Package ‘mlr3tuning’

September 14, 2021

Title Tuning for 'mlr3'

Version 0.9.0

Description Implements methods for hyperparameter tuning with 'mlr3', e.g. grid search, random search, generalized simulated annealing and iterated racing. Various termination criteria can be set and combined. The class 'AutoTuner' provides a convenient way to perform nested resampling in combination with 'mlr3'.

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 (>= 0.12.0), paradox (>= 0.7.0), R (>= 3.1.0)

Imports bbotk (>= 0.4.0), checkmate (>= 2.0.0), data.table, digest, lgr, mlr3misc (>= 0.9.4), R6

Suggests adagio, GenSA, irace, mlr3pipelines, nloptr, rpart, testthat (>= 3.0.0)

Config/testthat/edition 3

Config/testthat/parallel true

Encoding UTF-8

NeedsCompilation no

RoxygenNote 7.1.1

Collate 'ArchiveTuning.R' 'AutoTuner.R' 'ObjectiveTuning.R'
'mlr_tuners.R' 'Tuner.R' 'TunerCmaes.R' 'TunerDesignPoints.R'
'TunerFromOptimizer.R' 'TunerGenSA.R' 'TunerGridSearch.R'
'TunerIrace.R' 'TunerNLoptr.R' 'TunerRandomSearch.R'
'TuningInstanceMulticrit.R' 'TuningInstanceSingleCrit.R'
'assertions.R' 'auto_tuner.R' 'bibentries.R'
'extract_inner_tuning_archives.R'
'extract_inner_tuning_results.R' 'helper.R' 'reexport.R'
'sugar.R' 'tune.R' 'tune_nested.R' 'zzz.R'
mlr3tuning-package

**Description**

Implements methods for hyperparameter tuning with 'mlr3', e.g. grid search, random search, generalized simulated annealing and iterated racing. Various termination criteria can be set and combined. The class 'AutoTuner' provides a convenient way to perform nested resampling in combination with 'mlr3'.
ArchiveTuning

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

 ArchiveTuning Logging Object for Evaluated Hyperparameter Configurations

Description

Container around a \texttt{data.table::data.table()} which stores all evaluated hyperparameter configurations and performance scores.

Data structure

The table (\$data) has the following columns:

• One column for each hyperparameter of the search space (\$search\_space).
• 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 \texttt{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 \texttt{mlr3::BenchmarkResult}.

Each row corresponds to a single evaluation of a hyperparameter configuration.

The archive stores additionally a \texttt{mlr3::BenchmarkResult} (\$benchmark\_result) that records the resampling experiments. Each experiment corresponds to a single evaluation of a hyperparameter configuration. The table (\$data) and the benchmark result (\$benchmark\_result) are linked by the uhash column. If the results are viewed with \texttt{as.data.table()}, both are joined automatically.
Analysis

For analyzing the tuning results, it is recommended to pass the archive 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.
  `ArchiveTuning -> data.table::data.table()

  - x (ArchiveTuning)
  - 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 class

`bbotk::Archive` -> `ArchiveTuning`

Public fields

- `benchmark_result (mlr3::BenchmarkResult)`
  Stores benchmark result.

Methods

Public methods:

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

*Usage:*
```r
ArchiveTuning$learner(i = NULL, uhash = NULL)
```

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

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

*Usage:*
```r
ArchiveTuning$learners(i = NULL, uhash = NULL)
```

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

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

*Usage:*
```r
ArchiveTuning$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:*
```r
ArchiveTuning$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:*
```r
```
AutoTuner

```r
classname(resample_result) = ArchiveTuning
Arguments:
i (integer(1))
    The iteration value to filter for.
uhash (logical(1))
    The uhash value to filter for.
```

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

```r
classname(print) = ArchiveTuning
Usage:
archiveTuning$print()
Arguments:
... (ignored).
```

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

```r
classname(clone) = ArchiveTuning
Usage:
archiveTuning$clone(deep = FALSE)
Arguments:
deep  Whether to make a deep clone.
```

---

**AutoTuner**

---

**Description**

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$learner`. Access the tuned hyperparameters via `at$learner$param_set$values`.
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.

Note that this approach allows to perform nested resampling by passing an `AutoTuner` object 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).

**Super class**

`mlr3::Learner` -> `AutoTuner`


**Public fields**

instance_args (list())
All arguments from construction to create the TuningInstanceSingleCrit.
tuner (Tuner).

