures_regr.bias`, `mlr_measures_regr.ktau`, `mlr_measures_regr.mae`, `mlr_measures_regr.mape`, `mlr_measures_regr.maxae`, `mlr_measures_regr.medae`, `mlr_measures_regr.medse`, `mlr_measures_regr.mse`, `mlr_measures_regr.msle`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

`mlr_measures_regr.pinball`

Average Pinball Loss

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

The pinball loss for quantile regression is defined as

$$\text{Average Pinball Loss} = \frac{1}{n}\sum_{i=1}^{n} w_i \begin{cases} q \cdot (t_i - r_i) & \text{if } t_i \geq r_i \\ (1-q) \cdot (r_i - t_i) & \text{if } t_i < r_i \end{cases}$$

where q is the quantile and w_i are normalized sample weights.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.pinball")
msr("regr.pinball")
```

Meta Information

- Task type: "regr"
- Range: $(-\infty, \infty)$
- Minimize: TRUE
- Average: macro
- Required Prediction: "quantiles"
- Required Packages: **mlr3**

Parameters

Id	Type	Default	Range
alpha	numeric	-	$[0, 1]$

Super classes

`mlr3::Measure` -> `mlr3::MeasureRegr` -> `MeasureRegrPinball`

Methods

Public methods:

- `MeasureRegrPinball$new()`
- `MeasureRegrPinball$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
MeasureRegrPinball$new(alpha = 0.5)
```

Arguments:

`alpha numeric(1)`
The quantile to compute the pinball loss. Must be one of the quantiles that the Learner was trained on.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-eval
- Package **mlr3measures** for the scoring functions. Dictionary of Measures: mlr_measures `as.data.table(mlr_measures)` for a table of available Measures in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other Measure: `Measure`, `MeasureClassif`, `MeasureRegr`, `MeasureSimilarity`, `mlr_measures`, `mlr_measures_aic`, `mlr_measures_bic`, `mlr_measures_classif.costs`, `mlr_measures_debug_classif`, `mlr_measures_elapsed_time`, `mlr_measures_internal_valid_score`, `mlr_measures_oob_error`, `mlr_measures_regr.rqr`, `mlr_measures_regr.rsq`, `mlr_measures_selected_features`

`mlr_measures_regr.rmse` *Root Mean Squared Error*

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

The Root Mean Squared Error is defined as

$$\sqrt{\frac{1}{n} \sum_{i=1}^{n} w_i \left(t_i - r_i\right)^2},$$

where w_i are normalized sample weights.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

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

Parameters

Empty ParamSet

Meta Information

- Type: `"regr"`
- Range: $[0, \infty)$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::rmse()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: `mlr_measures_regr.bias`, `mlr_measures_regr.ktau`, `mlr_measures_regr.mae`, `mlr_measures_regr.mape`, `mlr_measures_regr.maxae`, `mlr_measures_regr.medae`, `mlr_measures_regr.medse`, `mlr_measures_regr.mse`, `mlr_measures_regr.msle`, `mlr_measures_regr.pbias`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

`mlr_measures_regr.rmsle`

Root Mean Squared Log Error

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

The Root Mean Squared Log Error is defined as

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n} w_i \left(\ln(1+t_i) - \ln(1+r_i)\right)^2},$$

where w_i are normalized sample weights.

This measure is undefined if any element of t or r is less than or equal to -1.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.rmsle")
msr("regr.rmsle")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"regr"`
- Range: $[0, \infty)$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::rmsle()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: `mlr_measures_regr.bias`, `mlr_measures_regr.ktau`, `mlr_measures_regr.mae`, `mlr_measures_regr.mape`, `mlr_measures_regr.maxae`, `mlr_measures_regr.medae`, `mlr_measures_regr.medse`, `mlr_measures_regr.mse`, `mlr_measures_regr.msle`, `mlr_measures_regr.pbias`, `mlr_measures_regr.rmse`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

`mlr_measures_regr.rqr` *R-Squared for Quantile Regression*

Description

Measure to compare true observed response with predicted quantiles in regression tasks.

Details

$R^1(\alpha)$ is defined as

$$1 - \frac{\sum_{i=1}^{n} \rho_\alpha \left(t_i - r_i(\alpha)\right)}{\sum_{i=1}^{n} \rho_\alpha \left(t_i - q_\alpha\right)},$$

where for a quantile α, ρ_α is the pinball function, $r_i(\alpha)$ are the predictions for the quantile and q_α is the empirical α-quantile of the test or training data.

$R^1(\alpha)$ is analogous to R^2 for regression tasks. It compares the pinball function of the predictions relative to a naive model predicting the empirical quantile.

This measure is undefined for constant t.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.rqr")
msr("regr.rqr")
```

Meta Information

- Task type: "regr"
- Range: $(-\infty, 1]$
- Minimize: FALSE
- Average: macro
- Required Prediction: "quantiles"
- Required Packages: **mlr3**

Parameters

Id	Type	Default	Range
alpha	numeric	-	$[0, 1]$

Super classes

`mlr3::Measure` -> `mlr3::MeasureRegr` -> `MeasureRQR`

Methods

Public methods:

- `MeasureRegrRQR$new()`
- `MeasureRegrRQR$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
MeasureRegrRQR$new(alpha = 0.5, pred_set_mean = TRUE)
```

Arguments:

`alpha numeric(1)`
: The quantile for which to compute the measure. Must be one of the quantiles that the Learner was trained on.

`pred_set_mean logical(1)`
: If `TRUE`, the mean of the true values is calculated on the prediction set. If `FALSE`, the mean of the true values is calculated on the training set.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Koenker, Roger, Machado, F. JA (1999). "Goodness of Fit and Related Inference Processes for Quantile Regression." *Journal of the American Statistical Association*, **94**(448), 1296–1310. doi:10.1080/01621459.1999.10473882.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-eval
- Package **mlr3measures** for the scoring functions. Dictionary of Measures: mlr_measures `as.data.table(mlr_measures)` for a table of available Measures in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other Measure: `Measure`, `MeasureClassif`, `MeasureRegr`, `MeasureSimilarity`, `mlr_measures`, `mlr_measures_aic`, `mlr_measures_bic`, `mlr_measures_classif.costs`, `mlr_measures_debug_classif`, `mlr_measures_elapsed_time`, `mlr_measures_internal_valid_score`, `mlr_measures_oob_error`, `mlr_measures_regr.pinball`, `mlr_measures_regr.rsq`, `mlr_measures_selected_features`

`mlr_measures_regr.rsq` *R-Squared*

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

R Squared is defined as

$$1 - \frac{\sum_{i=1}^{n} w_i \left(t_i - r_i\right)^2}{\sum_{i=1}^{n} w_i \left(t_i - \bar{t}\right)^2},$$

where $\bar{t} = \frac{1}{n} \sum_{i=1}^{n} t_i$ and w_i are weights.

Also known as coefficient of determination or explained variation. It compares the squared error of the predictions relative to a naive model predicting the mean.

Note that weights are used to scale the squared error of individual predictions (both in the numerator and in the denominator), but the "plug in" value $\bar{t}$ is computed without weights.

This measure is undefined for constant t.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.rsq")
msr("regr.rsq")
```

Meta Information

- Task type: "regr"
- Range: $(-\infty, 1]$
- Minimize: FALSE
- Average: macro
- Required Prediction: "response"
- Required Packages: **mlr3**

Parameters

Empty ParamSet

Super classes

`mlr3::Measure` -> `mlr3::MeasureRegr` -> `MeasureRSQ`

Methods

Public methods:

- `MeasureRegrRSQ$new()`
- `MeasureRegrRSQ$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
MeasureRegrRSQ$new(pred_set_mean = TRUE)
```

Arguments:

`pred_set_mean logical(1)`
If `TRUE`, the mean of the true values is calculated on the prediction set. If `FALSE`, the mean of the true values is calculated on the training set.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-eval
- Package **mlr3measures** for the scoring functions. Dictionary of Measures: mlr_measures `as.data.table(mlr_measures)` for a table of available Measures in the running session (depending on the loaded packages).
- Extension packages for additional task types:

- **mlr3proba** for probabilistic supervised regression and survival analysis.
- **mlr3cluster** for unsupervised clustering.

Other Measure: Measure, MeasureClassif, MeasureRegr, MeasureSimilarity, mlr_measures, mlr_measures_aic, mlr_measures_bic, mlr_measures_classif.costs, mlr_measures_debug_classif, mlr_measures_elapsed_time, mlr_measures_internal_valid_score, mlr_measures_oob_error, mlr_measures_regr.pinball, mlr_measures_regr.rqr, mlr_measures_selected_features

---

`mlr_measures_regr.sae` *Sum of Absolute Errors*

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

The Sum of Absolute Errors is defined as

$$\sum_{i=1}^{n} w_i \left| t_i - r_i \right| .$$

where w_i are unnormalized weights for each observation x_i, defaulting to 1.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.sae")
msr("regr.sae")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"regr"`
- Range: $[0, \infty)$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::sae()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: `mlr_measures_regr.bias`, `mlr_measures_regr.ktau`, `mlr_measures_regr.mae`, `mlr_measures_regr.mape`, `mlr_measures_regr.maxae`, `mlr_measures_regr.medae`, `mlr_measures_regr.medse`, `mlr_measures_regr.mse`, `mlr_measures_regr.msle`, `mlr_measures_regr.pbias`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

`mlr_measures_regr.smape`

Symmetric Mean Absolute Percent Error

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

The Symmetric Mean Absolute Percent Error is defined as

$$\frac{2}{n} \sum_{i=1}^{n} \frac{|t_i - r_i|}{|t_i| + |r_i|}.$$

This measure is undefined if if any $|t| + |r|$ is equal to 0.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.smape")
msr("regr.smape")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"regr"`
- Range: $[0, 2]$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::smape()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: `mlr_measures_regr.bias`, `mlr_measures_regr.ktau`, `mlr_measures_regr.mae`, `mlr_measures_regr.mape`, `mlr_measures_regr.maxae`, `mlr_measures_regr.medae`, `mlr_measures_regr.medse`, `mlr_measures_regr.mse`, `mlr_measures_regr.msle`, `mlr_measures_regr.pbias`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.sae`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

`mlr_measures_regr.srho`

Spearman's rho

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

Spearman's rho is defined as Spearman's rank correlation coefficient between truth and response. Calls `stats::cor()` with `method` set to `"spearman"`.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.srho")
msr("regr.srho")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"regr"`
- Range: $[-1, 1]$
- Minimize: `FALSE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::srho()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: `mlr_measures_regr.bias`, `mlr_measures_regr.ktau`, `mlr_measures_regr.mae`, `mlr_measures_regr.mape`, `mlr_measures_regr.maxae`, `mlr_measures_regr.medae`, `mlr_measures_regr.medse`, `mlr_measures_regr.mse`, `mlr_measures_regr.msle`, `mlr_measures_regr.pbias`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.sse`

`mlr_measures_regr.sse` *Sum of Squared Errors*

Description

Measure to compare true observed response with predicted response in regression tasks.

Details

The Sum of Squared Errors is defined as

$$\sum_{i=1}^{n} w_i \left(t_i - r_i\right)^2 .$$

where w_i are unnormalized weights for each observation x_i, defaulting to 1.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.sse")
msr("regr.sse")
```

Parameters

Empty ParamSet

Meta Information

- Type: `"regr"`
- Range: $[0, \infty)$
- Minimize: `TRUE`
- Required prediction: `response`

Note

The score function calls `mlr3measures::sse()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other regression measures: `mlr_measures_regr.bias`, `mlr_measures_regr.ktau`, `mlr_measures_regr.mae`, `mlr_measures_regr.mape`, `mlr_measures_regr.maxae`, `mlr_measures_regr.medae`, `mlr_measures_regr.medse`, `mlr_measures_regr.mse`, `mlr_measures_regr.msle`, `mlr_measures_regr.pbias`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`

`mlr_measures_selected_features`

Selected Features Measure

Description

Measures the number of selected features by extracting it from learners with property `"selected_features"`. If parameter `normalize` is set to `TRUE`, the relative number of features instead of the absolute number of features is returned. Note that the models must be stored to be able to extract this information. If the learner does not support the extraction of used features, `NA` is returned.

This measure requires the Task and the Learner for scoring.

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("selected_features")
msr("selected_features")
```

Meta Information

- Task type: "NA"
- Range: $[0, \infty)$
- Minimize: TRUE
- Average: macro
- Required Prediction: "NA"
- Required Packages: **mlr3**

Parameters

Id	Type	Default	Levels
normalize	logical	-	TRUE, FALSE

Super class

`mlr3::Measure` -> `MeasureSelectedFeatures`

Methods

Public methods:

- `MeasureSelectedFeatures$new()`
- `MeasureSelectedFeatures$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
MeasureSelectedFeatures$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-eval
- Package **mlr3measures** for the scoring functions. Dictionary of Measures: mlr_measures `as.data.table(mlr_measures)` for a table of available Measures in the running session (depending on the loaded packages).
- Extension packages for additional task types:

- **mlr3proba** for probabilistic supervised regression and survival analysis.
- **mlr3cluster** for unsupervised clustering.

Other Measure: Measure, MeasureClassif, MeasureRegr, MeasureSimilarity, mlr_measures, mlr_measures_aic, mlr_measures_bic, mlr_measures_classif.costs, mlr_measures_debug_classif, mlr_measures_elapsed_time, mlr_measures_internal_valid_score, mlr_measures_oob_error, mlr_measures_regr.pinball, mlr_measures_regr.rqr, mlr_measures_regr.rsq

Examples

```
task = tsk("german_credit")
learner = lrn("classif.rpart")
rr = resample(task, learner, rsmp("cv", folds = 3), store_models = TRUE)

scores = rr$score(msr("selected_features"))
scores[, c("iteration", "selected_features")]
```

---

mlr_measures_sim.jaccard

Jaccard Similarity Index

Description

Measure to compare two or more sets w.r.t. their similarity.

Details

For two sets A and B, the Jaccard Index is defined as

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

If more than two sets are provided, the mean of all pairwise scores is calculated.

This measure is undefined if two or more sets are empty.

Parameters

Empty ParamSet

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function msr():

```
mlr_measures$get("sim.jaccard")
msr("sim.jaccard")
```

Meta Information

- Type: `"similarity"`
- Range: $[0, 1]$
- Minimize: `FALSE`

Note

This measure requires learners with property `"selected_features"`. The extracted feature sets are passed to `mlr3measures::jaccard()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other similarity measures: `mlr_measures_sim.phi`

`mlr_measures_sim.phi` *Phi Coefficient Similarity*

Description

Measure to compare two or more sets w.r.t. their similarity.

Details

The Phi Coefficient is defined as the Pearson correlation between the binary representation of two sets A and B. The binary representation for A is a logical vector of length p with the i-th element being 1 if the corresponding element is in A, and 0 otherwise.

If more than two sets are provided, the mean of all pairwise scores is calculated.

This measure is undefined if one set contains none or all possible elements.

Parameters

Id	Type	Default	Range
p	integer	-	$[1, \infty)$

Dictionary

This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function `msr()`:

```
mlr_measures$get("sim.phi")
msr("sim.phi")
```

Meta Information

- Type: `"similarity"`
- Range: $[-1, 1]$
- Minimize: `FALSE`

Note

This measure requires learners with property `"selected_features"`. The extracted feature sets are passed to `mlr3measures::phi()` from package **mlr3measures**.

If the measure is undefined for the input, `NaN` is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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

Other similarity measures: `mlr_measures_sim.jaccard`

`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()`/`rsmps()`.

Format

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

Methods

See mlr3misc::Dictionary.

S3 methods

- `as.data.table(dict, ..., objects = FALSE)`
  mlr3misc::Dictionary -> `data.table::data.table()`
  Returns a `data.table::data.table()` with columns "key", "label", "params", and "iters". If `objects` is set to `TRUE`, the constructed objects are returned in the list column named `object`.

See Also

Sugar functions: `rsmp()`, `rsmps()`

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

Other Resampling: `Resampling`, `mlr_resamplings_bootstrap`, `mlr_resamplings_custom`, `mlr_resamplings_custom_c`
`mlr_resamplings_cv`, `mlr_resamplings_holdout`, `mlr_resamplings_insample`, `mlr_resamplings_loo`,
`mlr_resamplings_repeated_cv`, `mlr_resamplings_subsampling`

Examples

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

---

mlr_resamplings_bootstrap

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.

Dictionary

This Resampling can be instantiated via the dictionary mlr_resamplings or with the associated sugar function `rsmp()`:

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

Parameters

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

Super class

mlr3::Resampling -> ResamplingBootstrap

Active bindings

iters (integer(1))
: Returns the number of resampling iterations, depending on the values stored in the param_set.

Methods

Public methods:

- ResamplingBootstrap$new()
- ResamplingBootstrap$clone()

Method new(): Creates a new instance of this R6 class.

Usage:

```
ResamplingBootstrap$new()
```

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

References

Bischl B, Mersmann O, Trautmann H, Weihs C (2012). "Resampling Methods for Meta-Model Validation with Recommendations for Evolutionary Computation." *Evolutionary Computation*, **20**(2), 249–275. doi:10.1162/evco_a_00069.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-resampling
- Package **mlr3spatiotempcv** for spatio-temporal resamplings.
- Dictionary of Resamplings: mlr_resamplings
- as.data.table(mlr_resamplings) for a table of available Resamplings in the running session (depending on the loaded packages).
- **mlr3spatiotempcv** for additional Resamplings for spatio-temporal tasks.

