Package ‘mlr3tuning’

October 7, 2020

Title Tuning for 'mlr3'

Version 0.4.0

Description Implements methods for hyperparameter tuning with 'mlr3', e.g. Grid Search, Random Search, or Simulated Annealing. 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 R (>= 3.1.0)

Imports bbotk (>= 0.2.0), checkmate (>= 2.0.0), data.table, lgr, mlr3, mlr3misc (>= 0.5.0), paradox (>= 0.3.0), R6

Suggests GenSA, mlr3pipelines, nloptr, rpart, testthat

Encoding UTF-8

NeedsCompilation no

RoxygenNote 7.1.1

Collate 'ArchiveTuning.R' 'AutoTuner.R' 'ObjectiveTuning.R' 'mlr_tuners.R' 'Tuner.R' 'TunerDesignPoints.R' 'TunerFromOptimizer.R' 'TunerGenSA.R' 'TunerGridSearch.R' 'TunerNLoptr.R' 'TunerRandomSearch.R' 'TuningInstanceMulticrit.R' 'TuningInstanceSingleCrit.R' 'assertions.R' 'bibentries.R' 'helper.R' 'reexport.R' 'sugar.R' 'zzz.R'

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

Maintainer Marc Becker <marcbecker@posteo.de>

Repository CRAN
Date/Publication 2020-10-07 21:10:02 UTC

R topics documented:

`mlr3tuning-package` *mlr3tuning: Tuning for 'mlr3'*

Description

Implements methods for hyperparameter tuning with 'mlr3', e.g. Grid Search, Random Search, or Simulated Annealing. Various termination criteria can be set and combined. The class 'AutoTuner' provides a convenient way to perform nested resampling in combination with 'mlr3'.

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 objective function evaluations*

Description

Container around a data.table::data.table which stores all performed function calls of the Objective and the associated mlr3::BenchmarkResult.
$benchmark_result stores a mlr3::BenchmarkResult which contains the mlr3::ResampleResult of all performed function calls. The mlr3::BenchmarkResult is connected to the data.table::data.table via the `uhash` column.

Technical details

The data is stored in a private `.data` field that contains a data.table::data.table which logs all performed function calls of the ObjectiveTuning. This data.table::data.table is accessed with the public $data() method. New values can be added with the $add_evals() method. This however is usually done through the evaluation of the TuningInstanceSingleCrit or TuningInstanceMultiCrit by the Tuner.

Super class

`bbotk::Archive` -> `ArchiveTuning`

Public fields

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

Methods

Public methods:

- `ArchiveTuning$clone()`

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

AutoTuner *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.
3. A final model is fit on the complete training data using the now parametrized wrapped learner.

During $predict() the AutoTuner just calls the predict method of the wrapped (inner) learner.

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

`tuning_instance` (TuningInstanceSingleCrit)
    Internally created tuning instance with all intermediate results.

`tuning_result` (named `list()`)
    Short-cut to `result` from TuningInstanceSingleCrit.

`param_set` paradox::ParamSet.

Methods

Public methods:

- `AutoTuner$new()`
- `AutoTuner$clone()`

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

Usage:

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

Arguments:

`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` (list of mlr3::Measure)
: Performance measure to optimize.

`search_space` (paradox::ParamSet)
: Hyperparameter search space, see TuningInstanceSingleCrit.

`terminator` (bbotk::Terminator)
: When to stop tuning, see TuningInstanceSingleCrit.

`tuner` (Tuner)
: Tuning algorithm to run.

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

`store_benchmark_result (logical(1))`
: Store benchmark result in archive?

`store_models (logical(1))`
: Store models in benchmark result?

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

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

```
library(mlr3)
library(paradox)
task = tsk("iris")
learner = lrn("classif.rpart")
resampling = rsmp("holdout")
measure = msr("classif.ce")
search_space = ParamSet$new(
  params = list(ParamDbl$new("cp", lower = 0.001, upper = 0.1)))

terminator = trm("evals", n_evals = 5)
tuner = tnr("grid_search")
at = AutoTuner$new(
  learner, resampling, measure, search_space, terminator,
  tuner, store_tuning_instance = TRUE)

at$train(task)
at$model
at$learner

# Nested resampling
at = AutoTuner$new(learner, resampling, measure, search_space, terminator,
  tuner, store_tuning_instance = TRUE)

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

# Aggregate performance of outer results
rr$aggregate()

# Retrieve inner tuning results.
as.data.table(rr)$learner[[1]]$tuning_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()`.

Usage

```
mlr_tuners
```

Format

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

Methods

See mlr3misc::Dictionary.

See Also

Sugar functions: `tnr()`, `tnrs()`

Examples

```
mlr_tuners$get("grid_search")
tnr("random_search")
```

---

mlr_tuners_design_points

TunerDesignPoints

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 `tnr()`:

```
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.

