Package ‘mlr3tuning’

July 23, 2025

Title Hyperparameter Optimization for 'mlr3'

Version 1.4.0

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 (>= 0.23.0), paradox (>= 1.0.1), R (>= 3.1.0)

Imports bbotk (>= 1.6.0), checkmate (>= 2.0.0), cli, data.table, lgr, mlr3misc (>= 0.15.1), R6

Suggests adagio, future, GenSA, irace (>= 4.1.0), knitr, mirai, 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 7.3.2

Collate 'ArchiveAsyncTuning.R' 'ArchiveAsyncTuningFrozen.R' 'ArchiveBatchTuning.R' 'AutoTuner.R' 'CallbackAsyncTuning.R' 'CallbackBatchTuning.R' 'ContextAsyncTuning.R' 'ContextBatchTuning.R' 'ObjectiveTuning.R'

'ObjectiveTuningAsync.R' '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'
'TuningInstanceAsyncMulticrit.R'
'TuningInstanceAsyncSingleCrit.R'
'TuningInstanceBatchMulticrit.R' 'TuningInstanceMultiCrit.R'
'TuningInstanceSingleCrit.R' 'as_search_space.R' 'as_tuner.R'
'assertions.R' 'auto_tuner.R' 'bibentries.R'
'extract_inner_tuning_archives.R'
'extract_inner_tuning_results.R' 'helper.R' 'mlr_callbacks.R'
'reexport.R' 'sugar.R' 'tune.R' 'tune_nested.R' 'zzz.R'

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

Maintainer Marc Becker <marcbecker@posteo.de>

Repository CRAN

Date/Publication 2025-06-04 11:10:02 UTC

Contents

`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.

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

---

`ArchiveAsyncTuning` *Rush Data Storage*

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.
  ArchiveAsyncTuning -> `data.table::data.table()`
  - `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.

`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.

`ArchiveAsyncTuningFrozen` *Frozen Rush Data Storage*

Description

Freezes the Redis data base of an ArchiveAsyncTuning to a `data.table::data.table()`. No further points can be added to the archive but the data can be accessed and analyzed. Useful when the Redis data base is not permanently available. Use the callback mlr3tuning.async_freeze_archive to freeze the archive after the optimization has finished.

S3 Methods

- `as.data.table(archive)`
  ArchiveAsyncTuningFrozen -> `data.table::data.table()`
  Returns a tabular view of all performed function calls of the Objective. The `x_domain` column is unnested to separate columns.

Super classes

`bbotk::Archive` -> `bbotk::ArchiveAsync` -> `bbotk::ArchiveAsyncFrozen` -> `ArchiveAsyncTuningFrozen`

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:

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

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

Usage:

```
ArchiveAsyncTuningFrozen$new(archive)
```

Arguments:

`archive` (ArchiveAsyncTuning)
The archive to freeze.

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:

```
ArchiveAsyncTuningFrozen$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:

```
ArchiveAsyncTuningFrozen$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:

```
ArchiveAsyncTuningFrozen$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:

```
ArchiveAsyncTuningFrozen$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:

```
ArchiveAsyncTuningFrozen$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:

```
ArchiveAsyncTuningFrozen$print()
```

Arguments:

`...` (ignored).

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

Usage:

```
ArchiveAsyncTuningFrozen$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 (`$benchmark_result`) that records the resampling experiments. Each experiment corresponds to to a single evaluation of a hyperparameter configuration. The table (`$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 **mlr3viz** package provides visualizations for tuning results.

S3 Methods

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

  - `x` (ArchiveBatchTuning)
  - `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::ArchiveBatch -> ArchiveBatchTuning

Public fields

`benchmark_result` (mlr3::BenchmarkResult)
Benchmark result.

Active bindings

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

Methods

Public methods:

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

Method `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` (bbotk::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.

