Package ‘mlr3’

April 24, 2024

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

**Version**  0.19.0

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

**License**  LGPL-3


**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 (>= 0.4.1), mlr3misc (>= 0.15.0), parallelly, palmerpenguins, paradox (>= 0.10.0), RhpcBLASctl, uuid

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

**Encoding**  UTF-8

**Config/testthat/edition**  3

**Config/testthat/parallel**  false

**NeedsCompilation**  no

**RoxygenNote**  7.3.1

**Collate**  'mlr_reflections.R' 'BenchmarkResult.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'
R topics documented:

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

Maintainer Michel Lang <michellang@gmail.com>
Repository CRAN
Date/Publication 2024-04-24 16:00:02 UTC

R topics documented:

mlr3-package ................................................................. 6
as_benchmark_result .......................................................... 8
as_data_backend.Matrix ...................................................... 9
as_learner ................................................................. 10
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 ............................................................ 17
as_task ................................................................. 19
as_task_classif ........................................................... 19
as_task_regr .............................................................. 22
as_task_unsupervised ....................................................... 24
benchmark .............................................................. 25
BenchmarkResult ............................................................ 28
benchmark_grid ............................................................ 35
convert_task ............................................................. 37
DataBackend .............................................................. 38
DataBackendDataTable .................................................... 40
DataBackendMatrix ........................................................ 43
default_measures .......................................................... 45
HotstartStack ............................................................. 46
install_pkgs ............................................................... 48
Learner ................................................................. 50
LearnerClassif ............................................................ 57
LearnerRegr ............................................................... 60
Measure ............................................................... 62
MeasureClassif ............................................................ 67
MeasureRegr .............................................................. 70
MeasureSimilarity .......................................................... 72
mlr_learners .............................................................. 75
mlr_learners_classif.debug ............................................... 76
mlr_learners_classif.featureless ........................................ 79
mlr_learners_classif.rpart ................................................ 82
mlr_learners_regr.debug .................................................. 84
### R topics documented:

<table>
<thead>
<tr>
<th>Package/Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mlr_learners_regr.featureless</td>
<td>86</td>
</tr>
<tr>
<td>mlr_learners_regr.rpart</td>
<td>88</td>
</tr>
<tr>
<td>mlr_measures</td>
<td>90</td>
</tr>
<tr>
<td>mlr_measures_aic</td>
<td>91</td>
</tr>
<tr>
<td>mlr_measures_bic</td>
<td>93</td>
</tr>
<tr>
<td>mlr_measures_classif.acc</td>
<td>94</td>
</tr>
<tr>
<td>mlr_measures_classif.auc</td>
<td>95</td>
</tr>
<tr>
<td>mlr_measures_classif.bacc</td>
<td>97</td>
</tr>
<tr>
<td>mlr_measures_classif.bbrier</td>
<td>98</td>
</tr>
<tr>
<td>mlr_measures_classif.ce</td>
<td>99</td>
</tr>
<tr>
<td>mlr_measures_classif.costs</td>
<td>101</td>
</tr>
<tr>
<td>mlr_measures_classif.dor</td>
<td>103</td>
</tr>
<tr>
<td>mlr_measures_classif.fbeta</td>
<td>104</td>
</tr>
<tr>
<td>mlr_measures_classif.fdr</td>
<td>106</td>
</tr>
<tr>
<td>mlr_measures_classif.fn</td>
<td>107</td>
</tr>
<tr>
<td>mlr_measures_classif.fnr</td>
<td>108</td>
</tr>
<tr>
<td>mlr_measures_classif.fomr</td>
<td>110</td>
</tr>
<tr>
<td>mlr_measures_classif.fp</td>
<td>111</td>
</tr>
<tr>
<td>mlr_measures_classif.fpr</td>
<td>112</td>
</tr>
<tr>
<td>mlr_measures_classif.logloss</td>
<td>114</td>
</tr>
<tr>
<td>mlr_measures_classif.mauc_au1p</td>
<td>115</td>
</tr>
<tr>
<td>mlr_measures_classif.mauc_au1u</td>
<td>117</td>
</tr>
<tr>
<td>mlr_measures_classif.mauc_aunp</td>
<td>118</td>
</tr>
<tr>
<td>mlr_measures_classif.mauc_aunu</td>
<td>120</td>
</tr>
<tr>
<td>mlr_measures_classif.mbrier</td>
<td>121</td>
</tr>
<tr>
<td>mlr_measures_classif.mcc</td>
<td>123</td>
</tr>
<tr>
<td>mlr_measures_classif.npv</td>
<td>124</td>
</tr>
<tr>
<td>mlr_measures_classifppv</td>
<td>126</td>
</tr>
<tr>
<td>mlr_measures_classif.prauc</td>
<td>127</td>
</tr>
<tr>
<td>mlr_measures_classif.precision</td>
<td>128</td>
</tr>
<tr>
<td>mlr_measures_classif.recall</td>
<td>130</td>
</tr>
<tr>
<td>mlr_measures_classif.sensitivity</td>
<td>131</td>
</tr>
<tr>
<td>mlr_measures_classif.specificity</td>
<td>132</td>
</tr>
<tr>
<td>mlr_measures_classif.tn</td>
<td>134</td>
</tr>
<tr>
<td>mlr_measures_classif.tnr</td>
<td>135</td>
</tr>
<tr>
<td>mlr_measures_classif.tp</td>
<td>137</td>
</tr>
<tr>
<td>mlr_measures_classif.tpr</td>
<td>138</td>
</tr>
<tr>
<td>mlr_measures_debug_classif</td>
<td>139</td>
</tr>
<tr>
<td>mlr_measures_elapsed_time</td>
<td>141</td>
</tr>
<tr>
<td>mlr_measures_oob_error</td>
<td>143</td>
</tr>
<tr>
<td>mlr_measures_regr.bias</td>
<td>144</td>
</tr>
<tr>
<td>mlr_measures_regr.ktau</td>
<td>145</td>
</tr>
<tr>
<td>mlr_measures_regr.mae</td>
<td>146</td>
</tr>
<tr>
<td>mlr_measures_regr.mape</td>
<td>147</td>
</tr>
<tr>
<td>mlr_measures_regr.maxae</td>
<td>148</td>
</tr>
<tr>
<td>mlr_measures_regr.medae</td>
<td>150</td>
</tr>
<tr>
<td>mlr_measures_regr.medse</td>
<td>151</td>
</tr>
<tr>
<td>mlr_measures_regr.mse</td>
<td>152</td>
</tr>
</tbody>
</table>
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

- 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
mlr3-package

- 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: Michel Lang <michellang@gmail.com> (ORCID)

Authors:

- 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>
as_benchmark_result

- Florian Pfisterer <pfistererf@googlemail.com> (ORCID)
- Raphael Sonabend <raphaelsonabend@gmail.com> (ORCID)
- Marc Becker <marcbecker@posteo.de> (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]
- Sebastian Fischer <sebf.fischer@gmail.com> (ORCID) [contributor]

References


See Also

Useful links:
- https://mlr3.mlr-org.com
- https://github.com/mlr-org/mlr3

--

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

- `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

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

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

Arguments

- `data` *(data.frame())*
  
  The input data.frame(). Automatically converted to a data.table::data.table().

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

- `dense` *(data.frame())*
  
  Dense data.

- `...` *(any)*
  
  Additional arguments passed to the respective DataBackend method.

- `keep_rownames` *(logical(1)|character(1))*
  
  If TRUE or a single string, keeps the row names of data as a new column. The column is named like the provided string, defaulting to "..rownames" for keep_rownames == TRUE. Note that the created column will be used as a regular feature by the task unless you manually change the column role. Also see data.table::as.data.table().
as_learner

Value

DataBackend.

See Also


• 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

Convert to a Learner

Description

Convert object to a Learner or a list of Learner.

Usage

as_learner(x, ...)

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

as_learners(x, ...)

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

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

Arguments

x (any)
Object to convert.

... (any)
Additional arguments.

clone (logical(1))
If TRUE, ensures that the returned object is not the same as the input x.

discard_state (logical(1)) Whether to discard the state.
**as_measure**

**Value**

Learner.

---

**as_measure**

*Convert to a Measure*

**Description**

Convert object to a **Measure** or a list of **Measure**.

**Usage**

```r
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 `mlr_reflections$default_measures`.

- `clone` *(logical(1))*
  If TRUE, ensures that the returned object is not the same as the input `x`.

**Value**

**Measure**.
as_prediction

Convert to a Prediction

Description

Convert object to a Prediction or a list of Prediction.

Usage

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

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

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

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

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

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

Arguments

x (any)
Object to convert.

check (logical(1))
Perform argument checks and type conversions?

... (any)
Additional arguments.

predict_sets (character())
Prediction sets to operate on, used in aggregate() to extract the matching predict_sets from the ResampleResult. Multiple predict sets are calculated by the respective Learner during resample()/benchmark(). Must be a non-empty subset of \{"train", "test", "holdout"\}. If multiple sets are provided, these are first combined to a single prediction object. Default is "test".

Value

Prediction.
as_prediction_classif  Convert to a Classification Prediction

Description
Convert object to a PredictionClassif.

Usage
as_prediction_classif(x, ...)

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

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

Arguments
x (any)
Object to convert.

... (any)
Additional arguments.

Value
PredictionClassif.

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

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

# convert back to a Prediction
as_prediction_classif(tab)

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

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

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

as_prediction_data

PredictionData

Description

Convert object to a PredictionData or a list of PredictionData.

Usage

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

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

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

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

Arguments

x (any)
Object to convert.

task (Task).

row_ids integer()
Row indices.

check (logical(1))
Perform argument checks and type conversions?

... (any)
Additional arguments.

train_task (Task)
Task used for training the learner.

Value

PredictionData.
as_prediction_regr

Convert to a Regression Prediction

Description

Convert object to a PredictionRegr.

Usage

as_prediction_regr(x, ...)

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

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

Arguments

x (any)
Object to convert.

... (any)
Additional arguments.

Value

PredictionRegr.

Examples

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

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

# convert back to a Prediction
as_prediction_regr(tab)

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

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

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

as_resample_result  Convert to ResampleResult

**Description**

Convert object to a `ResampleResult`.

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

**Usage**

```r
as_resample_result(x, ...)
```

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

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

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

**Arguments**

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

**Value**

- (ResampleResult).
as_resampling  Convert to a Resampling

Description
Convert object to a Resampling or a list of Resampling.

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.

close (logical(1))
If TRUE, ensures that the returned object is not the same as the input x.

as_result_data  Convert to ResultData

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

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

Arguments

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

Value

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

Examples

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

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

rdata = as_result_data(task, learners, resampling, iterations, predictions)
ResampleResult$new(rdata)
```
**as_task**  
*Convert to a Task*

**Description**
Convert object to a Task or a list of Task.

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

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

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

**as_task_classif**  
*Convert to a Classification Task*

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

1. TaskClassif: 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 = deparse(substitute(x)),
  positive = NULL,
  label = NA_character_,
  ...
)

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

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

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

## S3 method for class 'TaskRegr'
as_task_classif(
  x,
  target = NULL,
as_task_classif

    drop_original_target = FALSE,
    drop_levels = TRUE,
    ...
  )

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

Arguments

x (any)
Object to convert.

... (any)
Additional arguments.

clone (logical(1))
If TRUE, ensures that the returned object is not the same as the input x.

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

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

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

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

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

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

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

Value

TaskClassif.

Examples

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

Convert to a Regression Task

Description

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

1. TaskRegr: 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 = deparse(substitute(x)),
  label = NA_character_,
  ...
)

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

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

## S3 method for class 'DataBackend'
as_task_regr(
  x,
  target = NULL,
  id = deparse(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 = deparse(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.
as_task_unsupervised

- **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**

```r
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**

```r
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 = deparse(substitute(x)),
  label = NA_character_,
  ...
)

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

as_tasks_unsupervised(x, ...)

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

**Description**

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

**Usage**

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

**Arguments**

- `design` (data.frame()): Data frame (or `data.table::data.table()`) with three columns: "task", "learner", and "resampling". Each row defines a resampling by providing a `Task`, `Learner` and an instantiated `Resampling` strategy. The helper function `benchmark_grid()` can assist in generating an exhaustive design (see examples) and instantiate the `Resamplings` per `Task`. Additionally, you can set the additional column 'param_values', see `benchmark_grid()`.

---

**Benchmark Multiple Learners on Multiple Tasks**

Runs a benchmark on arbitrary combinations of tasks (`Task`), learners (`Learner`), and resampling strategies (`Resampling`), possibly in parallel.
store_models (logical(1))
Store the fitted model in the resulting object= Set to TRUE if you want to further analyse the models or want to extract information like variable importance.

store_backends (logical(1))
Keep the DataBackend of the Task in the ResampleResult? Set to TRUE if your performance measures require a Task, or to analyse results more conveniently. Set to FALSE to reduce the file size and memory footprint after serialization. The current default is TRUE, but this eventually will be changed in a future release.

encapsulate (character(1))
If not NA, enables encapsulation by setting the field Learner$encapsulate to one of the supported values: "none" (disable encapsulation), "try" (captures errors but output is printed to the console and not logged), "evaluate" (execute via evaluate) and "callr" (start in external session via callr). If NA, encapsulation is not changed, i.e. the settings of the individual learner are active. Additionally, if encapsulation is set to "evaluate" or "callr", the fallback learner is set to the featureless learner if the learner does not already have a fallback configured.

allow_hotstart (logical(1))
Determines if learner(s) are hot started with trained models in $hotstart_stack. See also HotstartStack.

clon (character(1))
Select the input objects to be cloned before proceeding by providing a set with possible values "task", "learner" and "resampling" for Task, Learner and Resampling, respectively. Per default, all input objects are cloned.

unmarshal Learner
Whether to unmarshal learners that were marshaled during the execution. Setting this to FALSE does not guarantee that the learners are marshaled. For example, with sequential execution and no encapsulation, marshaling is not necessary. If you want to ensure that all learners are in marshaled form, you need to call $marshal() on the result object.

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 sent to an apply function from future::apply in a single batch. To select a parallel backend, use future::plan().

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 database, 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

- 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 = rsmtp("cv", folds = 3)

design = benchmark_grid(tasks, learners, resamplings)
BenchmarkResult

**Description**

This is the result container object returned by `benchmark()`. A *BenchmarkResult* consists of the data of multiple *ResampleResults*. 

```r
print(design)
set.seed(123)
bmr = benchmark(design)

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

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

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

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

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

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

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

## Get the training set of the 2nd iteration of the featureless learner on penguins
rr = bmr$aggregate()[learner_id == "classif.featureless"]$resample_result[[1]]
rr$resampling$train_set(2)
```
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$aggregate()`
- `BenchmarkResult$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 $\text{clone()}$ 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 \text{marshal\_model}().

Method unmarshal(): Unmarshals all stored models.

Usage:

BenchmarkResult$unmarshal(...)  

Arguments:

... (any)

Additional arguments passed to \text{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 \text{Measure}.

Usage:

BenchmarkResult$score(  
  measures = NULL,  
  ids = TRUE,  
  conditions = FALSE,  
  predict\_sets = "test"  
)

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.

predict\_sets (character())

Prediction sets to operate on, used in \text{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", "holdout"`. If multiple sets are provided, these are first combined to a single prediction object. Default is "test".

