Package 'bbotk'

July 24, 2020

Title Black-Box Optimization Toolkit

Version 0.2.0

Description Provides a common framework for optimization of black-box functions for other packages, e.g. 'mlr3'. It offers various optimization methods e.g. grid search, random search and generalized simulated annealing.

License LGPL-3

URL https://bbotk.mlr-org.com, https://github.com/mlr-org/bbotk

BugReports https://github.com/mlr-org/bbotk/issues

Depends R (>= 3.1.0)

Imports checkmate (>= 2.0.0), data.table, lgr, mlr3misc (>= 0.3.0), paradox (>= 0.3), R6

Suggests bibtex, GenSA, knitr, nloptr, rmarkdown, testthat

VignetteBuilder knitr

RdMacros mlr3misc

Encoding UTF-8

Language en-US

NeedsCompilation yes

RoxygenNote 7.1.1

Collate 'Archive.R' 'Objective.R' 'ObjectiveRFun.R' 'ObjectiveRFunDt.R' 'OptimInstance.R' 'OptimInstanceMultiCrit.R' 'OptimInstanceSingleCrit.R' 'mlr_optimizers.R' 'Optimizer.R' 'OptimizerDesignPoints.R' 'OptimizerGenSA.R' 'OptimizerGridSearch.R' 'OptimizerNLoptr.R' 'OptimizerRandomSearch.R' 'mlr_terminators.R' 'Terminator.R' 'TerminatorClockTime.R' 'TerminatorCombo.R' 'TerminatorEvals.R' 'TerminatorNone.R' 'TerminatorPerfReached.R' 'TerminatorRunTime.R' 'TerminatorStagnation.R' 'TerminatorStagnationBatch.R' 'assertions.R' 'bbotk_reflections.R' 'helper.R' 'sugar.R' 'zzz.R'

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

Maintainer Marc Becker <marcbecker@posteo.de>

Repository CRAN

Date/Publication 2020-07-24 17:20:02 UTC

R topics documented:

`bbotk-package` *bbotk: Black-Box Optimization Toolkit*

Description

Provides a common framework for optimization of black-box functions for other packages, e.g. 'mlr3'. It offers various optimization methods e.g. grid search, random search and generalized simulated annealing.

Author(s)

Maintainer: Marc Becker <marcbecker@posteo.de> (ORCID)

Authors:

- Jakob Richter <jakob1richter@gmail.com> (ORCID)
- Michel Lang <michellang@gmail.com> (ORCID)
- Bernd Bischl <bernd_bischl@gmx.net> (ORCID)
- Martin Binder <martin.binder@mail.com>

Other contributors:

- Olaf Mersmann <olafm@statistik.tu-dortmund.de> [contributor]

See Also

Useful links:

- https://bbotk.mlr-org.com
- https://github.com/mlr-org/bbotk
- Report bugs at https://github.com/mlr-org/bbotk/issues

`Archive` *Logging object for objective function evaluations*

Description

Container around a data.table::data.table which stores all performed function calls of the Objective.

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 Objective. 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 OptimInstance by the Optimizer.

Public fields

`search_space` (paradox::ParamSet)
: Search space of objective.

`codomain` (paradox::ParamSet)
: Codomain of objective function.

`start_time` (POSIXct).

`check_values` (`logical(1)`)

Active bindings

`n_evals` (`integer(1)`)
: Number of evaluations stored in the archive.

`n_batch` (`integer(1)`)
: Number of batches stored in the archive.

`cols_x` (`character()`). Column names of search space parameters.

`cols_y` (`character()`). Column names of codomain parameters.

Methods

Public methods:

- `Archive$new()`
- `Archive$add_evals()`
- `Archive$best()`
- `Archive$data()`
- `Archive$format()`
- `Archive$print()`
- `Archive$clear()`
- `Archive$clone()`

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

Usage:

```
Archive$new(search_space, codomain, check_values = TRUE)
```

Arguments:

`search_space` (paradox::ParamSet)
: Specifies the search space for the Optimizer. The paradox::ParamSet describes either a subset of the `domain` of the Objective or it describes a set of parameters together with a `trafo` function that transforms values from the search space to values of the domain. Depending on the context, this value defaults to the domain of the objective.

`codomain` (paradox::ParamSet)
: Specifies codomain of function. Most importantly the tags of each output "Parameter" define whether it should be minimized or maximized. The default is to minimize each component.

`check_values` (`logical(1)`)
: Should x-values that are added to the archive be checked for validity? Search space that is logged into archive.

Method `add_evals()`**:** Adds function evaluations to the archive table.

Usage:

```
Archive$add_evals(xdt, xss_trafoed, ydt)
```

Arguments:

`xdt` (`data.table::data.table()`)
Set of untransformed points / points from the *search space*. One point per row, e.g. `data.table(x1 = c(1,3),x2 = c(2,4))`. Column names have to match ids of the `search_space`. However, `xdt` can contain additional columns.

`xss_trafoed` (`list()`)
Transformed point(s) in the *domain space*.

`ydt` (`data.table::data.table()`)
Optimal outcome.