**Active bindings**

archive ArchiveTuning
Archive of the TuningInstanceSingleCrit.
learner (mlr3::Learner)
Trained learner

**Methods**

- **Public methods:**
  - AutoTuner$new()
  - AutoTuner$base_learner()
  - AutoTuner$print()
  - AutoTuner$clone()

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

*Usage:*

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

*Arguments:*
AutoTuner

learner (mlr3::Learner)
    Learner to tune, see TuningInstanceSingleCrit.
resampling (mlr3::Resampling)
    Resampling strategy during tuning, see TuningInstanceSingleCrit. This mlr3::Resampling
    is meant to be the inner resampling, operating on the training set of an arbitrary outer
    resampling. For this reason it is not feasible to pass an instantiated mlr3::Resampling here.
measure (mlr3::Measure)
    Performance measure to optimize.
terminator (bbotk::Terminator)
    When to stop tuning, see TuningInstanceSingleCrit.
tuner (Tuner)
    Tuning algorithm to run.
search_space (paradox::ParamSet)
    Hyperparameter search space. If NULL, the search space is constructed from the TuneToken
    in the ParamSet of the learner.
store_tuning_instance (logical(1))
    If TRUE (default), stores the internally created TuningInstanceSingleCrit with all intermedi-
    ate results in slot $tuning_instance.
store_benchmark_result (logical(1))
    If TRUE (default), stores the mlr3::BenchmarkResult in archive.
store_models (logical(1))
    If FALSE (default), the fitted models are not stored in the mlr3::BenchmarkResult. If store_benchmark_result
    = FALSE, the models are only stored temporarily and not accessible after the tuning. This
    combination might be useful for measures that require a model.
check_values (logical(1))
    Should parameters before the evaluation and the results be checked for validity?

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: Learner, Printer.

Method print():

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

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

Usage:
AutoTuner$clone(deep = FALSE)
Arguments:
deepl Whether to make a deep clone.
Examples

```r
task = tsk("pima")
train.set = sample(task$nrow, 0.8 * task$nrow)
test.set = setdiff(seq_len(task$nrow), train.set)

at = AutoTuner$new(
  learner = lrn("classif.rpart", cp = to_tune(1e-04, 1e-1, logscale = TRUE)),
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  terminator = trm("evals", n_evals = 5),
  tuner = tnr("random_search"))

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

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

# 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 = AutoTuner$new(
  learner = lrn("classif.rpart", cp = to_tune(1e-04, 1e-1, logscale = TRUE)),
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  terminator = trm("evals", n_evals = 5),
  tuner = tnr("random_search"))

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

Syntactic Sugar for Automatic Tuning

Description

Function to create an AutoTuner object.

Usage

auto_tuner(
    method,
    learner,
    resampling,
    measure,
    term_evals = NULL,
    term_time = NULL,
    search_space = NULL,
    ...
)

Arguments

method (character(1))
Key to retrieve tuner from mlr_tuners dictionary.

learner (mlr3::Learner).

resampling (mlr3::Resampling)
Resampling that is used to evaluated 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.

term_evals (integer(1))
Number of allowed evaluations.

term_time (integer(1))
Maximum allowed time in seconds.

search_space (paradox::ParamSet)
Hyperparameter search space. If NULL, the search space is constructed from the TuneToken in the ParamSet of the learner.

... (named list())
Named arguments to be set as parameters of the tuner.
**extract_inner_tuning_archives**

**Value**

AutoTuner

**Examples**

```r
at = auto_tuner(
  method = "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"))
```

**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 `resample()` or `benchmark()` must be called with `store_models = TRUE`.

**Usage**

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

**Arguments**

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

**Value**

`data.table::data.table()`.
Data structure

The returned data table has the following columns:

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

Examples

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

at = auto_tuner(
  method = "grid_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_tuning_archives(rr)
```
**Description**

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

**Usage**

```r
extract_inner_tuning_results(x)
```

**Arguments**

- `x`  
  
  *(mlr3::ResampleResult | mlr3::BenchmarkResult).*

**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.

- `task_id` (character(1)).

- `learner_id` (character(1)).

