Package `mlr3tuning`

June 29, 2024

Title Hyperparameter Optimization for 'mlr3'

Version 1.0.0

Description Hyperparameter optimization package of the 'mlr3' ecosystem. It features highly configurable search spaces via the 'paradox' package and finds optimal hyperparameter configurations for any 'mlr3' learner. 'mlr3tuning' works with several optimization algorithms e.g. Random Search, Iterated Racing, Bayesian Optimization (in 'mlr3mbo') and Hyperband (in 'mlr3hyperband'). Moreover, it can automatically optimize learners and estimate the performance of optimized models with nested resampling.

License LGPL-3

URL https://mlr3tuning.mlr-org.com,
    https://github.com/mlr-org/mlr3tuning

BugReports https://github.com/mlr-org/mlr3tuning/issues

Depends mlr3 (>= 0.20.0), paradox (>= 1.0.0), R (>= 3.1.0)

Imports bbotk (>= 1.0.0), checkmate (>= 2.0.0), data.table, lgr,
       mlr3misc (>= 0.15.1), R6

Suggests adagio, future, GenSA, irace, knitr, mlflow, mlr3learners (>= 0.7.0), mlr3pipelines (>= 0.5.2), nloptr, rush, rmarkdown,
       rpart, testthat (>= 3.0.0), xgboost

VignetteBuilder knitr

Config/testthat/edition 3

Config/testthat/parallel false

Encoding UTF-8

NeedsCompilation no

RoxygenNote 7.3.1

Collate 'ArchiveAsyncTuning.R' 'ArchiveBatchTuning.R' 'AutoTuner.R'
    'CallbackAsyncTuning.R' 'CallbackBatchTuning.R'
    'ContextAsyncTuning.R' 'ContextBatchTuning.R'
    'ObjectiveTuning.R' 'ObjectiveTuningAsync.R'
## Contents

- `ObjectiveTuningBatch.R` `mlr_tuners.R` `Tuner.R`
- `TunerAsync.R` `TunerAsyncDesignPoints.R`
- `TunerAsyncFromOptimizerAsync.R` `TunerAsyncGridSearch.R`
- `TunerAsyncRandomSearch.R` `TunerBatch.R` `TunerBatchCmaes.R`
- `TunerBatchDesignPoints.R` `TunerBatchFromBatchOptimizer.R`
- `TunerBatchGenSA.R` `TunerBatchGridSearch.R`
- `TunerBatchInternal.R` `TunerBatchIrace.R` `TunerBatchNloptr.R`
- `TunerBatchRandomSearch.R` `TuningInstanceBatchSingleCrit.R`
- `TuningInstanceAsyncSingleCrit.R`
- `TuningInstanceBatchMulticrit.R` `TuningInstanceMultiCrit.R`
- `TuningInstanceSingleCrit.R` `as_search_space.R` `as_tuner.R`
- `extract_inner_tuning_archives.R`
- `extract_inner_tuning_results.R` `helper.R` `mlr_callbacks.R`

**Author**  
Marc Becker [cre, aut] (<https://orcid.org/0000-0002-8115-0400>),  
Michel Lang [aut] (<https://orcid.org/0000-0001-9754-0393>),  
Jakob Richter [aut] (<https://orcid.org/0000-0003-4481-5554>),  
Bernd Bischl [aut] (<https://orcid.org/0000-0001-6002-6980>),  
Daniel Schalk [aut] (<https://orcid.org/0000-0003-0950-1947>)

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

**Repository**  
CRAN

**Date/Publication**  
2024-06-29 06:40:11 UTC
mlr3tuning-package

mlr3tuning-package

mlr3tuning: Hyperparameter Optimization for 'mlr3'

Description

Hyperparameter optimization package of the 'mlr3' ecosystem. It features highly configurable search spaces via the 'paradox' package and finds optimal hyperparameter configurations for any 'mlr3' learner. 'mlr3tuning' works with several optimization algorithms e.g. Random Search, Iterated Racing, Bayesian Optimization (in 'mlr3mbo') and Hyperband (in 'mlr3hyperband'). Moreover, it can automatically optimize learners and estimate the performance of optimized models with nested resampling.
ArchiveAsyncTuning

Author(s)

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

Authors:

- Michel Lang <michellang@gmail.com> (ORCID)
- Jakob Richter <jakob1richter@gmail.com> (ORCID)
- Bernd Bischl <bernd_bischl@gmx.net> (ORCID)
- Daniel Schalk <daniel.schalk@stat.uni-muenchen.de> (ORCID)

See Also

Useful links:

- https://mlr3tuning.mlr-org.com
- https://github.com/mlr-org/mlr3tuning
- Report bugs at https://github.com/mlr-org/mlr3tuning/issues

Description

The ‘ArchiveAsyncTuning’ stores all evaluated hyperparameter configurations and performance scores in a rush::Rush database.

Details

The ArchiveAsyncTuning is a connector to a rush::Rush database.

Data Structure

The table ($data) has the following columns:

- One column for each hyperparameter of the search space ($search_space).
- One (list-)column for the internal_tuned_values
- One column for each performance measure ($codomain).
- x_domain (list())
  Lists of (transformed) hyperparameter values that are passed to the learner.
- runtime_learners (numeric(1))
  Sum of training and predict times logged in learners per mlr3::ResampleResult / evaluation. This does not include potential overhead time.
- timestamp (POSIXct)
  Time stamp when the evaluation was logged into the archive.
- batch_nr (integer(1))
  Hyperparameters are evaluated in batches. Each batch has a unique batch number.
Analysis

For analyzing the tuning results, it is recommended to pass the `ArchiveAsyncTuning` to `as.data.table()`. The returned data table contains the `mlr3::ResampleResult` for each hyperparameter evaluation.

S3 Methods

- `as.data.table.ArchiveTuning(x, unnest = "x_domain", exclude_columns = "uhash", measures = NULL)`
  Returns a tabular view of all evaluated hyperparameter configurations.

  - `x (ArchiveAsyncTuning)`
  - `unnest (character())`
    Transforms list columns to separate columns. Set to NULL if no column should be unnested.
  - `exclude_columns (character())`
    Exclude columns from table. Set to NULL if no column should be excluded.
  - `measures (List of mlr3::Measure)`
    Score hyperparameter configurations on additional measures.

Super classes

`bbotk::Archive` -> `bbotk::ArchiveAsync` -> `ArchiveAsyncTuning`

Active bindings

- `internal_search_space (paradox::ParamSet)`
  The search space containing those parameters that are internally optimized by the `mlr3::Learner`.
- `benchmark_result (mlr3::BenchmarkResult)`
  Benchmark result.

Methods

Public methods:

- `ArchiveAsyncTuning$new()`
- `ArchiveAsyncTuning$learner()`
- `ArchiveAsyncTuning$learners()`
- `ArchiveAsyncTuning$learner_param_vals()`
- `ArchiveAsyncTuning$predictions()`
- `ArchiveAsyncTuning$resample_result()`
- `ArchiveAsyncTuning$print()`
- `ArchiveAsyncTuning$clone()`

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

Usage:
ArchiveAsyncTuning$new(
  search_space,
  codomain,
  rush,  
  internal_search_space = NULL
)

Arguments:

search_space (paradox::ParamSet)
  Hyperparameter search space. If NULL (default), the search space is constructed from the
  paradox::TuneToken of the learner’s parameter set (learner$param_set).

codomain (bbotk::Codomain)
  Specifies codomain of objective function i.e. a set of performance measures. Internally
  created from provided mlr3::Measures.

rush (Rush)
  If a rush instance is supplied, the tuning runs without batches.

internal_search_space (paradox::ParamSet or NULL)
  The internal search space of the tuner. This includes parameters that the learner can optimize
  internally during $train(), such as the number of epochs via early stopping.

check_values (logical(1))
  If TRUE (default), hyperparameter configurations are check for validity.

Method learner(): Retrieve mlr3::Learner of the i-th evaluation, by position or by unique hash
  uhash. i and uhash are mutually exclusive. Learner does not contain a model. Use $learners() to get
  learners with models.

Usage:
ArchiveAsyncTuning$learner(i = NULL, uhash = NULL)

Arguments:

i (integer(1))
  The iteration value to filter for.

uhash (logical(1))
  The uhash value to filter for.

Method learners(): Retrieve list of trained mlr3::Learner objects of the i-th evaluation, by
  position or by unique hash uhash. i and uhash are mutually exclusive.

Usage:
ArchiveAsyncTuning$learners(i = NULL, uhash = NULL)

Arguments:

i (integer(1))
  The iteration value to filter for.

uhash (logical(1))
  The uhash value to filter for.

Method learner_param_vals(): Retrieve param values of the i-th evaluation, by position or
  by unique hash uhash. i and uhash are mutually exclusive.

Usage:
ArchiveAsyncTuning$learner_param_vals(i = NULL, uhash = NULL)

Arguments:
   i (integer(1))
       The iteration value to filter for.
   uhash (logical(1))
       The uhash value to filter for.

Method predictions(): Retrieve list of `mlr3::Prediction` objects of the i-th evaluation, by position or by unique hash uhash. i and uhash are mutually exclusive.

Usage:
ArchiveAsyncTuning$predictions(i = NULL, uhash = NULL)

Arguments:
   i (integer(1))
       The iteration value to filter for.
   uhash (logical(1))
       The uhash value to filter for.

Method resample_result(): Retrieve `mlr3::ResampleResult` of the i-th evaluation, by position or by unique hash uhash. i and uhash are mutually exclusive.

Usage:
ArchiveAsyncTuning$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.

Method print(): Printer.

Usage:
ArchiveAsyncTuning$print()

Arguments:
   ... (ignored).

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

Usage:
ArchiveAsyncTuning$clone(deep = FALSE)

Arguments:
   deep Whether to make a deep clone.
ArchiveBatchTuning Class for Logging Evaluated Hyperparameter Configurations

Description

The ArchiveBatchTuning stores all evaluated hyperparameter configurations and performance scores in a `data.table::data.table()`.

Details

The ArchiveBatchTuning is a container around a `data.table::data.table()`. Each row corresponds to a single evaluation of a hyperparameter configuration. See the section on Data Structure for more information. The archive stores additionally a `mlr3::BenchmarkResult` ($benchmark_result) that records the resampling experiments. Each experiment corresponds to a single evaluation of a hyperparameter configuration. The table ($data) and the benchmark result ($benchmark_result) are linked by the `uhash` column. If the archive is passed to `as.data.table()`, both are joined automatically.

Data Structure

The table ($data) has the following columns:

- One column for each hyperparameter of the search space ($search_space).
- One (list-)column for the `internal_tuned_values`
- One column for each performance measure ($codomain).
- `x_domain (list())`
  Lists of (transformed) hyperparameter values that are passed to the learner.
- `runtime_learners (numeric(1))`
  Sum of training and predict times logged in learners per `mlr3::ResampleResult / evaluation`. This does not include potential overhead time.
- `timestamp (POSIXct)`
  Time stamp when the evaluation was logged into the archive.
- `batch_nr (integer(1))`
  Hyperparameters are evaluated in batches. Each batch has a unique batch number.
- `uhash (character(1))`
  Connects each hyperparameter configuration to the resampling experiment stored in the `mlr3::BenchmarkResult`.

Analysis

For analyzing the tuning results, it is recommended to pass the ArchiveBatchTuning to `as.data.table()`. The returned data table is joined with the benchmark result which adds the `mlr3::ResampleResult` for each hyperparameter evaluation.

The archive provides various getters (e.g. `$learners()`) to ease the access. All getters extract by position (i) or unique hash (uhash). For a complete list of all getters see the methods section.
The benchmark result (\$benchmark_result) allows to score the hyperparameter configurations again on a different measure. Alternatively, measures can be supplied to as.data.table(). The \texttt{mlr3viz} package provides visualizations for tuning results.

**S3 Methods**

- \texttt{as.data.table.ArchiveTuning(x, unnest = "x\_domain", exclude\_columns = "uhash", measures = NULL)}
  Returns a tabular view of all evaluated hyperparameter configurations.
  \texttt{ArchiveBatchTuning -> data.table::data.table()}

  - \texttt{x (ArchiveBatchTuning)}
  - \texttt{unnest (character())}
    Transforms list columns to separate columns. Set to NULL if no column should be unnested.
  - \texttt{exclude\_columns (character())}
    Exclude columns from table. Set to NULL if no column should be excluded.
  - \texttt{measures (List of mlr3\_Measure)}
    Score hyperparameter configurations on additional measures.

**Super classes**

\texttt{bbotk::Archive -> bbotk::ArchiveBatch -> ArchiveBatchTuning}

**Public fields**

- \texttt{benchmark\_result (mlr3::BenchmarkResult)}
  Benchmark result.

**Active bindings**

- \texttt{internal\_search\_space (paradox::ParamSet)}
  The search space containing those parameters that are internally optimized by the \texttt{mlr3::Learner}.

**Methods**

**Public methods:**

- \texttt{ArchiveBatchTuning\$new()}
- \texttt{ArchiveBatchTuning\$learner()}
- \texttt{ArchiveBatchTuning\$learners()}
- \texttt{ArchiveBatchTuning\$learner\_param\_vals()}
- \texttt{ArchiveBatchTuning\$predictions()}
- \texttt{ArchiveBatchTuning\$resample\_result()}
- \texttt{ArchiveBatchTuning\$print()}
- \texttt{ArchiveBatchTuning\$clone()}

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

**Usage:**
ArchiveBatchTuning$new(
  search_space,
  codomain,
  check_values = FALSE,
  internal_search_space = NULL
)

Arguments:

- **search_space** (*paradox::ParamSet*)
  Hyperparameter search space. If NULL (default), the search space is constructed from the *paradox::TuneToken* of the learner's parameter set (learner$param_set).

- **codomain** (*bott::Codomain*)
  Specifies codomain of objective function i.e. a set of performance measures. Internally created from provided *mlr3::Measures*.

- **check_values** (*logical(1)*)
  If TRUE (default), hyperparameter configurations are check for validity.

- **internal_search_space** (*paradox::ParamSet or NULL*)
  The internal search space of the tuner. This includes parameters that the learner can optimize internally during $train(), such as the number of epochs via early stopping.

**Method** learner(): Retrieve *mlr3::Learner* of the i-th evaluation, by position or by unique hash uhash. i and uhash are mutually exclusive. Learner does not contain a model. Use $learners() to get learners with models.

*Usage:*

```r
ArchiveBatchTuning$learner(i = NULL, uhash = NULL)
```

*Arguments:*

- **i** (*integer(1)*)
  The iteration value to filter for.

- **uhash** (*logical(1)*)
  The uhash value to filter for.

**Method** learners(): Retrieve list of trained *mlr3::Learner* objects of the i-th evaluation, by position or by unique hash uhash. i and uhash are mutually exclusive.

*Usage:*

```r
ArchiveBatchTuning$learners(i = NULL, uhash = NULL)
```

*Arguments:*

- **i** (*integer(1)*)
  The iteration value to filter for.

- **uhash** (*logical(1)*)
  The uhash value to filter for.

**Method** learner_param_vals(): Retrieve param values of the i-th evaluation, by position or by unique hash uhash. i and uhash are mutually exclusive.

*Usage:*

```r
ArchiveBatchTuning$learner_param_vals(i = NULL, uhash = NULL)
```

*Arguments:*

...
i (integer(1))
  The iteration value to filter for.

uhash (logical(1))
  The uhash value to filter for.

**Method** predictions(): Retrieve list of `mldr3::Prediction` objects of the i-th evaluation, by position or by unique hash uhash. i and uhash are mutually exclusive.

*Usage:*

```r
ArchiveBatchTuning$predictions(i = NULL, uhash = NULL)
```

*Arguments:*

- **i** (integer(1))
  - The iteration value to filter for.
- **uhash** (logical(1))
  - The uhash value to filter for.

**Method** resample_result(): Retrieve `mldr3::ResampleResult` of the i-th evaluation, by position or by unique hash uhash. i and uhash are mutually exclusive.