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:

```
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:

```
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:

```
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 mlr3::Prediction objects of the i-th evaluation, by position or by unique hash `uhash`. `i` and `uhash` are mutually exclusive.

Usage:

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

Usage:

```
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:

```
ArchiveBatchTuning$print()
```

Arguments:

`...` (ignored).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`assert_async_tuning_callback` *Assertions for Callbacks*

Description

Assertions for CallbackAsyncTuning class.

Usage

```
assert_async_tuning_callback(callback, null_ok = FALSE)

assert_async_tuning_callbacks(callbacks)
```

Arguments

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

Value

[CallbackAsyncTuning | List of CallbackAsyncTunings.

`assert_batch_tuning_callback` *Assertions for Callbacks*

Description

Assertions for CallbackBatchTuning class.

Usage

```
assert_batch_tuning_callback(callback, null_ok = FALSE)

assert_batch_tuning_callbacks(callbacks)
```

Arguments

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

Value

[CallbackBatchTuning | List of CallbackBatchTunings.

`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:
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

The `AutoTuner` itself does **not** have the `"validation"` property. To enable validation during the tuning, set the `$validate` field of the tuned learner. This is also possible via `set_validate()`.

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.

Task	Default Measure	Package
`"classif"`	`"classif.ce"`	**mlr3**
`"regr"`	`"regr.mse"`	**mlr3**
`"surv"`	`"surv.cindex"`	**mlr3proba**

"dens"	"dens.logloss"	**mlr3proba**
"classif_st"	"classif.ce"	**mlr3spatial**
"regr_st"	"regr.mse"	**mlr3spatial**
"clust"	"clust.dunn"	**mlr3cluster**

Resources

There are several sections about hyperparameter optimization in the mlr3book.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

Super class

`mlr3::Learner` -> `AutoTuner`

Public fields

`instance_args` (`list()`)
: All arguments from construction to create the TuningInstanceBatchSingleCrit.

`tuner` (Tuner)
: Optimization algorithm.

Active bindings

`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`. A few learners already use the predict type during training. So there is no guarantee that changing the predict type after tuning and training will have any effect or does not lead to errors.

`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,
  rush = NULL,
  id = 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.

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

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(...)
```

Arguments:

`...` (any)
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:

`deep` Whether to make a deep clone.

Examples

```
# 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

# shortcut tuning instance
at$tuning_instance


# Nested Resampling
```

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

resampling_outer = rsmp("cv", folds = 3)
rr = resample(task, at, resampling_outer, store_models = TRUE)

# retrieve inner tuning results.
extract_inner_tuning_results(rr)

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

# unbiased performance of the final model trained on the full data set
rr$aggregate()
```

`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,
  rush = NULL,
  id = 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.
- `rush` — (`Rush`)
  If a rush instance is supplied, the tuning runs without batches.
- `id` — (`character(1)`)
  Identifier for the new instance.

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.

Value

AutoTuner.

Default Measures

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

Task	Default Measure	Package
`"classif"`	`"classif.ce"`	**mlr3**
`"regr"`	`"regr.mse"`	**mlr3**
`"surv"`	`"surv.cindex"`	**mlr3proba**
`"dens"`	`"dens.logloss"`	**mlr3proba**
`"classif_st"`	`"classif.ce"`	**mlr3spatial**
`"regr_st"`	`"regr.mse"`	**mlr3spatial**
`"clust"`	`"clust.dunn"`	**mlr3cluster**

Resources

There are several sections about hyperparameter optimization in the mlr3book.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