- `resampling_id` (character(1)).
Examples

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

at = auto_tuner(
  method = "grid_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_tuning_results(rr)
```

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() / tnr().

Usage

mlr_tuners

Format

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

Methods

See mlr3misc::Dictionary.

See Also

Sugar functions: tnr(), tnr()
Hyperparameter Tuning with Covariance Matrix Adaptation Evolution Strategy

Description

Subclass that implements CMA-ES calling \texttt{adagio::pureCMAES()} from package \texttt{adagio}.

Dictionary

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

\begin{verbatim}
TunerCmaes$new()
mlr_tuners$get("cmaes")
tnr("cmaes")
\end{verbatim}

Parameters

\begin{itemize}
\item \texttt{sigma}: numeric(1)
\item \texttt{start_values}: character(1)
\end{itemize}

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 \texttt{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

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

Logging

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

Super classes

\texttt{mlr3tuning::Tuner} \rightarrow \texttt{mlr3tuning::TunerFromOptimizer} \rightarrow \texttt{TunerCmaes}

Methods

Public methods:

\begin{itemize}
\item \texttt{TunerCmaes$new()}
\item \texttt{TunerCmaes$clone()}
\end{itemize}
Method `new()`: Creates a new instance of this R6 class.

Usage:
```
TunerCmaes$new()
```

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

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

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

Source

See Also
Package `mlr3hyperband` for hyperband tuning.

Other Tuner: `mlr_tuners_design_points, mlr_tuners_gensa, mlr_tuners_grid_search, mlr_tuners_irace, mlr_tuners_nloptr, mlr_tuners_random_search`

Examples
```
library(data.table)

# retrieve task
task = tsk("pima")

# 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))
)

# hyperparameter tuning on the pima indians diabetes data set
instance = tune(
  method = "cmaes",
  task = task,
  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(task)

mlr_tuners_design_points

Hyperparameter Tuning with via 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 via the dictionary mlr_tuners or with the associated sugar function

TunerDesignPoints$new()
mlr_tuners$get("design_points")
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.

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.

Progress Bars

$optimize() supports progress bars via the package progressr combined with a Terminator. Sim-
ply 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

\texttt{mlr3tuning::Tuner -> mlr3tuning::TunerFromOptimizer -> TunerDesignPoints}

Methods

Public methods:

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

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

Usage:

\texttt{TunerDesignPoints\$new()}

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

Usage:

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

Arguments:

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

See Also

Package \texttt{mlr3hyperband} for hyperband tuning.

Other Tuner: \texttt{mlr_tuners_cmaes, mlr_tuners_gensa, mlr_tuners_grid_search, mlr_tuners_irace, mlr_tuners_nloptr, mlr_tuners_random_search}

Examples

\begin{verbatim}
library(data.table)

# retrieve task
task = tsk("pima")

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

# hyperparameter tuning on the pima indians diabetes data set
instance = tune(
    method = "design_points",
    task = task,
    learner = learner,
    resampling = rsmp("holdout"),
    measure = msr("classif.ce"),
    design = data.table(cp = c(log(1e-1), log(1e-2)))
)

# best performing hyperparameter configuration
instance\$result
\end{verbatim}
# 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_gensa**

*Hyperparameter Tuning with Generalized Simulated Annealing*

**Description**

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

**Dictionary**

This Tuner can be instantiated via the dictionary `mlr_tuners` or with the associated sugar function `tnr()`:

```r
TunerGenSA$new()
mlr_tuners$get("gensa")
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.

**Parameters**

```r
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.
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::TunerFromOptimizer` -> TunerGenSA

Methods

Public methods:

- `TunerGenSA$new()`
- `TunerGenSA$clone()`

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

Usage:

```
TunerGenSA$new()
```

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

Usage:

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

Arguments:

- `deep` Whether to make a deep clone.

Source


See Also

Package `mlr3hyperband` for hyperband tuning.

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

Examples

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

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

# hyperparameter tuning on the pima indians diabetes data set
instance = tune(
```
mlr_tuners_grid_search

```r
method = "gensa",
task = task,
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(task)
```

---

**mlr_tuners_grid_search**

*Hyperparameter Tuning with Grid Search*

---

**Description**

Subclass for grid search tuning.

The grid is constructed as a Cartesian product over discretized values per parameter, see `paradox::generate_design_grid()`.

The points of the grid are evaluated in a random order.

**Dictionary**

This Tuner can be instantiated via the dictionary `mlr_tuners` or with the associated sugar function `tnr()`:

```r
TunerGridSearch$new()
mlr_tuners$get("grid_search")
tnr("grid_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.