Other Resampling: Resampling, mlr_resamplings, mlr_resamplings_custom, mlr_resamplings_custom_cv, mlr_resamplings_cv, mlr_resamplings_holdout, mlr_resamplings_insample, mlr_resamplings_loo, mlr_resamplings_repeated_cv, mlr_resamplings_subsampling

Examples

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

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

# Individual sets:
bootstrap$train_set(1)
bootstrap$test_set(1)

# Disjunct sets:
intersect(bootstrap$train_set(1), bootstrap$test_set(1))

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

---

mlr_resamplings_custom

Custom Resampling

Description

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

Dictionary

This Resampling can be instantiated via the dictionary mlr_resamplings or with the associated sugar function `rsmp()`:

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

Super class

`mlr3::Resampling` -> `ResamplingCustom`

Active bindings

`iters (integer(1))`
Returns the number of resampling iterations, depending on the values stored in the `param_set`.

Methods

Public methods:

- `ResamplingCustom$new()`
- `ResamplingCustom$instantiate()`
- `ResamplingCustom$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
ResamplingCustom$new()
```

Method `instantiate()`**:** Instantiate this Resampling with custom splits into training and test set.

Usage:

```
ResamplingCustom$instantiate(task, train_sets, test_sets)
```

Arguments:

`task` Task
: Mainly used to check if `train_sets` and `test_sets` are feasible.

`train_sets` (list of `integer()`)
: List with row ids for training, one list element per iteration. Must have the same length as `test_sets`.

`test_sets` (list of `integer()`)
: List with row ids for testing, one list element per iteration. Must have the same length as `train_sets`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: `https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-resampling`
- Package **mlr3spatiotempcv** for spatio-temporal resamplings.
- Dictionary of Resamplings: mlr_resamplings
- `as.data.table(mlr_resamplings)` for a table of available Resamplings in the running session (depending on the loaded packages).
- **mlr3spatiotempcv** for additional Resamplings for spatio-temporal tasks.

Other Resampling: `Resampling`, `mlr_resamplings`, `mlr_resamplings_bootstrap`, `mlr_resamplings_custom_cv`, `mlr_resamplings_cv`, `mlr_resamplings_holdout`, `mlr_resamplings_insample`, `mlr_resamplings_loo`, `mlr_resamplings_repeated_cv`, `mlr_resamplings_subsampling`

Examples

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

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

custom$train_set(1)
custom$test_set(1)
```

---

mlr_resamplings_custom_cv

Custom Cross-Validation

Description

Splits data into training and test sets in a cross-validation fashion based on a user-provided categorical vector. This vector can be passed during instantiation either via an arbitrary factor `f` with the same length as `task$nrow`, or via a single string `col` referring to a column in the task.

An alternative but equivalent approach using leave-one-out resampling is showcased in the examples of mlr_resamplings_loo.

Dictionary

This Resampling can be instantiated via the dictionary mlr_resamplings or with the associated sugar function `rsmp()`:

```
mlr_resamplings$get("custom_cv")
rsmp("custom_cv")
```

Super class

`mlr3::Resampling` -> `ResamplingCustomCV`

Active bindings

`iters (integer(1))`
: Returns the number of resampling iterations, depending on the values stored in the `param_set`.

Methods

Public methods:

- `ResamplingCustomCV$new()`
- `ResamplingCustomCV$instantiate()`
- `ResamplingCustomCV$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
ResamplingCustomCV$new()
```

Method `instantiate()`**:** Instantiate this Resampling as cross-validation with custom splits.

Usage:

```
ResamplingCustomCV$instantiate(task, f = NULL, col = NULL)
```

Arguments:

`task` Task
: Used to extract row ids.

`f` (`factor()` | `character()`)
: Vector of type factor or character with the same length as `task$nrow`. Row ids are split on this vector, each distinct value results in a fold. Empty factor levels are dropped and row ids corresponding to missing values are removed, c.f. `split()`.

`col` (`character(1)`)
: Name of the task column to use for splitting. Alternative and mutually exclusive to providing the factor levels as a vector via parameter `f`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-resampling
- Package **mlr3spatiotempcv** for spatio-temporal resamplings.
- Dictionary of Resamplings: mlr_resamplings
- `as.data.table(mlr_resamplings)` for a table of available Resamplings in the running session (depending on the loaded packages).
- **mlr3spatiotempcv** for additional Resamplings for spatio-temporal tasks.

Other Resampling: `Resampling`, `mlr_resamplings`, `mlr_resamplings_bootstrap`, `mlr_resamplings_custom`, `mlr_resamplings_cv`, `mlr_resamplings_holdout`, `mlr_resamplings_insample`, `mlr_resamplings_loo`, `mlr_resamplings_repeated_cv`, `mlr_resamplings_subsampling`

Examples

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

# Instantiate Resampling:
custom_cv = rsmp("custom_cv")
f = factor(c(rep(letters[1:3], each = 3), NA))
custom_cv$instantiate(task, f = f)
custom_cv$iters # 3 folds

# Individual sets:
custom_cv$train_set(1)
custom_cv$test_set(1)

# Disjunct sets:
intersect(custom_cv$train_set(1), custom_cv$test_set(1))
```

---

mlr_resamplings_cv *Cross-Validation Resampling*

Description

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

Dictionary

This Resampling can be instantiated via the dictionary mlr_resamplings or with the associated sugar function `rsmp()`:

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

Parameters

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

Super class

`mlr3::Resampling` -> `ResamplingCV`

Active bindings

`iters` (`integer(1)`)
Returns the number of resampling iterations, depending on the values stored in the `param_set`.

Methods

Public methods:

- `ResamplingCV$new()`
- `ResamplingCV$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
ResamplingCV$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Bischl B, Mersmann O, Trautmann H, Weihs C (2012). "Resampling Methods for Meta-Model Validation with Recommendations for Evolutionary Computation." *Evolutionary Computation*, **20**(2), 249–275. doi:10.1162/evco_a_00069.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-resampling
- Package **mlr3spatiotempcv** for spatio-temporal resamplings.
- Dictionary of Resamplings: mlr_resamplings
- `as.data.table(mlr_resamplings)` for a table of available Resamplings in the running session (depending on the loaded packages).
- **mlr3spatiotempcv** for additional Resamplings for spatio-temporal tasks.

Other Resampling: `Resampling`, `mlr_resamplings`, `mlr_resamplings_bootstrap`, `mlr_resamplings_custom`, `mlr_resamplings_custom_cv`, `mlr_resamplings_holdout`, `mlr_resamplings_insample`, `mlr_resamplings_loo`, `mlr_resamplings_repeated_cv`, `mlr_resamplings_subsampling`

Examples

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

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

# Individual sets:
cv$train_set(1)
```

```
cv$test_set(1)

# Disjunct sets:
intersect(cv$train_set(1), cv$test_set(1))

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

`mlr_resamplings_holdout` *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).

Dictionary

This Resampling can be instantiated via the dictionary mlr_resamplings or with the associated sugar function `rsmp()`:

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

Parameters

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

Super class

`mlr3::Resampling` -> `ResamplingHoldout`

Active bindings

`iters` (`integer(1)`)
Returns the number of resampling iterations, depending on the values stored in the `param_set`.

Methods

Public methods:

- `ResamplingHoldout$new()`
- `ResamplingHoldout$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
ResamplingHoldout$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Bischl B, Mersmann O, Trautmann H, Weihs C (2012). "Resampling Methods for Meta-Model Validation with Recommendations for Evolutionary Computation." *Evolutionary Computation*, **20**(2), 249–275. doi:10.1162/evco_a_00069.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-resampling
- Package **mlr3spatiotempcv** for spatio-temporal resamplings.
- Dictionary of Resamplings: mlr_resamplings
- `as.data.table(mlr_resamplings)` for a table of available Resamplings in the running session (depending on the loaded packages).
- **mlr3spatiotempcv** for additional Resamplings for spatio-temporal tasks.

Other Resampling: `Resampling`, `mlr_resamplings`, `mlr_resamplings_bootstrap`, `mlr_resamplings_custom`, `mlr_resamplings_custom_cv`, `mlr_resamplings_cv`, `mlr_resamplings_insample`, `mlr_resamplings_loo`, `mlr_resamplings_repeated_cv`, `mlr_resamplings_subsampling`

Examples

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

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

# Individual sets:
holdout$train_set(1)
holdout$test_set(1)

# Disjunct sets:
intersect(holdout$train_set(1), holdout$test_set(1))

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

`mlr_resamplings_insample` *Insample Resampling*

Description

Uses all observations as training and as test set.

Dictionary

This Resampling can be instantiated via the dictionary mlr_resamplings or with the associated sugar function `rsmp()`:

```
mlr_resamplings$get("insample")
rsmp("insample")
```

Super class

`mlr3::Resampling` -> `ResamplingInsample`

Active bindings

`iters (integer(1))`
Returns the number of resampling iterations, depending on the values stored in the `param_set`.

Methods

Public methods:

- `ResamplingInsample$new()`
- `ResamplingInsample$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
ResamplingInsample$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-resampling
- Package **mlr3spatiotempcv** for spatio-temporal resamplings.
- Dictionary of Resamplings: mlr_resamplings
- `as.data.table(mlr_resamplings)` for a table of available Resamplings in the running session (depending on the loaded packages).
- **mlr3spatiotempcv** for additional Resamplings for spatio-temporal tasks.

Other Resampling: `Resampling`, `mlr_resamplings`, `mlr_resamplings_bootstrap`, `mlr_resamplings_custom`, `mlr_resamplings_custom_cv`, `mlr_resamplings_cv`, `mlr_resamplings_holdout`, `mlr_resamplings_loo`, `mlr_resamplings_repeated_cv`, `mlr_resamplings_subsampling`

Examples

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

# Instantiate Resampling
insample = rsmp("insample")
insample$instantiate(task)

# Train set equal to test set:
setequal(insample$train_set(1), insample$test_set(1))

# Internal storage:
insample$instance # just row ids
```

---

mlr_resamplings_loo *Leave-One-Out Cross-Validation*

Description

Splits data using leave-one-observation-out. This is identical to cross-validation with the number of folds set to the number of observations.

If this resampling is combined with the grouping features of tasks, it is possible to create custom splits based on an arbitrary factor variable, see the examples.

Dictionary

This Resampling can be instantiated via the dictionary mlr_resamplings or with the associated sugar function `rsmp()`:

```
mlr_resamplings$get("loo")
rsmp("loo")
```

Super class

`mlr3::Resampling` -> `ResamplingLOO`

Active bindings

`iters (integer(1))`
Returns the number of resampling iterations which is the number of rows of the task provided to instantiate. Is `NA` if the resampling has not been instantiated.

Methods

Public methods:

- `ResamplingLOO$new()`
- `ResamplingLOO$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
ResamplingLOO$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Bischl B, Mersmann O, Trautmann H, Weihs C (2012). "Resampling Methods for Meta-Model Validation with Recommendations for Evolutionary Computation." *Evolutionary Computation*, **20**(2), 249–275. doi:10.1162/evco_a_00069.

See Also

- Chapter in the mlr3book: `https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-resampling`
- Package **mlr3spatiotempcv** for spatio-temporal resamplings.
- Dictionary of Resamplings: mlr_resamplings
- `as.data.table(mlr_resamplings)` for a table of available Resamplings in the running session (depending on the loaded packages).
- **mlr3spatiotempcv** for additional Resamplings for spatio-temporal tasks.

Other Resampling: `Resampling`, `mlr_resamplings`, `mlr_resamplings_bootstrap`, `mlr_resamplings_custom`, `mlr_resamplings_custom_cv`, `mlr_resamplings_cv`, `mlr_resamplings_holdout`, `mlr_resamplings_insample`, `mlr_resamplings_repeated_cv`, `mlr_resamplings_subsampling`

Examples

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

# Instantiate Resampling
loo = rsmp("loo")
loo$instantiate(task)

# Individual sets:
loo$train_set(1)
loo$test_set(1)

# Disjunct sets:
intersect(loo$train_set(1), loo$test_set(1))

# Internal storage:
loo$instance # vector

# Combine with group feature of tasks:
task = tsk("penguins")
task$set_col_roles("island", add_to = "group")
loo$instantiate(task)
loo$iters # one fold for each level of "island"
```

---

mlr_resamplings_repeated_cv

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.

Iteration numbers can be translated into folds or repeats with provided methods.

Dictionary

This Resampling can be instantiated via the dictionary mlr_resamplings or with the associated sugar function `rsmp()`:

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

Parameters

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

Super class

`mlr3::Resampling` -> `ResamplingRepeatedCV`

Active bindings

`iters` (`integer(1)`)
Returns the number of resampling iterations, depending on the values stored in the `param_set`.

Methods

Public methods:

- `ResamplingRepeatedCV$new()`
- `ResamplingRepeatedCV$folds()`
- `ResamplingRepeatedCV$repeats()`
- `ResamplingRepeatedCV$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
ResamplingRepeatedCV$new()
```

Method `folds()`**:** Translates iteration numbers to fold numbers.

Usage:

```
ResamplingRepeatedCV$folds(iters)
```

Arguments:

`iters` (`integer()`)
Iteration number.

Returns: `integer()` of fold numbers.

Method `repeats()`**:** Translates iteration numbers to repetition numbers.

Usage:

```
ResamplingRepeatedCV$repeats(iters)
```

Arguments:

`iters` (`integer()`)
Iteration number.

Returns: `integer()` of repetition numbers.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Bischl B, Mersmann O, Trautmann H, Weihs C (2012). "Resampling Methods for Meta-Model Validation with Recommendations for Evolutionary Computation." *Evolutionary Computation*, **20**(2), 249–275. doi:10.1162/evco_a_00069.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-resampling
- Package **mlr3spatiotempcv** for spatio-temporal resamplings.
- Dictionary of Resamplings: mlr_resamplings
- `as.data.table(mlr_resamplings)` for a table of available Resamplings in the running session (depending on the loaded packages).
- **mlr3spatiotempcv** for additional Resamplings for spatio-temporal tasks.

Other Resampling: Resampling, mlr_resamplings, mlr_resamplings_bootstrap, mlr_resamplings_custom, mlr_resamplings_custom_cv, mlr_resamplings_cv, mlr_resamplings_holdout, mlr_resamplings_insample, mlr_resamplings_loo, mlr_resamplings_subsampling

Examples

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

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

# Individual sets:
repeated_cv$train_set(1)
repeated_cv$test_set(1)

# Disjunct sets:
intersect(repeated_cv$train_set(1), repeated_cv$test_set(1))

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

`mlr_resamplings_subsampling`

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.

Dictionary

This Resampling can be instantiated via the dictionary mlr_resamplings or with the associated sugar function `rsmp()`:

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

Parameters

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

Super class

`mlr3::Resampling` -> `ResamplingSubsampling`

Active bindings

`iters` (`integer(1)`)
Returns the number of resampling iterations, depending on the values stored in the `param_set`.

Methods

Public methods:

- `ResamplingSubsampling$new()`
- `ResamplingSubsampling$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
ResamplingSubsampling$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Bischl B, Mersmann O, Trautmann H, Weihs C (2012). "Resampling Methods for Meta-Model Validation with Recommendations for Evolutionary Computation." *Evolutionary Computation*, **20**(2), 249–275. doi:10.1162/evco_a_00069.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-resampling
- Package **mlr3spatiotempcv** for spatio-temporal resamplings.
- Dictionary of Resamplings: mlr_resamplings
- `as.data.table(mlr_resamplings)` for a table of available Resamplings in the running session (depending on the loaded packages).
- **mlr3spatiotempcv** for additional Resamplings for spatio-temporal tasks.

Other Resampling: `Resampling`, `mlr_resamplings`, `mlr_resamplings_bootstrap`, `mlr_resamplings_custom`, `mlr_resamplings_custom_cv`, `mlr_resamplings_cv`, `mlr_resamplings_holdout`, `mlr_resamplings_insample`, `mlr_resamplings_loo`, `mlr_resamplings_repeated_cv`

Examples

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

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

# Individual sets:
subsampling$train_set(1)
subsampling$test_set(1)

# Disjunct sets:
intersect(subsampling$train_set(1), subsampling$test_set(1))

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

`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.

Helper function to configure the `$validate` field(s) of a `Learner`.

This is especially useful for learners such as `AutoTuner` of **mlr3tuning** or `GraphLearner` of **mlr3pipelines** which have multiple levels of `$validate` fields., where the `$validate` fields need to be configured on multiple levels.

Usage

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

tsks(.keys, ...)

tgen(.key, ...)

tgens(.keys, ...)

lrn(.key, ...)

lrns(.keys, ...)

rsmp(.key, ...)

rsmps(.keys, ...)

msr(.key, ...)

msrs(.keys, ...)

set_validate(learner, validate, ...)
```

Arguments

`.key`	(`character(1)`) Key passed to the respective dictionary to retrieve the object.
`...`	(any) Additional arguments.
`.keys`	(`character()`) Keys passed to the respective dictionary to retrieve multiple objects.
`learner`	(any) The learner.
`validate`	(`numeric(1)`, `"predefined"`, `"test"`, or `NULL`) Which validation set to use.

Value

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

Modified `Learner`

Examples

```
# penguins task with new id
tsk("penguins", id = "penguins2")

# 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'
lrns(c("classif.featureless", "classif.rpart"), predict_type = "prob")
learner = lrn("classif.debug")
set_validate(learner, 0.2)
learner$validate
```

---

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, e.g. **mlr3data**, **mlr3proba** or **mlr3cluster**. **mlr3oml** allows to interact with OpenML.

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

Format

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

Methods

See mlr3misc::Dictionary.

S3 methods

- `as.data.table(dict, ..., objects = FALSE)`
  mlr3misc::Dictionary -> `data.table::data.table()`
  Returns a `data.table::data.table()` with columns "key", "label", "task_type", "nrow", "ncol", "properties", and the number of features of type "lgl", "int", "dbl", "chr", "fct" and "ord", respectively. If `objects` is set to `TRUE`, the constructed objects are returned in the list column named `object`.