Method `best()`**:** Returns the best scoring evaluation. For single-crit optimization, the solution that minimizes / maximizes the objective function. For multi-crit optimization, the Pareto set / front.

Usage:

```
Archive$best(m = NULL)
```

Arguments:

`m` (`integer()`)
Take only batches `m` into account. Default is all batches.

Returns: `data.table::data.table()`.

Method `data()`**:** Returns a data.table::data.table which contains all performed Objective function calls.

Usage:

```
Archive$data(unnest = NULL)
```

Arguments:

`unnest` (`character()`)
Set of column names for columns to unnest via `mlr3misc::unnest()`. Unnested columns are stored in separate columns instead of list-columns.

Returns: `data.table::data.table()`.

Method `format()`**:** Helper for print outputs.

Usage:

```
Archive$format()
```

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

Usage:

```
Archive$print()
```

Arguments:

`...` (ignored).

Method `clear()`**:** Clear all evaluation results from archive.

Usage:

```
Archive$clear()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`is_dominated` *Calculate which points are dominated*

Description

Calculates which points are not dominated, i.e. points that belong to the Pareto front.

Usage

```
is_dominated(ymat)
```

Arguments

`ymat`	(`matrix()`) A numeric matrix. Each column (!) contains one point.

`mlr_optimizers` *Dictionary of Optimizer*

Description

A simple mlr3misc::Dictionary storing objects of class Optimizer. Each optimizer has an associated help page, see `mlr_optimizer_[id]`.

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

For a more convenient way to retrieve and construct optimizer, see `opt()`/`opts()`.

Format

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

Methods

See mlr3misc::Dictionary.

See Also

Sugar functions: `opt()`, `opts()`

Examples

```
opt("random_search", batch_size = 10)
```

`mlr_optimizers_design_points`
Optimization via Design Points

Description

`OptimizerDesignPoints` class that implements optimization 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.

Dictionary

This Optimizer can be instantiated via the dictionary mlr_optimizers or with the associated sugar function `opt()`:

```
mlr_optimizers$get("design_points")
opt("design_points")
```

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 class

`bbotk::Optimizer` -> `OptimizerDesignPoints`

Methods

Public methods:

- `OptimizerDesignPoints$new()`
- `OptimizerDesignPoints$clone()`

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

Usage:

```
OptimizerDesignPoints$new()
```

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

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

domain = ParamSet$new(list(ParamDbl$new("x", lower = -1, upper = 1)))

search_space = ParamSet$new(list(ParamDbl$new("x", lower = -1, upper = 1)))

codomain = ParamSet$new(list(ParamDbl$new("y", tags = "minimize")))

objective_function = function(xs) {
  list(y = as.numeric(xs)^2)
}

objective = ObjectiveRFun$new(fun = objective_function,
  domain = domain,
  codomain = codomain)
terminator = trm("evals", n_evals = 10)
instance = OptimInstanceSingleCrit$new(objective = objective,
  search_space = search_space,
  terminator = terminator)

design = data.table(x = c(0, 1))

optimizer = opt("design_points", design = design)

# Modifies the instance by reference
optimizer$optimize(instance)

# Returns best scoring evaluation
instance$result

# Allows access of data.table of full path of all evaluations
instance$archive$data()
```

`mlr_optimizers_gensa` *Optimization via Generalized Simulated Annealing*

Description

OptimizerGenSA class that implements generalized simulated annealing. Calls GenSA::GenSA() from package **GenSA**.

Dictionary

This Optimizer can be instantiated via the dictionary mlr_optimizers or with the associated sugar function opt():

```
mlr_optimizers$get("gensa")
opt("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.

Super class

bbotk::Optimizer -> OptimizerGenSA

Methods

Public methods:

- OptimizerGenSA$new()
- OptimizerGenSA$clone()

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

Usage:

```
OptimizerGenSA$new()
```

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

Usage:

```
OptimizerGenSA$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(paradox)

domain = ParamSet$new(list(ParamDbl$new("x", lower = -1, upper = 1)))

search_space = ParamSet$new(list(ParamDbl$new("x", lower = -1, upper = 1)))

codomain = ParamSet$new(list(ParamDbl$new("y", tags = "minimize")))

objective_function = function(xs) {
  list(y = as.numeric(xs)^2)
}

objective = ObjectiveRFun$new(fun = objective_function,
                              domain = domain,
                              codomain = codomain)
terminator = trm("evals", n_evals = 10)
instance = OptimInstanceSingleCrit$new(objective = objective,
                             search_space = search_space,
                             terminator = terminator)

optimizer = opt("gensa")

# Modifies the instance by reference
optimizer$optimize(instance)

# Returns best scoring evaluation
instance$result

# Allows access of data.table of full path of all evaluations
instance$archive$data()
```

`mlr_optimizers_grid_search`

Optimization via Grid Search

Description

`OptimizerGridSearch` class that implements grid search. 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.

Dictionary

This Optimizer can be instantiated via the dictionary mlr_optimizers or with the associated sugar function `opt()`:

```
mlr_optimizers$get("grid_search")
opt("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 points to try in a batch.