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 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::TunerFromOptimizer -> TunerGridSearch

Methods

Public methods:

• TunerGridSearch$new()
• TunerGridSearch$clone()

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

Usage:
TunerGridSearch$new()

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

Usage:
TunerGridSearch$clone(deep = FALSE)

Arguments:
deepe Whether to make a deep clone.

See Also

Package mlr3hyperband for hyperband tuning.

Other Tuner: mlr_tuners_cmaes, mlr_tuners_design_points, mlr_tuners_gensa, mlr_tuners_irace, 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))

# hyperparameter tuning on the pima indians diabetes data set
instance = tune(
    method = "grid_search",
    task = task,
    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(task)

Description

TunerIrace class that implements iterated racing. Calls `irace::irace()` from package `irace`.

Dictionary

This Tuner can be instantiated via the dictionary `mlr_tuners` or with the associated sugar function `tnr()`:

```r
TunerIrace$new()
mlr_tuners$get("irace")
tnr("irace")
```

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 `TerminatorEvals` instead. Other terminators do not work with TunerIrace.
Archive

The `ArchiveTuning` 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 `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.

Super classes

`mlr3tuning::Tuner` -> `mlr3tuning::TunerFromOptimizer` -> `TunerIrace`

Methods

Public methods:

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

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

Usage:

`TunerIrace$new()`

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

Usage:

`TunerIrace$optimize(inst)`
Arguments:

inst (TuningInstanceSingleCrit).

Returns: data.table::data.table.

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

Usage:

TunerIrace$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

Source


See Also

Other Tuner: mlr_tuners_cmaes, mlr_tuners_design_points, mlr_tuners_gensa, mlr_tuners_grid_search, 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))

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

# best performing hyperparameter configuration
instance$result

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

# fit final model on complete data set
learner$param_set$values = instance$result_learner_param_vals
learner$train(task)
mlr_tuners_nloptr  Hyperparameter Tuning with Non-linear Optimization

Description

TunerNLoptr class that implements non-linear optimization. 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 via the dictionary mlr_tuners or with the associated sugar function tnr():

TunerNLoptr$new()
mlr_tuners$get("nloptr")
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.

Parameters

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

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

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

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

Super classes

$\texttt{mlr3tuning::Tuner} \rightarrow \texttt{mlr3tuning::TunerFromOptimizer} \rightarrow \texttt{TunerNLoptr}$

Methods

Public methods:

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

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

Usage:

$\texttt{TunerNLoptr$new()}$

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

Usage:

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

Arguments:

- deep Whether to make a deep clone.

Source


See Also

Package $\texttt{mlr3hyperband}$ for hyperband tuning.

Other Tuner: $\texttt{mlr_tuners_cmaes, mlr_tuners_design_points, mlr_tuners_gensa, mlr_tuners_grid_search, mlr_tuners_irace, mlr_tuners_random_search}$

Examples

```r
## Not run:
# retrieve task
task = tsk("pima")

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

# hyperparameter tuning on the pima indians diabetes data set
instance = tune(
    method = "nl0ptr",
```
mlr_tuners_random_search

Hyperparameter Tuning with Random Search

Description

Subclass for random search tuning.

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

Dictionary

This Tuner can be instantiated via the dictionary `mlr_tuners` or with the associated sugar function `tnr()`:

```
TunerRandomSearch$new()
mlr_tuners$get("random_search")
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 \texttt{lgr}) from package \texttt{bbotk}. Use \texttt{lgr::get_logger("bbotk")} to access and control the logger.

Parameters

\begin{verbatim}
batch_size integer(1)
  Maximum number of points to try in a batch.
\end{verbatim}

Progress Bars

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

Super classes

\texttt{mlr3tuning::Tuner --> mlr3tuning::TunerFromOptimizer --> TunerRandomSearch}

Methods

Public methods:

\begin{verbatim}
• TunerRandomSearch$new()
• TunerRandomSearch$clone()
\end{verbatim}

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

Usage:

\texttt{TunerRandomSearch$new()}

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Source


See Also

Package \texttt{mlr3hyperband} for hyperband tuning.

Other Tuner: \texttt{mlr_tuners_cmaes, mlr_tuners_design_points, mlr_tuners_gensa, mlr_tuners_grid_search, mlr_tuners_irace, mlr_tuners_nloptr}
Examples

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

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

# hyperparameter tuning on the pima indians diabetes data set
instance = tune(
  method = "random_search",
  task = task,
  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(task)
```

Description

Stores the objective function that estimates the performance of hyperparameter configurations. This class is usually constructed internally by the `TuningInstanceSingleCrit / TuningInstanceMultiCrit`.

