Package 'bbotk'

November 6, 2025

Title Black-Box Optimization Toolkit

Version 1.8.0

Description Features highly configurable search spaces via the 'paradox' package and optimizes every user-defined objective function. The package includes several optimization algorithms e.g. Random Search, Iterated Racing, Bayesian Optimization (in 'mlr3mbo') and Hyperband (in 'mlr3hyperband'). bbotk is the base package of 'mlr3tuning', 'mlr3fselect' and 'miesmuschel'.

License LGPL-3

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

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

Depends paradox (>= 1.0.0), R (>= 3.1.0)

Imports checkmate (>= 2.0.0), cli, data.table, lgr, methods, mlr3misc (>= 0.15.1), R6

Suggests adagio, emoa, GenSA, irace (>= 4.0.0), knitr, mirai, nloptr, processx, progressr, redux, RhpcBLASctl, rush (>= 0.4.1), testthat (>= 3.0.0)

Config/testthat/edition 3

Config/testthat/parallel false

Encoding UTF-8

Language en-US

NeedsCompilation yes

RoxygenNote 7.3.3

Collate 'Archive.R' 'ArchiveAsync.R' 'ArchiveAsyncFrozen.R' 'ArchiveBatch.R' 'CallbackAsync.R' 'CallbackBatch.R' 'Codomain.R' 'ContextAsync.R' 'ContextBatch.R' 'Objective.R' 'ObjectiveRFun.R' 'ObjectiveRFunDt.R' 'ObjectiveRFunMany.R' 'OptimInstance.R' 'OptimInstanceAsync.R' 'OptimInstanceAsyncMultiCrit.R' 'OptimInstanceAsyncSingleCrit.R' 'OptimInstanceBatch.R' 'OptimInstanceBatchMultiCrit.R'

'OptimInstanceBatchSingleCrit.R' 'OptimInstanceMultiCrit.R'
'OptimInstanceSingleCrit.R' 'mlr_optimizers.R' 'Optimizer.R'
'OptimizerAsync.R' 'OptimizerAsyncDesignPoints.R'
'OptimizerAsyncGridSearch.R' 'OptimizerAsyncRandomSearch.R'
'OptimizerBatch.R' 'OptimizerBatchChain.R'
'OptimizerBatchCmaes.R' 'OptimizerBatchDesignPoints.R'
'OptimizerBatchFocusSearch.R' 'OptimizerBatchGenSA.R'
'OptimizerBatchGridSearch.R' 'OptimizerBatchIrace.R'
'OptimizerBatchLocalSearch.R' 'OptimizerBatchNLoptr.R'
'OptimizerBatchRandomSearch.R' 'Progressor.R'
'mlr_terminators.R' 'Terminator.R' 'TerminatorClockTime.R'
'TerminatorCombo.R' 'TerminatorEvals.R' 'TerminatorNone.R'
'TerminatorPerfReached.R' 'TerminatorRunTime.R'
'TerminatorStagnation.R' 'TerminatorStagnationBatch.R'
'TerminatorStagnationHypervolume.R' 'as_terminator.R'
'assertions.R' 'bb_optimize.R' 'bbotk_reflections.R'
'bibentries.R' 'helper.R' 'local_search.R' 'mlr_callbacks.R'
'nds_selection.R' 'reexport.R' 'sugar.R' 'worker_loops.R'
'zzz.R'

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

Maintainer Marc Becker <marcbecker@posteo.de>

Repository CRAN

Date/Publication 2025-11-06 16:30:02 UTC

Contents

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

Description

Features highly configurable search spaces via the 'paradox' package and optimizes every user-defined objective function. The package includes several optimization algorithms e.g. Random Search, Iterated Racing, Bayesian Optimization (in 'mlr3mbo') and Hyperband (in 'mlr3hyperband'). bbotk is the base package of 'mlr3tuning', 'mlr3fselect' and 'miesmuschel'.

Package Options

- `"bbotk.debug"`: If set to `TRUE`, asynchronous optimization is run in the main process.
- `"bbotk.tiny_logging"`: If set to `TRUE`, the logging is simplified to only show points and results. NA values are removed.

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` *Data Storage*

Description

The `Archive` class stores all evaluated points and performance scores

Details

The `Archive` is an abstract class that implements the base functionality each archive must provide.

Public fields

`search_space` (paradox::ParamSet)
: Specification of the search space for the Optimizer.

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

`start_time` (POSIXct)
: Time stamp of when the optimization started. The time is set by the Optimizer.

`check_values` (`logical(1)`)
: Determines if points and results are checked for validity.

Active bindings

`label` (`character(1)`)
: Label for this object. Can be used in tables, plot and text output instead of the ID.

`man` (`character(1)`)
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

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

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

Methods

Public methods:

- `Archive$new()`
- `Archive$format()`
- `Archive$print()`
- `Archive$clear()`
- `Archive$help()`
- `Archive$clone()`

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

Usage:

```
Archive$new(
  search_space,
  codomain,
  check_values = FALSE,
  label = NA_character_,
  man = NA_character_
)
```

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.

`label` (`character(1)`)
: Label for this object. Can be used in tables, plot and text output instead of the ID.

`man` (`character(1)`)
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

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

Usage:

```
Archive$format(...)
```

Arguments:

`...` (ignored).

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

Usage:

```
Archive$print()
```

Arguments:

`...` (ignored).

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

Usage:

```
Archive$clear()
```

Method `help()`**:** Opens the corresponding help page referenced by field `$man`.

Usage:

```
Archive$help()
```

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.

See Also

ArchiveBatch, ArchiveAsync

`ArchiveAsync` *Rush Data Storage*

Description

The `ArchiveAsync` stores all evaluated points and performance scores in a rush::Rush data base.

S3 Methods

- `as.data.table(archive)`
  ArchiveAsync -> `data.table::data.table()`
  Returns a tabular view of all performed function calls of the Objective. The `x_domain` column is unnested to separate columns.

Super class

`bbotk::Archive` -> `ArchiveAsync`

Public fields

`rush` (`Rush`)
Rush controller for parallel optimization.

Active bindings

`data` (data.table::data.table)
Data table with all finished points.

`queued_data` (data.table::data.table)
Data table with all queued points.

`running_data` (data.table::data.table)
Data table with all running points.

`finished_data` (data.table::data.table)
Data table with all finished points.

`failed_data` (data.table::data.table)
Data table with all failed points.

`n_queued` (`integer(1)`)
Number of queued points.

`n_running` (`integer(1)`)
Number of running points.

`n_finished` (`integer(1)`)
Number of finished points.

`n_failed` (`integer(1)`)
Number of failed points.

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

Methods

Public methods:

- `ArchiveAsync$new()`
- `ArchiveAsync$push_points()`
- `ArchiveAsync$pop_point()`
- `ArchiveAsync$push_running_point()`
- `ArchiveAsync$push_result()`
- `ArchiveAsync$push_failed_point()`
- `ArchiveAsync$data_with_state()`
- `ArchiveAsync$best()`
- `ArchiveAsync$nds_selection()`
- `ArchiveAsync$clear()`
- `ArchiveAsync$clone()`

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

Usage:

```
ArchiveAsync$new(search_space, codomain, check_values = FALSE, rush)
```

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 points before the evaluation and the results be checked for validity?

`rush` (`Rush`)
If a rush instance is supplied, the tuning runs without batches.

Method `push_points()`**:** Push queued points to the archive.

Usage:

```
ArchiveAsync$push_points(xss)
```

Arguments:

`xss` (list of named `list()`)
List of named lists of point values.

Method `pop_point()`**:** Pop a point from the queue.

Usage:

```
ArchiveAsync$pop_point()
```

Method `push_running_point()`**:** Push running point to the archive.

Usage:

```
ArchiveAsync$push_running_point(xs, extra = NULL)
```

Arguments:

`xs` (named `list`)
Named list of point values.

`extra` (`list()`)
Named list of additional information.

Method `push_result()`**:** Push result to the archive.

Usage:

```
ArchiveAsync$push_result(key, ys, x_domain, extra = NULL)
```

Arguments:

`key` (`character()`)
Key of the point.

`ys` (`list()`)
Named list of results.

`x_domain` (`list()`)
Named list of transformed point values.

`extra` (`list()`)
Named list of additional information.

Method `push_failed_point()`**:** Push failed point to the archive.

Usage:

```
ArchiveAsync$push_failed_point(key, message)
```

Arguments:

`key` (`character()`)
Key of the point.

`message` (`character()`)
Error message.

Method `data_with_state()`**:** Fetch points with a specific state.

Usage:

```
ArchiveAsync$data_with_state(
  fields = c("xs", "ys", "xs_extra", "worker_extra", "ys_extra", "condition"),
  states = c("queued", "running", "finished", "failed"),
  reset_cache = FALSE
)
```

Arguments:

`fields` (`character()`)
: Fields to fetch. Defaults to `c("xs", "ys", "xs_extra", "worker_extra", "ys_extra")`.

`states` (`character()`)
: States of the tasks to be fetched. Defaults to `c("queued", "running", "finished", "failed")`.

`reset_cache` (`logical(1)`)
: Whether to reset the cache of the finished points.

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

Usage:

```
ArchiveAsync$best(n_select = 1, ties_method = "first")
```

Arguments:

`n_select` (`integer(1L)`)
: Amount of points to select. Ignored for multi-crit optimization.

`ties_method` (`character(1L)`)
: Method to break ties when multiple points have the same score. Either `"first"` (default) or `"random"`. Ignored for multi-crit optimization. If `n_select > 1L`, the tie method is ignored and the first point is returned.

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

Method `nds_selection()`**:** Calculate best points w.r.t. non dominated sorting with hypervolume contribution.

Usage:

```
ArchiveAsync$nds_selection(n_select = 1, ref_point = NULL)
```

Arguments:

`n_select` (`integer(1L)`)
: Amount of points to select.

`ref_point` (`numeric()`)
: Reference point for hypervolume.

