

# Package ‘mlr3’

October 18, 2024

**Title** Machine Learning in R - Next Generation

**Version** 0.21.1

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

**License** LGPL-3

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

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

**Depends** R (>= 3.1.0)

**Imports** R6 (>= 2.4.1), backports, checkmate (>= 2.0.0), data.table (>= 1.15.0), evaluate, future, future.apply (>= 1.5.0), lgr (>= 0.3.4), mlbench, mlr3measures (>= 1.0.0), mlr3misc (>= 0.15.0), parallelly, palmerpenguins, paradox (>= 1.0.1), uuid

**Suggests** Matrix, callr, codetools, datasets, future.callr, mlr3data, progressr, remotes, RhpcBLASctl, rpart, testthat (>= 3.1.0)

**Encoding** UTF-8

**Config/testthat/edition** 3

**Config/testthat/parallel** false

**NeedsCompilation** no

**RoxygenNote** 7.3.2

**Collate** 'mlr\_reflections.R' 'BenchmarkResult.R' 'warn\_deprecated.R'  
'DataBackend.R' 'DataBackendCbind.R' 'DataBackendDataTable.R'  
'DataBackendMatrix.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' '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' 'TaskGeneratorSimplex.R'  
 'TaskGeneratorSmiley.R' 'TaskGeneratorSpirals.R'  
 'TaskGeneratorXor.R' 'TaskRegr.R' 'TaskRegr\_boston\_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' 'helper\_print.R' 'install\_pkgs.R' 'marshal.R'  
 'mlr\_sugar.R' 'mlr\_test\_helpers.R' 'partition.R' 'predict.R'  
 'reexports.R' 'resample.R' 'set\_threads.R' 'set\_validate.R'  
 'task\_converters.R' 'worker.R' 'zzz.R'

**Author** Michel Lang [aut] (<<https://orcid.org/0000-0001-9754-0393>>),  
 Bernd Bischl [aut] (<<https://orcid.org/0000-0001-6002-6980>>),  
 Jakob Richter [aut] (<<https://orcid.org/0000-0003-4481-5554>>),  
 Patrick Schratz [aut] (<<https://orcid.org/0000-0003-0748-6624>>),  
 Giuseppe Casalicchio [ctb] (<<https://orcid.org/0000-0001-5324-5966>>),  
 Stefan Coors [ctb] (<<https://orcid.org/0000-0002-7465-2146>>),  
 Quay Au [ctb] (<<https://orcid.org/0000-0002-5252-8902>>),  
 Martin Binder [aut],  
 Florian Pfisterer [aut] (<<https://orcid.org/0000-0001-8867-762X>>),  
 Raphael Sonabend [aut] (<<https://orcid.org/0000-0001-9225-4654>>),

Lennart Schneider [ctb] (<<https://orcid.org/0000-0003-4152-5308>>),  
 Marc Becker [cre, aut] (<<https://orcid.org/0000-0002-8115-0400>>),  
 Sebastian Fischer [aut] (<<https://orcid.org/0000-0002-9609-3197>>),  
 Lona Koers [ctb]

**Maintainer** Marc Becker <marcbecker@posteo.de>

**Repository** CRAN

**Date/Publication** 2024-10-18 10:40:14 UTC

## Contents

mlr3-package . . . . .	6
as_benchmark_result . . . . .	9
as_data_backend.Matrix . . . . .	9
as_learner . . . . .	11
as_measure . . . . .	11
as_prediction . . . . .	12
as_prediction_classif . . . . .	13
as_prediction_data . . . . .	14
as_prediction_regr . . . . .	15
as_resample_result . . . . .	16
as_resampling . . . . .	17
as_result_data . . . . .	18
as_task . . . . .	19
as_task_classif . . . . .	20
as_task_regr . . . . .	23
as_task_unsupervised . . . . .	25
benchmark . . . . .	26
BenchmarkResult . . . . .	30
benchmark_grid . . . . .	37
convert_task . . . . .	39
DataBackend . . . . .	40
DataBackendDataTable . . . . .	42
DataBackendMatrix . . . . .	45
default_fallback . . . . .	47
default_measures . . . . .	48
HotstartStack . . . . .	49
install_pkgs . . . . .	51
Learner . . . . .	52
LearnerClassif . . . . .	61
LearnerRegr . . . . .	64
marshaling . . . . .	67
Measure . . . . .	68
MeasureClassif . . . . .	74
MeasureRegr . . . . .	76
MeasureSimilarity . . . . .	79
mlr_learners . . . . .	82
mlr_learners_classif.debug . . . . .	83

mlr_learners_classif.featureless . . . . .	86
mlr_learners_classif.rpart . . . . .	89
mlr_learners_regr.debug . . . . .	91
mlr_learners_regr.featureless . . . . .	93
mlr_learners_regr.rpart . . . . .	95
mlr_measures . . . . .	98
mlr_measures_aic . . . . .	99
mlr_measures_bic . . . . .	100
mlr_measures_classif.acc . . . . .	102
mlr_measures_classif.auc . . . . .	103
mlr_measures_classif.bacc . . . . .	105
mlr_measures_classif.bbrier . . . . .	106
mlr_measures_classif.ce . . . . .	107
mlr_measures_classif.costs . . . . .	109
mlr_measures_classif.dor . . . . .	111
mlr_measures_classif.fbeta . . . . .	112
mlr_measures_classif.fdr . . . . .	114
mlr_measures_classif.fn . . . . .	115
mlr_measures_classif.fnr . . . . .	117
mlr_measures_classif.fomr . . . . .	118
mlr_measures_classif.fp . . . . .	119
mlr_measures_classif.fpr . . . . .	121
mlr_measures_classif.logloss . . . . .	122
mlr_measures_classif.mauc_aulp . . . . .	123
mlr_measures_classif.mauc_aulu . . . . .	125
mlr_measures_classif.mauc_aunp . . . . .	127
mlr_measures_classif.mauc_aunu . . . . .	128
mlr_measures_classif.mauc_mu . . . . .	130
mlr_measures_classif.mbrier . . . . .	132
mlr_measures_classif.mcc . . . . .	133
mlr_measures_classif.npv . . . . .	135
mlr_measures_classif.ppv . . . . .	136
mlr_measures_classif.prauc . . . . .	138
mlr_measures_classif.precision . . . . .	139
mlr_measures_classif.recall . . . . .	140
mlr_measures_classif.sensitivity . . . . .	142
mlr_measures_classif.specificity . . . . .	143
mlr_measures_classif.tn . . . . .	144
mlr_measures_classif.tnr . . . . .	146
mlr_measures_classif.tp . . . . .	147
mlr_measures_classif.tpr . . . . .	148
mlr_measures_debug_classif . . . . .	150
mlr_measures_elapsed_time . . . . .	152
mlr_measures_internal_valid_score . . . . .	154
mlr_measures_oob_error . . . . .	155
mlr_measures_regr.bias . . . . .	157
mlr_measures_regr.ktau . . . . .	158
mlr_measures_regr.mae . . . . .	159

mlr_measures_regr.mape	160
mlr_measures_regr.maxae	161
mlr_measures_regr.medae	162
mlr_measures_regr.medse	163
mlr_measures_regr.mse	164
mlr_measures_regr.msle	166
mlr_measures_regr.pbias	167
mlr_measures_regr.pinball	168
mlr_measures_regr.rae	169
mlr_measures_regr.rmse	170
mlr_measures_regr.rmsle	171
mlr_measures_regr.rrse	173
mlr_measures_regr.rse	174
mlr_measures_regr.rsq	175
mlr_measures_regr.sae	177
mlr_measures_regr.smape	178
mlr_measures_regr.srho	179
mlr_measures_regr.sse	180
mlr_measures_selected_features	181
mlr_measures_sim.jaccard	183
mlr_measures_sim.phi	184
mlr_resamplings	185
mlr_resamplings_bootstrap	186
mlr_resamplings_custom	188
mlr_resamplings_custom_cv	189
mlr_resamplings_cv	192
mlr_resamplings_holdout	193
mlr_resamplings_insample	195
mlr_resamplings_loo	197
mlr_resamplings_repeated_cv	199
mlr_resamplings_subsampling	201
mlr_sugar	203
mlr_tasks	205
mlr_tasks_boston_housing	206
mlr_tasks_breast_cancer	207
mlr_tasks_german_credit	208
mlr_tasks_iris	210
mlr_tasks_mtcars	211
mlr_tasks_penguins	212
mlr_tasks_pima	214
mlr_tasks_sonar	215
mlr_tasks_spam	216
mlr_tasks_wine	218
mlr_tasks_zoo	219
mlr_task_generators	220
mlr_task_generators_2dnormals	221
mlr_task_generators_cassini	223
mlr_task_generators_circle	225

mlr_task_generators_friedman1 . . . . .	227
mlr_task_generators_moons . . . . .	228
mlr_task_generators_simplex . . . . .	230
mlr_task_generators_smiley . . . . .	231
mlr_task_generators_spirals . . . . .	233
mlr_task_generators_xor . . . . .	235
mlr_test_helpers . . . . .	237
partition . . . . .	239
predict.Learner . . . . .	239
Prediction . . . . .	240
PredictionClassif . . . . .	243
PredictionData . . . . .	246
PredictionRegr . . . . .	248
resample . . . . .	250
ResampleResult . . . . .	253
Resampling . . . . .	258
set_threads . . . . .	262
Task . . . . .	263
TaskClassif . . . . .	274
TaskGenerator . . . . .	277
TaskRegr . . . . .	280

<b>Index</b>	<b>282</b>
--------------	------------

---

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.allow_utf8_names"`: If set to TRUE, checks on the feature names are relaxed, allowing non-ascii characters in column names. This is an experimental and temporal option to pave the way for text analysis, and will likely be removed in a future version of the package. analysis.
- `"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.

**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]

**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>



---

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.Matrix    *Create a Data Backend*

---

### Description

Wraps a [DataBackend](#) around data. **mlr3** ships with methods for `data.frame` (converted to a [DataBackendDataTable](#) and `Matrix` from package **Matrix** (converted to a [DataBackendMatrix](#)).

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**

```
## S3 method for class 'Matrix'
as_data_backend(data, primary_key = NULL, dense = NULL, ...)

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	( <a href="#">data.frame()</a> ) The input <a href="#">data.frame()</a> . Automatically converted to a <a href="#">data.table::data.table()</a> .
primary_key	( <a href="#">character(1)</a>   <a href="#">integer()</a> ) Name of the primary key column, or integer vector of row ids.
dense	( <a href="#">data.frame()</a> ). Dense data.
...	( <a href="#">any</a> ) Additional arguments passed to the respective <a href="#">DataBackend</a> method.
keep_rownames	( <a href="#">logical(1)</a>   <a href="#">character(1)</a> ) 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 <code>keep_rownames == TRUE</code> . Note that the created column will be used as a regular feature by the task unless you manually change the column role. Also see <a href="#">data.table::as.data.table()</a> .

**Value**

[DataBackend](#).

**See Also**

- Chapter in the [mlr3book](https://mlr3book.mlr-org.com/chapters/chapter10/advanced_technical_aspects_of_mlr3.html#sec-backends): [https://mlr3book.mlr-org.com/chapters/chapter10/advanced\\_technical\\_aspects\\_of\\_mlr3.html#sec-backends](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](#), [DataBackendMatrix](#)

**Examples**

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

---

as_learner	<i>Convert to a Learner</i>
------------	-----------------------------

---

**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	<i>Convert to a Measure</i>
------------	-----------------------------

---

**Description**

Convert object to a [Measure](#) or a list of [Measure](#).

**Usage**

```

as_measure(x, ...)

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

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

as_measures(x, ...)

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

## S3 method for class '`NULL`'
as_measures(x, task_type = NULL, ...)

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

```

**Arguments**

x	(any) Object to convert.
...	(any) Additional arguments.
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 <code>mlr_reflections\$default_measures</code> .
clone	(logical(1)) If TRUE, ensures that the returned object is not the same as the input x.

**Value**

[Measure](#).

---

as_prediction	<i>Convert to a Prediction</i>
---------------	--------------------------------

---

**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 <a href="#">ResampleResult</a> . Multiple predict sets are calculated by the respective <a href="#">Learner</a> during <a href="#">resample()/benchmark()</a> . 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	( <a href="#">Task</a> ).
row_ids	integer() Row indices.
check	(logical(1)) Perform argument checks and type conversions?
...	(any) Additional arguments.
train_task	( <a href="#">Task</a> ) Task used for training the learner.

**Value**

[PredictionData](#).

---

as_prediction_regr	<i>Convert to a Regression Prediction</i>
--------------------	---

---

**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())
```



**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**

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

**Value**

([ResampleResult](#)).