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)).
- archive (ArchiveTuning).
Methods

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

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

Usage:

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

Arguments:
- `task` (`mlr3::Task`): Task to operate on.
- `learner` (`mlr3::Learner`): A Learner.
- `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. If NULL, `mlr3`’s default measure is used.
- `check_values` (logical(1)): Should parameters before the evaluation and the results be checked for validity?
- `store_benchmark_result` (logical(1)): If TRUE (default), stores the `mlr3::BenchmarkResult` in archive.
- `store_models` (logical(1)): If FALSE (default), the fitted models are not stored in the `mlr3::BenchmarkResult`. If `store_benchmark_result` = FALSE, the models are only stored temporarily and not accessible after the tuning. This combination might be useful for measures that require a model.

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

Usage:

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

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

**Syntactic Sugar for Tuner Construction**

**Description**

This function complements `mlr_tuners` with functions in the spirit of `mlr3::mlr_sugar`.

**Usage**

```r
tnr(.key, ...)
tnrs(.keys, ...)
```

**Arguments**

- `.key` (character(1))
  Key passed to the respective dictionary to retrieve the object.
- `...` (named list())
  Named arguments passed to the constructor, to be set as parameters in the `param_set` or to be set as public field. See `mlr3misc::dictionary_sugar_get()` for more details.
- `.keys` (character())
  Keys passed to the respective dictionary to retrieve multiple objects.

**Value**

- Tuner for `tnr()`
- list of Tuner for `tnrs()`

**Examples**

```r
tnr("random_search")
```

---

tune

**Function for Tuning**

**Description**

Function to tune a `mlr3::Learner`.
tune

Usage

tune(
    method,
    task,
    learner,
    resampling,
    measures,
    term_evals = NULL,
    term_time = NULL,
    search_space = NULL,
    store_models = FALSE,
    ...
)
)

Arguments

method (character(1))
    Key to retrieve tuner from mlr_tuners dictionary.

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

learner (mlr3::Learner)

resampling (mlr3::Resampling)
    Resampling that is used to evaluated 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. If NULL, mlr3’s default measure is used.

term_evals (integer(1))
    Number of allowed evaluations.

term_time (integer(1))
    Maximum allowed time in seconds.

search_space (paradox::ParamSet)
    Hyperparameter search space. If NULL, the search space is constructed from the TuneToken in the ParamSet of the learner.

store_models (logical(1))
    If FALSE (default), the fitted models are not stored in the mlr3::BenchmarkResult. If store_benchmark_result = FALSE, the models are only stored temporarily and not accessible after the tuning. This combination might be useful for measures that require a model.

... (named list())
    Named arguments to be set as parameters of the tuner.
Tuner

Description

Abstract Tuner class that implements the base functionality each tuner must provide. A tuner is an object that describes the tuning strategy, i.e. how to optimize the black-box function and its feasible set defined by the TuningInstanceSingleCrit / TuningInstanceMultiCrit object.

A tuner must write its result into the TuningInstanceSingleCrit / TuningInstanceMultiCrit using the assign_result method of the bbotk::OptimInstance at the end of its tuning in order to store the best selected hyperparameter configuration and its estimated performance vector.

Private Methods

- optimize(instance) \rightarrow \text{NULL}
  
  Abstract base method. Implement to specify tuning of your subclass. See technical details sections.

- assign_result(instance) \rightarrow \text{NULL}
  
  Abstract base method. Implement to specify how the final configuration is selected. See technical details sections.

Technical Details and Subclasses

A subclass is implemented in the following way:

- Inherit from Tuner.
- Specify the private abstract method $.tune() and use it to call into your optimizer.
- You need to call instance$eval_batch() to evaluate design points.
• The batch evaluation is requested at the `TuningInstanceSingleCrit` / `TuningInstanceMultiCrit` 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.

Active bindings

`param_set` (paradox::ParamSet).
`param_classes` (character()).
`properties` (character()).
`packages` (character()).