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

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

Usage:

```
ArchiveAsync$clear()
```

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
if (mlr3misc::require_namespaces(c("rush", "redux", "mirai"), quietly = TRUE) &&
  redux::redis_available()) {
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# start workers
rush::rush_plan(worker_type = "remote")
mirai::daemons(1)

# initialize instance
instance = oi_async(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)

# load optimizer
optimizer = opt("async_random_search")

# trigger optimization
optimizer$optimize(instance)

# all evaluated configuration
instance$archive
```

```
# best performing configuration
instance$archive$best()

# covert to data.table
as.data.table(instance$archive)

# reset the rush data base
instance$rush$reset()
}
```

`ArchiveAsyncFrozen` *Frozen Rush Data Storage*

Description

Freezes the Redis data base of an ArchiveAsync to a `data.table::data.table()`. No further points can be added to the archive but the data can be accessed and analyzed. Useful when the Redis data base is not permanently available. Use the callback bbotk.async_freeze_archive to freeze the archive after the optimization has finished.

S3 Methods

- `as.data.table(archive)`
  ArchiveAsync -> `data.table::data.table()`
  Returns a tabular view of all performed function calls of the Objective. The `x_domain` column is unnested to separate columns.

Super classes

`bbotk::Archive` -> `bbotk::ArchiveAsync` -> `ArchiveAsyncFrozen`

Active bindings

`data` (data.table::data.table)
: Data table with all finished points.

`queued_data` (data.table::data.table)
: Data table with all queued points.

`running_data` (data.table::data.table)
: Data table with all running points.

`finished_data` (data.table::data.table)
: Data table with all finished points.

`failed_data` (data.table::data.table)
: Data table with all failed points.

`n_queued` (`integer(1)`)
: Number of queued points.

`n_running` (`integer(1)`)
Number of running points.

`n_finished` (`integer(1)`)
Number of finished points.

`n_failed` (`integer(1)`)
Number of failed points.

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

Methods

Public methods:

- `ArchiveAsyncFrozen$new()`
- `ArchiveAsyncFrozen$push_points()`
- `ArchiveAsyncFrozen$pop_point()`
- `ArchiveAsyncFrozen$push_running_point()`
- `ArchiveAsyncFrozen$push_result()`
- `ArchiveAsyncFrozen$push_failed_point()`
- `ArchiveAsyncFrozen$data_with_state()`
- `ArchiveAsyncFrozen$clear()`
- `ArchiveAsyncFrozen$clone()`

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

Usage:

```
ArchiveAsyncFrozen$new(archive)
```

Arguments:

`archive` (ArchiveAsync)
The archive to freeze.

Method `push_points()`**:** Push queued points to the archive.

Usage:

```
ArchiveAsyncFrozen$push_points(xss)
```

Arguments:

`xss` (list of named `list()`)
List of named lists of point values.

Method `pop_point()`**:** Pop a point from the queue.

Usage:

```
ArchiveAsyncFrozen$pop_point()
```

Method `push_running_point()`**:** Push running point to the archive.

Usage:

```
ArchiveAsyncFrozen$push_running_point(xs, extra = NULL)
```

Arguments:

`xs` (named `list`)
: Named list of point values.

`extra` (`list()`)
: Named list of additional information.

Method `push_result()`**:** Push result to the archive.

Usage:

```
ArchiveAsyncFrozen$push_result(key, ys, x_domain, extra = NULL)
```

Arguments:

`key` (`character()`)
: Key of the point.

`ys` (`list()`)
: Named list of results.

`x_domain` (`list()`)
: Named list of transformed point values.

`extra` (`list()`)
: Named list of additional information.

Method `push_failed_point()`**:** Push failed point to the archive.

Usage:

```
ArchiveAsyncFrozen$push_failed_point(key, message)
```

Arguments:

`key` (`character()`)
: Key of the point.

`message` (`character()`)
: Error message.

Method `data_with_state()`**:** Fetch points with a specific state.

Usage:

```
ArchiveAsyncFrozen$data_with_state(
  fields = c("xs", "ys", "xs_extra", "worker_extra", "ys_extra", "condition"),
  states = c("queued", "running", "finished", "failed"),
  reset_cache = FALSE
)
```

Arguments:

`fields` (`character()`)
: Fields to fetch. Defaults to `c("xs", "ys", "xs_extra", "worker_extra", "ys_extra")`.

`states` (`character()`)
: States of the tasks to be fetched. Defaults to `c("queued", "running", "finished", "failed")`.

`reset_cache` (`logical(1)`)
: Whether to reset the cache of the finished points.

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

Usage:

```
ArchiveAsyncFrozen$clear()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

ArchiveAsync

Examples

```
# example only runs if a Redis server is available
if (mlr3misc::require_namespaces(c("rush", "redux", "mirai"), quietly = TRUE) &&
  redux::redis_available()) {
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# start workers
rush::rush_plan(worker_type = "remote")
mirai::daemons(1)

# initialize instance
instance = oi_async(
  objective = objective,
  terminator = trm("evals", n_evals = 20),
  callback = clbk("bbotk.async_freeze_archive")
)
```

```
# load optimizer
optimizer = opt("async_random_search")

# trigger optimization
optimizer$optimize(instance)

# frozen archive
instance$archive

# best performing configuration
instance$archive$best()

# covert to data.table
as.data.table(instance$archive)
}
```

`ArchiveBatch` *Data Table Storage*

Description

The `ArchiveBatch` stores all evaluated points and performance scores in a `data.table::data.table()`.

S3 Methods

- `as.data.table(archive)`
  ArchiveBatch -> `data.table::data.table()`
  Returns a tabular view of all performed function calls of the Objective. The `x_domain` column is unnested to separate columns.

Super class

`bbotk::Archive` -> `ArchiveBatch`

Public fields

`data` (data.table::data.table)
: Contains all performed Objective function calls.

`data_extra` (named `list`)
: Data created by specific `Optimizer`s that does not relate to any individual function evaluation and can therefore not be held in `$data`. Every optimizer should create and refer to its own entry in this list, named by its `class()`.

Active bindings

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

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

Methods

Public methods:

- `ArchiveBatch$new()`
- `ArchiveBatch$add_evals()`
- `ArchiveBatch$best()`
- `ArchiveBatch$nds_selection()`
- `ArchiveBatch$clear()`
- `ArchiveBatch$clone()`

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

Usage:

```
ArchiveBatch$new(search_space, codomain, check_values = FALSE)
```

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:

```
ArchiveBatch$add_evals(xdt, xss_trafoed = NULL, 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(s). For single-crit optimization, the solution that minimizes / maximizes the objective function. For multi-crit optimization, the Pareto set / front.

Usage:

```
ArchiveBatch$best(batch = NULL, n_select = 1L, ties_method = "first")
```

Arguments:

`batch (integer())`
The batch number(s) to limit the best results to. Default is all batches.

`n_select (integer(1L))`
Amount of points to select. Ignored for multi-crit optimization.

`ties_method (character(1L))`
Method to break ties when multiple points have the same score. Either `"first"` (default) or `"random"`. Ignored for multi-crit optimization. If `n_select > 1L`, the tie method is ignored and the first point is returned.

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

Method `nds_selection()`**:** Calculate best points w.r.t. non dominated sorting with hypervolume contribution.

Usage:

```
ArchiveBatch$nds_selection(batch = NULL, n_select = 1, ref_point = NULL)
```

Arguments:

`batch (integer())`
The batch number(s) to limit the best points to. Default is all batches.

`n_select (integer(1L))`
Amount of points to select.

`ref_point (numeric())`
Reference point for hypervolume.

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

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

Usage:

```
ArchiveBatch$clear()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)
```

```
# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# initialize instance
instance = oi(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)

# load optimizer
optimizer = opt("random_search")

# trigger optimization
optimizer$optimize(instance)

# all evaluated configuration
instance$archive

# best performing configuration
instance$archive$best()

# covert to data.table
as.data.table(instance$archive)
```

`as_terminator` *Convert to a Terminator*

Description

Convert object to a Terminator or a list of Terminator.

Usage

```
as_terminator(x, ...)

## S3 method for class 'Terminator'
as_terminator(x, clone = FALSE, ...)

as_terminators(x, ...)
```

```
## Default S3 method:
as_terminators(x, ...)

## S3 method for class 'list'
as_terminators(x, ...)
```

Arguments

x	(any) Object to convert.
...	(any) Additional arguments.
clone	(logical(1)) If TRUE, ensures that the returned object is not the same as the input x.

See Also

Terminator

---

bbotk.async_freeze_archive *Freeze Archive Callback*

Description

This CallbackAsync freezes the ArchiveAsync to ArchiveAsyncFrozen after the optimization has finished.

Examples

```
clbk("bbotk.async_freeze_archive")
```

---

bbotk.backup *Backup Archive Callback*

Description

This CallbackBatch writes the Archive after each batch to disk.

Examples

```
clbk("bbotk.backup", path = "backup.rds")
```

bb_optimize *Black-Box Optimization*

Description

This function optimizes a function or Objective with a given method.

Usage

```
bb_optimize(
  x,
  method = "random_search",
  max_evals = 1000,
  max_time = NULL,
  ...
)

## S3 method for class '`function`'
bb_optimize(
  x,
  method = "random_search",
  max_evals = 1000,
  max_time = NULL,
  lower = NULL,
  upper = NULL,
  maximize = FALSE,
  ...
)

## S3 method for class 'Objective'
bb_optimize(
  x,
  method = "random_search",
  max_evals = 1000,
  max_time = NULL,
  search_space = NULL,
  ...
)
```

Arguments

x	(function | Objective).
method	(character(1) | Optimizer) Key to retrieve optimizer from mlr_optimizers dictionary or Optimizer.
max_evals	(integer(1)) Number of allowed evaluations.

`max_time`	(`integer(1)`) Maximum allowed time in seconds.
`...`	(named `list()`) Named arguments passed to objective function. Ignored if Objective is optimized.
`lower`	(`numeric()`) Lower bounds on the parameters. If named, names are used to create the domain.
`upper`	(`numeric()`) Upper bounds on the parameters.
`maximize`	(`logical()`) Logical vector used to create the codomain e.g. c(TRUE, FALSE) -> ps(y1 = p_dbl(tags = "maximize"), y2 = pd_dbl(tags = "minimize")). If named, names are used to create the codomain.
`search_space`	(paradox::ParamSet).

Value

`list` of

- `"par"` - Best found parameters
- `"value"` - Optimal outcome
- `"instance"` - OptimInstanceBatchSingleCrit | OptimInstanceBatchMultiCrit

Note

If both `max_evals` and `max_time` are `NULL`, TerminatorNone is used. This is useful if the Optimizer can terminate itself. If both are given, TerminatorCombo is created and the optimization stops if the time or evaluation budget is exhausted.

Examples

```
# function and bounds
fun = function(xs) {
  -(xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10
}

bb_optimize(fun, lower = c(-10, -5), upper = c(10, 5), max_evals = 10)

# function and constant
fun = function(xs, c) {
  -(xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + c
}

bb_optimize(fun, lower = c(-10, -5), upper = c(10, 5), max_evals = 10, c = 1)

# objective
fun = function(xs) {
  c(z = -(xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}
```

```
# define domain and codomain using a `ParamSet` from paradox
domain = ps(x1 = p_dbl(-10, 10), x2 = p_dbl(-5, 5))
codomain = ps(z = p_dbl(tags = "minimize"))
objective = ObjectiveRFun$new(fun, domain, codomain)

bb_optimize(objective, method = "random_search", max_evals = 10)
```

`branin` *Branin Function*

Description

Classic 2-D Branin function with noise `branin(x1, x2, noise)` and Branin function with fidelity parameter `branin_wu(x1, x2, fidelity)`.

Usage

```
branin(x1, x2, noise = 0)

branin_wu(x1, x2, fidelity)
```

Arguments

`x1`	(`numeric()`).
`x2`	(`numeric()`).
`noise`	(`numeric()`).
`fidelity`	(`numeric()`).

Value

`numeric()`

Source

Wu J, Toscano-Palmerin S, Frazier PI, Wilson AG (2019). "Practical Multi-fidelity Bayesian Optimization for Hyperparameter Tuning." 1903.04703.

Examples