---

<code>as_resampling</code>	<i>Convert to a Resampling</i>
----------------------------	--------------------------------

---

**Description**

Convert object to a [Resampling](#) or a list of [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	<i>Convert to ResultData</i>
----------------	------------------------------

---

**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,
  store_backends = TRUE
)

```

**Arguments**

task	( <a href="#">Task</a> ).
learners	(list of trained <a href="#">Learners</a> ).
resampling	( <a href="#">Resampling</a> ).
iterations	(integer()).
predictions	(list of list of <a href="#">Predictions</a> ).
learner_states	(list()) Learner states. If not provided, the states of learners are automatically extracted.
store_backends	(logical(1)) If set to FALSE, the backends of the <a href="#">Tasks</a> 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 = rsmpl("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](#).

**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	<i>Convert to a Classification Task</i>
-----------------	---

---

**Description**

Convert object to a [TaskClassif](#). This is a S3 generic. mlr3 ships with methods for the following objects:

1. [TaskClassif](#): ensure the identity
2. [formula](#), [data.frame\(\)](#), [matrix\(\)](#), [Matrix::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 = NULL,  
  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 'Matrix'  
as_task_classif(  
  x,  
  target,  
  id = deparse1(substitute(x)),  
  label = NA_character_,  
  ...  
)  
  
## S3 method for class 'DataBackend'  
as_task_classif(  
  x,  
  target = NULL,  
  id = deparse1(substitute(x)),  
  positive = NULL,  
  label = NA_character_,  
  ...  
)  
  
## S3 method for class 'TaskRegr'  
as_task_classif(  
  x,  
  target = NULL,  
  drop_original_target = FALSE,  
  drop_levels = TRUE,  
  ...  
)  
  
## S3 method for class 'formula'  
as_task_classif(  
  x,  
  target = NULL,  
  id = deparse1(substitute(x)),  
  positive = NULL,  
  label = NA_character_,  
  ...  
)
```

```

  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 <a href="#">TaskClassif</a> .
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](#): ensure the identity
2. [formula](#), [data.frame\(\)](#), [matrix\(\)](#), [Matrix::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 = NULL,  
  id = deparse1(substitute(x)),  
  label = NA_character_,  
  ...  
)  
  
## S3 method for class 'matrix'  
as_task_regr(  
  x,  
  target = NULL,  
  id = deparse1(substitute(x)),  
  label = NA_character_,  
  ...  
)  
  
## S3 method for class 'Matrix'  
as_task_regr(  
  x,  
  target = NULL,  
  id = deparse1(substitute(x)),  
  label = NA_character_,  
  ...  
)
```

```

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

## S3 method for class 'TaskClassif'
as_task_regr(
  x,
  target = NULL,
  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
)
```

**Arguments**

design	( <a href="#">data.frame()</a> ) Data frame (or <a href="#">data.table::data.table()</a> ) with three columns: "task", "learner", and "resampling". Each row defines a resampling by providing a <a href="#">Task</a> , <a href="#">Learner</a> and an instantiated <a href="#">Resampling</a> strategy. The helper function <a href="#">benchmark_grid()</a> can assist in generating an exhaustive design (see examples) and instantiate the <a href="#">Resamplings</a> per <a href="#">Task</a> . Additionally, you can set the additional column 'param_values', see <a href="#">benchmark_grid()</a> .
store_models	( <a href="#">logical(1)</a> ) 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	( <a href="#">logical(1)</a> ) Keep the <a href="#">DataBackend</a> of the <a href="#">Task</a> in the <a href="#">ResampleResult</a> ? Set to TRUE if your performance measures require a <a href="#">Task</a> , 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	( <a href="#">character(1)</a> ) If not NA, enables encapsulation by setting the field <code>Learner\$encapsulate</code> 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 <b>evaluate</b> ) and "callr" (start in external session via <b>callr</b> ). 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	( <a href="#">logical(1)</a> ) Determines if learner(s) are hot started with trained models in <code>\$hotstart_stack</code> . See also <a href="#">HotstartStack</a> .
clone	( <a href="#">character()</a> ) Select the input objects to be cloned before proceeding by providing a set with possible values "task", "learner" and "resampling" for <a href="#">Task</a> , <a href="#">Learner</a> and <a href="#">Resampling</a> , respectively. Per default, all input objects are cloned.
unmarshal	<a href="#">Learner</a> 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.

**Value**

[BenchmarkResult](#).

**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** package. One job is one resampling iteration, and all jobs are send to an apply function from **future.apply** in a single batch. To select a parallel backend, use `future::plan()`. More on parallelization can be found in the book: [https://mlr3book.mlr-org.com/chapters/chapter10/advanced\\_technical\\_aspects\\_of\\_mlr3.html](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](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 = rsmpl("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(rsmpl("cv", folds = 3), rsmpl("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")`  
`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](#).

## 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\\$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()`.

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

*Usage:*

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

*Arguments:*

... (any)

Additional arguments passed to `unmarshal_model()`.

**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()`.

**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.

**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\(\)](#).

**Method** `filter()`: Subsets the benchmark result. If `task_ids` is not `NULL`, keeps all tasks with provided task ids and discards all others tasks. Same procedure for `learner_ids` and `resampling_ids`.

*Usage:*

```
BenchmarkResult$filter(
  task_ids = NULL,
  task_hashes = NULL,
  learner_ids = NULL,
  learner_hashes = NULL,
  resampling_ids = NULL,
  resampling_hashes = NULL
)
```

*Arguments:*

task\_ids (character())

Ids of [Tasks](#) to keep.

task\_hashes (character())

Hashes of [Tasks](#) to keep.

learner\_ids (character())

Ids of [Learners](#) to keep.

learner\_hashes (character())

Hashes of [Learners](#) to keep.

resampling\_ids (character())

Ids of [Resamplings](#) to keep.

resampling\_hashes (character())

Hashes of [Resamplings](#) to keep.

*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** `resample_result()`: Retrieve the *i*-th [ResampleResult](#), by position or by unique hash *uhash*. *i* and *uhash* are mutually exclusive.

*Usage:*

```
BenchmarkResult$resample_result(i = NULL, uhash = NULL)
```

*Arguments:*

*i* (integer(1))

The iteration value to filter for.

*uhash* (logical(1))

The *uhash* value to filter for.

*Returns:* [ResampleResult](#).

**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 keep the object in its previous state.

**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](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 = rsmpl("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)
```

---

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 number of rows, 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. I.e., 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 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.

**See Also**

- Chapter in the `mlr3book`: [https://mlr3book.mlr-org.com/chapters/chapter3/evaluation\\_and\\_benchmarking.html#sec-benchmarking](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(rsmpl("cv"), rsmpl("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 = rsmpl("holdout")
res2 = rsmpl("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)
```

---

convert_task	<i>Convert a Task from One Type to Another</i>
--------------	--

---

### 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	( <a href="#">Task</a> ) A <a href="#">Task</a> 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 <code>mlr_reflections\$task_types</code> . 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 comes with two implementations for backends:

- [DataBackendDataTable](#) which stores the data as `data.table::data.table()`.
- [DataBackendMatrix](#) which stores the data as sparse `Matrix::sparseMatrix()`.

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](#) or [DataBackendMatrix](#) for exemplary implementations of the interface).

## Public fields

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

## Active bindings

`data_formats` (character())  
Supported data format. Always "data.table".. This is deprecated and will be removed in the future.

`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 [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`.



## 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 [DataBackendMatrix](#), or via the S3 method [as\\_data\\_backend\(\)](#).

*Usage:*

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

*Arguments:*

`data` (any)

The format of the input data depends on the specialization. E.g., [DataBackendDataTable](#) expects a `data.table::data.table()` and [DataBackendMatrix](#) expects a `Matrix::Matrix()` from **Matrix**.

`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.

`data_formats` (character())

Deprecated: ignored, and will be removed in the future.

**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): [https://mlr3book.mlr-org.com/chapters/chapter10/advanced\\_technical\\_aspects\\_of\\_mlr3.html#sec-backends](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](#), [DataBackendMatrix](#), [as\\_data\\_backend.Matrix\(\)](#)

**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 in the specified format. Currently, the only supported formats are "data.table" and "Matrix". The rows must be addressed as vector of primary key values, columns must be referred to via column names. Queries for rows with no matching row id and queries for columns with no matching column name are silently ignored. Rows are guaranteed to be returned in the same order as rows, columns may be returned in an arbitrary order. Duplicated row ids result in duplicated rows, duplicated column names lead to an exception.

*Usage:*

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

*Arguments:*

`rows` (`positive integer()`)

Vector or row indices. Always refers to the complete data set, even after filtering.

`cols` (`character()`)

Vector of column names.

`data_format` (`character(1)`)

Deprecated. Ignored, and will be removed in the future.

**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): [https://mlr3book.mlr-org.com/chapters/chapter10/advanced\\_technical\\_aspects\\_of\\_mlr3.html#sec-backends](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](#), [DataBackendMatrix](#), [as\\_data\\_backend.Matrix\(\)](#)

### 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$head()
b$head()
b$data(rows = 100:101, cols = "species")

b$nrow
head(b$rownames)

b$ncol
b$colnames

# alternative construction
as_data_backend(palmerpenguins::penguins)
```

---

DataBackendMatrix      *DataBackend for Matrix*

---

## Description

[DataBackend](#) for **Matrix**. Data is split into a (numerical) sparse part and an optional dense part. These parts are automatically merged to a sparse format during `$data()`. Note that merging both parts potentially comes with a data loss, as all dense columns are converted to numeric columns.

## Super class

`m1r3::DataBackend` -> DataBackendMatrix

## 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:

- `DataBackendMatrix$new()`
- `DataBackendMatrix$data()`
- `DataBackendMatrix$head()`
- `DataBackendMatrix$distinct()`
- `DataBackendMatrix$missings()`

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

*Usage:*

```
DataBackendMatrix$new(data, dense, primary_key = NULL)
```

*Arguments:*

`data` `Matrix::Matrix()`

The input `Matrix::Matrix()`.

`dense` `data.frame()`. Dense data, converted to `data.table::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 as "data.table". 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:*

```
DataBackendMatrix$data(rows, cols, data_format)
```

*Arguments:*

`rows` (positive integer())

Vector or row indices. Always refers to the complete data set, even after filtering.

`cols` (character())

Vector of column names.

`data_format` (character(1))

Deprecated. Ignored, and will be removed in the future.

**Method** `head()`: Retrieve the first n rows.

*Usage:*

```
DataBackendMatrix$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:*

```
DataBackendMatrix$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:*

```
DataBackendMatrix$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): [https://mlr3book.mlr-org.com/chapters/chapter10/advanced\\_technical\\_aspects\\_of\\_mlr3.html#sec-backends](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](#), [as\\_data\\_backend.Matrix\(\)](#)

### Examples

```
requireNamespace("Matrix")
data = Matrix::Matrix(sample(0:1, 20, replace = TRUE), ncol = 2)
colnames(data) = c("x1", "x2")
dense = data.frame(
  ..row_id = 1:10,
  num = runif(10),
  fact = factor(sample(c("a", "b"), 10, replace = TRUE), levels = c("a", "b"))
)

b = as_data_backend(data, dense = dense, primary_key = "..row_id")
b$head()
b$data(1:3, b$colnames)
```

---

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	<a href="#">Learner</a> The learner for which a fallback learner should be created.
...	any ignored.

**Value**

[Learner](#)

---

default_measures	<i>Get the Default Measure</i>
------------------	--------------------------------

---

**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 <code>task_type</code> , e.g., <code>"classif"</code> or <code>"regr"</code> . If <code>task_type</code> is NULL, an empty list is returned.
-----------	---

**Value**

list of [Measure](#).

**Examples**

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



## 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".
- `loglik(...)`: Extracts the log-likelihood (c.f. `stats::logLik()`). This can be used in measures like `mlr_measures_aic` or `mlr_measures_bic`.
- `internal_valid_scores`: Returns the internal validation score(s) of the model as a named `list()`. Only available for **Learners** 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 **Learners** with the "internal\_tuning" property. If the learner is not trained yet, this returns `NULL`.

## Setting Hyperparameters

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

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

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

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

If the learner has additional hyperparameters which are not encoded in the `ParamSet`, you can easily extend the learner. Here, we add a 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  $1 - \text{ratio}$  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 hyperparameter tuning with the standard hyperparameter optimization implemented via `mlr3tuning`, one must:

- 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 [Learners](#) 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.

## 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`.

`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`.

`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.ml-org.com/chapters/chapter10/advanced\\_technical\\_aspects\\_of\\_ml3.html#sec-error-handling](https://mlr3book.ml-org.com/chapters/chapter10/advanced_technical_aspects_of_ml3.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

`data_formats` (character())

Supported data format. Always "data.table".. This is deprecated and will be removed in the future.

`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.

`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`.

`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`.

## 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\\$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(),
  data_formats,
  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.
- "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.

`data_formats` (character())  
 Deprecated: ignored, and will be removed in the future.

`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 keep the object in its previous state.

**Method** `predict()`: Uses the information stored during `$train()` in `$state` to create a new [Prediction](#) for a set of observations of the provided task.