*Usage:*

```r
ArchiveBatchTuning$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.

**Method** print(): Printer.

*Usage:*

```r
ArchiveBatchTuning$print()
```

*Arguments:*

- ... (ignored).

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

*Usage:*

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

*Arguments:*

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

Convert to a Search Space

Description
Convert object to a search space.

Usage
as_search_space(x, ...)

## S3 method for class 'Learner'
as_search_space(x, ...)

## S3 method for class 'ParamSet'
as_search_space(x, ...)

Arguments
x  
(any)
Object to convert to search space.

...  
(any)
Additional arguments.

Value
paradox::ParamSet.

as_tuner  

Convert to a Tuner

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

Usage
as_tuner(x, ...)

## S3 method for class 'Tuner'
as_tuner(x, clone = FALSE, ...)

as_tuners(x, ...)

## Default S3 method:
AutoTuner

as_tuners(x, ...)

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

Arguments

- **x** (any)
  - Object to convert.
- **...** (any)
  - Additional arguments.
- **clone** (logical(1))
  - Whether to clone the object.

AutoTuner  

Class for Automatic Tuning

Description

The **AutoTuner** wraps a mlr3::Learner and augments it with an automatic tuning process for a given set of hyperparameters. The **auto_tuner()** function creates an **AutoTuner** object.

Details

The **AutoTuner** is a mlr3::Learner which wraps another mlr3::Learner and performs the following steps during **$train()**:

1. The hyperparameters of the wrapped (inner) learner are trained on the training data via resampling. The tuning can be specified by providing a Tuner, a bbotk::Terminator, a search space as paradox::ParamSet, a mlr3::Resampling and a mlr3::Measure.
2. The best found hyperparameter configuration is set as hyperparameters for the wrapped (inner) learner stored in at$learner. Access the tuned hyperparameters via at$tuning_result.
3. A final model is fit on the complete training data using the now parametrized wrapped learner. The respective model is available via field at$learner$model.

During **$predict()** the AutoTuner just calls the predict method of the wrapped (inner) learner. A set timeout is disabled while fitting the final model.

Validation

Both, the tuned mlr3::Learner and the AutoTuner itself can make use of validation data. the $validate field of the AutoTuner determines how validation is done during the final model fit. In most cases, this should be left as NULL. The $validate field of the tuned mlr3::Learner specifies how the validation data is constructed during the hyperparameter optimization.
Resources

There are several sections about hyperparameter optimization in the mlr3book.

- Automate the tuning.
- Estimate the model performance with nested resampling.

The gallery features a collection of case studies and demos about optimization.

Nested Resampling

Nested resampling is performed by passing an AutoTuner to mlr3::resample() or mlr3::benchmark(). To access the inner resampling results, set store_tuning_instance = TRUE and execute mlr3::resample() or mlr3::benchmark() with store_models = TRUE (see examples). The mlr3::Resampling passed to the AutoTuner is meant to be the inner resampling, operating on the training set of an arbitrary outer resampling. For this reason, the inner resampling should be not instantiated. If an instantiated resampling is passed, the AutoTuner fails when a row id of the inner resampling is not present in the training set of the outer resampling.

Default Measures

If no measure is passed, the default measure is used. The default measure depends on the task type.

<table>
<thead>
<tr>
<th>Task</th>
<th>Default Measure</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;classif&quot;</td>
<td>&quot;classif.ce&quot;</td>
<td>mlr3</td>
</tr>
<tr>
<td>&quot;regr&quot;</td>
<td>&quot;regr.mse&quot;</td>
<td>mlr3</td>
</tr>
<tr>
<td>&quot;surv&quot;</td>
<td>&quot;surv.cindex&quot;</td>
<td>mlr3proba</td>
</tr>
<tr>
<td>&quot;dens&quot;</td>
<td>&quot;dens.logloss&quot;</td>
<td>mlr3proba</td>
</tr>
<tr>
<td>&quot;classif_st&quot;</td>
<td>&quot;classif.ce&quot;</td>
<td>mlr3spatial</td>
</tr>
<tr>
<td>&quot;regr_st&quot;</td>
<td>&quot;regr.mse&quot;</td>
<td>mlr3spatial</td>
</tr>
<tr>
<td>&quot;clust&quot;</td>
<td>&quot;clust.dunn&quot;</td>
<td>mlr3cluster</td>
</tr>
</tbody>
</table>

Super class

mlr3::Learner -> AutoTuner

Public fields

- instance_args (list())
  All arguments from construction to create the TuningInstanceBatchSingleCrit.
- tuner (Tuner)
  Optimization algorithm.

Active bindings

- internal_valid_scores Retrieves the inner validation scores as a named list(). Returns NULL if learner is not trained yet.
- validate How to construct the internal validation data. This parameter can be either NULL, a ratio in $(0, 1)$, "test", or "predefined".
AutoTuner

archive ArchiveBatchTuning Archive of the TuningInstanceBatchSingleCrit.
learner (mlr3::Learner) Trained learner
tuning_instance (TuningInstanceAsyncSingleCrit | TuningInstanceBatchSingleCrit) Internally created tuning instance with all intermediate results.
tuning_result (data.table::data.table) Short-cut to result from tuning instance.
predict_type (character(1)) Stores the currently active predict type, e.g. "response". Must be an element of $predict_types.
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).

Methods

Public methods:

• AutoTuner$new()
• AutoTuner$base_learner()
• AutoTuner$importance()
• AutoTuner$selected_features()
• AutoTuner$oob_error()
• AutoTuner$loglik()
• AutoTuner$print()
• AutoTuner$marshal()
• AutoTuner$unmarshal()
• AutoTuner$marshaled()
• AutoTuner$clone()

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

Usage:
AutoTuner$new(
tuner,
learner,
resampling,
measure = NULL,
terminator,
search_space = NULL,
store_tuning_instance = TRUE,
store_benchmark_result = TRUE,
store_models = FALSE,
check_values = FALSE,
callbacks = NULL,
AutoTuner

rash = NULL,
validate = NULL
)

Arguments:
tuner (Tuner)
  Optimization algorithm.
learner (mlr3::Learner)
  Learner to tune.
resampling (mlr3::Resampling)
  Resampling that is used to evaluate the performance of the hyperparameter configurations. Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits. Already instantiated resamplings are kept unchanged. Specialized Tuner change the resampling e.g. to evaluate a hyperparameter configuration on different data splits. This field, however, always returns the resampling passed in construction.
measure (mlr3::Measure)
  Measure to optimize. If NULL, default measure is used.
terminator (bbotk::Terminator)
  Stop criterion of the tuning process.
search_space (paradox::ParamSet)
  Hyperparameter search space. If NULL (default), the search space is constructed from the paradox::TuneToken of the learner's parameter set (learner$param_set).
store_tuning_instance (logical(1))
  If TRUE (default), stores the internally created TuningInstanceBatchSingleCrit with all intermediate results in slot $tuning_instance.
store_benchmark_result (logical(1))
  If TRUE (default), store resample result of evaluated hyperparameter configurations in archive as mlr3::BenchmarkResult.
store_models (logical(1))
  If TRUE, fitted models are stored in the benchmark result (archive$benchmark_result). If store_benchmark_result = FALSE, models are only stored temporarily and not accessible after the tuning. This combination is needed for measures that require a model.
check_values (logical(1))
  If TRUE, hyperparameter values are checked before evaluation and performance scores after. If FALSE (default), values are unchecked but computational overhead is reduced.
callbacks (list of mlr3misc::Callback)
  List of callbacks.
rush (Rush)
  If a rush instance is supplied, the tuning runs without batches.
validate (numeric(1), "test", "predefined" or NULL)
  How to construct the internal validation data.

Method base_learner(): Extracts the base learner from nested learner objects like GraphLearner in mlr3pipelines. If recursive = 0, the (tuned) learner is returned.
Usage:
AutoTuner$base_learner(recursive = Inf)
Arguments:
recursive (integer(1))
   Depth of recursion for multiple nested objects.

Returns: mlr3::Learner.

Method importance(): The importance scores of the final model.
Usage:
AutoTuner$importance()
Returns: Named numeric().

Method selected_features(): The selected features of the final model.
Usage:
AutoTuner$selected_features()
Returns: character().

Method oob_error(): The out-of-bag error of the final model.
Usage:
AutoTuner$oob_error()
Returns: numeric(1).

Method loglik(): The log-likelihood of the final model.
Usage:
AutoTuner$loglik()
Returns: logLik. Printer.

Method print():
Usage:
AutoTuner$print()
Arguments:
... (ignored).

Method marshal(): Marshal the learner.
Usage:
AutoTuner$marshal(...)
Arguments:
... (any)
   Additional parameters.
Returns: self

Method unmarshal(): Unmarshal the learner.
Usage:
AutoTuner$unmarshal(...)
Additional parameters.

Returns: self

Method `marshaled()`: Whether the learner is marshaled.

Usage:
AutoTuner$marshaled()

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

Usage:
AutoTuner$clone(deep = FALSE)

Arguments:
depth Whether to make a deep clone.

Examples

```r
# Automatic Tuning

# split to train and external set
task = tsk("penguins")
split = partition(task, ratio = 0.8)

# load learner and set search space
learner = lrn("classif.rpart",
  cp = to_tune(1e-04, 1e-1, logscale = TRUE)
)

# create auto tuner
at = auto_tuner(
  tuner = tnr("random_search"),
  learner = learner,
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  term_evals = 4)

# tune hyperparameters and fit final model
at$train(task, row_ids = split$train)

# predict with final model
at$predict(task, row_ids = split$test)

# show tuning result
at$tuning_result

# model slot contains trained learner and tuning instance
at$model

# shortcut trained learner
at$learner
```
auto_tuner

Function for Automatic Tuning

Description

The AutoTuner wraps a mlr3::Learner and augments it with an automatic tuning process for a given set of hyperparameters. The auto_tuner() function creates an AutoTuner object.

Usage

auto_tuner(  
tuner,  
learner,  
resampling,  
measure = NULL,  
term_evals = NULL,  
term_time = NULL,  
terminator = NULL,  
search_space = NULL,  
store_tuning_instance = TRUE,  
store_benchmark_result = TRUE,  
store_models = FALSE,  
check_values = FALSE,  
callbacks = NULL,
validate = NULL,
rush = NULL
)

Arguments

tuner (Tuner)
Optimization algorithm.

learner (mlr3::Learner)
Learner to tune.

resampling (mlr3::Resampling)
Resampling that is used to evaluate the performance of the hyperparameter configurations. Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits. Already instantiated resamplings are kept unchanged. Specialized Tuner change the resampling e.g. to evaluate a hyperparameter configuration on different data splits. This field, however, always returns the resampling passed in construction.

measure (mlr3::Measure)
Measure to optimize. If NULL, default measure is used.

term_evals (integer(1))
Number of allowed evaluations. Ignored if terminator is passed.

term_time (integer(1))
Maximum allowed time in seconds. Ignored if terminator is passed.

terminator (bbotk::Terminator)
Stop criterion of the tuning process.

search_space (paradox::ParamSet)
Hyperparameter search space. If NULL (default), the search space is constructed from the paradox::TuneToken of the learner’s parameter set (learner$param_set).

store_tuning_instance (logical(1))
If TRUE (default), stores the internally created TuningInstanceBatchSingleCrit with all intermediate results in slot $tuning_instance.

store_benchmark_result (logical(1))
If TRUE (default), store resample result of evaluated hyperparameter configurations in archive as mlr3::BenchmarkResult.

store_models (logical(1))
If TRUE, fitted models are stored in the benchmark result (archive$benchmark_result). If store_benchmark_result = FALSE, models are only stored temporarily and not accessible after the tuning. This combination is needed for measures that require a model.

check_values (logical(1))
If TRUE, hyperparameter values are checked before evaluation and performance scores after. If FALSE (default), values are unchecked but computational overhead is reduced.
callbacks (list of mlr3misc::Callback)
List of callbacks.

validate (numeric(1), "test", "predefined" or NULL)
How to construct the internal validation data.

rush (Rush)
If a rush instance is supplied, the tuning runs without batches.

Details

The AutoTuner is a mlr3::Learner which wraps another mlr3::Learner and performs the following steps during $train()$:

1. The hyperparameters of the wrapped (inner) learner are trained on the training data via resampling. The tuning can be specified by providing a Tuner, a bbotk::Terminator, a search space as paradox::ParamSet, a mlr3::Resampling and a mlr3::Measure.
2. The best found hyperparameter configuration is set as hyperparameters for the wrapped (inner) learner stored in $\text{learner.learner}$. Access the tuned hyperparameters via $\text{tuning_result}$.
3. A final model is fit on the complete training data using the now parametrized wrapped learner. The respective model is available via field $\text{learner.model}$.

During $\text{predict()}$ the AutoTuner just calls the predict method of the wrapped (inner) learner. A set timeout is disabled while fitting the final model.

Value

AutoTuner.

Default Measures

If no measure is passed, the default measure is used. The default measure depends on the task type.

<table>
<thead>
<tr>
<th>Task</th>
<th>Default Measure</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;classif&quot;</td>
<td>&quot;classif.ce&quot;</td>
<td>mlr3</td>
</tr>
<tr>
<td>&quot;regr&quot;</td>
<td>&quot;regr.mse&quot;</td>
<td>mlr3</td>
</tr>
<tr>
<td>&quot;surv&quot;</td>
<td>&quot;surv.cindex&quot;</td>
<td>mlr3proba</td>
</tr>
<tr>
<td>&quot;dens&quot;</td>
<td>&quot;dens.logloss&quot;</td>
<td>mlr3proba</td>
</tr>
<tr>
<td>&quot;classif_st&quot;</td>
<td>&quot;classif.ce&quot;</td>
<td>mlr3spatial</td>
</tr>
<tr>
<td>&quot;regr_st&quot;</td>
<td>&quot;regr.mse&quot;</td>
<td>mlr3spatial</td>
</tr>
<tr>
<td>&quot;clust&quot;</td>
<td>&quot;clust.dunn&quot;</td>
<td>mlr3cluster</td>
</tr>
</tbody>
</table>

Resources

There are several sections about hyperparameter optimization in the mlr3book.

- Automate the tuning.
- Estimate the model performance with nested resampling.

The gallery features a collection of case studies and demos about optimization.
Nested Resampling

Nested resampling is performed by passing an `AutoTuner` to `mlr3::resample()` or `mlr3::benchmark()`. To access the inner resampling results, set `store_tuning_instance = TRUE` and execute `mlr3::resample()` or `mlr3::benchmark()` with `store_models = TRUE` (see examples). The `mlr3::Resampling` passed to the `AutoTuner` is meant to be the inner resampling, operating on the training set of an arbitrary outer resampling. For this reason, the inner resampling should be not instantiated. If an instantiated resampling is passed, the `AutoTuner` fails when a row id of the inner resampling is not present in the training set of the outer resampling.

Examples

```r
at = auto_tuner(
  tuner = tnr("random_search"),
  learner = lrn("classif.rpart", cp = to_tune(1e-04, 1e-1, logscale = TRUE)),
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  term_evals = 4)

at$train(tsk("pima"))
```

---

### CallbackAsyncTuning

**Create Asynchronous Tuning Callback**

**Description**

Specialized `bbotk::CallbackAsync` for asynchronous tuning. Callbacks allow to customize the behavior of processes in `mlr3tuning`. The `callback_async_tuning()` function creates a `CallbackAsyncTuning`. Predefined callbacks are stored in the dictionary `mlr_callbacks` and can be retrieved with `clbk()`. For more information on tuning callbacks see `callback_async_tuning()`.

**Super classes**

`mlr3misc::Callback` -> `bbotk::CallbackAsync` -> `CallbackAsyncTuning`

**Public fields**

- `on_eval_after_xs (function())`
  Stage called after xs is passed. Called in `ObjectiveTuning$eval()`.

- `on_eval_after_resample (function())`
  Stage called after hyperparameter configurations are evaluated. Called in `ObjectiveTuning$eval()`.

- `on_eval_before_archive (function())`
  Stage called before performance values are written to the archive. Called in `ObjectiveTuning$eval()`.
CallbackBatchTuning

Methods

Public methods:

• CallbackAsyncTuning$clone()

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

Usage:
CallbackAsyncTuning$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

CallbackBatchTuning Create Batch Tuning Callback

Description

Specialized bbotk::CallbackBatch for batch tuning. Callbacks allow to customize the behavior of processes in mlr3tuning. The callback_batch_tuning() function creates a CallbackBatchTuning. Predefined callbacks are stored in the dictionary mlr_callbacks and can be retrieved with clbk(). For more information on tuning callbacks see callback_batch_tuning().

Super classes

mlr3misc::Callback -> bbotk::CallbackBatch -> CallbackBatchTuning

Public fields

on_eval_after_design (function())
Stage called after design is created. Called in ObjectiveTuning$eval_many().

on_eval_after_benchmark (function())
Stage called after hyperparameter configurations are evaluated. Called in ObjectiveTuning$eval_many().

on_eval_before_archive (function())
Stage called before performance values are written to the archive. Called in ObjectiveTuning$eval_many().

Methods

Public methods:

• CallbackBatchTuning$clone()

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

Usage:
CallbackBatchTuning$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.
Examples

```r
# write archive to disk
callback_batch_tuning("mlr3tuning.backup",
   on_optimization_end = function(callback, context) {
     saveRDS(context$instance$archive, "archive.rds")
   }
)
```

Description

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

Tuning callbacks can be called from different stages of the tuning process. The stages are prefixed with `on_*`.

Start Tuning
- `on_optimization_begin`

Start Worker
- `on_worker_begin`

Start Evaluation
- `on_eval_after_xs`
- `on_eval_after_resample`
- `on_eval_before_archive`

End Evaluation
- `on_worker_end`

End Worker
- `on_result`
- `on_optimization_end`

End Tuning

See also the section on parameters for more information on the stages. A tuning callback works with `ContextAsyncTuning`.

Usage

```r
callback_async_tuning(
   id, 
   label = NA_character_,
   man = NA_character_,
   on_optimization_begin = NULL,
   on_worker_begin = NULL,
   on_eval_after_xs = NULL,
   on_eval_after_resample = NULL,
   on_eval_before_archive = NULL,
)```
callback_async_tuning

```r
don_eval_before_archive = NULL,
don_worker_end = NULL,
don_result = NULL,
don_optimization_end = NULL
)
```

**Arguments**

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

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

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

- **on_optimization_begin** (function())
  Stage called at the beginning of the optimization. Called in `Optimizer$optimize()`.

- **on_worker_begin** (function())
  Stage called at the beginning of the optimization on the worker. Called in the worker loop.

- **on_eval_after_xs** (function())
  Stage called after `xs` is passed. Called in `ObjectiveTuning$eval()`.

- **on_eval_after_resample** (function())
  Stage called after a hyperparameter configuration is evaluated. Called in `ObjectiveTuning$eval()`.

- **on_eval_before_archive** (function())
  Stage called before performance values are written to the archive. Called in `ObjectiveTuning$eval()`.

- **on_worker_end** (function())
  Stage called at the end of the optimization on the worker. Called in the worker loop.

- **on_result** (function())
  Stage called after the result is written. Called in `OptimInstance$assign_result()`.

- **on_optimization_end** (function())
  Stage called at the end of the optimization. Called in `Optimizer$optimize()`.

**Details**

When implementing a callback, each function must have two arguments named `callback` and `context`. A callback can write data to the state (`$state`), e.g. settings that affect the callback itself. Tuning callbacks access `ContextAsyncTuning`. 
callback_batch_tuning  Create Batch Tuning Callback

Description

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

Tuning callbacks can be called from different stages of the tuning process. The stages are prefixed with on_*.