Super class

`bbotk::Optimizer` -> `OptimizerGridSearch`

Methods

Public methods:

- `OptimizerGridSearch$new()`
- `OptimizerGridSearch$clone()`

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

Usage:

```
OptimizerGridSearch$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
library(paradox)

domain = ParamSet$new(list(ParamDbl$new("x", lower = -1, upper = 1)))

search_space = ParamSet$new(list(ParamDbl$new("x", lower = -1, upper = 1)))
```

```
codomain = ParamSet$new(list(ParamDbl$new("y", tags = "minimize")))

objective_function = function(xs) {
  list(y = as.numeric(xs)^2)
}

objective = ObjectiveRFun$new(fun = objective_function,
                              domain = domain,
                              codomain = codomain)
terminator = trm("evals", n_evals = 10)
instance = OptimInstanceSingleCrit$new(objective = objective,
                             search_space = search_space,
                             terminator = terminator)

optimizer = opt("grid_search")

# Modifies the instance by reference
optimizer$optimize(instance)

# Returns best scoring evaluation
instance$result

# Allows access of data.table of full path of all evaluations
instance$archive$data()
```

`mlr_optimizers_random_search`

Optimization via Random Search

Description

`OptimizerRandomSearch` class that implements a simple Random Search.

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.

Dictionary

This Optimizer can be instantiated via the dictionary mlr_optimizers or with the associated sugar function `opt()`:

```
mlr_optimizers$get("random_search")
opt("random_search")
```

Parameters

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

Super class

`bbotk::Optimizer` -> `OptimizerRandomSearch`

Methods

Public methods:

- `OptimizerRandomSearch$new()`
- `OptimizerRandomSearch$clone()`

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

Usage:

```
OptimizerRandomSearch$new()
```

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

Usage:

```
OptimizerRandomSearch$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. http://jmlr.org/papers/v13/bergstra12a.html.

Examples

```
library(paradox)

domain = ParamSet$new(list(ParamDbl$new("x", lower = -1, upper = 1)))

search_space = ParamSet$new(list(ParamDbl$new("x", lower = -1, upper = 1)))

codomain = ParamSet$new(list(ParamDbl$new("y", tags = "minimize")))

objective_function = function(xs) {
  list(y = as.numeric(xs)^2)
}

objective = ObjectiveRFun$new(fun = objective_function,
                              domain = domain,
                              codomain = codomain)
terminator = trm("evals", n_evals = 10)
instance = OptimInstanceSingleCrit$new(objective = objective,
                             search_space = search_space,
                             terminator = terminator)

optimizer = opt("random_search")
```

```
# Modifies the instance by reference
optimizer$optimize(instance)

# Returns best scoring evaluation
instance$result

# Allows access of data.table of full path of all evaluations
instance$archive$data()
```

`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 `trm()`/`trms()`.

Format

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

Methods

See mlr3misc::Dictionary.

See Also

Sugar functions: `trm()`, `trms()`

Other Terminator: `Terminator`, `mlr_terminators_clock_time`, `mlr_terminators_combo`, `mlr_terminators_evals`, `mlr_terminators_none`, `mlr_terminators_perf_reached`, `mlr_terminators_run_time`, `mlr_terminators_stagnatio`
`mlr_terminators_stagnation`

Examples

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

`mlr_terminators_clock_time` *Terminator that stops according to the clock time*

Description

Class to terminate the optimization after a fixed time point has been reached (as reported by `Sys.time()`).

Dictionary

This Terminator can be instantiated via the dictionary mlr_terminators or with the associated sugar function `trm()`:

```
mlr_terminators$get("clock_time")
trm("clock_time")
```

Parameters

`stop_time POSIXct(1)`
: Terminator stops after this point in time.

Super class

`bbotk::Terminator` -> `TerminatorClockTime`

Methods

Public methods:

- `TerminatorClockTime$new()`
- `TerminatorClockTime$is_terminated()`
- `TerminatorClockTime$clone()`

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

Usage:

```
TerminatorClockTime$new()
```

Method `is_terminated()`**:** Is `TRUE` iff the termination criterion is positive, and `FALSE` otherwise.

Usage:

```
TerminatorClockTime$is_terminated(archive)
```

Arguments:

`archive` (Archive).

Returns: `logical(1)`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Terminator: `Terminator`, `mlr_terminators_combo`, `mlr_terminators_evals`, `mlr_terminators_none`, `mlr_terminators_perf_reached`, `mlr_terminators_run_time`, `mlr_terminators_stagnation_batch`, `mlr_terminators_stagnation`, `mlr_terminators`

Examples

```
stop_time = as.POSIXct("2030-01-01 00:00:00")
trm("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.

Dictionary

This Terminator can be instantiated via the dictionary mlr_terminators or with the associated sugar function `trm()`:

```
mlr_terminators$get("combo")
trm("combo")
```

Parameters

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

Super class

`bbotk::Terminator` -> `TerminatorCombo`

Public fields

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

Methods

Public methods:

- `TerminatorCombo$new()`
- `TerminatorCombo$is_terminated()`
- `TerminatorCombo$print()`
- `TerminatorCombo$clone()`

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

Usage:

```
TerminatorCombo$new(terminators = list(TerminatorNone$new()))
```

Arguments:

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

Method `is_terminated()`**:** Is `TRUE` iff the termination criterion is positive, and `FALSE` otherwise.

