Package ‘mlr3tuning’

March 12, 2021

Title Tuning for 'mlr3'

Version 0.8.0

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 mlr3 (>= 0.7.0), paradox (>= 0.7.0), R (>= 3.1.0)
Imports bbotk (>= 0.3.0), checkmate (>= 2.0.0), data.table, lgr,
    mlr3misc (>= 0.7.0), R6

Suggests adagio, GenSA, 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'
    '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),
mlr3tuning-package

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 2021-03-12 13:40:03 UTC

R topics documented:

- mlr3tuning-package
- ArchiveTuning
- AutoTuner
- auto_tuner
- extract_inner_tuning_results
- mlr_tuners
- mlr_tuners_cmaes
- mlr_tuners_design_points
- mlr_tuners_gensa
- mlr_tuners_grid_search
- mlr_tuners_nloptr
- mlr_tuners_random_search
- ObjectiveTuning
- tnr
- tune
- Tuner
- tune_nested
- TuningInstanceMultiCrit
- TuningInstanceSingleCrit

Index

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)
ArchiveTuning

• 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.

Super class

`bbotk::Archive` -> `ArchiveTuning`

Public fields

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

Active bindings

`extended_archive` (`data.table::data.table()`)  
Joins each performed function call of the Objective with the corresponding `mlr3::ResampleResult`.

Methods

Public methods:

• `ArchiveTuning$learner()`  
• `ArchiveTuning$learners()`  
• `ArchiveTuning$learner_param_vals()`  
• `ArchiveTuning$predictions()`  
• `ArchiveTuning$resample_result()`  
• `ArchiveTuning$print()`
**Method** `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:**
`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:**
`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:**
`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:**
`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.
AutoTuner

Usage:
ArchiveTuning$resample_result(i = NULL, uhash = NULL)

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

Method print(): Printer.
Usage:
ArchiveTuning$print()

Arguments:
... (ignored).

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.

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
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.
predict_type (character(1))
Stores the currently active predict type, e.g. "response". Must be an element of $predict_types.

Methods

Public methods:
• AutoTuner$new()
• AutoTuner$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:
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 (bbopt::Terminator)
  When to stop tuning, see TuningInstanceSingleCrit.
tuner (Tuner)
  Tuning algorithm to run.
search_space (paradox::ParamSet)
  Hyperparameter search space, see TuningInstanceSingleCrit.
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))
  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 clone(): The objects of this class are cloneable with this method.
Usage:
AutoTuner$clone(deep = FALSE)
Arguments:
deeep  Whether to make a deep clone.

Examples
library(mlr3)
library(paradox)

  task = tsk("iris")
  search_space = ParamSet$new(
    params = list(ParamDbl$new("cp", lower = 0.001, upper = 0.1)))

  at = AutoTuner$new(
    learner = lrn("classif.rpart"),
    resampling = rsmp("holdout"),
    measure = msr("classif.ce"),
    terminator = trm("evals", n_evals = 5),
    tuner = tnr("grid_search"),
    search_space = search_space,
    store_tuning_instance = TRUE)

  at$train(task)
  at$model
  at$learner
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)
Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits.
extract_inner_tuning_results

**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).  
...

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

**Value**

AutoTuner

**Examples**

```r
learner = lrn("classif.rpart")
learner$param_set$values$minsplit = to_tune(1, 10)

at = auto_tuner(
  method = "random_search",
  learner = learner,
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  term_evals = 50,
  batch_size = 10)

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

---

**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. For mlr3::BenchmarkResult, the number of the experiment is added to the table.

**Usage**

```r
extract_inner_tuning_results(x)
```

**Arguments**

- **x**  
  (mlr3::ResampleResult | mlr3::BenchmarkResult)  
  Must contain an AutoTuner.
Value

\texttt{data.table::data.table()}. 

Examples

\begin{verbatim}
 task = tsk("iris")
 search_space = ParamSet$new(
   params = list(ParamDbl$new("cp", lower = 0.001, upper = 0.1))
)

 at = AutoTuner$new(
   learner = lrn("classif.rpart"),
   resampling = rsmp("holdout"),
   measure = msr("classif.ce"),
   terminator = trm("evals", n_evals = 5),
   tuner = tnr("grid_search"),
   search_space = search_space,
   store_tuning_instance = TRUE)

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

 extract_inner_tuning_results(rr)
\end{verbatim}

\textbf{mlr\_tuners} 

\textit{Dictionary of Tuners}

\textbf{Description}

A simple \texttt{mlr3misc::Dictionary} storing objects of class \texttt{Tuner}. Each tuner has an associated help page, see \texttt{mlr\_tuners[\texttt{id}]}.

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

For a more convenient way to retrieve and construct tuner, see \texttt{tnr()/tnrs()}. 

\textbf{Usage}

\texttt{mlr\_tuners}

\textbf{Format}

\texttt{R6::R6Class} object inheriting from \texttt{mlr3misc::Dictionary}.

\textbf{Methods}

See \texttt{mlr3misc::Dictionary}.

\textbf{See Also}

Sugar functions: \texttt{tnr()}, \texttt{tnrs()}

Examples

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

---

**mlr_tuners_cmaes**  
**Hyperparameter Tuning with Covariance Matrix Adaptation Evolution Strategy**

---

**Description**

Subclass that implements CMA-ES calling `adagio::pureCMAES()` from package `adagio`.

**Dictionary**

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