Start Tuning
- on_optimization_begin
  Start Tuner Batch
  - on_optimizer_before_eval
    Start Evaluation
    - on_eval_after_design
    - on_eval_after_benchmark
    - on_eval_before_archive
    End Evaluation
    - on_optimizer_after_eval
  End Tuner Batch
  - on_result
  - on_optimization_end
End Tuning

See also the section on parameters for more information on the stages. A tuning callback works with ContextBatchTuning.

Usage

callback_batch_tuning(  id,  label = NA_character_,  man = NA_character_,  on_optimization_begin = NULL,  on_optimizer_before_eval = NULL,  on_eval_after_design = NULL,  on_eval_after_benchmark = NULL,  on_eval_before_archive = NULL,  on_optimizer_after_eval = NULL,  on_result = NULL,  on_optimization_end = NULL)
callback_batch_tuning

Arguments

**id**
(\texttt{character(1)})
Identifier for the new instance.

**label**
(\texttt{character(1)})
Label for the new instance.

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

**on\_optimization\_begin**
(function())
Stage called at the beginning of the optimization. Called in \texttt{Optimizer$optimize()}."

**on\_optimizer\_before\_eval**
(function())
Stage called after the optimizer proposes points. Called in \texttt{OptimInstance$eval\_batch()}."

**on\_eval\_after\_design**
(function())
Stage called after the design is created. Called in \texttt{ObjectiveTuning$eval\_many()}. The context available is \texttt{ContextBatchTuning}.

**on\_eval\_after\_benchmark**
(function())
Stage called after hyperparameter configurations are evaluated. Called in \texttt{ObjectiveTuning$eval\_many()}. The context available is \texttt{ContextBatchTuning}.

**on\_eval\_before\_archive**
(function())
Stage called before performance values are written to the archive. Called in \texttt{ObjectiveTuning$eval\_many()}. The context available is \texttt{ContextBatchTuning}.

**on\_optimizer\_after\_eval**
(function())
Stage called after points are evaluated. Called in \texttt{OptimInstance$eval\_batch()}."

**on\_result**
(function())
Stage called after the result is written. Called in \texttt{OptimInstance$assign\_result()}."

**on\_optimization\_end**
(function())
Stage called at the end of the optimization. Called in \texttt{Optimizer$optimize()}."

Details

When implementing a callback, each function must have two arguments named \texttt{callback} and \texttt{context}. A callback can write data to the state ($\texttt{state}$), e.g. settings that affect the callback itself. Tuning callbacks access \texttt{ContextBatchTuning}.

Examples

```r
# write archive to disk
callback_batch_tuning("mlr3tuning.backup",
```
ContextAsyncTuning

Asynchronous Tuning Context

Description

A CallbackAsyncTuning accesses and modifies data during the optimization via the ContextAsyncTuning. See the section on active bindings for a list of modifiable objects. See callback_async_tuning() for a list of stages that access ContextAsyncTuning.

Details

Changes to $instance and $optimizer in the stages executed on the workers are not reflected in the main process.

Super classes

mlr3misc::Context -> bbotk::ContextAsync -> ContextAsyncTuning

Active bindings

xs (list())
  The hyperparameter configuration currently evaluated. Contains the values on the learner scale i.e. transformations are applied.
resample_result (mlr3::BenchmarkResult)
  The resample result of the hyperparameter configuration currently evaluated.
aggregated_performance (list())
  Aggregated performance scores and training time of the evaluated hyperparameter configuration. This list is passed to the archive. A callback can add additional elements which are also written to the archive.

Methods

Public methods:
  * ContextAsyncTuning$clone()

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

Usage:
ContextAsyncTuning$clone(deep = FALSE)

Arguments:
  deep  Whether to make a deep clone.
**ContextBatchTuning**  

**Batch Tuning Context**

### Description

A **CallbackBatchTuning** accesses and modifies data during the optimization via the `ContextBatchTuning`. See the section on active bindings for a list of modifiable objects. See `callback_batch_tuning()` for a list of stages that access `ContextBatchTuning`.

### Super classes

`mlr3misc::Context -> bbotk::ContextBatch -> ContextBatchTuning`

### Active bindings

- `xss` *(list())*
  - The hyperparameter configurations of the latest batch. Contains the values on the learner scale i.e. transformations are applied. See `$xdt` for the untransformed values.

- `design` *(data.table::data.table)*
  - The benchmark design of the latest batch.

- `benchmark_result` *(mlr3::BenchmarkResult)*
  - The benchmark result of the latest batch.

- `aggregated_performance` *(data.table::data.table)*
  - Aggregated performance scores and training time of the latest batch. This data table is passed to the archive. A callback can add additional columns which are also written to the archive.

### Methods

**Public methods:**

- `ContextBatchTuning$clone()`

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

**Usage:**

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

**Arguments:**

- `deep` Whether to make a deep clone.
**Extract Inner Tuning Archives**

**Description**

Extract inner tuning archives of nested resampling. Implemented for `mlr3::ResampleResult` and `mlr3::BenchmarkResult`. The function iterates over the `AutoTuner` objects and binds the tuning archives to a `data.table::data.table()`. `AutoTuner` must be initialized with `store_tuning_instance = TRUE` and `mlr3::resample()` or `mlr3::benchmark()` must be called with `store_models = TRUE`.

**Usage**

```r
extract_inner_tuning_archives(
  x,
  unnest = "x_domain",
  exclude_columns = "uhash"
)
```

**Arguments**

- `x` *(mlr3::ResampleResult | mlr3::BenchmarkResult)*
- `unnest` *(character())*
  Transforms list columns to separate columns. By default, `x_domain` is unnested. Set to `NULL` if no column should be unnested.
- `exclude_columns` *(character())*
  Exclude columns from result table. Set to `NULL` if no column should be excluded.

**Value**

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

**Data structure**

The returned data table has the following columns:

- `experiment` *(integer(1))*
  Index, giving the according row number in the original benchmark grid.
- `iteration` *(integer(1))*
  Iteration of the outer resampling.
- One column for each hyperparameter of the search spaces.
- One column for each performance measure.
- `runtime_learners` *(numeric(1))*
  Sum of training and predict times logged in learners per `mlr3::ResampleResult` / evaluation. This does not include potential overhead time.
• timestamp (POSIXct)
  Time stamp when the evaluation was logged into the archive.
• batch_nr (integer(1))
  Hyperparameters are evaluated in batches. Each batch has a unique batch number.
• x_domain (list())
  List of transformed hyperparameter values. By default this column is unnested.
• x_domain_* (any)
  Separate column for each transformed hyperparameter.
• resample_result (mldr3::ResampleResult)
  Resample result of the inner resampling.
• task_id (character(1)).
• learner_id (character(1)).
• resampling_id (character(1)).

Examples

# Nested Resampling on Palmer Penguins Data Set
learner = lrn("classif.rpart", 
  cp = to_tune(1e-04, 1e-1, logscale = TRUE))

# create auto tuner
at = auto_tuner(
  tuner = tnr("random_search"),
  learner = learner,
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  term_evals = 4)

resampling_outer = rsmp("cv", folds = 2)
rr = resample(tsk("iris"), at, resampling_outer, store_models = TRUE)

# extract inner archives
extract_inner_tuning_archives(rr)

extract_inner_tuning_results

Extract Inner Tuning Results

Description

Extract inner tuning results of nested resampling. Implemented for mldr3::ResampleResult and mldr3::BenchmarkResult.
Usage
extract_inner_tuning_results(x, tuning_instance, ...)

## S3 method for class 'ResampleResult'
extract_inner_tuning_results(x, tuning_instance = FALSE, ...)

## S3 method for class 'BenchmarkResult'
extract_inner_tuning_results(x, tuning_instance = FALSE, ...)

Arguments

x (mlr3::ResampleResult | mlr3::BenchmarkResult).
tuning_instance
  (logical(1))
  If TRUE, tuning instances are added to the table.
...
  (any)
  Additional arguments.

Details
The function iterates over the AutoTuner objects and binds the tuning results to a data.table::data.table(). The AutoTuner must be initialized with store_tuning_instance = TRUE and mlr3::resample() or mlr3::benchmark() must be called with store_models = TRUE. Optionally, the tuning instance can be added for each iteration.

Value
data.table::data.table().

Data structure
The returned data table has the following columns:

- experiment (integer(1))
  Index, giving the according row number in the original benchmark grid.
- iteration (integer(1))
  Iteration of the outer resampling.
- One column for each hyperparameter of the search spaces.
- One column for each performance measure.
- learner_param_vals (list())
  Hyperparameter values used by the learner. Includes fixed and proposed hyperparameter values.
- x_domain (list())
  List of transformed hyperparameter values.
- tuning_instance (TuningInstanceBatchSingleCrit | TuningInstanceBatchMultiCrit)
  Optionally, tuning instances.
- task_id (character(1)).
• learner_id(character(1)).
• resampling_id(character(1)).

Examples

# Nested Resampling on Palmer Penguins Data Set

learner = lrn("classif.rpart",
               cp = to_tune(1e-04, 1e-1, logscale = TRUE))

# create auto tuner
at = auto_tuner(
               tuner = tnr("random_search"),
               learner = learner,
               resampling = rsmp("holdout"),
               measure = msr("classif.ce"),
               term_evals = 4)

resampling_outer = rsmp("cv", folds = 2)
rr = resample(tsk("iris"), at, resampling_outer, store_models = TRUE)

# extract inner results
extract_inner_tuning_results(rr)

mlr3tuning.asnyc_mlflow

MLflow Connector Callback

Description

This mlr3misc::Callback logs the hyperparameter configurations and the performance of the configurations to MLflow.

Examples

clbk("mlr3tuning.async_mlflow", tracking_uri = "http://localhost:5000")

## Not run:
rush::rush_plan(n_workers = 4)

learner = lrn("classif.rpart",
              minsplit = to_tune(2, 128),
              cp = to_tune(1e-04, 1e-1))

instance = TuningInstanceAsyncSingleCrit$new(
            task = tsk("pima"),
            learner = learner,
            resampling = rsmp("cv", folds = 3),
measure = msr("classif.ce"),
terminator = trm("evals", n_evals = 20),
store_benchmark_result = FALSE,
callbacks = clbk("mlr3tuning.rush_mlflow", tracking_uri = "http://localhost:8080")
)
tuner = tnr("random_search_v2")
tuner$optimize(instance)

## End(Not run)

### mlr3tuning.async_default_configuration

**Default Configuration Callback**

**Description**

These `CallbackAsyncTuning` and `CallbackBatchTuning` evaluate the default hyperparameter values of a learner.

### mlr3tuning.async_save_logs

**Save Logs Callback**

**Description**

This `CallbackAsyncTuning` saves the logs of the learners to the archive.

### mlr3tuning.backup

**Backup Benchmark Result Callback**

**Description**

This `mlr3misc::Callback` writes the `mlr3::BenchmarkResult` after each batch to disk.
Examples

clbk("mlr3tuning.backup", path = "backup.rds")

# tune classification tree on the pima data set
instance = tune(
  tuner = tnr("random_search", batch_size = 2),
  task = tsk("pima"),
  learner = lrn("classif.rpart", cp = to_tune(1e-04, 1e-1, logscale = TRUE)),
  resampling = rsmp("cv", folds = 3),
  measures = msr("classif.ce"),
  term_evals = 4,
  callbacks = clbk("mlr3tuning.backup", path = tempfile(fileext = ".rds"))
)

mlr3tuning.measures | Measure Callback
--- | ---

Description

This `mlr3misc::Callback` scores the hyperparameter configurations on additional measures while tuning. Usually, the configurations can be scored on additional measures after tuning (see `ArchiveBatchTuning`). However, if the memory is not sufficient to store the `mlr3::BenchmarkResult`, it is necessary to score the additional measures while tuning. The measures are not taken into account by the tuner.

Examples

clbk("mlr3tuning.measures")

# additionally score the configurations on the accuracy measure
instance = tune(
  tuner = tnr("random_search", batch_size = 2),
  task = tsk("pima"),
  learner = lrn("classif.rpart", cp = to_tune(1e-04, 1e-1, logscale = TRUE)),
  resampling = rsmp("cv", folds = 3),
  measures = msr("classif.ce"),
  term_evals = 4,
  callbacks = clbk("mlr3tuning.measures", measures = msr("classif.acc"))
)
Dictionary of Tuners

Description

A simple `mlr3misc::Dictionary` storing objects of class `Tuner`. Each tuner has an associated help page, see `mlr_tuners_[id]`.

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

For a more convenient way to retrieve and construct tuner, see `tnr()`/`tnrs()`.

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", "param_classes", "properties" 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: `tnr()`, `tnrs()`

Other Tuner: `Tuner`, `mlr_tuners_cmaes`, `mlr_tuners_design_points`, `mlr_tuners_genoa`, `mlr_tuners_grid_search`, `mlr_tuners_internal`, `mlr_tuners_irace`, `mlr_tuners_nloptr`, `mlr_tuners_random_search`

Examples

```r
as.data.table(mlr_tuners)
mlr_tuners$get("random_search")
tnr("random_search")
```
Hyperparameter Tuning with Asynchronous Design Points

Description

Subclass for asynchronous design points tuning.

Dictionary

This Tuner can be instantiated with the associated sugar function `tnr()`:

```r
tnr("async_design_points")
```

Parameters

- **design** `data.table::data.table`
  Design points to try in search, one per row.

Super classes

```r
mlr3tuning::Tuner -> mlr3tuning::TunerAsync -> mlr3tuning::TunerAsyncFromOptimizerAsync -> TunerAsyncDesignPoints
```

Methods

Public methods:

- `TunerAsyncDesignPoints$new()`
- `TunerAsyncDesignPoints$clone(deep = FALSE)`

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

Usage:

```r
TunerAsyncDesignPoints$new()
```

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

Usage:

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

Arguments:

deep  Whether to make a deep clone.

See Also

Other TunerAsync: `mlr_tuners_async_grid_search`, `mlr_tuners_async_random_search`
Hyperparameter Tuning with Asynchronous Grid Search

Description

Subclass for asynchronous grid search tuning.

Dictionary

This Tuner can be instantiated with the associated sugar function `tnr()`:

```r
tnr("async_design_points")
```

Parameters

- `batch.size` integer(1)
  - Maximum number of points to try in a batch.

Super classes

```
mlr3tuning::Tuner -> mlr3tuning::TunerAsync -> mlr3tuning::TunerAsyncFromOptimizerAsync
-> TunerAsyncGridSearch
```

Methods

Public methods:

- `TunerAsyncGridSearch$new()`
- `TunerAsyncGridSearch$clone()`

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

Usage:

```r
TunerAsyncGridSearch$new()
```

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

Usage:

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

Arguments:

- `deep` Whether to make a deep clone.

See Also

Other TunerAsync: `mlr_tuners_async_design_points`, `mlr_tuners_async_random_search`
Description

Subclass for asynchronous random search tuning.

Details

The random points are sampled by `paradox::generate_design_random()`.

Dictionary

This Tuner can be instantiated with the associated sugar function `tnr()`:

`tnr("async_random_search")`

Super classes

`mlr3tuning::Tuner` -> `mlr3tuning::TunerAsync` -> `mlr3tuning::TunerAsyncFromOptimizerAsync` -> `TunerAsyncRandomSearch`

Methods

Public methods:

- `TunerAsyncRandomSearch$new()`
- `TunerAsyncRandomSearch$clone()`

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

Usage:

```
TunerAsyncRandomSearch$new()
```

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

Usage:

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

Arguments:

deepr Whether to make a deep clone.

Source


See Also

Other TunerAsync: `mlr_tuners_async_design_points`, `mlr_tuners_async_grid_search`
Description

Subclass for Covariance Matrix Adaptation Evolution Strategy (CMA-ES). Calls `adagio::pureCMAES()` from package `adagio`.

Dictionary

This Tuner can be instantiated with the associated sugar function `tnr()`:

`tnr("cmaes")`

Control Parameters

`start_values` character(1)
Create random start values or based on center of search space? In the latter case, it is the center of the parameters before a trafo is applied.

For the meaning of the control parameters, see `adagio::pureCMAES()`. Note that we have removed all control parameters which refer to the termination of the algorithm and where our terminators allow to obtain the same behavior.

Progress Bars

`optimize()` supports progress bars via the package `progressr` combined with a `bbotk::Terminator`. Simply wrap the function in `progressr::with_progress()` to enable them. We recommend to use package `progress` as backend; enable with `progressr::handlers("progress")`.

Logging

All Tuners use a logger (as implemented in `lgr`) from package `bbotk`. Use `lgr::get_logger("bbotk")` to access and control the logger.

Optimizer

This Tuner is based on `bbotk::OptimizerBatchCmaes` which can be applied on any black box optimization problem. See also the documentation of `bbotk`.

Resources

There are several sections about hyperparameter optimization in the `mlr3book`.

- An overview of all tuners can be found on our website.
- Learn more about tuners.

The gallery features a collection of case studies and demos about optimization.

- Use the Hyperband optimizer with different budget parameters.
Super classes

\texttt{mlr3tuning::Tuner} -> \texttt{mlr3tuning::TunerBatch} -> \texttt{mlr3tuning::TunerBatchFromOptimizerBatch} -> \texttt{TunerBatchCmaes}

Methods

**Public methods:**

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

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

*Usage:*

\texttt{TunerBatchCmaes$new()}

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

*Usage:*

\texttt{TunerBatchCmaes$clone(deep = FALSE)}

*Arguments:*

- deep: Whether to make a deep clone.