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

**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:**

```r
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 keeps 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 $\text{clone()}$ 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
- Package mlr3viz for some generic visualizations.
- mlr3benchmark for post-hoc analysis of benchmark results.

Other benchmark: benchmark(), benchmark_grid()

Examples
```r
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 = rsm("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
```
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).

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)
param_values (list())
If you want to try many parameter settings for learners, you can pass them through the design which is optimized to be faster than creating learners for each setting.

A list of lists of named lists, from outer to inner:
1. One list element for each Learner.
2. One list element for each hyperparameter configuration to try.
3. Named list of hyperparameter settings to set in the Learner, possibly overwriting already set set hyperparameters in the Learner.

paired (logical(1))
Set this to TRUE if the resamplings are instantiated on the tasks, i.e., the tasks and resamplings are paired. You need to provide the same number of tasks and instantiated resamplings.

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

See Also
- Package mlr3viz for some generic visualizations.
- mlr3benchmark for post-hoc analysis of benchmark results.

Other benchmark: BenchmarkResult, benchmark()

Examples

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

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

# paired
learner = lrn("classif.rpart")
task1 = tsk("penguins")
task2 = tsk("german_credit")
res1 = rsmp("holdout")
res2 = rsmp("holdout")
convert_task

```r
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**

*Convert a Task from One Type to Another*

---

**Description**

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

**Usage**

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

**Arguments**

- **intask** *(Task)*
  A Task to be converted.

- **target** *(character(1))*
  New target to be set, must be a column in the intask data. If NULL, no new target is set and task is converted as-is.

- **new_type** *(character(1))*
  The new task type. Must be in `mlr_reflections$task_types`. If NULL (default), a new task with the same task_type is created.
DataBackend

\begin{itemize}
  \item \texttt{drop_original_target} (\texttt{logical(1)})
    If \texttt{FALSE} (default), the original target is added as a feature. Otherwise the original target is dropped.
  \item \texttt{drop_levels} (\texttt{logical(1)})
    If \texttt{TRUE} (default), unused levels of the new target variable are dropped.
\end{itemize}

\textbf{Value}

- \texttt{Task} of requested type.

\begin{tabular}{ll}
\hline
\textbf{DataBackend} & \textbf{DataBackend} \\
\hline
\end{tabular}

\textbf{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 \texttt{Task}.

This package comes with two implementations for backends:

- \texttt{DataBackendDataTable} which stores the data as \texttt{data.table::data.table()}.
- \texttt{DataBackendMatrix} which stores the data as sparse \texttt{Matrix::sparseMatrix()}.

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

\textbf{Details}

The required set of fields and methods to implement a custom DataBackend is listed in the respective sections (see \texttt{DataBackendDataTable} or \texttt{DataBackendMatrix} for exemplary implementations of the interface).

\textbf{Public fields}

- \texttt{primary_key} (\texttt{character(1)})
  Column name of the primary key column of positive and unique integer row ids.
- \texttt{data_formats} (\texttt{character()})
  Set of supported formats, e.g. "\texttt{data.table}" or "\texttt{Matrix}".
Active bindings

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

col_hashes (named character)
   Hash (unique identifier) for all columns except the primary_key: A character vector, named by
   the columns that each element refers to.
   Columns of different 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 = "data.table")

Arguments:

data (any)
   The format of the input data depends on the specialization. E.g., DataBackendDataTable ex-
   pects 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 in-
   teger values, referenced here by primary_key. The use of this variable may differ between
   backends.

data_formats (character())
   Set of supported data formats which can be processed during $train() and $predict(),
   e.g. "data.table".

Method format(): Helper for print outputs.

Usage:
DataBackend$format(...)  

Arguments:
   ... (ignored).

Method print(): Printer.

Usage:
DataBackend$print()
See Also

- Package mlr3db to interface out-of-memory data, e.g. SQL servers or duckdb.

Other DataBackend: DataBackendDataTable, DataBackendMatrix, as_data_backend.Matrix()

Examples

```r
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 = "data.table")
```

*Arguments:*

- `rows` ([`positive integer()`](#))
  Vector or row indices.
- `cols` (`character()`)  
  Vector of column names.
- `data_format` ([`character(1)`](#))
  Desired data format, e.g. "data.table" or "Matrix".

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

Arguments:
- `rows` (positive integer())
  - Vector of row indices.
- `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:
```r
DataBackendDataTable$missings(rows, cols)
```

Arguments:
- `rows` (positive integer())
  - Vector of row indices.
- `cols` (character())
  - Vector of column names.

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

See Also
- Package mlr3db to interface out-of-memory data, e.g. SQL servers or duckdb.

Other DataBackend: `DataBackend, DataBackendMatrix, as_data_backend.Matrix()`

Examples
```r
data = as.data.table(palmerpenguins::penguins)
data$id = seq_len(nrow(palmerpenguins::penguins))
b = DataBackendDataTable$new(data = data, primary_key = "id")
print(b)
b$head()
b$data(rows = 100:101, cols = "species")

b$nrow
head(b$rownames)

b$ncol
b$colnames
```
# alternative construction
as_data_backend(palmerpenguins::penguins)

---

## DataBackendMatrix

### 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

`mlr3::DataBackend` \rightarrow 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 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:
DataBackendMatrix$data(rows, cols, data_format = "data.table")

Arguments:
rows (positive integer())
    Vector or row indices.
cols (character())
    Vector of column names.
data_format (character(1))
    Desired data format, e.g. "data.table" or "Matrix".

Method head(): Retrieve the first \( n \) rows.

Usage:
DataBackendMatrix$head(n = 6L)

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

Returns: \texttt{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 \texttt{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.
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.
- cols (character()): Vector of column names.

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

**See Also**
- Package mlr3db to interface out-of-memory data, e.g. SQL servers or duckdb.

Other DataBackend: DataBackend, DataBackendDataTable, as_data_backend.Matrix()

**Examples**

```r
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$colnames, data_format = "Matrix")
b$data(1:3, b$colnames, data_format = "data.table")
```

**Description**

Gets the default measures using the information in mlr_reflections$default_measures:
- "classif.ce" for classification ("classif").
- "regr.mse" for regression ("regr").
- Add-on package may register additional default measures for their own task types.

**Usage**

default_measures(task_type)
Arguments

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

Value

- list of Measure.

Examples

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

---

**HotstartStack**

*Stack for Hot Start Learners*

**Description**

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

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

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

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

**Public fields**

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

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

Public methods:

• HotstartStack$new()
• HotstartStack$add()
• HotstartStack$start_cost()
• HotstartStack$format()
• HotstartStack$print()
• HotstartStack$clone()

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

Usage:
HotstartStack$new(learners = NULL, hotstart_threshold = NULL)

Arguments:
learners (List of Learners)
Learners are added to the hotstart stack. If NULL (default), empty stack is created.
hotstart_threshold (named numeric(1))
Threshold for storing learners in the stack.

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

Usage:
HotstartStack$add(learners)

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

Returns: self (invisibly).

Method start_cost(): Calculates the cost for each learner of the stack to hot start the target learner.
The following cost values can be returned:

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

Usage:
HotstartStack$start_cost(learner, task_hash)

Arguments:
learner Learner
Target learner.
task_hash Task
Hash of the task on which the target learner is trained.

Method format(): Helper for print outputs.

Usage:
HotstartStack$format(...)  
**Arguments:**  
... (ignored).

**Method** `print()`: Printer.  
**Usage:**  
HotstartStack$print(...)  
**Arguments:**  
... (ignored).

**Method** `clone()`: The objects of this class are cloneable with this method.  
**Usage:**  
HotstartStack$clone(deep = FALSE)  
**Arguments:**  
deep   Whether to make a deep clone.

### Examples

```r
# 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)
```

### Description

`install_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.
install_pkgs

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 $\cdot\text{train()}/\cdot\text{predict()}$.
- 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 $\cdot\text{model}$, available after calling $\cdot\text{train()}$.

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.
Setting Hyperparameters

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

```r
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()`:

```r
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":

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

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`.
- **data_formats** (character())
  - Supported data format, e.g. "data.table" or "Matrix".
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", "holdout").
   Each set yields a separate Prediction object. Those can be combined via getters in Resam-
   pleResult/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.

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

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

Active bindings

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". 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) or feature selection (feature names).
Learner

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.

encapsulate (named character())
Controls how to execute the code in internal train and predict methods. Must be a named character vector with names "train" and "predict". Possible values are "none", "try", "evaluate" (requires package evaluate) and "callr" (requires package callr). See mlr3misc::encapsulate() for more details.

fallback (Learner)
Learner which is fitted to impute predictions in case that either the model fitting or the prediction of the top learner is not successful. Requires encapsulation, otherwise errors are not caught and the execution is terminated before the fallback learner kicks in. If you have not set encapsulation manually before, setting the fallback learner automatically activates encapsulation using the evaluate package. 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

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$base_learner()
- Learner$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 = "data.table",
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).
    • "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())
    Set of supported data formats which can be processed during $train() and $predict(),
    e.g. "data.table".

packages (character())
    Set of required packages. A warning is signaled by the constructor if at least one of the pack-
    ages 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 refer-
    enced help package can be opened via method $help().

Method format(): Helper for print outputs.

Usage:
Learner

Learner$format(...)

Arguments:
... (ignored).

Method print(): Printer.

Usage:
Learner$print(...)

Arguments:
... (ignored).

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

Usage:
Learner$help()

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

Usage:
Learner$train(task, row_ids = NULL)

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

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

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 keeps 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 `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

- 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.


---

**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](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 = "data.table",
  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 `$selected_features()` extractor function (see section on optional extractors in Learner).
  - "oob_error": The learner supports extraction of estimated out of bag error, i.e. comes with a `oob_error()` extractor function (see section on optional extractors in Learner).
  - "marshal": To save learners with this property, you need to call `$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())
  Set of supported data formats which can be processed during `$train()` and `$predict()`, e.g. "data.table".

- **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:
deept Whether to make a deep clone.

See Also

- 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.

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:
ln = lrn("classif.rpart")
print(ln)

# train the learner:
task = tsk("penguins")
ln$train(task, 1:200)

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

Methods

**Public methods:**

- LearnerRegr$new()
- LearnerRegr$clone()

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

*Usage:*

```r
LearnerRegr$new(
  id,
  param_set = ps(),
  predict_types = "response",
  feature_types = character(),
  properties = character(),
  data_formats = "data.table",
  packages = character(),
  label = NA_character_,
  man = NA_character_
)
```

*Arguments:*
LearnerRegr

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 \texttt{mlr_reflections$learner_predict_types}.

feature_types (character())
   Feature types the learner operates on. Must be a subset of \texttt{mlr_reflections$task_feature_types}.

properties (character())
   Set of properties of the Learner. Must be a subset of \texttt{mlr_reflections$learner_properties}.
   The following properties are currently standardized and understood by learners in \texttt{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).
   • "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())
   Set of supported data formats which can be processed during \$train() and \$predict(), e.g. "data.table".

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 \texttt{[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 \texttt{mlr3book}: \url{https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-learners}
   • Package \texttt{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.


Examples

```r
# 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")
```

<table>
<thead>
<tr>
<th>Measure</th>
<th>Measure Class</th>
</tr>
</thead>
</table>

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`.
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.

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

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", "holdout"\}. 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.

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

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,  
  properties = character(),  
  predict_type = "response",  
  predict_sets = "test",  
  task_properties = character(),  
  packages = character(),  
  label = NA_character_,

Measure

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.
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).
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", "holdout"}.
  If multiple sets are provided, these are first combined to a single prediction object. Default is "test".
Measure

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 format(): Helper for print outputs.
  Usage:
  Measure$format(...)
  Arguments:
  ... (ignored).

Method print(): Printer.
  Usage:
  Measure$print(...)
  Arguments:
  ... (ignored).

Method help(): Opens the corresponding help page referenced by field $man.
  Usage:
  Measure$help()

Method score(): Takes a Prediction (or a list of Prediction objects named with valid predict_sets) and calculates a numeric score. If the measure if flagged with the properties "requires_task", "requires_learner", "requires_model" or "requires_train_set", you must additionally pass the respective Task, the (trained) Learner or the training set indices. This is handled internally during resample()/benchmark().
  Usage:
  Measure$score(prediction, task = NULL, learner = NULL, train_set = NULL)
  Arguments:
  prediction (Prediction | named list of Prediction).
  task (Task).
  learner (Learner).
  train_set (integer()).
  Returns: numeric(1).

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

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


• 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_oob_error, mlr_measures_selected_features

MeasureClassif Classification Measure

Description

This measure specializes Measure for classification problems:

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

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

Super class

mlr3::Measure -> MeasureClassif
Methods

Public methods:

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

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

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

**Arguments:**

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

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

- `range` (numeric(2))  
  Feasible range for this measure as c(lower_bound, upper_bound). Both bounds may be infinite.

- `minimize` (logical(1))  
  Set to TRUE if good predictions correspond to small values, and to FALSE if good predictions correspond to large values. If set to NA (default), tuning this measure is not possible.

- `average` (character(1))  
  How to average multiple Predictions from a ResampleResult. The default, "macro", calculates the individual performances scores for each Prediction and then uses the function defined in $aggregator to average them to a single number. 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).

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", "holdout"}. 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:
deepl Whether to make a deep clone.

See Also
• 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_oob_error, mlr_measures_selected_features`

<table>
<thead>
<tr>
<th>MeasureRegr</th>
<th>Regression Measure</th>
</tr>
</thead>
</table>

**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:

```r
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).
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", "holdout"}. 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().
MeasureSimilarity

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:
depth Whether to make a deep clone.

See Also
- Chapter in the mlrbook: https://mlrbook.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_oob_error, mlr_measures_selected_features

MeasureSimilarity Similarity Measure

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

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

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

Super class
mlr3::Measure -> MeasureSimilarity
Methods

Public methods:

• `MeasureSimilarity$new()`
• `MeasureSimilarity$clone()`

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

Usage:

```r
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).

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", "holdout"}. 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:
deepp Whether to make a deep clone.

See Also
• 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.
mlr_learners

-- **mlr3cluster** for unsupervised clustering.