See Also

Sugar functions: tsk(), tsks()

Extension Packages: **mlr3data**

Other Dictionary: mlr_learners, mlr_measures, mlr_resamplings, mlr_task_generators

Other Task: Task, TaskClassif, TaskRegr, TaskSupervised, TaskUnsupervised, california_housing, mlr_tasks_breast_cancer, mlr_tasks_german_credit, mlr_tasks_iris, mlr_tasks_mtcars, mlr_tasks_penguins, mlr_tasks_pima, mlr_tasks_sonar, mlr_tasks_spam, mlr_tasks_wine, mlr_tasks_zoo

Examples

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

# Add a new task, based on a subset of penguins:
data = palmerpenguins::penguins
data$species = factor(ifelse(data$species == "Adelie", "1", "0"))
task = TaskClassif$new("penguins.binary", data, target = "species", positive = "1")

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

# list available tasks
mlr_tasks$keys()

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

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

---

mlr_tasks_breast_cancer

Wisconsin Breast Cancer Classification Task

Description

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

- Column "Id" has been removed.
- Column names have been converted to snake_case.
- Positive class is set to "malignant".
- 16 incomplete cases have been removed from the data set.
- All factor features have been converted to ordered factors.

Format

R6::R6Class inheriting from TaskClassif.

Dictionary

This Task can be instantiated via the dictionary mlr_tasks or with the associated sugar function `tsk()`:

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

Meta Information

- Task type: "classif"
- Dimensions: 683x10
- Properties: "twoclass"
- Has Missings: `FALSE`
- Target: "class"
- Features: "bare_nuclei", "bl_cromatin", "cell_shape", "cell_size", "cl_thickness", "epith_c_size", "marg_adhesion", "mitoses", "normal_nucleoli"

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3data** for more toy tasks.
- Package **mlr3oml** for downloading tasks from https://www.openml.org.
- Package **mlr3viz** for some generic visualizations.
- Dictionary of Tasks: mlr_tasks
- `as.data.table(mlr_tasks)` for a table of available Tasks in the running session (depending on the loaded packages).
- **mlr3fselect** and **mlr3filters** for feature selection and feature filtering.
- Extension packages for additional task types:
  - Unsupervised clustering: **mlr3cluster**
  - Probabilistic supervised regression and survival analysis: https://mlr3proba.mlr-org.com/.

Other Task: `Task`, `TaskClassif`, `TaskRegr`, `TaskSupervised`, `TaskUnsupervised`, `california_housing`, `mlr_tasks`, `mlr_tasks_german_credit`, `mlr_tasks_iris`, `mlr_tasks_mtcars`, `mlr_tasks_penguins`, `mlr_tasks_pima`, `mlr_tasks_sonar`, `mlr_tasks_spam`, `mlr_tasks_wine`, `mlr_tasks_zoo`

`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.

Dictionary

This Task can be instantiated via the dictionary mlr_tasks or with the associated sugar function `tsk()`:

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

Meta Information

- Task type: “classif”
- Dimensions: 1000x21
- Properties: “twoclass”
- Has Missings: `FALSE`
- Target: “credit_risk”
- Features: “age”, “amount”, “credit_history”, “duration”, “employment_duration”, “foreign_worker”, “housing”, “installment_rate”, “job”, “number_credits”, “other_debtors”, “other_installment_plans”, “people_liable”, “personal_status_sex”, “present_residence”, “property”, “purpose”, “savings”, “status”, “telephone”

Source

Data set originally published on UCI. This is the preprocessed version taken from package **rchallenge** with factors instead of dummy variables, and corrected as proposed by Ulrike Grömping.

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

References

Grömping U (2019). "South German Credit Data: Correcting a Widely Used Data Set." Reports in Mathematics, Physics and Chemistry 4, Department II, Beuth University of Applied Sciences Berlin.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3data** for more toy tasks.
- Package **mlr3oml** for downloading tasks from https://www.openml.org.
- Package **mlr3viz** for some generic visualizations.
- Dictionary of Tasks: mlr_tasks
- `as.data.table(mlr_tasks)` for a table of available Tasks in the running session (depending on the loaded packages).
- **mlr3fselect** and **mlr3filters** for feature selection and feature filtering.
- Extension packages for additional task types:
  - Unsupervised clustering: **mlr3cluster**
  - Probabilistic supervised regression and survival analysis: https://mlr3proba.mlr-org.com/.

Other Task: Task, TaskClassif, TaskRegr, TaskSupervised, TaskUnsupervised, california_housing, mlr_tasks, mlr_tasks_breast_cancer, mlr_tasks_iris, mlr_tasks_mtcars, mlr_tasks_penguins, mlr_tasks_pima, mlr_tasks_sonar, mlr_tasks_spam, mlr_tasks_wine, mlr_tasks_zoo

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.

Dictionary

This Task can be instantiated via the dictionary mlr_tasks or with the associated sugar function `tsk()`:

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

Meta Information

- Task type: "classif"
- Dimensions: 150x5
- Properties: "multiclass"
- Has Missings: `FALSE`
- Target: "Species"
- Features: "Petal.Length", "Petal.Width", "Sepal.Length", "Sepal.Width"

Source

https://en.wikipedia.org/wiki/Iris_flower_data_set

Anderson E (1936). "The Species Problem in Iris." *Annals of the Missouri Botanical Garden*, **23**(3), 457. doi:10.2307/2394164.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3data** for more toy tasks.
- Package **mlr3oml** for downloading tasks from https://www.openml.org.
- Package **mlr3viz** for some generic visualizations.
- Dictionary of Tasks: mlr_tasks
- `as.data.table(mlr_tasks)` for a table of available Tasks in the running session (depending on the loaded packages).
- **mlr3fselect** and **mlr3filters** for feature selection and feature filtering.
- Extension packages for additional task types:
  - Unsupervised clustering: **mlr3cluster**
  - Probabilistic supervised regression and survival analysis: https://mlr3proba.mlr-org.com/.

Other Task: Task, TaskClassif, TaskRegr, TaskSupervised, TaskUnsupervised, california_housing, mlr_tasks, mlr_tasks_breast_cancer, mlr_tasks_german_credit, mlr_tasks_mtcars, mlr_tasks_penguins, mlr_tasks_pima, mlr_tasks_sonar, mlr_tasks_spam, mlr_tasks_wine, mlr_tasks_zoo

`mlr_tasks_mtcars` *Motor Trend Regression Task*

Description

A regression task for the datasets::mtcars data set. Target variable is `mpg` (Miles/(US) gallon). Rownames are stored as variable `"..rownames"` with column role `"model"`.

Format

R6::R6Class inheriting from TaskRegr.

Construction

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

Meta Information

- Task type: "regr"
- Dimensions: 32x11
- Properties: -
- Has Missings: `FALSE`
- Target: "mpg"
- Features: "am", "carb", "cyl", "disp", "drat", "gear", "hp", "qsec", "vs", "wt"

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3data** for more toy tasks.
- Package **mlr3oml** for downloading tasks from https://www.openml.org.
- Package **mlr3viz** for some generic visualizations.
- Dictionary of Tasks: mlr_tasks
- `as.data.table(mlr_tasks)` for a table of available Tasks in the running session (depending on the loaded packages).
- **mlr3fselect** and **mlr3filters** for feature selection and feature filtering.
- Extension packages for additional task types:
  - Unsupervised clustering: **mlr3cluster**
  - Probabilistic supervised regression and survival analysis: https://mlr3proba.mlr-org.com/.

Other Task: Task, TaskClassif, TaskRegr, TaskSupervised, TaskUnsupervised, california_housing, mlr_tasks, mlr_tasks_breast_cancer, mlr_tasks_german_credit, mlr_tasks_iris, mlr_tasks_penguins, mlr_tasks_pima, mlr_tasks_sonar, mlr_tasks_spam, mlr_tasks_wine, mlr_tasks_zoo

`mlr_tasks_penguins` *Palmer Penguins Data Set*

Description

Classification data to predict the species of penguins from the **palmerpenguins** package, see palmerpenguins::penguins. A better alternative to the iris data set.

Format

R6::R6Class inheriting from TaskClassif.

Dictionary

This Task can be instantiated via the dictionary mlr_tasks or with the associated sugar function `tsk()`:

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

Meta Information

- Task type: "classif"
- Dimensions: 344x8
- Properties: "multiclass"
- Has Missings: `TRUE`
- Target: "species"
- Features: "bill_depth", "bill_length", "body_mass", "flipper_length", "island", "sex", "year"

Pre-processing

- The unit of measurement have been removed from the column names. Lengths are given in millimeters (mm), weight in gram (g).

Source

palmerpenguins

References

Gorman KB, Williams TD, Fraser WR (2014). "Ecological Sexual Dimorphism and Environmental Variability within a Community of Antarctic Penguins (Genus Pygoscelis)." *PLoS ONE*, **9**(3), e90081. doi:10.1371/journal.pone.0090081.

https://github.com/allisonhorst/palmerpenguins

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3data** for more toy tasks.
- Package **mlr3oml** for downloading tasks from https://www.openml.org.
- Package **mlr3viz** for some generic visualizations.
- Dictionary of Tasks: mlr_tasks
- `as.data.table(mlr_tasks)` for a table of available Tasks in the running session (depending on the loaded packages).
- **mlr3fselect** and **mlr3filters** for feature selection and feature filtering.
- Extension packages for additional task types:
  - Unsupervised clustering: **mlr3cluster**
  - Probabilistic supervised regression and survival analysis: https://mlr3proba.mlr-org.com/.

Other Task: Task, TaskClassif, TaskRegr, TaskSupervised, TaskUnsupervised, california_housing, mlr_tasks, mlr_tasks_breast_cancer, mlr_tasks_german_credit, mlr_tasks_iris, mlr_tasks_mtcars, mlr_tasks_pima, mlr_tasks_sonar, mlr_tasks_spam, mlr_tasks_wine, mlr_tasks_zoo

---

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.

Dictionary

This Task can be instantiated via the dictionary mlr_tasks or with the associated sugar function tsk():

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

Meta Information

- Task type: "classif"
- Dimensions: 768x9
- Properties: "twoclass"
- Has Missings: TRUE
- Target: "diabetes"
- Features: "age", "glucose", "insulin", "mass", "pedigree", "pregnant", "pressure", "triceps"

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3data** for more toy tasks.
- Package **mlr3oml** for downloading tasks from https://www.openml.org.
- Package **mlr3viz** for some generic visualizations.
- Dictionary of Tasks: mlr_tasks
- `as.data.table(mlr_tasks)` for a table of available Tasks in the running session (depending on the loaded packages).
- **mlr3fselect** and **mlr3filters** for feature selection and feature filtering.
- Extension packages for additional task types:
  - Unsupervised clustering: **mlr3cluster**
  - Probabilistic supervised regression and survival analysis: https://mlr3proba.mlr-org.com/.

Other Task: Task, TaskClassif, TaskRegr, TaskSupervised, TaskUnsupervised, california_housing, mlr_tasks, mlr_tasks_breast_cancer, mlr_tasks_german_credit, mlr_tasks_iris, mlr_tasks_mtcars, mlr_tasks_penguins, mlr_tasks_sonar, mlr_tasks_spam, mlr_tasks_wine, mlr_tasks_zoo

---

`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.

Dictionary

This Task can be instantiated via the dictionary mlr_tasks or with the associated sugar function `tsk()`:

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

Meta Information

- Task type: “classif”
- Dimensions: 208x61
- Properties: “twoclass”
- Has Missings: FALSE
- Target: “Class”
- Features: “V1”, “V10”, “V11”, “V12”, “V13”, “V14”, “V15”, “V16”, “V17”, “V18”, “V19”, “V2”, “V20”, “V21”, “V22”, “V23”, “V24”, “V25”, “V26”, “V27”, “V28”, “V29”, “V3”, “V30”, “V31”, “V32”, “V33”, “V34”, “V35”, “V36”, “V37”, “V38”, “V39”, “V4”, “V40”, “V41”, “V42”, “V43”, “V44”, “V45”, “V46”, “V47”, “V48”, “V49”, “V5”, “V50”, “V51”, “V52”, “V53”, “V54”, “V55”, “V56”, “V57”, “V58”, “V59”, “V6”, “V60”, “V7”, “V8”, “V9”

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3data** for more toy tasks.
- Package **mlr3oml** for downloading tasks from https://www.openml.org.
- Package **mlr3viz** for some generic visualizations.
- Dictionary of Tasks: mlr_tasks
- as.data.table(mlr_tasks) for a table of available Tasks in the running session (depending on the loaded packages).
- **mlr3fselect** and **mlr3filters** for feature selection and feature filtering.
- Extension packages for additional task types:
  - Unsupervised clustering: **mlr3cluster**
  - Probabilistic supervised regression and survival analysis: https://mlr3proba.mlr-org.com/.

Other Task: Task, TaskClassif, TaskRegr, TaskSupervised, TaskUnsupervised, california_housing, mlr_tasks, mlr_tasks_breast_cancer, mlr_tasks_german_credit, mlr_tasks_iris, mlr_tasks_mtcars, mlr_tasks_penguins, mlr_tasks_pima, mlr_tasks_spam, mlr_tasks_wine, mlr_tasks_zoo

---

mlr_tasks_spam *Spam Classification Task*

Description

Spam data set from the UCI machine learning repository (http://archive.ics.uci.edu/dataset/94/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.

Dictionary

This Task can be instantiated via the dictionary mlr_tasks or with the associated sugar function `tsk()`:

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

Meta Information

- Task type: "classif"
- Dimensions: 4601x58
- Properties: "twoclass"
- Has Missings: `FALSE`
- Target: "type"
- Features: "address", "addresses", "all", "business", "capitalAve", "capitalLong", "capitalTotal", "charDollar", "charExclamation", "charHash", "charRoundbracket", "charSemicolon", "charSquarebracket", "conference", "credit", "cs", "data", "direct", "edu", "email", "font", "free", "george", "hp", "hpl", "internet", "lab", "labs", "mail", "make", "meeting", "money", "num000", "num1999", "num3d", "num415", "num650", "num85", "num857", "order", "original", "our", "over", "parts", "people", "pm", "project", "re", "receive", "remove", "report", "table", "technology", "telnet", "will", "you", "your"

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, Dheeru, Graff, Casey (2017). "UCI Machine Learning Repository." http://archive.ics.uci.edu/datasets.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3data** for more toy tasks.
- Package **mlr3oml** for downloading tasks from https://www.openml.org.
- Package **mlr3viz** for some generic visualizations.

- Dictionary of Tasks: mlr_tasks
- `as.data.table(mlr_tasks)` for a table of available Tasks in the running session (depending on the loaded packages).
- **mlr3fselect** and **mlr3filters** for feature selection and feature filtering.
- Extension packages for additional task types:
  - Unsupervised clustering: **mlr3cluster**
  - Probabilistic supervised regression and survival analysis: https://mlr3proba.mlr-org.com/.

Other Task: Task, TaskClassif, TaskRegr, TaskSupervised, TaskUnsupervised, california_housing, mlr_tasks, mlr_tasks_breast_cancer, mlr_tasks_german_credit, mlr_tasks_iris, mlr_tasks_mtcars, mlr_tasks_penguins, mlr_tasks_pima, mlr_tasks_sonar, mlr_tasks_wine, mlr_tasks_zoo

---

`mlr_tasks_wine` *Wine Classification Task*

Description

Wine data set from the UCI machine learning repository (http://archive.ics.uci.edu/dataset/109/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.

Dictionary

This Task can be instantiated via the dictionary mlr_tasks or with the associated sugar function `tsk()`:

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

Meta Information

- Task type: "classif"
- Dimensions: 178x14
- Properties: "multiclass"
- Has Missings: FALSE
- Target: "type"
- Features: "alcalinity", "alcohol", "ash", "color", "dilution", "flavanoids", "hue", "magnesium", "malic", "nonflavanoids", "phenols", "proanthocyanins", "proline"

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, Dheeru, Graff, Casey (2017). "UCI Machine Learning Repository." http://archive.ics.uci.edu/datasets.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3data** for more toy tasks.
- Package **mlr3oml** for downloading tasks from https://www.openml.org.
- Package **mlr3viz** for some generic visualizations.
- Dictionary of Tasks: mlr_tasks
- `as.data.table(mlr_tasks)` for a table of available Tasks in the running session (depending on the loaded packages).
- **mlr3fselect** and **mlr3filters** for feature selection and feature filtering.
- Extension packages for additional task types:
  - Unsupervised clustering: **mlr3cluster**
  - Probabilistic supervised regression and survival analysis: https://mlr3proba.mlr-org.com/.

Other Task: Task, TaskClassif, TaskRegr, TaskSupervised, TaskUnsupervised, california_housing, mlr_tasks, mlr_tasks_breast_cancer, mlr_tasks_german_credit, mlr_tasks_iris, mlr_tasks_mtcars, mlr_tasks_penguins, mlr_tasks_pima, mlr_tasks_sonar, mlr_tasks_spam, mlr_tasks_zoo

---

mlr_tasks_zoo *Zoo Classification Task*

Description

A classification task for the mlbench::Zoo data set. Rownames are stored as variable `"..rownames"` with column role `"name"`.

Format

R6::R6Class inheriting from TaskClassif.

Dictionary

This Task can be instantiated via the dictionary mlr_tasks or with the associated sugar function `tsk()`:

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

Meta Information

- Task type: "classif"
- Dimensions: 101x17
- Properties: "multiclass"
- Has Missings: `FALSE`
- Target: "type"
- Features: "airborne", "aquatic", "backbone", "breathes", "catsize", "domestic", "eggs", "feathers", "fins", "hair", "legs", "milk", "predator", "tail", "toothed", "venomous"

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3data** for more toy tasks.
- Package **mlr3oml** for downloading tasks from https://www.openml.org.
- Package **mlr3viz** for some generic visualizations.
- Dictionary of Tasks: mlr_tasks
- `as.data.table(mlr_tasks)` for a table of available Tasks in the running session (depending on the loaded packages).
- **mlr3fselect** and **mlr3filters** for feature selection and feature filtering.
- Extension packages for additional task types:
  - Unsupervised clustering: **mlr3cluster**
  - Probabilistic supervised regression and survival analysis: https://mlr3proba.mlr-org.com/.