```
branin(x1 = 12, x2 = 2, noise = 0.05)
branin_wu(x1 = 12, x2 = 2, fidelity = 1)
```

CallbackAsync *Create Asynchronous Optimization Callback*

Description

Specialized mlr3misc::Callback for asynchronous optimization. Callbacks allow to customize the behavior of processes in bbotk. The `callback_async()` function creates a CallbackAsync. Predefined callbacks are stored in the dictionary mlr_callbacks and can be retrieved with `clbk()`. For more information on optimization callbacks see `callback_async()`.

Super class

`mlr3misc::Callback` -> `CallbackAsync`

Public fields

`on_optimization_begin (function())`
Stage called at the beginning of the optimization in the main process. Called in `Optimizer$optimize()`.

`on_worker_begin (function())`
Stage called at the beginning of the optimization on the worker. Called in the worker loop.

`on_optimizer_before_eval (function())`
Stage called after the optimizer proposes points. Called in `OptimInstance$.eval_point()`.

`on_optimizer_after_eval (function())`
Stage called after points are evaluated. Called in `OptimInstance$.eval_point()`.

`on_optimizer_queue_before_eval (function())`
Stage called after the optimizer proposes points. Called in `OptimInstance$.eval_queue()`.

`on_optimizer_queue_after_eval (function())`
Stage called after points are evaluated. Called in `OptimInstance$.eval_queue()`.

`on_worker_end (function())`
Stage called at the end of the optimization on the worker. Called in the worker loop.

`on_result_begin (function())`
Stage called before the results are written. Called in `OptimInstance$assign_result()`.

`on_result_end (function())`
Stage called after the results are written. Called in `OptimInstance$assign_result()`.

`on_optimization_end (function())`
Stage called at the end of the optimization in the main process. Called in `Optimizer$optimize()`.

Methods

Public methods:

- `CallbackAsync$clone()`

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

callback_async()

CallbackBatch *Create Batch Optimization Callback*

Description

Specialized mlr3misc::Callback for batch optimization. Callbacks allow to customize the behavior of processes in bbotk. The callback_batch() function creates a CallbackBatch. Predefined callbacks are stored in the dictionary mlr_callbacks and can be retrieved with clbk(). For more information on optimization callbacks see callback_batch().

Super class

mlr3misc::Callback -> CallbackBatch

Public fields

on_optimization_begin (function())
: Stage called at the beginning of the optimization. Called in Optimizer$optimize().

on_optimizer_before_eval (function())
: Stage called after the optimizer proposes points. Called in OptimInstance$eval_batch().

on_optimizer_after_eval (function())
: Stage called after points are evaluated. Called in OptimInstance$eval_batch().

on_result_begin (function())
: Stage called before the results are written. Called in OptimInstance$assign_result().

on_result_end (function())
: Stage called after the results are written. Called in OptimInstance$assign_result().

on_optimization_end (function())
: Stage called at the end of the optimization. Called in Optimizer$optimize().

Methods

Public methods:

- CallbackBatch$clone()

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

callback_batch()

Examples

```
# write archive to disk
callback_batch("bbotk.backup",
  on_optimization_end = function(callback, context) {
    saveRDS(context$instance$archive, "archive.rds")
  }
)
```

---

callback_async *Create Asynchronous Optimization Callback*

Description

Function to create a CallbackAsync.

Optimization callbacks can be called from different stages of optimization process. The stages are prefixed with on_*.

```
Start Optimization
     - on_optimization_begin
    Start Worker
         - on_worker_begin
           Start Optimization on Worker
             - on_optimizer_before_eval
             - on_optimizer_after_eval
           End Optimization on Worker
         - on_worker_end
    End Worker
     - on_result_begin
     - on_result_end
     - on_optimization_end
End Optimization
```

See also the section on parameters for more information on the stages. A optimization callback works with ContextAsync.

Usage

```
callback_async(
  id,
  label = NA_character_,
  man = NA_character_,
  on_optimization_begin = NULL,
  on_worker_begin = NULL,
```

```
  on_optimizer_before_eval = NULL,
  on_optimizer_after_eval = NULL,
  on_optimizer_queue_before_eval = NULL,
  on_optimizer_queue_after_eval = NULL,
  on_worker_end = NULL,
  on_result_begin = NULL,
  on_result_end = NULL,
  on_result = NULL,
  on_optimization_end = NULL
)
```

Arguments

id	(character(1)) Identifier for the new instance.
label	(character(1)) Label for the new instance.
man	(character(1)) String in the format [pkg]::[topic] pointing to a manual page for this object. The referenced help package can be opened via method $help().
on_optimization_begin	(function()) Stage called at the beginning of the optimization in the main process. Called in Optimizer$optimize(). The functions must have two arguments named callback and context.
on_worker_begin	(function()) Stage called at the beginning of the optimization on the worker. Called in the worker loop. The functions must have two arguments named callback and context.
on_optimizer_before_eval	(function()) Stage called after the optimizer proposes points. Called in OptimInstance$.eval_point(). The functions must have two arguments named callback and context. The argument of instance$.eval_point(xs) and xs_trafoed and extra are available in the context. Or xs and xs_trafoed of instance$.eval_queue() are available in the context.
on_optimizer_after_eval	(function()) Stage called after points are evaluated. Called in OptimInstance$.eval_point(). The functions must have two arguments named callback and context. The outcome y is available in the context.
on_optimizer_queue_before_eval	(function()) Stage called before a point in the queue is evaluated. Called in OptimInstance$.eval_queue(). The functions must have two arguments named callback and context. The argument of instance$.eval_queue(xs) and xs_trafoed and extra are available in the context.

`on_optimizer_queue_after_eval`
`(function())`
Stage called after a point in the queue is evaluated. Called in `OptimInstance$.eval_queue()`. The functions must have two arguments named `callback` and `context`. The outcome `y` is available in the `context`.

`on_worker_end` `(function())`
Stage called at the end of the optimization on the worker. Called in the worker loop. The functions must have two arguments named `callback` and `context`.

`on_result_begin`
`(function())`
Stage called before result are written. Called in `OptimInstance$assign_result()`. The functions must have two arguments named `callback` and `context`. The arguments of `$.assign_result(xdt, y, extra)` are available in the `context`.

`on_result_end` `(function())`
Stage called after result are written. Called in `OptimInstance$assign_result()`. The functions must have two arguments named `callback` and `context`. The final result `instance$result` is available in the `context`.

`on_result` `(function())`
Deprecated. Use `on_result_end` instead. Stage called after result are written. Called in `OptimInstance$assign_result()`. The functions must have two arguments named `callback` and `context`.

`on_optimization_end`
`(function())`
Stage called at the end of the optimization in the main process. Called in `Optimizer$optimize()`. The functions must have two arguments named `callback` and `context`.

Details

A callback can write data to its state (`$state`), e.g. settings that affect the callback itself. The ContextAsync allows to modify the instance, archive, optimizer and final result.

Value

CallbackAsync

`callback_batch` *Create Batch Optimization Callback*

Description

Function to create a CallbackBatch.

Optimization callbacks can be called from different stages of optimization process. The stages are prefixed with `on_*`.

```
Start Optimization
     - on_optimization_begin
    Start Optimizer Batch
         - on_optimizer_before_eval
         - on_optimizer_after_eval
    End Optimizer Batch
     - on_result_begin
     - on_result_end
     - on_optimization_end
End Optimization
```

See also the section on parameters for more information on the stages. A optimization callback works with ContextBatch.

Usage

```
callback_batch(
  id,
  label = NA_character_,
  man = NA_character_,
  on_optimization_begin = NULL,
  on_optimizer_before_eval = NULL,
  on_optimizer_after_eval = NULL,
  on_result_begin = NULL,
  on_result_end = NULL,
  on_result = NULL,
  on_optimization_end = NULL
)
```

Arguments

`id`	(`character(1)`) Identifier for the new instance.
`label`	(`character(1)`) Label for the new instance.
`man`	(`character(1)`) String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.
`on_optimization_begin`	(`function()`) Stage called at the beginning of the optimization. Called in `Optimizer$optimize()`. The functions must have two arguments named `callback` and `context`.
`on_optimizer_before_eval`	(`function()`) Stage called after the optimizer proposes points. Called in `OptimInstance$eval_batch()`. The functions must have two arguments named `callback` and `context`. The argument of `$eval_batch(xdt)` is available in `context`.

`on_optimizer_after_eval`
: `(function())`
Stage called after points are evaluated. Called in `OptimInstance$eval_batch()`. The functions must have two arguments named `callback` and `context`. The new points and outcomes in `instance$archive` are available in `context`.

`on_result_begin`
: `(function())`
Stage called before result are written to the instance. Called in `OptimInstance$assign_result()`. The functions must have two arguments named `callback` and `context`. The arguments of `$assign_result(xdt, y, extra)` are available in `context`.

`on_result_end` `(function())`
: Stage called after result are written to the instance. Called in `OptimInstance$assign_result()`. The functions must have two arguments named `callback` and `context`. The final result `instance$result` is available in `context`.

`on_result` `(function())`
: Deprecated. Use `on_result_end` instead. Stage called after result are written. Called in `OptimInstance$assign_result()`. The functions must have two arguments named `callback` and `context`.

`on_optimization_end`
: `(function())`
Stage called at the end of the optimization. Called in `Optimizer$optimize()`. The functions must have two arguments named `callback` and `context`.

Details

A callback can write data to its state (`$state`), e.g. settings that affect the callback itself. The ContextBatch allows to modify the instance, archive, optimizer and final result.

Value

CallbackBatch

Examples

```
# write archive to disk
callback_batch("bbotk.backup",
  on_optimization_end = function(callback, context) {
    saveRDS(context$instance$archive, "archive.rds")
  }
)
```

`Codomain` *Codomain of Function*

Description

A paradox::ParamSet defining the codomain of a function. The parameter set must contain at least one target parameter tagged with `"minimize"` or `"maximize"`. The codomain may contain extra parameters which are ignored when calling the Archive methods `$best()`, `$nds_selection()` and `$cols_y`. This class is usually constructed internally from a paradox::ParamSet when Objective is initialized.

Super class

`paradox::ParamSet` -> `Codomain`

Active bindings

`is_target` (named `logical()`)
Position is `TRUE` for target parameters.

`target_length` (`integer()`)
Returns number of target parameters.

`target_ids` (`character()`)
IDs of contained target parameters.

`target_tags` (named `list()` of `character()`)
Tags of target parameters.

`maximization_to_minimization` (`integer()`)
Returns a numeric vector with values -1 and 1. Multiply with the outcome of a maximization problem to turn it into a minimization problem.

`direction` (`integer()`)
Returns `1` for minimization and `-1` for maximization. If the codomain contains multiple parameters an integer vector is returned. Multiply with the outcome of a maximization problem to turn it into a minimization problem.

Methods

Public methods:

- `Codomain$new()`
- `Codomain$clone()`

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

Usage:

```
Codomain$new(params)
```

Arguments:

`params` (`list()`)
Named list with which to initialize the codomain. This argument is analogous to paradox::ParamSet's `$initialize()` `params` argument.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
# define objective function
fun = function(xs) {
  c(y = -(xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize"),
  time = p_dbl()
)

# create Objective object
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)
```

ContextAsync *Asynchronous Optimization Context*

Description

A CallbackAsync accesses and modifies data during the optimization via the `ContextAsync`. See the section on active bindings for a list of modifiable objects. See `callback_async()` for a list of stages which access `ContextAsync`.