Other Measure: \texttt{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_oob_error, mlr_measures_selected_features}

Examples

```r
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 \texttt{mlr3misc::Dictionary} storing objects of class \texttt{Learner}. Each learner has an associated help page, see \texttt{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 \texttt{mlr3learners} package. More learners are connected via \url{https://github.com/mlr-org/mlr3extralearners}.

For a more convenient way to retrieve and construct learners, see \texttt{lrn()/lrns()}. 

**Format**

\texttt{R6::R6Class} object inheriting from \texttt{mlr3misc::Dictionary}.

**Methods**

See \texttt{mlr3misc::Dictionary}.

**S3 methods**

- \texttt{as.data.table(dict, ..., objects = FALSE)}

  \texttt{mlr3misc::Dictionary -> data.table::data.table()}

  Returns a \texttt{data.table::data.table()} with fields "key", "label", "task_type", "feature_types", "packages", "properties", and "predict_types" as columns. If \texttt{objects} is set to \texttt{TRUE}, the constructed objects are returned in the list column named \texttt{object}. 

See Also

Sugar functions: lrn(), lrns()

Extension Packages: mlr3learners

Other Dictionary: mlr_measures, mlr_resamplings, mlr_task_generators, mlr_tasks


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 raise 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 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 provoke a segfault during predict.
- **segfault_train**: Probability to provoke 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.

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()`:

```r
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**

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Levels</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_predict</td>
<td>numeric</td>
<td>0</td>
<td></td>
<td>[0, 1]</td>
</tr>
<tr>
<td>error_train</td>
<td>numeric</td>
<td>0</td>
<td></td>
<td>[0, 1]</td>
</tr>
<tr>
<td>message_predict</td>
<td>numeric</td>
<td>0</td>
<td></td>
<td>[0, 1]</td>
</tr>
<tr>
<td>message_train</td>
<td>numeric</td>
<td>0</td>
<td></td>
<td>[0, 1]</td>
</tr>
<tr>
<td>predict_missing</td>
<td>numeric</td>
<td>0</td>
<td></td>
<td>[0, 1]</td>
</tr>
<tr>
<td>predict_missing_type</td>
<td>character</td>
<td>na</td>
<td>na, omit</td>
<td>-</td>
</tr>
<tr>
<td>save_tasks</td>
<td>logical</td>
<td>FALSE</td>
<td></td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>segfault_predict</td>
<td>numeric</td>
<td>0</td>
<td></td>
<td>[0, 1]</td>
</tr>
<tr>
<td>segfault_train</td>
<td>numeric</td>
<td>0</td>
<td></td>
<td>[0, 1]</td>
</tr>
<tr>
<td>sleep_train</td>
<td>untyped</td>
<td>-</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>sleep_predict</td>
<td>untyped</td>
<td>-</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>threads</td>
<td>integer</td>
<td>-</td>
<td></td>
<td>[1, ∞)</td>
</tr>
<tr>
<td>warning_predict</td>
<td>numeric</td>
<td>0</td>
<td></td>
<td>[0, 1]</td>
</tr>
<tr>
<td>warning_train</td>
<td>numeric</td>
<td>0</td>
<td></td>
<td>[0, 1]</td>
</tr>
<tr>
<td>x</td>
<td>numeric</td>
<td>-</td>
<td></td>
<td>[0, 1]</td>
</tr>
<tr>
<td>iter</td>
<td>integer</td>
<td>1</td>
<td></td>
<td>[1, ∞)</td>
</tr>
<tr>
<td>count_marshaling</td>
<td>logical</td>
<td>FALSE</td>
<td></td>
<td>TRUE, FALSE</td>
</tr>
</tbody>
</table>

```
Super classes

\texttt{mlr3::Learner} -\texttt{mlr3::LearnerClassif} -\texttt{LearnerClassifDebug}

Active bindings

marshaled \texttt{(logical(1))}
   Whether the learner has been marshaled.

Methods

Public methods:

- \texttt{LearnerClassifDebug\$new()}
- \texttt{LearnerClassifDebug\$marshal()}
- \texttt{LearnerClassifDebug\$unmarshal()}
- \texttt{LearnerClassifDebug\$clone()}

Method \texttt{new()}: Creates a new instance of this \texttt{R6} class.
   \texttt{Usage:}
   \texttt{LearnerClassifDebug\$new()}

Method \texttt{marshal()}: Marshal the learner’s model.
   \texttt{Usage:}
   \texttt{LearnerClassifDebug\$marshal(...)

   Arguments:
   \texttt{... (any)}
   Additional arguments passed to \texttt{marshal\_model()}.}

Method \texttt{unmarshal()}: Unmarshal the learner’s model.
   \texttt{Usage:}
   \texttt{LearnerClassifDebug\$unmarshal(...)

   Arguments:
   \texttt{... (any)}
   Additional arguments passed to \texttt{unmarshal\_model()}.}

Method \texttt{clone()}: The objects of this class are cloneable with this method.
   \texttt{Usage:}
   \texttt{LearnerClassifDebug\$clone(deep = FALSE)

   Arguments:
   \texttt{deep} Whether to make a deep clone.
See Also

- 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

```r
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)
```

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()`:

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

**Meta Information**

- Task type: “classif”
- Predict Types: “response”, “prob”
- Required Packages: `mlr`

**Parameters**

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>character</td>
<td>mode</td>
<td>mode, sample, weighted.sample</td>
</tr>
</tbody>
</table>

**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:*

```r
LearnerClassifFeatureless$new()
```

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

*Usage:*

```r```
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:*

depth Whether to make a deep clone.

### See Also


- 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
Description

A LearnerClassif for a classification tree implemented in \texttt{rpart::rpart()} in package \texttt{rpart}.

Initial parameter values

- Parameter \texttt{xval} is initialized to 0 in order to save some computation time.

Custom mlr3 parameters

- Parameter \texttt{model} has been renamed to \texttt{keep_model}.

Dictionary

This Learner can be instantiated via the dictionary \texttt{mlr_learners} or with the associated sugar function \texttt{lrn()}:

\begin{verbatim}
mlr_learners$get("classif.rpart")
lrn("classif.rpart")
\end{verbatim}

Meta Information

- Task type: “classif”
- Predict Types: “response”, “prob”
- Feature Types: “logical”, “integer”, “numeric”, “factor”, “ordered”
- Required Packages: \texttt{mlr3, rpart}

Parameters

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Levels</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp</td>
<td>numeric</td>
<td>0.01</td>
<td>0, 1</td>
<td></td>
</tr>
<tr>
<td>keep_model</td>
<td>logical</td>
<td>FALSE</td>
<td>TRUE, FALSE</td>
<td>-</td>
</tr>
<tr>
<td>maxcompete</td>
<td>integer</td>
<td>4</td>
<td>0, 30</td>
<td></td>
</tr>
<tr>
<td>maxdepth</td>
<td>integer</td>
<td>30</td>
<td>0, 30</td>
<td></td>
</tr>
<tr>
<td>maxsurrogate</td>
<td>integer</td>
<td>5</td>
<td>0, 30</td>
<td></td>
</tr>
<tr>
<td>minbucket</td>
<td>integer</td>
<td>-</td>
<td>1, 30</td>
<td></td>
</tr>
<tr>
<td>msplit</td>
<td>integer</td>
<td>20</td>
<td>0, 1</td>
<td></td>
</tr>
<tr>
<td>surrogatestyle</td>
<td>integer</td>
<td>0</td>
<td>0, 1</td>
<td></td>
</tr>
<tr>
<td>usesurrogate</td>
<td>integer</td>
<td>2</td>
<td>0, 2</td>
<td></td>
</tr>
<tr>
<td>xval</td>
<td>integer</td>
<td>10</td>
<td>0, 30</td>
<td></td>
</tr>
</tbody>
</table>
Super classes

\texttt{mlr3::Learner} -> \texttt{mlr3::LearnerClassif} -> \texttt{LearnerClassifRpart}

Methods

Public methods:

- \texttt{LearnerClassifRpart$new()}
- \texttt{LearnerClassifRpart$importance()}
- \texttt{LearnerClassifRpart$selected_features()}
- \texttt{LearnerClassifRpart$clone()}

Method \texttt{new()}: Creates a new instance of this R6 class.

\textit{Usage}:
\texttt{LearnerClassifRpart$new()}

Method \texttt{importance()}: The importance scores are extracted from the model slot \texttt{variable.importance}.

\textit{Usage}:
\texttt{LearnerClassifRpart$importance()}

\textit{Returns}: Named \texttt{numeric()}. 

Method \texttt{selected_features()}: Selected features are extracted from the model slot \texttt{frame$var}.

\textit{Usage}:
\texttt{LearnerClassifRpart$selected_features()}

\textit{Returns}: \texttt{character()}. 

Method \texttt{clone()}: The objects of this class are cloneable with this method.

\textit{Usage}:
\texttt{LearnerClassifRpart$clone(deep = FALSE)}

\textit{Arguments}:
\texttt{deep} Whether to make a deep clone.

References


See Also

- Chapter in the \texttt{mlr3book}: \url{https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-learners}
- Package \texttt{mlr3learners} for a solid collection of essential learners.
- Package \texttt{mlr3extralearners} for more learners.
mlr_learners_regr.debug

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 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():

```r
mlr_learners$get("regr.debug")
lrn("regr.debug")
```

Meta Information

- Task type: “regr”
- Predict Types: “response”, “se”
- Feature Types: “logical”, “integer”, “numeric”, “character”, “factor”, “ordered”
- Required Packages: mlr3
### Parameters

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Levels</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>predict_missing</td>
<td>numeric</td>
<td>0</td>
<td>[0, 1]</td>
<td></td>
</tr>
<tr>
<td>predict_missing_type</td>
<td>character</td>
<td>na</td>
<td>na, omit</td>
<td>-</td>
</tr>
<tr>
<td>save_tasks</td>
<td>logical</td>
<td>FALSE</td>
<td>TRUE, FALSE</td>
<td>-</td>
</tr>
<tr>
<td>threads</td>
<td>integer</td>
<td>-</td>
<td>[1, ∞)</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>numeric</td>
<td>-</td>
<td>[0, 1]</td>
<td>-</td>
</tr>
</tbody>
</table>

### Super classes

```
mlr3::Learner -> mlr3::LearnerRegr -> LearnerRegrDebug
```

### Methods

#### Public methods:

- `LearnerRegrDebug$new()`
- `LearnerRegrDebug$clone()`

#### Method `new()`:

Creates a new instance of this R6 class.

**Usage:**

```
LearnerRegrDebug$new()
```

#### 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

- Package **mlr3learners** for a solid collection of essential learners.
- Package **mlr3extralearners** for more learners.
- Dictionary of Learners: [mlr_learners](https://mlr3book.mlr-org.com/chapters/chapter2/data_and_basic_modeling.html#sec-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

```r
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()**:  

```r
dlrs$get("regr.featureless")
lrn("regr.featureless")
```

### Meta Information

- **Task type:** "regr"
- **Predict Types:** "response", "se"
- **Feature Types:** "logical", "integer", "numeric", "character", "factor", "ordered", "POSIXct"
- **Required Packages:** mlr3, 'stats'

### Parameters

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>robust</td>
<td>logical</td>
<td>TRUE</td>
<td>TRUE, FALSE</td>
</tr>
</tbody>
</table>
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:*

```r
LearnerRegrFeatureless$new()
```

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

*Usage:*

```r
LearnerRegrFeatureless$importance()
```

*Returns:* Named numeric().

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

*Usage:*

```r
LearnerRegrFeatureless$selected_features()
```

*Returns:* character(0).

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

*Usage:*

```r
LearnerRegrFeatureless$clone(deep = FALSE)
```

*Arguments:*

depth Whether to make a deep clone.

See Also

- 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.
mlr_learners_regr.rpart

Regression Tree Learner

Description

A LearnerRegr for a regression tree implemented in \texttt{rpart::rpart()} in package \texttt{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 \texttt{mlr_learners} or with the associated sugar function \texttt{lrn()}:

```r
classif$get("regr.rpart")
lrn("regr.rpart")
```

Meta Information

- Task type: “regr”
- Predict Types: “response”
- Feature Types: “logical”, “integer”, “numeric”, “factor”, “ordered”
- Required Packages: \texttt{mlr3, rpart}
Parameters

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Levels</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp</td>
<td>numeric</td>
<td>0.01</td>
<td>TRUE, FALSE</td>
<td>[0, 1]</td>
</tr>
<tr>
<td>keep_model</td>
<td>logical</td>
<td>FALSE</td>
<td>TRUE, FALSE</td>
<td>-</td>
</tr>
<tr>
<td>maxcompete</td>
<td>integer</td>
<td>4</td>
<td></td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>maxdepth</td>
<td>integer</td>
<td>30</td>
<td></td>
<td>(1, 30]</td>
</tr>
<tr>
<td>maxsurrogate</td>
<td>integer</td>
<td>5</td>
<td></td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>minbucket</td>
<td>integer</td>
<td>-</td>
<td></td>
<td>(1, ∞)</td>
</tr>
<tr>
<td>msplit</td>
<td>integer</td>
<td>20</td>
<td></td>
<td>(1, ∞)</td>
</tr>
<tr>
<td>surrogatestyle</td>
<td>integer</td>
<td>0</td>
<td></td>
<td>(0, 1]</td>
</tr>
<tr>
<td>usesurrogate</td>
<td>integer</td>
<td>2</td>
<td></td>
<td>(0, 2]</td>
</tr>
<tr>
<td>xval</td>
<td>integer</td>
<td>10</td>
<td></td>
<td>(0, ∞)</td>
</tr>
</tbody>
</table>

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


See Also

- Package mlr3learners for a solid collection of essential learners.
- Package mlr3xtralearners 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

---

**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_taskGenerators`, `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_oob_error`, `mlr_measures_selected_features`

Examples

```r
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()`:

```r
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

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>integer</td>
<td>-</td>
<td>[0, ∞)</td>
</tr>
</tbody>
</table>

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:
deepto Whether to make a deep clone.

See Also


• 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_oob_error, 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()`:

```r
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:

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

Usage:
`MeasureBIC$clone(deep = FALSE)`

Arguments:
deepl Whether to make a deep clone.

See Also
- 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_oob_error`, `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

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

Dictionary

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

`mlr_measures$get("classif.acc")`

`msr("classif.acc")`

Parameters

Empty ParamSet
**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.

**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.


Dictionary

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

```r
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.

**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, the sample weights $w$ are normalized per class:

$$
\hat{w}_i = \frac{w_i}{\sum_j 1(y_j = y_i)w_i}.
$$

The balanced accuracy is calculated as

$$
\frac{1}{\sum_i w_i} \sum_i 1(r_i = t_i)\hat{w}_i.
$$

**Dictionary**

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

```r
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`. 
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, \( I_i \) is 1 if observation \( i \) 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 \texttt{mbrier()}) divided by 2.

Dictionary

This \textit{Measure} can be instantiated via the dictionary \texttt{mlr_measures} or with the associated sugar function \texttt{msr()}:

\begin{verbatim}
mlr_measures$get("classif.bbrier")
msr("classif.bbrier")
\end{verbatim}
**mlr_measures_classif.ce**

**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.


---

**Description**

Measure to compare true observed labels with predicted labels in multiclass classification tasks.
The Classification Error is defined as
\[ \frac{1}{n} \sum_{i=1}^{n} w_i (t_i \neq r_i). \]

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

```r
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 multiclass classification measures: mlr_measures_classif.acc, mlr_measures_classif.bacc, mlr_measures_classif.brier, mlr_measures_classif.costs, mlr_measures_classif.logloss, mlr_measures_classif.mauc_au1p,
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():

```r
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

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>normalize</td>
<td>logical</td>
<td></td>
<td>TRUE, FALSE</td>
</tr>
</tbody>
</table>

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:
dee p Whether to make a deep clone.

See Also

• 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_auc, mlr_measures_bic, mlr_measures_debug_classif, mlr_measures_elapsed_time, mlr_measures_oob_error, mlr_measures_selected_features


Examples

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

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

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

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

# fit models and evaluate with the cost measure
learner = lrn("classif.rpart")
rr = resample(task, learner, rsmp("cv", folds = 3))
rr$aggregate(m)
```

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

\[
\frac{TP}{FP} / \frac{FN}{TN}.
\]

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():

```r
mlr_measures$get("classif.dor")
msr("classif.dor")
```

Parameters

Empty ParamSet
mlr_measures_classif.fbeta

**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.


---

**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 \texttt{precision()} and $R$ as \texttt{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 \texttt{precision} or \texttt{recall} is undefined, i.e. $TP + FP = 0$ or $TP + FN = 0$.

Dictionary

This Measure can be instantiated via the dictionary \texttt{mlr_measures} or with the associated sugar function \texttt{msr()}:

\begin{verbatim}
mlr_measures$\texttt{get("classif.fbeta")}
msr("classif.fbeta")
\end{verbatim}

Parameters


Meta Information

- Type: "binary"
- Range: $[0, 1]$
- Minimize: \texttt{FALSE}
- Required prediction: \texttt{response}

Note

The score function calls \texttt{mlr3measures::fbeta()} from package \texttt{mlr3measures}.

If the measure is undefined for the input, \texttt{NaN} is returned. This can be customized by setting the field \texttt{na_value}.

See Also

Dictionary of Measures: \texttt{mlr_measures}
as.\texttt{data.table(mlr_measures)} for a complete table of all (also dynamically created) Measure implementations.

**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()`:

```r
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.


---

**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 "false alarm".

**Dictionary**

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

```r
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.


---

**mlr_measures_classif.fnr**

*False Negative Rate*

**Description**

Measure to compare true observed labels with predicted labels in binary classification tasks.
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():

```r
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.


