Package 'mlr3tuning'

January 31, 2020

Title Tuning for 'mlr3'

Version 0.1.2

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 checkmate (>= 1.9.4), data.table, lgr, mlr3 (>= 0.1.4), mlr3misc (>= 0.1.7), paradox, R6

Suggests GenSA, mlr3pipelines, rpart, testthat

Encoding UTF-8

NeedsCompilation no

RoxygenNote 7.0.2

Collate 'AutoTuner.R' 'mlr_terminators.R' 'Terminator.R' 'TerminatorClockTime.R' 'TerminatorCombo.R' 'TerminatorEvals.R' 'TerminatorModelTime.R' 'TerminatorNone.R' 'TerminatorPerfReached.R' 'TerminatorStagnation.R' 'mlr_tuners.R' 'Tuner.R' 'TunerDesignPoints.R' 'TunerGenSA.R' 'TunerGridSearch.R' 'TunerRandomSearch.R' 'TuningInstance.R' 'assertions.R' 'helper.R' 'sugar.R' 'zzz.R'

Author Michel Lang [cre, 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 Michel Lang <michellang@gmail.com>

Repository CRAN

Date/Publication 2020-01-31 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: Michel Lang <michellang@gmail.com> (ORCID)

Authors:

- 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

AutoTuner *AutoTuner*

Description

The AutoTuner is a mlr3::Learner which auto-tunes by first tuning the hyperparameters of its encapsulated learner on the training data, then setting the optimal configuration in the learner, then finally fitting the model on the complete training data. Note that this class allows to perform nested resampling by passing an AutoTuner object to `mlr3::resample()` or `mlr3::benchmark()`.

Format

R6::R6Class object inheriting from mlr3::Learner.

Construction

```
at = AutoTuner$new(learner, resampling, measures, tune_ps, terminator, tuner, bm_args = list())
```

- `learner` :: mlr3::Learner
  Learner to tune, see TuningInstance.
- `resampling` :: mlr3::Resampling
  Resampling strategy during tuning, see TuningInstance. 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.
- `measures` :: list of mlr3::Measure
  Performance measures. The first one is optimized, see TuningInstance.
- `tune_ps` :: paradox::ParamSet
  Hyperparameter search space, see TuningInstance.
- `terminator` :: Terminator
  When to stop tuning, see TuningInstance.
- `tuner` :: Tuner
  Tuning algorithm to run.
- `bm_args` :: `list()`
  Further arguments for `mlr3::benchmark()`, see TuningInstance.

Fields

All fields from Learner, and additionally:

- `instance_args` :: `list` All arguments from construction to create the TuningInstance.
- `tuner` :: Tuner; from construction.
- `store_tuning_instance` :: `logical(1)`
  If TRUE, stores the internally created TuningInstance with all intermediate results in slot $tuning_instance. By default, this is TRUE.
- `learner` :: mlr3::Learner Trained learner

- `tuning_instance` :: TuningInstance
  Internally created tuning instance with all intermediate results.
- `tuning_result` :: named `list`
  Short-cut to `result` from TuningInstance.

Methods

All methods from Learner, and additionally:

- `archive(unnest = "params")`
  `character(1)` -> `data.table::data.table()`
  Short-cut to method from TuningInstance.

Examples

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

terminator = term("evals", n_evals = 5)
tuner = tnr("grid_search")
at = AutoTuner$new(learner, resampling, measures, param_set, terminator, tuner)
at$store_tuning_instance = TRUE

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

`mlr_terminators` *Dictionary of Terminators*

Description

A simple mlr3misc::Dictionary storing objects of class Terminator. Each terminator has an associated help page, see `mlr_terminators_[id]`.

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

For a more convenient way to retrieve and construct terminator, see `term()`.

Format

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

Methods

See mlr3misc::Dictionary.

See Also

Sugar function: `term()`

Other Terminator: `Terminator`, `mlr_terminators_clock_time`, `mlr_terminators_combo`, `mlr_terminators_evals`, `mlr_terminators_model_time`, `mlr_terminators_none`, `mlr_terminators_perf_reached`, `mlr_terminators_stagnat`

Examples

```
term("evals", n_evals = 10)
```

---

mlr_terminators_clock_time

Terminator that stops according to the clock time

Description

Class to terminate the tuning either after the complete process took a number of seconds on the clock or a fixed time point has been reached (as reported by `Sys.time()`).