Other Task: `Task`, `TaskClassif`, `TaskRegr`, `TaskSupervised`, `TaskUnsupervised`, `california_housing`, `mlr_tasks`, `mlr_tasks_breast_cancer`, `mlr_tasks_german_credit`, `mlr_tasks_iris`, `mlr_tasks_mtcars`, `mlr_tasks_penguins`, `mlr_tasks_pima`, `mlr_tasks_sonar`, `mlr_tasks_spam`, `mlr_tasks_wine`

`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()`/`tgens()`.

Format

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

Methods

See mlr3misc::Dictionary.

S3 methods

- `as.data.table(dict, ..., objects = FALSE)`
  mlr3misc::Dictionary -> `data.table::data.table()`
  Returns a `data.table::data.table()` with fields "key", "label", "task_type", "params", and "packages" as columns. If `objects` is set to `TRUE`, the constructed objects are returned in the list column named `object`.

See Also

Sugar functions: `tgen()`, `tgens()`

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

Other TaskGenerator: `TaskGenerator`, `mlr_task_generators_2dnormals`, `mlr_task_generators_cassini`, `mlr_task_generators_circle`, `mlr_task_generators_friedman1`, `mlr_task_generators_moons`, `mlr_task_generators_peak`, `mlr_task_generators_simplex`, `mlr_task_generators_smiley`, `mlr_task_generators_spirals`, `mlr_task_generators_xor`

Examples

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

`mlr_task_generators_2dnormals`

2D Normals Classification Task Generator

Description

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

Dictionary

This TaskGenerator can be instantiated via the dictionary mlr_task_generators or with the associated sugar function `tgen()`:

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

Parameters

Id	Type	Default	Range
cl	integer	-	$[2, \infty)$
r	numeric	-	$[1, \infty)$
sd	numeric	-	$[0, \infty)$

Super class

`mlr3::TaskGenerator` -> `TaskGenerator2DNormals`

Methods

Public methods:

- `TaskGenerator2DNormals$new()`
- `TaskGenerator2DNormals$plot()`
- `TaskGenerator2DNormals$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
TaskGenerator2DNormals$new()
```

Method `plot()`**:** Creates a simple plot of generated data.

Usage:

```
TaskGenerator2DNormals$plot(n = 200L, pch = 19L, ...)
```

Arguments:

`n (integer(1))`
Number of samples to draw for the plot. Default is 200.

`pch (integer(1))`
Point char. Passed to `plot()`.

`...` (any)
Additional arguments passed to `plot()`.

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

Usage:

```
TaskGenerator2DNormals$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

- Dictionary of TaskGenerators: mlr_task_generators
- `as.data.table(mlr_task_generators)` for a table of available TaskGenerators in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other TaskGenerator: `TaskGenerator`, `mlr_task_generators`, `mlr_task_generators_cassini`, `mlr_task_generators_circle`, `mlr_task_generators_friedman1`, `mlr_task_generators_moons`, `mlr_task_generators_peak`, `mlr_task_generators_simplex`, `mlr_task_generators_smiley`, `mlr_task_generators_spirals`, `mlr_task_generators_xor`

Examples

```
generator = tgen("2dnormals")
plot(generator, n = 200)

task = generator$generate(200)
str(task$data())
```

---

mlr_task_generators_cassini

Cassini Classification Task Generator

Description

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

Dictionary

This TaskGenerator can be instantiated via the dictionary mlr_task_generators or with the associated sugar function `tgen()`:

```
mlr_task_generators$get("cassini")
tgen("cassini")
```

Parameters

Id	Type	Default	Range
relsize1	integer	2	$[1, \infty)$
relsize2	integer	2	$[1, \infty)$
relsize3	integer	1	$[1, \infty)$

Super class

`mlr3::TaskGenerator` -> `TaskGeneratorCassini`

Methods

Public methods:

- `TaskGeneratorCassini$new()`
- `TaskGeneratorCassini$plot()`
- `TaskGeneratorCassini$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
TaskGeneratorCassini$new()
```

Method `plot()`**:** Creates a simple plot of generated data.

Usage:

```
TaskGeneratorCassini$plot(n = 200L, pch = 19L, ...)
```

Arguments:

`n (integer(1))`
: Number of samples to draw for the plot. Default is 200.

`pch (integer(1))`
: Point char. Passed to `plot()`.

`...` (any)
: Additional arguments passed to `plot()`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Dictionary of TaskGenerators: mlr_task_generators
- `as.data.table(mlr_task_generators)` for a table of available TaskGenerators in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other TaskGenerator: `TaskGenerator`, `mlr_task_generators`, `mlr_task_generators_2dnormals`, `mlr_task_generators_circle`, `mlr_task_generators_friedman1`, `mlr_task_generators_moons`, `mlr_task_generators_peak`, `mlr_task_generators_simplex`, `mlr_task_generators_smiley`, `mlr_task_generators_spirals`, `mlr_task_generators_xor`

Examples

```
generator = tgen("cassini")
plot(generator, n = 200)

task = generator$generate(200)
str(task$data())
```

---

mlr_task_generators_circle

Circle Classification Task Generator

Description

A TaskGenerator for the circle binary classification task in `mlbench::mlbench.circle()`. Creates a large circle containing a smaller circle.

Dictionary

This TaskGenerator can be instantiated via the dictionary mlr_task_generators or with the associated sugar function `tgen()`:

```
mlr_task_generators$get("circle")
tgen("circle")
```

Parameters

Id	Type	Default	Range
d	integer	2	$[2, \infty)$

Super class

`mlr3::TaskGenerator` -> `TaskGeneratorCircle`

Methods

Public methods:

- `TaskGeneratorCircle$new()`
- `TaskGeneratorCircle$plot()`
- `TaskGeneratorCircle$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
TaskGeneratorCircle$new()
```

Method `plot()`**:** Creates a simple plot of generated data.

Usage:

```
TaskGeneratorCircle$plot(n = 200L, pch = 19L, ...)
```

Arguments:

`n (integer(1))`
Number of samples to draw for the plot. Default is 200.

`pch (integer(1))`
Point char. Passed to `plot()`.

`...` (any)
Additional arguments passed to `plot()`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Dictionary of TaskGenerators: mlr_task_generators
- `as.data.table(mlr_task_generators)` for a table of available TaskGenerators in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other TaskGenerator: `TaskGenerator`, `mlr_task_generators`, `mlr_task_generators_2dnormals`, `mlr_task_generators_cassini`, `mlr_task_generators_friedman1`, `mlr_task_generators_moons`, `mlr_task_generators_peak`, `mlr_task_generators_simplex`, `mlr_task_generators_smiley`, `mlr_task_generators_spirals`, `mlr_task_generators_xor`

Examples

```
generator = tgen("circle")
plot(generator, n = 200)

task = generator$generate(200)
str(task$data())
```

`mlr_task_generators_friedman1`

Friedman1 Regression Task Generator

Description

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

Dictionary

This TaskGenerator can be instantiated via the dictionary mlr_task_generators or with the associated sugar function `tgen()`:

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

Parameters

Id	Type	Default	Range
sd	numeric	1	$[0, \infty)$

Super class

`mlr3::TaskGenerator` -> `TaskGeneratorFriedman1`

Methods

Public methods:

- `TaskGeneratorFriedman1$new()`
- `TaskGeneratorFriedman1$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
TaskGeneratorFriedman1$new()
```

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

Usage:

```
TaskGeneratorFriedman1$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

- Dictionary of TaskGenerators: mlr_task_generators
- `as.data.table(mlr_task_generators)` for a table of available TaskGenerators in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other TaskGenerator: `TaskGenerator`, `mlr_task_generators`, `mlr_task_generators_2dnormals`, `mlr_task_generators_cassini`, `mlr_task_generators_circle`, `mlr_task_generators_moons`, `mlr_task_generators_peak`, `mlr_task_generators_simplex`, `mlr_task_generators_smiley`, `mlr_task_generators_spirals`, `mlr_task_generators_xor`

Examples

```
generator = tgen("friedman1")
task = generator$generate(200)
str(task$data())
```

---

mlr_task_generators_moons

Moons Classification Task Generator

Description

A TaskGenerator creating two interleaving half circles ("moons") as binary classification problem.

Dictionary

This TaskGenerator can be instantiated via the dictionary mlr_task_generators or with the associated sugar function `tgen()`:

```
mlr_task_generators$get("moons")
tgen("moons")
```

Parameters

Id	Type	Default	Range
sigma	numeric	-	$[0, \infty)$

Super class

`mlr3::TaskGenerator` -> `TaskGeneratorMoons`

Methods

Public methods:

- `TaskGeneratorMoons$new()`
- `TaskGeneratorMoons$plot()`
- `TaskGeneratorMoons$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
TaskGeneratorMoons$new()
```

Method `plot()`**:** Creates a simple plot of generated data.

Usage:

```
TaskGeneratorMoons$plot(n = 200L, pch = 19L, ...)
```

Arguments:

`n (integer(1))`
Number of samples to draw for the plot. Default is 200.

`pch (integer(1))`
Point char. Passed to `plot()`.

`...` (any)
Additional arguments passed to `plot()`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Dictionary of TaskGenerators: mlr_task_generators
- `as.data.table(mlr_task_generators)` for a table of available TaskGenerators in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other TaskGenerator: `TaskGenerator`, `mlr_task_generators`, `mlr_task_generators_2dnormals`, `mlr_task_generators_cassini`, `mlr_task_generators_circle`, `mlr_task_generators_friedman1`, `mlr_task_generators_peak`, `mlr_task_generators_simplex`, `mlr_task_generators_smiley`, `mlr_task_generators_spirals`, `mlr_task_generators_xor`

Examples

```
generator = tgen("moons")
plot(generator, n = 200)

task = generator$generate(200)
str(task$data())
```

---

mlr_task_generators_peak

Peak Regression Task Generator

Description

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

Dictionary

This TaskGenerator can be instantiated via the dictionary mlr_task_generators or with the associated sugar function `tgen()`:

```
mlr_task_generators$get("peak")
tgen("peak")
```

Parameters

Id	Type	Default	Range
d	integer	20	$[1, \infty)$

Super class

`mlr3::TaskGenerator` -> `TaskGeneratorPeak`

Methods

Public methods:

- `TaskGeneratorPeak$new()`
- `TaskGeneratorPeak$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
TaskGeneratorPeak$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Dictionary of TaskGenerators: mlr_task_generators
- `as.data.table(mlr_task_generators)` for a table of available TaskGenerators in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other TaskGenerator: `TaskGenerator`, `mlr_task_generators`, `mlr_task_generators_2dnormals`, `mlr_task_generators_cassini`, `mlr_task_generators_circle`, `mlr_task_generators_friedman1`, `mlr_task_generators_moons`, `mlr_task_generators_simplex`, `mlr_task_generators_smiley`, `mlr_task_generators_spirals`, `mlr_task_generators_xor`

Examples

```
generator = tgen("peak", d = 5)
task = generator$generate(200)
str(task$data())
```

`mlr_task_generators_simplex` *Simplex Classification Task Generator*

Description

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

Note that the generator implemented in **mlbench** returns fewer samples than requested.

Dictionary

This TaskGenerator can be instantiated via the dictionary mlr_task_generators or with the associated sugar function `tgen()`:

```
mlr_task_generators$get("simplex")
tgen("simplex")
```

Parameters

Id	Type	Default	Levels	Range
center	logical	TRUE	TRUE, FALSE	-
d	integer	3		$[1, \infty)$
sd	numeric	0.1		$[0, \infty)$
sides	integer	1		$[1, \infty)$

Super class

`mlr3::TaskGenerator` -> `TaskGeneratorSimplex`

Methods

Public methods:

- `TaskGeneratorSimplex$new()`
- `TaskGeneratorSimplex$plot()`
- `TaskGeneratorSimplex$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
TaskGeneratorSimplex$new()
```

Method `plot()`**:** Creates a simple plot of generated data.

Usage:

```
TaskGeneratorSimplex$plot(n = 200L, pch = 19L, ...)
```

Arguments:

`n (integer(1))`
Number of samples to draw for the plot. Default is 200.

`pch (integer(1))`
Point char. Passed to `plot()`.

`... (any)`
Additional arguments passed to `plot()`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Dictionary of TaskGenerators: mlr_task_generators
- `as.data.table(mlr_task_generators)` for a table of available TaskGenerators in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other TaskGenerator: `TaskGenerator`, `mlr_task_generators`, `mlr_task_generators_2dnormals`, `mlr_task_generators_cassini`, `mlr_task_generators_circle`, `mlr_task_generators_friedman1`, `mlr_task_generators_moons`, `mlr_task_generators_peak`, `mlr_task_generators_smiley`, `mlr_task_generators_sp` `mlr_task_generators_xor`

Examples

```
generator = tgen("simplex")
plot(generator, n = 200)

task = generator$generate(200)
str(task$data())
```

---

mlr_task_generators_smiley

Smiley Classification Task Generator

Description

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

Dictionary

This TaskGenerator can be instantiated via the dictionary mlr_task_generators or with the associated sugar function `tgen()`:

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

Parameters

Id	Type	Default	Range
sd1	numeric	-	$[0, \infty)$
sd2	numeric	-	$[0, \infty)$

Super class

`mlr3::TaskGenerator` -> `TaskGeneratorSmiley`

Methods

Public methods:

- `TaskGeneratorSmiley$new()`
- `TaskGeneratorSmiley$plot()`
- `TaskGeneratorSmiley$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
TaskGeneratorSmiley$new()
```

Method `plot()`**:** Creates a simple plot of generated data.

Usage:

```
TaskGeneratorSmiley$plot(n = 200L, pch = 19L, ...)
```

Arguments:

`n (integer(1))`
Number of samples to draw for the plot. Default is 200.

`pch (integer(1))`
Point char. Passed to `plot()`.

`...` (any)
Additional arguments passed to `plot()`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Dictionary of TaskGenerators: mlr_task_generators
- `as.data.table(mlr_task_generators)` for a table of available TaskGenerators in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other TaskGenerator: `TaskGenerator`, `mlr_task_generators`, `mlr_task_generators_2dnormals`, `mlr_task_generators_cassini`, `mlr_task_generators_circle`, `mlr_task_generators_friedman1`, `mlr_task_generators_moons`, `mlr_task_generators_peak`, `mlr_task_generators_simplex`, `mlr_task_generators_spirals`, `mlr_task_generators_xor`

Examples

```
generator = tgen("smiley")
plot(generator, n = 200)

task = generator$generate(200)
str(task$data())
```

`mlr_task_generators_spirals`

Spiral Classification Task Generator

Description

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

Dictionary

This TaskGenerator can be instantiated via the dictionary mlr_task_generators or with the associated sugar function `tgen()`:

```
mlr_task_generators$get("spirals")
tgen("spirals")
```

Parameters

Id	Type	Default	Range
cycles	integer	1	$[1, \infty)$
sd	numeric	0	$[0, \infty)$

Super class

`mlr3::TaskGenerator` -> `TaskGeneratorSpirals`

Methods

Public methods:

- `TaskGeneratorSpirals$new()`
- `TaskGeneratorSpirals$plot()`
- `TaskGeneratorSpirals$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
TaskGeneratorSpirals$new()
```

Method `plot()`**:** Creates a simple plot of generated data.

Usage:

```
TaskGeneratorSpirals$plot(n = 200L, pch = 19L, ...)
```

Arguments:

`n (integer(1))`
: Number of samples to draw for the plot. Default is 200.

`pch (integer(1))`
: Point char. Passed to `plot()`.