---

**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()`:

```r
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`. 
**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.

**Dictionary**

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

```r
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.brier`
- `mlr_measures_classif.ce`
- `mlr_measures_classif.ccs`
- `mlr_measures_classif.costs`
- `mlr_measures_classif.dor`
- `mlr_measures_classif.fbeta`
- `mlr_measures_classif.fdr`
- `mlr_measures_classif.fn`
- `mlr_measures_classif.fnr`
- `mlr_measures_classif.fomr`
- `mlr_measures_classif.fpr`
- `mlr_measures_classif.logloss`
- `mlr_measures_classif.mauc_au1p`
- `mlr_measures_classif.mauc_au1u`
- `mlr_measures_classif.mauc_aunp`
- `mlr_measures_classif.mauc_aunu`
- `mlr_measures_classif.mbrier`
- `mlr_measures_classif.mcc`
- `mlr_measures_classif.npv`
- `mlr_measures_classif.ppv`
- `mlr_measures_classif.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.brier`
- `mlr_measures_classif.dor`
- `mlr_measures_classif.fbeta`
- `mlr_measures_classif.fdr`
- `mlr_measures_classif.fn`
- `mlr_measures_classif.fnr`
- `mlr_measures_classif.fomr`
- `mlr_measures_classif.fpr`
- `mlr_measures_classif.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`

Description

Measure to compare true observed labels with predicted labels in binary classification tasks.
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 \(FP + TN = 0\).

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

```r
mlr_measures$get("classif.fpr")
msr("classif.fpr")
```

Empty ParamSet

• Type: "binary"
• Range: \([0, 1]\)
• Minimize: \(\text{TRUE}\)
• Required prediction: response

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`.

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.logloss`
- `mlr_measures_classif.mauc_au1p`
- `mlr_measures_classif.mauc_au1u`
- `mlr_measures_classif.mauc_aunp`
- `mlr_measures_classif.mauc_aunu`
- `mlr_measures_classif.mbrier`
- `mlr_measures_classif.mcc`
- `mlr_measures_classif.npv`
- `mlr_measures_classif.ppv`
- `mlr_measures_classif.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.fnr`
- `mlr_measures_classif.fomr`
- `mlr_measures_classif.fp`
- `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`

---

**Description**

Measure to compare true observed labels with predicted probabilities in multiclass classification tasks.

**Details**

The Log 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 \( i \).

**Dictionary**

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

```r
mlr_measures$get("classif.logloss")
msr("classif.logloss")
```

**Parameters**

Empty ParamSet

**Meta Information**

- Type: "classif"
- Range: \([0, \infty)\)
- Minimize: TRUE
- Required prediction: prob

**Note**

The score function calls `mlr3measures::logloss()` from package `mlr3measures`.

If the measure is undefined for the input, NaN is returned. This can be customized by setting the field `na_value`. 
See Also

Dictionary of Measures: mlr_measures

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

Other classification measures:

- mlr_measures_classif.acc
- mlr_measures_classif.auc
- mlr_measures_classif.bacc
- mlr_measures_classif.bbrier
- mlr_measures_classif.ce
- mlr_measures_classif.costs
- mlr_measures_classif.dor
- mlr_measures_classif.fbeta
- mlr_measures_classif.fdr
- mlr_measures_classif.fn
- mlr_measures_classif.fnr
- mlr_measures_classif.fomr
- mlr_measures_classif.fp
- mlr_measures_classif.fpr
- mlr_measures_classif.mauc_au1p
- mlr_measures_classif.mauc_au1u
- mlr_measures_classif.mauc_aunp
- mlr_measures_classif.mauc_aunu
- mlr_measures_classif.mbrier
- mlr_measures_classif.mcc
- mlr_measures_classif.npv
- mlr_measures_classif.ppv
- mlr_measures_classif.prauc
- mlr_measures_classif.precision
- mlr_measures_classif.recall
- mlr_measures_classif.sensitivity
- mlr_measures_classif.specificity
- mlr_measures_classif.tn
- mlr_measures_classif.tnr
- mlr_measures_classif.tp
- mlr_measures_classif.tpr

Other multiclass classification measures:

- mlr_measures_classif.acc
- mlr_measures_classif.bacc
- mlr_measures_classif.ce
- mlr_measures_classif.costs
- mlr_measures_classif.mauc_au1p
- mlr_measures_classif.mauc_au1u
- mlr_measures_classif.mauc_aunp
- mlr_measures_classif.mauc_aunu
- mlr_measures_classif.mbrier

mlr_measures_classif.mauc_au1p

Multiclass AUC Scores

Description

Measure to compare true observed labels with predicted probabilities in multiclass classification tasks.

Details

Multiclass AUC measures.

- **AUNU**: AUC of each class against the rest, using the uniform class distribution. Computes the AUC treating a c-dimensional classifier as c two-dimensional 1-vs-rest classifiers, where classes are assumed to have uniform distribution, in order to have a measure which is independent of class distribution change (Fawcett 2001).

- **AUNP**: AUC of each class against the rest, using the a-priori class distribution. Computes the AUC treating a c-dimensional classifier as c two-dimensional 1-vs-rest classifiers, taking into account the prior probability of each class (Fawcett 2001).

- **AU1U**: AUC of each class against each other, using the uniform class distribution. Computes something like the AUC of c(c - 1) binary classifiers (all possible pairwise combinations). See Hand (2001) for details.

- **AU1P**: AUC of each class against each other, using the a-priori class distribution. Computes something like AUC of c(c - 1) binary classifiers while considering the a-priori distribution of the classes as suggested in Ferri (2009). Note we deviate from the definition in Ferri (2009) by a factor of c. The person implementing this function and writing this very documentation right now cautions against using this measure because it is an imperfect generalization of AU1U.
Dictionary

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

```r
mlr_measures$get("classif.mauc_au1p")
msr("classif.mauc_au1p")
```

Parameters

Empty ParamSet

Meta Information

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

Note

The score function calls `mlr3measures::mauc_au1p()` from package mlr3measures.

If the measure is undefined for the input, NaN is returned. This can be customized by setting the field `na_value`.

See Also

Dictionary of Measures: mlr_measures

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


Multiclass AUC Scores

Description

Measure to compare true observed labels with predicted probabilities in multiclass classification tasks.

Details

Multiclass AUC measures.

- **AUNU**: AUC of each class against the rest, using the uniform class distribution. Computes the AUC treating a c-dimensional classifier as c two-dimensional 1-vs-rest classifiers, where classes are assumed to have uniform distribution, in order to have a measure which is independent of class distribution change (Fawcett 2001).

- **AUNP**: AUC of each class against the rest, using the a-priori class distribution. Computes the AUC treating a c-dimensional classifier as c two-dimensional 1-vs-rest classifiers, taking into account the prior probability of each class (Fawcett 2001).

- **AU1U**: AUC of each class against each other, using the uniform class distribution. Computes something like the AUC of \(c(c - 1)\) binary classifiers (all possible pairwise combinations). See Hand (2001) for details.

- **AU1P**: AUC of each class against each other, using the a-priori class distribution. Computes something like AUC of \(c(c - 1)\) binary classifiers while considering the a-priori distribution of the classes as suggested in Ferri (2009). Note we deviate from the definition in Ferri (2009) by a factor of c. The person implementing this function and writing this very documentation right now cautions against using this measure because it is an imperfect generalization of AU1U.

Dictionary

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

```r
mlr_measures$get("classif.mauc_au1u")
msr("classif.mauc_au1u")
```

Parameters

Empty ParamSet

Meta Information

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

The score function calls `mlr3measures::mauc_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.


• **AU1U**: AUC of each class against each other, using the uniform class distribution. Computes something like the AUC of \( c(c - 1) \) binary classifiers (all possible pairwise combinations). See Hand (2001) for details.

• **AU1P**: AUC of each class against each other, using the a-priori class distribution. Computes something like AUC of \( c(c - 1) \) binary classifiers while considering the a-priori distribution of the classes as suggested in Ferri (2009). Note we deviate from the definition in Ferri (2009) by a factor of \( c \). The person implementing this function and writing this very documentation right now cautions against using this measure because it is an imperfect generalization of AU1U.

**Dictionary**

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

```r
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.

Multiclass AUC Scores

Description

Measure to compare true observed labels with predicted probabilities in multiclass classification tasks.

Details

Multiclass AUC measures.

- **AUNU**: AUC of each class against the rest, using the uniform class distribution. Computes the AUC treating a c-dimensional classifier as c two-dimensional 1-vs-rest classifiers, where classes are assumed to have uniform distribution, in order to have a measure which is independent of class distribution change (Fawcett 2001).

- **AUNP**: AUC of each class against the rest, using the a-priori class distribution. Computes the AUC treating a c-dimensional classifier as c two-dimensional 1-vs-rest classifiers, taking into account the prior probability of each class (Fawcett 2001).

- **AU1U**: AUC of each class against each other, using the uniform class distribution. Computes something like the AUC of c(c - 1) binary classifiers (all possible pairwise combinations). See Hand (2001) for details.

- **AU1P**: AUC of each class against each other, using the a-priori class distribution. Computes something like AUC of c(c - 1) binary classifiers while considering the a-priori distribution of the classes as suggested in Ferri (2009). Note we deviate from the definition in Ferri (2009) by a factor of c. The person implementing this function and writing this very documentation right now cautions against using this measure because it is an imperfect generalization of AU1U.

Dictionary

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

```r
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 \texttt{mlr3measures::mauc_aunu()} from package \texttt{mlr3measures}.

If the measure is undefined for the input, NaN is returned. This can be customized by setting the field \texttt{na.value}.

### See Also

- Dictionary of Measures: \texttt{mlr_measures}
- \texttt{as.data.table(mlr_measures)} for a complete table of all (also dynamically created) Measure implementations.


### Description

Measure to compare true observed labels with predicted probabilities in multiclass classification tasks.
Details

Brier score for multi-class classification problems with \( r \) labels defined as

\[
\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{r} (I_{ij} - p_{ij})^2.
\]

\( I_{ij} \) is 1 if observation \( i \) has true label \( j \), and 0 otherwise.

Note that there also is the more common definition of the Brier score for binary classification problems in \texttt{bbrier()}. 

Dictionary

This Measure can be instantiated via the dictionary \texttt{mlr_measures} or with the associated sugar function \texttt{msr()}:

\[
\text{mlr_measures$get("classif.mbrier")}
\]
\[
\text{msr("classif.mbrier")}
\]

Parameters

- Empty ParamSet

Meta Information

- Type: "classif"
- Range: \([0, 2]\)
- Minimize: TRUE
- Required prediction: prob

Note

The score function calls \texttt{mlr3measures::mbrier()} from package \texttt{mlr3measures}.

If the measure is undefined for the input, NaN is returned. This can be customized by setting the field \texttt{na.value}.

See Also

Dictionary of Measures: \texttt{mlr_measures}

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

**mlr_measures_classif.mcc**

**Matthews Correlation Coefficient**

**Description**

Measure to compare true observed labels with predicted labels in binary classification tasks.

**Details**

The Matthews Correlation Coefficient is defined as

\[
\frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}.
\]

This above formula is undefined if any of the four sums in the denominator is 0. The denominator is then set to 1.

**Dictionary**

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

```r
mlr_measures$get("classif.mcc")
msr("classif.mcc")
```

**Parameters**

Empty ParamSet

**Meta Information**

- Type: "binary"
- Range: \([-1, 1]\]
- Minimize: FALSE
- Required prediction: response
The score function calls \texttt{mlr3measures::mcc()} from package \texttt{mlr3measures}.

If the measure is undefined for the input, NaN is returned. This can be customized by setting the field \texttt{na_value}.

\textbf{See Also}

\texttt{Dictionary of Measures: mlr_measures}

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


---

\texttt{mlr_measures_classif.npv}

\textit{Negative Predictive Value}

\textbf{Description}

Measure to compare true observed labels with predicted labels in binary classification tasks.

\textbf{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()`:

```r
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.


**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

\[
\text{Positive Predictive Value} = \frac{TP}{TP + FP}
\]

Also known 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`.
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 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 `mlr_measures::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.


---

**mlr.measures.classif.precision**

*Positive Predictive Value*

**Description**

Measure to compare true observed labels with predicted labels in binary classification tasks.
**mlr_measures_classif.precision**

### Details

The Positive Predictive Value is defined as

\[
\frac{TP}{TP + FP}
\]

Also known 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():

```r
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.

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 \( \frac{TP}{TP + FN} \).

Also known as "recall" or "sensitivity".
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():

```r
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.
**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

\[
\frac{TP}{TP + FN}
\]

Also know as "recall" or "sensitivity".

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()`:

```r
mlr_measures$get("classif.sensitivity")
msr("classif.sensitivity")
```
**mlr_measures_classif.specificity**

**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.


---

**mlr_measures_classif.specificity**

*True Negative Rate*

**Description**

Measure to compare true observed labels with predicted labels in binary classification tasks.
The True Negative Rate is defined as
\[
\frac{TN}{FP + TN}
\]
Also known as "specificity". This measure is undefined if \( FP + TN = 0 \).

This Measure can be instantiated via the dictionary \texttt{mlr_measures} or with the associated sugar function \texttt{msr()}:

\begin{verbatim}
mlr_measures$gget("classif.specificity")
msr("classif.specificity")
\end{verbatim}

This measure is undefined if \( FP + TN = 0 \).

The score function calls \texttt{mlr3measures::specificity()} from package \texttt{mlr3measures}.
If the measure is undefined for the input, \texttt{NaN} is returned. This can be customized by setting the field \texttt{na_value}.

Other classification measures:

---

`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.

**Dictionary**

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

```r
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`.

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 known as "specificity".

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.

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.

Dictionary
This Measure can be instantiated via the dictionary mlr_measures or with the associated sugar function msr():

```r
mlr_measures$get("classif.tp")
msr("classif.tp")
```

Parameters
Empty ParamSet

Meta Information
- Type: "binary"
- Range: [0, ∞)
- 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.
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

\[
\frac{TP}{TP + FN}
\]

Also know as "recall" or "sensitivity".

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():

```r
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.


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():

```r
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

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>na_ratio</td>
<td>numeric</td>
<td>-</td>
<td>[0, 1]</td>
</tr>
</tbody>
</table>

Super class

mlr3::Measure -> MeasureDebugClassif

Methods

Public methods:

- MeasureDebugClassif$new()
- MeasureDebugClassif$clone()

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

Usage:

```r
MeasureDebugClassif$new()
```

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

Usage:

```r
MeasureDebugClassif$clone(deep = FALSE)
```

Arguments:

deep  Whether to make a deep clone.
See Also

- 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_oob_error, mlr_measures_selected_features

Examples

```r
  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").

Dictionary

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

```r
  mlr_measures$get("time_train")
  msr("time_train")
```

Meta Information

- Task type: “NA”
- Range: [0, ∞)
- 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

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_oob_error`, `mlr_measures_selected_features`
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: (−∞, ∞)
- 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

• 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_selected_features

Description

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

Details

The Bias is defined as

\[
\frac{1}{n} \sum_{i=1}^{n} w_i (t_i - r_i).
\]

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: (−∞, ∞)
• 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.

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. 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")
**mlr_measures_regr.mae**

**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.

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

**Details**
The Mean Absolute Error is defined as

\[
\frac{1}{n} \sum_{i=1}^{n} w_i |t_i - r_i|.
\]

**Dictionary**
This Measure can be instantiated via the dictionary `mlr_measures` or with the associated sugar function `msr()`:

```r
mlr_measures$get("regr.mae")
msr("regr.mae")
```
**mlr_measures_regr.mape**

**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.


---

**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

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

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():

```r
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.