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

```
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` *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 ObjectiveTuningAsync$eval().

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

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

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

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

on_eval_after_resample (function())
: Stage called after hyperparameter configurations are evaluated. Called in ObjectiveTuningAsync$eval().

on_eval_before_archive (function())
: Stage called before performance values are written to the archive. Called in ObjectiveTuningAsync$eval().

on_tuning_result_begin (function())
: Stage called before the results are written. Called in TuningInstance*$assign_result().

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 ObjectiveTuningBatch$eval_many().

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

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

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

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

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

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

on_tuning_result_begin (function())
: Stage called before the results are written. Called in TuningInstance*$assign_result().

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

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

callback_async_tuning *Create Asynchronous Tuning Callback*

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 Optimization on Worker
           - on_optimizer_before_eval
             Start Evaluation
               - on_eval_after_xs
                 Start Resampling Iteration
                   - on_resample_begin
                   - on_resample_before_train
                   - on_resample_before_predict
                   - on_resample_end
                 End Resampling Iteration
               - on_eval_after_resample
               - on_eval_before_archive
             End Evaluation
          - on_optimizer_after_eval
         End Optimization on Worker
         - on_worker_end
    End Worker
     - on_tuning_result_begin
     - on_result_begin
     - on_result_end
     - on_optimization_end
End Tuning
```

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

Usage

```
callback_async_tuning(
  id,
  label = NA_character_,
  man = NA_character_,
```

```
  on_optimization_begin = NULL,
  on_worker_begin = NULL,
  on_optimizer_before_eval = NULL,
  on_eval_after_xs = NULL,
  on_resample_begin = NULL,
  on_resample_before_train = NULL,
  on_resample_before_predict = NULL,
  on_resample_end = NULL,
  on_eval_after_resample = NULL,
  on_eval_before_archive = NULL,
  on_optimizer_after_eval = NULL,
  on_worker_end = NULL,
  on_tuning_result_begin = NULL,
  on_result_begin = NULL,
  on_result_end = NULL,
  on_result = NULL,
  on_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()`. The functions must have two arguments named `callback` and `context`.

`on_worker_begin`
(`function()`)
Stage called at the beginning of the optimization on the worker. Called in the worker loop. The functions must have two arguments named `callback` and `context`.

`on_optimizer_before_eval`
(`function()`)
Stage called after the optimizer proposes points. Called in `OptimInstance$.eval_point()`. The functions must have two arguments named `callback` and `context`. The argument of `instance$.eval_point(xs)` and `xs_trafoed` and `extra` are available in the `context`. Or `xs` and `xs_trafoed` of `instance$.eval_queue()` are available in the `context`.

`on_eval_after_xs`
(`function()`)
Stage called after xs is passed to the objective. Called in `ObjectiveTuningAsync$eval()`.

The functions must have two arguments named `callback` and `context`. The argument of `$.eval(xs)` is available in the `context`.

`on_resample_begin`
: (`function()`)
Stage called at the beginning of a resampling iteration. Called in `workhorse()` (internal). See also `mlr3::callback_resample()`. The functions must have two arguments named `callback` and `context`.

`on_resample_before_train`
: (`function()`)
Stage called before training the learner. Called in `workhorse()` (internal). See also `mlr3::callback_resample()`. The functions must have two arguments named `callback` and `context`.

`on_resample_before_predict`
: (`function()`)
Stage called before predicting. Called in `workhorse()` (internal). See also `mlr3::callback_resample()`. The functions must have two arguments named `callback` and `context`.

`on_resample_end`
: (`function()`)
Stage called at the end of a resampling iteration. Called in `workhorse()` (internal). See also `mlr3::callback_resample()`. The functions must have two arguments named `callback` and `context`.

`on_eval_after_resample`
: (`function()`)
Stage called after a hyperparameter configuration is evaluated. Called in `ObjectiveTuningAsync$eval()`. The functions must have two arguments named `callback` and `context`. The `resample_result` is available in the ‘context

`on_eval_before_archive`
: (`function()`)
Stage called before performance values are written to the archive. Called in `ObjectiveTuningAsync$eval()`. The functions must have two arguments named `callback` and `context`. The `aggregated_performance` is available in `context`.

`on_optimizer_after_eval`
: (`function()`)
Stage called after points are evaluated. Called in `OptimInstance$.eval_point()`. The functions must have two arguments named `callback` and `context`.

`on_worker_end`
: (`function()`)
Stage called at the end of the optimization on the worker. Called in the worker loop. The functions must have two arguments named `callback` and `context`.

`on_tuning_result_begin`
: (`function()`)
Stage called at the beginning of the result writing. Called in `TuningInstance*$assign_result()`. The functions must have two arguments named `callback` and `context`. The arguments of `$assign_result(xdt, y, learner_param_vals, extra)` are available in `context`.

`on_result_begin`
: (`function()`)
Stage called at the beginning of the result writing. Called in `OptimInstance$assign_result()`.

The functions must have two arguments named `callback` and `context`. The arguments of `$.assign_result(xdt, y, extra)` are available in the `context`.

`on_result_end` (`function()`)
Stage called after the result is written. Called in `OptimInstance$assign_result()`. The functions must have two arguments named `callback` and `context`. The final result `instance$result` is available in the `context`.

`on_result` (`function()`)
Deprecated. Use `on_result_end` instead. 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 and mlr3::ContextResample.

`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
                 Start Resampling Iteration
                   - on_resample_begin
                   - on_resample_before_train
                   - on_resample_before_predict
                   - on_resample_end
                 End Resampling Iteration
             - on_eval_after_benchmark
             - on_eval_before_archive
        End Evaluation
         - on_optimizer_after_eval
    End Tuner Batch
```

```
    - on_tuning_result_begin
    - on_result_begin
    - on_result_end
    - on_optimization_end
End Tuning
```