Super classes

`mlr3tuning::Tuner` -> `mlr3tuning::TunerFromOptimizer` -> `TunerDesignPoints`

Methods

Public methods:

- `TunerDesignPoints$new()`
- `TunerDesignPoints$clone()`

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

Usage:

```
TunerDesignPoints$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
library(mlr3)
library(paradox)
library(data.table)
search_space = ParamSet$new(list(
  ParamDbl$new("cp", lower = 0.001, upper = 0.1)
))
terminator = trm("evals", n_evals = 3)
instance = TuningInstanceSingleCrit$new(
  task = tsk("iris"),
  learner = lrn("classif.rpart"),
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  search_space = search_space,
  terminator = terminator
)
```

```
design = data.table(cp = c(0.1, 0.01))
tt = tnr("design_points", design = design)
# modifies the instance by reference
tt$optimize(instance)
# returns best configuration and best performance
instance$result
# allows access of data.table of full path of all evaluations
instance$archive
```

mlr_tuners_gensa *TunerGenSA*

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()`:

```
mlr_tuners$get("gensa")
tnr("gensa")
```

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

`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.

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

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.

Examples

```
library(mlr3)
library(paradox)
search_space = ParamSet$new(list(
  ParamDbl$new("cp", lower = 0.001, upper = 0.1)
))
terminator = trm("evals", n_evals = 3)
instance = TuningInstanceSingleCrit$new(
  task = tsk("iris"),
  learner = lrn("classif.rpart"),
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  search_space = search_space,
  terminator = terminator
)
tt = tnr("gensa")

# modifies the instance by reference
tt$optimize(instance)

# returns best configuration and best performance
instance$result

# allows access of data.table of full path of all evaluations
instance$archive
```

mlr_tuners_grid_search *TunerGridSearch*

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()`:

```
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.

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:

`deep` Whether to make a deep clone.

Examples

```
library(mlr3)
library(paradox)
search_space = ParamSet$new(list(
  ParamDbl$new("cp", lower = 0.001, upper = 0.1)
))
terminator = trm("evals", n_evals = 3)
instance = TuningInstanceSingleCrit$new(
  task = tsk("iris"),
  learner = lrn("classif.rpart"),
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  search_space = search_space,
  terminator = terminator
)
tt = tnr("grid_search")

# modifies the instance by reference
tt$optimize(instance)

# returns best configuration and best performance
instance$result

# allows access of data.table of full path of all evaluations
instance$archive
```

mlr_tuners_nloptr *TuneNLoptr*

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():

```
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)

x0 numeric()

eval_g_ineq function()

xtol_rel numeric(1)

xtol_abs numeric(1)

ftol_rel numeric(1)

ftol_abs numeric(1)

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.

Super classes

mlr3tuning::Tuner -> mlr3tuning::TunerFromOptimizer -> TunerNLoptr

Methods

Public methods:

- TunerNLoptr$new()
- TunerNLoptr$clone()

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

Usage:

```
TunerNLoptr$new()
```

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

Usage:

```
TunerNLoptr$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.

Examples

```
## Not run:
library(mlr3)
library(paradox)
library(data.table)
search_space = ParamSet$new(list(
  ParamDbl$new("cp", lower = 0.001, upper = 0.1)
))
# We use the internal termination criterion xtol_rel
terminator = trm("none")
instance = TuningInstanceSingleCrit$new(
  task = tsk("iris"),
  learner = lrn("classif.rpart"),
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  search_space = search_space,
  terminator = terminator
)
tt = tnr("nloptr", x0 = 0.1, algorithm = "NLOPT_LN_BOBYQA")
# modifies the instance by reference
tt$optimize(instance)
# returns best configuration and best performance
instance$result
# allows access of data.table of full path of all evaluations
```

```
instance$archive

## End(Not run)
```

---

mlr_tuners_random_search *TunerRandomSearch*

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()`:

```
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 **lgr**) from package **bbotk**. Use `lgr::get_logger("bbotk")` to access and control the logger.

Parameters

`algorithm` `character(1)`

`x0` `numeric()`

`eval_g_ineq` function()

`xtol_rel` `numeric(1)`

`xtol_abs` `numeric(1)`

`ftol_rel` `numeric(1)`

`ftol_abs` `numeric(1)`

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`.

Super classes

`mlr3tuning::Tuner` -> `mlr3tuning::TunerFromOptimizer` -> `TunerRandomSearch`

Methods

Public methods:

- `TunerRandomSearch$new()`
- `TunerRandomSearch$clone()`

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

Usage:

```
TunerRandomSearch$new()
```

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

Usage:

```
TunerRandomSearch$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.

Examples

```
library(mlr3)
library(paradox)
search_space = ParamSet$new(list(
  ParamDbl$new("cp", lower = 0.001, upper = 0.1)
))
terminator = trm("evals", n_evals = 3)
instance = TuningInstanceSingleCrit$new(
  task = tsk("iris"),
  learner = lrn("classif.rpart"),
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  search_space = search_space,
```

```
  terminator = terminator
)
tt = tnr("random_search")

# modifies the instance by reference
tt$optimize(instance)

# returns best configuration and best performance
instance$result

# allows access of data.table of full path of all evaluations
instance$archive
```

`ObjectiveTuning` *ObjectiveTuning*

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:

```
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).