Source


See Also

Other Tuner: \texttt{Tuner}, \texttt{mlr_tuners}, \texttt{mlr_tuners_design_points}, \texttt{mlr_tuners_gensa}, \texttt{mlr_tuners_grid_search}, \texttt{mlr_tuners_internal}, \texttt{mlr_tuners_irace}, \texttt{mlr_tuners_nloptr}, \texttt{mlr_tuners_random_search}

Examples

```r
# Hyperparameter Optimization

# load learner and set search space
learner = lrn("classif.rpart",
    cp = to_tune(1e-04, 1e-1, logscale = TRUE),
    minsplit = to_tune(p_dbl(2, 128, trafo = as.integer)),
    minbucket = to_tune(p_dbl(1, 64, trafo = as.integer))
)

# run hyperparameter tuning on the Palmer Penguins data set
instance = tune(
    tuner = tnr("cmaes"),
    task = tsk("penguins"),
    learner = learner,
    resampling = rsmp("holdout"),
    measure = msr("classif.ce"),
    term_evals = 10)
```
# best performing hyperparameter configuration
instance$result

# all evaluated hyperparameter configuration
as.data.table(instance$archive)

# fit final model on complete data set
learner$param_set$values = instance$result_learner_param_vals
learner$train(tsk("penguins"))

---

## Description

Subclass for tuning w.r.t. fixed design points.

We simply search over a set of points fully specified by the user. The points in the design are evaluated in order as given.

### Dictionary

This Tuner can be instantiated with the associated sugar function `tnr()`:

```
tnr("design_points")
```

### Parallelization

In order to support general termination criteria and parallelization, we evaluate points in a batch-fasion of size `batch_size`. Larger batches mean we can parallelize more, smaller batches imply a more fine-grained checking of termination criteria. A batch contains of `batch_size` times `resampling$iters` jobs. E.g., if you set a batch size of 10 points and do a 5-fold cross validation, you can utilize up to 50 cores.

Parallelization is supported via package `future` (see `mlr3::benchmark()`’s section on parallelization for more details).

### Logging

All Tuners use a logger (as implemented in `lgr`) from package `bbotk`. Use `lgr::get_logger("bbotk")` to access and control the logger.

### Optimizer

This Tuner is based on `bbotk::OptimizerBatchDesignPoints` which can be applied on any black box optimization problem. See also the documentation of `bbotk`.

---

**mlr_tuners_design_points**

*Hyperparameter Tuning with Design Points*
Parameters

batch.size integer(1)
  Maximum number of configurations to try in a batch.
design data.table::data.table
  Design points to try in search, one per row.

Resources

There are several sections about hyperparameter optimization in the mlr3book.

- An overview of all tuners can be found on our website.
- Learn more about tuners.

The gallery features a collection of case studies and demos about optimization.

- Use the Hyperband optimizer with different budget parameters.

Progress Bars

$optimize() supports progress bars via the package progressr combined with a Terminator. Simply wrap the function in progressr::with_progress() to enable them. We recommend to use package progress as backend; enable with progressr::handlers("progress").

Super classes

mlr3tuning::Tuner -> mlr3tuning::TunerBatch -> mlr3tuning::TunerBatchFromOptimizerBatch
  -> TunerBatchDesignPoints

Methods

Public methods:

- TunerBatchDesignPoints$new()
- TunerBatchDesignPoints$clone()

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

Usage:
  TunerBatchDesignPoints$new()

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

Usage:
  TunerBatchDesignPoints$clone(deep = FALSE)

Arguments:
  deep Whether to make a deep clone.

See Also

Package mlr3hyperband for hyperband tuning.

Other Tuner: Tuner, mlr_tuners, mlr_tuners_cmaes, mlr_tuners_gensa, mlr_tuners_grid_search, mlr_tuners_internal, mlr_tuners_irace, mlr_tuners_nloptr, mlr_tuners_random_search
Examples

# Hyperparameter Optimization

# load learner and set search space
learner = lrn("classif.rpart",
  cp = to_tune(1e-04, 1e-1),
  minsplit = to_tune(2, 128),
  minbucket = to_tune(1, 64)
)

# create design
design = mlr3misc::rowwise_table(
  ~cp, ~minsplit, ~minbucket,
  0.1, 2, 64,
  0.01, 64, 32,
  0.001, 128, 1
)

# run hyperparameter tuning on the Palmer Penguins data set
instance = tune(
  tuner = tnr("design_points", design = design),
  task = tsk("penguins"),
  learner = learner,
  resampling = rsmp("holdout"),
  measure = msr("classif.ce")
)

# best performing hyperparameter configuration
instance$result

# all evaluated hyperparameter configuration
as.data.table(instance$archive)

# fit final model on complete data set
learner$param_set$values = instance$result_learner_param_vals
learner$train(tsk("penguins"))

mlr_tuners_gensa

Hyperparameter Tuning with Generalized Simulated Annealing

Description

Subclass for generalized simulated annealing tuning. Calls GenSA::GenSA() from package GenSA.

Details

In contrast to the GenSA::GenSA() defaults, we set smooth = FALSE as a default.
Dictionary

This Tuner can be instantiated with the associated sugar function `tnr()`:

```r
tnr("gensa")
```

Parallelization

In order to support general termination criteria and parallelization, we evaluate points in a batch-fashion of size `batch_size`. Larger batches mean we can parallelize more, smaller batches imply a more fine-grained checking of termination criteria. A batch contains of `batch_size` times `resampling$iters` jobs. E.g., if you set a batch size of 10 points and do a 5-fold cross validation, you can utilize up to 50 cores.

Parallelization is supported via package `future` (see `mlr3::benchmark()`’s section on parallelization for more details).

Logging

All Tuners use a logger (as implemented in `lgr`) from package `bbotk`. Use `lgr::get_logger("bbotk")` to access and control the logger.

Optimizer

This Tuner is based on `bbotk::OptimizerBatchGenSA` which can be applied on any black box optimization problem. See also the documentation of `bbotk`.

Parameters

- `smooth` logical(1)
- `temperature` numeric(1)
- `acceptance.param` numeric(1)
- `verbose` logical(1)
- `trace.mat` logical(1)

For the meaning of the control parameters, see `GenSA::GenSA()`. Note that we have removed all control parameters which refer to the termination of the algorithm and where our terminators allow to obtain the same behavior.

In contrast to the `GenSA::GenSA()` defaults, we set `trace.mat = FALSE`. Note that `GenSA::GenSA()` uses `smooth = TRUE` as a default. In the case of using this optimizer for Hyperparameter Optimization you may want to set `smooth = FALSE`.

Resources

There are several sections about hyperparameter optimization in the `mlr3book`.

- An overview of all tuners can be found on our website.
- Learn more about tuners.

The gallery features a collection of case studies and demos about optimization.

- Use the Hyperband optimizer with different budget parameters.
Progress Bars

$optimize()$ supports progress bars via the package `progressr` combined with a Terminator. Simply wrap the function in `progressr::with_progress()` to enable them. We recommend to use package `progress` as backend; enable with `progressr::handlers("progress")`.

Super classes

```r
mlr3tuning::Tuner -> mlr3tuning::TunerBatch -> mlr3tuning::TunerBatchFromOptimizerBatch
  -> TunerBatchGenSA
```

Methods

**Public methods:**

- `TunerBatchGenSA$new()`
- `TunerBatchGenSA$clone()`

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

**Usage:**

```r
TunerBatchGenSA$new()
```

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

**Usage:**

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

**Arguments:**

- `deep` Whether to make a deep clone.

Source


See Also

Other Tuner: `Tuner`, `mlr_tuners`, `mlr_tuners_cmaes`, `mlr_tuners_design_points`, `mlr_tuners_grid_search`, `mlr_tuners_internal`, `mlr_tuners_irace`, `mlr_tuners_nloptr`, `mlr_tuners_random_search`

Examples

```r
# Hyperparameter Optimization

# load learner and set search space
learner = lrn("classif.rpart",
  cp = to_tune(1e-04, 1e-1, logscale = TRUE)
)

# run hyperparameter tuning on the Palmer Penguins data set
```
instance = tune(
    tuner = tnr("gensa"),
    task = tsk("penguins"),
    learner = learner,
    resampling = rsmp("holdout"),
    measure = msr("classif.ce"),
    term_evals = 10
)

# best performing hyperparameter configuration
instance$result

# all evaluated hyperparameter configuration
as.data.table(instance$archive)

# fit final model on complete data set
learner$param_set$values = instance$result_learner_param_vals
learner$train(tsk("penguins"))

---

### Hyperparameter Tuning with Grid Search

**Description**

Subclass for grid search tuning.

**Details**

The grid is constructed as a Cartesian product over discretized values per parameter, see `paradox::generate_design_grid()`. If the learner supports hotstarting, the grid is sorted by the hotstart parameter (see also `mlr3::HotstartStack`). If not, the points of the grid are evaluated in a random order.

**Dictionary**

This Tuner can be instantiated with the associated sugar function `tnr()`:

```r
tnr("grid_search")
```

**Control Parameters**

- `resolution` integer(1)
  - Resolution of the grid, see `paradox::generate_design_grid()`.
- `param_resolutions` named integer()
  - Resolution per parameter, named by parameter ID, see `paradox::generate_design_grid()`.
- `batch_size` integer(1)
  - Maximum number of points to try in a batch.
Progress Bars
$optimize()$ supports progress bars via the package $\texttt{progressr}$ combined with a $\texttt{bbotk::Terminator}$. Simply wrap the function in $\texttt{progressr::with_progress()}$ to enable them. We recommend to use package $\texttt{progress}$ as backend; enable with $\texttt{progressr::handlers("progress")}$.

Parallelization
In order to support general termination criteria and parallelization, we evaluate points in a batch-fashion of size $\texttt{batch.size}$. Larger batches mean we can parallelize more, smaller batches imply a more fine-grained checking of termination criteria. A batch contains of $\texttt{batch.size}$ times $\texttt{resampling\$iters}$ jobs. E.g., if you set a batch size of 10 points and do a 5-fold cross validation, you can utilize up to 50 cores.

Parallelization is supported via package $\texttt{future}$ (see $\texttt{mlr3::benchmark()}$’s section on parallelization for more details).

Logging
All $\texttt{Tuner}$s use a logger (as implemented in $\texttt{lgr}$) from package $\texttt{bbotk}$. Use $\texttt{lgr::get_logger("bbotk")}$ to access and control the logger.

Optimizer
This $\texttt{Tuner}$ is based on $\texttt{bbotk::OptimizerBatchGridSearch}$ which can be applied on any black box optimization problem. See also the documentation of $\texttt{bbotk}$.

Resources
There are several sections about hyperparameter optimization in the $\texttt{mlr3book}$.

• An overview of all tuners can be found on our $\texttt{website}$.  
• Learn more about $\texttt{tuners}$.

The gallery features a collection of case studies and demos about optimization.

• Use the $\texttt{Hyperband}$ optimizer with different budget parameters.

Super classes
$\texttt{mlr3tuning::Tuner} \rightarrow \texttt{mlr3tuning::TunerBatch} \rightarrow \texttt{mlr3tuning::TunerBatchFromOptimizerBatch} \rightarrow \texttt{TunerBatchGridSearch}$

Methods

Public methods:

• $\texttt{TunerBatchGridSearch\$new()}$
• $\texttt{TunerBatchGridSearch\$clone()}$

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

Usage:
TunerBatchGridSearch$new()

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

Usage:
TunerBatchGridSearch$clone(deep = FALSE)

Arguments:
depth Whether to make a deep clone.

See Also
Other Tuner: Tuner, mlr_tuners, mlr_tuners_cmaes, mlr_tuners_design_points, mlr_tuners_gensa, mlr_tuners_internal, mlr_tuners_irace, mlr_tuners_nloptr, mlr_tuners_random_search

Examples

# Hyperparameter Optimization

# load learner and set search space
learner = lrn("classif.rpart",
  cp = to_tune(1e-04, 1e-1, logscale = TRUE)
)

# run hyperparameter tuning on the Palmer Penguins data set
instance = tune(
  tuner = tnr("grid_search"),
  task = tsk("penguins"),
  learner = learner,
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  term_evals = 10
)

# best performing hyperparameter configuration
instance$result

# all evaluated hyperparameter configuration
as.data.table(instance$archive)

# fit final model on complete data set
learner$param_set$values = instance$result_learner_param_vals
learner$train(tsk("penguins"))
Dictionary

This Tuner can be instantiated with the associated sugar function \texttt{tnr()}:

\begin{verbatim}
 tnr("internal")
\end{verbatim}

Progress Bars

\texttt{\$optimize()} supports progress bars via the package \texttt{progressr} combined with a \texttt{bbotk::Terminator}. Simply wrap the function in \texttt{progressr::with_progress()} to enable them. We recommend to use package \texttt{progress} as backend; enable with \texttt{progressr::handlers("progress")}.

Logging

All Tuner use a logger (as implemented in \texttt{lgr}) from package \texttt{bbotk}. Use \texttt{lgr::get_logger("bbotk")} to access and control the logger.

Resources

There are several sections about hyperparameter optimization in the \texttt{mlr3book}.

- An overview of all tuners can be found on our website.
- Learn more about tuners.

The gallery features a collection of case studies and demos about optimization.

- Use the \texttt{Hyperband} optimizer with different budget parameters.

Super classes

\texttt{mlr3tuning::Tuner} \rightarrow \texttt{mlr3tuning::TunerBatch} \rightarrow \texttt{TunerBatchInternal}

Methods

Public methods:

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

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

\texttt{Usage:}
TunerBatchInternal\$new()

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

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

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

The selected \texttt{mlr3::Measure} does not influence the tuning result. To change the loss-function for the internal tuning, consult the hyperparameter documentation of the tuned \texttt{mlr3::Learner}.

### See Also

Other Tuner: \texttt{Tuner,mlr\_tuners,mlr\_tuners\_cmaes,mlr\_tuners\_design\_points,mlr\_tuners\_gensa,mlr\_tuners\_grid\_search,mlr\_tuners\_irace,mlr\_tuners\_nloptr,mlr\_tuners\_random\_search}

### Examples

```r
library(mlr3learners)

# Retrieve task
task = tsk("pima")

# Load learner and set search space
learner = lrn("classif.xgboost",
    nrounds = to_tune(upper = 1000, internal = TRUE),
    early_stopping_rounds = 10,
    validate = "test"
)

# Internal hyperparameter tuning on the pima indians diabetes data set
instance = tune(
    tnr("internal"),
    tsk("iris"),
    learner,
    rsmp("cv", folds = 3),
    msr("classif.ce")
)

# best performing hyperparameter configuration
instance$result_learner_param_vals

instance$result_learner_param_vals$internal_tuned_values
```

### Description

Hyperparameter Tuning with Iterated Racing.

Subclass for iterated racing. Calls \texttt{irace::irace()} from package \texttt{irace}.

### Dictionary

This Tuner can be instantiated with the associated sugar function \texttt{tnr()}: 

\texttt{tnr("irace")}
Control Parameters

n_instances integer(1)
   Number of resampling instances.

For the meaning of all other parameters, see `irace::defaultScenario()`. Note that we have removed all control parameters which refer to the termination of the algorithm. Use `bbotk::TerminatorEvals` instead. Other terminators do not work with `TunerIrace`.

Archive

The `ArchiveBatchTuning` holds the following additional columns:

- "race" (integer(1))
  Race iteration.
- "step" (integer(1))
  Step number of race.
- "instance" (integer(1))
  Identifies resampling instances across races and steps.
- "configuration" (integer(1))
  Identifies configurations across races and steps.

Result

The tuning result (instance$result) is the best-performing elite of the final race. The reported performance is the average performance estimated on all used instances.

Progress Bars

$optimize() supports progress bars via the package `progressr` combined with a `bbotk::Terminator`. Simply wrap the function in `progressr::with_progress()` to enable them. We recommend to use package `progress` as backend; enable with `progressr::handlers("progress")`.

Logging

All `Tuners` use a logger (as implemented in `lgr`) from package `bbotk`. Use `lgr::get_logger("bbotk")` to access and control the logger.

Optimizer

This `Tuner` is based on `bbotk::OptimizerBatchIrace` which can be applied on any black box optimization problem. See also the documentation of `bbotk`.

Resources

There are several sections about hyperparameter optimization in the `mlr3book`.

- An overview of all tuners can be found on our `website`.
- Learn more about `tuners`.

The `gallery` features a collection of case studies and demos about optimization.

- Use the `Hyperband` optimizer with different budget parameters.
Super classes

```
mlr3tuning::Tuner -> mlr3tuning::TunerBatch -> mlr3tuning::TunerBatchFromOptimizerBatch -> TunerBatchIrace
```

Methods

Public methods:

\begin{itemize}
  \item TunerBatchIrace$new() \item TunerBatchIrace$optimize() \item TunerBatchIrace$clone()
\end{itemize}

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

*Usage:*

```
TunerBatchIrace$new()
```

**Method** `optimize()`: Performs the tuning on a `TuningInstanceBatchSingleCrit` until termination. The single evaluations and the final results will be written into the `ArchiveBatchTuning` that resides in the `TuningInstanceBatchSingleCrit`. The final result is returned.

*Usage:*

```
TunerBatchIrace$optimize(inst)
```

*Arguments:*

\begin{itemize}
  \item `inst` (`TuningInstanceBatchSingleCrit`).
\end{itemize}

*Returns:* `data.table::data.table`.