---

**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():

```r
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.

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()`:

```r
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`
**Description**

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

**Details**

The Median Squared Error is defined as

\[
\text{median}_i \left[ (t_i - r_i)^2 \right].
\]

**Dictionary**

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

```r
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.


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

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

Dictionary

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

mlr_measures$get("regr.mse")
msr("regr.mse")

Parameters

Empty ParamSet

Meta Information

- Type: "regr"
- Range: \([0, \infty)\)
- Minimize: TRUE
- Required prediction: response
The score function calls \texttt{mlr3measures::mse()} from package \texttt{mlr3measures}. If the measure is undefined for the input, NaN is returned. This can be customized by setting the field \texttt{na_value}.

See Also

Dictionary of Measures: \texttt{mlr_measures}

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

Other regression measures: \texttt{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.rae, mlr_measures_regr.rmse, mlr_measures_regr.rmsle, mlr_measures_regr.rrse, mlr_measures_regr.rse, mlr_measures_regr.rsq, mlr_measures_regr.sae, mlr_measures_regr.smape, mlr_measures_regr.srho, mlr_measures_regr.sse}

\begin{verbatim}
mlr_measures_regr.msle

\textit{Mean Squared Log Error}
\end{verbatim}

Description

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

Details

The Mean Squared Log Error is defined as

\[
\frac{1}{n} \sum_{i=1}^{n} w_i (\ln(1 + t_i) - \ln(1 + r_i))^2 .
\]

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 \texttt{mlr_measures} or with the associated sugar function \texttt{msr()}:

\begin{verbatim}
mlr_measures$get("regr.msle")
msr("regr.msle")
\end{verbatim}

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, \(\text{NaN} \) 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.rae`
- `mlr_measures_regr.rmse`
- `mlr_measures_regr.rmsle`
- `mlr_measures_regr.rrse`
- `mlr_measures_regr.rse`
- `mlr_measures_regr.rsq`
- `mlr_measures_regr.sae`
- `mlr_measures_regr.smape`
- `mlr_measures_regr.srho`
- `mlr_measures_regr.sse`

---

**Description**

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

**Details**

The Percent Bias is defined as

\[
\frac{1}{n} \sum_{i=1}^{n} w_i \frac{(t_i - r_i)}{|t_i|}.
\]

Good predictions score close to 0.

**Dictionary**

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

```r
mlr_measures$get("regr.pbias")
msr("regr.pbias")
```
**mlr_measures_regr.rae**

**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.


---

**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

\[
\frac{\sum_{i=1}^{n} |t_i - r_i|}{\sum_{i=1}^{n} |t_i - \bar{t}|}.
\]

Can be interpreted as absolute 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():

```r
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.


---

**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

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

Dictionary

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

```r
mlr_measures$get("regr.rmse")
msr("regr.rmse")
```

Parameters

Empty ParamSet

Meta Information

- Type: "regr"
- Range: [0, ∞)
- 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.

Description

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

Details

The Root Mean Squared Log Error is defined as

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

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():

```r
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.


---

**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

\[ \sqrt{\frac{\sum_{i=1}^{n} (t_i - r_i)^2}{\sum_{i=1}^{n} (t_i - \bar{t})^2}}. \]

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()`:

```r
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.

Meta Information

- Type: "regr"
- Range: [0, ∞)
- 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.


---

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

\[ 1 - \frac{\sum_{i=1}^{n} (t_i - r_i)^2}{\sum_{i=1}^{n} (t_i - \bar{t})^2}. \]

Also known as coefficient of determination or explained variation. Subtracts the `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()`:

```r
mlr_measures$get("regr.rsq")
msr("regr.rsq")
```
Parameters
Empty ParamSet

Meta Information
• Type: "regr"
• Range: $(-\infty, 1]$  
• Minimize: FALSE  
• Required prediction: response

Note
The score function calls `mlr3measures::rsq()` 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.


---

**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")`
mlr_measures_regr.smape

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.


---

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 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.


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()`:

```r
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.

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():

```r
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,
**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()`:

```r
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**

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>normalize</td>
<td>logical</td>
<td>-</td>
<td>TRUE, FALSE</td>
</tr>
</tbody>
</table>

**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


• 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`

Examples

task = tsk("german_credit")
learner = lrn("classif.rpart")
rr = resample(task, learner, rsmp("cv", folds = 3), store_models = TRUE)

scores = rr$score(msr("selected_features"))
scores[, c("iteration", "selected_features")]

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():

```r
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():

```r
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_cv`, `mlr_resamplings_cv`, `mlr_resamplings_holdout`, `mlr_resamplings_insample`, `mlr_resamplings_loo`, `mlr_resamplings_repeated_cv`, `mlr_resamplings_subsampling`

**Examples**

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

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():

```r
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:

```r
ResamplingBootstrap$new()
```

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

Usage:
mlr_resamplings_bootstrap

`ResamplingBootstrap$clone(deep = FALSE)`

**Arguments:**

depth Whether to make a deep clone.

**References**


**See Also**

- 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_resamplingsRepeated_cv, mlr_resamplings_subsampling`

**Examples**

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

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

# Individual sets:
bootstrap$train_set(1)
bootstrap$test_set(1)

# Disjunct sets:
intersect(bootstrap$train_set(1), bootstrap$test_set(1))

# Internal storage:
bootstrap$instance$M # Matrix of counts
```
**mlr_resamplings_custom**

*Custom Resampling*

**Description**

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

**Dictionary**

This **Resampling** can be instantiated via the dictionary **mlr_resamplings** or with the associated sugar function `rsmp()`:

```r
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:*

```r
ResamplingCustom$new()
```

**Method `instantiate()`**: Instantiate this **Resampling** with custom splits into training and test set.

*Usage:*

```r
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

- Package mlr3spatiotempcv for spatio-temporal resamplings.
- Dictionary of Resamplings: mlr_resamplings
- as.data.table(mlr_resamplings) for a table of available Resamplings in the running session (depending on the loaded packages).
- mlr3spatiotempcv for additional Resamplings for spatio-temporal tasks.

Other Resampling: Resampling, mlr_resamplings, mlr_resamplings_bootstrap, mlr_resamplings_custom_cv, mlr_resamplings_cv, mlr_resamplings_holdout, mlr_resamplings_insample, mlr_resamplings_loo, mlr_resamplings_repeated_cv, mlr_resamplings_subsampling

Examples

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

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

   custom$train_set(1)
   custom$test_set(1)
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 \( \text{task}\_nrow \), or via a single string \( \text{col} \) referring to a column in the task.

An alternative but equivalent approach using leave-one-out resampling is showcased in the examples of \texttt{mlr\_resamplings\_loo}.

Dictionary

This \texttt{Resampling} can be instantiated via the dictionary \texttt{mlr\_resamplings} or with the associated sugar function \texttt{rsmp()}:

\begin{verbatim}
mlr_resamplings$get("custom\_cv")
rsmp("custom\_cv")
\end{verbatim}

Super class

\texttt{mlr3::Resampling} -> \texttt{ResamplingCustomCV}

Active bindings

\texttt{iters (integer(1))}

Returns the number of resampling iterations, depending on the values stored in the \texttt{param\_set}.

Methods

Public methods:

- \texttt{ResamplingCustomCV\_new()}
- \texttt{ResamplingCustomCV\_instantiate()}
- \texttt{ResamplingCustomCV\_clone()}

Method \texttt{new()}: Creates a new instance of this \texttt{R6} class.

\textit{Usage:}

\texttt{ResamplingCustomCV\_new()}

Method \texttt{instantiate()}: Instantiate this \texttt{Resampling} as cross-validation with custom splits.

\textit{Usage:}

\texttt{ResamplingCustomCV\_instantiate(task, f = NULL, col = NULL)}

\textit{Arguments:}

- \texttt{task Task}
  - Used to extract row ids.
- \texttt{f (factor() | character())}
  - Vector of type factor or character with the same length as \texttt{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. \texttt{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:**
depth Whether to make a deep clone.

**See Also**
- Package [mlr3spatiotempcv](https://mlr3spatiotempcv.mlr-org.com/) for spatio-temporal resamplings.
- Dictionary of Resamplings: [mlr_resamplings](https://mlr-org.com/)
- `as.data.table(mlr_resamplings)` for a table of available Resamplings in the running session (depending on the loaded packages).
- [mlr3spatiotempcv](https://mlr3spatiotempcv.mlr-org.com/) 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
```r
task = tsk("penguins")
task$filter(1:10)
```

# Instantiate Resampling:
```r
custom_cv = rsmp("custom_cv")
f = factor(c(rep(letters[1:3], each = 3), NA))
custom_cv$instantiate(task, f = f)
custom_cv$iters # 3 folds
```

# Individual sets:
```r
custom_cv$train_set(1)
custom_cv$test_set(1)
```

# Disjunct sets:
```r
intersect(custom_cv$train_set(1), custom_cv$test_set(1))
```
**mlr_resamplings_cv**  
*Cross-Validation Resampling*

**Description**

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

**Dictionary**

This Resampling can be instantiated via the dictionary `mlr_resamplings` or with the associated sugar function `rsmp()`:

```r
dl_resamplings$get("cv")
rsmp("cv")
```

**Parameters**

- **folds (integer(1))**  
  Number of folds.

**Super class**

`mlr3::Resampling` -> `ResamplingCV`

**Active bindings**

- **iters (integer(1))**  
  Returns the number of resampling iterations, depending on the values stored in the `param_set`.

**Methods**

**Public methods:**

- `ResamplingCV$new()`  
- `ResamplingCV$clone()`

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

*Usage:*

```r
ResamplingCV$new()
```

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

*Usage:*

```r
ResamplingCV$clone(deep = FALSE)
```

*Arguments:*

- **deep** Whether to make a deep clone.
References


See Also

- 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

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

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

# Individual sets:
cv$train_set(1)
cv$test_set(1)

# Disjunct sets:
intersect(cv$train_set(1), cv$test_set(1))

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

---

**mlr_resamplings_holdout**

*Holdout Resampling*

**Description**

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

This Resampling can be instantiated via the dictionary `mlr_resamplings` or with the associated sugar function `rsmp()`:

```r
mlr_resamplings$get("holdout")
rsmp("holdout")
```

Parameters

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

Super class

`mlr3::Resampling` -> `ResamplingHoldout`

Public fields

- `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:

```r
ResamplingHoldout$new()
```

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

Usage:

```r
ResamplingHoldout$clone(deep = FALSE)
```

Arguments:

- `deep` Whether to make a deep clone.

References

See Also

- Package mlr3spatiotempcv for spatio-temporal resamplings.
- Dictionary of Resamplings: mlr_resamplings
- as.data.table(mlr_resamplings) for a table of available Resamplings in the running session (depending on the loaded packages).
- mlr3spatiotempcv for additional Resamplings for spatio-temporal tasks.

Other Resampling: Resampling, mlr_resamplings, mlr_resamplings_bootstrap, mlr_resamplings_custom, mlr_resamplings_custom_cv, mlr_resamplings_cv, mlr_resamplings_insample, mlr_resamplings_loo, mlr_resamplings_repeated_cv, mlr_resamplings_subsampling

Examples

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

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

# Individual sets:
holdout$train_set(1)
holdout$test_set(1)

# Disjunct sets:
intersect(holdout$train_set(1), holdout$test_set(1))

# Internal storage:
holdout$instance # simple list

---

mlr_resamplings_insample

*Insample Resampling*

Description

Uses all observations as training and as test set.

Dictionary

This Resampling can be instantiated via the dictionary mlr_resamplings or with the associated sugar function rsmp():
Super class

\textit{mlr3::Resampling} -> \textit{ResamplingInsample}

Public fields

\begin{verbatim}
iters (integer(1))
\end{verbatim}

Returns the number of resampling iterations, depending on the values stored in the \texttt{param_set}.

Methods

Public methods:

- \texttt{ResamplingInsample$new()}
- \texttt{ResamplingInsample$clone()}

Method \texttt{new()}: Creates a new instance of this \texttt{R6} class.

Usage:
\begin{verbatim}
ResamplingInsample$new()
\end{verbatim}

Method \texttt{clone()}: The objects of this class are cloneable with this method.

Usage:
\begin{verbatim}
ResamplingInsample$clone(deep = FALSE)
\end{verbatim}

Arguments:
- \texttt{deep} Whether to make a deep clone.

See Also


- Package \texttt{mlr3spatiotempcv} for spatio-temporal resamplings.

- Dictionary of Resamplings: \texttt{mlr_resamplings}

- \texttt{as.data.table(mlr_resamplings)} for a table of available Resamplings in the running session (depending on the loaded packages).

- \texttt{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

\begin{verbatim}
# Create a task with 10 observations
task = tsk("penguins")

# Instantiate Resampling
insample = rmp("insample")

insample$instantiate(task)
\end{verbatim}
# Train set equal to test set:
setequal(insample$train_set(1), insample$test_set(1))

# Internal storage:
insample$instance # just row ids

---

**mlr_resamplings_loo**  
*Leave-One-Out Cross-Validation*

**Description**

Splits data using leave-one-observation-out. This is identical to cross-validation with the number of folds set to the number of observations.

If this resampling is combined with the grouping features of tasks, it is possible to create custom splits based on an arbitrary factor variable, see the examples.

**Dictionary**

This Resampling can be instantiated via the dictionary `mlr_resamplings` or with the associated sugar function `rsmp()`:

```r
mlr_resamplings$get("loo")
rsmp("loo")
```

**Super class**

`mlr3::Resampling` -> `ResamplingLOO`

**Active bindings**

`iters (integer(1))`

Returns the number of resampling iterations which is the number of rows of the task provided to instantiate. Is NA if the resampling has not been instantiated.

**Methods**

**Public methods:**

- `ResamplingLOO$new()`
- `ResamplingLOO$clone()`

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

*Usage:*

```r
ResamplingLOO$new()
```

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

*Usage:*

```r
ResamplingLOO$clone(deep = FALSE)
```

*Arguments:*

deep  Whether to make a deep clone.
References


See Also

- 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

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

# Instantiate Resampling
loo = rsmp("loo")
loo$instance(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$set_col_roles("island", add_to = "group")
loo$instance(task)
loo$iters # one fold for each level of "island"
```
Repeated Cross-Validation Resampling

Description
Splits data repeats (default: 10) times using a folds-fold (default: 10) cross-validation. The iteration counter translates to repeats blocks of folds cross-validations, i.e., the first folds iterations belong to a single cross-validation. Iteration numbers can be translated into folds or repeats with provided methods.

Dictionary
This Resampling can be instantiated via the dictionary mlr_resamplings or with the associated sugar function rsmp():

```r
mlr_resamplings$get("repeated_cv")
rsmp("repeated_cv")
```

Parameters
- **repeats** (integer(1))
  Number of repetitions.
- **folds** (integer(1))
  Number of folds.

Super class
mlr3::Resampling -> ResamplingRepeatedCV

Active bindings
- **iters** (integer(1))
  Returns the number of resampling iterations, depending on the values stored in the param_set.

Methods
- **Public methods:**
  - `ResamplingRepeatedCV$new()`
  - `ResamplingRepeatedCV$folds()`
  - `ResamplingRepeatedCV$repeats()`
  - `ResamplingRepeatedCV$clone()`

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

  **Usage:**
ResamplingRepeatedCV$new()

Method folds(): Translates iteration numbers to fold numbers.

Usage:
ResamplingRepeatedCV$folds(iters)

Arguments:
iters (integer())
  Iteration number.

Returns: integer() of fold numbers.

Method repeats(): Translates iteration numbers to repetition numbers.

Usage:
ResamplingRepeatedCV$repeats(iters)

Arguments:
iters (integer())
  Iteration number.

Returns: integer() of repetition numbers.