See also the section on parameters for more information on the stages. A tuning callback works with ContextBatchTuning and mlr3::ContextResample.

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_resample_begin = NULL,
  on_resample_before_train = NULL,
  on_resample_before_predict = NULL,
  on_resample_end = NULL,
  on_eval_after_benchmark = NULL,
  on_eval_before_archive = NULL,
  on_optimizer_after_eval = NULL,
  on_tuning_result_begin = NULL,
  on_result_begin = NULL,
  on_result_end = NULL,
  on_result = NULL,
  on_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()`. The functions must have two arguments named `callback` and `context`.

`on_optimizer_before_eval`
(`function()`)
Stage called after the optimizer proposes points. Called in `OptimInstance$eval_batch()`.

The functions must have two arguments named `callback` and `context`. The argument of `$eval_batch(xdt)` is available in `context`.

`on_eval_after_design`
: (`function()`)
Stage called after the design is created. Called in `ObjectiveTuningBatch$eval_many()`. The functions must have two arguments named `callback` and `context`. The arguments of `$eval_many(xss, resampling)` are available in `context`. Additionally, the `design` is available in `context`.

`on_resample_begin`
: (`function()`)
Stage called at the beginning of a resampling iteration. Called in `workhorse()` (internal). See also `mlr3::callback_resample()`. The functions must have two arguments named `callback` and `context`.

`on_resample_before_train`
: (`function()`)
Stage called before training the learner. Called in `workhorse()` (internal). See also `mlr3::callback_resample()`. The functions must have two arguments named `callback` and `context`.

`on_resample_before_predict`
: (`function()`)
Stage called before predicting. Called in `workhorse()` (internal). See also `mlr3::callback_resample()`. The functions must have two arguments named `callback` and `context`.

`on_resample_end`
: (`function()`)
Stage called at the end of a resampling iteration. Called in `workhorse()` (internal). See also `mlr3::callback_resample()`. The functions must have two arguments named `callback` and `context`.

`on_eval_after_benchmark`
: (`function()`)
Stage called after hyperparameter configurations are evaluated. Called in `ObjectiveTuningBatch$eval_`
The functions must have two arguments named `callback` and `context`. The `benchmark_result` is available in `context`.

`on_eval_before_archive`
: (`function()`)
Stage called before performance values are written to the archive. Called in `ObjectiveTuningBatch$eval_many()`. The functions must have two arguments named `callback` and `context`. The `aggregated_performance` is available in `context`.

`on_optimizer_after_eval`
: (`function()`)
Stage called after points are evaluated. Called in `OptimInstance$eval_batch()`. The functions must have two arguments named `callback` and `context`. The new configurations and performances in `instance$archive` are available in `context`.