*Usage:*

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

*Arguments:*

task ([Task](#)).

row\_ids ([integer\(\)](#))

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

*Returns:* [Prediction](#).

**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.

*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 the sequence `1:nrow(newdata)`.

task ([Task](#)).

Returns: [Prediction](#).

**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 keep the object in its previous state.

**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 seg-faults.

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 completely to predict predictions sets. 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.

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): [https://mlr3book.mlr-org.com/chapters/chapter10/advanced\\_technical\\_aspects\\_of\\_mlr3.html#sec-error-handling](https://mlr3book.mlr-org.com/chapters/chapter10/advanced_technical_aspects_of_mlr3.html#sec-error-handling)

Usage:

```
Learner$encapsulate(method, fallback = 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.

Returns: `self` (invisibly).

**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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-learners](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](#)

---

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$new()`
- `LearnerClassif$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(),
  data_formats,
  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.
- "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.

`data_formats` (character())

Deprecated: ignored, and will be removed in the future.

`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:*

```
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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-learners](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.featureless](#), [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>).

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\\$clone\(\)](#)

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

*Usage:*

```
LearnerRegr$new(
  id,
  param_set = ps(),
  predict_types = "response",
  feature_types = character(),
  properties = character(),
  data_formats,
  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.
- "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.

`data_formats` (character())  
 Deprecated: ignored, and will be removed in the future.

`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:*

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

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

- Chapter in the `mlr3book`: [https://mlr3book.ml-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-learners](https://mlr3book.ml-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.featureless`, `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")
```

## Description

Marshaling is the process of processing the model of a trained [Learner](#) so it can 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 therefore 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-serialized 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 available. 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 guaranteed 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 in-place 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

<code>.learner</code>	<a href="#">Learner</a> The learner.
<code>...</code>	(any) Additional parameters, currently unused.
<code>model</code>	(any) Model to marshal.
<code>inplace</code>	(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_marshallable_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 `$score()` which quantifies the performance by comparing the truth and predictions.
2. A function `$aggregator()` which combines multiple performance scores returned by `$score()` to a single numeric value.

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

Predefined measures are stored in the 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 `mlr3infern`) 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.

**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.

`properties` (`character()`)

Properties of this measure.

`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.

`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.
- "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.

**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.

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](#)), and
- "na\_score" (the measure is expected to occasionally return NA or NaN).
- "primary\_iters" (the measure explicitly 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 is 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)`.

**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)`.

**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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-eval](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.rsq](#), [mlr\\_measures\\_selected\\_features](#)

---

MeasureClassif	<i>Classification Measure</i>
----------------	-------------------------------

---

## 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 `$aggregator` to average them to a single number.  
If set to "micro", the individual [Prediction](#) objects are first combined into a single new [Prediction](#) object which is then used to assess the performance. The function in `$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$measure_properties`. Supported by mlr3:
- "requires\_task" (requires the complete [Task](#)),
  - "requires\_learner" (requires the trained [Learner](#)),
  - "requires\_model" (requires the trained [Learner](#), including the fitted model),
  - "requires\_train\_set" (requires the training indices from the [Resampling](#)), and
  - "na\_score" (the measure is expected to occasionally return NA or NaN).
  - "primary\_iters" (the measure explicitly handles resamplings that only use a subset of their iterations for the point estimate).
  - "requires\_no\_prediction" (No prediction is required; This usually means that the measure extracts some information from the learner state.)
- `predict_type` (character(1))  
Required predict type of the [Learner](#). Possible values are stored in `mlr_reflections$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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-eval](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.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.

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](#)), and
- "na\_score" (the measure is expected to occasionally return NA or NaN).
- "primary\_iters" (the measure explicitly 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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-eval](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.rsq](#), [mlr\\_measures\\_selected\\_features](#)

---

MeasureSimilarity	<i>Similarity Measure</i>
-------------------	---------------------------

---

## 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_,
  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.

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](#)), and
- "na\_score" (the measure is expected to occasionally return NA or NaN).
- "primary\_iters" (the measure explicitly 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:*

```
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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-eval](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.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 = rsmpl("cv", folds = 3)
```

```
grid = benchmark_grid(task, learners, resampling)
bmr = benchmark(grid, store_models = TRUE)
bmr$aggregate(msrs(c("classif.ce", "sim.jaccard")))
```

---

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.featureless`, `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	-
sefault_predict	numeric	0		[0, 1]
sefault_train	numeric	0		[0, 1]
sleep_train	untyped	-		-
sleep_predict	untyped	-		-
threads	integer	-		[1, ∞)
warning_predict	numeric	0		[0, 1]
warning_train	numeric	0		[0, 1]
x	numeric	-		[0, 1]
iter	integer	1		[1, ∞)
early_stopping	logical	FALSE	TRUE, FALSE	-
count_marshaling	logical	FALSE	TRUE, FALSE	-
check_pid	logical	TRUE	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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-learners](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$param_set$values = list(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.

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

**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.

### 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”
- 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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-learners](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
<code>cp</code>	numeric	0.01		[0, 1]
<code>keep_model</code>	logical	FALSE	TRUE, FALSE	-
<code>maxcompete</code>	integer	4		[0, ∞)
<code>maxdepth</code>	integer	30		[1, 30]
<code>maxsurrogate</code>	integer	5		[0, ∞)
<code>minbucket</code>	integer	-		[1, ∞)
<code>minsplit</code>	integer	20		[1, ∞)
<code>surrogatestyle</code>	integer	0		[0, 1]
<code>usesurrogate</code>	integer	2		[0, 2]
<code>xval</code>	integer	10		[0, ∞)

**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.ml-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-learners](https://mlr3book.ml-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**

**Parameters**

Id	Type	Default	Levels	Range
predict_missing	numeric	0		[0, 1]
predict_missing_type	character	na	na, omit	-
save_tasks	logical	FALSE	TRUE, FALSE	-
threads	integer	-		[1, $\infty$ )
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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-learners](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.

**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”
- 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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-learners](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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-learners](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.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](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.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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-eval](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.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

$$\frac{1}{n} \sum_{i=1}^n w_i \mathbf{1}(t_i = r_i),$$

where  $w_i$  are normalized weights 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.fb`, `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.p`, `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.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.npv`, `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.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.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`, `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.prc`, `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.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.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.p`, `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

$$\frac{1}{n} \sum_{i=1}^n w_i \mathbf{1}(t_i \neq r_i),$$

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.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.pra`, `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_aup`, `mlr_measures_classif.mauc_aulu`, `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
<code>normalize</code>	logical	-	<code>TRUE</code> , <code>FALSE</code>

## 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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-eval](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.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.fl](#), [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.p](#), [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, rsmpl("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

$$\frac{TP/FP}{FN/TN}$$

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.log`, `mlr_measures_classif.mauc_aup`, `mlr_measures_classif.mauc_aup1p`, `mlr_measures_classif.mauc_aup1u`, `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.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  $\beta$  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.log, mlr\_measures\_classif.mauc\_aup, mlr\_measures\_classif.mauc\_aulu, 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.fdr, mlr\_measures\_classif.fn, mlr\_measures\_classif.fnr, mlr\_measures\_classif.fomr, mlr\_measures\_classif.fp, mlr\_measures\_classif.fpr, mlr\_measures\_classif.log, 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

$$\frac{FP}{TP + FP}$$

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.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.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.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.fpr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.log`, `mlr_measures_classif.mauc_aup`, `mlr_measures_classif.mauc_aul`, `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.fpr`, `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

$$\frac{FN}{TP + FN}$$

Also known 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.fr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `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.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.fr`, `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

$$\frac{FN}{FN + TN}$$

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.fpr`, `mlr_measures_classif.fnr`, `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.fpr`, `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.fpr`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `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.fpr`, `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

$$\frac{FP}{FP + TN}$$

Also known 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.fpr`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_aup`, `mlr_measures_classif.mauc_aupl`, `mlr_measures_classif.mauc_aulu`, `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.fpr`, `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

$$-\frac{1}{n} \sum_{i=1}^n w_i \log(p_i)$$

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.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.mauc_aup`, `mlr_measures_classif.mauc_aulu`, `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.pra`, `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_aup`, `mlr_measures_classif.mauc_aulu`, `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_aup`

*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).
- *AUIU*: 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.
- *AUIP*: 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_aup")
msr("classif.mauc_aup")
```

## Parameters

Empty ParamSet

## Meta Information

- Type: "classif"
- Range: [0, 1]
- Minimize: FALSE
- Required prediction: prob

## Note

The score function calls `mlr3measures::mauc_aup()` 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.fpr`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_aulu`, `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.pra`, `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_aulu`, `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_aulu`

*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).
- *AUIU*: 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.
- *AUIP*: 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_aulu")
msr("classif.mauc_aulu")
```

### Parameters

Empty ParamSet

### Meta Information

- Type: "classif"
- Range: [0, 1]
- Minimize: FALSE
- Required prediction: prob

### Note

The score function calls `mlr3measures::mauc_aulu()` 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.fpr`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_aulu`, `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.pra`, `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_aulu`, `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).
- *AUIU*: 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.
- *AUIP*: 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.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_aulu`, `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.pra`, `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_aulu`, `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).
- *AUIU*: 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.
- *AUIP*: 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.fpr`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_aup`, `mlr_measures_classif.mauc_aul`, `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.pra`, `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_aup`, `mlr_measures_classif.mauc_aul`, `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).
- *AUIU*: 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.
- *AUIP*: 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.fpr`, `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_aulu`, `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.pra`, `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_aulu`, `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.fpr`, `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.pra`, `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

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

where  $TP$ ,  $FP$ ,  $TN$ ,  $FN$  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:

$$\frac{c \cdot s - \sum_k p_k \cdot t_k}{\sqrt{(s^2 - \sum_k p_k^2) \cdot (s^2 - \sum_k t_k^2)}}$$

where

- $s = \sum_i \sum_j C_{ij}$ : total number of samples
- $c = \sum_k C_{kk}$ : total number of correctly predicted samples

- $t_k = \sum_i C_{ik}$ : number of predictions for each class  $k$
- $p_k = \sum_j C_{kj}$ : 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  $s^2 - \sum_k p_k^2$  or  $s^2 - \sum_k t_k^2$  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.fpr`, `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

$$\frac{TN}{FN + TN}$$

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.fpr`, `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_aulu`, `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.fpr`, `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

$$\frac{TP}{TP + FP}$$

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.fpr`, `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.fpr`, `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.fpr`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_aup`, `mlr_measures_classif.mauc_aul`, `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.fpr`, `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

$$\frac{TP}{TP + FP}$$

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.fpr`, `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.fpr`, `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

$$\frac{TP}{TP + FN}$$

This is also known 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.fpr`, `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.fpr`, `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.ppr`, `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

$$\frac{TP}{TP + FN}$$

This is also known 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.fpr`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_aup`, `mlr_measures_classif.mauc_aulu`, `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.fpr`, `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.p`, `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

$$\frac{TN}{FP + TN}$$

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.fpr`, `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.fpr`, `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.p`, `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.fpr`, `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_aulu`, `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.fpr`, `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.p  
 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

$$\frac{TN}{FP + TN}$$

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.fpr`, `mlr_measures_classif.fnr`, `mlr_measures_classif.fomr`, `mlr_measures_classif.fp`, `mlr_measures_classif.fpr`, `mlr_measures_classif.logloss`, `mlr_measures_classif.mauc_aup`, `mlr_measures_classif.mauc_aulu`, `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.fpr`, `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.p`, `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_aulu`, `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.pauc`, `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.pauc`, `mlr_measures_classif.p`, `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

$$\frac{TP}{TP + FN}$$

This is also known 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.fpr`, `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.fpr`, `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.p`, `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
<code>na_ratio</code>	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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-eval](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.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.

### 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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-eval](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.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 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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-eval](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.rsq](#), [mlr\\_measures\\_selected\\_features](#)

### Examples

```
rr = resample(tsk("iris"), lrn("classif.debug", validate = 0.3), rsmpl("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.ml-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html#sec-eval](https://mlr3book.ml-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.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

$$\frac{1}{n} \sum_{i=1}^n w_i (t_i - r_i),$$

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.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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rrse`, `mlr_measures_regr.rse`, `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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rrse`, `mlr_measures_regr.rse`, `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

$$\frac{1}{n} \sum_{i=1}^n w_i |t_i - r_i|,$$

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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rrse`, `mlr_measures_regr.rse`, `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.pinball](#), [mlr\\_measures\\_regr.rae](#), [mlr\\_measures\\_regr.rmse](#), [mlr\\_measures\\_regr.rmsle](#), [mlr\\_measures\\_regr.rrse](#), [mlr\\_measures\\_regr.rse](#), [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

$$\max(|t_i - r_i|).$$

**Dictionary**

This [Measure](#) can be instantiated via the [dictionary mlr\\_measures](#) or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rrse`, `mlr_measures_regr.rse`, `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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rrse`, `mlr_measures_regr.rse`, `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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rrse`, `mlr_measures_regr.rse`, `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

$$\frac{1}{n} \sum_{i=1}^n w_i (t_i - r_i)^2,$$

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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rrse`, `mlr_measures_regr.rse`, `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

$$\frac{1}{n} \sum_{i=1}^n w_i (\ln(1 + t_i) - \ln(1 + r_i))^2,$$

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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rrse`, `mlr_measures_regr.rse`, `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

$$\frac{1}{n} \sum_{i=1}^n w_i \frac{(t_i - r_i)}{|t_i|},$$

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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rrse`, `mlr_measures_regr.rse`, `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")
```

**Parameters**

Empty ParamSet



**Meta Information**

- Type: "regr"
- Range:  $(-\infty, \infty)$
- Minimize: TRUE
- Required prediction: response

**Note**

The score function calls `mlr3measures::pinball()` 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.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rrse`, `mlr_measures_regr.rse`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

---

`mlr_measures_regr.rae` *Relative Absolute Error*

---

**Description**

Measure to compare true observed response with predicted response in regression tasks.