Methods

Public methods:

• `Tuner$new()`
• `Tuner$format()`
• `Tuner$print()`
• `Tuner$optimize()`
• `Tuner$clone()`

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

Usage:

```r
Tuner$new(param_set, param_classes, properties, packages = character())
```

Arguments:

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

`param_classes` (character())
  Supported parameter classes for learner hyperparameters that the tuner can optimize, sub-classes of paradox::Param.

`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.

Method format(): Helper for print outputs.
  Usage:
  Tuner$format()

Method print(): Print method.
  Usage:
  Tuner$print()
  Returns: (character()).

Method optimize(): Performs the tuning on a TuningInstanceSingleCrit or TuningInstanceMultiCrit until termination. The single evaluations will be written into the ArchiveTuning that resides in the TuningInstanceSingleCrit/TuningInstanceMultiCrit. The result will be written into the instance object.
  Usage:
  Tuner$optimize(inst)
  Arguments:
  inst (TuningInstanceSingleCrit | TuningInstanceMultiCrit).
  Returns: data.table::data.table

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.

Examples

instance = TuningInstanceSingleCrit$new(
  task = tsk("iris"),
  learner = lrn("classif.rpart", cp = to_tune(1e-04, 1e-1, logscale = TRUE)),
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  terminator = trm("evals", n_evals = 3)
)

tuner = tnr("random_search")

# optimize hyperparameter
# modifies the instance by reference
tuner$optimize(instance)

# returns best configuration and best performance
instance$result

# allows access of data.table of full path of all evaluations
instance$archive
Function for Nested Resampling

Description

Function to conduct nested resampling.

Usage

tune_nested(
  method,  
  task,  
  learner,  
  inner_resampling,  
  outer_resampling,  
  measure,  
  term_evals = NULL,  
  term_time = NULL,  
  search_space = NULL,  
  ...  
)

Arguments

method (character(1))
  Key to retrieve tuner from mlr_tuners dictionary.

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

learner (mlr3::Learner).

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.

term_evals (integer(1))
  Number of allowed evaluations.

term_time (integer(1))
  Maximum allowed time in seconds.

search_space (paradox::ParamSet)
  Hyperparameter search space. If NULL, the search space is constructed from the TuneToken in the ParamSet of the learner.

... (named list())
  Named arguments to be set as parameters of the tuner.
**Value**

`mlr3::ResampleResult`

**Examples**

```r
tune_nested(
    method = "random_search",
    task = tsk("pima"),
    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,
    batch_size = 2)
```

# 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()`

---

**TuningInstanceMultiCrit**

*Multi Criteria Tuning Instance*

**Description**

Specifies a general multi-criteria tuning scenario, including objective function and archive for Tuners to act upon. This class stores an `ObjectiveTuning` object that encodes the black box objective function which a Tuner has to optimize. It allows the basic operations of querying the objective at design points (`$eval_batch()`), storing the evaluations in the internal `Archive` and accessing the final result (`$result`).

Evaluations of hyperparameter configurations are performed in batches by calling `mlr3::benchmark()` internally. 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`.

**Super classes**

`bbotk::OptimInstance` -> `bbotk::OptimInstanceMultiCrit` -> `TuningInstanceMultiCrit`
Active bindings

result_learner_param_vals (list())

List of param values for the optimal learner call.

Methods

Public methods:

• TuningInstanceMultiCrit$new()
• TuningInstanceMultiCrit$assign_result()
• TuningInstanceMultiCrit$clone()

Method new(): Creates a new instance of this R6 class.
This defines the resampled performance of a learner on a task, a feasibility region for the parameters the tuner is supposed to optimize, and a termination criterion.

Usage:

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

Arguments:

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

learner (mlr3::Learner)

resampling (mlr3::Resampling)
  Resampling that is used to evaluated 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. If NULL, mlr3’s default measure is used.

terminator (Terminator)

search_space (paradox::ParamSet)
  Hyperparameter search space. If NULL, the search space is constructed from the TuneToken in the ParamSet of the learner.

store_models (logical(1))
  If FALSE (default), the fitted models are not stored in the mlr3::BenchmarkResult. If store_benchmark_result = FALSE, the models are only stored temporarily and not accessible after the tuning. This combination might be useful for measures that require a model.
check_values (logical(1))
  Should parameters before the evaluation and the results be checked for validity?
store_benchmark_result (logical(1))
  If TRUE (default), stores the mlr3::BenchmarkResult in archive.