Format

R6::R6Class object inheriting from Terminator.

Construction

```
TerminatorClockTime$new()
term("clock_time")
```

Parameters

- `secs` :: `numeric(1)`
  Maximum allowed time, in seconds, default is 100. Mutually exclusive with argument `stop_time`.
- `stop_time` :: `POSIXct(1)`
  Terminator stops after this point in time. Mutually exclusive with argument `secs`.

See Also

Other Terminator: `Terminator`, `mlr_terminators_combo`, `mlr_terminators_evals`, `mlr_terminators_model_time`, `mlr_terminators_none`, `mlr_terminators_perf_reached`, `mlr_terminators_stagnation`, `mlr_terminators`

Examples

```
term("clock_time", secs = 1800)

stop_time = as.POSIXct("2030-01-01 00:00:00")
term("clock_time", stop_time = stop_time)
```

mlr_terminators_combo *Combine Terminators*

Description

This class takes multiple Terminators and terminates as soon as one or all of the included terminators are positive.

Format

R6::R6Class object inheriting from Terminator.

Construction

```
TerminatorCombo$new(terminators = list(TerminatorNone$new()))
term("combo")
```

- `terminators` :: `list()`
  List of objects of class Terminator.

Parameters

- `any` :: `logical(1)`
  Terminate iff any included terminator is positive? (not all), default is 'TRUE.

See Also

Other Terminator: `Terminator`, `mlr_terminators_clock_time`, `mlr_terminators_evals`, `mlr_terminators_model_tim`
`mlr_terminators_none`, `mlr_terminators_perf_reached`, `mlr_terminators_stagnation`, `mlr_terminators`

Examples

```
term("combo",
  list(term("model_time", secs = 60), term("evals", n_evals = 10)),
  any = FALSE
)
```

mlr_terminators_evals *Terminator that stops after a number of evaluations*

Description

Class to terminate the tuning depending on the number of evaluations. An evaluation is defined by one resampling of a parameter value.

Format

R6::R6Class object inheriting from Terminator.

Construction

```
TerminatorEvals$new()
term("evals")
```

Parameters

- `n_evals` :: `integer(1)`
  Number of allowed evaluations, default is 100L

See Also

Other Terminator: `Terminator`, `mlr_terminators_clock_time`, `mlr_terminators_combo`, `mlr_terminators_model_tim`
`mlr_terminators_none`, `mlr_terminators_perf_reached`, `mlr_terminators_stagnation`, `mlr_terminators`

Examples

```
TerminatorEvals$new()
term("evals", n_evals = 5)
```

---

mlr_terminators_model_time

Terminator that stops after a budget of model evaluation time is depleted

---

Description

Class to terminate the tuning after a given model evaluation budget is exceeded. The terminator measures the used time to train and predict all models contained in the archive.

Format

R6::R6Class object inheriting from Terminator.

Construction

```
TerminatorModelTime$new()
term("model_time")
```

Parameters

- `secs` :: `numeric(1)`
  Maximum allowed time, in seconds, default is 0.

See Also

Other Terminator: Terminator, mlr_terminators_clock_time, mlr_terminators_combo, mlr_terminators_evals, mlr_terminators_none, mlr_terminators_perf_reached, mlr_terminators_stagnation, mlr_terminators

Examples

```
TerminatorModelTime$new()
term("model_time", secs = 10 * 3600)
```

---

mlr_terminators_none *Terminator that never stops.*

Description

Mainly useful for grid search, or maybe other tuners, where the stopping is inherently controlled by the tuner itself.

Format

R6::R6Class object inheriting from Terminator.

Construction

```
t = TerminatorNone$new()
term("none")
```

See Also

Other Terminator: Terminator, mlr_terminators_clock_time, mlr_terminators_combo, mlr_terminators_evals, mlr_terminators_model_time, mlr_terminators_perf_reached, mlr_terminators_stagnation, mlr_terminators

---

mlr_terminators_perf_reached *Terminator that stops when a performance level has been reached*

Description

Class to terminate the tuning after a performance level has been hit.

Format

R6::R6Class object inheriting from Terminator.