**Details**

The Relative Absolute Error is defined as

$$\frac{\sum_{i=1}^n |t_i - r_i|}{\sum_{i=1}^n |t_i - \bar{t}|},$$

where  $\bar{t} = \sum_{i=1}^n t_i$ . This measure is undefined for constant  $t$ .

Can be interpreted as absolute error of the predictions relative to a naive model predicting the mean.

**Dictionary**

This [Measure](#) can be instantiated via the [dictionary](#) `mlr_measures` or with the associated sugar function `msr()`:

```
mlr_measures$get("regr.rae")
msr("regr.rae")
```

**Parameters**

Empty ParamSet

**Meta Information**

- Type: "regr"
- Range:  $[0, \infty)$
- Minimize: TRUE
- Required prediction: response

**Note**

The score function calls `mlr3measures::rae()` 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.pinball`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rrse`, `mlr_measures_regr.rse`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

---

`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

$$\sqrt{\frac{1}{n} \sum_{i=1}^n w_i (t_i - r_i)^2},$$

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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rrse`, `mlr_measures_regr.rse`, `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

$$\sqrt{\frac{1}{n} \sum_{i=1}^n w_i (\ln(1 + t_i) - \ln(1 + r_i))^2},$$

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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rrse`, `mlr_measures_regr.rse`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

---

 mlr\_measures\_regr.rrse

*Root Relative Squared Error*


---

## Description

Measure to compare true observed response with predicted response in regression tasks.

## Details

The Root Relative Squared Error is defined as

$$\sqrt{\frac{\sum_{i=1}^n (t_i - r_i)^2}{\sum_{i=1}^n (t_i - \bar{t})^2}},$$

where  $\bar{t} = \sum_{i=1}^n t_i$ .

Can be interpreted as root of the squared error of the predictions relative to a naive model predicting the mean.

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.rrse")
msr("regr.rrse")
```

## Parameters

Empty ParamSet

## Meta Information

- Type: "regr"
- Range:  $[0, \infty)$
- Minimize: TRUE
- Required prediction: response

## Note

The score function calls `mlr3measures::rrse()` 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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rse`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

`mlr_measures_regr.rse` *Relative Squared Error*

**Description**

Measure to compare true observed response with predicted response in regression tasks.

**Details**

The Relative Squared Error is defined as

$$\frac{\sum_{i=1}^n (t_i - r_i)^2}{\sum_{i=1}^n (t_i - \bar{t})^2},$$

where  $\bar{t} = \sum_{i=1}^n t_i$ .

Can be interpreted as squared error of the predictions relative to a naive model predicting the mean.

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.rse")
msr("regr.rse")
```

**Parameters**

Empty ParamSet

**Meta Information**

- Type: "regr"
- Range:  $[0, \infty)$
- Minimize: TRUE
- Required prediction: response

**Note**

The score function calls `mlr3measures::rse()` 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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rsse`, `mlr_measures_regr.sae`, `mlr_measures_regr.smape`, `mlr_measures_regr.srho`, `mlr_measures_regr.sse`

---

`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

$$1 - \frac{\sum_{i=1}^n (t_i - r_i)^2}{\sum_{i=1}^n (t_i - \bar{t})^2},$$

where  $\bar{t} = \sum_{i=1}^n t_i$ .

Also known as coefficient of determination or explained variation. Subtracts the `mlr3measures::rse()` from 1, hence it compares the squared error of the predictions relative to a naive model predicting the mean.

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](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\\_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

$$\sum_{i=1}^n |t_i - r_i|.$$

## Dictionary

This [Measure](#) can be instantiated via the [dictionary mlr\\_measures](#) or with the associated sugar function [msr\(\)](#):

```
mlr_measures$get("regr.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.pinball](#), [mlr\\_measures\\_regr.rae](#), [mlr\\_measures\\_regr.rmse](#), [mlr\\_measures\\_regr.rmsle](#), [mlr\\_measures\\_regr.rrse](#), [mlr\\_measures\\_regr.rse](#), [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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rrse`, `mlr_measures_regr.rse`, `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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rse`, `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

$$\sum_{i=1}^n (t_i - r_i)^2.$$

**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.pinball`, `mlr_measures_regr.rae`, `mlr_measures_regr.rmse`, `mlr_measures_regr.rmsle`, `mlr_measures_regr.rrse`, `mlr_measures_regr.rse`, `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](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.rsq](#)

## Examples

```
task = tsk("german_credit")
learner = lrn("classif.rpart")
rr = resample(task, learner, rsmpl("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.

## 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.

**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): [https://mlr3book.mlr-org.com/chapters/chapter3/evaluation\\_and\\_benchmarking.html#sec-resampling](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 = rsmpl("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): [https://mlr3book.mlr-org.com/chapters/chapter3/evaluation\\_and\\_benchmarking.html#sec-resampling](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 = rsmpl("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)
```

**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): [https://mlr3book.mlr-org.com/chapters/chapter3/evaluation\\_and\\_benchmarking.html#sec-resampling](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 = rsmpl("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 `rsmpl()`:

```
mlr_resamplings$get("cv")
rsmpl("cv")
```

## Parameters

- `fold` (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): [https://mlr3book.mlr-org.com/chapters/chapter3/evaluation\\_and\\_benchmarking.html#sec-resampling](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 = rsmpl("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): [https://mlr3book.mlr-org.com/chapters/chapter3/evaluation\\_and\\_benchmarking.html#sec-resampling](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 = rsmpl("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 `rsmpl()`:

```
mlr_resamplings$get("insample")
rsmpl("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.ml-org.com/chapters/chapter3/evaluation\\_and\\_benchmarking.html#sec-resampling](https://mlr3book.ml-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 = rsmpl("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 `rsmpl()`:

```
mlr_resamplings$get("loo")
rsmpl("loo")
```

## Super class

```
mlr3::Resampling -> ResamplingL00
```

## 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:

- [ResamplingL00\\$new\(\)](#)
- [ResamplingL00\\$clone\(\)](#)

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

*Usage:*

```
ResamplingL00$new()
```

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

*Usage:*

```
ResamplingL00$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](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 = rsmpl("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 `rsmpl()`:

```
mlr_resamplings$get("repeated_cv")
rsmpl("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](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 = rsm("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 `rsm()`:

```
mlr_resamplings$get("subsampling")
rsm("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): [https://mlr3book.mlr-org.com/chapters/chapter3/evaluation\\_and\\_benchmarking.html#sec-resampling](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 = rsmpl("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`.
- `rsmpl()` for a **Resampling** from `mlr_resamplings`.
- `rsmpls()` 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 <a href="#">dictionary</a> to retrieve the object.
...	(any) Additional arguments.
.keys	(character()) Keys passed to the respective <a href="#">dictionary</a> 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`, `mlr_tasks_boston_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_boston_housing
```

*Boston Housing Regression Task*

---

## Description

A regression task for the [mlbench::BostonHousing2](#) data set. This is the corrected data using the corrected median value (cmedv) as target. The uncorrected target (medv) is removed from the data.

## Format

[R6::R6Class](#) inheriting from [TaskRegr](#).

## Construction

```

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

```

## Meta Information

- Task type: "regr"
- Dimensions: 506x18
- Properties: -
- Has Missings: FALSE
- Target: "cmedv"
- Features: "age", "b", "chas", "crim", "dis", "indus", "lat", "lon", "lstat", "nox", "ptratio", "rad", "rm", "tax", "town", "tract", "zn"

**See Also**

- Chapter in the **mlr3book**: [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html](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](#)

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](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).
- **mlr3select** 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\\_boston\\_housing](#), [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](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).

- **mlr3select** 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\\_boston\\_housing](#), [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	<i>Iris Classification Task</i>
----------------	---------------------------------

---

## 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](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](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\\_boston\\_housing](#), [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](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\\_boston\\_housing](#), [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](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).
- **mlr3select** 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_boston_housing`, `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](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\\_boston\\_housing](#), [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	<i>Sonar Classification Task</i>
-----------------	----------------------------------

---

## 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](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\\_boston\\_housing](#), [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](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\\_boston\\_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\\_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](mailto: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](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\\_boston\\_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\\_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](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\\_boston\\_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\_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\\_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\\_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
resize1	integer	2	[1, ∞)
resize2	integer	2	[1, ∞)
resize3	integer	1	[1, ∞)

**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\\_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\\_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
<code>sd</code>	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\\_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\\_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_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\\_smiley](#), [mlr\\_task\\_generators\\_spirals](#), [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\\_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, ∞)
sd	numeric	0	[0, ∞)

**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\\_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\\_simplex](#), [mlr\\_task\\_generators\\_smiley](#), [mlr\\_task\\_generators\\_spirals](#)

## Examples

```
generator = tgen("xor")
plot(generator, n = 200)
```

```
task = generator$generate(200)
str(task$data())
```

## 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.*\\.R$", 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 types 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	<i>Manually Partition into Training, Test and Validation Set</i>
-----------	--

---

### 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)
```

### Arguments

task	( <a href="#">Task</a> ) Task to operate on.
ratio	( <a href="#">numeric()</a> ) 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("boston_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	<i>Predict Method for Learners</i>
-----------------	------------------------------------

---

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

**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.

**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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html](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,
  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".

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 keep 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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html](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 an 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

<code>task</code>	( <code>Task</code> ).
<code>learner</code>	( <code>Learner</code> ).
<code>pdata</code>	( <code>PredictionData</code> ) Named list inheriting from "PredictionData".
<code>...</code>	(one or more <code>PredictionData</code> objects).
<code>row_ids</code>	<code>integer()</code> Row indices.
<code>train_task</code>	( <code>Task</code> ) 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,
  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.
- `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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html](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("boston_housing")
learner = lrn("regr.featureless", predict_type = "se")
p = learner$train(task)$predict(task)
p$predict_types
head(as.data.table(p))
```

resample

*Resample a Learner on a Task***Description**

Runs a resampling (possibly in parallel): 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
)
```

**Arguments**

task	( <a href="#">Task</a> ).
learner	( <a href="#">Learner</a> ).
resampling	( <a href="#">Resampling</a> ).
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 <a href="#">DataBackend</a> of the <a href="#">Task</a> in the <a href="#">ResampleResult</a> ? Set to TRUE if your performance measures require a <a href="#">Task</a> , 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 <code>Learner\$encapsulate</code> 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 <b>evaluate</b> ) and "callr" (start in external session via <b>callr</b> ). 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 <a href="#">HotstartStack</a> .
clone	(character()) Select the input objects to be cloned before proceeding by providing a set with possible values "task", "learner" and "resampling" for <a href="#">Task</a> , <a href="#">Learner</a> and <a href="#">Resampling</a> , respectively. Per default, all input objects are cloned.
unmarshal	<a href="#">Learner</a> 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.

**Value**

[ResampleResult](#).

**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** package. One job is one resampling iteration, and all jobs are send to an apply function from **future.apply** in a single batch. To select a parallel backend, use `future::plan()`. More on parallelization can be found in the book: [https://mlr3book.mlr-org.com/chapters/chapter10/advanced\\_technical\\_aspects\\_of\\_mlr3.html](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.ml-org.com/chapters/chapter3/evaluation\\_and\\_benchmarking.html#sec-resampling](https://mlr3book.ml-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 = rsmpl("cv")

# Explicitly instantiate the resampling for this task for reproducibility
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	<i>Container for Results of <code>resample()</code></i>
----------------	---

---

### 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.
- `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$unmarshal()`
- `ResampleResult$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.

**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.

**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()`.

**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.

**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().

**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 keep the object in its previous state.

**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 keep the object in its previous state.

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

*Usage:*

```
ResampleResult$marshal(...)
```

*Arguments:*

```
... (any)
```

Additional arguments passed to `marshal_model()`.

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

*Usage:*

```
ResampleResult$unmarshal(...)
```

*Arguments:*

```
... (any)
```

Additional arguments passed to `unmarshal_model()`.

**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): [https://mlr3book.mlr-org.com/chapters/chapter3/evaluation\\_and\\_benchmarking.html#sec-resampling](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

```

---

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](#).