`on_tuning_result_begin`
: (`function()`)
Stage called at the beginning of the result writing. Called in `TuningInstanceBatch$assign_result()`.

The functions must have two arguments named callback and context. The arguments of $assign_result(xdt, y, learner_param_vals, extra) are available in context.

on_result_begin
(function())
Stage called at the beginning of the result writing. Called in OptimInstance$assign_result(). The functions must have two arguments named callback and context. The arguments of $assign_result(xdt, y, extra) are available in context.

on_result_end
(function())
Stage called after the result is written. Called in OptimInstance$assign_result(). The functions must have two arguments named callback and context. The final result instance$result is available in context.

on_result
(function())
Deprecated. Use on_result_end instead. Stage called after the result is written. Called in OptimInstance$assign_result(). The functions must have two arguments named callback and context.

on_optimization_end
(function())
Stage called at the end of the optimization. Called in Optimizer$optimize(). The functions must have two arguments named callback and context.

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 ContextBatchTuning.

Examples

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

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_learner` (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.

`result_learner_param_vals` (list())
: The learner parameter values passed to `instance$assign_result()`.

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.

`result_learner_param_vals` (list())
: The learner parameter values passed to `instance$assign_result()`.

Methods

Public methods:

- `ContextBatchTuning$clone()`

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`extract_inner_tuning_archives` *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

```
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` (mlr3::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 mlr3::ResampleResult and mlr3::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_freeze_archive *Freeze Archive Callback*

Description

This CallbackAsyncTuning freezes the ArchiveAsyncTuning to ArchiveAsyncTuningFrozen after the optimization has finished.

Examples

```
clbk("mlr3tuning.async_freeze_archive")
```

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"))
)
```

mlr3tuning.one_se_rule *One Standard Error Rule Callback*

Description

The one standard error rule takes the number of features into account when selecting the best hyperparameter configuration. Many learners support internal feature selection, which can be accessed via `$selected_features()`. The callback selects the hyperparameter configuration with the smallest feature set within one standard error of the best performing configuration. If there are multiple such hyperparameter configurations with the same number of features, the first one is selected.

Source

Kuhn, Max, Johnson, Kjell (2013). "Applied Predictive Modeling." In chapter Over-Fitting and Model Tuning, 61–92. Springer New York, New York, NY. ISBN 978-1-4614-6849-3.

Examples

```
clbk("mlr3tuning.one_se_rule")

# Run optimization on the pima data set with the callback
instance = tune(
  tuner = tnr("random_search", batch_size = 15),
  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 = 30,
  callbacks = clbk("mlr3tuning.one_se_rule")
)

# Hyperparameter configuration with the smallest feature set within one standard error of the best
instance$result
```

---

mlr_tuners *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_gensa`, `mlr_tuners_grid_search`, `mlr_tuners_internal`, `mlr_tuners_irace`, `mlr_tuners_nloptr`, `mlr_tuners_random_search`

Examples

```
as.data.table(mlr_tuners)
mlr_tuners$get("random_search")
tnr("random_search")
```

---

mlr_tuners_async_design_points

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()`:

`tnr("async_design_points")`

Parameters

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

Super classes

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

Methods

Public methods:

- `TunerAsyncDesignPoints$new()`
- `TunerAsyncDesignPoints$clone()`

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

Usage:

`TunerAsyncDesignPoints$new()`

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

Usage:

`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`

mlr_tuners_async_grid_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()`:

```
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:

```
TunerAsyncGridSearch$new()
```

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

Usage:

```
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`

`mlr_tuners_async_random_search`
Hyperparameter Tuning with Asynchronous 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:

`deep` Whether to make a deep clone.

Source

Bergstra J, Bengio Y (2012). "Random Search for Hyper-Parameter Optimization." *Journal of Machine Learning Research*, **13**(10), 281–305. https://jmlr.csail.mit.edu/papers/v13/bergstra12a.html.