```r
mlr_tuners$get("cmaes")
tnr("cmaes")
```

**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**

- `sigma` 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 `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**

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

**Super classes**

```
mlr3tuning::Tuner -> mlr3tuning::TunerFromOptimizer -> TunerCmaes
```
Methods

Public methods:

- TunerCmaes$new()
- TunerCmaes$clone()

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:
dep Whether to make a deep clone.

See Also

Package mlr3hyperband for hyperband tuning.

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

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 = 10)
instance = TuningInstanceSingleCrit$new(
  task = tsk("iris"),
  learner = lrn("classif.rpart"),
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  search_space = search_space,
  terminator = terminator
)
tt = tnr("cmaes")
# modifies the instance by reference
tt$optimize(instance)
# returns best configuration and best performance
instance$result
# allows access of data.table of full path of all evaluations
instance$archive
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 tnr():

```r
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. 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 -> 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.

See Also

Package mlr3hyperband for hyperband tuning.

Other Tuner: mlr_tuners_cmaes, mlr_tuners_gensa, mlr_tuners_grid_search, mlr_tuners_nloptr, mlr_tuners_random_search

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
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
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.

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:
depth 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_nloptr, mlr_tuners_random_search

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

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
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:
deept 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_nloptr, mlr_tuners_random_search

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")
mlr_tuners_nloptr

# 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  
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():

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 con-
ditions (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
Soptimize() 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
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
nlopt.

See Also
Package mlr3hyperband for hyperband tuning.

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

```r
## 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$new(
  task = tsk("iris"),
  learner = lrn("classif.rpart"),
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  search_space = search_space,
  terminator = terminator
)
tt = tnr("nloptr", algorithm = "NLOPT_LN_BOBYQA")
# modifies the instance by reference
tt$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**

*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()`:

```r
mlr_tuners$get("random_search")
tnr("random_search")
```
Parallelization

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

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

Logging

All \texttt{Tuner} 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:

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

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

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

Source


See Also

Package mlr3hyperband for hyperband tuning.

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

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**

**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))
  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.
**tnr**

**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**

```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 `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**

```r
tnr("random_search")
```
Function for Tuning

Function to tune a `mlr3::Learner`.

Usage

tune(
  method,
  task,
  learner,
  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)
- **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.
- **term_evals** (integer(1))
  Number of allowed evaluations.
- **term_time** (integer(1))
  Maximum allowed time in seconds.
- **search_space** (paradox::ParamSet)
- **...** (named list())
  Named arguments to be set as parameters of the tuner.

Value

`TuningInstanceSingleCrit`
Examples

```r
learner = lrn("classif.rpart")
learner$param_set$values$minsplit = to_tune(1, 10)

instance = tune(
  method = "random_search",
  task = tsk("pima"),
  learner = learner,
  resampling = rsmp("holdout"),
  measure = msr("classif.ce"),
  term_evals = 50,
  batch_size = 10
)

# Apply hyperparameter values to learner
learner$param_set$values = instance$result_learner_param_vals
```

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.

### 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:**

```
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

```r
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
```
tune_nested  

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

`mlr3::ResampleResult`

Examples

```R
learner = lrn("classif.rpart")
learner$param_set$values$minsplit = to_tune(1, 10)

rr = tune_nested(
  method = "random_search",
  task = tsk("pima"),
  learner = learner,
  inner_resampling = rsmp("holdout"),
  outer_resampling = rsmp("cv", folds = 2),
  measure = msr("classif.ce"),
  term_evals = 2,
  batch_size = 2)

# check the inner results
extract_inner_tuning_results(rr)

# aggregate performance of outer results
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

\texttt{result\_learner\_param\_vals (list())}

List of param values for the optimal learner call.

Methods

Public methods:

- \texttt{TuningInstanceMultiCrit$new()}
- \texttt{TuningInstanceMultiCrit$assign\_result()}
- \texttt{TuningInstanceMultiCrit$clone()}

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

\textit{Usage:}

\texttt{TuningInstanceMultiCrit$new(}
\texttt{  task,}
\texttt{  learner,}
\texttt{  resampling,}
\texttt{  measures,}
\texttt{  terminator,}
\texttt{  search\_space = NULL,}
\texttt{  store\_models = FALSE,}
\texttt{  check\_values = FALSE,}
\texttt{  store\_benchmark\_result = TRUE}
\texttt{)}

\textit{Arguments:}

\texttt{task (mlr3::Task)}

Task to operate on.

\texttt{learner (mlr3::Learner)}

\texttt{resampling (mlr3::Resampling)}

Uninstantiated resamplings are instantiated during construction so that all configurations are evaluated on the same data splits.

\texttt{measures (list of mlr3::Measure)}

Measures to optimize. If NULL, \texttt{mlr3}’s default measure is used.