Details

Changes to `$instance` and `$optimizer` in the stages executed on the workers are not reflected in the main process.

Super class

`mlr3misc::Context` -> `ContextAsync`

Public fields

`instance` (OptimInstance).

`optimizer` (Optimizer).

Active bindings

`xs` (list())
: The point to be evaluated in `instance$.eval_point()`.

`xs_trafoed` (list())
: The transformed point to be evaluated in `instance$.eval_point()`.

`extra` (list())
: Additional information of the point to be evaluated in `instance$.eval_point()`.

`ys` (list())
: The result of the evaluation in `instance$.eval_point()`.

`result_xdt` (data.table::data.table)
: The xdt passed to `instance$assign_result()`.

`result_y` (`numeric(1)`)
: The y passed to `instance$assign_result()`. Only available for single criterion optimization.

`result_ydt` (data.table::data.table)
: The ydt passed to `instance$assign_result()`. Only available for multi criterion optimization.

`result_extra` (data.table::data.table)
: Additional information about the result passed to `instance$assign_result()`.

`result` (data.table::data.table)
: The result of the optimization in `instance$assign_result()`.

Methods

Public methods:

- `ContextAsync$new()`
- `ContextAsync$clone()`

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

Usage:

```
ContextAsync$new(inst, optimizer)
```

Arguments:

`inst` (OptimInstance).

`optimizer` (Optimizer).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

CallbackAsync

`ContextBatch` *Batch Optimization Context*

Description

A CallbackBatch accesses and modifies data during the optimization via the `ContextBatch`. See the section on active bindings for a list of modifiable objects. See `callback_batch()` for a list of stages which that `ContextBatch`.

Super class

`mlr3misc::Context` -> `ContextBatch`

Public fields

`instance` (OptimInstance).

`optimizer` (Optimizer).

Active bindings

`xdt` (data.table::data.table)
: The points of the latest batch in `instance$eval_batch()`. Contains the values in the search space i.e. transformations are not yet applied.

`result_xdt` (data.table::data.table)
: The xdt passed to `instance$assign_result()`.

`result_y` (`numeric(1)`)
: The y passed to `instance$assign_result()`. Only available for single criterion optimization.

`result_ydt` (data.table::data.table)
: The ydt passed to `instance$assign_result()`. Only available for multi criterion optimization.

`result_extra` (data.table::data.table)
: Additional information about the result passed to `instance$assign_result()`.

`result` (data.table::data.table)
: The result of the optimization in `instance$assign_result()`.

Methods

Public methods:

- `ContextBatch$new()`
- `ContextBatch$clone()`

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

Usage:

```
ContextBatch$new(inst, optimizer)
```

Arguments:

`inst` (OptimInstance).

`optimizer` (Optimizer).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

CallbackBatch

`is_dominated` *Calculate which points are dominated*

Description

Returns which points from a set are dominated by another point in the set.

Usage

```
is_dominated(ymat)
```

Arguments

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

Value

`logical()` with `TRUE` if a point (column of `ymat`) is dominated.

`local_search` *Local Search*

Description

Runs a local search on the objective function. Somewhat similar to what is used in SMAC for acquisition function optimization of mixed type search spaces with hierarchical dependencies.

The function always minimizes. If the objective is to be maximized, we handle it by multiplying with "obj_mult" (which will be -1).

'Currently, automatically applying the search space transformations is not supported, if you need this, do this yourself in the objective function or use `OptimInstanceBatchLocalSearch`.

Usage

```
local_search(
  objective,
  search_space,
  control = local_search_control(),
  init_points = NULL
)
```

Arguments

`objective`	(`function(xdt)`) Objective to optimize. The first arg (name 'xdt' is not enforced) will be a data.table with (scalar) columns corresponding exactly the search space, in the same order. The function should must return numeric vector of exactly the same length as the number of rows in the dt, containing the objective values.
`search_space`	(paradox::ParamSet) Search space for decision variables. Must be non-empty, can only contain `p_int`, `p_dbl`, `p_fct`, `p_lgl`, all must be bounded.
`control`	(local_search_control) Control parameters for the local search, generated by `local_search_control()`.
`init_points`	(`data.table`) Initial points to start the local search from, same format as described for the argument of 'objective'. Must have as many rows as 'control$n_searches'. If NULL, we generate "n_searches" random points.

Details

We run "n_searches" in parallel. Each search runs "n_steps" iterations. For each search in every iteration we generate "n_neighs" neighbors. A neighbor is the current point, but with exactly one parameter mutated.

Mutation works like this: For num params: we scale to 0,1, add Gaussian noise with sd "mut_sd", and scale back. We then clip to the lower and upper bounds. For int params: We do the same as

for numeric parameters, but round at the end. For factor params: We sample a new level from the unused levels of the parameter. For logical params: We flip the bit.

Hierarchical dependencies are handled like this: Only active params can be mutated. After a mutation has happened, we check the conditions of the search space in topological order. If a condition is not met, we set the param to NA (making it inactive); if all conditions are met for a param, but it currently has is NA, we set it a random valid value.

After the neighbors are generated, we evaluate them. We go to the best neighbor, or stay at the current point if the best neighbor is worse.

There is a restart mechanism to avoid local minima. For each search, we keep track of the number of no-improvement steps. If this number exceeds "stagnate_max", we restart the search with a random point.

Value

(named `list`). List with elements:

- 'x': (`list`)
  The best point found, length and element names and their order correspond exactly to the search space.
- 'y': (`numeric(1)`)
  The objective value of the best point.

`local_search_control` *Local Search Control*

Description

Control parameters for local search optimizer, see `local_search()` for details.

Usage

```
local_search_control(
  minimize = TRUE,
  n_searches = 10L,
  n_steps = 5L,
  n_neighs = 10L,
  mut_sd = 0.1,
  stagnate_max = 10L
)
```

Arguments

`minimize`	(`logical(1)`) Whether to minimize the objective.
`n_searches`	(`integer(1)`) Number of local searches.

`n_steps` (integer(1))
Number of steps per local search.

`n_neighs` (integer(1))
Number of neighbors per local search.

`mut_sd` (numeric(1))
Standard deviation of the mutation.

`stagnate_max` (integer(1))
Maximum number of no-improvement steps for a local search before it is randomly restarted.

Value

(local_search_control)
List with control params as S3 object.

`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.

S3 methods

- `as.data.table(dict, ..., objects = FALSE)`
  mlr3misc::Dictionary -> `data.table::data.table()`
  Returns a `data.table::data.table()` with fields "key", "label", "param_classes", "properties" and "packages" as columns. If `objects` is set to `TRUE`, the constructed objects are returned in the list column named `object`.

See Also

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

Examples

```
as.data.table(mlr_optimizers)
mlr_optimizers$get("random_search")
opt("random_search")
```

`mlr_optimizers_async_design_points`
Asynchronous Optimization via Design Points

Description

`OptimizerAsyncDesignPoints` class that implements optimization w.r.t. fixed design points. We simply search over a set of points fully specified by the ser.

Dictionary

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

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

Parameters

`design` data.table::data.table
: Design points to try in search, one per row.

Super classes

`bbotk::Optimizer` -> `bbotk::OptimizerAsync` -> `OptimizerAsyncDesignPoints`

Methods

Public methods:

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

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

Usage:

```
OptimizerAsyncDesignPoints$new()
```

Method `optimize()`**:** Starts the asynchronous optimization.

Usage:

```
OptimizerAsyncDesignPoints$optimize(inst)
```

Arguments:

`inst` (OptimInstance).

Returns: data.table::data.table.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# example only runs if a Redis server is available
if (mlr3misc::require_namespaces(c("rush", "redux", "mirai"), quietly = TRUE) &&
  redux::redis_available()) {
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# start workers
rush::rush_plan(worker_type = "remote")
mirai::daemons(1)

# initialize instance
instance = oi_async(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)

# load optimizer
design = data.table::data.table(x1 = c(0, 1), x2 = c(0, 1))
optimizer = opt("async_design_points", design = design)
```

```
# trigger optimization
optimizer$optimize(instance)

# all evaluated configurations
instance$archive

# best performing configuration
instance$archive$best()

# covert to data.table
as.data.table(instance$archive)
}
```

`mlr_optimizers_async_grid_search`
Asynchronous Optimization via Grid Search

Description

`OptimizerAsyncGridSearch` class that implements a 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.

Dictionary

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

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

Parameters

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

Super classes

`bbotk::Optimizer` -> `bbotk::OptimizerAsync` -> `OptimizerAsyncGridSearch`

Methods

Public methods:

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

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

Usage:

```
OptimizerAsyncGridSearch$new()
```

Method `optimize()`**:** Starts the asynchronous optimization.

Usage:

```
OptimizerAsyncGridSearch$optimize(inst)
```

Arguments:

`inst` (OptimInstance).

Returns: data.table::data.table.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Source

Bergstra J, Bengio Y (2012). "Random Search for Hyper-Parameter Optimization." *Journal of Machine Learning Research*, **13**(10), 281–305. https://jmlr.csail.mit.edu/papers/v13/bergstra12a.html.

Examples

```
# example only runs if a Redis server is available
if (mlr3misc::require_namespaces(c("rush", "redux", "mirai"), quietly = TRUE) &&
  redux::redis_available()) {
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
```

```
  codomain = codomain,
  properties = "deterministic"
)

# start workers
rush::rush_plan(worker_type = "remote")
mirai::daemons(1)

# initialize instance
instance = oi_async(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)

# load optimizer
optimizer = opt("async_grid_search", resolution = 10)

# trigger optimization
optimizer$optimize(instance)

# all evaluated configurations
instance$archive

# best performing configuration
instance$archive$best()

# covert to data.table
as.data.table(instance$archive)
}
```

`mlr_optimizers_async_random_search`

Asynchronous Optimization via Random Search

Description

`OptimizerAsyncRandomSearch` class that implements a simple Random Search.

Dictionary

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

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

Super classes

`bbotk::Optimizer` -> `bbotk::OptimizerAsync` -> `OptimizerAsyncRandomSearch`

Methods

Public methods:

- `OptimizerAsyncRandomSearch$new()`
- `OptimizerAsyncRandomSearch$clone()`

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

Usage:

```
OptimizerAsyncRandomSearch$new()
```

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Source

Bergstra J, Bengio Y (2012). "Random Search for Hyper-Parameter Optimization." *Journal of Machine Learning Research*, **13**(10), 281–305. https://jmlr.csail.mit.edu/papers/v13/bergstra12a.html.

Examples