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

*Usage:*

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

*Arguments:*

\begin{itemize}
  \item `deep` Whether to make a deep clone.
\end{itemize}

Source


See Also

Other Tuner: `Tuner`, `mlr_tuners`, `mlr_tuners_cmaes`, `mlr_tuners_design_points`, `mlr_tuners_gensa`, `mlr_tuners_grid_search`, `mlr_tuners_internal`, `mlr_tuners_nloptr`, `mlr_tuners_random_search`
Examples

```r
# retrieve task
task = tsk("pima")

# load learner and set search space
learner = lrn("classif.rpart", cp = to_tune(1e-04, 1e-1, logscale = TRUE))

# hyperparameter tuning on the pima indians diabetes data set
instance = tune(
  tuner = tnr("irace"),
  task = task,
  learner = learner,
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  term_evals = 42
)

# best performing hyperparameter configuration
instance$result

# all evaluated hyperparameter configuration
as.data.table(instance$archive)

# fit final model on complete data set
learner$param_set$values = instance$result_learner_param_vals
learner$train(task)
```

### Description

Subclass for non-linear optimization (NLopt). Calls `nloptr::nloptr` from package `nloptr`.

### Details

The termination conditions `stopval`, `maxtime` and `maxeval` of `nloptr::nloptr()` are deactivated and replaced by the `bboptk::Terminator` subclasses. The x and function value tolerance termination conditions `xtol_rel = 10^{-4}`, `xtol_abs = rep(0,0, length(x0))`, `ftol_rel = 0.0` and `ftol_abs = 0.0`) are still available and implemented with their package defaults. To deactivate these conditions, set them to `-1`.

### Dictionary

This Tuner can be instantiated with the associated sugar function `tnr()`:

```r
tnr("nloptr")
```
Logging

All Tuners use a logger (as implemented in lgr) from package bbotk. Use `lgr::get_logger("bbotk")` to access and control the logger.

Optimizer

This Tuner is based on `bbotk::OptimizerBatchNLoptr` which can be applied on any black box optimization problem. See also the documentation of bbotk.

Parameters

- `algorithm` character(1)
- `eval_g_ineq` function()
- `xtol_rel` numeric(1)
- `xtol_abs` numeric(1)
- `ftol_rel` numeric(1)
- `ftol_abs` numeric(1)
- `start_values` character(1)

Create random start values or based on center of search space? In the latter case, it is the center of the parameters before a trafo is applied.

For the meaning of the control parameters, see `nloptr::nloptr()` and `nloptr::nloptr.print.options()`.

The termination conditions `stopval`, `maxtime` and `maxeval` of `nloptr::nloptr()` are deactivated and replaced by the Terminator subclasses. The x and function value tolerance termination conditions (`xtol_rel = 10^-4`, `xtol_abs = rep(0.0, length(x0))`, `ftol_rel = 0.0` and `ftol_abs = 0.0`) are still available and implemented with their package defaults. To deactivate these conditions, set them to -1.

Resources

There are several sections about hyperparameter optimization in the mlr3book.

- An overview of all tuners can be found on our website.
- Learn more about tuners.

The gallery features a collection of case studies and demos about optimization.

- Use the Hyperband optimizer with different budget parameters.

Progress Bars

`$optimize()` supports progress bars via the package `progressr` combined with a Terminator. Simply wrap the function in `progressr::with_progress()` to enable them. We recommend to use package `progress` as backend; enable with `progressr::handlers("progress")`.

Super classes

- mlr3tuning::Tuner -> mlr3tuning::TunerBatch -> mlr3tuning::TunerBatchFromOptimizerBatch -> TunerBatchNLoptr
Methods

Public methods:

- TunerBatchNLoptr$new()
- TunerBatchNLoptr$clone()

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

Usage:

TunerBatchNLoptr$new()

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

Usage:

TunerBatchNLoptr$clone(deep = FALSE)

Arguments:

deepl Whether to make a deep clone.

Source


See Also

Other Tuner: Tuner, mlr_tuners, mlr_tuners_cmaes, mlr_tuners_design_points, mlr_tuners_gensa, mlr_tuners_grid_search, mlr_tuners_internal, mlr_tuners_irace, mlr_tuners_random_search

Examples

# Hyperparameter Optimization

# load learner and set search space
learner = lrn("classif.rpart",
    cp = to_tune(1e-04, 1e-1, logscale = TRUE)
)

# run hyperparameter tuning on the Palmer Penguins data set
instance = tune(
    tuner = tnr("nloptr", algorithm = "NLOPT_LN_BOBYQA"),
    task = tsk("penguins"),
    learner = learner,
    resampling = rsmp("holdout"),
    measure = msr("classif.ce")
)

# best performing hyperparameter configuration
instance$result

# all evaluated hyperparameter configuration
as.data.table(instance$archive)
# fit final model on complete data set
learner$param_set$values = instance$result_learner_param_vals
learner$train(tsk("penguins"))

mlr_tuners_random_search

**Hyperparameter Tuning with Random Search**

**Description**
Subclass for random search tuning.

**Details**
The random points are sampled by `paradox::generate_design_random()`.

**Dictionary**
This Tuner can be instantiated with the associated sugar function `tnr()`:

`tnr("random_search")`

**Parallelization**
In order to support general termination criteria and parallelization, we evaluate points in a batch-fashion of size `batch_size`. Larger batches mean we can parallelize more, smaller batches imply a more fine-grained checking of termination criteria. A batch contains of `batch_size` times `resampling$iters` jobs. E.g., if you set a batch size of 10 points and do a 5-fold cross validation, you can utilize up to 50 cores.

Parallelization is supported via package `future` (see `mlr3::benchmark()`’s section on parallelization for more details).

**Logging**
All Tuners use a logger (as implemented in `lgr`) from package `bbotk`. Use `lgr::get_logger("bbotk")` to access and control the logger.

**Optimizer**
This Tuner is based on `bbotk::OptimizerBatchRandomSearch` which can be applied on any black box optimization problem. See also the documentation of `bbotk`.

**Parameters**
- `batch_size` integer(1)
  Maximum number of points to try in a batch.
Resources

There are several sections about hyperparameter optimization in the mlr3book.

- An overview of all tuners can be found on our website.
- Learn more about tuners.

The gallery features a collection of case studies and demos about optimization.

- Use the Hyperband optimizer with different budget parameters.

Progress Bars

$optimize() supports progress bars via the package progressr combined with a Terminator. Simply wrap the function in progressr::with_progress() to enable them. We recommend to use package progressr as backend; enable with progressr::handlers("progress").

Super classes

mlr3tuning::Tuner -> mlr3tuning::TunerBatch -> mlr3tuning::TunerBatchFromOptimizerBatch -> TunerBatchRandomSearch

Methods

Public methods:

- TunerBatchRandomSearch$new()
- TunerBatchRandomSearch$clone()

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

Usage:
TunerBatchRandomSearch$new()

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

Usage:
TunerBatchRandomSearch$clone(deep = FALSE)

Arguments:
dee. Whether to make a deep clone.

Source


See Also

Package mlr3hyperband for hyperband tuning.

Other Tuner: Tuner, mlr_tuners, mlr_tuners_cmaes, mlr_tuners_design_points, mlr_tuners_gensa, mlr_tuners_grid_search, mlr_tuners_internal, mlr_tuners_irace, mlr_tuners_nloptr
# Hyperparameter Optimization

```r
# load learner and set search space
learner = lrn("classif.rpart",
            cp = to_tune(1e-04, 1e-1, logscale = TRUE)
)

# run hyperparameter tuning on the Palmer Penguins data set
instance = tune(
            tuner = tnr("random_search"),
            task = tsk("penguins"),
            learner = learner,
            resampling = rsmp("holdout"),
            measure = msr("classif.ce"),
            term_evals = 10
)

# best performing hyperparameter configuration
instance$result

# all evaluated hyperparameter configuration
as.data.table(instance$archive)

# fit final model on complete data set
learner$param_set$values = instance$result_learner_param_vals
learner$train(tsk("penguins"))
```

---

**ObjectiveTuning**  
*Class for Tuning Objective*

**Description**

Stores the objective function that estimates the performance of hyperparameter configurations. This class is usually constructed internally by the `TuningInstanceBatchSingleCrit` or `TuningInstanceBatchMultiCrit`.

**Super class**

`bbotk::Objective` -> `ObjectiveTuning`

**Public fields**

- `task` (mlr3::Task).
- `learner` (mlr3::Learner).
- `resampling` (mlr3::Resampling).
- `measures` (list of mlr3::Measure).
- `store_models` (logical(1)).
store_benchmark_result (logical(1)).
callbacks (List of mlr3misc::Callback).
default_values (named list()).

Methods

**Public methods:**
- ObjectiveTuning$new()
- ObjectiveTuning$clone()

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

Usage:
```r
ObjectiveTuning$new(
  task,
  learner,
  resampling,
  measures,
  store_benchmark_result = TRUE,
  store_models = FALSE,
  check_values = FALSE,
  callbacks = NULL
)
```

Arguments:
- **task** (mlr3::Task)
  Task to operate on.
- **learner** (mlr3::Learner)
  Learner to tune.
- **resampling** (mlr3::Resampling)
  Resampling that is used to evaluate the performance of the hyperparameter configurations. Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits. Already instantiated resamplings are kept unchanged. Specialized Tuner change the resampling e.g. to evaluate a hyperparameter configuration on different data splits. This field, however, always returns the resampling passed in construction.
- **measures** (list of mlr3::Measure)
  Measures to optimize.
- **store_benchmark_result** (logical(1))
  If TRUE (default), store resample result of evaluated hyperparameter configurations in archive as mlr3::BenchmarkResult.
- **store_models** (logical(1))
  If TRUE, fitted models are stored in the benchmark result (archive$benchmark_result). If store_benchmark_result = FALSE, models are only stored temporarily and not accessible after the tuning. This combination is needed for measures that require a model.
- **check_values** (logical(1))
  If TRUE, hyperparameter values are checked before evaluation and performance scores after. If FALSE (default), values are unchecked but computational overhead is reduced.
**ObjectiveTuningAsync**

callbacks (list of mlr3misc::Callback)
List of callbacks.

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

**Usage:**
ObjectiveTuning$clone(deep = FALSE)

**Arguments:**
depth Whether to make a deep clone.

---

**ObjectiveTuningAsync  Class for Tuning Objective**

**Description**
Stores the objective function that estimates the performance of hyperparameter configurations. This class is usually constructed internally by the TuningInstanceBatchSingleCrit or TuningInstanceBatchMultiCrit.

**Super classes**

bbotk::Objective -> mlr3tuning::ObjectiveTuning -> ObjectiveTuningAsync

**Methods**

**Public methods:**

- ObjectiveTuningAsync$clone()

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

**Usage:**
ObjectiveTuningAsync$clone(deep = FALSE)

**Arguments:**
depth Whether to make a deep clone.
ObjectiveTuningBatch  Class for Tuning Objective

Description
Stores the objective function that estimates the performance of hyperparameter configurations. This class is usually constructed internally by the TuningInstanceBatchSingleCrit or TuningInstanceBatchMultiCrit.

Super classes
bbotk::Objective -> mlr3tuning::ObjectiveTuning -> ObjectiveTuningBatch

Public fields
archive (ArchiveBatchTuning).

Methods
Public methods:
• ObjectiveTuningBatch$new()
• ObjectiveTuningBatch$clone()

Method new(): Creates a new instance of this R6 class.
Usage:
ObjectiveTuningBatch$new(
  task,
  learner,
  resampling,
  measures,
  store_benchmark_result = TRUE,
  store_models = FALSE,
  check_values = FALSE,
  archive = NULL,
  callbacks = NULL
)
Arguments:
task (mlr3::Task)
  Task to operate on.
learner (mlr3::Learner)
  Learner to tune.
resampling (mlr3::Resampling)
  Resampling that is used to evaluate the performance of the hyperparameter configurations. Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits. Already instantiated resamplings are kept unchanged.
set_validate(AutoTuner)

Configure Validation for AutoTuner

Description

Configure validation for the final model fit (final_validate), as well as during the tuning (validate).

Usage

## S3 method for class 'AutoTuner'
set_validate(learner, validate, final_validate, ...)

Arguments

learner (AutoTuner)
The autotuner for which to enable validation.

Specialized Tuner change the resampling e.g. to evaluate a hyperparameter configuration on different data splits. This field, however, always returns the resampling passed in construction.

measures (list of mlr3::Measure)
  Measures to optimize.

store_benchmark_result (logical(1))
  If TRUE (default), store resample result of evaluated hyperparameter configurations in archive as mlr3::BenchmarkResult.

store_models (logical(1))
  If TRUE, fitted models are stored in the benchmark result (archive$benchmark_result). If store_benchmark_result = FALSE, models are only stored temporarily and not accessible after the tuning. This combination is needed for measures that require a model.

check_values (logical(1))
  If TRUE, hyperparameter values are checked before evaluation and performance scores after.
  If FALSE (default), values are unchecked but computational overhead is reduced.

archive (ArchiveBatchTuning)
  Reference to archive of TuningInstanceBatchSingleCrit | TuningInstanceBatchMultiCrit. If NULL (default), benchmark result and models cannot be stored.

callbacks (list of mlr3misc::Callback)
  List of callbacks.

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

Usage:
ObjectiveTuningBatch$clone(deep = FALSE)

Arguments:
  deep Whether to make a deep clone.
validate (numeric(1), "predefined", "test", or NULL) How to configure the validation during the hyperparameter tuning.

final_validate (numeric(1), "predefined", "test" or NULL) How to configure the validation during the final model fit. The default behavior is to not change the value. Rarely needed.

... (any) Passed when calling set_validate() on the wrapped learner.

Examples

```r
at = auto_tuner(
  tuner = tnr("random_search"),
  learner = lrn("classif.debug", early_stopping = TRUE,
    iter = to_tune(upper = 1000L, internal = TRUE), validate = 0.2),
  resampling = rsmp("holdout")
)

# use the test set as validation data during tuning
set_validate(at, validate = "test")
at$learner$validate
```

---

### ti

#### Syntactic Sugar for Tuning Instance Construction

**Description**

Function to construct a `TuningInstanceBatchSingleCrit` or `TuningInstanceBatchMultiCrit`.

**Usage**

```r
ti(
  task, 
  learner, 
  resampling, 
  measures = NULL, 
  terminator, 
  search_space = NULL, 
  store_benchmark_result = TRUE, 
  store_models = FALSE, 
  check_values = FALSE, 
  callbacks = NULL 
)
```

**Arguments**

- **task** *(mlr3::Task)*
  Task to operate on.
- **learner** *(mlr3::Learner)*
  Learner to tune.
resampling (mlr3::Resampling)
Resampling that is used to evaluate the performance of the hyperparameter configurations. Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits. Already instantiated resamplings are kept unchanged. Specialized Tuner change the resampling e.g. to evaluate a hyperparameter configuration on different data splits. This field, however, always returns the resampling passed in construction.

measures (mlr3::Measure or list of mlr3::Measure)
A single measure creates a TuningInstanceBatchSingleCrit and multiple measures a TuningInstanceBatchMultiCrit. If NULL, default measure is used.

terminator (bbotk::Terminator)
Stop criterion of the tuning process.

search_space (paradox::ParamSet)
Hyperparameter search space. If NULL (default), the search space is constructed from the paradox::TuneToken of the learner’s parameter set (learner$param_set).

store_benchmark_result (logical(1))
If TRUE (default), store resample result of evaluated hyperparameter configurations in archive as mlr3::BenchmarkResult.

store_models (logical(1))
If TRUE, fitted models are stored in the benchmark result (archive$benchmark_result). If store_benchmark_result = FALSE, models are only stored temporarily and not accessible after the tuning. This combination is needed for measures that require a model.

check_values (logical(1))
If TRUE, hyperparameter values are checked before evaluation and performance scores after. If FALSE (default), values are unchecked but computational overhead is reduced.

callbacks (list of mlr3misc::Callback)
List of callbacks.

Resources

There are several sections about hyperparameter optimization in the mlr3book.

- Getting started with hyperparameter optimization.
- Tune a simple classification tree on the Sonar data set.
- Learn about tuning spaces.

The gallery features a collection of case studies and demos about optimization.

- Learn more advanced methods with the practical tuning series.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Make use of proven search space.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
Default Measures

If no measure is passed, the default measure is used. The default measure depends on the task type.

<table>
<thead>
<tr>
<th>Task</th>
<th>Default Measure</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;classif&quot;</td>
<td>&quot;classif.ce&quot;</td>
<td>mlr3</td>
</tr>
<tr>
<td>&quot;regr&quot;</td>
<td>&quot;regr.mse&quot;</td>
<td>mlr3</td>
</tr>
<tr>
<td>&quot;surv&quot;</td>
<td>&quot;surv.cindex&quot;</td>
<td>mlr3proba</td>
</tr>
<tr>
<td>&quot;dens&quot;</td>
<td>&quot;dens.logloss&quot;</td>
<td>mlr3proba</td>
</tr>
<tr>
<td>&quot;classif_st&quot;</td>
<td>&quot;classif.ce&quot;</td>
<td>mlr3spatial</td>
</tr>
<tr>
<td>&quot;regr_st&quot;</td>
<td>&quot;regr.mse&quot;</td>
<td>mlr3spatial</td>
</tr>
<tr>
<td>&quot;clust&quot;</td>
<td>&quot;clust.dunn&quot;</td>
<td>mlr3cluster</td>
</tr>
</tbody>
</table>

Examples

```r
# Hyperparameter optimization on the Palmer Penguins data set
task = tsk("penguins")

# Load learner and set search space
learner = lrn("classif.rpart,
  cp = to_tune(1e-04, 1e-1, logscale = TRUE)
)

# Construct tuning instance
instance = ti(
  task = task,
  learner = learner,
  resampling = rsmp("cv", folds = 3),
  measures = msr("classif.ce"),
  terminator = trm("evals", n_evals = 4)
)

# Choose optimization algorithm
tuner = tnr("random_search", batch_size = 2)

# Run tuning
tuner$optimize(instance)

# Set optimal hyperparameter configuration to learner
learner$param_set$values = instance$result_learner_param_vals

# Train the learner on the full data set
learner$train(task)

# Inspect all evaluated configurations
as.data.table(instance$archive)
```
ti_async  Syntactic Sugar for Asynchronous Tuning Instance Construction

Description

Function to construct a TuningInstanceAsyncSingleCrit or TuningInstanceAsyncMultiCrit.