### 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.

### 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$instantiate()`.
- `task_nrow` (integer(1))  
The number of observations of the [Task](#) which was passed to `r$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.

### 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 keep the object in its previous state.

**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.

**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.

**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](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$param_set$values = list(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

```

---

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 <a href="#">Learner</a> . This object is modified in-place.
n	(integer(1)) Number of threads to use. Defaults to <a href="#">parallely::availableCores()</a> .
...	(any) Additional arguments.

**Value**

Same object as input x (changed in-place), with possibly updated parameter values.

---

Task	<i>Task Class</i>
------	-------------------

---

**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 [boston\\_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 affect, 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()`.

### 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$task_types$types`.

`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, formatted 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 XG-Boost. 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.

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 stored in `mlr_reflections$task_properties`. The following properties are currently standardized and understood by tasks in [mlr3](#):

- "strata": The task is resampled using one or more stratification variables (role "stratum").
- "groups": The task comes with grouping/blocking information (role "group").
- "weights": The task comes with observation weights (role "weight").

Note that above listed properties are calculated from the `$col_roles` and 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.
- "weight": Observation weights. Not more than one numeric column may have this role.

`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.

`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.

`data_formats` (`character()`)

Supported data format. Always "data.table".. This is deprecated and will be removed in the future.

`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\(\)](#))

If the task has a column with designated role "weight", a table with two columns:

- `row_id` (`integer()`), and
- observation weights `weight` (`numeric()`).

Returns NULL if there are is no weight column.

`labels` (`named character()`)

Retrieve labels (prettier formatted 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`.

## 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$add_strata()`
- `Task$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". 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, data_format, 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.

`data_format` (`character(1)`)

Deprecated. Ignored, and will be removed in the future.

`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()`.

**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()`.

**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.

**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().

**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().

**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 keep the object in its previous state.

**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 keep the object in its previous state.

**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 with the roles "target", "feature", "weight", "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 keep the object in its previous state.

**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()`).

**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 keep the object in its previous state.

**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 keep the object in its previous state.

**Method `set_col_roles()`:** Modifies the roles in `$col_roles` **in-place**.

*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 column 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 keep the object in its previous state.

**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.

**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.

**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).

**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](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](#), [mlr\\_tasks](#), [mlr\\_tasks\\_boston\\_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**

```
# 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$ncol
task$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)
```

**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): [https://mlr3book.mlr-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html](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.ml-org.com/>.

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

```
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\\_simplex](#), [mlr\\_task\\_generators\\_smiley](#), [mlr\\_task\\_generators\\_spirals](#), [mlr\\_task\\_generators\\_xor](#)

---

TaskRegr	<i>Regression Task</i>
----------	------------------------

---

## 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.ml-org.com/chapters/chapter2/data\\_and\\_basic\\_modeling.html](https://mlr3book.ml-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).
- **mlr3select** 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.ml-org.com/>.

Other Task: [Task](#), [TaskClassif](#), [TaskSupervised](#), [TaskUnsupervised](#), [mlr\\_tasks](#), [mlr\\_tasks\\_boston\\_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

```
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])
```

# Index

- \* **DataBackend**
  - as\_data\_backend.Matrix, 9
  - DataBackend, 40
  - DataBackendDataTable, 42
  - DataBackendMatrix, 45
- \* **Dictionary**
  - mlr\_learners, 82
  - mlr\_measures, 98
  - mlr\_resamplings, 185
  - mlr\_task\_generators, 220
  - mlr\_tasks, 205
- \* **Learner**
  - Learner, 52
  - LearnerClassif, 61
  - LearnerRegr, 64
  - mlr\_learners, 82
  - mlr\_learners\_classif.debug, 83
  - mlr\_learners\_classif.featureless, 86
  - mlr\_learners\_classif.rpart, 89
  - mlr\_learners\_regr.debug, 91
  - mlr\_learners\_regr.featureless, 93
  - mlr\_learners\_regr.rpart, 95
- \* **Measure**
  - Measure, 68
  - MeasureClassif, 74
  - MeasureRegr, 76
  - MeasureSimilarity, 79
  - mlr\_measures, 98
  - mlr\_measures\_aic, 99
  - mlr\_measures\_bic, 100
  - mlr\_measures\_classif.costs, 109
  - mlr\_measures\_debug\_classif, 150
  - mlr\_measures\_elapsed\_time, 152
  - mlr\_measures\_internal\_valid\_score, 154
  - mlr\_measures\_oob\_error, 155
  - mlr\_measures\_regr.rsq, 175
  - mlr\_measures\_selected\_features, 181
- \* **Prediction**
  - Prediction, 240
  - PredictionClassif, 243
  - PredictionRegr, 248
- \* **Resampling**
  - mlr\_resamplings, 185
  - mlr\_resamplings\_bootstrap, 186
  - mlr\_resamplings\_custom, 188
  - mlr\_resamplings\_custom\_cv, 189
  - mlr\_resamplings\_cv, 192
  - mlr\_resamplings\_holdout, 193
  - mlr\_resamplings\_insample, 195
  - mlr\_resamplings\_loo, 197
  - mlr\_resamplings\_repeated\_cv, 199
  - mlr\_resamplings\_subsampling, 201
  - Resampling, 258
- \* **TaskGenerator**
  - mlr\_task\_generators, 220
  - mlr\_task\_generators\_2dnormals, 221
  - mlr\_task\_generators\_cassini, 223
  - mlr\_task\_generators\_circle, 225
  - mlr\_task\_generators\_friedman1, 227
  - mlr\_task\_generators\_moons, 228
  - mlr\_task\_generators\_simplex, 230
  - mlr\_task\_generators\_smiley, 231
  - mlr\_task\_generators\_spirals, 233
  - mlr\_task\_generators\_xor, 235
  - TaskGenerator, 277
- \* **Task**
  - mlr\_tasks, 205
  - mlr\_tasks\_boston\_housing, 206
  - mlr\_tasks\_breast\_cancer, 207
  - mlr\_tasks\_german\_credit, 208
  - mlr\_tasks\_iris, 210
  - mlr\_tasks\_mtcars, 211
  - mlr\_tasks\_penguins, 212
  - mlr\_tasks\_pima, 214
  - mlr\_tasks\_sonar, 215

- mlr\_tasks\_spam, 216
- mlr\_tasks\_wine, 218
- mlr\_tasks\_zoo, 219
- Task, 263
- TaskClassif, 274
- TaskRegr, 280
- \* **benchmark**
  - benchmark, 26
  - benchmark\_grid, 37
  - BenchmarkResult, 30
- \* **binary classification measures**
  - mlr\_measures\_classif.auc, 103
  - mlr\_measures\_classif.bbrier, 106
  - mlr\_measures\_classif.dor, 111
  - mlr\_measures\_classif.fbeta, 112
  - mlr\_measures\_classif.fdr, 114
  - mlr\_measures\_classif.fn, 115
  - mlr\_measures\_classif.fnr, 117
  - mlr\_measures\_classif.fomr, 118
  - mlr\_measures\_classif.fp, 119
  - mlr\_measures\_classif.fpr, 121
  - mlr\_measures\_classif.npv, 135
  - mlr\_measures\_classif.ppv, 136
  - mlr\_measures\_classif.prauc, 138
  - mlr\_measures\_classif.precision, 139
  - mlr\_measures\_classif.recall, 140
  - mlr\_measures\_classif.sensitivity, 142
  - mlr\_measures\_classif.specificity, 143
  - mlr\_measures\_classif.tn, 144
  - mlr\_measures\_classif.tnr, 146
  - mlr\_measures\_classif.tp, 147
  - mlr\_measures\_classif.tpr, 148
- \* **classification measures**
  - mlr\_measures\_classif.acc, 102
  - mlr\_measures\_classif.auc, 103
  - mlr\_measures\_classif.bacc, 105
  - mlr\_measures\_classif.bbrier, 106
  - mlr\_measures\_classif.ce, 107
  - mlr\_measures\_classif.costs, 109
  - mlr\_measures\_classif.dor, 111
  - mlr\_measures\_classif.fbeta, 112
  - mlr\_measures\_classif.fdr, 114
  - mlr\_measures\_classif.fn, 115
  - mlr\_measures\_classif.fnr, 117
  - mlr\_measures\_classif.fomr, 118
  - mlr\_measures\_classif.fp, 119
  - mlr\_measures\_classif.fpr, 121
  - mlr\_measures\_classif.logloss, 122
  - mlr\_measures\_classif.mauc\_aup, 123
  - mlr\_measures\_classif.mauc\_aulu, 125
  - mlr\_measures\_classif.mauc\_aunp, 127
  - mlr\_measures\_classif.mauc\_aunu, 128
  - mlr\_measures\_classif.mauc\_mu, 130
  - mlr\_measures\_classif.mbrier, 132
  - mlr\_measures\_classif.mcc, 133
  - mlr\_measures\_classif.npv, 135
  - mlr\_measures\_classif.ppv, 136
  - mlr\_measures\_classif.prauc, 138
  - mlr\_measures\_classif.precision, 139
  - mlr\_measures\_classif.recall, 140
  - mlr\_measures\_classif.sensitivity, 142
  - mlr\_measures\_classif.specificity, 143
  - mlr\_measures\_classif.tn, 144
  - mlr\_measures\_classif.tnr, 146
  - mlr\_measures\_classif.tp, 147
  - mlr\_measures\_classif.tpr, 148
- \* **datasets**
  - mlr\_learners, 82
  - mlr\_measures, 98
  - mlr\_resamplings, 185
  - mlr\_task\_generators, 220
  - mlr\_tasks, 205
- \* **multiclass classification measures**
  - mlr\_measures\_classif.acc, 102
  - mlr\_measures\_classif.bacc, 105
  - mlr\_measures\_classif.ce, 107
  - mlr\_measures\_classif.costs, 109
  - mlr\_measures\_classif.logloss, 122
  - mlr\_measures\_classif.mauc\_aup, 123
  - mlr\_measures\_classif.mauc\_aulu, 125
  - mlr\_measures\_classif.mauc\_aunp, 127
  - mlr\_measures\_classif.mauc\_aunu, 128

- mlr\_measures\_classif.mauc\_mu, 130
- mlr\_measures\_classif.mbrier, 132
- mlr\_measures\_classif.mcc, 133
- \* regression measures**
  - mlr\_measures\_regr.bias, 157
  - mlr\_measures\_regr.ktau, 158
  - mlr\_measures\_regr.mae, 159
  - mlr\_measures\_regr.mape, 160
  - mlr\_measures\_regr.maxae, 161
  - mlr\_measures\_regr.medae, 162
  - mlr\_measures\_regr.medse, 163
  - mlr\_measures\_regr.mse, 164
  - mlr\_measures\_regr.msle, 166
  - mlr\_measures\_regr.pbias, 167
  - mlr\_measures\_regr.pinball, 168
  - mlr\_measures\_regr.rae, 169
  - mlr\_measures\_regr.rmse, 170
  - mlr\_measures\_regr.rmsle, 171
  - mlr\_measures\_regr.rrse, 173
  - mlr\_measures\_regr.rse, 174
  - mlr\_measures\_regr.sae, 177
  - mlr\_measures\_regr.smape, 178
  - mlr\_measures\_regr.srho, 179
  - mlr\_measures\_regr.sse, 180
- \* resample**
  - resample, 250
  - ResampleResult, 253
- \* similarity measures**
  - mlr\_measures\_sim.jaccard, 183
  - mlr\_measures\_sim.phi, 184
- acc(), 105
- as\_benchmark\_result, 9
- as\_benchmark\_result(), 18, 252, 257
- as\_data\_backend
  - (as\_data\_backend.Matrix), 9
- as\_data\_backend(), 41, 59
- as\_data\_backend.Matrix, 9, 41, 44, 47
- as\_learner, 11
- as\_learners(as\_learner), 11
- as\_measure, 11
- as\_measures(as\_measure), 11
- as\_prediction, 12
- as\_prediction(), 247
- as\_prediction\_classif, 13
- as\_prediction\_data, 14
- as\_prediction\_data(), 247
- as\_prediction\_regr, 15
- as\_predictions(as\_prediction), 12
- as\_resample\_result, 16
- as\_resample\_result(), 254
- as\_resampling, 17
- as\_resamplings(as\_resampling), 17
- as\_result\_data, 18
- as\_result\_data(), 17, 31, 254
- as\_task, 19
- as\_task\_classif, 20
- as\_task\_classif(), 275
- as\_task\_regr, 23
- as\_task\_regr(), 280
- as\_task\_unsupervised, 25
- as\_tasks(as\_task), 19
- as\_tasks\_unsupervised
  - (as\_task\_unsupervised), 25
- bbrier(), 132
- benchmark, 26, 36, 38
- benchmark(), 7, 13, 30, 49, 54–56, 59, 70, 72, 73, 75, 78, 81, 152, 258
- benchmark\_grid, 28, 36, 37
- benchmark\_grid(), 27
- BenchmarkResult, 9, 18, 27, 28, 30, 30, 31, 32, 35, 38, 55, 59, 70, 252–254, 257
- bootstrap, 258
- boston\_housing, 263
- c(), 32
- c.PredictionDataClassif
  - (PredictionData), 246
- c.PredictionDataRegr (PredictionData), 246
- check\_prediction\_data (PredictionData), 246
- classif.auc, 68
- classif.ce, 74
- classif.rpart, 52
- convert\_task, 39
- convert\_task(), 20, 23, 265, 268, 276, 280
- create\_empty\_prediction\_data
  - (PredictionData), 246
- cut(), 273
- cv, 258
- data.frame(), 10, 20, 23, 27, 45, 240, 267
- data.table(), 43, 254
- data.table::as.data.table(), 10
- data.table::copy(), 43