Construction

```
TerminatorPerfReached$new()
term("perf_reached")
```

Parameters

- `level` :: `numeric(1)`
  Performance level that needs to be reached, default is 0. Terminates if the performance exceeds (respective measure has to be maximized) or falls below (respective measure has to be minimized) this value.

See Also

Other Terminator: `Terminator`, `mlr_terminators_clock_time`, `mlr_terminators_combo`, `mlr_terminators_evals`, `mlr_terminators_model_time`, `mlr_terminators_none`, `mlr_terminators_stagnation`, `mlr_terminators`

Examples

```
TerminatorPerfReached$new()
term("perf_reached")
```

---

mlr_terminators_stagnation

Terminator that stops when tuning does not improve

Description

Class to terminate the tuning after the performance stagnates, i.e. does not improve more than `threshold` over the last `iters` iterations.

Format

R6::R6Class object inheriting from Terminator.

Construction

```
t = TerminatorStagnation$new()
```

Parameters

- `iters` :: `integer(1)`
  Number of iterations to evaluate the performance improvement on, default is 10.
- `threshold` :: `numeric(1)`
  If the improvement is less than `threshold`, tuning is stopped, default is `0`.

See Also

Other Terminator: Terminator, mlr_terminators_clock_time, mlr_terminators_combo, mlr_terminators_evals, mlr_terminators_model_time, mlr_terminators_none, mlr_terminators_perf_reached, mlr_terminators

Examples

```
TerminatorStagnation$new()
term("stagnation", iters = 5, threshold = 1e-5)
```

`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()`.

Format

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

Methods

See mlr3misc::Dictionary.

See Also

Sugar function: `tnr()`

Other Tuner: Tuner, mlr_tuners_design_points, mlr_tuners_gensa, mlr_tuners_grid_search, mlr_tuners_random_search

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.

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.

Format

R6::R6Class object inheriting from Tuner.

Construction

```
TunerDesignPoints$new()
tnr("design_points")
```

Parameters

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

See Also

Other Tuner: `Tuner`, `mlr_tuners_gensa`, `mlr_tuners_grid_search`, `mlr_tuners_random_search`, `mlr_tuners`

Examples

```
# see ?Tuner
```

`mlr_tuners_gensa` *TunerGenSA*

Description

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

Format

R6::R6Class object inheriting from Tuner.

Construction

```
TunerGenSA$new()
tnr("gensa")
```

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.

See Also

Other Tuner: `Tuner`, `mlr_tuners_design_points`, `mlr_tuners_grid_search`, `mlr_tuners_random_search`, `mlr_tuners`

Examples

```
# see ?Tuner
```

`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.

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.

Format

R6::R6Class object inheriting from Tuner.

Construction

```
TunerGridSearch$new()
tnr("grid_search")
```

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 configurations to try in a batch.

See Also

Other Tuner: `Tuner`, `mlr_tuners_design_points`, `mlr_tuners_gensa`, `mlr_tuners_random_search`, `mlr_tuners`

Examples

```
# see ?Tuner
```

mlr_tuners_random_search *TunerRandomSearch*

Description

Subclass for random search tuning.

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

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.

Format

R6::R6Class object inheriting from Tuner.

Construction

```
TunerRandomSearch$new(batch_size = 1L)
tnr("random_search")
```

Parameters

- `batch_size` :: `integer(1)`
  Maximum number of configurations to try in a batch.

See Also

Other Tuner: `Tuner`, `mlr_tuners_design_points`, `mlr_tuners_gensa`, `mlr_tuners_grid_search`, `mlr_tuners`

Examples

```
# see ?Tuner
```

Terminator *Abstract Terminator Class*

Description

Abstract `Terminator` class that implements the base functionality each terminator must provide. A terminator is an object that determines when to stop the tuning.

Termination of tuning works as follows:

- Evaluations in a tuner are performed in batches.
- Before each batch evaluation, the Terminator is checked, and if it is positive, we stop.
- The tuning algorithm itself might decide not to produce any more points, or even might decide to do a smaller batch in its last evaluation.

Therefore the following note seems in order: While it is definitely possible to execute a fine-grained control for termination, and for many tuners we can specify exactly when to stop, it might happen that too few or even too many evaluations are performed, especially if multiple points are evaluated in a single batch (c.f. batch size parameter of many tuners). So it is advised to check the size of the returned archive, in particular if you are benchmarking multiple tuners.