Usage

```r
ti_async(
  task,
  learner,
  resampling,
  measures = NULL,
  terminator,
  search_space = NULL,
  store_benchmark_result = TRUE,
  store_models = FALSE,
  check_values = FALSE,
  callbacks = NULL,
  rush = NULL
)
```

Arguments

- **task** *(mlr3::Task)*
  Task to operate on.
- **learner** *(mlr3::Learner)*
  Learner to tune.
- **resampling** *(mlr3::Resampling)*
  Resampling that is used to evaluate the performance of the hyperparameter configurations. Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits. Already instantiated resamplings are kept unchanged. Specialized Tuner change the resampling e.g. to evaluate a hyperparameter configuration on different data splits. This field, however, always returns the resampling passed in construction.
- **measures** *(mlr3::Measure or list of mlr3::Measure)*
  A single measure creates a TuningInstanceAsyncSingleCrit and multiple measures a TuningInstanceAsyncMultiCrit. If NULL, default measure is used.
- **terminator** *(bbotk::Terminator)*
  Stop criterion of the tuning process.
- **search_space** *(paradox::ParamSet)*
  Hyperparameter search space. If NULL (default), the search space is constructed from the paradox::TuneToken of the learner’s parameter set (learner$param_set).
store_benchmark_result (logical(1))
If TRUE (default), store resample result of evaluated hyperparameter configurations in archive as mlr3::BenchmarkResult.

store_models (logical(1))
If TRUE, fitted models are stored in the benchmark result (archive$benchmark_result). If store_benchmark_result = FALSE, models are only stored temporarily and not accessible after the tuning. This combination is needed for measures that require a model.

check_values (logical(1))
If TRUE, hyperparameter values are checked before evaluation and performance scores after. If FALSE (default), values are unchecked but computational overhead is reduced.

callbacks (list of mlr3misc::Callback)
List of callbacks.

rush (Rush)
If a rush instance is supplied, the tuning runs without batches.

Resources

There are several sections about hyperparameter optimization in the mlr3book.
  • Getting started with hyperparameter optimization.
  • Tune a simple classification tree on the Sonar data set.
  • Learn about tuning spaces.

The gallery features a collection of case studies and demos about optimization.
  • Learn more advanced methods with the practical tuning series.
  • Simultaneously optimize hyperparameters and use early stopping with XGBoost.
  • Make use of proven search space.
  • Learn about hotstarting models.
  • Run the default hyperparameter configuration of learners as a baseline.

Default Measures

If no measure is passed, the default measure is used. The default measure depends on the task type.

<table>
<thead>
<tr>
<th>Task</th>
<th>Default Measure</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;classif&quot;</td>
<td>&quot;classif.ce&quot;</td>
<td>mlr3</td>
</tr>
<tr>
<td>&quot;regr&quot;</td>
<td>&quot;regr.mse&quot;</td>
<td>mlr3</td>
</tr>
<tr>
<td>&quot;surv&quot;</td>
<td>&quot;surv.cindex&quot;</td>
<td>mlr3proba</td>
</tr>
<tr>
<td>&quot;dens&quot;</td>
<td>&quot;dens.logloss&quot;</td>
<td>mlr3proba</td>
</tr>
<tr>
<td>&quot;classif_st&quot;</td>
<td>&quot;classif.ce&quot;</td>
<td>mlr3spatial</td>
</tr>
<tr>
<td>&quot;regr_st&quot;</td>
<td>&quot;regr.mse&quot;</td>
<td>mlr3spatial</td>
</tr>
<tr>
<td>&quot;clust&quot;</td>
<td>&quot;clust.dunn&quot;</td>
<td>mlr3cluster</td>
</tr>
</tbody>
</table>
Examples

```r
# Hyperparameter optimization on the Palmer Penguins data set
task = tsk("penguins")

# Load learner and set search space
learner = lrn("classif.rpart",
  cp = to_tune(1e-04, 1e-1, logscale = TRUE)
)

# Construct tuning instance
instance = ti(
  task = task,
  learner = learner,
  resampling = rsmp("cv", folds = 3),
  measures = msr("classif.ce"),
  terminator = trm("evals", n_evals = 4)
)

# Choose optimization algorithm
tuner = tnr("random_search", batch_size = 2)

# Run tuning
tuner$optimize(instance)

# Set optimal hyperparameter configuration to learner
learner$param_set$values = instance$result_learner_param_vals

# Train the learner on the full data set
learner$train(task)

# Inspect all evaluated configurations
as.data.table(instance$archive)
```

### tnr

**Syntactic Sugar for Tuning Objects Construction**

**Description**

Functions to retrieve objects, set parameters and assign to fields in one go. Relies on `mlr3misc::dictionary_sugar_get()` to extract objects from the respective `mlr3misc::Dictionary`:

- `tnr()` for a Tuner from `mlr_tuners`.
- `tnrs()` for a list of Tuners from `mlr_tuners`.
- `trm()` for a `bbotk::Terminator` from `mlr_terminators`.
- `trms()` for a list of Terminators from `mlr_terminators`. 
Usage

tnr(.key, ...)

tnrs(.keys, ...)

Arguments

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

... (any)
Additional arguments.

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

Value

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

Examples

# random search tuner with batch size of 5
tnr("random_search", batch_size = 5)

# run time terminator with 20 seconds
trm("run_time", secs = 20)

---

tune Function for Tuning a Learner

Description

Function to tune a mlr3::Learner. The function internally creates a TuningInstanceBatchSingleCrit or TuningInstanceBatchMultiCrit which describes the tuning problem. It executes the tuning with the Tuner (tuner) and returns the result with the tuning instance ($result). The ArchiveBatchTuning and ArchiveAsyncTuning ($archive) stores all evaluated hyperparameter configurations and performance scores.

You can find an overview of all tuners on our website.

Usage

tune(
  tuner,
  task,
  learner,
  resampling,
  measures = NULL,
tune

    term_evals = NULL,
    term_time = NULL,
    terminator = NULL,
    search_space = NULL,
    store_benchmark_result = TRUE,
    store_models = FALSE,
    check_values = FALSE,
    callbacks = NULL,
    rush = NULL
)

Arguments

tuner (Tuner)
    Optimization algorithm.

task (mlr3::Task)
    Task to operate on.

learner (mlr3::Learner)
    Learner to tune.

resampling (mlr3::Resampling)
    Resampling that is used to evaluate the performance of the hyperparameter configurations. Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits. Already instantiated resamplings are kept unchanged. Specialized Tuner change the resampling e.g. to evaluate a hyperparameter configuration on different data splits. This field, however, always returns the resampling passed in construction.

measures (mlr3::Measure or list of mlr3::Measure)
    A single measure creates a TuningInstanceBatchSingleCrit and multiple measures a TuningInstanceBatchMultiCrit. If NULL, default measure is used.

term_evals (integer(1))
    Number of allowed evaluations. Ignored if terminator is passed.

term_time (integer(1))
    Maximum allowed time in seconds. Ignored if terminator is passed.

terminator (bbotk::Terminator)
    Stop criterion of the tuning process.

search_space (paradox::ParamSet)
    Hyperparameter search space. If NULL (default), the search space is constructed from the paradox::TuneToken of the learner’s parameter set (learner$param_set).

store_benchmark_result (logical(1))
    If TRUE (default), store resample result of evaluated hyperparameter configurations in archive as mlr3::BenchmarkResult.

store_models (logical(1))
    If TRUE, fitted models are stored in the benchmark result (archive$benchmark_result). If store_benchmark_result = FALSE, models are only stored temporarily and not accessible after the tuning. This combination is needed for measures that require a model.
check_values (logical(1))
If TRUE, hyperparameter values are checked before evaluation and performance scores after. If FALSE (default), values are unchecked but computational overhead is reduced.

callbacks (list of mlr3misc::Callback)
List of callbacks.
rush (Rush)
If a rush instance is supplied, the tuning runs without batches.

Details
The mlr3::Task, mlr3::Learner, mlr3::Resampling, mlr3::Measure and bbotk::Terminator are used to construct a TuningInstanceBatchSingleCrit. If multiple performance mlr3::Measures are supplied, a TuningInstanceBatchMultiCrit is created. The parameter term_evals and term_time are shortcuts to create a bbotk::Terminator. If both parameters are passed, a bbotk::TerminatorCombo is constructed. For other Terminators, pass one with terminator. If no termination criterion is needed, set term_evals, term_time and terminator to NULL. The search space is created from paradox::TuneToken or is supplied by search_space.

Value
TuningInstanceBatchSingleCrit | TuningInstanceBatchMultiCrit

Resources
There are several sections about hyperparameter optimization in the mlr3book.

- Simplify tuning with the `tune()` function.
- Learn about tuning spaces.

The gallery features a collection of case studies and demos about optimization.

- Optimize an rpart classification tree with only a few lines of code.
- Tune an XGBoost model with early stopping.
- Make use of proven search space.
- Learn about hotstarting models.

Default Measures
If no measure is passed, the default measure is used. The default measure depends on the task type.

<table>
<thead>
<tr>
<th>Task</th>
<th>Default Measure</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;classif&quot;</td>
<td>&quot;classif.ce&quot;</td>
<td>mlr3</td>
</tr>
<tr>
<td>&quot;regr&quot;</td>
<td>&quot;regr.mse&quot;</td>
<td>mlr3</td>
</tr>
<tr>
<td>&quot;surv&quot;</td>
<td>&quot;surv.cindex&quot;</td>
<td>mlr3proba</td>
</tr>
<tr>
<td>&quot;dens&quot;</td>
<td>&quot;dens.logloss&quot;</td>
<td>mlr3proba</td>
</tr>
<tr>
<td>&quot;classif_st&quot;</td>
<td>&quot;classif.ce&quot;</td>
<td>mlr3spatial</td>
</tr>
<tr>
<td>&quot;regr_st&quot;</td>
<td>&quot;regr.mse&quot;</td>
<td>mlr3spatial</td>
</tr>
</tbody>
</table>
Analysis

For analyzing the tuning results, it is recommended to pass the ArchiveBatchTuning to as.data.table(). The returned data table is joined with the benchmark result which adds the mlr3::ResampleResult for each hyperparameter evaluation.

The archive provides various getters (e.g. $learners()) to ease the access. All getters extract by position (i) or unique hash (uhash). For a complete list of all getters see the methods section.

The benchmark result ($benchmark_result) allows to score the hyperparameter configurations again on a different measure. Alternatively, measures can be supplied to as.data.table().

The mlr3viz package provides visualizations for tuning results.

Examples

```r
# Hyperparameter optimization on the Palmer Penguins data set
task = tsk("pima")

# Load learner and set search space
learner = lrn("classif.rpart", 
  cp = to_tune(1e-04, 1e-1, logscale = TRUE)
)

# Run tuning
instance = tune(
  tuner = tnr("random_search", batch_size = 2),
  task = tsk("pima"),
  learner = learner,
  resampling = rsmp("holdout"),
  measures = msr("classif.ce"),
  terminator = trm("evals", n_evals = 4)
)

# Set optimal hyperparameter configuration to learner
learner$param_set$values = instance$result_learner_param_vals

# Train the learner on the full data set
learner$train(task)

# Inspect all evaluated configurations
as.data.table(instance$archive)
```

Description

The Tuner implements the optimization algorithm.
Details

Tuner is an abstract base class that implements the base functionality each tuner must provide.

Resources

There are several sections about hyperparameter optimization in the mlr3book.

- An overview of all tuners can be found on our website.
- Learn more about tuners.

The gallery features a collection of case studies and demos about optimization.

- Use the Hyperband optimizer with different budget parameters.

Extension Packages

Additional tuners are provided by the following packages.

- mlr3hyperband adds the Hyperband and Successive Halving algorithm.
- mlr3mbo adds Bayesian optimization methods.

Public fields

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

Active bindings

```r
param_set (paradox::ParamSet)
  Set of control parameters.

param_classes (character())
  Supported parameter classes for learner hyperparameters that the tuner can optimize, as given in the paradox::ParamSet $class field.

properties (character())
  Set of properties of the tuner. Must be a subset of mlr_reflections$tuner_properties.

packages (character())
  Set of required packages. Note that these packages will be loaded via requireNamespace(), and are not attached.

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

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().
```
Methods

Public methods:

• Tuner$new()
• Tuner$format()
• Tuner$print()
• Tuner$help()
• Tuner$clone()

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

Usage:
Tuner$new(id = "tuner",
          param_set,
          param_classes,
          properties,
          packages = character(),
          label = NA.character_,
          man = NA.character_)
Arguments:
id (character(1))
  Identifier for the new instance.
param_set (paradox::ParamSet)
  Set of control parameters.
param_classes (character())
  Supported parameter classes for learner hyperparameters that the tuner can optimize, as
given in the paradox::ParamSet $class field.
properties (character())
  Set of properties of the tuner. Must be a subset of mlr_reflections$tuner_properties.
packages (character())
  Set of required packages. Note that these packages will be loaded via requireNamespace(),
and are not attached.
label (character(1))
  Label for this object. Can be used in tables, plot and text output instead of the ID.
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:
Tuner$format(...)
Arguments:
... (ignored).
Returns: (character()).
Method `print()`: Print method.

Usage:
```
Tuner$print()
```

Returns: (character()).

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

Usage:
```
Tuner$help()
```

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

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

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

See Also

Other Tuner: `mlr_tuners`, `mlr_tuners_cmaes`, `mlr_tuners_design_points`, `mlr_tuners_genSA`, `mlr_tuners_grid_search`, `mlr_tuners_internal`, `mlr_tuners_irace`, `mlr_tuners_nloptr`, `mlr_tuners_random_search`

### Description

The `TunerAsync` implements the asynchronous optimization algorithm.

### Details

`TunerAsync` is an abstract base class that implements the base functionality each asynchronous tuner must provide.

### Resources

There are several sections about hyperparameter optimization in the `mlr3book`.

- An overview of all tuners can be found on our [website](https://mlr3.readthedocs.io/en/latest/).
- Learn more about [tuners](https://mlr3.readthedocs.io/en/latest/).

The [gallery](https://mlr3.readthedocs.io/en/latest/) features a collection of case studies and demos about optimization.

- Use the [Hyperband](https://mlr3.readthedocs.io/en/latest/) optimizer with different budget parameters.

### Super class

`mlr3tuning::Tuner` -> `TunerAsync`
Methods

Public methods:

- TunerAsync$optimize()
- TunerAsync$clone()

Method optimize(): Performs the tuning on a TuningInstanceAsyncSingleCrit or TuningInstanceAsyncMultiCrit until termination. The single evaluations will be written into the ArchiveAsyncTuning that resides in the TuningInstanceAsyncSingleCrit/TuningInstanceAsyncMultiCrit. The result will be written into the instance object.

Usage:
TunerAsync$optimize(inst)

Arguments:
inst (TuningInstanceAsyncSingleCrit | TuningInstanceAsyncMultiCrit).

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

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

Usage:
TunerAsync$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

Description

The TunerBatch implements the optimization algorithm.

Details

TunerBatch is an abstract base class that implements the base functionality each tuner must provide. A subclass is implemented in the following way:

- Inherit from Tuner.
- Specify the private abstract method $.optimize() and use it to call into your optimizer.
- You need to call instance$eval_batch() to evaluate design points.
- The batch evaluation is requested at the TuningInstanceBatchSingleCrit/TuningInstanceBatchMultiCrit object instance, so each batch is possibly executed in parallel via mlr3::benchmark(), and all evaluations are stored inside of instance$archive.
- Before the batch evaluation, the bbotk::Terminator is checked, and if it is positive, an exception of class "terminated_error" is generated. In the later case the current batch of evaluations is still stored in instance, but the numeric scores are not sent back to the handling optimizer as it has lost execution control.
• After such an exception was caught we select the best configuration from instance$archive and return it.
• Note that therefore more points than specified by the bbotk::Terminator may be evaluated, as the Terminator is only checked before a batch evaluation, and not in-between evaluation in a batch. How many more depends on the setting of the batch size.
• Overwrite the private super-method .assign_result() if you want to decide yourself how to estimate the final configuration in the instance and its estimated performance. The default behavior is: We pick the best resample-experiment, regarding the given measure, then assign its configuration and aggregated performance to the instance.

Private Methods

• .optimize(instance) -> NULL
  Abstract base method. Implement to specify tuning of your subclass. See details sections.
• .assign_result(instance) -> NULL
  Abstract base method. Implement to specify how the final configuration is selected. See details sections.

Resources

There are several sections about hyperparameter optimization in the mlr3book.

• An overview of all tuners can be found on our website.
• Learn more about tuners.

The gallery features a collection of case studies and demos about optimization.

• Use the Hyperband optimizer with different budget parameters.

Super class

mlr3tuning::Tuner -> TunerBatch

Methods

Public methods:
  • TunerBatch$new()
  • TunerBatch$optimize()
  • TunerBatch$clone()

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

Usage:
TunerBatch$new(
  id = "tuner_batch",
  param_set,
  param_classes,
  properties,
  packages = character(),
)
Arguments:

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

param_set (paradox::ParamSet)
   Set of control parameters.

param_classes (character())
   Supported parameter classes for learner hyperparameters that the tuner can optimize, as given in the paradox::ParamSet $class field.

properties (character())
   Set of properties of the tuner. Must be a subset of mlr_reflections$tuner_properties.

packages (character())
   Set of required packages. Note that these packages will be loaded via requireNamespace(), and are not attached.

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

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 optimize(): Performs the tuning on a TuningInstanceBatchSingleCrit or TuningInstanceBatchMultiCrit until termination. The single evaluations will be written into the ArchiveBatchTuning that resides in the TuningInstanceBatchSingleCrit/TuningInstanceBatchMultiCrit. The result will be written into the instance object.