- `data.table::data.table()`, [10](#), [27](#), [30](#), [31](#),  
[33](#), [34](#), [38](#), [40](#), [41](#), [43](#), [45](#), [46](#), [49](#), [56](#),  
[82](#), [98](#), [185](#), [205](#), [221](#), [241](#), [253](#), [256](#),  
[264–267](#), [269](#), [270](#)
- `DataBackend`, [9](#), [10](#), [20](#), [23](#), [27](#), [35](#), [40](#), [40](#), [42](#),  
[44](#), [45](#), [47](#), [59](#), [250](#), [257](#), [263–265](#),  
[267–271](#), [276](#), [280](#)
- `DataBackendDataTable`, [9](#), [10](#), [40](#), [41](#), [42](#), [47](#),  
[268](#), [276](#), [280](#)
- `DataBackendMatrix`, [9](#), [10](#), [40](#), [41](#), [44](#), [45](#)
- `datasets::iris`, [210](#)
- `datasets::mtcars`, [211](#)
- `default_fallback`, [47](#)
- `default_measures`, [48](#)
- `default_measures()`, [255](#), [256](#)
- `Dictionary`, [61](#), [63](#), [66](#), [73](#), [76](#), [79](#), [81](#), [86](#), [88](#),  
[90](#), [93](#), [95](#), [97](#), [100](#), [102–104](#),  
[106–108](#), [110](#), [112](#), [113](#), [115](#), [116](#),  
[118–120](#), [122](#), [123](#), [125](#), [126](#), [128](#),  
[130](#), [131](#), [133](#), [134](#), [136](#), [137](#),  
[139–141](#), [143–145](#), [147–149](#), [151](#),  
[153](#), [155](#), [156](#), [158–165](#), [167–172](#),  
[174–176](#), [178–182](#), [184](#), [185](#), [187](#),  
[189](#), [191](#), [193](#), [195](#), [196](#), [198](#), [200](#),  
[202](#), [207–209](#), [211–214](#), [216](#), [217](#),  
[219](#), [220](#), [223](#), [224](#), [226](#), [228](#), [229](#),  
[231](#), [233](#), [234](#), [236](#), [261](#), [274](#), [277](#),  
[279](#), [281](#)
- `dictionary`, [52](#), [61](#), [64](#), [68](#), [74](#), [76](#), [79](#), [83](#), [87](#),  
[89](#), [91](#), [93](#), [95](#), [99](#), [101](#), [102](#),  
[104–106](#), [108](#), [109](#), [111](#), [113](#), [114](#),  
[116–118](#), [120–122](#), [124](#), [126](#), [127](#),  
[129](#), [131](#), [132](#), [134](#), [135](#), [137–139](#),  
[141–143](#), [145–147](#), [149](#), [150](#), [152](#),  
[154](#), [155](#), [157–162](#), [164–169](#),  
[171–175](#), [177–181](#), [183](#), [184](#), [186](#),  
[188](#), [190](#), [192](#), [194](#), [195](#), [197](#), [199](#),  
[201](#), [204](#), [207](#), [209](#), [210](#), [213–216](#),  
[218](#), [219](#), [221](#), [223](#), [225](#), [227](#), [228](#),  
[230](#), [232](#), [233](#), [235](#), [258](#), [263](#), [275](#),  
[277](#), [280](#)
- `expand.grid()`, [37](#)
- `extract_pkgs(install_pkgs)`, [51](#)
- `filter_prediction_data`  
(`PredictionData`), [246](#)
- `formula`, [20](#), [23](#)
- `formula()`, [269](#), [270](#)
- `future::future()`, [7](#)
- `future::multicore`, [262](#)
- `future::nbrOfWorkers()`, [56](#)
- `future::plan`, [262](#)
- `future::plan()`, [28](#), [56](#), [251](#)
- `head()`, [264](#)
- `HotstartStack`, [27](#), [49](#), [57](#), [251](#)
- `install_pkgs`, [51](#)
- `intersect()`, [266](#)
- `iris data set`, [212](#)
- `is_marshaled_model` (`marshaling`), [67](#)
- `is_missing_prediction_data`  
(`PredictionData`), [246](#)
- `Learner`, [11](#), [13](#), [17](#), [19](#), [26](#), [27](#), [30](#), [32](#), [34](#), [35](#),  
[37](#), [48](#), [50](#), [51](#), [52](#), [53–55](#), [57](#), [58](#),  
[60–70](#), [72](#), [73](#), [75](#), [78–83](#), [86–89](#), [91](#),  
[93](#), [95](#), [97](#), [154](#), [155](#), [181](#), [203](#), [204](#),  
[237–240](#), [242](#), [247](#), [250](#), [251](#), [253](#),  
[255](#), [257](#), [263](#), [265](#)
- `learner_marshal` (`marshaling`), [67](#)
- `learner_marshaled` (`marshaling`), [67](#)
- `learner_unmarshal` (`marshaling`), [67](#)
- `LearnerClassif`, [52](#), [57](#), [61](#), [61](#), [66](#), [82](#), [83](#),  
[86](#), [88](#), [89](#), [91](#), [93](#), [95](#), [97](#), [243](#)
- `LearnerClassifDebug`, [54](#), [55](#), [68](#)
- `LearnerClassifDebug`  
(`mlr_learners_classif.debug`),  
[83](#)
- `LearnerClassifFeatureless`  
(`mlr_learners_classif.featureless`),  
[86](#)
- `LearnerClassifRpart`  
(`mlr_learners_classif.rpart`),  
[89](#)
- `LearnerRegr`, [52](#), [57](#), [61](#), [63](#), [64](#), [82](#), [86](#), [88](#),  
[91](#), [93](#), [95](#), [97](#), [248](#)
- `LearnerRegrDebug`  
(`mlr_learners_regr.debug`), [91](#)
- `LearnerRegrFeatureless`  
(`mlr_learners_regr.featureless`),  
[93](#)
- `LearnerRegrRpart`  
(`mlr_learners_regr.rpart`), [95](#)
- `Learners`, [61](#), [63](#), [66](#), [86](#), [88](#), [90](#), [93](#), [95](#), [97](#)
- `lgr::lgr-package`, [28](#), [252](#)
- `lrn` (`mlr_sugar`), [203](#)

- lrn(), [82](#), [83](#), [87](#), [89](#), [91](#), [93](#), [95](#)
- lrns (mlr\_sugar), [203](#)
- lrns(), [82](#)
- mad(), [93](#)
- marshal\_model (marshaling), [67](#)
- marshal\_model(), [32](#), [85](#), [257](#)
- marshaling, [55](#), [67](#)
- matrix(), [20](#), [23](#)
- Matrix::Matrix(), [20](#), [23](#), [41](#), [45](#)
- Matrix::sparseMatrix(), [40](#)
- max.col(), [245](#)
- mbrier(), [106](#)
- mean(), [71](#), [75](#), [77](#), [80](#), [93](#)
- Measure, [11](#), [12](#), [32–34](#), [48](#), [51](#), [55](#), [68](#), [70](#), [74](#), [76](#), [79](#), [81](#), [98–175](#), [177–181](#), [183–185](#), [203](#), [242](#), [243](#), [255](#), [256](#)
- MeasureAIC (mlr\_measures\_aic), [99](#)
- MeasureBIC (mlr\_measures\_bic), [100](#)
- MeasureClassif, [68](#), [70](#), [73](#), [74](#), [79](#), [81](#), [98](#), [100](#), [102](#), [110](#), [151](#), [153](#), [155](#), [156](#), [177](#), [183](#)
- MeasureClassifCosts, [208](#)
- MeasureClassifCosts (mlr\_measures\_classif.costs), [109](#)
- MeasureDebugClassif (mlr\_measures\_debug\_classif), [150](#)
- MeasureElapsedTime (mlr\_measures\_elapsed\_time), [152](#)
- MeasureInternalValidScore (mlr\_measures\_internal\_valid\_score), [154](#)
- MeasureOOBError (mlr\_measures\_oob\_error), [155](#)
- MeasureRegr, [68](#), [70](#), [73](#), [76](#), [76](#), [81](#), [98](#), [100](#), [102](#), [110](#), [151](#), [153](#), [155](#), [156](#), [177](#), [183](#)
- MeasureRegrRSQ (mlr\_measures\_regr.rsq), [175](#)
- Measures, [73](#), [76](#), [79](#), [81](#), [100](#), [102–104](#), [106–108](#), [110](#), [112](#), [113](#), [115](#), [116](#), [118–120](#), [122](#), [123](#), [125](#), [126](#), [128](#), [130](#), [131](#), [133](#), [134](#), [136](#), [137](#), [139–141](#), [143–145](#), [147–149](#), [151](#), [153](#), [155](#), [156](#), [158–165](#), [167–172](#), [174–176](#), [178–182](#), [184](#), [185](#)
- MeasureSelectedFeatures (mlr\_measures\_selected\_features), [181](#)
- MeasureSimilarity, [73](#), [76](#), [79](#), [79](#), [98](#), [100](#), [102](#), [110](#), [151](#), [153](#), [155](#), [156](#), [177](#), [183](#)
- median(), [93](#)
- mlbench::BostonHousing2, [206](#)
- mlbench::BreastCancer, [207](#)
- mlbench::mlbench.2dnormals(), [221](#)
- mlbench::mlbench.cassini(), [223](#)
- mlbench::mlbench.circle(), [225](#)
- mlbench::mlbench.friedman1(), [227](#)
- mlbench::mlbench.simplex(), [230](#)
- mlbench::mlbench.smiley(), [231](#)
- mlbench::mlbench.spirals(), [233](#)
- mlbench::mlbench.xor(), [235](#)
- mlbench::PimaIndiansDiabetes2, [214](#)
- mlbench::Sonar, [215](#)
- mlbench::Zoo, [219](#)
- mlr3 (mlr3-package), [6](#)
- mlr3-package, [6](#)
- mlr3::DataBackend, [42](#), [45](#)
- mlr3::Learner, [62](#), [64](#), [84](#), [87](#), [90](#), [92](#), [94](#), [96](#)
- mlr3::LearnerClassif, [84](#), [87](#), [90](#)
- mlr3::LearnerRegr, [92](#), [94](#), [96](#)
- mlr3::Measure, [74](#), [76](#), [79](#), [99](#), [101](#), [109](#), [150](#), [152](#), [154](#), [156](#), [176](#), [182](#)
- mlr3::MeasureClassif, [109](#)
- mlr3::MeasureRegr, [176](#)
- mlr3::Prediction, [244](#), [248](#)
- mlr3::Resampling, [186](#), [188](#), [190](#), [192](#), [194](#), [196](#), [197](#), [199](#), [201](#)
- mlr3::Task, [275](#), [280](#)
- mlr3::TaskGenerator, [222](#), [224](#), [225](#), [227](#), [228](#), [230](#), [232](#), [234](#), [235](#)
- mlr3::TaskSupervised, [275](#), [280](#)
- mlr3measures::acc(), [103](#)
- mlr3measures::auc(), [104](#)
- mlr3measures::bacc(), [105](#)
- mlr3measures::bbrier(), [107](#)
- mlr3measures::bias(), [157](#)
- mlr3measures::ce(), [108](#)
- mlr3measures::dor(), [112](#)
- mlr3measures::fbeta(), [113](#)
- mlr3measures::fdr(), [115](#)
- mlr3measures::fn(), [116](#)
- mlr3measures::fnr(), [117](#)