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

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

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

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

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

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

**Examples**

```r
library(data.table)

# define search space
search_space = ps(
  cp = p_dbl(lower = 0.001, upper = 0.1),
  msplit = p_int(lower = 1, upper = 10)
)

# initialize instance
instance = TuningInstanceMultiCrit$new(
  task = tsk("iris"),
  learner = lrn("classif.rpart"),
  resampling = rsmp("holdout"),
  measure = msrs(c("classif.ce", "classif.acc")),
  search_space = search_space,
  terminator = trm("evals", n_evals = 5)
)

# generate design
design = data.table(cp = c(0.05, 0.01), msplit = c(5, 3))

# eval design
```
TuningInstanceSingleCrit

instance$eval_batch(design)

# show archive
instance$archive

---

TuningInstanceSingleCrit

Single Criterion Tuning Instance

---

Description

Specifies a general single-criteria tuning scenario, including objective function and archive for Tuners to act upon. This class stores an ObjectiveTuning object that encodes the black box objective function which a Tuner has to optimize. It allows the basic operations of querying the objective at design points ($eval_batch()), storing the evaluations in the internal Archive and accessing the final result ($result).

Evaluations of hyperparameter configurations are performed in batches by calling mlr3::benchmark() internally. 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.

Super classes

bbotk::OptimInstance -> bbotk::OptimInstanceSingleCrit -> TuningInstanceSingleCrit

Active bindings

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

Methods

Public methods:

- TuningInstanceSingleCrit$new()
- TuningInstanceSingleCrit$assign_result()
- TuningInstanceSingleCrit$clone()

Method new(): Creates a new instance of this R6 class.
This defines the resampled performance of a learner on a task, a feasibility region for the parameters the tuner is supposed to optimize, and a termination criterion.

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

Arguments:

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

- **learner** (*mlr3::Learner*)
  Resampling that is used to evaluated 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.

- **resampling** (*mlr3::Resampling*)
  Resampling that is used to evaluated 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.

- **terminator** (*Terminator*)
  Terminator.

- **search_space** (*paradox::ParamSet*)
  Hyperparameter search space. If NULL, the search space is constructed from the TuneToken in the ParamSet of the learner.

- **store_benchmark_result** (*logical(1)*)
  If TRUE (default), stores the mlr3::BenchmarkResult in archive.

- **store_models** (*logical(1)*)
  If FALSE (default), the fitted models are not stored in the mlr3::BenchmarkResult. If store_benchmark_result = FALSE, the models are only stored temporarily and not accessible after the tuning. This combination might be useful for measures that require a model.

- **check_values** (*logical(1)*)
  Should parameters before the evaluation and the results be checked for validity?

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

Usage:

TuningInstanceSingleCrit$assign_result(xdt, y, learner_param_vals = NULL)

Arguments:

- **xdt** (*data.table::data.table()*)
  x values as data.table. Each row is one point. Contains the value in the search space of the TuningInstanceMultiCrit object. Can contain additional columns for extra information.
\begin{verbatim}
library(data.table)

# define search space
search_space = ps(
  cp = p_dbl(lower = 0.001, upper = 0.1),
  minsplit = p_int(lower = 1, upper = 10)
)

# initialize instance
instance = TuningInstanceSingleCrit$new(
  task = tsk("iris"),
  learner = lrn("classif.rpart"),
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  search_space = search_space,
  terminator = trm("evals", n_evals = 5)
)

# generate design
design = data.table(cp = c(0.05, 0.01), minsplit = c(5, 3))

# eval design
instance$eval_batch(design)

# show archive
instance$archive

### error handling

# get a learner which breaks with 50% probability
# set encapsulation + fallback
learner = lrn("classif.debug", error_train = 0.5)
learner$encapsulate = c(train = "evaluate", predict = "evaluate")
learner$fallback = lrn("classif.featureless")

# define search space
search_space = ps(
  x = p_dbl(lower = 0, upper = 1)
)
\end{verbatim}
```r
instance = TuningInstanceSingleCrit$new(
  task = tsk("wine"),
  learner = learner,
  resampling = rsmp("cv", folds = 3),
  measure = msr("classif.ce"),
  search_space = search_space,
  terminator = trm("evals", n_evals = 5)
)

instance$eval_batch(data.table(x = 1:5 / 5))
```
Index

* **Tuner**
  - mlr_tuners_cmaes, 15
  - mlr_tuners_design_points, 17
  - mlr_tuners_gensa, 19
  - mlr_tuners_grid_search, 21
  - mlr_tuners_irace, 23
  - mlr_tuners_nloptr, 26
  - mlr_tuners_random_search, 28