Usage:
   TunerBatch$optimize(inst)

Arguments:
inst (TuningInstanceBatchSingleCrit | TuningInstanceBatchMultiCrit).

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

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

Usage:
   TunerBatch$clone(deep = FALSE)

Arguments:
   deep Whether to make a deep clone.
tune_nested  

Function for Nested Resampling

Description

Function to conduct nested resampling.

Usage

tune_nested(
  tuner,
  task,
  learner,
  inner_resampling,
  outer_resampling,
  measure = NULL,
  term_evals = NULL,
  term_time = NULL,
  terminator = NULL,
  search_space = NULL,
  store_tuning_instance = TRUE,
  store_benchmark_result = TRUE,
  store_models = FALSE,
  check_values = FALSE,
  callbacks = NULL
)

Arguments

tuner
  (Tuner)
  Optimization algorithm.

task
  (mlr3::Task)
  Task to operate on.

learner
  (mlr3::Learner)
  Learner to tune.

inner_resampling
  (mlr3::Resampling)
  Resampling used for the inner loop.

outer_resampling
  (mlr3::Resampling)
  Resampling used for the outer loop.

measure
  (mlr3::Measure)
  Measure to optimize. If NULL, default measure is used.

term_evals
  (integer(1))
  Number of allowed evaluations. Ignored if terminator is passed.
term_time (integer(1))
Maximum allowed time in seconds. Ignored if terminator is passed.

terminator (bbotk::Terminator)
Stop criterion of the tuning process.

search_space (paradox::ParamSet)
Hyperparameter search space. If NULL (default), the search space is constructed from the paradox::TuneToken of the learner’s parameter set (learner$param_set).

store_tuning_instance (logical(1))
If TRUE (default), stores the internally created TuningInstanceBatchSingleCrit with all intermediate results in slot $tuning_instance.

store_benchmark_result (logical(1))
If TRUE (default), store resample result of evaluated hyperparameter configurations in archive as mlr3::BenchmarkResult.

store_models (logical(1))
If TRUE, fitted models are stored in the benchmark result (archive$benchmark_result). If store_benchmark_result = FALSE, models are only stored temporarily and not accessible after the tuning. This combination is needed for measures that require a model.

check_values (logical(1))
If TRUE, hyperparameter values are checked before evaluation and performance scores after. If FALSE (default), values are unchecked but computational overhead is reduced.

callbacks (list of mlr3misc::Callback)
List of callbacks.

Value
mlr3::ResampleResult

Examples
# Nested resampling on Palmer Penguins data set
rr = tune_nested(  
  tuner = tnr("random_search", batch_size = 2),  
  task = tsk("penguins"),  
  learner = lrn("classif.rpart", cp = to_tune(1e-04, 1e-1, logscale = TRUE)),  
  inner_resampling = rsm("holdout"),  
  outer_resampling = rsm("cv", folds = 2),  
  measure = msr("classif.ce"),  
  term_evals = 2)

# Performance scores estimated on the outer resampling
rr$score()

# Unbiased performance of the final model trained on the full data set
rr$aggregate()
**TuningInstanceAsyncMultiCrit**

*Multi-Criteria Tuning with Rush*

---

**Description**

The TuningInstanceAsyncMultiCrit specifies a tuning problem for a Tuner. The function `ti_async()` creates a TuningInstanceAsyncMultiCrit and the function `tune()` creates an instance internally.

**Details**

The instance contains an ObjectiveTuningAsync object that encodes the black box objective function a Tuner has to optimize. The instance allows the basic operations of querying the objective at design points (`$eval_async()`). This operation is usually done by the Tuner. Hyperparameter configurations are asynchronously sent to workers and evaluated by calling `mlr3::resample()`. The evaluated hyperparameter configurations are stored in the ArchiveAsyncTuning (`$archive`). Before a batch is evaluated, the `bbotk::Terminator` is queried for the remaining budget. If the available budget is exhausted, an exception is raised, and no further evaluations can be performed from this point on. The tuner is also supposed to store its final result, consisting of a selected hyperparameter configuration and associated estimated performance values, by calling the method `instance$.assign_result`.

**Resources**

There are several sections about hyperparameter optimization in the mlr3book.

- Learn about multi-objective optimization.

The gallery features a collection of case studies and demos about optimization.

**Analysis**

For analyzing the tuning results, it is recommended to pass the ArchiveAsyncTuning to `as.data.table()`.

The returned data table contains the `mlr3::ResampleResult` for each hyperparameter evaluation.

**Super classes**

`bbotk::OptimInstance -> bbotk::OptimInstanceAsync -> bbotk::OptimInstanceAsyncMultiCrit -> TuningInstanceAsyncMultiCrit`

**Active bindings**

- `result_learner_param_vals (list())`
  
  List of param values for the optimal learner call.

- `internal_search_space (paradox::ParamSet)`
  
  The search space containing those parameters that are internally optimized by the `mlr3::Learner`.
Methods

Public methods:

- `TuningInstanceAsyncMultiCrit$new()`
- `TuningInstanceAsyncMultiCrit$assign_result()`
- `TuningInstanceAsyncMultiCrit$clone()`

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

Usage:

```
TuningInstanceAsyncMultiCrit$new(
  task,
  learner,
  resampling,
  measures,
  terminator,
  search_space = NULL,
  store_benchmark_result = TRUE,
  store_models = FALSE,
  check_values = FALSE,
  callbacks = NULL,
  rush = NULL
)
```

Arguments:

- `task` (mlr3::Task)
  Task to operate on.
- `learner` (mlr3::Learner)
  Learner to tune.
- `resampling` (mlr3::Resampling)
  Resampling that is used to evaluate the performance of the hyperparameter configurations. Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits. Already instantiated resamplings are kept unchanged. Specialized Tuner change the resampling e.g. to evaluate a hyperparameter configuration on different data splits. This field, however, always returns the resampling passed in construction.
- `measures` (list of mlr3::Measure)
  Measures to optimize.
- `terminator` (bbotk::Terminator)
  Stop criterion of the tuning process.
- `search_space` (paradox::ParamSet)
  Hyperparameter search space. If NULL (default), the search space is constructed from the paradox::TuneToken of the learner’s parameter set (learner$param_set).
- `store_benchmark_result` (logical(1))
  If TRUE (default), store resample result of evaluated hyperparameter configurations in archive as mlr3::BenchmarkResult.
- `store_models` (logical(1))
  If TRUE, fitted models are stored in the benchmark result (archive$benchmark_result). If
store_benchmark_result = FALSE, models are only stored temporarily and not accessible after the tuning. This combination is needed for measures that require a model.

check_values (logical(1))
If TRUE, hyperparameter values are checked before evaluation and performance scores after.
If FALSE (default), values are unchecked but computational overhead is reduced.

callbacks (list of mlr3misc::Callback)
List of callbacks.
rush (Rush)
If a rush instance is supplied, the tuning runs without batches.

Method assign_result(): The TunerAsync writes the best found points and estimated performance values here (probably the Pareto set / front). For internal use.

Usage:
TuningInstanceAsyncMultiCrit$assign_result(xdt, ydt, learner_param_vals = NULL)
Arguments:

xdt (data.table::data.table())
Hyperparameter values as data.table::data.table(). Each row is one configuration.
Contains values in the search space. Can contain additional columns for extra information.

ydt (numeric(1))
Optimal outcomes, e.g. the Pareto front.

learner_param_vals (List of named list())s
Fixed parameter values of the learner that are neither part of the

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

Usage:
TuningInstanceAsyncMultiCrit$clone(deep = FALSE)
Arguments:

deep Whether to make a deep clone.
TuningInstanceAsyncSingleCrit

Details

The instance contains an ObjectiveTuningAsync object that encodes the black box objective function a Tuner has to optimize. The instance allows the basic operations of querying the objective at design points ($eval_async()$). This operation is usually done by the Tuner. Hyperparameter configurations are asynchronously sent to workers and evaluated by calling \redirectinline{mlr3::resample()}. The evaluated hyperparameter configurations are stored in the ArchiveAsyncTuning ($archive$). Before a batch is evaluated, the bbotk::Terminator is queried for the remaining budget. If the available budget is exhausted, an exception is raised, and no further evaluations can be performed from this point on. The tuner is also supposed to store its final result, consisting of a selected hyperparameter configuration and associated estimated performance values, by calling the method instance$.assign_result$.

Default Measures

If no measure is passed, the default measure is used. The default measure depends on the task type.

<table>
<thead>
<tr>
<th>Task</th>
<th>Default Measure</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;classif&quot;</td>
<td>&quot;classif.ce&quot;</td>
<td>mlr3</td>
</tr>
<tr>
<td>&quot;regr&quot;</td>
<td>&quot;regr.mse&quot;</td>
<td>mlr3</td>
</tr>
<tr>
<td>&quot;surv&quot;</td>
<td>&quot;surv.cindex&quot;</td>
<td>mlr3proba</td>
</tr>
<tr>
<td>&quot;dens&quot;</td>
<td>&quot;dens.logloss&quot;</td>
<td>mlr3proba</td>
</tr>
<tr>
<td>&quot;classif_st&quot;</td>
<td>&quot;classif.ce&quot;</td>
<td>mlr3spatial</td>
</tr>
<tr>
<td>&quot;regr_st&quot;</td>
<td>&quot;regr.mse&quot;</td>
<td>mlr3spatial</td>
</tr>
<tr>
<td>&quot;clust&quot;</td>
<td>&quot;clust.dunn&quot;</td>
<td>mlr3cluster</td>
</tr>
</tbody>
</table>

Analysis

For analyzing the tuning results, it is recommended to pass the ArchiveAsyncTuning to as.data.table(). The returned data table contains the \redirectinline{mlr3::ResampleResult} for each hyperparameter evaluation.

Resources

There are several sections about hyperparameter optimization in the mlr3book.

- Getting started with hyperparameter optimization.
- Tune a simple classification tree on the Sonar data set.
- Learn about tuning spaces.

The gallery features a collection of case studies and demos about optimization.

- Learn more advanced methods with the practical tuning series.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Make use of proven search space.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
Extension Packages

mlr3tuning is extended by the following packages.

- mlr3tuningspaces is a collection of search spaces from scientific articles for commonly used learners.
- mlr3hyperband adds the Hyperband and Successive Halving algorithm.
- mlr3mbo adds Bayesian optimization methods.

Super classes

bbotk::OptimInstance -> bbotk::OptimInstanceAsync -> bbotk::OptimInstanceAsyncSingleCrit
-> TuningInstanceAsyncSingleCrit

Active bindings

result_learner_param_vals (list())
Param values for the optimal learner call.

internal_search_space (paradox::ParamSet)
The search space containing those parameters that are internally optimized by the mlr3::Learner.

Methods

Public methods:

- TuningInstanceAsyncSingleCrit$new()
- TuningInstanceAsyncSingleCrit$assign_result()
- TuningInstanceAsyncSingleCrit$clone()

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

Usage:
TuningInstanceAsyncSingleCrit$new(
  task,
  learner,
  resampling,
  measure = NULL,
  terminator,
  search_space = NULL,
  store_benchmark_result = TRUE,
  store_models = FALSE,
  check_values = FALSE,
  callbacks = NULL,
  rush = NULL
)

Arguments:

- task (mlr3::Task)
  Task to operate on.
- learner (mlr3::Learner)
  Learner to tune.
resampling (mlr3::Resampling)
Resampling that is used to evaluate the performance of the hyperparameter configurations. Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits. Already instantiated resamplings are kept unchanged. Specialized Tuner change the resampling e.g. to evaluate a hyperparameter configuration on different data splits. This field, however, always returns the resampling passed in construction.

measure (mlr3::Measure)
Measure to optimize. If NULL, default measure is used.

terminator (bbotk::Terminator)
Stop criterion of the tuning process.

search_space (paradox::ParamSet)
Hyperparameter search space. If NULL (default), the search space is constructed from the paradox::TuneToken of the learner’s parameter set (learner$param_set).

store_benchmark_result (logical(1))
If TRUE (default), store resample result of evaluated hyperparameter configurations in archive as mlr3::BenchmarkResult.

store_models (logical(1))
If TRUE, fitted models are stored in the benchmark result (archive$benchmark_result). If store_benchmark_result = FALSE, models are only stored temporarily and not accessible after the tuning. This combination is needed for measures that require a model.

check_values (logical(1))
If TRUE, hyperparameter values are checked before evaluation and performance scores after. If FALSE (default), values are unchecked but computational overhead is reduced.

callbacks (list of mlr3misc::Callback)
List of callbacks.

rush (Rush)
If a rush instance is supplied, the tuning runs without batches.

Method assign_result(): The TunerAsync object writes the best found point and estimated performance value here. For internal use.

Usage:
TuningInstanceAsyncSingleCrit$assign_result(xdt, y, learner_param_vals = NULL)

Arguments:
xdt (data.table::data.table())
Hyperparameter values as data.table::data.table(). Each row is one configuration. Contains values in the search space. Can contain additional columns for extra information.
y (numeric(1))
Optimal outcome.
learner_param_vals (List of named list())s
Fixed parameter values of the learner that are neither part of the

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

Usage:
TuningInstanceAsyncSingleCrit$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.
TuningInstanceBatchMultiCrit

Class for Multi Criteria Tuning

Description

The TuningInstanceBatchMultiCrit specifies a tuning problem for a Tuner. The function \texttt{ti()} creates a TuningInstanceBatchMultiCrit and the function \texttt{tune()} creates an instance internally.

Details

The instance contains an \texttt{ObjectiveTuningBatch} object that encodes the black box objective function a Tuner has to optimize. The instance allows the basic operations of querying the objective at design points (\texttt{eval\_batch()}). This operation is usually done by the Tuner. Evaluations of hyperparameter configurations are performed in batches by calling \texttt{mlr3::benchmark()} internally. The evaluated hyperparameter configurations are stored in the \texttt{ArchiveBatchTuning (archive)}. Before a batch is evaluated, the \texttt{bbotk::Terminator} is queried for the remaining budget. If the available budget is exhausted, an exception is raised, and no further evaluations can be performed from this point on. The tuner is also supposed to store its final result, consisting of a selected hyperparameter configuration and associated estimated performance values, by calling the method \texttt{instance assign\_result}.

Resources

There are several sections about hyperparameter optimization in the mlr3book.

- Learn about multi-objective optimization.

The gallery features a collection of case studies and demos about optimization.

Analysis

For analyzing the tuning results, it is recommended to pass the ArchiveBatchTuning to \texttt{as.data.table()}. The returned data table is joined with the benchmark result which adds the \texttt{mlr3::ResampleResult} for each hyperparameter evaluation.

The archive provides various getters (e.g. \texttt{$learners()}) to ease the access. All getters extract by position (\texttt{i}) or unique hash (\texttt{uhash}). For a complete list of all getters see the methods section.

The benchmark result (\texttt{$benchmark\_result}) allows to score the hyperparameter configurations again on a different measure. Alternatively, measures can be supplied to \texttt{as.data.table()}

The mlr3viz package provides visualizations for tuning results.

Super classes

\texttt{bbotk::OptimInstance -> bbotk::OptimInstanceBatch -> bbotk::OptimInstanceBatchMultiCrit -> TuningInstanceBatchMultiCrit}
**TuningInstanceBatchMultiCrit**

**Active bindings**

- `result_learner_param_vals (list())`
  - List of param values for the optimal learner call.
- `internal_search_space (paradox::ParamSet)`
  - The search space containing those parameters that are internally optimized by the `mlr3::Learner`.

**Methods**

**Public methods:**

- `TuningInstanceBatchMultiCrit$new()`  
- `TuningInstanceBatchMultiCrit$assign_result()`  
- `TuningInstanceBatchMultiCrit$clone()`

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

*Usage:*

```r
TuningInstanceBatchMultiCrit$new(  
  task,  
  learner,  
  resampling,  
  measures,  
  terminator,  
  search_space = NULL,  
  store_benchmark_result = TRUE,  
  store_models = FALSE,  
  check_values = FALSE,  
  callbacks = NULL  
)
```

*Arguments:*

- `task (mlr3::Task)`
  - Task to operate on.
- `learner (mlr3::Learner)`
  - Learner to tune.
- `resampling (mlr3::Resampling)`
  - Resampling that is used to evaluate the performance of the hyperparameter configurations. Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits. Already instantiated resamplings are kept unchanged. Specialized Tuner change the resampling e.g. to evaluate a hyperparameter configuration on different data splits. This field, however, always returns the resampling passed in construction.
- `measures` (list of `mlr3::Measure`)
  - Measures to optimize.
- `terminator (bbotk::Terminator)`
  - Stop criterion of the tuning process.
- `search_space (paradox::ParamSet)`
  - Hyperparameter search space. If NULL (default), the search space is constructed from the `paradox::TuneToken` of the learner’s parameter set (learner$param_set).
store_benchmark_result (logical(1))
  If TRUE (default), store resample result of evaluated hyperparameter configurations in archive as mlr3::BenchmarkResult.
store_models (logical(1))
  If TRUE, fitted models are stored in the benchmark result (archive$benchmark_result). If store_benchmark_result = FALSE, models are only stored temporarily and not accessible after the tuning. This combination is needed for measures that require a model.
check_values (logical(1))
  If TRUE, hyperparameter values are checked before evaluation and performance scores after. If FALSE (default), values are unchecked but computational overhead is reduced.
callbacks (list of mlr3misc::Callback)
  List of callbacks.

Method assign_result(): The Tuner object writes the best found points and estimated performance values here. For internal use.