Format

R6::R6Class object.

Construction

```
t = Terminator$new(param_set = ParamSet$new())
```

- `param_set` :: paradox::ParamSet
  Set of control parameters for terminator.

Fields

- `param_set` :: paradox::ParamSet; from construction.

Methods

- `is_terminated(instance)`
  TuningInstance -> `logical(1)`
  Is `TRUE` iff the termination criterion is positive, and `FALSE` otherwise. Must be implemented in each subclass.

See Also

Other Terminator: `mlr_terminators_clock_time`, `mlr_terminators_combo`, `mlr_terminators_evals`, `mlr_terminators_model_time`, `mlr_terminators_none`, `mlr_terminators_perf_reached`, `mlr_terminators_stagnat`
`mlr_terminators`

tnr *Syntactic Sugar for Tuner and Terminator Construction*

Description

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

Usage

```
tnr(.key, ...)

term(.key, ...)
```

Arguments

`.key`	:: `character(1)` Key passed to the respective mlr3misc::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.

Value

Tuner for `tnr()` and Terminator for `term()`.

Examples

```
term("evals", n_evals = 10)
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 TuningInstance object.

A list of measures can be passed to the instance, and they will always be all evaluated. However, single-criteria tuners optimize only the first measure.

A tuner must write its result to the `assign_result` method of the Tuninginstance at the end of its tuning in order to store the best selected hyperparameter configuration and its estimated performance vector.

Format

R6::R6Class object.

Construction

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

- `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.

Fields

- `param_set` :: paradox::ParamSet; from construction.
- `param_classes` :: `character()`
- `properties` :: ‘character(); from construction.
- `packages` :: `character()`; from construction.

Methods

- `tune(instance)`
  TuningInstance -> `self`
  Performs the tuning on a TuningInstance until termination.

Private Methods

- `tune_internal(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_internal() and use it to call into your optimizer.

- You need to call `instance$eval_batch()` to evaluate design points.
- The batch evaluation is requested at the TuningInstance object `instance`, so each batch is possibly executed in parallel via `mlr3::benchmark()`, and all evaluations are stored inside of `instance$bmr`.
- Before the batch evaluation, the 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$bmr` and return it.
- Note that therefore more points than specified by the 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 first measure, then assign its configuration and aggregated performance to the instance.

See Also

Other Tuner: `mlr_tuners_design_points`, `mlr_tuners_gensa`, `mlr_tuners_grid_search`, `mlr_tuners_random_search`
`mlr_tuners`

Examples

```
library(mlr3)
library(paradox)
param_set = ParamSet$new(list(
  ParamDbl$new("cp", lower = 0.001, upper = 0.1)
))
terminator = term("evals", n_evals = 3)
instance = TuningInstance$new(
  task = tsk("iris"),
  learner = lrn("classif.rpart"),
  resampling = rsmp("holdout"),
  measures = msr("classif.ce"),
  param_set = param_set,
  terminator = terminator
)
tt = tnr("random_search") # swap this line to use a different Tuner
tt$tune(instance) # modifies the instance by reference
instance$result # returns best configuration and best performance
instance$archive() # allows access of data.table / benchmark result of full path of all evaluations
```

TuningInstance *TuningInstance Class*

Description

Specifies a general tuning scenario, including performance evaluator and archive for Tuners to act upon. This class encodes the black box objective function, that a Tuner has to optimize. It allows the basic operations of querying the objective at design points ($eval_batch()), storing the evaluations in an internal archive and querying the archive ($archive()).

Evaluations of hyperparameter configurations are performed in batches by calling `mlr3::benchmark()` internally. Before a batch is evaluated, the 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.

A list of measures can be passed to the instance, and they will always be all evaluated. However, single-criteria tuners optimize only the first measure.

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`.

Format

R6::R6Class object.

Construction