\texttt{terminator (Terminator)}

\texttt{search\_space (paradox::ParamSet)}

\texttt{store\_models (logical(1))}

If FALSE (default), the fitted models are not stored in the \texttt{mlr3::BenchmarkResult}. If \texttt{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.

\texttt{check\_values (logical(1))}

Should parameters before the evaluation and the results be checked for validity?

\texttt{store\_benchmark\_result (logical(1))}

If TRUE (default), stores the \texttt{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.

---

**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:

```r
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` (*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.
- `terminator` (*Terminator*)
- `search_space` (*paradox::ParamSet*)
- `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:

```r
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**

```r
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$eval_batch(data.table(cp = 0.01, minsplit = 7)),
  terminated_error = function(e) message(as.character(e))
)
```
tryCatch(
    inst$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 = as.data.table(inst$archive)

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

* **Tuner**  
  mlr_tuners_cmaes, 11  
  mlr_tuners_design_points, 13  
  mlr_tuners_gensa, 15  
  mlr_tuners_grid_search, 17  
  mlr_tuners_nloptr, 19  
  mlr_tuners_random_search, 21  

* **datasets**  
  mlr_tuners, 10  

  adagio::pureCMaES(), 11  
  ArchiveTuning, 3, 6, 24, 29  
  auto_tuner, 8  
  AutoTuner, 5, 5, 8, 9  

  bbotk::Archive, 3  
  bbotk::Objective, 23  
  bbotk::OptimInstance, 27, 31, 33  
  bbotk::OptimInstanceMultiCrit, 31  
  bbotk::OptimInstanceSingleCrit, 33  
  bbotk::Terminator, 5, 7, 19, 28, 31, 33  

  data.table::data.table, 3, 13  
  data.table::data.table(), 3, 9, 10  
  dictionary, 11, 13, 15, 17, 19, 21, 25  

  extract_inner_tuning_results, 9  

  GenSA::GenSA(), 15  

  mlr3::benchmark(), 5, 13, 17, 22, 27, 31, 33  
  mlr3::BenchmarkResult, 3, 7, 9, 24, 32, 34  
  mlr3::Learner, 4–6, 8, 24, 26, 30, 32, 34  
  mlr3::Measure, 5, 7, 9, 24, 26, 30, 32, 34  
  mlr3::mlr_sugar, 25  
  mlr3::Prediction, 4  
  mlr3::resample(), 5  
  mlr3::ResampleResult, 3, 4, 9, 31  
  mlr3::Resampling, 5, 6, 8, 24, 26, 30, 32, 34  
  mlr3::Task, 24, 26, 30, 32, 34  
  mlr3misc::Dictionary, 10  

  mlr3misc::dictionary_sugar_get(), 25  
  mlr3tuning (mlr3tuning-package), 2  
  mlr3tuning-package, 2  
  mlr3tuning::Tuner, 11, 14, 15, 18, 20, 22  
  mlr3tuning::TunerFromOptimizer, 11, 14, 15, 18, 20, 22  
  mlr_reflections$tuner_properties, 28  
  mlr_tuners, 8, 10, 11, 13, 15, 17, 19, 21, 25, 26, 30  

  mlr_tuners_cmaes, 11, 14, 16, 18, 20, 23  
  mlr_tuners_design_points, 12, 13, 16, 18, 20, 23  
  mlr_tuners_gensa, 12, 14, 15, 18, 20, 23  
  mlr_tuners_grid_search, 12, 14, 16, 17, 20, 23  
  mlr_tuners_nloptr, 12, 14, 16, 18, 19, 23  
  mlr_tuners_random_search, 12, 14, 16, 18, 20, 21  

  nloptr::nloptr, 19  
  nloptr::nloptr(), 19, 20  
  nloptr::nloptr.print.options(), 20  

  Objective, 3  
  ObjectiveTuning, 23  

  paradox::generate_design_grid(), 17  
  paradox::generate_design_random(), 21  
  paradox::Param, 28  
  paradox::ParamSet, 5, 7, 9, 25, 26, 28, 30, 32, 34  

  R6, 6, 12, 14, 16, 18, 20, 22, 24, 28, 32, 34  
  R6::R6Class, 10  
  requireNamespace(), 28  

  Terminator, 11, 13, 15, 18, 20, 22, 32, 34  
  tnr, 25  
  tnr(), 10, 11, 13, 15, 17, 19, 21  
  tnsr(tnr), 25  
  tnsrs(), 10
tune, 26

Tuner, 5–7, 10, 11, 13, 15, 17, 19, 21, 22, 25, 27, 31, 33, 34

Tunercmaes (mlr_tuners_cmaes), 11
TunerDesignPoints
  (mlr_tuners_design_points), 13
TunerGenSA (mlr_tuners_gensa), 15
TunerGridSearch
  (mlr_tuners_grid_search), 17
TunerNloptr (mlr_tuners_nloptr), 19
TunerRandomSearch
  (mlr_tuners_random_search), 21
TuningInstanceMultiCrit, 23, 27, 29, 31, 33, 35
TuningInstanceSingleCrit, 6, 7, 23, 27, 29, 33