```
# example only runs if a Redis server is available
if (mlr3misc::require_namespaces(c("rush", "redux", "mirai"), quietly = TRUE) &&
  redux::redis_available()) {
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)
```

```
# start workers
rush::rush_plan(worker_type = "remote")
mirai::daemons(1)

# initialize instance
instance = oi_async(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)

# load optimizer
optimizer = opt("async_random_search")

# trigger optimization
optimizer$optimize(instance)

# all evaluated configurations
instance$archive

# best performing configuration
instance$archive$best()

# covert to data.table
as.data.table(instance$archive)
}
```

`mlr_optimizers_chain` *Run Optimizers Sequentially*

Description

`OptimizerBatchChain` allows to run multiple OptimizerBatch sequentially.

For each OptimizerBatch an (optional) additional Terminator can be specified during construction. While the original Terminator of the OptimInstanceBatch guards the optimization process as a whole, the additional Terminators guard each individual OptimizerBatch.

The optimization process works as follows: The first OptimizerBatch is run on the OptimInstanceBatch relying on a TerminatorCombo of the original Terminator of the OptimInstanceBatch and the (optional) additional Terminator as passed during construction. Once this TerminatorCombo indicates termination (usually via the additional Terminator), the second OptimizerBatch is run. This continues for all optimizers unless the original Terminator of the OptimInstanceBatch indicates termination.

OptimizerBatchChain can also be used for random restarts of the same Optimizer (if applicable) by setting the Terminator of the OptimInstanceBatch to TerminatorNone and setting identical additional Terminators during construction.

Dictionary

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

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

Parameters

Parameters are inherited from the individual OptimizerBatch and collected as a paradox::ParamSetCollection (with `set_ids` potentially postfixed via `_1`, `_2`, ..., if the same OptimizerBatch are used multiple times).

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

`bbotk::Optimizer` -> `bbotk::OptimizerBatch` -> `OptimizerBatchChain`

Methods

Public methods:

- `OptimizerBatchChain$new()`
- `OptimizerBatchChain$clone()`

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

Usage:

```
OptimizerBatchChain$new(
  optimizers,
  terminators = rep(list(NULL), length(optimizers))
)
```

Arguments:

`optimizers` (list of Optimizers).

`terminators` (list of Terminators | NULL).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# example only runs if GenSA is available
if (mlr3misc::require_namespaces("GenSA", quietly = TRUE)) {
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# initialize instance
instance = oi(
  objective = objective,
  terminator = trm("evals", n_evals = 10)
)

# load optimizer
optimizer = opt("chain",
  optimizers = list(opt("random_search"), opt("grid_search")),
  terminators = list(trm("evals", n_evals = 5), trm("evals", n_evals = 5))
)

# trigger optimization
optimizer$optimize(instance)

# all evaluated configurations
instance$archive

# best performing configuration
instance$result
}
```

`mlr_optimizers_cmaes` *Optimization via Covariance Matrix Adaptation Evolution Strategy*

Description

OptimizerBatchCmaes class that implements CMA-ES. Calls adagio::pureCMAES() from package **adagio**. The algorithm is typically applied to search space dimensions between three and fifty. Lower search space dimensions might crash.

Dictionary

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

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

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. If set to "custom", the start values can be passed via the start parameter.

start numeric()
: Custom start values. Only applicable if start_values parameter is set to "custom".

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

$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

bbotk::Optimizer -> bbotk::OptimizerBatch -> OptimizerBatchCmaes

Methods

Public methods:

- OptimizerBatchCmaes$new()
- OptimizerBatchCmaes$clone()

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

Usage:

```
OptimizerBatchCmaes$new()
```

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
# example only runs if GenSA is available
if (mlr3misc::require_namespaces("adagio", quietly = TRUE)) {
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 - (xs[[3]] + 4)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5),
  x3 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# initialize instance
instance = oi(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)

# load optimizer
optimizer = opt("cmaes")

# trigger optimization
optimizer$optimize(instance)

# all evaluated configurations
instance$archive

# best performing configuration
instance$result
}
```

`mlr_optimizers_design_points` *Optimization via Design Points*

Description

`OptimizerBatchDesignPoints` 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.

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

`bbotk::Optimizer` -> `bbotk::OptimizerBatch` -> `OptimizerBatchDesignPoints`

Methods

Public methods:

- `OptimizerBatchDesignPoints$new()`
- `OptimizerBatchDesignPoints$clone()`

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

Usage:

```
OptimizerBatchDesignPoints$new()
```

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# initialize instance
instance = oi(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)

# load optimizer
design = data.table::data.table(x1 = c(0, 1), x2 = c(0, 1))
optimizer = opt("design_points", design = design)

# trigger optimization
optimizer$optimize(instance)

# all evaluated configurations
instance$archive

# best performing configuration
instance$result
```

`mlr_optimizers_focus_search` *Optimization via Focus Search*

Description

`OptimizerBatchFocusSearch` class that implements a Focus Search.

Focus Search starts with evaluating `n_points` drawn uniformly at random. For 1 to `maxit` batches, `n_points` are then drawn uniformly at random and if the best value of a batch outperforms the previous best value over all batches evaluated so far, the search space is shrinked around this new best point prior to the next batch being sampled and evaluated.

For details on the shrinking, see shrink_ps.

Depending on the Terminator this procedure simply restarts after `maxit` is reached.

Dictionary

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

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

Parameters

`n_points integer(1)`
: Number of points to evaluate in each random search batch.

`maxit integer(1)`
: Number of random search batches to run.

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

`bbotk::Optimizer` -> `bbotk::OptimizerBatch` -> `OptimizerBatchFocusSearch`

Methods

Public methods:

- `OptimizerBatchFocusSearch$new()`
- `OptimizerBatchFocusSearch$clone()`

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

Usage:

```
OptimizerBatchFocusSearch$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# initialize instance
instance = oi(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)

# load optimizer
optimizer = opt("focus_search", n_points = 10, maxit = 10)

# trigger optimization
optimizer$optimize(instance)

# all evaluated configurations
instance$archive

# best performing configuration
instance$result
```

`mlr_optimizers_gensa` *Generalized Simulated Annealing*

Description

`OptimizerBatchGenSA` 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

`par numeric()`
: Initial parameter values. Default is `NULL`, in which case, default values will be generated automatically.

`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. By default, `nloptr` will generate start values automatically. Custom start values can be passed via the `par` parameter.

For the meaning of the control parameters, see `GenSA::GenSA()`. Note that `GenSA::GenSA()` uses `smooth = TRUE` as a default. In the case of using this optimizer for Hyperparameter Optimization you may want to set `smooth = FALSE`.

Internal Termination Parameters

The algorithm can terminated with all Terminators. Additionally, the following internal termination parameters can be used:

`maxit integer(1)`
: Maximum number of iterations. Original default is `5000`. Overwritten with `.Machine$integer.max`.

`threshold.stop numeric(1)`
: Threshold stop. Deactivated with `NULL`. Default is `NULL`.

`nb.stop.improvement integer(1)`
: Number of stop improvement. Deactivated with `-1L`. Default is `-1L`.

`max.call integer(1)`
: Maximum number of calls. Original default is `1e7`. Overwritten with `.Machine$integer.max`.

`max.time integer(1)`
: Maximum time. Deactivate with `NULL`. Default is `NULL`.

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

`bbotk::Optimizer` -> `bbotk::OptimizerBatch` -> `OptimizerBatchGenSA`

Methods

Public methods:

- `OptimizerBatchGenSA$new()`
- `OptimizerBatchGenSA$clone()`

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

Usage:

```
OptimizerBatchGenSA$new()
```

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

Usage:

```
OptimizerBatchGenSA$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

```
# example only runs if GenSA is available
if (mlr3misc::require_namespaces("GenSA", quietly = TRUE)) {
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
```

```
  codomain = ps(
    y = p_dbl(tags = "maximize")
  )

  # create objective
  objective = ObjectiveRFun$new(
    fun = fun,
    domain = domain,
    codomain = codomain,
    properties = "deterministic"
  )

  # initialize instance
  instance = oi(
    objective = objective,
    terminator = trm("evals", n_evals = 20)
  )

  # load optimizer
  optimizer = opt("gensa")

  # trigger optimization
  optimizer$optimize(instance)

  # all evaluated configurations
  instance$archive

  # best performing configuration
  instance$result
  }
```

`mlr_optimizers_grid_search`

Optimization via Grid Search

Description

`OptimizerBatchGridSearch` 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.

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

`bbotk::Optimizer` -> `bbotk::OptimizerBatch` -> `OptimizerBatchGridSearch`

Methods

Public methods:

- `OptimizerBatchGridSearch$new()`
- `OptimizerBatchGridSearch$clone()`

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

Usage:

```
OptimizerBatchGridSearch$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
```

```
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# initialize instance
instance = oi(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)

# load optimizer
optimizer = opt("grid_search", resolution = 10)

# trigger optimization
optimizer$optimize(instance)

# all evaluated configurations
instance$archive

# best performing configuration
instance$result
```

`mlr_optimizers_irace` *Iterated Racing*

Description

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

Parameters

`instances list()`
: A list of instances where the configurations executed on.

`targetRunnerParallel function()`
: A function that executes the objective function with a specific parameter configuration and instance. A default function is provided, see section "Target Runner and Instances".

For the meaning of all other parameters, see `irace::defaultScenario()`.

Internal Termination Parameters

The algorithm can terminated with TerminatorEvals. Other Terminators do not work with `OptimizerBatchIrace`. Additionally, the following internal termination parameters can be used:

`maxExperiments integer(1)`
: Maximum number of runs (invocations of targetRunner) that will be performed. It determines the maximum budget of experiments for the tuning. Default is 0.

`minExperiments integer(1)`
: Minimum number of runs (invocations of targetRunner) that will be performed. It determines the minimum budget of experiments for the tuning. The actual budget depends on the number of parameters and minSurvival. Default is NA.

`maxTime integer(1)`
: Maximum total execution time for the executions of targetRunner. targetRunner must return two values: cost and time. This value and the one returned by targetRunner must use the same units (seconds, minutes, iterations, evaluations, ...). Default is 0.

`budgetEstimation numeric(1)`
: Fraction (smaller than 1) of the budget used to estimate the mean computation time of a configuration. Only used when maxTime > 0 Default is 0.05.

`minMeasurableTime numeric(1)`
: Minimum time unit that is still (significantly) measureable. Default is 0.01.

Initial parameter values

- `digits`:
  - Adjusted default: 15.
  - This represents double parameters with a higher precision and avoids rounding errors.

Target Runner and Instances

The irace package uses a `targetRunner` script or R function to evaluate a configuration on a particular instance. Usually it is not necessary to specify a `targetRunner` function when using `OptimizerBatchIrace`. A default function is used that forwards several configurations and instances to the user defined objective function. As usually, the user defined function has a `xs`, `xss` or `xdt` parameter depending on the used Objective class. For irace, the function needs an additional `instances` parameter.