* **datasets**
  - mlr_tuners, 14

adagio::pureCMAES(), 15
ArchiveTuning, 3, 4, 7, 24, 30, 36
auto_tuner, 10
AutoTuner, 6, 6, 10, 11, 13

bbotk::Archive, 4
bbotk::Objective, 30
bbotk::OptimInstance, 34, 38, 41
bbotk::OptimInstanceMultiCrit, 38
bbotk::OptimInstanceSingleCrit, 41
bbotk::Terminator, 6, 8, 26, 35, 38, 41

data.table::data.table, 7, 17, 25, 36
data.table::data.table(), 3, 4, 11, 13
dictionary, 13, 17, 19, 21, 23, 26, 28, 32

extract_inner_tuning_archives, 11
extract_inner_tuning_results, 13

GenSA::GenSA(), 19

irace::defaultScenario(), 23
irace::irace(), 23

Learner, 8

mlr3::benchmark(), 6, 17, 19, 21, 28, 35, 38, 41
mlr3::BenchmarkResult, 3, 4, 8, 11, 13, 31, 33, 39, 40, 42

mlr3::Learner, 5–8, 10, 30–33, 37, 39, 42
mlr3::Measure, 4, 6, 8, 10, 30, 31, 33, 37, 39, 42
mlr3::mlr_sugar, 32
mlr3::Prediction, 5
mlr3::resample(), 6
mlr3::ResampleResult, 3–5, 11–13, 38
mlr3::Resampling, 6, 8, 10, 30, 31, 33, 37, 39, 42
mlr3::Task, 30, 31, 33, 37, 39, 42
mlr3misc::Dictionary, 14
mlr3misc::dictionary_sugar_get(), 32
mlr3tuning::TunerFromOptimizer, 15, 18, 20, 22, 24, 27, 29
mlr3tuning::Tuner, 15, 18, 20, 22, 24, 27, 29
mlr3::TunerProperties, 14
mlr_tuners, 10, 14, 15, 17, 19, 21, 23, 26, 28, 32, 33, 37
mlr_tuners_cmaes, 15, 18, 20, 22, 25, 27, 29
mlr_tuners_design_points, 16, 17, 20, 22, 25, 27, 29
mlr_tuners_gensa, 16, 18, 19, 22, 25, 27, 29
mlr_tuners_grid_search, 16, 18, 20, 21, 25, 27, 29
mlr_tuners_irace, 16, 18, 20, 22, 23, 27, 29
mlr_tuners_nloptr, 16, 18, 20, 22, 23, 27, 29
mlr_tuners_random_search, 16, 18, 20, 22, 25, 27, 28

nloptr::nloptr, 26
nloptr::nloptr(), 26
nloptr::nloptr.print.options(), 26

ObjectiveTuning, 30

paradox::generate_design_grid(), 21, 22
paradox::generate_design_random(), 28
paradox::Param, 35
paradox::ParamSet, 6, 8, 10, 32, 33, 35, 37, 39, 42
R6, 7, 16, 18, 20, 22, 24, 27, 29, 31, 35, 39, 41
R6::R6Class, 14
requireNamespace(), 36
Terminator, 15, 17, 20, 22, 24, 26, 27, 29, 39, 42
TerminatorEvals, 23
tnr(), 32
tnr(), 14, 15, 17, 19, 21, 23, 26, 28
tnrs(tnr), 32
tnrs(), 14
tune(), 32
tune_nested, 37
Tuner, 6–8, 10, 14, 15, 17, 19, 21–24, 26, 28, 29, 31–33, 34, 38–42
TunerCmaes (mlr_tuners_cmaes), 15
TunerDesignPoints
    (mlr_tuners_design_points), 17
TunerGenSA (mlr_tuners_gensa), 19
TunerGridSearch
    (mlr_tuners_grid_search), 21
TunerIrace (mlr_tuners_irace), 23
TunerNloptr (mlr_tuners_nloptr), 26
TunerRandomSearch
    (mlr_tuners_random_search), 28
TuneToken, 8, 10, 33, 37, 39, 42
TuningInstanceMultiCrit, 30, 34–36, 38, 40, 42
TuningInstanceSingleCrit, 7, 8, 24, 25, 30, 34–36, 41