`...` (any)
: Additional arguments passed to `plot()`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Dictionary of TaskGenerators: mlr_task_generators
- `as.data.table(mlr_task_generators)` for a table of available TaskGenerators in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other TaskGenerator: `TaskGenerator`, `mlr_task_generators`, `mlr_task_generators_2dnormals`, `mlr_task_generators_cassini`, `mlr_task_generators_circle`, `mlr_task_generators_friedman1`, `mlr_task_generators_moons`, `mlr_task_generators_peak`, `mlr_task_generators_simplex`, `mlr_task_generators_smiley`, `mlr_task_generators_xor`

Examples

```
generator = tgen("spirals")
plot(generator, n = 200)

task = generator$generate(200)
str(task$data())
```

`mlr_task_generators_xor`

XOR Classification Task Generator

Description

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

Dictionary

This TaskGenerator can be instantiated via the dictionary mlr_task_generators or with the associated sugar function `tgen()`:

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

Parameters

Id	Type	Default	Range
d	integer	1	$[1, \infty)$

Super class

`mlr3::TaskGenerator` -> `TaskGeneratorXor`

Methods

Public methods:

- `TaskGeneratorXor$new()`
- `TaskGeneratorXor$plot()`
- `TaskGeneratorXor$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
TaskGeneratorXor$new()
```

Method `plot()`**:** Creates a simple plot of generated data.

Usage:

```
TaskGeneratorXor$plot(n = 200L, pch = 19L, ...)
```

Arguments:

`n (integer(1))`
Number of samples to draw for the plot. Default is 200.

`pch (integer(1))`
Point char. Passed to `plot()`.

`...` (any)
Additional arguments passed to `plot()`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Dictionary of TaskGenerators: mlr_task_generators
- `as.data.table(mlr_task_generators)` for a table of available TaskGenerators in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other TaskGenerator: `TaskGenerator`, `mlr_task_generators`, `mlr_task_generators_2dnormals`, `mlr_task_generators_cassini`, `mlr_task_generators_circle`, `mlr_task_generators_friedman1`, `mlr_task_generators_moons`, `mlr_task_generators_peak`, `mlr_task_generators_simplex`, `mlr_task_generators_smiley`, `mlr_task_generators_spirals`

Examples

```
generator = tgen("xor")
plot(generator, n = 200)

task = generator$generate(200)
str(task$data())
```

`mlr_test_helpers` *Documentation of mlr3 test helpers*

Description

The mlr3 package contains various helper functions to test the validity of objects such as learners. These functions are not contained in the mlr3 namespaces and are instead located in the `inst/testthat` directory of the source package or the `testthat` directory of the installed package.

These files can be sourced with the following line of code:

```
lapply(list.files(system.file("testthat", package = "mlr3"), pattern = "^helper.*\\.[rR]", full.names
```

Other extension packages such as `mlr3proba` have similar files that can be sourced accordingly.

This manual page documents the most important helper functions that are relevant when users implement their own custom learners.

run_autotest()

This function runs a Learner's automatic test suite.

During the autotests, multiple tasks are generated depending on the properties of the learner. The `run_autotest()` function then trains the learner on each task and predicts with all supported predict types. (see argument `predict_types`). To debug, simply run `result = run_autotest(learner)` and proceed with investigating the task, learner and prediction of the returned `result`.

For example usages you can look at the autotests in various mlr3 source repositories such as mlr3learners. More information can be found in the `inst/testthat/autotest.R` file.

Parameters:

- `learner` (`Learner`)
  The learner to check.
- `N` (`integer(1)`)
  The number of rows of the generated tasks.
- `exclude` (`character()`)
  Each task on which the learner is trained has an id. If for some reason, one or more such tests ought to be disabled, this argument takes in a regular expression that disables all tasks whose id matches the regular expression.
- `predict_types` (`character()`)
  The predict types of the learner to check. Defaults to all predict typpes of the learner.
- `check_replicable` (`logical(1)`)
  Whether to check that running the learner twice with the same seed should result in identical predictions. Default is `TRUE`.
- `configure_learner` (`function(learner, task)`)
  Before running a `learner` on a `task`, this function allows to change its parameter values depending on the input task.

run_paramtest()

Description:

Checks parameters of mlr3 Learners against parameters defined in the upstream functions of the respective learner. The goal is to detect if parameters have been dropped or added in the upstream implementation. Some learners do not have all of their parameters stored within the learner function that is called during training. Sometimes learners come with a "control" function, e.g. `glmnet.control()` from package **glmnet**. Such learners need to be checked as well since they make up the full ParamSet of the respective learner.

To work nicely with the defined ParamSet, certain parameters need to be excluded because these are only present in either the "control" object or the actual top-level function call. Such exclusions should go into argument `exclude` with a comment for the reason of the exclusion. See examples for more information.

For example usages you can look at the parameter tests in various mlr3 source repositories such as **mlr3learners**.

Parameters:

- `learner` (`Learner`)
  The learner whose parameter set is being checked.
- `fun` (`function()` or list of `functions()`s)
  The function(s) containing the parameters that must be implemented by the learner.
- `exclude` (`character()`)
  Argument names that specified through this argument are exempt from checking. This can be used when parameters that are available in the `fun` function(s) are not implemented in the learner, or when the learner implements additional parameters that are not available in the `fun` function(s).
- `tag` (`character(1)`)
  Only parameters that are tagged with this tag are being checked. If `NULL` (default), all parameters are checked.

expect_learner()

Checks various properties that learners have to satisfy. Used for testing learner implementations, especially if all methods and fields are implement as document.

Parameters

- `lrn` :: (`Learner`)
  The learner whose properties are being verified.
- `tsk` :: (`Task`)
  Optional argument (default is `NULL`). If provided, some additional checks are being run that check the compatibility of the learner and task.
- `check_man` :: (`logical(1)`)
  Whether to check if the learner has a man page.

partition *Manually Partition into Training, Test and Validation Set*

Description

Creates a split of the row ids of a Task into a training and a test set, and optionally a validation set.

Usage

```
partition(task, ratio = 0.67)

## S3 method for class 'Task'
partition(task, ratio = 0.67)
```

Arguments

task	(Task) Task to operate on.
ratio	(numeric()) Ratio of observations to put into the training set. If a 2 element vector is provided, the first element is the ratio for the training set, the second element is the ratio for the test set. The validation set will contain the remaining observations.

Examples

```
# regression task partitioned into training and test set
task = tsk("california_housing")
split = partition(task, ratio = 0.5)
data = data.frame(
  y = c(task$truth(split$train), task$truth(split$test)),
  split = rep(c("train", "predict"), lengths(split[c("train", "test")]))
)
boxplot(y ~ split, data = data)

# classification task partitioned into training, test and validation set
task = tsk("pima")
split = partition(task, c(0.66, 0.14))
```

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.

Note that this object is usually constructed via a derived classes, e.g. PredictionClassif or PredictionRegr.

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.

Public fields

`data` (named `list()`)
: Internal data structure.

`task_type` (`character(1)`)
: Required type of the Task.

`task_properties` (`character()`)
: Required properties of the Task.

`predict_types` (`character()`)
: Set of predict types this object stores.

`man` (`character(1)`)
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. Defaults to `NA`, but can be set by child classes.

Active bindings

`row_ids` (`integer()`)
: Vector of row ids for which predictions are stored.

`truth` (any)
: True (observed) outcome.

`missing` (`integer()`)
: Returns `row_ids` for which the predictions are missing or incomplete.

`weights` (`numeric()`)
: Vector of measure weights, obtained from the `weights_measure` column of the Task if present. This is `NULL` if no weights are present.

Methods

Public methods:

- `Prediction$format()`
- `Prediction$print()`
- `Prediction$help()`
- `Prediction$score()`
- `Prediction$obs_loss()`
- `Prediction$filter()`
- `Prediction$clone()`

Method `format()`**:** Helper for print outputs.

Usage:

```
Prediction$format(...)
```

Arguments:

`...` (ignored).

Method `print()`**:** Printer.

Usage:

```
Prediction$print(...)
```

Arguments:

`...` (ignored).

Method `help()`**:** Opens the corresponding help page referenced by field `$man`.

Usage:

```
Prediction$help()
```

Method `score()`**:** 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. Note that the `predict_sets` of the `measures` are ignored by this method, instead all predictions are used.

Usage:

```
Prediction$score(
  measures = NULL,
  task = NULL,
  learner = NULL,
  train_set = NULL
)
```

Arguments:

`measures` (Measure | list of Measure)
: Measure(s) to calculate.

`task` (Task).

`learner` (Learner).

`train_set` (`integer()`).

Returns: Prediction.

Method `obs_loss()`**:** Calculates the observation-wise loss via the loss function set in the Measure's field `obs_loss`. Returns a `data.table()` with the columns `row_ids`, `truth`, `response` and one additional numeric column for each measure, named with the respective measure id. If there is no observation-wise loss function for the measure, the column is filled with `NA` values. Note that some measures such as RMSE, do have an `$obs_loss`, but they require an additional transformation after aggregation, in this example taking the square-root.

Usage:

```
Prediction$obs_loss(measures = NULL)
```

Arguments:

`measures` (Measure | list of Measure)
Measure(s) to calculate.

Method `filter()`**:** Filters the Prediction, keeping only predictions for the provided row_ids. This changes the object in-place, you need to create a clone to preserve the original Prediction.

Usage:

```
Prediction$filter(row_ids)
```

Arguments:

`row_ids integer()`
Row indices.

Returns: `self`, modified.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3viz** for some generic visualizations.
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

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.

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 that there are the following edge cases for threshold equal to `0` which are handled specially:

  1. With threshold 0 the resulting ratio gets `Inf` and thus gets always selected. If there are multiple ratios with value `Inf`, one is selected according to `ties_method` (randomly per default).
  2. If additionally the predicted probability is also 0, the ratio `0/0` results in `NaN` values. These are simply replaced by `0` and thus will never get selected.

Super class

`mlr3::Prediction` -> `PredictionClassif`

Active bindings

`response (factor())`
Access to the stored predicted class labels.

`prob (matrix())`
Access to the stored probabilities.

`confusion` (`matrix()`)
Confusion matrix, as resulting from the comparison of truth and response. Truth is in columns, predicted response is in rows.

Methods

Public methods:

- `PredictionClassif$new()`
- `PredictionClassif$set_threshold()`
- `PredictionClassif$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
PredictionClassif$new(
  task = NULL,
  row_ids = task$row_ids,
  truth = task$truth(),
  response = NULL,
  prob = NULL,
  weights = NULL,
  check = TRUE
)
```

Arguments:

`task` (TaskClassif)
Task, used to extract defaults for `row_ids` and `truth`.

`row_ids` (`integer()`)
Row ids of the predicted observations, i.e. the row ids of 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 `max.col()` with `ties.method` set to `"random"`.

`weights` (`numeric()`)
Vector of measure weights for each observation. Should be constructed from the `Task`'s `weights_measure` column.

`check` (`logical(1)`)
If `TRUE`, performs some argument checks and predict type conversions.

Method `set_threshold()`**:** Sets the prediction response based on the provided threshold. See the section on thresholding for more information.

Usage:

```
PredictionClassif$set_threshold(threshold, ties_method = "random")
```

Arguments:

`threshold` (`numeric()`).

`ties_method` (`character(1)`)
One of `"random"`, `"first"` or `"last"` (c.f. `max.col()`) to determine how to deal with tied probabilities.

Returns: Returns the object itself, but modified **by reference**. You need to explicitly `$clone()` the object beforehand if you want to keeps the object in its previous state.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

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

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3viz** for some generic visualizations.
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other Prediction: `Prediction`, `PredictionRegr`

Examples

```
task = tsk("penguins")
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"))
```

`PredictionData` *Convert to PredictionData*

Description

Objects of type `PredictionData` serve as a intermediate representation for objects of type Prediction. It is an internal data structure, implemented to optimize runtime and solve some issues emerging while serializing R6 objects. End-users typically do not need to worry about the details, package developers are advised to continue reading for some technical information.

Unlike most other **mlr3** objects, `PredictionData` relies on the S3 class system. The following operations must be supported to extend mlr3 for new task types:

- `as_prediction_data()` converts objects to class `PredictionData`, e.g. objects of type Prediction.
- `as_prediction()` converts objects to class Prediction, e.g. objects of type `PredictionData`.
- `check_prediction_data()` is called on the return value of the predict method of a Learner to perform assertions and type conversions. Returns an update object of class `PredictionData`.
- `is_missing_prediction_data()` is used for the fallback learner (see Learner) to impute missing predictions. Returns vector with row ids which need imputation.

Usage

```
create_empty_prediction_data(task, learner)

check_prediction_data(pdata, ...)

is_missing_prediction_data(pdata, ...)

filter_prediction_data(pdata, row_ids, ...)

## S3 method for class 'PredictionDataClassif'
check_prediction_data(pdata, train_task, ...)

## S3 method for class 'PredictionDataClassif'
is_missing_prediction_data(pdata, ...)

## S3 method for class 'PredictionDataClassif'
c(..., keep_duplicates = TRUE)

## S3 method for class 'PredictionDataRegr'
check_prediction_data(pdata, ...)
```

```
## S3 method for class 'PredictionDataRegr'
is_missing_prediction_data(pdata, ...)

## S3 method for class 'PredictionDataRegr'
c(..., keep_duplicates = TRUE)
```

Arguments

task	(Task).
learner	(Learner).
pdata	(PredictionData) Named list inheriting from "PredictionData".
...	(one or more PredictionData objects).
row_ids	integer() Row indices.
train_task	(Task) Task used for training the learner.
keep_duplicates	(logical(1)) If TRUE, the combined PredictionData object is filtered for duplicated row ids (starting from last).

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. Additionally, probability distributions implemented in package distr6 are supported.

Super class

mlr3::Prediction -> PredictionRegr

Active bindings

response (numeric())
: Access the stored predicted response.

se (numeric())
: Access the stored standard error.

quantiles (matrix())
: Matrix of predicted quantiles. Observations are in rows, quantile (in ascending order) in columns.

distr (VectorDistribution)
: Access the stored vector distribution. Requires package distr6(in repository https://raphaels1.r-universe.dev) .

Methods

Public methods:

- PredictionRegr$new()
- PredictionRegr$clone()

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
PredictionRegr$new(
  task = NULL,
  row_ids = task$row_ids,
  truth = task$truth(),
  response = NULL,
  se = NULL,
  quantiles = NULL,
  distr = NULL,
  weights = NULL,
  check = TRUE
)
```

Arguments:

`task` (TaskRegr)
: Task, used to extract defaults for `row_ids` and `truth`.

`row_ids` (`integer()`)
: Row ids of the predicted observations, i.e. the row ids of 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.

`quantiles` (`matrix()`)
: Numeric matrix of predicted quantiles. One row per observation, one column per quantile.

`distr` (`VectorDistribution`)
: `VectorDistribution` from package distr6 (in repository https://raphaels1.r-universe.dev). Each individual distribution in the vector represents the random variable 'survival time' for an individual observation.

`weights` (`numeric()`)
: Vector of measure weights for each observation. Should be constructed from the `Task`'s `weights_measure` column.

`check` (`logical(1)`)
: If `TRUE`, performs some argument checks and predict type conversions.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3viz** for some generic visualizations.
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other Prediction: Prediction, PredictionClassif

Examples

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

---

print.roc_measures *Print ROC Measures*

Description

Print the confusion matrix and a set of roc performance measures.

Usage

```
## S3 method for class 'roc_measures'
print(x, abbreviations = TRUE, digits = 2, ...)
```

Arguments

x	(roc_measures) The object returned by score_roc_measures.
abbreviations	(logical(1)) If TRUE, print a list of abbreviations for the measures.
digits	(integer(1)) Number of digits to round the measures to.
...	(any) Additional parameters, currently unused.

`resample` *Resample a Learner on a Task*

Description

Runs a resampling (possibly in parallel): Repeatedly apply Learner `learner` on a training set of Task `task` to train a model, then use the trained model to predict observations of a test set. Training and test sets are defined by the Resampling `resampling`.

Usage

```
resample(
  task,
  learner,
  resampling,
  store_models = FALSE,
  store_backends = TRUE,
  encapsulate = NA_character_,
  allow_hotstart = FALSE,
  clone = c("task", "learner", "resampling"),
  unmarshal = TRUE,
  callbacks = NULL
)
```

Arguments

`task`	(Task).
`learner`	(Learner).
`resampling`	(Resampling).
`store_models`	(`logical(1)`) Store the fitted model in the resulting object= Set to `TRUE` if you want to further analyse the models or want to extract information like variable importance.
`store_backends`	(`logical(1)`) Keep the DataBackend of the Task in the ResampleResult? Set to `TRUE` if your performance measures require a Task, or to analyse results more conveniently. Set to `FALSE` to reduce the file size and memory footprint after serialization. The current default is `TRUE`, but this eventually will be changed in a future release.
`encapsulate`	(`character(1)`) If not `NA`, enables encapsulation by setting the field `Learner$encapsulate` to one of the supported values: `"none"` (disable encapsulation), `"try"` (captures errors but output is printed to the console and not logged), `"evaluate"` (execute via **evaluate**) and `"callr"` (start in external session via **callr**). If `NA`, encapsulation is not changed, i.e. the settings of the individual learner are active. Additionally, if encapsulation is set to `"evaluate"` or `"callr"`, the fallback learner is set to the featureless learner if the learner does not already have a fallback configured.

`allow_hotstart` (`logical(1)`)
Determines if learner(s) are hot started with trained models in `$hotstart_stack`. See also HotstartStack.

`clone` (`character()`)
Select the input objects to be cloned before proceeding by providing a set with possible values `"task"`, `"learner"` and `"resampling"` for Task, Learner and Resampling, respectively. Per default, all input objects are cloned.

`unmarshal` `Learner`
Whether to unmarshal learners that were marshaled during the execution. If `TRUE` all models are stored in unmarshaled form. If `FALSE`, all learners (that need marshaling) are stored in marshaled form.

`callbacks` (List of mlr3misc::Callback)
Callbacks to be executed during the resampling process. See CallbackResample and ContextResample for details.

Value

ResampleResult.

Stochasticity

Note that uninstantiated `Resampling`s are instantiated on the task, making the procedure stochastic even in case of a deterministic learner.

Predict Sets

If you want to compare the performance of a learner on the training with the performance on the test set, you have to configure the Learner to predict on multiple sets by setting the field `predict_sets` to `c("train", "test")` (default is `"test"`). Each set yields a separate Prediction object during resampling. In the next step, you have to configure the measures to operate on the respective Prediction object:

```
m1 = msr("classif.ce", id = "ce.train", predict_sets = "train")
m2 = msr("classif.ce", id = "ce.test", predict_sets = "test")
```

The (list of) created measures can finally be passed to `$aggregate()` or `$score()`.

Parallelization

This function can be parallelized with the **future** or **mirai** package. One job is one resampling iteration. All jobs are send to an apply function from **future.apply** or `mirai::mirai_map()` in a single batch. To select a parallel backend, use `future::plan()`. To use `mirai`, call `mirai::daemons(.compute = "mlr3_parallelization")` before calling this function. The `future` package guarantees reproducible results independent of the parallel backend. The results of `mirai` will not be the same but can be made reproducible by setting a seed when calling `mirai::daemons()`. More on parallelization can be found in the book: https://mlr3book.mlr-org.com/chapters/chapter10/advanced_technical_aspects_of_mlr3.html

Progress Bars

This function supports progress bars via the package **progressr**. Simply wrap the function call in `progressr::with_progress()` to enable them. Alternatively, call `progressr::handlers()` with `global = TRUE` to enable progress bars globally. We recommend the **progress** package as backend which can be enabled with `progressr::handlers("progress")`.

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 computed in order to reduce memory consumption. If you need access to the models for later analysis, set `store_models` to `TRUE`.

See Also

- `as_benchmark_result()` to convert to a BenchmarkResult.
- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-resampling
- Package **mlr3viz** for some generic visualizations.

Other resample: `ResampleResult`

Examples

```
task = tsk("penguins")
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.