```
fun = function(xs, instances) {
 # function to evaluate configuration in `xs` on instance `instances`
}
```

Archive

The Archive holds the following additional columns:

- `"race"` `(integer(1))`
  Race iteration.

- "step" (integer(1))
  Step number of race.
- "instance" (integer(1))
  Identifies instances across races and steps.
- "configuration" (integer(1))
  Identifies configurations across races and steps.

Result

The optimization result (instance$result) is the best performing elite of the final race. The reported performance is the average performance estimated on all used instances.

Dictionary

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

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

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

bbotk::Optimizer -> bbotk::OptimizerBatch -> OptimizerBatchIrace

Methods

Public methods:

- OptimizerBatchIrace$new()
- OptimizerBatchIrace$clone()

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

Usage:

```
OptimizerBatchIrace$new()
```

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Source

Lopez-Ibanez M, Dubois-Lacoste J, Caceres LP, Birattari M, Stuetzle T (2016). "The irace package: Iterated racing for automatic algorithm configuration." *Operations Research Perspectives*, **3**, 43–58. doi:10.1016/j.orp.2016.09.002.

Examples

```
# example only runs if irace is available
if (mlr3misc::require_namespaces("irace", quietly = TRUE)) {
# runtime of the example is too long

library(data.table)

# set domain
domain = ps(
  x1 = p_dbl(-5, 10),
  x2 = p_dbl(0, 15)
)

# set codomain
codomain = ps(y = p_dbl(tags = "minimize"))

# branin function with noise
# the noise generates different instances of the branin function
# the noise values are passed via the `instances` parameter
fun = function(xdt, instances) {
  ys = branin(xdt[["x1"]], xdt[["x2"]], noise = as.numeric(instances))
  data.table(y = ys)
}

# define objective with instances as a constant
objective = ObjectiveRFunDt$new(
 fun = fun,
 domain = domain,
 codomain = codomain,
 constants = ps(instances = p_uty()))

instance = oi(
  objective = objective,
  terminator = trm("evals", n_evals = 96))

# create instances of branin function
instances = rnorm(10, mean = 0, sd = 0.1)

# load optimizer and set branin instances
optimizer = opt("irace", instances = instances)

# trigger optimization
optimizer$optimize(instance)

# all evaluated configurations
instance$archive
```

```
# best performing configuration
instance$result

}
```

`mlr_optimizers_local_search` *Local Search*

Description

Implements a simple Local Search, see `local_search()` for details. Currently, setting initial points is not supported.

Dictionary

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

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

Parameters

The same as for `local_search_control()`, with the same defaults (except for `minimize`).

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

`bbotk::Optimizer` -> `bbotk::OptimizerBatch` -> `OptimizerBatchLocalSearch`

Methods

Public methods:

- `OptimizerBatchLocalSearch$new()`
- `OptimizerBatchLocalSearch$clone()`

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

Usage:

```
OptimizerBatchLocalSearch$new()
```

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# initialize instance
instance = oi(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)

# load optimizer
optimizer = opt("local_search")

# trigger optimization
optimizer$optimize(instance)

# all evaluated configurations
instance$archive

# best performing configuration
instance$result
```

`mlr_optimizers_nloptr` *Non-linear Optimization*

Description

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

Parameters

`algorithm character(1)`
: Algorithm to use. See `nloptr::nloptr.print.options()` for available algorithms.

`x0 numeric()`
: Initial parameter values. Use `start_values` parameter to create `"random"` or `"center"` start values.

`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. Custom start values can be passed via the `x0` parameter.

`approximate_eval_grad_f logical(1)`
: Should gradients be numerically approximated via finite differences (nloptr::nl.grad). Only required for certain algorithms. Note that function evaluations required for the numerical gradient approximation will be logged as usual and are not treated differently than regular function evaluations by, e.g., Terminators.

For the meaning of other control parameters, see `nloptr::nloptr()` and `nloptr::nloptr.print.options()`.

Internal Termination Parameters

The algorithm can terminated with all Terminators. Additionally, the following internal termination parameters can be used:

`stopval numeric(1)`
: Stop value. Deactivate with `-Inf`. Default is `-Inf`.

`maxtime integer(1)`
: Maximum time. Deactivate with `-1L`. Default is `-1L`.

`maxeval integer(1)`
: Maximum number of evaluations. Deactivate with `-1L`. Default is `-1L`.

`xtol_rel numeric(1)`
: Relative tolerance. Original default is 10^-4. Deactivate with `-1`. Overwritten with `-1`.

`xtol_abs numeric(1)`
: Absolute tolerance. Deactivate with `-1`. Default is `-1`.

`ftol_rel numeric(1)`
: Relative tolerance. Deactivate with `-1`. Default is `-1`.

`ftol_abs numeric(1)`
: Absolute tolerance. Deactivate with `-1`. Default is `-1`.

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

`bbotk::Optimizer` -> `bbotk::OptimizerBatch` -> `OptimizerBatchNLoptr`

Methods

Public methods:

- `OptimizerBatchNLoptr$new()`
- `OptimizerBatchNLoptr$clone()`

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

Usage:

```
OptimizerBatchNLoptr$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Source

Johnson, G S (2020). "The NLopt nonlinear-optimization package." https://github.com/stevengj/nlopt.

Examples

```
# example only runs if nloptr is available
if (mlr3misc::require_namespaces("nloptr", quietly = TRUE)) {
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)
```

```
# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# initialize instance
instance = oi(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)

# load optimizer
optimizer = opt("nloptr", algorithm = "NLOPT_LN_BOBYQA")

# trigger optimization
optimizer$optimize(instance)

# all evaluated configurations
instance$archive

# best performing configuration
instance$result
}
```

`mlr_optimizers_random_search`

Optimization via Random Search

Description

`OptimizerBatchRandomSearch` 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.

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

`bbotk::Optimizer` -> `bbotk::OptimizerBatch` -> `OptimizerBatchRandomSearch`

Methods

Public methods:

- `OptimizerBatchRandomSearch$new()`
- `OptimizerBatchRandomSearch$clone()`

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

Usage:

```
OptimizerBatchRandomSearch$new()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Source

Bergstra J, Bengio Y (2012). "Random Search for Hyper-Parameter Optimization." *Journal of Machine Learning Research*, **13**(10), 281–305. https://jmlr.csail.mit.edu/papers/v13/bergstra12a.html.

Examples

```
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)
```

```
# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# initialize instance
instance = oi(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)

# load optimizer
optimizer = opt("random_search", batch_size = 10)

# trigger optimization
optimizer$optimize(instance)

# all evaluated configurations
instance$archive

# best performing configuration
instance$result
```

`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.

S3 methods

- `as.data.table(dict, ..., objects = FALSE)`
  mlr3misc::Dictionary -> `data.table::data.table()`
  Returns a `data.table::data.table()` with fields "key", "label", "properties" and "unit" as columns. If `objects` is set to `TRUE`, the constructed objects are returned in the list column named `object`.

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_batch`, `mlr_terminators_stagnation_hypervolume`

Examples

```
as.data.table(mlr_terminators)
mlr_terminators$get("evals")
trm("evals", n_evals = 10)
```

---

mlr_terminators_clock_time

Clock Time Terminator

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`, `mlr_terminators_combo`, `mlr_terminators_evals`, `mlr_terminators_none`, `mlr_terminators_perf_reached`, `mlr_terminators_run_time`, `mlr_terminators_stagnatio` `mlr_terminators_stagnation_batch`, `mlr_terminators_stagnation_hypervolume`

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$print()`
- `TerminatorCombo$remaining_time()`
- `TerminatorCombo$status_long()`
- `TerminatorCombo$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 `remaining_time()`: Returns the remaining runtime in seconds. If `any = TRUE`, the remaining runtime is determined by the time-based terminator with the shortest time remaining. If non-time-based terminators are used and `any = FALSE`, the the remaining runtime is always `Inf`.

Usage:

```
TerminatorCombo$remaining_time(archive)
```

Arguments:

`archive` (Archive).

Returns: `integer(1)`.

Method `status_long()`: Returns `max_steps` and `current_steps` for each terminator.

Usage:

```
TerminatorCombo$status_long(archive)
```

Arguments:

`archive` (Archive).

Returns: data.table::data.table.

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`, `mlr_terminators_clock_time`, `mlr_terminators_evals`, `mlr_terminators_none`, `mlr_terminators_perf_reached`, `mlr_terminators_run_time`, `mlr_terminators_stagnatio`
`mlr_terminators_stagnation_batch`, `mlr_terminators_stagnation_hypervolume`

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. The total number of evaluations B is defined as

$$B = \texttt{n\_evals} + \texttt{k} * D$$

where D is the dimension of the search space.

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)`
: See formula above. Default is 100.

`k integer(1)`
: See formula above. Default is 0.

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, mlr_terminators_clock_time, mlr_terminators_combo, mlr_terminators_none, mlr_terminators_perf_reached, mlr_terminators_run_time, mlr_terminators_stagnatio mlr_terminators_stagnation_batch, mlr_terminators_stagnation_hypervolume

Examples

```
TerminatorEvals$new()

# 5 evaluations in total
trm("evals", n_evals = 5)

# 3 * [dimension of search space] evaluations in total
trm("evals", n_evals = 0, k = 3)

# (3 * [dimension of search space] + 1) evaluations in total
trm("evals", n_evals = 1, k = 3)
```

---

mlr_terminators_none *None Terminator*

Description

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

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`, `mlr_terminators_clock_time`, `mlr_terminators_combo`, `mlr_terminators_evals`, `mlr_terminators_perf_reached`, `mlr_terminators_run_time`, `mlr_terminators_stagnati`
`mlr_terminators_stagnation_batch`, `mlr_terminators_stagnation_hypervolume`

`mlr_terminators_perf_reached` *Performance Level Terminator*

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`, `mlr_terminators_clock_time`, `mlr_terminators_combo`, `mlr_terminators_evals`, `mlr_terminators_none`, `mlr_terminators_run_time`, `mlr_terminators_stagnation`, `mlr_terminators_stagnation_batch`, `mlr_terminators_stagnation_hypervolume`

Examples

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

---

mlr_terminators_run_time

Run Time Terminator

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.

Note

This terminator only works if `archive$start_time` is set. This is usually done by the Optimizer.

See Also

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

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`, `mlr_terminators_clock_time`, `mlr_terminators_combo`, `mlr_terminators_evals`, `mlr_terminators_none`, `mlr_terminators_perf_reached`, `mlr_terminators_run_time`, `mlr_terminators_stagnation_batch`, `mlr_terminators_stagnation_hypervolume`

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_batch")
trm("stagnation_batch")
```

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`, `mlr_terminators_clock_time`, `mlr_terminators_combo`, `mlr_terminators_evals`, `mlr_terminators_none`, `mlr_terminators_perf_reached`, `mlr_terminators_run_time`, `mlr_terminators_stagnation`, `mlr_terminators_stagnation_hypervolume`

Examples

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

`mlr_terminators_stagnation_hypervolume`
Stagnation Hypervolume Terminator

Description

Class to terminate the optimization after the hypervolume 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_hypervolume")
trm("stagnation_hypervolume")
```

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` -> `TerminatorStagnationHypervolume`

Methods

Public methods:

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

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

Usage:

```
TerminatorStagnationHypervolume$new()
```

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

Usage:

```
TerminatorStagnationHypervolume$is_terminated(archive)
```

Arguments:

`archive` (Archive).