- `mlr3measures::fomr()`, 119
- `mlr3measures::fp()`, 120
- `mlr3measures::fpr()`, 121
- `mlr3measures::jaccard()`, 184
- `mlr3measures::ktau()`, 159
- `mlr3measures::logloss()`, 123
- `mlr3measures::mae()`, 160
- `mlr3measures::mape()`, 161
- `mlr3measures::mauc_aup()`, 124
- `mlr3measures::mauc_aupu()`, 126
- `mlr3measures::mauc_aunp()`, 128
- `mlr3measures::mauc_aunu()`, 129
- `mlr3measures::mauc_mu()`, 131
- `mlr3measures::maxae()`, 162
- `mlr3measures::mbrier()`, 132
- `mlr3measures::mcc()`, 134
- `mlr3measures::medae()`, 163
- `mlr3measures::medse()`, 164
- `mlr3measures::mse()`, 165
- `mlr3measures::msle()`, 166
- `mlr3measures::npv()`, 136
- `mlr3measures::pbias()`, 168
- `mlr3measures::phi()`, 185
- `mlr3measures::pinball()`, 169
- `mlr3measures::ppv()`, 137
- `mlr3measures::prauc()`, 138
- `mlr3measures::precision()`, 140
- `mlr3measures::rae()`, 170
- `mlr3measures::recall()`, 141
- `mlr3measures::rmse()`, 171
- `mlr3measures::rmsle()`, 172
- `mlr3measures::rrse()`, 173
- `mlr3measures::rse()`, 175
- `mlr3measures::sae()`, 177
- `mlr3measures::sensitivity()`, 142
- `mlr3measures::smape()`, 179
- `mlr3measures::specificity()`, 144
- `mlr3measures::srho()`, 180
- `mlr3measures::sse()`, 181
- `mlr3measures::tn()`, 145
- `mlr3measures::tnr()`, 146
- `mlr3measures::tp()`, 148
- `mlr3measures::tpr()`, 149
- `mlr3misc::Dictionary`, 82, 98, 185, 203, 205, 220, 221
- `mlr3misc::dictionary_sugar_get()`, 203
- `mlr3misc::encapsulate()`, 56
- `mlr3misc::insert_named()`, 53
- `mlr3misc::unnest()`, 34
- `mlr_learners`, 52, 61, 63, 64, 66, 82, 83, 86–91, 93, 95, 97, 98, 185, 203, 205, 221
- `mlr_learners_classif.debug`, 61, 63, 66, 82, 83, 88, 91, 93, 95, 97
- `mlr_learners_classif.featureless`, 61, 63, 66, 82, 86, 86, 91, 93, 95, 97
- `mlr_learners_classif.rpart`, 61, 63, 66, 82, 86, 88, 89, 93, 95, 97
- `mlr_learners_regr.debug`, 61, 63, 66, 82, 86, 88, 91, 91, 95, 97
- `mlr_learners_regr.featureless`, 61, 63, 66, 82, 86, 88, 91, 93, 93, 97
- `mlr_learners_regr.rpart`, 61, 63, 66, 82, 86, 88, 91, 93, 95, 95
- `mlr_measures`, 68, 73, 74, 76, 79, 81, 82, 98, 99–185, 203, 205, 221
- `mlr_measures_aic`, 53, 73, 76, 79, 81, 98, 99, 102, 110, 151, 153, 155, 156, 177, 183
- `mlr_measures_bic`, 53, 73, 76, 79, 81, 98, 100, 100, 110, 151, 153, 155, 156, 177, 183
- `mlr_measures_classif.acc`, 102, 104, 106–110, 112, 113, 115, 116, 118–120, 122, 123, 125, 126, 128, 130, 131, 133–137, 139–141, 143–145, 147–149
- `mlr_measures_classif.auc`, 103, 103, 106–108, 110, 112–116, 118–120, 122, 123, 125, 126, 128, 130, 131, 133, 134, 136, 137, 139–145, 147–150
- `mlr_measures_classif.bacc`, 103, 104, 105, 107–110, 112, 113, 115, 116, 118–120, 122, 123, 125, 126, 128, 130, 131, 133–137, 139–141, 143–145, 147–149
- `mlr_measures_classif.bbrier`, 103, 104, 106, 106, 108, 110, 112–116, 118–120, 122, 123, 125, 126, 128, 130, 131, 133, 134, 136, 137, 139–145, 147–150
- `mlr_measures_classif.ce`, 103, 104, 106, 107, 107, 110, 112, 113, 115, 116, 118–120, 122, 123, 125, 126, 128, 130, 131, 133–137, 139–141,

- 143–145, 147–149
- `mlr_measures_classif.costs`, 73, 76, 79, 81, 98, 100, 102–104, 106–109, 109, 112, 113, 115, 116, 118–120, 122, 123, 125, 126, 128, 130, 131, 133–137, 139–141, 143–145, 147–149, 151, 153, 155, 156, 177, 183
- `mlr_measures_classif.dor`, 103, 104, 106–108, 110, 111, 114–116, 118–120, 122, 123, 125, 126, 128, 130, 131, 133, 134, 136, 137, 139–145, 147–150
- `mlr_measures_classif.fbeta`, 103, 104, 106–108, 110, 112, 112, 115, 116, 118–120, 122, 123, 125, 126, 128, 130, 131, 133, 134, 136, 137, 139–145, 147–150
- `mlr_measures_classif.fdr`, 103, 104, 106–108, 110, 112, 114, 114, 116, 118–120, 122, 123, 125, 126, 128, 130, 131, 133, 134, 136, 137, 139–145, 147–150
- `mlr_measures_classif.fn`, 103, 104, 106–108, 110, 112, 114, 115, 115, 118–120, 122, 123, 125, 126, 128, 130, 131, 133, 134, 136, 137, 139–145, 147–150
- `mlr_measures_classif.fnr`, 103, 104, 106–108, 110, 112, 114–116, 117, 119, 120, 122, 123, 125, 126, 128, 130, 131, 133, 134, 136, 137, 139–145, 147–150
- `mlr_measures_classif.fomr`, 103, 104, 106–108, 110, 112, 114–116, 118, 118, 120, 122, 123, 125, 126, 128, 130, 131, 133, 134, 136, 137, 139–145, 147–150
- `mlr_measures_classif.fp`, 103, 104, 106–108, 110, 112, 114–116, 118, 119, 119, 122, 123, 125, 126, 128, 130, 131, 133, 134, 136, 137, 139–145, 147–150
- `mlr_measures_classif.fpr`, 103, 104, 106–108, 110, 112, 114–116, 118–120, 121, 123, 125, 126, 128, 130, 131, 133, 134, 136, 137, 139–145, 147–150
- `mlr_measures_classif.logloss`, 103, 104, 106–110, 112, 114–116, 118–120, 122, 122, 125, 126, 128, 130, 131, 133–137, 139–141, 143–145, 147–149
- `mlr_measures_classif.mauc_aup`, 103, 104, 106–110, 112, 114–116, 118–120, 122, 123, 123, 126, 128, 130, 131, 133–137, 139–141, 143–145, 147–149
- `mlr_measures_classif.mauc_aup1u`, 103, 104, 106–110, 112, 114–116, 118–120, 122, 123, 125, 125, 128, 130, 131, 133–137, 139–141, 143–145, 147–149
- `mlr_measures_classif.mauc_aunp`, 103, 104, 106–110, 112, 114–116, 118–120, 122, 123, 125, 126, 127, 130, 131, 133–137, 139–141, 143–145, 147–149
- `mlr_measures_classif.mauc_aunu`, 103, 104, 106–110, 112, 114–116, 118–120, 122, 123, 125, 126, 128, 128, 131, 133–137, 139–141, 143–145, 147–149
- `mlr_measures_classif.mauc_mu`, 103, 104, 106–110, 112, 114–116, 118–120, 122, 123, 125, 126, 128, 130, 130, 133–137, 139–141, 143–145, 147–149
- `mlr_measures_classif.mbrier`, 103, 104, 106–110, 112, 114–116, 118–120, 122, 123, 125, 126, 128, 130, 131, 132, 134–137, 139–141, 143–145, 147–149
- `mlr_measures_classif.mcc`, 103, 104, 106–110, 112, 114–116, 118–120, 122, 123, 125, 126, 128, 130, 131, 133, 133, 136, 137, 139–141, 143–145, 147–149
- `mlr_measures_classif.npv`, 103, 104, 106–108, 110, 112, 114–116, 118–120, 122, 123, 125, 126, 128, 130, 131, 133, 134, 135, 137, 139–150
- `mlr_measures_classif.ppv`, 103, 104, 106–108, 110, 112, 114–116, 118–123, 125, 126, 128, 130, 131,



- 133, 134, 136, 136, 139–150*  
 mlr\_measures\_classif.prauc, *103, 104, 106–108, 110, 112, 114–116, 118–123, 125, 126, 128, 130, 131, 133, 134, 136, 137, 138, 140–150*  
 mlr\_measures\_classif.precision, *103, 104, 106–108, 110, 112, 114–116, 118–123, 125, 126, 128, 130, 131, 133, 135–137, 139, 139, 141–150*  
 mlr\_measures\_classif.recall, *103, 104, 106–108, 110, 112, 114–116, 118–123, 125, 126, 128, 130, 131, 133, 135–137, 139, 140, 140, 143–150*  
 mlr\_measures\_classif.sensitivity, *103, 104, 106–108, 110, 112, 114–116, 118–123, 125, 126, 128, 130, 131, 133, 135–137, 139–142, 142, 144–150*  
 mlr\_measures\_classif.specificity, *103, 104, 106–108, 110, 112, 114–116, 118–123, 125, 126, 128, 130, 131, 133, 135–137, 139–143, 143, 145–150*  
 mlr\_measures\_classif.tn, *103, 104, 106–108, 110, 112, 114–116, 118–123, 125, 126, 128, 130, 131, 133, 135–137, 139–144, 144, 147–150*  
 mlr\_measures\_classif.tnr, *103, 104, 106–108, 110, 112, 114–116, 118–123, 125, 126, 128, 130, 131, 133, 135–137, 139–146, 146, 148–150*  
 mlr\_measures\_classif.tp, *103, 104, 106–108, 110, 112, 114–116, 118–123, 125, 126, 128, 130, 131, 133, 135–137, 139–147, 147, 149, 150*  
 mlr\_measures\_classif.tpr, *103, 104, 106–108, 110, 112, 114–116, 118–123, 125, 126, 128, 130, 131, 133, 135–137, 139–148, 148*  
 mlr\_measures\_debug\_classif, *73, 76, 79, 81, 98, 100, 102, 110, 150, 153, 155, 156, 177, 183*  
 mlr\_measures\_elapsed\_time, *73, 76, 79, 81, 98, 100, 102, 110, 151, 152, 155, 156, 177, 183*  
 mlr\_measures\_internal\_valid\_score, *73, 76, 79, 81, 98, 100, 102, 110, 151, 153, 154, 156, 177, 183*  
 mlr\_measures\_oob\_error, *73, 76, 79, 81, 98, 100, 102, 110, 151, 153, 155, 155, 177, 183*  
 mlr\_measures\_regr.bias, *157, 159–165, 167–172, 174, 175, 178–181*  
 mlr\_measures\_regr.ktau, *158, 158, 160–165, 167–172, 174, 175, 178–181*  
 mlr\_measures\_regr.mae, *158, 159, 159, 161–165, 167–172, 174, 175, 178–181*  
 mlr\_measures\_regr.mape, *158–160, 160, 162–165, 167–172, 174, 175, 178–181*  
 mlr\_measures\_regr.maxae, *158–161, 161, 163–165, 167–172, 174, 175, 178–181*  
 mlr\_measures\_regr.medae, *158–162, 162, 164, 165, 167–172, 174, 175, 178–181*  
 mlr\_measures\_regr.medse, *158–163, 163, 165, 167–172, 174, 175, 178–181*  
 mlr\_measures\_regr.mse, *158–164, 164, 167–172, 174, 175, 178–181*  
 mlr\_measures\_regr.msle, *158–165, 166, 168–172, 174, 175, 178–181*  
 mlr\_measures\_regr.pbias, *158–165, 167, 167, 169–172, 174, 175, 178–181*  
 mlr\_measures\_regr.pinball, *158–165, 167, 168, 168, 170–172, 174, 175, 178–181*  
 mlr\_measures\_regr.rae, *158–165, 167–169, 169, 171, 172, 174, 175, 178–181*  
 mlr\_measures\_regr.rmse, *158–165, 167–170, 170, 172, 174, 175, 178–181*  
 mlr\_measures\_regr.rmsle, *158–165, 167–171, 171, 174, 175, 178–181*  
 mlr\_measures\_regr.rrse, *158–165, 167–172, 173, 175, 178–181*  
 mlr\_measures\_regr.rse, *158–165, 167–172, 174, 174, 178–181*  
 mlr\_measures\_regr.rsq, *73, 76, 79, 81, 98, 100, 102, 110, 151, 153, 155, 156,*