ResampleResults can be visualized via **mlr3viz**'s `autoplot()` function.

S3 Methods

- `as.data.table(rr, reassemble_learners = TRUE, convert_predictions = TRUE, predict_sets = "test")`
  ResampleResult -> `data.table::data.table()`
  Returns a tabular view of the internal data.
- `c(...)`
  (ResampleResult, ...) -> BenchmarkResult
  Combines multiple objects convertible to BenchmarkResult into a new BenchmarkResult.

Active bindings

`task_type` (`character(1)`)
: Task type of objects in the `ResampleResult`, e.g. `"classif"` or `"regr"`. This is `NA` for empty ResampleResults.

`uhash` (`character(1)`)
: Unique hash for this object.

`iters` (`integer(1)`)
: Number of resampling iterations stored in the `ResampleResult`.

`task` (Task)
: The task `resample()` operated on.

`learner` (Learner)
: Learner prototype `resample()` operated on. For a list of **trained** learners, see methods `$learners()`.

`resampling` (Resampling)
: Instantiated Resampling object which stores the splits into training and test.

`learners` (list of Learner)
: List of trained learners, sorted by resampling iteration.

`data_extra` (list())
: Additional data stored in the ResampleResult.

`warnings` (`data.table::data.table()`)
: 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()`)
: 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.

Methods

Public methods:

- `ResampleResult$new()`
- `ResampleResult$format()`
- `ResampleResult$print()`
- `ResampleResult$help()`
- `ResampleResult$prediction()`
- `ResampleResult$predictions()`
- `ResampleResult$score()`
- `ResampleResult$obs_loss()`
- `ResampleResult$aggregate()`
- `ResampleResult$filter()`
- `ResampleResult$discard()`
- `ResampleResult$marshal()`
- `ResampleResult$unmarshal()`
- `ResampleResult$set_threshold()`
- `ResampleResult$clone()`

Method `new()`**:** Creates a new instance of this R6 class. An alternative construction method is provided by `as_resample_result()`.

Usage:

```
ResampleResult$new(data = ResultData$new(), view = NULL)
```

Arguments:

`data` (ResultData | `data.table()`)
: An object of type ResultData, either extracted from another ResampleResult, another BenchmarkResult, or manually constructed with `as_result_data()`.

`view` (`character()`)
Single `uhash` of the ResultData to operate on. Used internally for optimizations.

Method `format()`: Helper for print outputs.

Usage:

```
ResampleResult$format(...)
```

Arguments:

`...` (ignored).

Method `print()`: Printer.

Usage:

```
ResampleResult$print(...)
```

Arguments:

`...` (ignored).

Method `help()`: Opens the corresponding help page referenced by field `$man`.

Usage:

```
ResampleResult$help()
```

Method `prediction()`: Combined Prediction of all individual resampling iterations, and all provided predict sets. Note that, per default, most performance measures do not operate on this object directly, but instead on the prediction objects from the resampling iterations separately, and then combine the performance scores with the aggregate function of the respective Measure (macro averaging).
If you calculate the performance on this prediction object directly, this is called micro averaging.

Usage:

```
ResampleResult$prediction(predict_sets = "test")
```

Arguments:

`predict_sets` (`character()`)
Subset of `{"train", "test"}`.

Returns: Prediction or empty `list()` if no predictions are available.

Examples:

```
rr$prediction()
```

Method `predictions()`: List of prediction objects, sorted by resampling iteration. If multiple sets are given, these are combined to a single one for each iteration.
If you evaluate the performance on all of the returned prediction objects and then average them, this is called macro averaging. For micro averaging, operate on the combined prediction object as returned by `$prediction()`.

Usage:

```
ResampleResult$predictions(predict_sets = "test")
```

Arguments:

`predict_sets` (`character()`)
Subset of `{"train", "test", "internal_valid"}`.

Returns: List of Prediction objects, one per element in `predict_sets`. Or list of empty `list()`s if no predictions are available.

Examples:

```
rr$predictions()
```

Method `score()`**:** Returns a table with one row for each resampling iteration, including all involved objects: Task, Learner, Resampling, iteration number (`integer(1)`), and (if enabled) one Prediction for each predict set of the Learner. Additionally, a column with the individual (per resampling iteration) performance is added for each Measure in `measures`, named with the id of the respective measure id. If `measures` is `NULL`, `measures` defaults to the return value of `default_measures()`.

Usage:

```
ResampleResult$score(
  measures = NULL,
  ids = TRUE,
  conditions = FALSE,
  predictions = TRUE
)
```

Arguments:

`measures` (Measure | list of Measure)
: Measure(s) to calculate.

`ids` (`logical(1)`)
: If `ids` is `TRUE`, extra columns with the ids of objects (`"task_id"`, `"learner_id"`, `"resampling_id"`) are added to the returned table. These allow to subset more conveniently.

`conditions` (`logical(1)`)
: Adds condition messages (`"warnings"`, `"errors"`) as extra list columns of character vectors to the returned table

`predictions` (`logical(1)`)
: Additionally return prediction objects, one column for each `predict_set` of the learner. Columns are named `"prediction_train"`, `"prediction_test"` and `"prediction_internal_valid"`, if present.

Returns: `data.table::data.table()`.

Examples:

```
rr$score(msr("classif.acc"))
```

Method `obs_loss()`**:** Calculates the observation-wise loss via the loss function set in the Measure's field `obs_loss`. Returns a `data.table()` with the columns of the matching Prediction object plus one additional numeric column for each measure, named with the respective measure id. If there is no observation-wise loss function for the measure, the column is filled with `NA` values. Note that some measures such as RMSE, do have an `$obs_loss`, but they require an additional transformation after aggregation, in this example taking the square-root.

Usage:

```
ResampleResult$obs_loss(measures = NULL, predict_sets = "test")
```

Arguments:

measures (Measure | list of Measure)
Measure(s) to calculate.

predict_sets (character())
The predict sets.

Examples:

```
rr$obs_loss(msr("classif.acc"))
```

Method `aggregate()`**:** 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()`.

Usage:

```
ResampleResult$aggregate(measures = NULL)
```

Arguments:

measures (Measure | list of Measure)
Measure(s) to calculate.

Returns: Named `numeric()`.

Examples:

```
rr$aggregate(msr("classif.acc"))
```

Method `filter()`**:** Subsets the ResampleResult, reducing it to only keep the iterations specified in `iters`.

Usage:

```
ResampleResult$filter(iters)
```

Arguments:

iters (integer())
Resampling iterations to keep.

Returns: Returns the object itself, but modified **by reference**. You need to explicitly `$clone()` the object beforehand if you want to keeps the object in its previous state.

Examples:

```
rr$filter(1L)
```

Method `discard()`**:** Shrinks the ResampleResult by discarding parts of the internally stored data. Note that certain operations might stop work, e.g. extracting importance values from learners or calculating measures requiring the task's data.

Usage:

```
ResampleResult$discard(backends = FALSE, models = FALSE)
```

Arguments:

backends (logical(1))
If `TRUE`, the DataBackend is removed from all stored Tasks.

models (logical(1))
If `TRUE`, the stored model is removed from all Learners.

Returns: Returns the object itself, but modified **by reference**. You need to explicitly `$clone()` the object beforehand if you want to keeps the object in its previous state.

Method `marshal()`**:** Marshals all stored models.

Usage:

```
ResampleResult$marshal(...)
```

Arguments:

`...` (any)
Additional arguments passed to `marshal_model()`.

Examples:

```
rr$marshal()
```

Method `unmarshal()`**:** Unmarshals all stored models.

Usage:

```
ResampleResult$unmarshal(...)
```

Arguments:

`...` (any)
Additional arguments passed to `unmarshal_model()`.

Examples:

```
rr$unmarshal()
```

Method `set_threshold()`**:** Sets the threshold for the response prediction of classification learners, given they have output a probability prediction for a binary classification task. This modifies the object in-place.

Usage:

```
ResampleResult$set_threshold(threshold, ties_method = "random")
```

Arguments:

`threshold (numeric(1))`
Threshold value.

`ties_method (character(1))`
Method to handle ties in probabilities when selecting a class label. Must be one of `"random"`, `"first"` or `"last"` (corresponding to the same options in `max.col()`).

- `"random"`: Randomly select one of the tied class labels (default).
- `"first"`: Select the first class label among tied values.
- `"last"`: Select the last class label among tied values.

Examples:

```
learner = lrn("classif.rpart", predict_type = "prob")
rr = resample(tsk("sonar"), learner, rsmp("cv", folds = 3))
rr$set_threshold(0.6)
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- as_benchmark_result() to convert to a BenchmarkResult.
- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-resampling
- Package **mlr3viz** for some generic visualizations.

Other resample: resample()

Examples

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

# combined predictions and predictions for each fold separately
rr$prediction()
rr$predictions()

# folds scored separately, then aggregated (macro)
rr$aggregate(msr("classif.acc"))

# predictions first combined, then scored (micro)
rr$prediction()$score(msr("classif.acc"))

# check for warnings and errors
rr$warnings
rr$errors

## ------------------------------------------------
## Method `ResampleResult$prediction`
## ------------------------------------------------

rr$prediction()

## ------------------------------------------------
## Method `ResampleResult$predictions`
## ------------------------------------------------

rr$predictions()

## ------------------------------------------------
## Method `ResampleResult$score`
## ------------------------------------------------

rr$score(msr("classif.acc"))

## ------------------------------------------------
## Method `ResampleResult$obs_loss`
## ------------------------------------------------
```

```
rr$obs_loss(msr("classif.acc"))

## ------------------------------------------------
## Method `ResampleResult$aggregate`
## ------------------------------------------------

rr$aggregate(msr("classif.acc"))

## ------------------------------------------------
## Method `ResampleResult$filter`
## ------------------------------------------------

rr$filter(1L)

## ------------------------------------------------
## Method `ResampleResult$marshal`
## ------------------------------------------------

rr$marshal()

## ------------------------------------------------
## Method `ResampleResult$unmarshal`
## ------------------------------------------------

rr$unmarshal()

## ------------------------------------------------
## Method `ResampleResult$set_threshold`
## ------------------------------------------------

learner = lrn("classif.rpart", predict_type = "prob")
rr = resample(tsk("sonar"), learner, rsmp("cv", folds = 3))
rr$set_threshold(0.6)
```

`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 dictionary mlr_resamplings, e.g. `cv` or `bootstrap`.

Stochasticity & Reproducibility

The `Resampling` class only defines an abstract resampling strategy. Concrete data splits are obtained by calling `$instantiate()` on a `Task`. To ensure repdocubility of results, you need to call `set.seed` before doing so. Note that `benchmark_grid` internally does instantiate resamplings, so you need to set the seed before calling it.

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. Note that this procedure can lead to set sizes that are slightly different from those without stratification.

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.

Inheriting

It is possible to overwrite both `private$.get_instance()` to have full control, or only `private$.sample()` when one wants to use the pre-defined mechanism for stratification and grouping.

Public fields

`label` (`character(1)`)
: Label for this object. Can be used in tables, plot and text output instead of the ID.

`param_set` (paradox::ParamSet)
: Set of hyperparameters.

`instance` (any)
: During `instantiate()`, the instance is stored in this slot in an arbitrary format. Note that if a grouping variable is present in the Task, a Resampling may operate on the group ids internally instead of the row ids (which may lead to confusion).

  It is advised to not work directly with the `instance`, but instead only use the getters `$train_set()` and `$test_set()`.

`task_hash` (`character(1)`)
: The hash of the Task which was passed to `r$instantiate()`.

`task_row_hash` (`character(1)`)
: The hash of the row ids of the Task which was passed to `r$instantiate()`.

`task_nrow` (`integer(1)`)
: The number of observations of the Task which was passed to `r$instantiate()`.

`duplicated_ids` (`logical(1)`)
: If `TRUE`, duplicated rows can occur within a single training set or within a single test set. E.g., this is `TRUE` for Bootstrap, and `FALSE` for cross-validation. Only used internally.

`man` (`character(1)`)
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. Defaults to `NA`, but can be set by child classes.

Active bindings

`id` (`character(1)`)
: Identifier of the object. Used in tables, plot and text output.

`is_instantiated` (`logical(1)`)
: Is `TRUE` if the resampling has been instantiated.

`hash` (`character(1)`)
: Hash (unique identifier) for this object. If the object has not been instantiated yet, `NA_character_` is returned. The hash is calculated based on the class name, the id, the parameter set, and the instance.

Methods

Public methods:

- `Resampling$new()`
- `Resampling$format()`
- `Resampling$print()`
- `Resampling$help()`
- `Resampling$instantiate()`
- `Resampling$train_set()`
- `Resampling$test_set()`
- `Resampling$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

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

Arguments:

`id (character(1))`
Identifier for the new instance.

`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.
Note that this object is typically constructed via a derived classes, e.g. ResamplingCV or ResamplingHoldout.

`label (character(1))`
Label for the new instance.

`man (character(1))`
String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

Method `format()`**:** Helper for print outputs.

Usage:

```
Resampling$format(...)
```

Arguments:

`...` (ignored).

Method `print()`**:** Printer.

Usage:

```
Resampling$print(...)
```

Arguments:

`...` (ignored).

Method `help()`**:** Opens the corresponding help page referenced by field `$man`.

Usage:

```
Resampling$help()
```

Method `instantiate()`**:** Materializes fixed training and test splits for a given task and stores them in `r$instance` in an arbitrary format.

Usage:

```
Resampling$instantiate(task)
```

Arguments:

`task` (Task)
Task used for instantiation.

Returns: Returns the object itself, but modified **by reference**. You need to explicitly `$clone()` the object beforehand if you want to keeps the object in its previous state.

Examples:

```
task = tsk("penguins")
resampling = rsmp("holdout")
resampling$instantiate(task)
```

Method `train_set()`: Returns the row ids of the i-th training set.

Usage:

```
Resampling$train_set(i)
```

Arguments:

`i` (`integer(1)`)
Iteration.

Returns: (`integer()`) of row ids.

Examples:

```
task = tsk("penguins")
resampling = rsmp("holdout")$instantiate(task)
resampling$train_set(1)
```

Method `test_set()`: Returns the row ids of the i-th test set.

Usage:

```
Resampling$test_set(i)
```

Arguments:

`i` (`integer(1)`)
Iteration.

Returns: (`integer()`) of row ids.

Examples:

```
task = tsk("penguins")
resampling = rsmp("holdout")$instantiate(task)
resampling$test_set(1)
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-resampling
- Package **mlr3spatiotempcv** for spatio-temporal resamplings.
- Dictionary of Resamplings: mlr_resamplings
- `as.data.table(mlr_resamplings)` for a table of available Resamplings in the running session (depending on the loaded packages).
- **mlr3spatiotempcv** for additional Resamplings for spatio-temporal tasks.

Other Resampling: `mlr_resamplings`, `mlr_resamplings_bootstrap`, `mlr_resamplings_custom`, `mlr_resamplings_custom_cv`, `mlr_resamplings_cv`, `mlr_resamplings_holdout`, `mlr_resamplings_insample`, `mlr_resamplings_loo`, `mlr_resamplings_repeated_cv`, `mlr_resamplings_subsampling`

Examples

```
r = rsmp("subsampling")

# Default parametrization
r$param_set$values

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

# Instantiate on penguins task
task = tsk("penguins")
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

## ------------------------------------------------
## Method `Resampling$instantiate`
## ------------------------------------------------

task = tsk("penguins")
resampling = rsmp("holdout")
resampling$instantiate(task)

## ------------------------------------------------
## Method `Resampling$train_set`
## ------------------------------------------------

task = tsk("penguins")
resampling = rsmp("holdout")$instantiate(task)
resampling$train_set(1)

## ------------------------------------------------
## Method `Resampling$test_set`
## ------------------------------------------------
```

```
task = tsk("penguins")
resampling = rsmp("holdout")$instantiate(task)
resampling$test_set(1)
```

`score_roc_measures` *Calculate ROC Measures*

Description

Calculate a set of roc performance measures based on the confusion matrix.

- `tpr` True positive rate (Sensitivity, Recall)
- `fpr` False positive rate (Fall-out)
- `fnr` False negative rate (Miss rate)
- `tnr` True negative rate (Specificity)
- `ppv` Positive predictive value (Precision)
- `fomr` False omission rate
- `lrp` Positive likelihood ratio (LR+)
- `fdr` False discovery rate
- `npv` Negative predictive value
- `acc` Accuracy
- `lrm` Negative likelihood ratio (LR-)
- `dor` Diagnostic odds ratio

Usage

```
score_roc_measures(pred)
```

Arguments

`pred`	(PredictionClassif) The prediction object.

Value

`list()`
A list containing two elements `confusion_matrix` which is the 2 times 2 confusion matrix of absolute frequencies and `measures`, a list of the above mentioned measures.

Examples

```
learner = lrn("classif.rpart", predict_type = "prob")
splits = partition(task = tsk("pima"), ratio = 0.7)
task = tsk("pima")
learner$train(task)
pred = learner$predict(task)
score_roc_measures(pred)
```

`set_threads` *Set the Number of Threads*

Description

Control the parallelism via threading while calling external packages from **mlr3**.

For example, the random forest implementation in package **ranger** (connected via **mlr3learners**) supports threading via OpenMP. The number of threads to use can be set via hyperparameter `num.threads`, and defaults to 1. By calling `set_threads(x, 4)` with `x` being a ranger learner, the hyperparameter is changed so that 4 cores are used.

If the object `x` does not support threading, `x` is returned as-is. If applied to a list, recurses through all list elements.

Note that threading is incompatible with other parallelization techniques such as forking via the future::plan future::multicore. For this reason all learners connected to **mlr3** have threading disabled in their defaults.

Usage

```
set_threads(x, n = availableCores(), ...)