`resampling` (mlr3::Resampling)
: Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits.

`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)`)
: Store benchmark result in archive?

`store_models` (`logical(1)`)
: Store models in benchmark result?

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.

`tnr` *Syntactic Sugar for Tuner Construction*

Description

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

Usage

```
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 paradox::ParamSet, or to be set as public field. See `mlr3misc::dictionary_sugar_get()` for more details.
`.keys`	(`character()`) Keys passed to the respective dictionary to retrieve multiple objects.

Value

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

Examples

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

`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)` -> `NULL`
  Abstract base method. Implement to specify tuning of your subclass. See technical details sections.
- `.assign_result(instance)` -> `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.

Public fields

`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:

```
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, subclasses 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: NULL

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

```
library(mlr3)
library(paradox)
search_space = ParamSet$new(list(
  ParamDbl$new("cp", lower = 0.001, upper = 0.1)
))
terminator = trm("evals", n_evals = 3)
instance = TuningInstanceSingleCrit$new(
  task = tsk("iris"),
```

```
  learner = lrn("classif.rpart"),
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  search_space = search_space,
  terminator = terminator
)
# swap this line to use a different Tuner
tt = tnr("random_search")
# modifies the instance by reference
tt$optimize(instance)
# returns best configuration and best performance
instance$result
# allows access of data.table / benchmark result of full path of all
# evaluations
instance$archive
```

`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,
  search_space,
  terminator,
  store_models = FALSE,
  check_values = FALSE,
  store_benchmark_result = TRUE
)
```

Arguments:

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

`learner` (mlr3::Learner).

`resampling` (mlr3::Resampling)
  Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits.

`measures` (list of mlr3::Measure)
  Measures to optimize. If `NULL`, **mlr3**'s default measure is used.

`search_space` (paradox::ParamSet).

`terminator` (Terminator).

`store_models (logical(1))`
  Store models in benchmark result?

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

`store_benchmark_result (logical(1))`
  Store benchmark result 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.

`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,
  search_space,
  terminator,
  store_benchmark_result = TRUE,
  store_models = FALSE,
  check_values = FALSE
)
```

Arguments:

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

`learner` (mlr3::Learner).

`resampling` (mlr3::Resampling)
: Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits.

`measure` (mlr3::Measure)
: Measure to optimize.

`search_space` (paradox::ParamSet).

`terminator` (Terminator).

`store_benchmark_result` (`logical(1)`)
: Store benchmark result in archive?

`store_models` (`logical(1)`)
: Store models in benchmark result?

`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.

`y` (`numeric(1)`)
: Optimal outcome.

`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:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
library(data.table)
library(paradox)
library(mlr3)

# Objects required to define the performance evaluator:
task = tsk("iris")
learner = lrn("classif.rpart")
resampling = rsmp("holdout")
measure = msr("classif.ce")
param_set = ParamSet$new(list(
  ParamDbl$new("cp", lower = 0.001, upper = 0.1),
  ParamInt$new("minsplit", lower = 1, upper = 10))
)

terminator = trm("evals", n_evals = 5)
inst = TuningInstanceSingleCrit$new(
  task = task,
  learner = learner,
  resampling = resampling,
  measure = measure,
  search_space = param_set,
  terminator = terminator
)

# first 4 points as cross product
design = CJ(cp = c(0.05, 0.01), minsplit = c(5, 3))
inst$eval_batch(design)
inst$archive

# try more points, catch the raised terminated message
tryCatch(
  inst$eval_batch(data.table(cp = 0.01, minsplit = 7)),
  terminated_error = function(e) message(as.character(e))
)

# try another point although the budget is now exhausted
# -> no extra evaluations
tryCatch(
  inst$eval_batch(data.table(cp = 0.01, minsplit = 9)),
  terminated_error = function(e) message(as.character(e))
)

inst$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")

param_set = ParamSet$new(list(
  ParamDbl$new("x", lower = 0, upper = 1)
))

inst = TuningInstanceSingleCrit$new(
  task = tsk("wine"),
  learner = learner,
  resampling = rsmp("cv", folds = 3),
  measure = msr("classif.ce"),
  search_space = param_set,
  terminator = trm("evals", n_evals = 5)
)

tryCatch(
  inst$eval_batch(data.table(x = 1:5 / 5)),
  terminated_error = function(e) message(as.character(e))
)

archive = inst$archive$data()

# column errors: multiple errors recorded
print(archive)
```

Index