Usage:

```
TerminatorCombo$is_terminated(archive)
```

Arguments:

`archive` (Archive).

Returns: `logical(1)`.

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

Usage:

```
TerminatorCombo$print(...)
```

Arguments:

`...` (ignored).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Terminator: `Terminator`, `mlr_terminators_clock_time`, `mlr_terminators_evals`, `mlr_terminators_none`, `mlr_terminators_perf_reached`, `mlr_terminators_run_time`, `mlr_terminators_stagnation_batch`, `mlr_terminators_stagnation`, `mlr_terminators`

Examples

```
trm("combo",
  list(trm("clock_time", stop_time = Sys.time() + 60),
    trm("evals", n_evals = 10)), any = FALSE
)
```

`mlr_terminators_evals` *Terminator that stops after a number of evaluations*

Description

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

Dictionary

This Terminator can be instantiated via the dictionary mlr_terminators or with the associated sugar function `trm()`:

```
mlr_terminators$get("evals")
trm("evals")
```

Parameters

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

Super class

`bbotk::Terminator` -> `TerminatorEvals`

Methods

Public methods:

- `TerminatorEvals$new()`
- `TerminatorEvals$is_terminated()`
- `TerminatorEvals$clone()`

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

Usage:

```
TerminatorEvals$new()
```

Method `is_terminated()`**:** Is `TRUE` iff the termination criterion is positive, and `FALSE` otherwise.

Usage:

```
TerminatorEvals$is_terminated(archive)
```

Arguments:

`archive` (Archive).

Returns: `logical(1)`.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Other Terminator: Terminator, mlr_terminators_clock_time, mlr_terminators_combo, mlr_terminators_none, mlr_terminators_perf_reached, mlr_terminators_run_time, mlr_terminators_stagnation_batch, mlr_terminators_stagnation, mlr_terminators

Examples

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

---

mlr_terminators_none *Terminator that never stops.*

Description

Mainly useful for optimization algorithms where the stopping is inherently controlled by the algorithm itself (e.g. OptimizerGridSearch).

Dictionary

This Terminator can be instantiated via the dictionary mlr_terminators or with the associated sugar function `trm()`:

```
mlr_terminators$get("none")
trm("none")
```

Super class

`bbotk::Terminator` -> `TerminatorNone`

Methods

Public methods:

- `TerminatorNone$new()`
- `TerminatorNone$is_terminated()`
- `TerminatorNone$clone()`

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

Usage:

```
TerminatorNone$new()
```

Method `is_terminated()`**:** Is `TRUE` iff the termination criterion is positive, and `FALSE` otherwise.

Usage:

```
TerminatorNone$is_terminated(archive)
```

Arguments:

`archive` (Archive).

Returns: `logical(1)`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Terminator: Terminator, mlr_terminators_clock_time, mlr_terminators_combo, mlr_terminators_evals, mlr_terminators_perf_reached, mlr_terminators_run_time, mlr_terminators_stagnation_batch, mlr_terminators_stagnation, mlr_terminators

---

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

Description

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

Dictionary

This Terminator can be instantiated via the dictionary mlr_terminators or with the associated sugar function `trm()`:

```
mlr_terminators$get("perf_reached")
trm("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.

Super class

`bbotk::Terminator` -> `TerminatorPerfReached`

Methods

Public methods:

- `TerminatorPerfReached$new()`
- `TerminatorPerfReached$is_terminated()`
- `TerminatorPerfReached$clone()`

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

Usage:

```
TerminatorPerfReached$new()
```

Method `is_terminated()`**:** Is `TRUE` iff the termination criterion is positive, and `FALSE` otherwise.

Usage:

```
TerminatorPerfReached$is_terminated(archive)
```

Arguments:

`archive` (Archive).

Returns: `logical(1)`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Terminator: `Terminator`, `mlr_terminators_clock_time`, `mlr_terminators_combo`, `mlr_terminators_evals`, `mlr_terminators_none`, `mlr_terminators_run_time`, `mlr_terminators_stagnation_batch`, `mlr_terminators_stagnation`, `mlr_terminators`

Examples

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

mlr_terminators_run_time

Terminator that stops according to the run time

Description

Class to terminate the optimization after the optimization process took a number of seconds on the clock.

Dictionary

This Terminator can be instantiated via the dictionary mlr_terminators or with the associated sugar function `trm()`:

```
mlr_terminators$get("run_time")
trm("run_time")
```

Parameters

`secs numeric(1)`
: Maximum allowed time, in seconds, default is 100.

Super class

`bbotk::Terminator` -> `TerminatorRunTime`

Methods

Public methods:

- `TerminatorRunTime$new()`
- `TerminatorRunTime$is_terminated()`
- `TerminatorRunTime$clone()`

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

Usage:

```
TerminatorRunTime$new()
```

Method `is_terminated()`**:** Is `TRUE` iff the termination criterion is positive, and `FALSE` otherwise.

Usage:

```
TerminatorRunTime$is_terminated(archive)
```

Arguments:

`archive` (Archive).

Returns: `logical(1)`.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

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_batch, mlr_terminators_stagnation, mlr_terminators

Examples