Returns: `logical(1)`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

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

Examples

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

`Objective` *Objective Function with Domain and Codomain*

Description

The `Objective` class describes a black-box objective function that maps an arbitrary domain to a numerical codomain.

Details

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

Public fields

`callbacks` (list of mlr3misc::Callback)
: Callbacks applied during the optimization.

`context` (ContextBatch)
: Stores the context for the callbacks.

`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.

`constants` (paradox::ParamSet).
: Changeable constants or parameters that are not subject to tuning can be stored and accessed here. Set constant values are passed to `$.eval()` and `$.eval_many()` as named arguments.

`check_values` (`logical(1)`)

Active bindings

`label (character(1))`
: Label for this object. Can be used in tables, plot and text output instead of the ID.

`man (character(1))`
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

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

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

`packages (character())`
: Set of required packages to run the objective function. Packages are loaded on each worker when the objective is called by OptimizerAsync.

Methods

Public methods:

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

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

Usage:

```
Objective$new(
  id = "f",
  properties = character(),
  domain,
  codomain = ps(y = p_dbl(tags = "minimize")),
  constants = ps(),
  packages = character(),
  check_values = TRUE,
  label = NA_character_,
  man = NA_character_
)
```

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.

`constants` (paradox::ParamSet)
Changeable constants or parameters that are not subject to tuning can be stored and accessed here.

`packages` (`character()`)
Set of required packages to run the objective function.

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

`label` (`character(1)`)
Label for this object. Can be used in tables, plot and text output instead of the ID.

`man` (`character(1)`)
String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

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

Usage:

```
Objective$format(...)
```

Arguments:

`...` (ignored).

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 `help()`**:** Opens the corresponding help page referenced by field `$man`.

Usage:

```
Objective$help()
```

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.

See Also

ObjectiveRFun, ObjectiveRFunMany, ObjectiveRFunDt

`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. If the return of the function is unnamed, it is named with the ids of the codomain.

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(),
  constants = ps(),
  packages = character(),
  check_values = TRUE
)
```

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()).

constants (paradox::ParamSet)
: Changeable constants or parameters that are not subject to tuning can be stored and accessed here.

`packages (character())`
: Set of required packages to run the objective function.

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

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.

See Also

ObjectiveRFunMany, ObjectiveRFunDt

Examples

```
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# evaluate objective function
```

```
objective$eval(list(x1 = 1, x2 = 2))

# evaluate multiple input values
objective$eval_many(list(list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4)))

# evaluate multiple input values as data.table
objective$eval_dt(data.table::data.table(x1 = 1:2, x2 = 3:4))
```

`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(),
  constants = ps(),
  packages = character(),
  check_values = TRUE
)
```

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()`).

`constants` (paradox::ParamSet)
: Changeable constants or parameters that are not subject to tuning can be stored and accessed here.

`packages` (`character()`)
: Set of required packages to run the objective function.

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

Method `eval_many()`**:** Evaluates multiple input values received as a list, converted to a `data.table()` on the objective function. Missing columns in xss are filled with `NA`s in `xdt`.

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.

See Also

ObjectiveRFun, ObjectiveRFunMany

Examples

```
# define objective function
fun = function(xdt) {
  data.table::data.table(y = xdt$x1 + xdt$x2)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFunDt$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# evaluate objective function
objective$eval(list(x1 = 1, x2 = 2))

# evaluate multiple input values
objective$eval_many(list(list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4)))

# evaluate multiple input values as data.table
objective$eval_dt(data.table::data.table(x1 = 1:2, x2 = 3:4))
```

`ObjectiveRFunMany` *Objective Interface with Custom R Function*

Description

Objective interface where the user can pass a custom R function that expects a list of configurations as input. If the return of the function is unnamed, it is named with the ids of the codomain.

Super class

`bbotk::Objective` -> `ObjectiveRFunMany`

Active bindings

`fun` (`function`)
Objective function.

Methods

Public methods:

- `ObjectiveRFunMany$new()`
- `ObjectiveRFunMany$eval_many()`
- `ObjectiveRFunMany$clone()`

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

Usage:

```
ObjectiveRFunMany$new(
  fun,
  domain,
  codomain = NULL,
  id = "function",
  properties = character(),
  constants = ps(),
  packages = character(),
  check_values = TRUE
)
```

Arguments:

`fun` (`function`)
R function that encodes objective and expects a list of lists that contains multiple x values, e.g. `list(list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4))`. The function must return a `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)`.

`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()`).

`constants` (paradox::ParamSet)
Changeable constants or parameters that are not subject to tuning can be stored and accessed here.

`packages` (`character()`)
Set of required packages to run the objective function.

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

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

Usage:

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

ObjectiveRFun, ObjectiveRFunDt

Examples

```
# define objective function
fun = function(xss) {
  res = lapply(xss, function(xs) -(xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
  data.table::data.table(y = as.numeric(res))
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)
```

```
# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFunMany$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# evaluate objective function
objective$eval(list(x1 = 1, x2 = 2))

# evaluate multiple input values
objective$eval_many(list(list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4)))

# evaluate multiple input values as data.table
objective$eval_dt(data.table::data.table(x1 = 1:2, x2 = 3:4))
```

`oi` *Syntactic Sugar for Optimization Instance Construction*

Description

Function to construct a OptimInstanceBatchSingleCrit and OptimInstanceBatchMultiCrit.

Usage

```
oi(
  objective,
  search_space = NULL,
  terminator,
  callbacks = NULL,
  check_values = TRUE,
  keep_evals = "all"
)
```

Arguments

`objective`	(Objective) Objective function.
`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 Termination criterion.
callbacks	(list of mlr3misc::Callback) List of callbacks.
check_values	(logical(1)) Should points before the evaluation and the results be checked for validity?
keep_evals	(character(1)) Keep all or only best evaluations in archive?

Examples

```
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# initialize instance
instance = oi(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)
```

---

oi_async *Syntactic Sugar for Asynchronous Optimization Instance Construction*

Description

Function to construct an OptimInstanceAsyncSingleCrit and OptimInstanceAsyncMultiCrit.

Usage

```
oi_async(
  objective,
  search_space = NULL,
  terminator,
  check_values = FALSE,
  callbacks = NULL,
  rush = NULL
)
```

Arguments

objective	(Objective) Objective function.
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 Termination criterion.
check_values	(logical(1)) Should points before the evaluation and the results be checked for validity?
callbacks	(list of mlr3misc::Callback) List of callbacks.
rush	(Rush) If a rush instance is supplied, the tuning runs without batches.

Examples

```
# example only runs if a Redis server is available
if (mlr3misc::require_namespaces(c("rush", "redux", "mirai"), quietly = TRUE) &&
  redux::redis_available()) {
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
```

```
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# start workers
rush::rush_plan(worker_type = "remote")
mirai::daemons(1)

# initialize instance
instance = oi_async(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)
}
```

`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*

Description

The `OptimInstance` specifies an optimization problem for an Optimizer.

Details

`OptimInstance` is an abstract base class that implements the base functionality each instance must provide. 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)
: Objective function of the instance.

`search_space` (paradox::ParamSet)
: Specification of the search space for the Optimizer.

`terminator` Terminator
: Termination criterion of the optimization.

`archive` (Archive)
: Contains all performed function calls of the Objective.

`progressor` (`progressor()`)
: Stores `progressor` function.

Active bindings

`label` (`character(1)`)
: Label for this object. Can be used in tables, plot and text output instead of the ID.

`man` (`character(1)`)
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

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

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

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

Methods

Public methods:

- `OptimInstance$new()`
- `OptimInstance$format()`
- `OptimInstance$print()`
- `OptimInstance$assign_result()`
- `OptimInstance$clear()`
- `OptimInstance$clone()`

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

Usage:

```
OptimInstance$new(
  objective,
  search_space = NULL,
  terminator,
  check_values = TRUE,
  callbacks = NULL,
  archive = NULL,
  label = NA_character_,
  man = NA_character_
)
```

Arguments:

`objective` (Objective)
: Objective function.

`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
: Termination criterion.

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

`callbacks` (list of mlr3misc::Callback)
: List of callbacks.

`archive` (Archive).

`label` (`character(1)`)
: Label for this object. Can be used in tables, plot and text output instead of the ID.

`man` (`character(1)`)
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

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

Usage:

```
OptimInstance$format(...)
```

Arguments:

`...` (ignored).

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

Usage:

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

Arguments:

`...` (ignored).

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::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.

`...` (`any`)
ignored.

Method `clear()`**:** Reset terminator and clear all evaluation results from archive and results.

Usage:

```
OptimInstance$clear()
```

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.

See Also

OptimInstanceBatch, OptimInstanceAsync

`OptimInstanceAsync` *Optimization Instance for Asynchronous Optimization*

Description

The `OptimInstanceAsync` specifies an optimization problem for an OptimizerAsync. The function `oi_async()` creates an OptimInstanceAsyncSingleCrit or OptimInstanceAsyncMultiCrit.

Details

`OptimInstanceAsync` is an abstract base class that implements the base functionality each instance must provide.

Super class

`bbotk::OptimInstance` -> `OptimInstanceAsync`

Public fields

`rush` (`Rush`)
Rush controller for parallel optimization.

Methods

Public methods:

- `OptimInstanceAsync$new()`
- `OptimInstanceAsync$print()`
- `OptimInstanceAsync$clear()`
- `OptimInstanceAsync$reconnect()`
- `OptimInstanceAsync$clone()`

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

Usage:

```
OptimInstanceAsync$new(
  objective,
  search_space = NULL,
  terminator,
  check_values = FALSE,
  callbacks = NULL,
  archive = NULL,
  rush = NULL,
  label = NA_character_,
  man = NA_character_
)
```

Arguments:

`objective` (Objective)
: Objective function.

`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
: Termination criterion.

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

`callbacks` (list of mlr3misc::Callback)
: List of callbacks.

`archive` (Archive).

`rush` (`Rush`)
: If a rush instance is supplied, the tuning runs without batches.

`label` (`character(1)`)
: Label for this object. Can be used in tables, plot and text output instead of the ID.

`man` (`character(1)`)
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

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

Usage:

```
OptimInstanceAsync$print(...)
```

Arguments:

`...` (ignored).

Method `clear()`**:** Reset terminator and clear all evaluation results from archive and results.

Usage:

```
OptimInstanceAsync$clear()
```

Method `reconnect()`**:** Reconnect to Redis. The connection breaks when the rush::Rush is saved to disk. Call this method to reconnect after loading the object.