See Also

Other TunerAsync: `mlr_tuners_async_design_points`, `mlr_tuners_async_grid_search`

mlr_tuners_cmaes *Hyperparameter Tuning with Covariance Matrix Adaptation Evolution Strategy*

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.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.

- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

Super classes

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

Methods

Public methods:

- `TunerBatchCmaes$new()`
- `TunerBatchCmaes$clone()`

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

Usage:

```
TunerBatchCmaes$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Source

Hansen N (2016). "The CMA Evolution Strategy: A Tutorial." 1604.00772.

See Also

Other Tuner: `Tuner`, `mlr_tuners`, `mlr_tuners_design_points`, `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, 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"))
```

`mlr_tuners_design_points` *Hyperparameter Tuning with Design Points*

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-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::OptimizerBatchDesignPoints which can be applied on any black box optimization problem. See also the documentation of bbotk.

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.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

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()`:

```
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.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

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` -> `TunerBatchGenSA`

Methods

Public methods:

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

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

Usage:

```
TunerBatchGenSA$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Source

Tsallis C, Stariolo DA (1996). "Generalized simulated annealing." *Physica A: Statistical Mechanics and its Applications*, **233**(1-2), 395–406. doi:10.1016/s03784371(96)002713.

Xiang Y, Gubian S, Suomela B, Hoeng J (2013). "Generalized Simulated Annealing for Global Optimization: The GenSA Package." *The R Journal*, **5**(1), 13. doi:10.32614/rj2013002.

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

```
# 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"))
```

`mlr_tuners_grid_search` *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()`:

```
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 **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")`.

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::OptimizerBatchGridSearch 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.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

Super classes

mlr3tuning::Tuner -> mlr3tuning::TunerBatch -> mlr3tuning::TunerBatchFromOptimizerBatch -> TunerBatchGridSearch

Methods

Public methods:

- TunerBatchGridSearch$new()
- TunerBatchGridSearch$clone()

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

Usage:

```
TunerBatchGridSearch$new()
```

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

Usage:

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

Arguments:

deep 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"))
```

`mlr_tuners_internal` *Hyperparameter Tuning with Internal Tuning*

Description

Subclass to conduct only internal hyperparameter tuning for a mlr3::Learner.

Dictionary

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

```
tnr("internal")
```

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.

Resources

There are several sections about hyperparameter optimization in the mlr3book.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

Super classes

mlr3tuning::Tuner -> mlr3tuning::TunerBatch -> TunerBatchInternal

Methods

Public methods:

- TunerBatchInternal$new()
- TunerBatchInternal$clone()

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

Usage:

```
TunerBatchInternal$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Note

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

See Also

Other Tuner: 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

```
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",
  eval_metric = "merror"
)

# Internal hyperparameter tuning on the pima indians diabetes data set
instance = tune(
  tnr("internal"),
  tsk("iris"),
  learner,
```

```
  rsmp("cv", folds = 3),
  msr("internal_valid_score", minimize = TRUE, select = "merror")
)

# best performing hyperparameter configuration
instance$result_learner_param_vals

instance$result_learner_param_vals$internal_tuned_values
```

`mlr_tuners_irace` *Hyperparameter Tuning with Iterated Racing.*

Description

Subclass for iterated racing. Calls `irace::irace()` from package **irace**.

Dictionary

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

```
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.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

Super classes

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

Methods

Public methods:

- `TunerBatchIrace$new()`
- `TunerBatchIrace$optimize()`
- `TunerBatchIrace$clone()`

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:

`inst` (TuningInstanceBatchSingleCrit).

Returns: data.table::data.table.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Source

Lopez-Ibanez M, Dubois-Lacoste J, Caceres LP, Birattari M, Stuetzle T (2016). "The irace package: Iterated racing for automatic algorithm configuration." *Operations Research Perspectives*, **3**, 43–58. doi:10.1016/j.orp.2016.09.002.