```
trm("run_time", secs = 1800)
```

---

mlr_terminators_stagnation

Terminator that stops when optimization does not improve

Description

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

Dictionary

This Terminator can be instantiated via the dictionary mlr_terminators or with the associated sugar function `trm()`:

```
mlr_terminators$get("stagnation")
trm("stagnation")
```

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`, optimization is stopped, default is `0`.

Super class

`bbotk::Terminator` -> `TerminatorStagnation`

Methods

Public methods:

- `TerminatorStagnation$new()`
- `TerminatorStagnation$is_terminated()`
- `TerminatorStagnation$clone()`

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

Usage:

```
TerminatorStagnation$new()
```

Method `is_terminated()`**:** Is `TRUE` iff the termination criterion is positive, and `FALSE` otherwise.

Usage:

```
TerminatorStagnation$is_terminated(archive)
```

Arguments:

`archive` (Archive).

Returns: `logical(1)`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Terminator: `Terminator`, `mlr_terminators_clock_time`, `mlr_terminators_combo`, `mlr_terminators_evals`, `mlr_terminators_none`, `mlr_terminators_perf_reached`, `mlr_terminators_run_time`, `mlr_terminators_stagnatio`
`mlr_terminators`

Examples

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

`mlr_terminators_stagnation_batch` *Terminator that stops when optimization does not improve*

Description

Class to terminate the optimization after the performance stagnates, i.e. does not improve more than `threshold` over the last `n` batches.

Dictionary

This Terminator can be instantiated via the dictionary mlr_terminators or with the associated sugar function `trm()`:

```
mlr_terminators$get("stagnation")
trm("stagnation")
```

Parameters

`n integer(1)`
: Number of batches to evaluate the performance improvement on, default is 1.

`threshold numeric(1)`
: If the improvement is less than `threshold`, optimization is stopped, default is `0`.

Super class

`bbotk::Terminator` -> `TerminatorStagnationBatch`

Methods

Public methods:

- `TerminatorStagnationBatch$new()`
- `TerminatorStagnationBatch$is_terminated()`
- `TerminatorStagnationBatch$clone()`

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

Usage:

```
TerminatorStagnationBatch$new()
```

Method `is_terminated()`**:** Is `TRUE` iff the termination criterion is positive, and `FALSE` otherwise.

Usage:

```
TerminatorStagnationBatch$is_terminated(archive)
```

Arguments:

`archive` (Archive).

Returns: `logical(1)`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Terminator: `Terminator`, `mlr_terminators_clock_time`, `mlr_terminators_combo`, `mlr_terminators_evals`, `mlr_terminators_none`, `mlr_terminators_perf_reached`, `mlr_terminators_run_time`, `mlr_terminators_stagnatio` `mlr_terminators`

Examples

```
TerminatorStagnationBatch$new()
trm("stagnation_batch", n = 1, threshold = 1e-5)
```

`Objective` *Objective function with domain and co-domain*

Description

Describes a black-box objective function that maps an arbitrary domain to a numerical codomain.

Technical details

`Objective` objects can have the following properties: `"noisy"`, `"deterministic"`, `"single-crit"` and `"multi-crit"`.

Public fields

`id (character(1))`.

`properties (character())`.

`domain` (paradox::ParamSet)
Specifies domain of function, hence its input parameters, their types and ranges.

`codomain` (paradox::ParamSet)
Specifies codomain of function, hence its feasible values.

`check_values (logical(1))`

Active bindings

`xdim (integer(1))`
Dimension of domain.

`ydim (integer(1))`
Dimension of codomain.

Methods

Public methods:

- Objective$new()
- Objective$format()
- Objective$print()
- Objective$eval()
- Objective$eval_many()
- Objective$eval_dt()
- Objective$clone()

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

Usage:

```
Objective$new(
  id = "f",
  properties = character(),
  domain,
  codomain = ParamSet$new(list(ParamDbl$new("y", tags = "minimize"))),
  check_values = TRUE
)
```

Arguments:

`id` (`character(1)`).

`properties` (`character()`).

`domain` (paradox::ParamSet)
: Specifies domain of function. The paradox::ParamSet should describe all possible input parameters of the objective function. This includes their `id`, their types and the possible range.

`codomain` (paradox::ParamSet)
: Specifies codomain of function. Most importantly the tags of each output "Parameter" define whether it should be minimized or maximized. The default is to minimize each component.

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

Method `format()`**:** Helper for print outputs.

Usage:

```
Objective$format()
```

Returns: `character()`.

Method `print()`**:** Print method.

Usage:

```
Objective$print()
```

Returns: `character()`.

Method `eval()`**:** Evaluates a single input value on the objective function. If `check_values = TRUE`, the validity of the point as well as the validity of the result is checked.

Usage:

```
Objective$eval(xs)
```

Arguments:

`xs (list())`
A list that contains a single x value, e.g. `list(x1 = 1, x2 = 2)`.

Returns: `list()` that contains the result of the evaluation, e.g. `list(y = 1)`. The list can also contain additional *named* entries that will be stored in the archive if called through the OptimInstance. These extra entries are referred to as *extras*.

Method `eval_many()`**:** Evaluates multiple input values on the objective function. If `check_values = TRUE`, the validity of the points as well as the validity of the results are checked. *bbotk* does not take care of parallelization. If the function should make use of parallel computing, it has to be implemented by deriving from this class and overwriting this function.

Usage:

```
Objective$eval_many(xss)
```

Arguments:

`xss (list())`
A list of lists that contains multiple x values, e.g. `list(list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4))`.

Returns: data.table::data.table()] that contains one y-column for single-criteria functions and multiple y-columns for multi-criteria functions, e.g. `data.table(y = 1:2)` or `data.table(y1 = 1:2, y2 = 3:4)`. It may also contain additional columns that will be stored in the archive if called through the OptimInstance. These extra columns are referred to as *extras*.

Method `eval_dt()`**:** Evaluates multiple input values on the objective function

Usage:

```
Objective$eval_dt(xdt)
```

Arguments:

`xdt (data.table::data.table())`
Set of untransformed points / points from the *search space*. One point per row, e.g. `data.table(x1 = c(1, 3), x2 = c(2, 4))`. Column names have to match ids of the `search_space`. However, `xdt` can contain additional columns.

Returns: data.table::data.table()] that contains one y-column for single-criteria functions and multiple y-columns for multi-criteria functions, e.g. `data.table(y = 1:2)` or `data.table(y1 = 1:2, y2 = 3:4)`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`ObjectiveRFun` *Objective interface with custom R function*

Description

Objective interface where the user can pass a custom R function that expects a list as input.

Super class

`bbotk::Objective` -> `ObjectiveRFun`

Active bindings

`fun (function)`
Objective function.

Methods

Public methods:

- `ObjectiveRFun$new()`
- `ObjectiveRFun$eval()`
- `ObjectiveRFun$clone()`

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

Usage:

```
ObjectiveRFun$new(
  fun,
  domain,
  codomain = NULL,
  id = "function",
  properties = character()
)
```

Arguments:

`fun (function)`
R function that encodes objective and expects a list with the input for a single point (e.g. `list(x1 = 1, x2 = 2)`) and returns the result either as a numeric vector or a list (e.g. `list(y = 3)`).

`domain (paradox::ParamSet)`
Specifies domain of function. The paradox::ParamSet should describe all possible input parameters of the objective function. This includes their `id`, their types and the possible range.

`codomain (paradox::ParamSet)`
Specifies codomain of function. Most importantly the tags of each output "Parameter" define whether it should be minimized or maximized. The default is to minimize each component.

`id` (`character(1)`).

`properties` (`character()`).

Method `eval()`**:** Evaluates input value(s) on the objective function. Calls the R function supplied by the user.

Usage:

```
ObjectiveRFun$eval(xs)
```

Arguments:

`xs` Input values.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`ObjectiveRFunDt` *Objective interface for basic R functions.*

Description

Objective interface where user can pass an R function that works on an `data.table()`.

Super class

`bbotk::Objective` -> `ObjectiveRFunDt`

Active bindings

`fun` (`function`)
Objective function.

Methods

Public methods:

- `ObjectiveRFunDt$new()`
- `ObjectiveRFunDt$eval_many()`
- `ObjectiveRFunDt$eval_dt()`
- `ObjectiveRFunDt$clone()`

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

Usage:

```
ObjectiveRFunDt$new(
  fun,
  domain,
  codomain = NULL,
  id = "function",
  properties = character()
)
```

Arguments:

`fun (function)`
: R function that encodes objective and expects an `data.table()` as input whereas each point is represented by one row.

`domain (paradox::ParamSet)`
: Specifies domain of function. The paradox::ParamSet should describe all possible input parameters of the objective function. This includes their `id`, their types and the possible range.

`codomain (paradox::ParamSet)`
: Specifies codomain of function. Most importantly the tags of each output "Parameter" define whether it should be minimized or maximized. The default is to minimize each component.

`id (character(1))`.

`properties (character())`.

Method `eval_many()`**:** Evaluates multiple input values received as a list, converted to a `data.table()` on the objective function.

Usage:

```
ObjectiveRFunDt$eval_many(xss)
```

Arguments:

`xss (list())`
: A list of lists that contains multiple x values, e.g. `list(list(x1 = 1,x2 = 2),list(x1 = 3,x2 = 4))`.

Returns: `data.table::data.table()` that contains one y-column for single-criteria functions and multiple y-columns for multi-criteria functions, e.g. `data.table(y = 1:2)` or `data.table(y1 = 1:2,y2 = 3:4)`.

Method `eval_dt()`**:** Evaluates multiple input values on the objective function supplied by the user.

Usage:

```
ObjectiveRFunDt$eval_dt(xdt)
```

Arguments:

`xdt (data.table::data.table())`
: Set of untransformed points / points from the *search space*. One point per row, e.g. `data.table(x1 = c(1,3),x2 = c(2,4))`. Column names have to match ids of the `search_space`. However, `xdt` can contain additional columns.

Returns: data.table::data.table()] that contains one y-column for single-criteria functions and multiple y-columns for multi-criteria functions, e.g. `data.table(y = 1:2)` or `data.table(y1 = 1:2,y2 = 3:4)`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`opt` *Syntactic Sugar Optimizer Construction*

Description

This function complements mlr_optimizers with functions in the spirit of `mlr_sugar` from **mlr3**.

Usage

```
opt(.key, ...)

opts(.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

- Optimizer for `opt()`.
- list of Optimizer for `opts()`.