Usage:
TuningInstanceBatchMultiCrit$assign_result(xdt, ydt, learner_param_vals = NULL)

Arguments:
xdt (data.table::data.table())
  Hyperparameter values as data.table::data.table(). Each row is one configuration. Contains values in the search space. Can contain additional columns for extra information.
ydt (data.table::data.table())
  Optimal outcomes, e.g. the Pareto front.
learner_param_vals (List of named list())s
  Fixed parameter values of the learner that are neither part of the

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

Usage:
TuningInstanceBatchMultiCrit$clone(deep = FALSE)

Arguments:
  deep Whether to make a deep clone.

Examples

# Hyperparameter optimization on the Palmer Penguins data set
task = tsk("penguins")

# Load learner and set search space
learner = lrn("classif.rpart",
  cp = to_tune(1e-04, 1e-1, logscale = TRUE)
)

# Construct tuning instance
instance = ti(
  task = task,
  learner = learner,
  resampling = rsmp("cv", folds = 3),
)
TuningInstanceBatchSingleCrit

Class for Single Criterion Tuning

Description

The TuningInstanceBatchSingleCrit specifies a tuning problem for a Tuner. The function ti() creates a TuningInstanceBatchSingleCrit and the function tune() creates an instance internally.

Details

The instance contains an ObjectiveTuningBatch object that encodes the black box objective function a Tuner has to optimize. The instance allows the basic operations of querying the objective at design points ($eval_batch()). This operation is usually done by the Tuner. Evaluations of hyperparameter configurations are performed in batches by calling mlr3::benchmark() internally. The evaluated hyperparameter configurations are stored in the ArchiveBatchTuning ($archive). Before a batch is evaluated, the bbotk::Terminator is queried for the remaining budget. If the available budget is exhausted, an exception is raised, and no further evaluations can be performed from this point on. The tuner is also supposed to store its final result, consisting of a selected hyperparameter configuration and associated estimated performance values, by calling the method instance$assign_result.

Default Measures

If no measure is passed, the default measure is used. The default measure depends on the task type.

```
<table>
<thead>
<tr>
<th>Task</th>
<th>Default Measure</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;classif&quot;</td>
<td>&quot;classif.ce&quot;</td>
<td>mlr3</td>
</tr>
<tr>
<td>&quot;regr&quot;</td>
<td>&quot;regr.mse&quot;</td>
<td>mlr3</td>
</tr>
<tr>
<td>&quot;surv&quot;</td>
<td>&quot;surv.cindex&quot;</td>
<td>mlr3proba</td>
</tr>
<tr>
<td>&quot;dens&quot;</td>
<td>&quot;dens.logloss&quot;</td>
<td>mlr3proba</td>
</tr>
<tr>
<td>&quot;classif_st&quot;</td>
<td>&quot;classif.ce&quot;</td>
<td>mlr3spatial</td>
</tr>
</tbody>
</table>
```
Resources

There are several sections about hyperparameter optimization in the mlr3book.

- Getting started with hyperparameter optimization.
- Tune a simple classification tree on the Sonar data set.
- Learn about tuning spaces.

The gallery features a collection of case studies and demos about optimization.

- Learn more advanced methods with the practical tuning series.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Make use of proven search space.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.

Extension Packages

mlr3tuning is extended by the following packages.

- mlr3tuningspaces is a collection of search spaces from scientific articles for commonly used learners.
- mlr3hyperband adds the Hyperband and Successive Halving algorithm.
- mlr3mbo adds Bayesian optimization methods.

Analysis

For analyzing the tuning results, it is recommended to pass the ArchiveBatchTuning to as.data.table(). The returned data table is joined with the benchmark result which adds the mlr3::ResampleResult for each hyperparameter evaluation.

The archive provides various getters (e.g. $learners()) to ease the access. All getters extract by position (i) or unique hash (uhash). For a complete list of all getters see the methods section.

The benchmark result ($benchmark_result) allows to score the hyperparameter configurations again on a different measure. Alternatively, measures can be supplied to as.data.table().

The mlr3viz package provides visualizations for tuning results.

Super classes

bbotk::OptimInstance -> bbotk::OptimInstanceBatch -> bbotk::OptimInstanceBatchSingleCrit -> TuningInstanceBatchSingleCrit
Active bindings

result_learner_param_vals (list())
Param values for the optimal learner call.

internal_search_space (paradox::ParamSet)
The search space containing those parameters that are internally optimized by the mlr3::Learner.

Methods

Public methods:
• TuningInstanceBatchSingleCrit$new()
• TuningInstanceBatchSingleCrit$assign_result()
• TuningInstanceBatchSingleCrit$clone()

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

Usage:
TuningInstanceBatchSingleCrit$new(
  task,
  learner,
  resampling,
  measure = NULL,
  terminator,
  search_space = NULL,
  store_benchmark_result = TRUE,
  store_models = FALSE,
  check_values = FALSE,
  callbacks = NULL
)

Arguments:
task (mlr3::Task)
  Task to operate on.
learner (mlr3::Learner)
  Learner to tune.
resampling (mlr3::Resampling)
  Resampling that is used to evaluate the performance of the hyperparameter configurations. Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits. Already instantiated resamplings are kept unchanged. Specialized Tuner change the resampling e.g. to evaluate a hyperparameter configuration on different data splits. This field, however, always returns the resampling passed in construction.
measure (mlr3::Measure)
  Measure to optimize. If NULL, default measure is used.
terminator (bbotk::Terminator)
  Stop criterion of the tuning process.
search_space (paradox::ParamSet)
  Hyperparameter search space. If NULL (default), the search space is constructed from the paradox::TuneToken of the learner’s parameter set (learner$param_set).
store_benchmark_result (logical(1))
  If TRUE (default), store resample result of evaluated hyperparameter configurations in archive as mlr3::BenchmarkResult.

store_models (logical(1))
  If TRUE, fitted models are stored in the benchmark result (archive$benchmark_result). If store_benchmark_result = FALSE, models are only stored temporarily and not accessible after the tuning. This combination is needed for measures that require a model.

check_values (logical(1))
  If TRUE, hyperparameter values are checked before evaluation and performance scores after.
  If FALSE (default), values are unchecked but computational overhead is reduced.

callbacks (list of mlr3misc::Callback)
  List of callbacks.

**Method** assign_result(): The Tuner object writes the best found point and estimated performance value here. For internal use.

*Usage:*
TuningInstanceBatchSingleCrit$assign_result(xdt, y, learner_param_vals = NULL)

*Arguments:*
  
  **xdt** (data.table::data.table())
  Hyperparameter values as data.table::data.table(). Each row is one configuration.
  Contains values in the search space. Can contain additional columns for extra information.

  **y** (numeric(1))
  Optimal outcome.

  **learner_param_vals** (List of named list())
  Fixed parameter values of the learner that are neither part of the

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

*Usage:*
TuningInstanceBatchSingleCrit$clone(deep = FALSE)

*Arguments:*
  
  **deep** Whether to make a deep clone.

**Examples**

# Hyperparameter optimization on the Palmer Penguins data set
task = tsk("penguins")

# Load learner and set search space
learner = lrn("classif.rpart",
  cp = to_tune(1e-04, 1e-1, logscale = TRUE)
)

# Construct tuning instance
instance = ti(
  task = task,
  learner = learner,
  resampling = rsmp("cv", folds = 3),
)
TuningInstanceMultiCrit

```r
measures = msr("classif.ce"),
terminator = trm("evals", n_evals = 4)

# Choose optimization algorithm
tuner = tnr("random_search", batch_size = 2)

# Run tuning
tuner$optimize(instance)

# Set optimal hyperparameter configuration to learner
learner$param_set$values = instance$result_learner_param_vals

# Train the learner on the full data set
learner$train(task)

# Inspect all evaluated configurations
as.data.table(instance$archive)
```

TuningInstanceMultiCrit

Multi Criteria Tuning Instance for Batch Tuning

Description

TuningInstanceMultiCrit is a deprecated class that is now a wrapper around TuningInstanceBatchMultiCrit.

Super classes

bbotk::OptimInstance -> bbotk::OptimInstanceBatch -> bbotk::OptimInstanceBatchMultiCrit -> mlr3tuning::TuningInstanceBatchMultiCrit -> TuningInstanceMultiCrit

Methods

Public methods:

- TuningInstanceMultiCrit$new()
- TuningInstanceMultiCrit$clone()

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

Usage:

```r
TuningInstanceMultiCrit$new(
  task,
  learner,
  resampling,
  measures,
  terminator,
  search_space = NULL,
)```
store_benchmark_result = TRUE,
store_models = FALSE,
check_values = FALSE,
callbacks = NULL
)

Arguments:
task (mlr3::Task)
  Task to operate on.
learner (mlr3::Learner)
  Learner to tune.
resampling (mlr3::Resampling)
  Resampling that is used to evaluate the performance of the hyperparameter configurations. Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits. Already instantiated resamplings are kept unchanged. Specialized Tuner change the resampling e.g. to evaluate a hyperparameter configuration on different data splits. This field, however, always returns the resampling passed in construction.
measures (list of mlr3::Measure)
  Measures to optimize.
terminator (bbotk::Terminator)
  Stop criterion of the tuning process.
search_space (paradox::ParamSet)
  Hyperparameter search space. If NULL (default), the search space is constructed from the paradox::TuneToken of the learner's parameter set (learner$param_set).
store_benchmark_result (logical(1))
  If TRUE (default), store resample result of evaluated hyperparameter configurations in archive as mlr3::BenchmarkResult.
store_models (logical(1))
  If TRUE, fitted models are stored in the benchmark result (archive$benchmark_result). If store_benchmark_result = FALSE, models are only stored temporarily and not accessible after the tuning. This combination is needed for measures that require a model.
check_values (logical(1))
  If TRUE, hyperparameter values are checked before evaluation and performance scores after. If FALSE (default), values are unchecked but computational overhead is reduced.
callbacks (list of mlr3misc::Callback)
  List of callbacks.

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

Usage:
TuningInstanceMultiCrit$clone(deep = FALSE)

Arguments:
deep  Whether to make a deep clone.
TuningInstanceSingleCrit

Single Criterion Tuning Instance for Batch Tuning

Description

TuningInstanceSingleCrit is a deprecated class that is now a wrapper around TuningInstanceBatchSingleCrit.

Super classes

- bbotk::OptimInstance -> bbotk::OptimInstanceBatch -> bbotk::OptimInstanceBatchSingleCrit
- mlr3tuning::TuningInstanceBatchSingleCrit -> TuningInstanceSingleCrit

Methods

Public methods:
- TuningInstanceSingleCrit$new()  
- TuningInstanceSingleCrit$clone()

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

Usage:

TuningInstanceSingleCrit$new(
  task,
  learner,
  resampling,
  measure = NULL,
  terminator,
  search_space = NULL,
  store_benchmark_result = TRUE,
  store_models = FALSE,
  check_values = FALSE,
  callbacks = NULL
)

Arguments:

- task (mlr3::Task)
  Task to operate on.
- learner (mlr3::Learner)
  Learner to tune.
- resampling (mlr3::Resampling)
  Resampling that is used to evaluate the performance of the hyperparameter configurations. Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits. Already instantiated resamplings are kept unchanged. Specialized Tuner change the resampling e.g. to evaluate a hyperparameter configuration on different data splits. This field, however, always returns the resampling passed in construction.
measure (mlr3::Measure)
  Measure to optimize. If NULL, default measure is used.

terminator (bbotk::Terminator)
  Stop criterion of the tuning process.

search_space (paradox::ParamSet)
  Hyperparameter search space. If NULL (default), the search space is constructed from the
  paradox::TuneToken of the learner's parameter set (learner$param_set).

store_benchmark_result (logical(1))
  If TRUE (default), store resample result of evaluated hyperparameter configurations in archive
  as mlr3::BenchmarkResult.

store_models (logical(1))
  If TRUE, fitted models are stored in the benchmark result (archive$benchmark_result). If
  store_benchmark_result = FALSE, models are only stored temporarily and not accessible
  after the tuning. This combination is needed for measures that require a model.

check_values (logical(1))
  If TRUE, hyperparameter values are checked before evaluation and performance scores after.
  If FALSE (default), values are unchecked but computational overhead is reduced.

callbacks (list of mlr3misc::Callback)
  List of callbacks.

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

Usage:
  TuningInstanceSingleCrit$clone(deep = FALSE)

Arguments:
  deep  Whether to make a deep clone.
Index

* Dictionary
  * mlr_tuners, 36
* TunerAsync
  * mlr_tuners_async_design_points, 37
  * mlr_tuners_async_grid_search, 38
  * mlr_tuners_async_random_search, 39
* Tuner
  * mlr_tuners, 36
  * mlr_tuners_cmaes, 40
  * mlr_tuners_design_points, 42
  * mlr_tuners_genSA, 44
  * mlr_tuners_grid_search, 47
  * mlr_tuners_internal, 49
  * mlr_tuners_irace, 51
  * mlr_tuners_nloptr, 54
  * mlr_tuners_random_search, 57
  * Tuner, 73
* datasets
  * mlr_tuners, 36
  * adagio::pureCMAES(), 40
  * ArchiveAsyncTuning, 4, 4, 5, 70, 77, 82, 85
  * ArchiveBatchTuning, 8, 8, 9, 15, 35, 52, 53, 62, 63, 70, 73, 79, 88, 91, 92
  * as_search_space, 12
  * as_tuner, 12
  * as_tuners(as_tuner), 12
  * auto_tuner, 19
  * auto_tuner(), 13, 19
  * AutoTuner, 13, 14, 19, 21, 22, 30, 32, 63
  * bbotk::Archive, 5, 9
  * bbotk::ArchiveAsync, 5
  * bbotk::ArchiveBatch, 9
  * bbotk::CallbackAsync, 22
  * bbotk::CallbackBatch, 23
  * bbotk::Codomain, 6, 10
  * bbotk::ContextAsync, 28
  * bbotk::ContextBatch, 29
  * bbotk::Objective, 59, 61, 62
  * bbotk::OptimInstance, 82, 86, 88, 92, 95, 97
  * bbotk::OptimInstanceAsync, 82, 86
  * bbotk::OptimInstanceAsyncMultiCrit, 82
  * bbotk::OptimInstanceAsyncSingleCrit, 86
  * bbotk::OptimInstanceBatch, 88, 92, 95, 97
  * bbotk::OptimInstanceBatchMultiCrit, 88, 95
  * bbotk::OptimInstanceBatchSingleCrit, 92, 97
  * bbotk::OptimizerBatchCmaes, 40
  * bbotk::OptimizerBatchDesignPoints, 42
  * bbotk::OptimizerBatchGenSA, 45
  * bbotk::OptimizerBatchGridSearch, 48
  * bbotk::OptimizerBatchIrace, 52
  * bbotk::OptimizerBatchNloptr, 55
  * bbotk::OptimizerBatchRandomSearch, 57
  * bbotk::TerminatorCombo, 72
  * bbotk::TerminatorEvals, 52
  * callback_async_tuning, 24
  * callback_async_tuning(), 22, 28
  * callback_batch_tuning, 26
  * callback_batch_tuning(), 23, 29
  * CallbackAsyncTuning, 22, 22, 24, 28, 34
  * CallbackBatchTuning, 23, 23, 26, 29, 34
callback(), 22–24, 26
  * ContextAsyncTuning, 24, 25, 28
  * ContextBatchTuning, 26, 27, 29
  * data.table::data.table, 15, 29, 37, 43, 53
data.table::data.table(), 5, 8, 9, 30, 32, 36, 77, 79
  * dictionary, 22–24, 26, 70
  * extract_inner_tuning_archives, 30
  * extract_inner_tuning_results, 31
INDEX

paradox::TuneToken, 6, 10, 16, 20, 65, 67, 71, 72, 81, 83, 87, 89, 93, 96, 98

R6, 5, 9, 15, 37–39, 41, 43, 46, 48, 50, 53, 56, 58, 60, 62, 75, 78, 83, 86, 89, 93, 95, 97
R6::R6Class, 36, 70
requireNamespace(), 74, 75
rush::Rush, 4

set_validate.AutoTuner, 63

Terminator, 43, 46, 55, 58
Terminators, 69, 72
ti, 64
ti(), 88, 91
ti_async, 67
ti_async(), 82, 84
tnr, 69
tnr(), 36–40, 42, 45, 47, 50, 51, 54, 57
tnrs(tnr), 69
tnrs(), 36
tune, 70
tune(), 82, 84, 88, 91
tune_nested, 80
Tuner, 12–14, 16, 20, 21, 36–43, 45–58, 60, 63, 65, 67, 69–71, 73, 80, 82, 83, 85, 87–91, 93, 94, 96, 97
TunerAsync, 76, 76, 84, 87
TunerAsyncDesignPoints
   (mlr_tuners_async_design_points), 37
TunerAsyncGridSearch
   (mlr_tuners_async_grid_search), 38
TunerAsyncRandomSearch
   (mlr_tuners_async_random_search), 39
TunerBatch, 77, 77
TunerBatchCmaes (mlr_tuners_cmaes), 40
TunerBatchDesignPoints
   (mlr_tuners_design_points), 42
TunerBatchGenSA (mlr_tuners_gensa), 44
TunerBatchGridSearch
   (mlr_tuners_grid_search), 47
TunerBatchInternal
   (mlr_tuners_internal), 49
TunerBatchIrace (mlr_tuners_irace), 51
TunerBatchNloptr (mlr_tuners_nloptr), 54
TunerBatchRandomSearch
   (mlr_tuners_random_search), 57
Tuners, 69
TuningInstanceAsyncMultiCrit, 67, 77, 82, 82
TuningInstanceAsyncSingleCrit, 15, 67, 77, 84, 84
TuningInstanceBatchMultiCrit, 32, 59, 61–65, 70–72, 77, 79, 88, 88, 95
TuningInstanceMultiCrit, 95
TuningInstanceSingleCrit, 97