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

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

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

# runtime of the example is too long
```

```
# 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 = 200
)

# 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)
```

`mlr_tuners_nloptr` *Hyperparameter Tuning with Non-linear Optimization*

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 bbotk::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()`:

```
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. If set to "custom", the start values can be passed via the `start` parameter.

`start` `numeric()`
: Custom start values. Only applicable if `start_values` parameter is set to "custom".

`approximate_eval_grad_f` `logical(1)`
: Should gradients be numerically approximated via finite differences (nloptr::nl.grad). Only required for certain algorithms. Note that function evaluations required for the numerical gradient approximation will be logged as usual and are not treated differently than regular function evaluations by, e.g., Terminators.

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.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

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:

`deep` Whether to make a deep clone.

Source

Johnson, G S (2020). "The NLopt nonlinear-optimization package." https://github.com/stevengj/nlopt.

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.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

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` -> `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:

`deep` Whether to make a deep clone.

Source

Bergstra J, Bengio Y (2012). "Random Search for Hyper-Parameter Optimization." *Journal of Machine Learning Research*, **13**(10), 281–305. https://jmlr.csail.mit.edu/papers/v13/bergstra12a.html.

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`

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("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 TuningInstance-BatchMultiCrit.

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()`).

`internal_search_space` (paradox::ParamSet). Internal search space for internal tuning.

Methods

Public methods:

- `ObjectiveTuning$new()`
- `ObjectiveTuning$clone()`

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

Usage:

```
ObjectiveTuning$new(
  task,
  learner,
  resampling,
  measures,
  store_benchmark_result = TRUE,
  store_models = FALSE,
  check_values = FALSE,
  callbacks = NULL,
  internal_search_space = 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.

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

`internal_search_space` (paradox::ParamSet or `NULL`)
The internal search space.

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

Usage:

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

Arguments:

`deep` 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 TuningInstance-BatchMultiCrit.

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:

`deep` 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 TuningInstance-BatchMultiCrit.

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,
  internal_search_space = 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.

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.

internal_search_space (paradox::ParamSet or NULL)
The internal search space.

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.

---

set_validate.AutoTuner *Configure Validation for AutoTuner*

Description

Configure validation data for the learner that is tuned by the AutoTuner.

Usage

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

Arguments

learner	(AutoTuner) The autotuner for which to enable validation.
validate	(numeric(1), "predefined", "test", or NULL) How to configure the validation during the hyperparameter tuning.
...	(any) Passed when calling set_validate() on the wrapped leaerner.

Examples

```
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

```
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.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

Default Measures

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

Task	Default Measure	Package
"classif"	"classif.ce"	**mlr3**
"regr"	"regr.mse"	**mlr3**
"surv"	"surv.cindex"	**mlr3proba**

"dens"	"dens.logloss"	**mlr3proba**
"classif_st"	"classif.ce"	**mlr3spatial**
"regr_st"	"regr.mse"	**mlr3spatial**
"clust"	"clust.dunn"	**mlr3cluster**

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),
  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

```
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.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

Default Measures

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

Task	Default Measure	Package
"classif"	"classif.ce"	**mlr3**
"regr"	"regr.mse"	**mlr3**
"surv"	"surv.cindex"	**mlr3proba**
"dens"	"dens.logloss"	**mlr3proba**
"classif_st"	"classif.ce"	**mlr3spatial**
"regr_st"	"regr.mse"	**mlr3spatial**
"clust"	"clust.dunn"	**mlr3cluster**

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),
  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,
```

```
  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

Default Measures

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

Task	Default Measure	Package
"classif"	"classif.ce"	**mlr3**
"regr"	"regr.mse"	**mlr3**
"surv"	"surv.cindex"	**mlr3proba**
"dens"	"dens.logloss"	**mlr3proba**
"classif_st"	"classif.ce"	**mlr3spatial**
"regr_st"	"regr.mse"	**mlr3spatial**
"clust"	"clust.dunn"	**mlr3cluster**

Resources

There are several sections about hyperparameter optimization in the mlr3book.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.

- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

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

```
# 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)
```

`Tuner` *Tuner*

Description

The `Tuner` implements the optimization algorithm.

Details

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

Extension Packages

Additional tuners are provided by the following packages.

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

Resources

There are several sections about hyperparameter optimization in the mlr3book.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

Public fields

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

Active bindings

`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_sear`

TunerAsync *Class for Asynchronous Tuning Algorithms*

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.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

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.

`TunerBatch` *Class for Batch Tuning Algorithms*

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.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

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(),
  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 `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 = rsmp ("holdout"),
  outer_resampling = rsmp("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.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

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`

Public fields

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

Active bindings

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

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,
  extra = 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

`extra` (`data.table::data.table()`)
Additional information.

`...` (any)
ignored.

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`
Single Criterion Tuning with Rush

Description

The `TuningInstanceAsyncSingleCrit` specifies a tuning problem for a TunerAsync. The function `ti_async()` creates a TuningInstanceAsyncSingleCrit 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`.

Default Measures

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

Task	Default Measure	Package
`"classif"`	`"classif.ce"`	**mlr3**
`"regr"`	`"regr.mse"`	**mlr3**
`"surv"`	`"surv.cindex"`	**mlr3proba**
`"dens"`	`"dens.logloss"`	**mlr3proba**
`"classif_st"`	`"classif.ce"`	**mlr3spatial**
`"regr_st"`	`"regr.mse"`	**mlr3spatial**
`"clust"`	`"clust.dunn"`	**mlr3cluster**

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.

Resources

There are several sections about hyperparameter optimization in the mlr3book.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

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`

Public fields

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

Active bindings

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

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,
  extra = 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

`extra` (`data.table::data.table()`)
: Additional information.

`...` (`any`)
: ignored.

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 `ti()` creates a TuningInstanceBatchMultiCrit 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`.

Resources

There are several sections about hyperparameter optimization in the mlr3book.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.
- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

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::OptimInstanceBatchMultiCrit` -> `TuningInstanceBatchMultiCrit`

Public fields

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

Active bindings

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

Methods

Public methods:

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

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

Usage:

```
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,
  extra = 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

`extra` (`data.table::data.table()`)
Additional information.

`...` (any)
ignored.

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),
  measures = msrs(c("classif.ce", "time_train")),
  terminator = trm("evals", n_evals = 4)
)

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

# Run tuning
tuner$optimize(instance)

# Optimal hyperparameter configurations
instance$result

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

`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.

Task	Default Measure	Package
`"classif"`	`"classif.ce"`	**mlr3**
`"regr"`	`"regr.mse"`	**mlr3**
`"surv"`	`"surv.cindex"`	**mlr3proba**
`"dens"`	`"dens.logloss"`	**mlr3proba**
`"classif_st"`	`"classif.ce"`	**mlr3spatial**
`"regr_st"`	`"regr.mse"`	**mlr3spatial**
`"clust"`	`"clust.dunn"`	**mlr3cluster**

Resources

There are several sections about hyperparameter optimization in the mlr3book.

- Getting started with hyperparameter optimization.
- An overview of all tuners can be found on our website.
- Tune a support vector machine on the Sonar data set.
- Learn about tuning spaces.
- Estimate the model performance with nested resampling.
- Learn about multi-objective optimization.
- Simultaneously optimize hyperparameters and use early stopping with XGBoost.
- Automate the tuning.

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

- Learn more advanced methods with the Practical Tuning Series.

- Learn about hotstarting models.
- Run the default hyperparameter configuration of learners as a baseline.
- Use the Hyperband optimizer with different budget parameters.

The cheatsheet summarizes the most important functions of mlr3tuning.

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`

Public fields

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

Active bindings

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

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,
  extra = 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

`extra` (`data.table::data.table()`)
: Additional information.

`...` (any)
: ignored.

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),
  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:

```
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