Examples

```
opt("random_search", batch_size = 10)
```

`OptimInstance` *Optimization Instance with budget and archive*

Description

Abstract base class.

Technical details

The Optimizer writes the final result to the `.result` field by using the $assign_result() method. `.result` stores a data.table::data.table consisting of x values in the *search space*, (transformed) x values in the *domain space* and y values in the *codomain space* of the Objective. The user can access the results with active bindings (see below).

Public fields

`objective` (Objective).

`search_space` (paradox::ParamSet).

`terminator` (Terminator).

`is_terminated` (`logical(1)`).

`archive` (Archive).

Active bindings

`result` (data.table::data.table)
Get result

`result_x_search_space` (data.table::data.table)
x part of the result in the *search space*.

`result_x_domain` (`list()`)
(transformed) x part of the result in the *domain space* of the objective.

`result_y` (`numeric()`)
Optimal outcome.

Methods

Public methods:

- `OptimInstance$new()`
- `OptimInstance$format()`
- `OptimInstance$print()`
- `OptimInstance$eval_batch()`
- `OptimInstance$assign_result()`
- `OptimInstance$objective_function()`
- `OptimInstance$clone()`

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

Usage:

```
OptimInstance$new(objective, search_space = NULL, terminator)
```

Arguments:

`objective` (Objective).

`search_space` (paradox::ParamSet)
: Specifies the search space for the Optimizer. The paradox::ParamSet describes either a subset of the `domain` of the Objective or it describes a set of parameters together with a `trafo` function that transforms values from the search space to values of the domain. Depending on the context, this value defaults to the domain of the objective.

`terminator` (Terminator).

Method `format()`**:** Helper for print outputs.

Usage:

```
OptimInstance$format()
```

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

Usage:

```
OptimInstance$print(...)
```

Arguments:

`...` (ignored).

Method `eval_batch()`**:** Evaluates all input values in `xdt` by calling the Objective. Applies possible transformations to the input values and writes the results to the Archive.

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 Optimizer.

Usage:

```
OptimInstance$eval_batch(xdt)
```

Arguments:

`xdt` (`data.table::data.table()`)
: x values as `data.table()` with one point per row. Contains the value in the *search space* of the OptimInstance object. Can contain additional columns for extra information.

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

Usage:

```
OptimInstance$assign_result(xdt, y)
```

Arguments:

`xdt` (`data.table::data.table()`)
: x values as `data.table()` with one row. Contains the value in the *search space* of the OptimInstance object. Can contain additional columns for extra information.

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

Method `objective_function()`: Evaluates (untransformed) points of only numeric values. Returns a numeric scalar for single-crit or a numeric vector for multi-crit. The return value(s) are negated if the measure is maximized. Internally, $eval_batch() is called with a single row. This function serves as a objective function for optimizers of numeric spaces - which should always be minimized.

Usage:

```
OptimInstance$objective_function(x)
```

Arguments:

`x (numeric())`
Untransformed points.

Returns: Objective value as `numeric(1)`, negated for maximization problems.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`OptimInstanceMultiCrit`

Optimization Instance with budget and archive

Description

Wraps a multi-criteria Objective function with extra services for convenient evaluation. Inherits from OptimInstance.

- Automatic storing of results in an Archive after evaluation.
- Automatic checking for termination. Evaluations of design points are performed in batches. 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.

Super class

`bbotk::OptimInstance` -> `OptimInstanceMultiCrit`

Active bindings

`result_x_domain (list())`
(transformed) x part of the result in the *domain space* of the objective.

`result_y (numeric(1))`
Optimal outcome.

Methods

Public methods:

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

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

Usage:

```
OptimInstanceMultiCrit$new(objective, search_space = NULL, terminator)
```

Arguments:

`objective` (Objective).

`search_space` (paradox::ParamSet)
Specifies the search space for the Optimizer. The paradox::ParamSet describes either a subset of the `domain` of the Objective or it describes a set of parameters together with a `trafo` function that transforms values from the search space to values of the domain. Depending on the context, this value defaults to the domain of the objective.

`terminator` (Terminator)
Multi-criteria terminator.

Method `assign_result()`**:** The Optimizer object writes the best found points and estimated performance values here (probably the Pareto set / front). For internal use.

Usage:

```
OptimInstanceMultiCrit$assign_result(xdt, ydt)
```

Arguments:

`xdt` (`data.table::data.table()`)
Set of untransformed points / points from the *search space*. One point per row, e.g. `data.table(x1 = c(1,3),x2 = c(2,4))`. Column names have to match ids of the `search_space`. However, `xdt` can contain additional columns.

`ydt` (`numeric(1)`)
Optimal outcomes, e.g. the Pareto front.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`OptimInstanceSingleCrit`

Optimization Instance with budget and archive

Description

Wraps a single-criteria Objective function with extra services for convenient evaluation. Inherits from OptimInstance.