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

Usage:
ResamplingRepeatedCV$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

References

See Also
- 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_insamp, mlr_resamplings_loo, mlr_resamplings_subsampling
Examples

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

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

# Individual sets:
repeated_cv$train_set(1)
repeated_cv$test_set(1)

# Disjunct sets:
intersect(repeated_cv$train_set(1), repeated_cv$test_set(1))

# Internal storage:
repeated_cv$instance # table

---

**mlr_resamplings_subsampling**

*Subsampling Resampling*

**Description**

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

**Dictionary**

This Resampling can be instantiated via the dictionary `mlr_resamplings` or with the associated sugar function `rsmp()`:

```r
mlr_resamplings$get("subsampling")
rsmp("subsampling")
```

**Parameters**

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

**Super class**

`mlr3::Resampling` -> `ResamplingSubsampling`
Active bindings

\texttt{iters} (integer(1))

Returns the number of resampling iterations, depending on the values stored in the \texttt{param_set}.

Methods

Public methods:

- \texttt{ResamplingSubsampling$new()}
- \texttt{ResamplingSubsampling$clone()}

Method \texttt{new}(): Creates a new instance of this R6 class.

\texttt{Usage:}
\begin{verbatim}
ResamplingSubsampling$new()
\end{verbatim}

Method \texttt{clone}(): The objects of this class are cloneable with this method.

\texttt{Usage:}
\begin{verbatim}
ResamplingSubsampling$clone(deep = FALSE)
\end{verbatim}

Arguments:

- \texttt{deep} Whether to make a deep clone.

References


See Also

- Package \texttt{mlr3spatiotempcv} for spatio-temporal resamplings.
- Dictionary of Resamplings: \texttt{mlr_resamplings}
- \texttt{as.data.table(mlr_resamplings)} for a table of available Resamplings in the running session (depending on the loaded packages).
- \texttt{mlr3spatiotempcv} for additional Resamplings for spatio-temporal tasks.

Other Resampling: \texttt{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

\begin{verbatim}
# Create a task with 10 observations
task = tsk("penguins")
task$filter(1:10)
# Instantiate Resampling
\end{verbatim}
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_get()` to extract objects from the respective `mlr3misc::Dictionary`:

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

Usage

tsk(.key, ...)

tsk(.keys, ...)

tgen(.key, ...)

tgens(.keys, ...)

lrn(.key, ...)

lrns(.keys, ...)

subsampling = rsmp("subsampling", repeats = 2, ratio = 0.5)
subsampling$instance$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
rsmp(.key, ...)  
rsmps(.keys, ...)  
msr(.key, ...)  
msrs(.keys, ...)

Arguments

.key (character(1))  
Key passed to the respective dictionary to retrieve the object.

... (named list())  
Named arguments passed to the constructor, to be set as parameters in the paradox::ParamSet, or to be set as public field. See mlr3misc::dictionary_sugar_get() for more details.

.keys (character())  
Keys passed to the respective dictionary to retrieve multiple objects.

Value

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

Examples

# 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")

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().
**mlr_tasks**

**Format**

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

**Methods**

See mlr3misc::Dictionary.

**S3 methods**

- as.data.table(dict, ..., objects = FALSE)
  
  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**

```r
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”

See Also

- 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).
- mlr3fsselect 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/.
**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():

```r
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

- 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_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”

Source

Data set originally published on UCI. This is the preprocessed version taken from packagerchallenge 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


See Also

- 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/.

Examples

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

---

**mlr_tasks_iris**  
**Iris Classification Task**

**Description**

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

**Format**

R6::R6Class inheriting from TaskClassif.

**Dictionary**

This Task can be instantiated via the dictionary mlr_tasks or with the associated sugar function tsk():

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

**Meta Information**

- Task type: “classif”
- Dimensions: 150x5
- Properties: “multiclass”
- Has Missings: FALSE
- Target: “Species”

**Source**


### 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”

---

**See Also**

- Package mlr3data for more toy tasks.
- Package mlr3oml for downloading tasks from [https://www.openml.org](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/](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
See Also

- 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

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


https://github.com/allisonhorst/palmerpenguins

See Also

• 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).
• mlr3fsselect 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

- 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/.
mlr_tasks_sonar

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  Sonar Classification Task

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

Format
R6::R6Class inheriting from TaskClassif.

Dictionary
This Task can be instantiated via the dictionary mlr_tasks or with the associated sugar function
tsk():

mlr_tasks$get("sonar")
ts(k("sonar")

Meta Information
• Task type: "classif"
• Dimensions: 208x61
• Properties: "twoclass"
• Has Missings: FALSE
• Target: "Class"
  "V52", "V53", "V54", "V55", "V56", "V57", "V58", "V59", "V6", "V60", "V7", "V8",
  "V9"

See Also
• 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

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”
mlr_tasks_spam


Source

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


See Also

- 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_penguins, mlr_tasks_pima, mlr_tasks_sonar, mlr_tasks_wine, mlr_tasks_zoo
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"

Source


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

References

mlr_tasks_zoo

See Also

- 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_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”

See Also

- 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).
- mlr3fselct 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.
**mlr_task_generators_2dnormals**

## 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

```r
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()`:

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

#### Parameters

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl</td>
<td>integer</td>
<td>-</td>
<td>[2, ∞)</td>
</tr>
<tr>
<td>r</td>
<td>numeric</td>
<td>-</td>
<td>[1, ∞)</td>
</tr>
<tr>
<td>sd</td>
<td>numeric</td>
<td>-</td>
<td>[0, ∞)</td>
</tr>
</tbody>
</table>
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

```r
generator = tgen("2dnormals")
plot(generator, n = 200)

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

---

### 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()`:

```r
mlr_task_generators$get("cassini")
tgen("cassini")
```

### Parameters

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>relsize1</td>
<td>integer</td>
<td>2</td>
<td>[1, ∞)</td>
</tr>
<tr>
<td>relsize2</td>
<td>integer</td>
<td>2</td>
<td>[1, ∞)</td>
</tr>
<tr>
<td>relsize3</td>
<td>integer</td>
<td>1</td>
<td>[1, ∞)</td>
</tr>
</tbody>
</table>

### 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())
Circle Classification Task Generator

Description

A TaskGenerator for the circle binary classification task in \texttt{mlbench::mlbench.circle()}. Creates a large circle containing a smaller circle.

Dictionary

This TaskGenerator can be instantiated via the dictionary \texttt{mlr_task_generators} or with the associated sugar function \texttt{tgen()}:  

\begin{verbatim}
mlr_task_generators$get("circle")
tgen("circle")
\end{verbatim}

Parameters

\begin{tabular}{lcc}
\hline
Id & Type & Default & Range \\
\hline
d & integer & 2 & $[2, \infty)$ \\
\hline
\end{tabular}

Super class

\texttt{mlr3::TaskGenerator} $\rightarrow$ TaskGeneratorCircle

Methods

Public methods:

- \texttt{TaskGeneratorCircle$\texttt{new()} }
- \texttt{TaskGeneratorCircle$\texttt{plot()} }
- \texttt{TaskGeneratorCircle$\texttt{clone()} }

\textbf{Method} \texttt{new()}: Creates a new instance of this \texttt{R6} class.

\textit{Usage:}

\texttt{TaskGeneratorCircle$\texttt{new()}}

\textbf{Method} \texttt{plot()}: Creates a simple plot of generated data.

\textit{Usage:}

\texttt{TaskGeneratorCircle$\texttt{plot(n = 200L, pch = 19L, ...)} }

\textit{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())
Dictionary

This TaskGenerator can be instantiated via the dictionary `mlr_task_generators` or with the associated sugar function `tgen()`:

```r
mlr_task_generators$get("friedman1")
tgen("friedman1")
```

Parameters

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>sd</td>
<td>numeric</td>
<td>1</td>
<td>[0, ∞)</td>
</tr>
</tbody>
</table>

Super class

`mlr3::TaskGenerator` -> `TaskGeneratorFriedman1`

Methods

Public methods:

- `TaskGeneratorFriedman1$new()`
- `TaskGeneratorFriedman1$clone()`

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

Usage:

```r
TaskGeneratorFriedman1$new()
```

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

Usage:

```r
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())

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

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>sigma</td>
<td>numeric</td>
<td>-</td>
<td>[0, ∞)</td>
</tr>
</tbody>
</table>

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())
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()`:

```r
mlr_task_generators$get("simplex")
tgen("simplex")
```

Parameters

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Levels</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>center</td>
<td>logical</td>
<td>TRUE</td>
<td>TRUE, FALSE</td>
<td>-</td>
</tr>
<tr>
<td>d</td>
<td>integer</td>
<td>3</td>
<td>[1, ∞)</td>
<td></td>
</tr>
<tr>
<td>sd</td>
<td>numeric</td>
<td>0.1</td>
<td>[0, ∞)</td>
<td></td>
</tr>
<tr>
<td>sides</td>
<td>integer</td>
<td>1</td>
<td>[1, ∞)</td>
<td></td>
</tr>
</tbody>
</table>

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())
Dictionary

This TaskGenerator can be instantiated via the dictionary `mlr_task_generators` or with the associated sugar function `tgen()`:

```r
mlr_task_generators$get("smiley")
tgen("smiley")
```

Parameters

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>sd1</td>
<td>numeric</td>
<td>-</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>sd2</td>
<td>numeric</td>
<td>-</td>
<td>(0, ∞)</td>
</tr>
</tbody>
</table>

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:
```r
TaskGeneratorSmiley$new()
```

Method `plot()`: Creates a simple plot of generated data.

Usage:
```r
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:
```r
TaskGeneratorSmiley$clone(deep = FALSE)
```

Arguments:
- `deep` Whether to make a deep clone.
See Also

- Dictionary of TaskGenerators: `mlr_taskGenerators`  
  - `as.data.table(mlr_taskGenerators)` 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.mlrr_taskGenerators.mlrr_taskGenerators_2dnormalss, mlrr_taskGenerators_cassini.mlrr_taskGenerators_circle, mlrr_taskGenerators_friedman1, mlrr_taskGenerators_moons.mlrr_taskGenerators_simplexx, mlrr_taskGenerators_spirals, mlrr_taskGenerators_xor`

Examples

```r
generator = tgen("smiley")  
plot(generator, n = 200)

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

---

mlrr_taskGenerators_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 `mlrr_taskGenerators` or with the associated sugar function `tgen()`:

```r
mlrr_taskGenerators$get("spirals")  
tgen("spirals")
```

Parameters

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycles</td>
<td>integer</td>
<td>1</td>
<td>[1, (\infty)]</td>
</tr>
<tr>
<td>sd</td>
<td>numeric</td>
<td>0</td>
<td>[0, (\infty)]</td>
</tr>
</tbody>
</table>
Super class

\[ \text{mlr3::TaskGenerator} \rightarrow \text{TaskGeneratorSpirals} \]

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:
deeep 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**

```r
generator = tgen("spirals")
plot(generator, n = 200)

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

---

**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()`:

```r
mlr_task_generators$get("xor")
tgen("xor")
```

**Parameters**

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>integer</td>
<td>1</td>
<td>[1, ∞)</td>
</tr>
</tbody>
</table>

**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:**

```r
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
```r
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:

```r
lapply(list.files(system.file("testthat", package = "mlr3"), pattern = "^helper.*\.[rR]", full.names = TRUE), source)
```

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.

**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`. 
run_paramtest()

**Description:**
Checks parameters of mlr3 Learners against parameters defined in the upstream functions of the respective learner. The goal is to detect if parameters have been dropped or added in the upstream implementation. Some learners do not have all of their parameters stored within the learner function that is called during training. Sometimes learners come with a "control" function, e.g. glmnet.control() from package glmnet. Such learners need to be checked as well since they make up the full ParamSet of the respective learner.

To work nicely with the defined ParamSet, certain parameters need to be excluded because these are only present in either the "control" object or the actual top-level function call. Such exclusions should go into argument exclude with a comment for the reason of the exclusion. See examples for more information.

For example usages you can look at the parameter tests in various mlr3 source repositories such as mlr3learners.

**Parameters:**
- **learner** (Learner)
  The learner whose parameter set is being checked.
- **fun** (function() or list of functions()s)
  The function(s) containing the parameters that must be implemented by the learner.
- **exclude** (character())
  Argument names that specified through this argument are exempt from checking. This can be used when parameters that are available in the fun function(s) are not implemented in the learner, or when the learner implements additional parameters that are not available in the fun function(s).
- **tag** (character(1))
  Only parameters that are tagged with this tag are being checked. If NULL (default), all parameters are checked.

expect_learner()

Checks various properties that learners have to satisfy. Used for testing learner implementations, especially if all methods and fields are implement as document.

**Parameters**
- **lrn** :: (Learner)
  The learner whose properties are being verified.
- **tsk** :: (Task)
  Optional argument (default is NULL). If provided, some additional checks are being run that check the compatibility of the learner and task.
- **check_man** :: (logical(1))
  Whether to check if the learner has a man page.
partition

Manually Partition into Training and Test Set

Description

Creates a split of the row ids of a Task into a training set and a test set while optionally stratifying on the target column.

For more complex partitions, see the example.

Usage

partition(task, ratio = 0.67, stratify = TRUE, ...)  

## S3 method for class 'TaskRegr'
partition(task, ratio = 0.67, stratify = TRUE, bins = 3L, ...)

## S3 method for class 'TaskClassif'
partition(task, ratio = 0.67, stratify = TRUE, ...)

Arguments

task  
(Task)

Task to operate on.

ratio  
(numeric(1))

Ratio of observations to put into the training set.

stratify  
(logical(1))

If TRUE, stratify on the target variable. For regression tasks, the target variable is first cut into bins bins. See Task$add_strata().

...  
(any)

Additional arguments, currently not used.

bins  
(integer(1))

Number of bins to cut the target variable into for stratification.

Examples

# regression task
task = tsk("boston_housing")

# roughly equal size split while stratifying on the binned response
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))
)
boxplot(y ~ split, data = data)
```r
# classification task
task = tsk("pima")
split = partition(task)

# roughly same distribution of the target label
prop.table(table(task$truth()))
prop.table(table(task$truth(split$train)))
prop.table(table(task$truth(split$test)))

# splitting into 3 disjunct sets, using ResamplingCV and stratification
task = tsk("iris")
task$set_col_roles(task$target_names, add_to = "stratum")
r = rsmp("cv", folds = 3)$instantiate(task)
sets = lapply(1:3, r$train_set)
lengths(sets)
prop.table(table(task$truth(sets[[1]])))
```

### predict.Learner

**Predict Method for Learners**

**Description**

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

Performs the following steps:

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

**Usage**

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

**Arguments**

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

Examples

```r
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$filter()
   • Prediction$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** `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**

- 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 keeps the object in its previous state.

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

Usage:
   PredictionClassif$clone(deep = FALSE)

Arguments:
   deep  Whether to make a deep clone.

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

See Also
   • 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$pset_threshold(th)$response
   p$p$score(measures = msr("classif.ce"))
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

```r
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

- `pdata` (PredictionData)
  Named list inheriting from "PredictionData".
- `row_ids` (one or more PredictionData objects).
- `train_task` (Task)
  Task used for training the learner.
- `keep_duplicates` (logical(1)) If TRUE, the combined PredictionData object is filtered for duplicated row ids (starting from last).

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.
- `distr` (VectorDistribution)

Methods

Public methods:

- `PredictionRegr$new()`
- `PredictionRegr$clone()`

Method `new()`: Creates a new instance of this R6 class.
PredictionRegr$new(
    task = NULL,
    row_ids = task$row_ids,
    truth = task$truth(),
    response = NULL,
    se = 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.