- 175, 183
- mlr\_measures\_regr.sae, 158–165, 167–172, 174, 175, 177, 179–181
- mlr\_measures\_regr.smape, 158–165, 167–172, 174, 175, 178, 178, 180, 181
- mlr\_measures\_regr.srho, 158–165, 167–172, 174, 175, 178, 179, 179, 181
- mlr\_measures\_regr.sse, 158–165, 167–172, 174, 175, 178–180, 180
- mlr\_measures\_selected\_features, 73, 76, 79, 81, 98, 100, 102, 110, 151, 153, 155, 156, 177, 181
- mlr\_measures\_sim.jaccard, 183, 185
- mlr\_measures\_sim.phi, 184, 184
- mlr\_measures\_time\_both  
(mlr\_measures\_elapsed\_time), 152
- mlr\_measures\_time\_predict  
(mlr\_measures\_elapsed\_time), 152
- mlr\_measures\_time\_train  
(mlr\_measures\_elapsed\_time), 152
- mlr\_reflections, 240
- mlr\_reflections\$default\_measures, 12, 48
- mlr\_reflections\$learner\_predict\_types, 55, 58, 62, 65, 72, 75, 78, 80
- mlr\_reflections\$learner\_properties, 55, 58, 62, 65
- mlr\_reflections\$measure\_properties, 71, 75, 78, 80
- mlr\_reflections\$task\_feature\_types, 55, 58, 62, 65, 264
- mlr\_reflections\$task\_properties, 265
- mlr\_reflections\$task\_types, 39
- mlr\_reflections\$task\_types\$type, 55, 58, 69, 71, 264, 268, 277, 278
- mlr\_resamplings, 82, 98, 185, 186–203, 205, 221, 258, 261
- mlr\_resamplings\_bootstrap, 185, 186, 189, 191, 193, 195, 196, 198, 200, 202, 261
- mlr\_resamplings\_custom, 185, 187, 188, 191, 193, 195, 196, 198, 200, 202, 261
- mlr\_resamplings\_custom\_cv, 185, 187, 189, 189, 193, 195, 196, 198, 200, 202, 261
- mlr\_resamplings\_cv, 185, 187, 189, 191, 192, 195, 196, 198, 200, 202, 261
- mlr\_resamplings\_holdout, 185, 187, 189, 191, 193, 193, 196, 198, 200, 202, 261
- mlr\_resamplings\_insample, 185, 187, 189, 191, 193, 195, 195, 198, 200, 202, 261
- mlr\_resamplings\_loo, 185, 187, 189–191, 193, 195, 196, 197, 200, 202, 261
- mlr\_resamplings\_repeated\_cv, 185, 187, 189, 191, 193, 195, 196, 198, 199, 202, 261
- mlr\_resamplings\_subsampling, 185, 187, 189, 191, 193, 195, 196, 198, 200, 201, 261
- mlr\_sugar, 203
- mlr\_task\_generators, 82, 98, 185, 203, 205, 220, 221, 223–236, 277, 279
- mlr\_task\_generators\_2dnormals, 221, 221, 224, 226, 228, 229, 231, 233, 234, 236, 279
- mlr\_task\_generators\_cassini, 221, 223, 223, 226, 228, 229, 231, 233, 234, 236, 279
- mlr\_task\_generators\_circle, 221, 223, 224, 225, 228, 229, 231, 233, 234, 236, 279
- mlr\_task\_generators\_friedman1, 221, 223, 224, 226, 227, 229, 231, 233, 234, 236, 279
- mlr\_task\_generators\_moons, 221, 223, 224, 226, 228, 228, 231, 233, 234, 236, 279
- mlr\_task\_generators\_simplex, 221, 223, 224, 226, 228, 229, 230, 233, 234, 236, 279
- mlr\_task\_generators\_smiley, 221, 223, 224, 226, 228, 229, 231, 231, 234, 236, 279
- mlr\_task\_generators\_spirals, 221, 223, 224, 226, 228, 229, 231, 233, 233, 236, 279
- mlr\_task\_generators\_xor, 221, 223, 224, 226, 228, 229, 231, 233, 234, 235,

- 279
- mlr\_tasks, 82, 98, 185, 203, 205, 207–221, 263, 274, 275, 277, 280, 281
- mlr\_tasks\_boston\_housing, 205, 206, 208, 210–212, 214–217, 219, 220, 274, 277, 281
- mlr\_tasks\_breast\_cancer, 205, 207, 207, 210–212, 214–217, 219, 220, 274, 277, 281
- mlr\_tasks\_german\_credit, 205, 207, 208, 208, 211, 212, 214–217, 219, 220, 274, 277, 281
- mlr\_tasks\_iris, 205, 207, 208, 210, 210, 212, 214–217, 219, 220, 274, 277, 281
- mlr\_tasks\_mtcars, 205, 207, 208, 210, 211, 211, 214–217, 219, 220, 274, 277, 281
- mlr\_tasks\_penguins, 205, 207, 208, 210–212, 212, 215–217, 219, 220, 274, 277, 281
- mlr\_tasks\_pima, 205, 207, 208, 210–212, 214, 214, 216, 217, 219, 220, 274, 277, 281
- mlr\_tasks\_sonar, 205, 207, 208, 210–212, 214, 215, 215, 217, 219, 220, 274, 277, 281
- mlr\_tasks\_spam, 205, 207, 208, 210–212, 214–216, 216, 219, 220, 274, 277, 281
- mlr\_tasks\_wine, 205, 207, 208, 210–212, 214–217, 218, 220, 274, 277, 281
- mlr\_tasks\_zoo, 205, 207, 208, 210–212, 214–217, 219, 219, 274, 277, 281
- mlr\_test\_helpers, 237
- msr (mlr\_sugar), 203
- msr(), 98, 99, 101, 102, 104–106, 108, 109, 111, 113, 114, 116–118, 120–122, 124, 126, 127, 129, 131, 132, 134, 135, 137–139, 141–143, 145–147, 149, 150, 152, 154, 155, 157–162, 164–169, 171–175, 177–181, 183, 184
- msrs (mlr\_sugar), 203
- msrs(), 98
- order(), 266
- palmerpenguins::penguins, 212
- paradox::ParamSet, 52, 53, 57, 58, 62, 65, 69, 71, 74, 77, 80, 259, 260, 278
- parallelly::availableCores(), 263
- ParamSet, 54
- partition, 239
- partition(), 59
- penguins, 263
- plot(), 222, 224, 226, 229, 231, 232, 234, 236
- precision, 113
- precision(), 113
- predict.Learner, 239
- Prediction, 12, 13, 17, 19, 27, 30, 32, 55, 59–61, 64, 70, 71, 73, 75, 77, 80, 239, 240, 240, 241–243, 246, 247, 249, 251, 255, 256
- PredictionClassif, 13, 14, 61, 150, 240, 243, 243, 249
- PredictionData, 14, 15, 246, 247, 248
- PredictionRegr, 15, 16, 64, 240, 243, 246, 248
- progressr::handlers(), 28, 251
- progressr::with\_progress(), 28, 251
- R6, 31, 41, 43, 45, 49, 57, 62, 65, 70, 74, 77, 79, 85, 87, 90, 92, 94, 96, 100, 101, 110, 151, 153, 154, 156, 176, 182, 186, 188, 190, 192, 194, 196, 197, 199, 202, 222, 224, 225, 227, 229, 230, 232, 234, 235, 244, 248, 254, 260, 268, 275, 278, 280
- R6::R6Class, 82, 98, 185, 204–208, 210–212, 214–216, 218–220
- recall, 113
- recall(), 113
- regr.mse, 76
- regr.rpart, 52
- remotes::install\_cran(), 52
- remotes::install\_github(), 52
- requireNamespace(), 58, 63, 66, 72, 75, 78, 81, 278
- resample, 250, 257
- resample(), 7, 13, 49, 54–56, 59, 70, 72, 73, 75, 78, 81, 152, 253, 258
- ResampleResult, 13, 17–19, 27, 30, 31, 33–35, 55, 59, 70–73, 75, 77, 78, 80, 81, 250–253, 253, 254, 256
- Resampling, 17, 19, 26, 27, 31, 32, 35, 37, 72, 75, 78, 80, 185–203, 250, 251, 253, 255, 258, 259, 261, 267

- ResamplingBootstrap, [258](#)
- ResamplingBootstrap
  - (mlr\_resamplings\_bootstrap), [186](#)
- ResamplingCustom
  - (mlr\_resamplings\_custom), [188](#)
- ResamplingCustomCV
  - (mlr\_resamplings\_custom\_cv), [189](#)
- ResamplingCV, [258](#), [260](#)
- ResamplingCV (mlr\_resamplings\_cv), [192](#)
- ResamplingHoldout, [260](#)
- ResamplingHoldout
  - (mlr\_resamplings\_holdout), [193](#)
- ResamplingInsample
  - (mlr\_resamplings\_insample), [195](#)
- ResamplingLOO (mlr\_resamplings\_loo), [197](#)
- ResamplingRepeatedCV
  - (mlr\_resamplings\_repeated\_cv), [199](#)
- Resamplings, [187](#), [189](#), [191](#), [193](#), [195](#), [196](#), [198](#), [200](#), [202](#), [261](#)
- ResamplingSubsampling
  - (mlr\_resamplings\_subsampling), [201](#)
- ResultData, [18](#), [254](#)
- rpart::rpart(), [89](#), [95](#)
- rsmp (mlr\_sugar), [203](#)
- rsmp(), [185](#), [186](#), [188](#), [190](#), [192](#), [194](#), [195](#), [197](#), [199](#), [201](#)
- rsmps (mlr\_sugar), [203](#)
- rsmps(), [185](#)
- sd(), [93](#)
- set\_threads, [262](#)
- set\_validate (mlr\_sugar), [203](#)
- setdiff(), [266](#)
- split(), [190](#)
- stats::AIC(), [99](#)
- stats::BIC(), [100](#)
- stats::cor(), [158](#), [179](#)
- stats::logLik(), [53](#)
- stats::model.matrix(), [269](#)
- stats::predict(), [239](#)
- summary(), [264](#)
- Task, [15](#), [17](#), [19](#), [26](#), [27](#), [30](#), [32](#), [34](#), [35](#), [37](#), [39](#), [40](#), [50](#), [54](#), [59](#), [60](#), [69](#), [72](#), [73](#), [75](#), [78](#), [80](#), [81](#), [181](#), [188](#), [190](#), [203](#), [205](#), [207–220](#), [238](#), [239](#), [241](#), [242](#), [247](#), [250](#), [251](#), [253](#), [255](#), [257–259](#), [261](#), [263](#), [264](#), [267](#), [275](#), [277](#), [279–281](#)
- TaskClassif, [20](#), [22](#), [23](#), [205](#), [207](#), [208](#), [210–212](#), [214–220](#), [245](#), [263](#), [268](#), [274](#), [274](#), [281](#)
- TaskGenerator, [51](#), [203](#), [220](#), [221](#), [223–236](#), [277](#)
- TaskGenerator2DNormals
  - (mlr\_task\_generators\_2dnormals), [221](#)
- TaskGeneratorCassini
  - (mlr\_task\_generators\_cassini), [223](#)
- TaskGeneratorCircle
  - (mlr\_task\_generators\_circle), [225](#)
- TaskGeneratorFriedman1
  - (mlr\_task\_generators\_friedman1), [227](#)
- TaskGeneratorMoons
  - (mlr\_task\_generators\_moons), [228](#)
- TaskGenerators, [223](#), [224](#), [226](#), [228](#), [229](#), [231](#), [233](#), [234](#), [236](#), [279](#)
- TaskGeneratorSimplex
  - (mlr\_task\_generators\_simplex), [230](#)
- TaskGeneratorSmiley
  - (mlr\_task\_generators\_smiley), [231](#)
- TaskGeneratorSpirals
  - (mlr\_task\_generators\_spirals), [233](#)
- TaskGeneratorXor
  - (mlr\_task\_generators\_xor), [235](#)
- TaskRegr, [20](#), [23](#), [25](#), [205–208](#), [210–212](#), [214–217](#), [219](#), [220](#), [249](#), [263](#), [268](#), [274](#), [277](#), [280](#)
- Tasks, [207–209](#), [211–217](#), [219](#), [220](#), [274](#), [277](#), [281](#)
- TaskSupervised, [205](#), [207](#), [208](#), [210–212](#), [214–217](#), [219](#), [220](#), [263](#), [274](#), [275](#), [277](#), [280](#), [281](#)
- TaskUnsupervised, [25](#), [205](#), [207](#), [208](#), [210–212](#), [214–217](#), [219](#), [220](#), [263](#), [274](#), [277](#), [281](#)
- tgen (mlr\_sugar), [203](#)

tgen(), [220](#), [221](#), [223](#), [225](#), [227](#), [228](#), [230](#),  
[232](#), [233](#), [235](#)  
tgens (mlr\_sugar), [203](#)  
tgens(), [220](#), [221](#)  
time\_train, [68](#)  
traceback(), [60](#)  
tsk (mlr\_sugar), [203](#)  
tsk(), [205](#), [207](#), [209](#), [210](#), [213–216](#), [218](#), [219](#)  
tsks (mlr\_sugar), [203](#)  
tsks(), [205](#)

union(), [266](#)  
unmarshal\_model (marshaling), [67](#)  
unmarshal\_model(), [32](#), [85](#), [257](#)

xor, [277](#)

zero-one, [71](#)