- Automatic storing of results in an Archive after evaluation.
- Automatic checking for termination. Evaluations of design points are performed in batches. 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.

Super class

`bbotk::OptimInstance` -> `OptimInstanceSingleCrit`

Methods

Public methods:

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

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

Usage:

```
OptimInstanceSingleCrit$new(objective, search_space = NULL, terminator)
```

Arguments:

`objective` (Objective).

`search_space` (paradox::ParamSet)
Specifies the search space for the Optimizer. The paradox::ParamSet describes either a subset of the `domain` of the Objective or it describes a set of parameters together with a `trafo` function that transforms values from the search space to values of the domain. Depending on the context, this value defaults to the domain of the objective.

`terminator` (Terminator).

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

Usage:

```
OptimInstanceSingleCrit$assign_result(xdt, y)
```

Arguments:

`xdt` (`data.table::data.table()`)
: Set of untransformed points / points from the *search space*. One point per row, e.g. `data.table(x1 = c(1,3),x2 = c(2,4))`. Column names have to match ids of the `search_space`. However, `xdt` can contain additional columns.

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

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`Optimizer` *Optimizer*

Description

Abstract `Optimizer` class that implements the base functionality each `Optimizer` subclass must provide. A `Optimizer` object describes the optimization strategy.

A `Optimizer` object must write its result to the $assign_result() method of the OptimInstance at the end in order to store the best point and its estimated performance vector.

Public fields

`param_set` (paradox::ParamSet).

`param_classes` (`character()`).

`properties` (`character()`).

`packages` (`character()`).

Methods

Public methods:

- `Optimizer$new()`
- `Optimizer$format()`
- `Optimizer$print()`
- `Optimizer$optimize()`
- `Optimizer$clone()`

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

Usage:

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

Arguments:

`param_set` (paradox::ParamSet).

`param_classes` (`character()`).

`properties` (`character()`).

`packages` (`character()`).

Method `format()`**:** Helper for print outputs.

Usage:

```
Optimizer$format()
```

Method `print()`**:** Print method.

Usage:

```
Optimizer$print()
```

Returns: (`character()`).

Method `optimize()`**:** Performs the optimization and writes optimization result into OptimInstance. The optimization result is returned but the complete optimization path is stored in Archive of OptimInstance.

Usage:

```
Optimizer$optimize(inst)
```

Arguments:

`inst` (OptimInstance).

Returns: data.table::data.table.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`OptimizerNLoptr` *Optimization via Non-linear Optimization*

Description

`OptimizerNLoptr` class that implements non-linear optimization. Calls `nloptr::nloptr()` from package **nloptr**.

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 class

`bbotk::Optimizer` -> `OptimizerNLoptr`

Methods

Public methods:

- `OptimizerNLoptr$new()`
- `OptimizerNLoptr$clone()`

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

Usage:

```
OptimizerNLoptr$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Source

Johnson SG (2020). "The NLopt nonlinear-optimization package." http://github.com/stevengj/nlopt.

Examples

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

domain = ParamSet$new(list(ParamDbl$new("x", lower = -1, upper = 1)))

search_space = ParamSet$new(list(ParamDbl$new("x", lower = -1, upper = 1)))

codomain = ParamSet$new(list(ParamDbl$new("y", tags = "minimize")))

objective_function = function(xs) {
  list(y = as.numeric(xs)^2)
}

objective = ObjectiveRFun$new(fun = objective_function,
  domain = domain,
  codomain = codomain)

# We use the internal termination criterion xtol_rel
terminator = trm("none")
instance = OptimInstanceSingleCrit$new(objective = objective,
  search_space = search_space,
  terminator = terminator)

optimizer = opt("nloptr", x0 = 1, algorithm = "NLOPT_LN_BOBYQA")

# Modifies the instance by reference
optimizer$optimize(instance)

# Returns best scoring evaluation
instance$result

# Allows access of data.table of full path of all evaluations
instance$archive$data()
```

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 optimization.

Termination of optimization works as follows:

- Evaluations in a instance are performed in batches.
- Before each batch evaluation, the Terminator is checked, and if it is positive, we stop.

- The optimization 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 optimization algorithms 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 optimization algorithms). So it is advised to check the size of the returned archive, in particular if you are benchmarking multiple optimization algorithms.

Public fields

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

`properties character()`
: Set of properties.

Methods

Public methods:

- `Terminator$new()`
- `Terminator$format()`
- `Terminator$print()`
- `Terminator$clone()`

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

Usage:

```
Terminator$new(param_set = ParamSet$new(), properties = character())
```

Arguments:

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

`properties` (`character()`)
: Set of properties.

Method `format()`**:** Helper for print outputs.

Usage:

```
Terminator$format()
```

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

Usage:

```
Terminator$print(...)
```

Arguments:

`...` (ignored).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

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

---

`trm` *Syntactic Sugar Terminator Construction*

Description

This function complements mlr_terminators with functions in the spirit of `mlr_sugar` from **mlr3**.

Usage

```
trm(.key, ...)

trms(.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

- Terminator for `trm()`.
- list of Terminator for `trms()`.

Examples

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

Index