```
inst = TuningInstance$new(task, learner, resampling, measures,
  param_set, terminator, bm_args = list())
```

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.

- `task` :: mlr3::Task.
- `learner` :: mlr3::Learner.
- `resampling` :: mlr3::Resampling
  Note that uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits.
- `measures` :: list of mlr3::Measure.
- `param_set` :: paradox::ParamSet.
- `terminator` :: Terminator.
- `bm_args` :: named `list()`
  Further arguments for `mlr3::benchmark()`.

Fields

- task :: mlr3::Task; from construction.
- learner :: mlr3::Learner; from construction.
- resampling :: mlr3::Resampling; from construction.
- measures :: list of mlr3::Measure; from construction.
- param_set :: paradox::ParamSet; from construction.
- terminator :: Terminator; from construction.
- bmr :: mlr3::BenchmarkResult
  A benchmark result, container object for all performed mlr3::ResampleResults when evaluating hyperparameter configurations.
- n_evals :: integer(1)
  Number of configuration evaluations stored in the container.
- start_time :: POSIXct(1)
  Time the tuning was started. This is set in the beginning of $tune() of Tuner.
- result :: named list()
  Result of the tuning, i.e., the optimal configuration and its estimated performance:
  - "perf": Named vector of estimated performance values of the best configuration found.
  - "tune_x": Named list of optimal hyperparameter settings, without potential trafo function applied.
  - "params": Named list of optimal hyperparameter settings, similar to tune_x, but with potential trafo function applied. Also, if the learner had some extra parameters statically set before tuning, these are included here.

Methods

- eval_batch(dt)
  data.table::data.table() -> named list()
  Evaluates all hyperparameter configurations in dt through resampling, where each configuration is a row, and columns are scalar parameters. Updates the internal BenchmarkResult $bmr by reference, and returns a named list with the following elements:
  - "batch_nr": Number of the new batch. This number is calculated in an auto-increment fashion and also stored inside the BenchmarkResult as column batch_nr
  - "uhashes": unique hashes of the added ResampleResults.
  - "perf": A data.table::data.table() of evaluated performances for each row of the dt. Has the same number of rows as dt, and the same number of columns as length of measures. Columns are named with measure-IDs. A cell entry is the (aggregated) performance of that configuration for that measure.

  Before each batch-evaluation, the Terminator is checked, and if it is positive, an exception of class terminated_error is raised. This function should be internally called by the tuner.
- tuner_objective(x)
  numeric() -> numeric(1)
  Evaluates a hyperparameter configuration (untransformed) of only numeric values, and returns a scalar objective value, where the return value is negated if the measure is maximized. Internally, $eval_batch() is called with a single row. This function serves as a objective function for tuners of numeric spaces - which should always be minimized.

- best(measure = NULL)
  (mlr3::Measure, character(1)) -> mlr3::ResampleResult
  Queries the mlr3::BenchmarkResult for the best mlr3::ResampleResult according to measure (default is the first measure in $measures). In case of ties, one of the tied values is selected randomly.
- archive(unnest = "no")
  character(1) -> data.table::data.table()
  Returns a table of contained resample results, similar to the one returned by mlr3::benchmark()'s $aggregate() method. Some interesting columns of this table are:
  - All evaluated measures are included as numeric columns, named with their measure ID.
  - tune_x: A list column that contains the parameter settings the tuner evaluated, without potential trafo applied.
  - params: A list column that contains the parameter settings that were actually used in the learner. Similar to tune_x, but with potential trafo applied. Also, if the learner had some extra parameters statically set before tuning, these are included here. unnest can have the values "no", "tune_x" or "params". If it is not set to "no", settings of the respective list-column are stored in separate columns instead of the list-column, and dependent, inactive parameters are encoded with NA.
- assign_result(tune_x,perf)
  (list, numeric) -> NULL
  The tuner writes the best found list of settings and estimated performance values here. For internal use.
  - tune_x: Must be a named list of settings only of parameters from param_set and be feasible, untransformed.
  - perf : Must be a named numeric vector of performance measures, named with performance IDs, regarding all elements in measures.

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")
measures = 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 = term("evals", n_evals = 5)
inst = TuningInstance$new(
  task = task,
  learner = learner,
  resampling = resampling,
  measures = measures,
```

```
  param_set = 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 = TuningInstance$new(
  task = tsk("wine"),
  learner = learner,
  resampling = rsmp("cv", folds = 3),
  measures = msr("classif.ce"),
  param_set = param_set,
  terminator = term("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()

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

Index