- **distr** (VectorDistribution)
  VectorDistribution from package distr6 (in repository https://raphael1.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

- 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

```r
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

```r
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 (Task).
- learner (Learner).
- resampling (Resampling).
- store_models (logical(1))
  Store the fitted model in the resulting object. Set to TRUE if you want to further analyse the models or want to extract information like variable importance.
- store_backends (logical(1))
  Keep the DataBackend of the Task in the ResampleResult? Set to TRUE if your performance measures require a Task, or to analyse results more conveniently. Set to FALSE to reduce the file size and memory footprint after serialization. The current default is TRUE, but this eventually will be changed in a future release.
resample

encapsulate (character(1))
If not NA, enables encapsulation by setting the field Learner$encapsulate to one of the supported values: "none" (disable encapsulation), "try" (captures errors but output is printed to the console and not logged), "evaluate" (execute via evaluate) and "callr" (start in external session via callr). If NA, encapsulation is not changed, i.e. the settings of the individual learner are active. Additionally, if encapsulation is set to "evaluate" or "callr", the fallback learner is set to the featureless learner if the learner does not already have a fallback configured.

allow_hotstart (logical(1))
Determines if learner(s) are hot started with trained models in $hotstart_stack. See also HotstartStack.

clone (character())
Select the input objects to be cloned before proceeding by providing a set with possible values "task", "learner" and "resampling" for Task, Learner and Resampling, respectively. Per default, all input objects are cloned.

unmarshal Learner
Whether to unmarshal learners that were marshaled during the execution. Setting this to FALSE does not guarantee that the learners are marshaled. For example, with sequential execution and no encapsulation, marshaling is not necessary. If you want to ensure that all learners are in marshaled form, you need to call $marshal() on the result object.

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().
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":

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

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

```r
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.
- Package `mlr3viz` for some generic visualizations.

Other resample: `ResampleResult`

Examples

```r
task = tsk("penguins")
learner = lrn("classif.rpart")
resampling = rsmp("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
```
```r
rr$score(msr("classif.ce"))
rr$aggregate(msr("classif.ce"))

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

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

# Convert results to BenchmarkResult, then combine them
bmr1 = as_benchmark_result(rr)
bmr2 = as_benchmark_result(rr_featureless)
print(bmr1$combine(bmr2))
```

---

**ResampleResult**

*Container for Results of resample()*

**Description**

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

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

*ResampleResults* can be visualized via [mlr3viz](https://mlr3viz.org)'s `autoplot()` function.

**S3 Methods**

- `as.data.table(rr, reassemble_learners = TRUE, convert_predictions = TRUE, predict_sets = "test")`
  - `ResampleResult` $\rightarrow$ `data.table::data.table()`
  - Returns a tabular view of the internal data.

- `c(...)`
  - `(ResampleResult, ...) $\rightarrow$ 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$score()
• ResampleResult$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())

Returns: Prediction. Subset of {"train", "test"}.

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"}.

Returns: List of Prediction objects, one per element in predict_sets.

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. 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,
    predict_sets = "test"
)

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.
predict_sets (character())
    Vector of predict sets ("train", "test") to construct the Prediction objects from. Default is "test".

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

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.
ResampleResult

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.
• 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)
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

<table>
<thead>
<tr>
<th>Resampling</th>
<th>Resampling Class</th>
</tr>
</thead>
</table>

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:*

```r
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:*

```r
Resampling$format(...)```

*Arguments:*

- `...` (ignored).

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

*Usage:*

```r
Resampling$print(...)```

*Arguments:*

- `...` (ignored).

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

*Usage:*

```r
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**

- 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

\[
r = \text{rsmp("subsampling")}
\]

# 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

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 Learner. This object is modified in-place.

- **n** (integer(1))
  Number of threads to use. Defaults to `parallelly::availableCores()`.

- **...** (any)
  Additional arguments.

Value

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

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.

• Modification of column or row roles via $set_col_roles() or $set_row_roles(), respective-
  tively.

• $filter() and $select() subset the set of active rows or features in $row_roles or $col_roles,
  respectively. This provides a different "view" on the data without altering the data itself.

• rbind() and cbind() change the task in-place by binding rows or columns to the data,
  but without modifying the original DataBackend. Instead, the methods first create a new
  DataBackendDataTable from the provided new data, and then merge both backends into an
  abstract DataBackend which merges the results on-demand.

• rename() wraps the DataBackend of the Task in an additional DataBackend which deals with
  the renaming. Also updates $col_roles and $col_info.

• set_levels() and droplevels() update the field col_info()'.

Public fields

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

task_type (character(1))
  Task type, e.g. "classif" or "regr".
  For a complete list of possible task types (depending on the loaded packages), see mlr_reflections$task_types$type

backend (DataBackend)
  Abstract interface to the data of the task.

col_info (data.table::data.table())
  Table with with 4 columns:
  • "id" (character()) stores the name of the column.
  • "type" (character()) holds the storage type of the variable, e.g. integer, numeric or
    character. See mlr_reflections$task_feature_types for a complete list of allowed types.
  • "levels" (list()) stores a vector of distinct values (levels) for ordered and unordered
    factor variables.
  • "label" (character()) stores a vector of prettier, formated column names.
• "fix_factor_levels" (logical()) stores flags which determine if the levels of the respective variable need to be reordered after querying the data from the DataBackend.

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.

eextra_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.

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 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.
• "test": Observations are hold back unless explicitly queried. Can be queried by a Learner to determine a good iteration to stop by evaluating the performance on external data, e.g. the XGboost learner in mlr3learners for parameter nrounds.
• "holdout": Observations are hold back unless explicitly queried. Can be used, e.g., as truly independent holdout set:
  1. Add "holdout" to the predict_sets of a Learner.
  2. Configure a Measure to use the "holdout" set by updating its predict_sets field.

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".

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()
Vector of supported data output formats. A specific format can be chosen in the $data() method.

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:
**Task**

- `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$new()`
- `Task$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 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 in the data format specified by `data_format`. 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:
```r
Task$data(
  rows = NULL,
  cols = NULL,
  data_format = "data.table",
  ordered = FALSE
)
```

Arguments:
- `rows` (positive integer()): Vector or row indices.
- `cols` (character()): Vector of column names.
- `data_format` (character(1)): Desired data format, e.g. "data.table" or "Matrix".
- `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:
```r
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:
```r
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.

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

**Method select():** Subsets the task, keeping only the features specified via column names `cols`.

Note that you cannot deselect the target column, for obvious reasons.

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

*Usage:*

```
Task$select(cols)
```

*Arguments:*

- `cols` (character())
  - Vector of column names.
Returns: Returns the object itself, but modified by reference. You need to explicitly $clone() the object beforehand if you want to keeps the object in its previous state.

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$rrbind(data)
```

Arguments:
`data` (data.frame).

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

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$rcbind(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 keeps 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.

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

Method 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 columns names cols from roles specified in remove_from. Other column roles are preserved.

Details: Roles are first set exclusively (argument roles), then added (argument add_to) and finally removed (argument remove_from) from different roles.

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

Method 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

- 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`

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

```r
# We use the inherited class TaskClassif here,
# because the base class 'Task' is not intended for direct use
task = TaskClassif$new("penguins", palmerpenguins::penguins, target = "species")

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

# de-select "year"
print(task$select(setdiff(task$feature_names, "year")))

print(task$feature_names)

# Add new column "foo"
print(task$cbind(data.frame(foo = 1:344)))
print(head(task))
```

---

### TaskClassif

TaskClassif

**Classification Task**

**Description**

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

Additional task properties include:

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

It is recommended to use `as_task_classif()` for construction. Predefined tasks are stored in the dictionary `mlr_tasks`.

**Super classes**

`mlr3::Task` -> `mlr3::TaskSupervised` -> `TaskClassif`
Active bindings

class_names (character())
Returns all class labels of the target column.

positive (character(1))
Stores the positive class for binary classification tasks, and NA for multiclass tasks. To switch the positive class, assign a level to this field.

negative (character(1))
Stores the negative class for binary classification tasks, and NA for multiclass tasks.

Methods

Public methods:

• TaskClassif$new()
• TaskClassif$truth()
• TaskClassif$droplevels()
• TaskClassif$clone()

Method new(): Creates a new instance of this R6 class. The function as_task_classif() provides an alternative way to construct classification tasks.

Usage:

TaskClassif$new(
  id,
  backend,
  target,
  positive = NULL,
  label = NA_character_,
  extra_args = list()
)

Arguments:

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

backend (DataBackend)
  Either a DataBackend, or any object which is convertible to a DataBackend with as_data_backend(). E.g., a data.frame() will be converted to a DataBackendDataTable.

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

positive (character(1))
  Only for binary classification: Name of the positive class. The levels of the target columns are reordered accordingly, so that the first element of $class_names is the positive class, and the second element is the negative class.

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

extra_args (named list())
  Named list of constructor arguments, required for converting task types via convert_task().
Method `truth()`: True response for specified `row_ids`. Format depends on the task type. Defaults to all rows with role "use".

Usage:
TaskClassif$truth(rows = NULL)

Arguments:
rows (positive integer())
  Vector or row indices.

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
- 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, 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_taskGenerators) 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_2dnormal, 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 Regression Task

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

It is recommended to use as_task_regr() for construction. Predefined tasks are stored in the dictionary mlr_tasks.

Super classes
mlr3::Task -> mlr3::TaskSupervised -> TaskRegr

Methods

Public methods:
- TaskRegr$new()
- TaskRegr$truth()
- TaskRegr$clone()

Method new(): Creates a new instance of this R6 class. The function as_task_regr() provides an alternative way to construct regression tasks.
Usage:
TaskRegr$new(id, backend, target, label = NA_character_, extra_args = list())

Arguments:
id (character(1))
   Identifier for the new instance.
backend (DataBackend)
   Either a DataBackend, or any object which is convertible to a DataBackend with as_data_backend().
   E.g., a data.frame() will be converted to a DataBackendDataTable.
target (character(1))
   Name of the target column.
label (character(1))
   Label for the new instance.
extra_args (named list())
   Named list of constructor arguments, required for converting task types via convert_task().

Method truth(): True response for specified row_ids. Format depends on the task type. Defaults to all rows with role "use".

Usage:
TaskRegr$truth(rows = NULL)

Arguments:
rows (positive integer())
   Vector or row indices.

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
- Package mlr3data for more toy tasks.
- Package mlr3ml 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).
- mlr3fsselect and mlr3filters for feature selection and feature filtering.
- Extension packages for additional task types:
  - Unsupervised clustering: mlr3cluster
Probabilistic supervised regression and survival analysis: https://mlr3proba.mlr-org.com/.