## Default S3 method:
set_threads(x, n = availableCores(), ...)

## S3 method for class 'R6'
set_threads(x, n = availableCores(), ...)

## S3 method for class 'list'
set_threads(x, n = availableCores(), ...)
```

Arguments

`x`	(any) Object to set threads for, e.g. a Learner. This object is modified in-place.
`n`	(`integer(1)`) Number of threads to use. Defaults to `parallelly::availableCores()`.
`...`	(any) Additional arguments.

Value

Same object as input `x` (changed in-place), with possibly updated parameter values.

`Task` *Task Class*

Description

This is the abstract base class for TaskSupervised and TaskUnsupervised. TaskClassif and TaskRegr inherit from TaskSupervised. More supervised tasks are implemented in **mlr3proba**, unsupervised cluster tasks in package **mlr3cluster**.

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 dictionary mlr_tasks, e.g. `penguins` or `california_housing`. More toy tasks can be found in the dictionary after loading **mlr3data**.

S3 methods

- `as.data.table(t)`
  Task -> `data.table::data.table()`
  Returns the complete data as `data.table::data.table()`.
- `head(t)`
  Calls `head()` on the task's data.
- `summary(t)`
  Calls `summary()` on the task's data.

Task mutators

The following methods change the task in-place:

- Any modification of the lists `$col_roles` or `$row_roles`. This provides a different "view" on the data without altering the data itself. This may affects, e.g., `$data`, `$nrow`, `$ncol`, `n_features`, `row_ids`, and `$feature_names`. Altering `$col_roles` may affect, e.g., `$data`, `$ncol`, `$n_features`, and `$feature_names`. Altering `$row_roles` may affect, e.g., `$data`, `$nrow`, and `$row_ids`.
- Modification of column or row roles via `$set_col_roles()` or `$set_row_roles()`, respectively. They are an alternative to directly accessing `$col_roles` or `$row_roles`, with the same side effects.
- `$select()` and `$filter()` subset the set of active features or rows in `$col_roles` or `$row_roles`, respectively.
- `$cbind()` and `$rbind()` change the task in-place by binding new columns or rows to the data.
- `$rename()` changes column names.
- `$set_levels()` and `$droplevels()` update the field `$col_info()` to automatically repair factor levels while querying data with `$data()`.

- `$materialize_view()` creates a new DataBackendDataTable which keeps only the data in the currently active view possibly freeing some memory consumed by the DataBackend stored in the `Task`.

Public fields

`label` (`character(1)`)
Label for this object. Can be used in tables, plot and text output instead of the ID.

`task_type` (`character(1)`)
Task type, e.g. `"classif"` or `"regr"`.

For a complete list of possible task types (depending on the loaded packages), see `mlr_reflections$task_types$type`

`backend` (DataBackend)
Abstract interface to the data of the task.

`col_info` (`data.table::data.table()`)
Table with with 4 columns, mainly for internal purposes:

- `"id"` (`character()`) stores the name of the column.
- `"type"` (`character()`) holds the storage type of the variable, e.g. `integer`, `numeric` or `character`. See mlr_reflections$task_feature_types for a complete list of allowed types.
- `"levels"` (`list()`) stores a vector of distinct values (levels) for ordered and unordered factor variables.
- `"label"` (`character()`) stores a vector of prettier, formated column names.
- `"fix_factor_levels"` (`logical()`) stores flags which determine if the levels of the respective variable need to be reordered after querying the data from the DataBackend.

Note that all columns of the DataBackend, also columns which are not selected or have any role, are listed in this table.

`man` (`character(1)`)
String in the format `[pkg]::[topic]` pointing to a manual page for this object. Defaults to `NA`, but can be set by child classes.

`extra_args` (named `list()`)
Additional arguments set during construction. Required for `convert_task()`.

`mlr3_version` (`package_version`)
Package version of `mlr3` used to create the task.

Active bindings

`id` (`character(1)`)
Identifier of the object. Used in tables, plot and text output.

`internal_valid_task` (`Task` or `integer()` or `NULL`)
Optional validation task that can, e.g., be used for early stopping with learners such as XGBoost. See also the `$validate` field of `Learner`. If integers are assigned they are removed from the primary task and an internal validation task with those ids is created from the primary task using only those ids. When assigning a new task, it is always cloned.

`hash` (`character(1)`)
Hash (unique identifier) for this object. The hash is calculated based on the complete task object and `$row_ids`. If an internal validation task is set, the hash is recalculated.

`row_hash` (`character(1)`)
Hash (unique identifier) calculated based on the row ids.

`row_ids` (positive `integer()`)
Returns the row ids of the DataBackend for observations with role "use".

`row_names` (`data.table::data.table()`)
Returns a table with two columns:

- `"row_id"` (`integer()`), and
- `"row_name"` (`character()`).

`feature_names` (`character()`)
Returns all column names with `role == "feature"`.

Note that this vector determines the default order of columns for `task$data(cols = NULL, ...)`. However, it is recommended to **not** rely on the order of columns, but instead always address columns by their name. The default order is not well defined after some operations, e.g. after `task$cbind()` or after processing via **mlr3pipelines**.

`target_names` (`character()`)
Returns all column names with role "target".

`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_learner"`: If the task has observation weights with this role, they are passed to the Learner during train. The use of weights can be disabled via by setting the learner's hyperparameter `use_weights` to `FALSE`.
- `"weights_measure"`: If the task has observation weights with this role, they are passed to the Measure for weighted scoring. The use of weights can be disabled via by setting the measure's hyperparameter `use_weights` to `FALSE`.
- `"offset"`: The task includes one or more offset columns specifying fixed adjustments for model training and possibly for prediction (role `"offset"`).
- `"ordered"`: The task has columns which define the row order (role `"order"`).

Note that above listed properties are calculated from the `$col_roles`, and may not be set explicitly.

`row_roles` (named `list()`)
Each row (observation) can have an arbitrary number of roles in the learning task:

- `"use"`: Use in train / predict / resampling.

`row_roles` is a named list whose elements are named by row role and each element is an `integer()` vector of row ids. To alter the roles, just modify the list, e.g. with R's set functions (`intersect()`, `setdiff()`, `union()`, ...).

`col_roles` (named `list()`)
Each column can be in one or more of the following groups to fulfill different roles:

- `"feature"`: Regular feature used in the model fitting process.
- `"target"`: Target variable. Most tasks only accept a single target column.
- `"name"`: Row names / observation labels. To be used in plots. Can be queried with `$row_names`. Not more than a single column can be associated with this role.

- `"order"`: Data returned by `$data()` is ordered by this column (or these columns). Columns must be sortable with `order()`.
- `"group"`: During resampling, observations with the same value of the variable with role "group" are marked as "belonging together". For each resampling iteration, observations of the same group will be exclusively assigned to be either in the training set or in the test set. Not more than a single column can be associated with this role.
- `"stratum"`: Stratification variables. Multiple discrete columns may have this role.
- `"weights_learner"`: If the task has observation weights with this role, they are passed to the Learner during train. The use of weights can be disabled via by setting the learner's hyperparameter `use_weights` to `FALSE`.
- `"weights_measure"`: If the task has observation weights with this role, they are passed to the Measure for weighted scoring. The use of weights can be disabled via by setting the measure's hyperparameter `use_weights` to `FALSE`.
- `"offset"`: Numeric columns used to specify fixed adjustments for model training. Some models use offsets to simply shift predictions, while others incorporate them to boost predictions from a baseline model. For learners supporting offsets in multiclass settings, an offset column must be provided for each target class. These columns must follow the naming convention `"offset_{target_class_name}"`. For an example of a learner that supports offsets, see `LearnerClassifXgboost` of **mlr3learners**.

`col_roles` is a named list whose 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()`, ...). The method `$set_col_roles` provides a convenient alternative to assign columns to roles.

The roles `weights_learner` and `weights_measure` may only point to a single numeric column, but they can all point to the same column or different columns. Weights must be non-negative numerics with at least one weight being > 0. They don't necessarily need to sum up to 1.

`nrow (integer(1))`
Returns the total number of rows with role "use".

`ncol (integer(1))`
Returns the total number of columns with role "target" or "feature".

`n_features (integer(1))`
Returns the total number of columns with role "feature" (i.e. the number of "active" features in the task).

`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.

`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()`) 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"`, a table with two columns:

- row_id (integer()), and
- grouping variable group (vector()).

Returns NULL if there are is no grouping column. See Resampling for more information on grouping.

order (data.table::data.table())
If the task has at least one column with designated role "order", a table with two columns:

- row_id (integer()), and
- ordering vector order (integer()).

Returns NULL if there are is no order column.

weights (data.table::data.table())
Deprecated, use $weights_learner instead.

weights_learner (data.table::data.table())
Returns the observation weights used for training a Learner (column role weights_learner) as a data.table with the following columns:

- row_id (integer()), and
- weight (numeric()).

Returns NULL if there are is no column with the designated role.

weights_measure (data.table::data.table())
Returns the observation weights used for scoring a prediction with a Measure (column role weights_measure) as a data.table with the following columns:

- row_id (integer()), and
- weight (numeric()).

Returns NULL if there are is no column with the designated role.

offset (data.table::data.table())
If the task has a column with designated role "offset", a table with two or more columns:

- row_id (integer()), and
- offset variable(s) (numeric()).

For regression or binary classification tasks, there will be only a single-column offset. For multiclass tasks, it may return multiple offset columns, one for each target class. If there is only one offset column, it will be named as offset.

If there are no columns with the "offset" role, NULL is returned.

labels (named character())
Retrieve labels (prettier formated names) from columns. Internally queries the column label of the table in field col_info. Columns ids referenced by the name of the vector, the labels are the actual string values.

Assigning to this column update the task by reference. You have to provide a character vector of labels, named with column ids. To remove a label, set it to NA. Alternatively, you can provide a data.frame() with the two columns "id" and "label".

col_hashes (named character)
Hash (unique identifier) for all columns except the primary_key: A character vector, named by the columns that each element refers to.
Columns of different Tasks or DataBackends that have agreeing col_hashes always represent the same data, given that the same rows are selected. The reverse is not necessarily true: There can be columns with the same content that have different col_hashes.

`characteristics (list())`
: List of characteristics of the task, e.g. `list(n = 5, p = 7)`.

`row_ids_backend (integer())`
: Returns all row ids from the backend, regardless of their roles. This is different from `$row_ids` which only returns rows with role "use".

Methods

Public methods:

- `Task$new()`
- `Task$divide()`
- `Task$help()`
- `Task$format()`
- `Task$print()`
- `Task$data()`
- `Task$formula()`
- `Task$head()`
- `Task$levels()`
- `Task$missings()`
- `Task$filter()`
- `Task$select()`
- `Task$rbind()`
- `Task$cbind()`
- `Task$rename()`
- `Task$set_row_roles()`
- `Task$set_col_roles()`
- `Task$set_levels()`
- `Task$droplevels()`
- `Task$add_strata()`
- `Task$materialize_view()`
- `Task$clone()`

Method `new()`**:** Creates a new instance of this R6 class.
Note that this object is typically constructed via a derived classes, e.g. TaskClassif or TaskRegr.

Usage:

```
Task$new(id, task_type, backend, label = NA_character_, extra_args = list())
```

Arguments:

`id (character(1))`
: Identifier for the new instance.

`task_type (character(1))`
: Type of task, e.g. `"regr"` or `"classif"`. 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.

`label` (`character(1)`)
Label for the new instance.

`extra_args` (named `list()`)
Named list of constructor arguments, required for converting task types via `convert_task()`.

Method `divide()`**:** Deprecated.

Usage:

```
Task$divide(ratio = NULL, ids = NULL, remove = TRUE)
```

Arguments:

`ratio` (`numeric(1)`)
The proportion of datapoints to use as validation data.

`ids` (`integer()`)
The row ids to use as validation data.

`remove` (`logical(1)`)
If `TRUE` (default), the `row_ids` are removed from the primary task's active `"use"` rows, ensuring a disjoint split between the train and validation data.

Returns: Modified `Self`.

Method `help()`**:** Opens the corresponding help page referenced by field `$man`.

Usage:

```
Task$help()
```

Method `format()`**:** Helper for print outputs.

Usage:

```
Task$format(...)
```

Arguments:

`...` (ignored).

Method `print()`**:** Printer.

Usage:

```
Task$print(...)
```

Arguments:

`...` (ignored).

Method `data()`**:** Returns a slice of the data from the DataBackend as a `data.table`. Rows default to observations with role `"use"`, and columns default to features with roles `"target"` or `"feature"`. Rows must be a subset of `$row_ids`. If `rows` or `cols` are specified which do not exist in the DataBackend, an exception is raised.

Rows and columns are returned in the order specified via the arguments `rows` and `cols`. If `rows` is `NULL`, rows are returned in the order of `task$row_ids`. If `cols` is `NULL`, the column order defaults to `c(task$target_names, task$feature_names)`. Note that it is recommended to **not** rely on the order of columns, and instead always address columns with their respective column name.

Usage:

```
Task$data(rows = NULL, cols = NULL, ordered = FALSE)
```

Arguments:

`rows` (positive `integer()`)
: Vector or row indices. Always refers to the complete data set, even after filtering.

`cols` (`character()`)
: Vector of column names.

`ordered` (`logical(1)`)
: If `TRUE`, data is ordered according to the columns with column role `"order"`.

Returns: Depending on the DataBackend, but usually a `data.table::data.table()`.

Examples:

```
task = tsk("penguins")
task$data(rows = 1:5, cols = c("species", "sex"))
```

Method `formula()`**:** Constructs a `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 `"."`).
Note that it is currently not possible to change the formula. However, **mlr3pipelines** provides a pipe operator interfacing `stats::model.matrix()` for this purpose: `"modelmatrix"`.

Usage:

```
Task$formula(rhs = ".")
```

Arguments:

`rhs` (`character(1)`)
: Right hand side of the formula. Defaults to `"."` (all features of the task).

Returns: `formula()`.

Examples:

```
task = tsk("penguins")
task$formula()
```

Method `head()`**:** Get the first `n` observations with role `"use"` of all columns with role `"target"` or `"feature"`.

Usage:

```
Task$head(n = 6L)
```

Arguments:

`n` (`integer(1)`).

Returns: `data.table::data.table()` with `n` rows.

Examples:

```
task = tsk("penguins")
task$head(3)
```

Method `levels()`**:** Returns the distinct values for columns referenced in `cols` with storage type "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 subsetting a task with `$filter()`, call `$droplevels()`.

Usage:

```
Task$levels(cols = NULL)
```

Arguments:

`cols` (`character()`)
Vector of column names.

Returns: named `list()`.

Examples:

```
task = tsk("penguins")
task$levels()
```

Method `missings()`**:** 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".

Usage:

```
Task$missings(cols = NULL)
```

Arguments:

`cols` (`character()`)
Vector of column names.

Returns: Named `integer()`.

Examples:

```
task = tsk("penguins")
task$missings()
```

Method `filter()`**:** Subsets the task, keeping only the rows specified via row ids `rows`.
This operation mutates the task in-place. See the section on task mutators for more information.

Usage:

```
Task$filter(rows)
```

Arguments:

`rows` (positive `integer()`)
Vector or row indices. Always refers to the complete data set, even after filtering.

Returns: Returns the object itself, but modified **by reference**. You need to explicitly `$clone()` the object beforehand if you want to keeps the object in its previous state.

Examples:

```
task = tsk("penguins")
task$filter(1:10)
task$nrow
```

Method `select()`**:** Subsets the task, keeping only the features specified via column names `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.

Usage:

```
Task$select(cols)
```

Arguments:

`cols (character())`
Vector of column names.

Returns: Returns the object itself, but modified **by reference**. You need to explicitly `$clone()` the object beforehand if you want to keeps the object in its previous state.

Examples:

```
task = tsk("penguins")
task$select(c("bill_length", "bill_depth"))
task$feature_names
```

Method `rbind()`**:** Adds additional rows to the DataBackend stored in `$backend`. New row ids are automatically created, unless `data` has a column whose name matches the primary key of the DataBackend (`task$backend$primary_key`). In case of name clashes of row ids, rows in `data` have higher precedence and virtually overwrite the rows in the DataBackend.

All columns roles `"target"`, `"feature"`, `"weights_learner"`, `"weights_measure"`, `"group"`, `"stratum"`, and `"order"` must be present in `data`. Columns only present in `data` but not in the DataBackend of `task` will be discarded.

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

Usage:

```
Task$rbind(data)
```

Arguments:

`data (data.frame())`.

Returns: Returns the object itself, but modified **by reference**. You need to explicitly `$clone()` the object beforehand if you want to keeps the object in its previous state.

Examples:

```
task = tsk("penguins")
extra = task$data(rows = 1:2)
task$rbind(extra)
```

Method `cbind()`**:** Adds additional columns to the DataBackend stored in `$backend`.

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 `$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.

Usage:

```
Task$cbind(data)
```

Arguments:

`data (data.frame())`.

Examples:

```
task = tsk("penguins")
task$cbind(data.table(extra_col = seq_len(task$nrow)))
head(task$data(cols = "extra_col"))
```

Method `rename()`**:** Renames columns by mapping column names in `old` to new column names in `new` (element-wise).

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

Usage:

```
Task$rename(old, new)
```

Arguments:

`old` (`character()`)
: Old names.

`new` (`character()`)
: New names.

Returns: Returns the object itself, but modified **by reference**. You need to explicitly `$clone()` the object beforehand if you want to keeps the object in its previous state.

Examples:

```
task = tsk("penguins")
task$rename("body_mass", "mass")
task$feature_names
```

Method `set_row_roles()`**:** Modifies the roles in `$row_roles` **in-place**.

Usage:

```
Task$set_row_roles(rows, roles = NULL, add_to = NULL, remove_from = NULL)
```

Arguments:

`rows` (`integer()`)
: Row ids for which to change the roles for.

`roles` (`character()`)
: Exclusively set rows to the specified `roles` (remove from other roles).

`add_to` (`character()`)
: Add rows with row ids `rows` to roles specified in `add_to`. Rows keep their previous roles.

`remove_from` (`character()`)
: Remove rows with row ids `rows` from roles specified in `remove_from`. Other row roles are preserved.

Details: Roles are first set exclusively (argument `roles`), then added (argument `add_to`) and finally removed (argument `remove_from`) from different roles. Duplicated row ids are explicitly allowed, so you can add replicate an observation by repeating its `row_id`.

Returns: Returns the object itself, but modified **by reference**. You need to explicitly `$clone()` the object beforehand if you want to keeps the object in its previous state.

Examples:

```
task = tsk("penguins")
task$set_row_roles(1:5, remove_from = "use")
```

Method `set_col_roles()`**:** Modifies the roles in `$col_roles` **in-place**. See `$col_roles` for a list of possible roles.

Usage:

```
Task$set_col_roles(cols, roles = NULL, add_to = NULL, remove_from = NULL)
```

Arguments:

`cols` (`character()`)
Column names for which to change the roles for.

`roles` (`character()`)
Exclusively set columns to the specified `roles` (remove from other roles).

`add_to` (`character()`)
Add columns with column names `cols` to roles specified in `add_to`. Columns keep their previous roles.

`remove_from` (`character()`)
Remove columns with columns names `cols` from roles specified in `remove_from`. Other column roles are preserved.

Details: Roles are first set exclusively (argument `roles`), then added (argument `add_to`) and finally removed (argument `remove_from`) from different roles. Duplicated columns are removed from the same role. For tasks that only allow one target, the target column cannot be set with `$set_col_roles()`. Use the `$col_roles` field to swap the target column.

Returns: Returns the object itself, but modified **by reference**. You need to explicitly `$clone()` the object beforehand if you want to keeps the object in its previous state.

Examples:

```
task = tsk("penguins")
task$set_col_roles("sex", roles = "stratum")
task$col_roles$stratum
```

Method `set_levels()`**:** Set levels for columns of type `factor` and `ordered` in field `col_info`. You can add, remove or reorder the levels, affecting the data returned by `$data()` and `$levels()`. If you just want to remove unused levels, use `$droplevels()` instead.

Note that factor levels which are present in the data but not listed in the task as valid levels are converted to missing values.

Usage:

```
Task$set_levels(levels)
```

Arguments:

`levels` (named `list()` of `character()`)
List of character vectors of new levels, named by column names.

Returns: Modified `self`.

Examples:

```
task = tsk("penguins")
task$set_levels(list(sex = c("male", "female", "unknown")))
task$levels("sex")
```

Method `droplevels()`**:** 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 "factor" or "ordered".

Usage:

```
Task$droplevels(cols = NULL)
```

Arguments:

`cols` (`character()`)
Vector of column names.

Returns: Modified `self`.

Examples:

```
task = tsk("penguins")
task$set_levels(list(sex = c("male", "female", "unknown")))
task$levels("sex")
```

Method `add_strata()`**:** Cuts numeric variables into new factors columns which are added to the task with role `"stratum"`. This ensures that all training and test splits contain observations from all bins. The columns are named `"..stratum_[col_name]"`.

Usage:

```
Task$add_strata(cols, bins = 3L)
```

Arguments:

`cols` (`character()`)
Names of columns to operate on.

`bins` (`integer()`)
Number of bins to cut into (passed to `cut()` as `breaks`). Replicated to have the same length as `cols`.

Returns: self (invisibly).

Examples:

```
task = tsk("penguins")
task$add_strata("flipper_length", bins = 4)
```

Method `materialize_view()`**:** Certain operations change the view on the data, e.g., `$filter()` or `$select()`. This operation queries the DataBackend for all data required in the active view and replaces the internal DataBackend with the new one. In some scenarios this helps to free up memory or speeds up accesses to the data, especially after several `$rbind()` and `$cbind()` operations.

Usage:

```
Task$materialize_view(internal_valid_task = TRUE)
```

Arguments:

`internal_valid_task` (`logical(1)`)
Also materialize the internal validation task. Default is `TRUE`.

Details: For tasks containing the same observation more than once (duplicates in `$row_ids`), the resulting backend contains it only once.

Returns: self (invisibly).

Examples:

```
task = tsk("iris")
task$backend$nrow
task$filter(1:120)
task$backend$nrow
```

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3data** for more toy tasks.
- Package **mlr3oml** for downloading tasks from https://www.openml.org.
- Package **mlr3viz** for some generic visualizations.
- Dictionary of Tasks: mlr_tasks
- `as.data.table(mlr_tasks)` for a table of available Tasks in the running session (depending on the loaded packages).
- **mlr3fselect** and **mlr3filters** for feature selection and feature filtering.
- Extension packages for additional task types:
  - Unsupervised clustering: **mlr3cluster**
  - Probabilistic supervised regression and survival analysis: https://mlr3proba.mlr-org.com/.

Other Task: TaskClassif, TaskRegr, TaskSupervised, TaskUnsupervised, california_housing, mlr_tasks, mlr_tasks_breast_cancer, mlr_tasks_german_credit, mlr_tasks_iris, mlr_tasks_mtcars, mlr_tasks_penguins, mlr_tasks_pima, mlr_tasks_sonar, mlr_tasks_spam, mlr_tasks_wine, mlr_tasks_zoo

Examples

```
# We use the inherited class TaskClassif here,
# because the base class `Task` is not intended for direct use
task = TaskClassif$new("penguings", palmerpenguins::penguins, target = "species")

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

# de-select "year"
task$select(setdiff(task$feature_names, "year"))

task$feature_names

# Add new column "foo"
task$cbind(data.frame(foo = 1:344))
head(task)

## ------------------------------------------------
## Method `Task$data`
```

```
## ------------------------------------------------

task = tsk("penguins")
task$data(rows = 1:5, cols = c("species", "sex"))

## ------------------------------------------------
## Method `Task$formula`
## ------------------------------------------------

task = tsk("penguins")
task$formula()

## ------------------------------------------------
## Method `Task$head`
## ------------------------------------------------

task = tsk("penguins")
task$head(3)

## ------------------------------------------------
## Method `Task$levels`
## ------------------------------------------------

task = tsk("penguins")
task$levels()

## ------------------------------------------------
## Method `Task$missings`
## ------------------------------------------------

task = tsk("penguins")
task$missings()

## ------------------------------------------------
## Method `Task$filter`
## ------------------------------------------------

task = tsk("penguins")
task$filter(1:10)
task$nrow

## ------------------------------------------------
## Method `Task$select`
## ------------------------------------------------

task = tsk("penguins")
task$select(c("bill_length", "bill_depth"))
task$feature_names

## ------------------------------------------------
## Method `Task$rbind`
## ------------------------------------------------
```

```
task = tsk("penguins")
extra = task$data(rows = 1:2)
task$rbind(extra)

## ------------------------------------------------
## Method `Task$cbind`
## ------------------------------------------------

task = tsk("penguins")
task$cbind(data.table(extra_col = seq_len(task$nrow)))
head(task$data(cols = "extra_col"))

## ------------------------------------------------
## Method `Task$rename`
## ------------------------------------------------

task = tsk("penguins")
task$rename("body_mass", "mass")
task$feature_names

## ------------------------------------------------
## Method `Task$set_row_roles`
## ------------------------------------------------

task = tsk("penguins")
task$set_row_roles(1:5, remove_from = "use")

## ------------------------------------------------
## Method `Task$set_col_roles`
## ------------------------------------------------

task = tsk("penguins")
task$set_col_roles("sex", roles = "stratum")
task$col_roles$stratum

## ------------------------------------------------
## Method `Task$set_levels`
## ------------------------------------------------

task = tsk("penguins")
task$set_levels(list(sex = c("male", "female", "unknown")))
task$levels("sex")

## ------------------------------------------------
## Method `Task$droplevels`
## ------------------------------------------------

task = tsk("penguins")
task$set_levels(list(sex = c("male", "female", "unknown")))
task$levels("sex")

## ------------------------------------------------
## Method `Task$add_strata`
```

```
## ------------------------------------------------

task = tsk("penguins")
task$add_strata("flipper_length", bins = 4)

## ------------------------------------------------
## Method `Task$materialize_view`
## ------------------------------------------------

task = tsk("iris")
task$backend$nrow
task$filter(1:120)
task$backend$nrow
```

`TaskClassif` *Classification Task*

Description

This task specializes Task and TaskSupervised for classification problems. The target column is assumed to be a factor or ordered 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.

It is recommended to use `as_task_classif()` for construction. Predefined tasks are stored in the dictionary mlr_tasks.

Super classes

`mlr3::Task` -> `mlr3::TaskSupervised` -> `TaskClassif`

Active bindings

`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

Public methods:

- TaskClassif$new()
- TaskClassif$truth()
- TaskClassif$droplevels()
- TaskClassif$clone()

Method `new()`**:** Creates a new instance of this R6 class. The function `as_task_classif()` provides an alternative way to construct classification tasks.

Usage:

```
TaskClassif$new(
  id,
  backend,
  target,
  positive = NULL,
  label = NA_character_,
  extra_args = list()
)
```

Arguments:

`id` (`character(1)`)
: Identifier for the new instance.

`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.

`label` (`character(1)`)
: Label for the new instance.

`extra_args` (named `list()`)
: Named list of constructor arguments, required for converting task types via `convert_task()`.

Method `truth()`**:** True response for specified `row_ids`. Format depends on the task type. Defaults to all rows with role `"use"`.

Usage:

```
TaskClassif$truth(rows = NULL)
```

Arguments:

`rows` (positive `integer()`)
: Vector or row indices. Always refers to the complete data set, even after filtering.

Returns: `factor()`.

Method `droplevels()`**:** 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 "factor" or "ordered". Also updates the task property `"twoclass"`/`"multiclass"`.

Usage:

```
TaskClassif$droplevels(cols = NULL)
```

Arguments:

`cols (character())`
Vector of column names.

Returns: Modified `self`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3data** for more toy tasks.
- Package **mlr3oml** for downloading tasks from https://www.openml.org.
- Package **mlr3viz** for some generic visualizations.
- Dictionary of Tasks: mlr_tasks
- `as.data.table(mlr_tasks)` for a table of available Tasks in the running session (depending on the loaded packages).
- **mlr3fselect** and **mlr3filters** for feature selection and feature filtering.
- Extension packages for additional task types:
  - Unsupervised clustering: **mlr3cluster**
  - Probabilistic supervised regression and survival analysis: https://mlr3proba.mlr-org.com/.

Other Task: `Task`, `TaskRegr`, `TaskSupervised`, `TaskUnsupervised`, `california_housing`, `mlr_tasks`, `mlr_tasks_breast_cancer`, `mlr_tasks_german_credit`, `mlr_tasks_iris`, `mlr_tasks_mtcars`, `mlr_tasks_penguins`, `mlr_tasks_pima`, `mlr_tasks_sonar`, `mlr_tasks_spam`, `mlr_tasks_wine`, `mlr_tasks_zoo`

Examples

```
data("Sonar", package = "mlbench")
task = as_task_classif(Sonar, target = "Class", positive = "M")

task$task_type
task$formula()
task$truth()
task$class_names
task$positive
task$data(rows = 1:3, cols = task$feature_names[1:2])
```

TaskGenerator *TaskGenerator Class*

Description

Creates a Task of arbitrary size. Predefined task generators are stored in the dictionary mlr_task_generators, e.g. xor.

Public fields

`id (character(1))`
Identifier of the object. Used in tables, plot and text output.

`label (character(1))`
Label for this object. Can be used in tables, plot and text output instead of the ID.

`task_type (character(1))`
Task type, e.g. "classif" or "regr".
For a complete list of possible task types (depending on the loaded packages), see mlr_reflections$task_types$type

`param_set (paradox::ParamSet)`
Set of hyperparameters.

`packages (character(1))`
Set of required packages. These packages are loaded, but not attached.

`man (character(1))`
String in the format [pkg]::[topic] pointing to a manual page for this object. Defaults to NA, but can be set by child classes.

Methods

Public methods:

- TaskGenerator$new()
- TaskGenerator$format()
- TaskGenerator$print()
- TaskGenerator$generate()
- TaskGenerator$clone()

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
TaskGenerator$new(
  id,
  task_type,
  packages = character(),
  param_set = ps(),
  label = NA_character_,
  man = NA_character_
)
```

Arguments:

`id (character(1))`
Identifier for the new instance.

`task_type (character(1))`
Type of task, e.g. `"regr"` or `"classif"`. Must be an element of mlr_reflections$task_types$type.

`packages (character())`
Set of required packages. A warning is signaled by the constructor if at least one of the packages is not installed, but loaded (not attached) later on-demand via `requireNamespace()`.

`param_set` (paradox::ParamSet)
Set of hyperparameters.

`label (character(1))`
Label for the new instance.

`man (character(1))`
String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

Method `format()`**:** Helper for print outputs.

Usage:

```
TaskGenerator$format(...)
```

Arguments:

`...` (ignored).

Method `print()`**:** Printer.

Usage:

```
TaskGenerator$print(...)
```

Arguments:

`...` (ignored).

Method `generate()`**:** Creates a task of type `task_type` with `n` observations, possibly using additional settings stored in `param_set`.

Usage:

```
TaskGenerator$generate(n)
```

Arguments:

`n (integer(1))`
Number of rows to generate.

Returns: Task.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

- Dictionary of TaskGenerators: mlr_task_generators
- `as.data.table(mlr_task_generators)` for a table of available TaskGenerators in the running session (depending on the loaded packages).
- Extension packages for additional task types:
  - **mlr3proba** for probabilistic supervised regression and survival analysis.
  - **mlr3cluster** for unsupervised clustering.

Other TaskGenerator: `mlr_task_generators`, `mlr_task_generators_2dnormals`, `mlr_task_generators_cassini`, `mlr_task_generators_circle`, `mlr_task_generators_friedman1`, `mlr_task_generators_moons`, `mlr_task_generators_peak`, `mlr_task_generators_simplex`, `mlr_task_generators_smiley`, `mlr_task_generators_spirals`, `mlr_task_generators_xor`

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 `"regr"`.

It is recommended to use `as_task_regr()` for construction. Predefined tasks are stored in the dictionary mlr_tasks.

Super classes

`mlr3::Task` -> `mlr3::TaskSupervised` -> `TaskRegr`

Methods

Public methods:

- `TaskRegr$new()`
- `TaskRegr$truth()`
- `TaskRegr$clone()`

Method `new()`**:** Creates a new instance of this R6 class. The function `as_task_regr()` provides an alternative way to construct regression tasks.

Usage:

```
TaskRegr$new(id, backend, target, label = NA_character_, extra_args = list())
```

Arguments:

`id` (`character(1)`)
: Identifier for the new instance.

`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.

label (character(1))
: Label for the new instance.

extra_args (named list())
: Named list of constructor arguments, required for converting task types via convert_task().

Method truth(): True response for specified row_ids. Format depends on the task type. Defaults to all rows with role "use".

Usage:

```
TaskRegr$truth(rows = NULL)
```

Arguments:

rows (positive integer())
: Vector or row indices. Always refers to the complete data set, even after filtering.

Returns: numeric().

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

- Chapter in the mlr3book: https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html
- Package **mlr3data** for more toy tasks.
- Package **mlr3oml** for downloading tasks from https://www.openml.org.
- Package **mlr3viz** for some generic visualizations.
- Dictionary of Tasks: mlr_tasks
- as.data.table(mlr_tasks) for a table of available Tasks in the running session (depending on the loaded packages).
- **mlr3fselect** and **mlr3filters** for feature selection and feature filtering.
- Extension packages for additional task types:
  - Unsupervised clustering: **mlr3cluster**
  - Probabilistic supervised regression and survival analysis: https://mlr3proba.mlr-org.com/.

Other Task: Task, TaskClassif, TaskSupervised, TaskUnsupervised, california_housing, mlr_tasks, mlr_tasks_breast_cancer, mlr_tasks_german_credit, mlr_tasks_iris, mlr_tasks_mtcars, mlr_tasks_penguins, mlr_tasks_pima, mlr_tasks_sonar, mlr_tasks_spam, mlr_tasks_wine, mlr_tasks_zoo

Examples

```
task = as_task_regr(palmerpenguins::penguins, target = "bill_length_mm")
task$task_type
task$formula()
task$truth()
task$data(rows = 1:3, cols = task$feature_names[1:2])
```

---

uhashes *Obtain specific uhashes from a BenchmarkResult*

Description

In a `BenchmarkResult`, each ResampleResult is *u*niquely identified by a *hash* (*uhash*). Operations that select specific ResampleResults from a BenchmarkResult operate using these hashes. This function allows to obtain uhashes for specific learners, tasks, and resamplings.

If you want more control, you can also directly obtain the uhash table from the `BenchmarkResult` via the field `$uhash_table`.

Usage

```
uhashes(bmr, learner_ids = NULL, task_ids = NULL, resampling_ids = NULL)

uhash(bmr, learner_id = NULL, task_id = NULL, resampling_id = NULL)
```

Arguments

`bmr`	(`BenchmarkResult`) Benchmark result.
`learner_ids`	(`character()` | `NULL`) Learner IDs.
`task_ids`	(`character()` | `NULL`) Task IDs.
`resampling_ids`	(`character()` | `NULL`) Resampling IDs.
`learner_id`	(`character(1)` | `NULL`) Learner ID.
`task_id`	(`character(1)` | `NULL`) Task ID.
`resampling_id`	(`character(1)` | `NULL`) Resampling ID.

Examples

```
design = benchmark_grid(
  tsks(c("sonar", "iris")),
  lrns(c("classif.debug", "classif.featureless", "classif.rpart")),
  rsmp("holdout")
)
bmr = benchmark(design)
bmr
bmr$uhashes
uhash(bmr, learner_id = "classif.debug", task_id = "sonar", resampling_id = "holdout")
uhashes(bmr, learner_ids = c("classif.debug", "classif.featureless"))
```

Index