Other Task: Task, TaskClassif, TaskSupervised, TaskUnsupervised, 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

```r
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, 38
  - DataBackendDataTable, 40
  - DataBackendMatrix, 43

* **Dictionary**
  - mlr_learners, 75
  - mlr_measures, 90
  - mlr_resamplings, 171
  - mlr_task_generators, 206
  - mlr_tasks, 190

* **Learner**
  - Learner, 50
  - LearnerClassif, 57
  - LearnerRegr, 60
  - mlr_learners, 75
  - mlr_learners_classif.debug, 76
  - mlr_learners_classif.featureless, 79
  - mlr_learners_classif.rpart, 82
  - mlr_learners_regr.debug, 84
  - mlr_learners_regr.featureless, 86
  - mlr_learners_regr.rpart, 88

* **Measure**
  - Measure, 62
  - MeasureClassif, 67
  - MeasureRegr, 70
  - MeasureSimilarity, 72
  - mlr_measures, 90
  - mlr_measures_aic, 91
  - mlr_measures_bic, 93
  - mlr_measures_classif.costs, 101
  - mlr_measures_debug_classif, 139
  - mlr_measures_elapsed_time, 141
  - mlr_measures_oob_error, 143
  - mlr_measures_selected_features, 167

* **Prediction**
  - Prediction, 227
  - PredictionClassif, 230
  - PredictionRegr, 234

* **Resampling**
  - mlr_resamplings, 171
  - mlr_resamplings_bootstrap, 172
  - mlr_resamplings_custom, 174
  - mlr_resamplings_custom_cv, 175
  - mlr_resamplings_cv, 178
  - mlr_resamplings_holdout, 179
  - mlr_resamplings_insample, 181
  - mlr_resamplings_loo, 183
  - mlr_resamplings_repeated_cv, 185
  - mlr_resamplings_subsampling, 187
  - Resampling, 244

* **TaskGenerator**
  - mlr_task_generators, 206
  - mlr_task_generators_2dnormals, 207
  - mlr_task_generators_cassini, 209
  - mlr_task_generators_circle, 211
  - mlr_task_generators_friedman1, 212
  - mlr_task_generators_moons, 214
  - mlr_task_generators_simplex, 216
  - mlr_task_generators_smiley, 217
  - mlr_task_generators_spirals, 219
  - mlr_task_generators_xor, 221
  - TaskGenerator, 263

* **Task**
  - mlr_tasks, 190
  - mlr_tasks_boston_housing, 192
  - mlr_tasks_breast_cancer, 193
  - mlr_tasks_german_credit, 194
  - mlr_tasks_iris, 196
  - mlr_tasks_mtcars, 197
  - mlr_tasks_penguins, 198
  - mlr_tasks_pima, 200
  - mlr_tasks_sonar, 201
  - mlr_tasks_spam, 202
  - mlr_tasks_wine, 204
  - mlr_tasks_zoo, 205
INDEX

269

Task, 249
TaskClassif, 260
TaskRegr, 265

* benchmark
  benchmark, 25
  benchmark_grid, 35
  BenchmarkResult, 28

* binary classification measures
  mlr_measures_classif.auc, 95
  mlr_measures_classif.bbrier, 98
  mlr_measures_classif.dor, 103
  mlr_measures_classif.fbeta, 104
  mlr_measures_classif.fdr, 106
  mlr_measures_classif.fn, 107
  mlr_measures_classif.fnr, 108
  mlr_measures_classif.fomr, 110
  mlr_measures_classif.fp, 111
  mlr_measures_classif.fpr, 112
  mlr_measures_classif.mcc, 123
  mlr_measures_classif.npv, 124
  mlr_measures_classif.ppv, 126
  mlr_measures_classif.prauc, 127
  mlr_measures_classif.precision, 128
  mlr_measures_classif.recall, 130
  mlr_measures_classif.sensitivity, 131
  mlr_measures_classif.specificity, 132
  mlr_measures_classif.tn, 134
  mlr_measures_classif.tnr, 135
  mlr_measures_classif.tp, 137
  mlr_measures_classif.tpr, 138

* classification measures
  mlr_measures_classif.acc, 94
  mlr_measures_classif.auc, 95
  mlr_measures_classif.bacc, 97
  mlr_measures_classif.bbrier, 98
  mlr_measures_classif.ce, 99
  mlr_measures_classif.costs, 101
  mlr_measures_classif.dor, 103
  mlr_measures_classif.fbeta, 104
  mlr_measures_classif.fdr, 106
  mlr_measures_classif.fn, 107
  mlr_measures_classif.fnr, 108
  mlr_measures_classif.fomr, 110
  mlr_measures_classif.fp, 111
  mlr_measures_classif.fpr, 112
  mlr_measures_classif.logloss, 114
  mlr_measures_classif.mauc_au1p, 115
  mlr_measures_classif.mauc_au1u, 117
  mlr_measures_classif.mauc_aunp, 118
  mlr_measures_classif.mauc_aunu, 120
  mlr_measures_classif.mbrier, 121
  mlr_measures_classif.mcc, 123
  mlr_measures_classif.npv, 124
  mlr_measures_classif.ppv, 126
  mlr_measures_classif.prauc, 127
  mlr_measures_classif.precision, 128
  mlr_measures_classif.recall, 130
  mlr_measures_classif.sensitivity, 131
  mlr_measures_classif.specificity, 132
  mlr_measures_classif.tn, 134
  mlr_measures_classif.tnr, 135
  mlr_measures_classif.tp, 137
  mlr_measures_classif.tpr, 138

* datasets
  mlr_learners, 75
  mlr_measures, 90
  mlr_resamplings, 171
  mlr_task_generators, 206
  mlr_tasks, 190

* multiclass classification measures
  mlr_measures_classif.acc, 94
  mlr_measures_classif.bacc, 97
  mlr_measures_classif.ce, 99
  mlr_measures_classif.costs, 101
  mlr_measures_classif.logloss, 114
  mlr_measures_classif.mauc_au1p, 115
  mlr_measures_classif.mauc_au1u, 117
  mlr_measures_classif.mauc_aunp, 118
  mlr_measures_classif.mauc_aunu, 120
  mlr_measures_classif.mbrier, 121

* regression measures
  mlr_measures_regr.bias, 144
mrl_measures_regr.ktau, 145
mrl_measures_regr.mae, 146
mrl_measures_regr.mape, 147
mrl_measures_regr.maxae, 148
mrl_measures_regr.medae, 150
mrl_measures_regr.medse, 151
mrl_measures_regr.mse, 152
mrl_measures_regr.msle, 153
mrl_measures_regr.pbias, 154
mrl_measures_regr.rae, 155
mrl_measures_regr.rmse, 156
mrl_measures_regr.rmsle, 158
mrl_measures_regr.rrse, 159
mrl_measures_regr.rsquare, 160
mrl_measures_regr.rsq, 161
mrl_measures_regr.sae, 162
mrl_measures_regr.smape, 163
mrl_measures_regr.srho, 164
mrl_measures_regr.srho, 164
mrl_measures_regr.smape, 163
mrl_measures_regr.srho, 164
mrl_measures_regr.sroba, 165
mrl_measures_regr.sse, 166

* resample
  resample, 236
  ResampleResult, 239

* similarity measures
  mrl_measures_sim.jaccard, 169
  mrl_measures_sim.phi, 170

as_benchmark_result, 8
as_benchmark_result(), 17, 238, 243
as_data_backend
  (as_data_backend.Matrix), 9
  as_data_backend(), 39, 56
  as_data_backend.Matrix, 9, 40, 42, 45
  as_learner, 10
  as_learners (as_learner), 10
  as_measure, 11
  as_measures (as_measure), 11
  as_prediction, 12
  as_prediction(), 233
  as_prediction_classif, 13
  as_prediction_data, 14
  as_prediction_data(), 233
  as_prediction_regr, 15
  as_predictions (as_prediction), 12
  as_resample_result, 16
  as_resample_result(), 240
  as_resampling, 17
  as_resamplings (as_resampling), 17
  as_result_data, 17
  as_result_data(), 16, 30, 240
  as_task, 19
  as_task_classif, 19
  as_task_classif(), 260, 261
  as_task_regr, 22
  as_task_regr(), 265
  as_task_unsupervised, 24
  as_tasks (as_task), 19
  as_tasks_unsupervised
    (as_task_unsupervised), 24

bbrier(), 122
benchmark, 25, 34, 36
benchmark(), 7, 12, 28, 32, 46, 52, 55, 63, 65,
  66, 69, 71, 74, 244
benchmark_grid, 27, 34, 35
benchmark_grid(), 25
BenchmarkResult, 8, 9, 17, 26–28, 28, 29–31,
  33, 36, 52, 55, 63, 238–240, 243
bootstrap, 244
boston_housing, 249
c(), 30
c.PredictionDataClassif
  (PredictionData), 233
c.PredictionDataRegr (PredictionData), 233
check_prediction_data (PredictionData), 233
classif.auc, 62
classif.ce, 67
classif.rpart, 50
convert_task, 37
convert_task(), 19, 22, 251, 254, 261, 266
cut(), 259
cv, 244
data.frame(), 9, 19, 22, 25, 43, 226, 253
data.table(), 41, 240
data.table::as.data.table(), 9
data.table::copy(), 41

data.table::data.table(), 9, 25, 29, 32,
  36, 38, 39, 41, 43, 44, 46, 52, 75, 91,
  171, 191, 207, 227, 239, 240, 242,
  250–253, 255, 256
DataBackend, 9, 10, 19, 22, 26, 33, 38, 39, 40,
  42, 43, 45, 56, 236, 243, 249–251,
  253–257, 261, 266
DataBackendDataTable, 9, 10, 38–40, 40, 45,
  250, 254, 261, 266
DataBackendMatrix, 9, 10, 38–40, 42, 43
datasets::iris, 196
datasets::mtcars, 197
default_measures, 45
default_measures(), 241, 242
expand.grid(), 35
extract_pkgs(install_pkgs), 48
filter_prediction_data
  (PredictionData), 233
formula, 19, 22
formula(), 255
future::future(), 7
future::multicore, 248
future::nbrofWorkers(), 52
future::plan, 248
future::plan(), 27, 52, 237
head(), 250
HotstartStack, 26, 46, 53, 237
install_pkgs, 48
intersect(), 252
iris data set, 198
is_missing_prediction_data
  (PredictionData), 233
LearnerClassif, 50, 53, 57, 62, 76, 79, 81, 82, 84, 86, 88, 90, 230
LearnerClassifDebug
  (mlr_learners_classif.debug), 76
LearnerClassifFeatureless
  (mlr_learners_classif.featureless), 79
LearnerClassifRpart
  (mlr_learners_classif.rpart), 82
LearnerRegr, 50, 53, 57, 60, 76, 79, 81, 84, 86, 88, 90, 234
LearnerRegrDebug
  (mlr_learners_regr.debug), 84
LearnerRegrFeatureless
  (mlr_learners_regr.featureless), 86
LearnerRegrRpart
  (mlr_learners_regr.rpart), 88
Learners, 56, 59, 62, 79, 81, 84, 85, 87, 90
lgr::lgr-package, 27, 238
lrn (mlr_sugar), 189
lrn(), 75–77, 80, 82, 84, 86, 88
lrns (mlr_sugar), 189
lrns(), 75, 76
mad(), 86
marshal_model(), 31, 78, 243
matrix(), 19, 22
Matrix::Matrix(), 19, 22, 39, 43
Matrix::sparseMatrix(), 38
max.col(), 231, 232
mbrier(), 98
mean(), 65, 68, 71, 73, 86
Measure, 11, 31, 32, 46, 48, 52, 62, 63, 67, 70, 72, 75, 90–170, 189, 229, 241, 242, 252
MeasureAIC (mlr_measures_aic), 91
MeasureBIC (mlr_measures_bic), 93
MeasureClassif, 62, 64, 67, 67, 72, 75, 91, 92, 94, 102, 141, 142, 144, 168
MeasureClassifCosts, 194
MeasureClassifCosts
  (mlr3::MeasureClassifCosts), 101
MeasureDebugClassif
  (mlr3::MeasureDebugClassif), 139
MeasureElapsedTime
  (mlr3::MeasureElapsedTime), 141
MeasureOOBEError
  (mlr3::MeasureOOBEError), 143
MeasureRegr, 62, 64, 67, 70, 70, 75, 91, 92, 94, 102, 141, 142, 144, 168
MeasureSelectedFeatures
  (mlr3measures::MeasureSelectedFeatures), 167
MeasureSimilarity, 67, 70, 72, 72, 91, 92, 94, 102, 141, 142, 144, 168
median(), 86
mlbench::BostonHousing2, 192
mlbench::BreastCancer, 193
mlbench::mlbench.2dnormals(), 207
mlbench::mlbench.cassini(), 209
mlbench::mlbench.circle(), 211
mlbench::mlbench.Friedman1(), 212
mlbench::mlbench.simplex(), 216
mlbench::mlbench.smiley(), 217
mlbench::mlbench.spirals(), 219
mlbench::mlbench.xor(), 221
mlbench::PimaIndiansDiabetes2, 200
mlbench::Sonar, 201
mlbench::Zoo, 205
mlr3::mlr3-package, 6
mlr3-package, 6
mlr3::DataBackend, 40, 43
mlr3::Learner, 57, 60, 78, 80, 83, 85, 87, 89
mlr3::LearnerClassif, 78, 80, 83
mlr3::LearnerRegr, 83, 87, 89
mlr3::Measure, 67, 70, 72, 92, 93, 101, 140,


mlr_measures_oob_error, 95, 96, 122, 124, 128–136, 135, 136, 138, 139
mlr_measures_regr.pbias, 145–149, 151–154, 154, 156, 157, 159–165, 167
mlr_measures_regr.rae, 145–149, 151–155, 155, 157, 159–165, 167
mlr_measures_regr.rmse, 145–149, 151–156, 156, 159–165, 167
mlr_measures_regr.sse, 145–149, 151–157, 159–165, 166
mlr_measures_selected_features, 67, 70, 72, 75, 91, 92, 94, 102, 141, 142, 144, 167
mlr_measures_sim.jaccard, 169, 170
mlr_measures_sim.phi, 169, 170
mlr_measures_time_both
  (mlr_measures_elapsed_time), 141
mlr_measures_time_predict
  (mlr_measures_elapsed_time), 141
mlr_measures_time_train
  (mlr_measures_elapsed_time), 141
mlr_reflections, 226
mlr_reflections$default_measures, 11, 45
mlr_reflections$learner_predict_types, 51, 54, 58, 61, 63, 69, 71, 74
mlr_reflections$learner_properties, 51, 54, 58, 61
mlr_reflections$measure_properties, 65, 69, 71, 74
mlr_reflections$task_feature_types, 51, 54, 58, 61, 250
mlr_reflections$task_properties, 251
mlr_reflections$task_types, 37
mlr_reflections$task_types$type, 51, 54, 63, 65, 250, 254, 263, 264
mlr_resamplings, 76, 91, 171, 172–189, 191, 207, 244, 247
mlr_resamplings_bootstrap, 171, 172, 175, 177, 179, 181, 182, 184, 186, 188, 247
mlr_resamplings_custom, 171, 173, 174, 177, 179, 181, 182, 184, 186, 188, 247
mlr_resamplings_custom_cv, 171, 173, 175, 175, 179, 181, 182, 184, 186, 247
mlr_resamplings_cv, 171, 173, 175, 177, 178, 181, 182, 184, 186, 188, 247
mlr_resamplings_holdout, 171, 173, 175, 177, 179, 179, 182, 184, 186, 247
mlr_resamplings_insample, 171, 173, 175, 177, 179, 181, 184, 186, 188, 247
mlr_resamplings_loo, 171, 173, 175–177, 179, 181, 182, 183, 186, 188, 247
mlr_resamplings_repeated_cv, 171, 173, 175, 175, 177, 179, 181, 182, 184, 185, 247
mlr_resamplings_subsampling, 171, 173, 175, 175, 177, 179, 181, 182, 184, 186, 187, 247
mlr_sugar, 189
mlr_task_generators, 76, 91, 171, 189, 191, 206, 207–222, 263, 265
mlr_task_generators_2dnormals, 207, 207, 210, 212, 213, 215, 217, 219, 220, 222, 265
mlr_task_generators_cassini, 207, 208, 209, 212, 213, 215, 217, 219, 220, 222, 265
mlr_task_generators_circle, 207, 208, 210, 211, 213, 215, 217, 219, 220, 222, 265
mlr_task_generators_friedman1, 207, 208, 210, 212, 212, 215, 217, 219, 220, 222, 265
mlr_task_generators_moons, 207, 208, 210,
msr (mlr_sugar), 189
msrs (mlr_sugar), 189
msrs(), 90, 91

order(), 252

palmerpenguins::penguins, 198
paradox::ParamSet, 50, 51, 53, 54, 58, 61, 63, 65, 68, 71, 73, 190, 245, 246, 263, 264
parallelly::availableCores(), 249
ParamSet, 51
partition, 225
partition(), 55
penguins, 249
plot(), 208, 210, 212, 215, 217, 218, 220, 222
precision, 105
precision(), 105
predict.Learner, 226
PredictionClassif, 13, 57, 139, 227, 229, 230, 235
PredictionData, 14, 233, 234
PredictionRegr, 15, 60, 227, 229, 232, 234
progressr::handler(), 27, 238
progressr::with_progress(), 27, 238

R6, 30, 39, 41, 43, 47, 53, 57, 60, 64, 68, 70, 73, 78, 80, 83, 85, 87, 89, 92, 93, 102, 140, 142, 143, 168, 172, 174, 176, 178, 180, 182, 183, 185, 188, 208, 209, 211, 213, 214, 216, 218, 220, 221, 231, 234, 240, 246, 254, 261, 264, 265
R6::R6Class, 75, 90, 171, 190–194, 196–198, 200–202, 204–206
recall, 105
recall(), 105
regr.mse, 70
regr.rpart, 50
remotes::install_cran(), 49
remotes::install_github(), 49
requireNamespace(). 54, 58, 61, 66, 69, 71, 74, 264
resample, 236, 243
resample(). 7, 12, 32, 46, 52, 55, 63, 65, 66, 69, 71, 74, 239, 240, 244
ResampleResult, 12, 16–18, 26, 28–30, 32, 33, 52, 55, 63–65, 67–69, 71, 73, 74, 236–239, 239, 240, 242
ResamplingBootstrap, 244
ResamplingBootstrap
(mlr_resamplings_bootstrap), 172
ResamplingCustom
(mlr_resamplings_custom), 174
ResamplingCustomCV
(mlr_resamplings_custom_cv), 175
ResamplingCV, 244, 246
ResamplingCV(mlr_resamplings_cv), 178
ResamplingHoldout, 246
ResamplingHoldout
(mlr_resamplings_holdout), 179
ResamplingInsample
(mlr_resamplings_insampling), 181
ResamplingLOO(mlr_resamplings_loo), 183
ResamplingRepeatedCV
(mlr_resamplings_repeated_cv), 185
Resamplings, 173, 175, 177, 179, 181, 182, 184, 186, 188, 247
ResamplingSubsampling
(mlr_resamplings_subsampling), 187
ResultData, 17, 240
rpart::rpart(), 82, 88
rse(), 161
rsmp(mlr_sugar), 189
rsmp(). 171, 172, 174, 176, 178, 180, 181, 183, 185, 187
rsmps(mlr_sugar), 189
rsmps(). 171
sd(), 86
set_threads, 248
setdiff(), 252
split(), 176
stats::AIC(), 91
stats::BIC(), 93
stats::cor(), 145, 165
stats::logLik(), 50
stats::model.matrix(), 255
stats::predict(), 226
summary(), 250

TaskClassif, 19, 21, 22, 191, 193, 194, 196–206, 231, 249, 254, 260, 262, 267
TaskGenerator, 48, 189, 206–222, 263
TaskGenerator2DNormals
(mlr_task_generators_2dnormals), 207
TaskGeneratorCassini
(mlr_task_generators_cassini), 209
TaskGeneratorCircle
(mlr_task_generators_circle), 211
TaskGeneratorFriedman
(mlr_task_generators_friedman), 212
TaskGeneratorMoons
(mlr_task_generators_moons), 214
TaskGenerators, 208, 210, 212, 213, 215, 217, 219, 220, 222, 265
TaskGeneratorSimplex
(mlr_task_generators_simplex), 216
TaskGeneratorSmiley
(mlr_task_generators_smiley), 217
TaskGeneratorSpirals
(mlr_task_generators_spirals), 219
TaskGeneratorXor
(mlr_task_generators_xor), 221
Tasks. 192, 194, 195, 197–200, 202, 203, 205, 206, 259, 262, 266
TaskSupervised, 191, 193, 194, 196–199, 201–203, 205, 206, 249, 260, 262, 265, 267
TaskUnsupervised, 24, 191, 193, 194, 196–199, 201–203, 205, 206, 249, 260, 262, 265, 267
tgen (mlr_sugar), 189
tgen(). 206, 207, 209, 211, 213, 214, 216, 218, 219, 221
tgens (mlr_sugar), 189
tgens(). 206, 207
time_train. 62
tsk (mlr_sugar), 189
tsk(). 190, 191, 193, 194, 196, 198, 200–202, 204, 205
tsks (mlr_sugar), 189
tsks(). 190, 191
union(). 252
unmarshal_model(). 31, 78, 243
xor. 263