Usage:

```
OptimInstanceAsync$reconnect()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

`oi_async()`, OptimInstanceAsyncSingleCrit, OptimInstanceAsyncMultiCrit

`OptimInstanceAsyncMultiCrit` *Multi Criteria Optimization Instance for Asynchronous Optimization*

Description

The OptimInstanceAsyncMultiCrit specifies an optimization problem for an OptimizerAsync. The function `oi_async()` creates an OptimInstanceAsyncMultiCrit.

Super classes

`bbotk::OptimInstance` -> `bbotk::OptimInstanceAsync` -> `OptimInstanceAsyncMultiCrit`

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:

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

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

Usage:

```
OptimInstanceAsyncMultiCrit$new(
  objective,
  search_space = NULL,
  terminator,
  check_values = FALSE,
  callbacks = NULL,
  archive = NULL,
  rush = NULL
)
```

Arguments:

`objective` (Objective)
Objective function.

`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
Termination criterion.

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

`callbacks` (list of mlr3misc::Callback)
List of callbacks.

`archive` (Archive).

`rush` (`Rush`)
If a rush instance is supplied, the tuning runs without batches.

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

Usage:

```
OptimInstanceAsyncMultiCrit$assign_result(xdt, ydt, extra = NULL, ...)
```

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.

`extra` (`data.table::data.table()`)
Additional information.

`...` (any)
ignored.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# example only runs if a Redis server is available
if (mlr3misc::require_namespaces(c("rush", "redux", "mirai"), quietly = TRUE) &&
  redux::redis_available()) {
# define the objective function
fun = function(xs) {
  data.table(
    y1 = xs$x1^2 +  xs$x2^2,
    y2 = (xs$x1 - 2)^2 + (xs$x2 - 1)^2
  )
}

# set domain
domain = ps(
```

```
  x1 = p_dbl(-5, 5),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y1 = p_dbl(tags = "minimize"),
  y2 = p_dbl(tags = "minimize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# start workers
rush::rush_plan(worker_type = "remote")
mirai::daemons(1)

# initialize instance
instance = oi_async(
  objective = objective,
  terminator = trm("evals", n_evals = 100))
}
```

`OptimInstanceAsyncSingleCrit` *Single Criterion Optimization Instance for Asynchronous Optimization*

Description

The `OptimInstanceAsyncSingleCrit` specifies an optimization problem for an OptimizerAsync. The function `oi_async()` creates an OptimInstanceAsyncSingleCrit.

Super classes

`bbotk::OptimInstance` -> `bbotk::OptimInstanceAsync` -> `OptimInstanceAsyncSingleCrit`

Active bindings

`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:

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

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

Usage:

```
OptimInstanceAsyncSingleCrit$new(
  objective,
  search_space = NULL,
  terminator,
  check_values = FALSE,
  callbacks = NULL,
  archive = NULL,
  rush = NULL
)
```

Arguments:

`objective` (Objective)
: Objective function.

`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
: Termination criterion.

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

`callbacks` (list of mlr3misc::Callback)
: List of callbacks.

`archive` (Archive).

`rush` (`Rush`)
: If a rush instance is supplied, the tuning runs without batches.

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

Usage:

```
OptimInstanceAsyncSingleCrit$assign_result(xdt, y, extra = NULL, ...)
```

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.

`extra` (`data.table::data.table()`)
Additional information.

`...` (any)
ignored.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# example only runs if a Redis server is available
if (mlr3misc::require_namespaces(c("rush", "redux", "mirai"), quietly = TRUE) &&
  redux::redis_available()) {
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# start workers
rush::rush_plan(worker_type = "remote")
mirai::daemons(1)

# initialize instance
instance = oi_async(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)
```

```
}
```

`OptimInstanceBatch` *Optimization Instance for Batch Optimization*

Description

The `OptimInstanceBatch` specifies an optimization problem for an OptimizerBatch. The function `oi()` creates an OptimInstanceAsyncSingleCrit or OptimInstanceAsyncMultiCrit.

Super class

`bbotk::OptimInstance` -> `OptimInstanceBatch`

Public fields

`objective_multiplicator (integer())`.

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.

`is_terminated (logical(1))`.

Methods

Public methods:

- `OptimInstanceBatch$new()`
- `OptimInstanceBatch$eval_batch()`
- `OptimInstanceBatch$objective_function()`
- `OptimInstanceBatch$clone()`

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

Usage:

```
OptimInstanceBatch$new(
  objective,
  search_space = NULL,
  terminator,
  check_values = TRUE,
```

```
  callbacks = NULL,
  archive = NULL,
  label = NA_character_,
  man = NA_character_
)
```

Arguments:

`objective` (Objective)
: Objective function.

`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
: Termination criterion.

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

`callbacks` (list of mlr3misc::Callback)
: List of callbacks.

`archive` (Archive).

`label` (`character(1)`)
: Label for this object. Can be used in tables, plot and text output instead of the ID.

`man` (`character(1)`)
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

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:

```
OptimInstanceBatch$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 `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:

```
OptimInstanceBatch$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:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

`oi()`, OptimInstanceBatchSingleCrit, OptimInstanceBatchMultiCrit

---

`OptimInstanceBatchMultiCrit` *Multi Criteria Optimization Instance for Batch Optimization*

Description

The OptimInstanceBatchMultiCrit specifies an optimization problem for an OptimizerBatch. The function `oi()` creates an OptimInstanceBatchMultiCrit.

Super classes

`bbotk::OptimInstance` -> `bbotk::OptimInstanceBatch` -> `OptimInstanceBatchMultiCrit`

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:

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

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

Usage:

```
OptimInstanceBatchMultiCrit$new(
  objective,
  search_space = NULL,
  terminator,
  check_values = TRUE,
  callbacks = NULL,
  archive = NULL
)
```

Arguments:

`objective` (Objective)
: Objective function.

`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
: Termination criterion.

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

`callbacks` (list of mlr3misc::Callback)
: List of callbacks.

`archive` (Archive).

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:

```
OptimInstanceBatchMultiCrit$assign_result(xdt, ydt, extra = NULL, ...)
```

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

`extra` (`data.table::data.table()`)
: Additional information.

`...` (`any`)
: ignored.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# define the objective function
fun = function(xs) {
  data.table(
    y1 = xs$x1^2 +  xs$x2^2,
    y2 = (xs$x1 - 2)^2 + (xs$x2 - 1)^2
  )
}

# set domain
domain = ps(
  x1 = p_dbl(-5, 5),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y1 = p_dbl(tags = "minimize"),
  y2 = p_dbl(tags = "minimize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# initialize instance
instance = oi(
  objective = objective,
  terminator = trm("evals", n_evals = 100))
```

OptimInstanceBatchSingleCrit

Single Criterion Optimization Instance for Batch Optimization

---

Description

The OptimInstanceBatchSingleCrit specifies an optimization problem for an OptimizerBatch. The function `oi()` creates an OptimInstanceBatchSingleCrit.

Super classes

`bbotk::OptimInstance` -> `bbotk::OptimInstanceBatch` -> `OptimInstanceBatchSingleCrit`

Methods

Public methods:

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

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

Usage:

```
OptimInstanceBatchSingleCrit$new(
  objective,
  search_space = NULL,
  terminator,
  check_values = TRUE,
  callbacks = NULL,
  archive = NULL
)
```

Arguments:

`objective` (Objective)
: Objective function.

`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
: Termination criterion.

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

`callbacks` (list of mlr3misc::Callback)
: List of callbacks.

`archive` (Archive).

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

Usage:

```
OptimInstanceBatchSingleCrit$assign_result(xdt, y, extra = NULL, ...)
```

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.

`extra` (`data.table::data.table()`)
: Additional information.

`...` (any)
ignored.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# define the objective function
fun = function(xs) {
  list(y = - (xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(
  y = p_dbl(tags = "maximize")
)

# create objective
objective = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)

# initialize instance
instance = oi(
  objective = objective,
  terminator = trm("evals", n_evals = 20)
)
```

`OptimInstanceMultiCrit` *Multi Criteria Optimization Instance for Batch Optimization*

Description

`OptimInstanceMultiCrit` is a deprecated class that is now a wrapper around OptimInstanceBatchMultiCrit.

Super classes

bbotk::OptimInstance -> bbotk::OptimInstanceBatch -> bbotk::OptimInstanceBatchMultiCrit -> OptimInstanceMultiCrit

Methods

Public methods:

- OptimInstanceMultiCrit$new()
- OptimInstanceMultiCrit$clone()

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

Usage:

```
OptimInstanceMultiCrit$new(
  objective,
  search_space = NULL,
  terminator,
  keep_evals = "all",
  check_values = TRUE,
  callbacks = NULL
)
```

Arguments:

`objective` (Objective)
: Objective function.

`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
: Termination criterion.

`keep_evals` (`character(1)`)
: Keep `all` or only `best` evaluations in archive?

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

`callbacks` (list of mlr3misc::Callback)
: List of callbacks.

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.

See Also

OptimInstanceBatchMultiCrit

OptimInstanceSingleCrit

Single Criterion Optimization Instance for Batch Optimization

Description

`OptimInstanceSingleCrit` is a deprecated class that is now a wrapper around `OptimInstanceBatchSingleCrit`.

Super classes

`bbotk::OptimInstance` -> `bbotk::OptimInstanceBatch` -> `bbotk::OptimInstanceBatchSingleCrit` -> `OptimInstanceSingleCrit`

Methods

Public methods:

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

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

Usage:

```
OptimInstanceSingleCrit$new(
  objective,
  search_space = NULL,
  terminator,
  keep_evals = "all",
  check_values = TRUE,
  callbacks = NULL
)
```

Arguments:

`objective` (Objective)
Objective function.

`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
Termination criterion.

`keep_evals` (`character(1)`)
Keep `all` or only `best` evaluations in archive?

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

`callbacks` (list of mlr3misc::Callback)
List of callbacks.

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.

See Also

OptimInstanceBatchSingleCrit

`Optimizer` *Optimizer*

Description

The `Optimizer` implements the optimization algorithm.

Details

`Optimizer` is an abstract base class that implements the base functionality each optimizer 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.

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")`.

Public fields

`id (character(1))`
: Identifier of the object. Used in tables, plot and text output.

Active bindings

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

`label (character(1))`
: Label for this object. Can be used in tables, plot and text output instead of the ID.

`man (character(1))`
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

`param_classes (character())`
: Supported parameter classes that the optimizer can optimize, as given in the `paradox::ParamSet` `$class` field.

`properties` (`character()`)
Set of properties of the optimizer. Must be a subset of `bbotk_reflections$optimizer_properties`.

`packages` (`character()`)
Set of required packages. A warning is signaled by the constructor if at least one of the packages is not installed, but loaded (not attached) later on-demand via `requireNamespace()`.

Methods

Public methods:

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

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

Usage:

```
Optimizer$new(
  id = "optimizer",
  param_set,
  param_classes,
  properties,
  packages = character(),
  label = NA_character_,
  man = NA_character_
)
```

Arguments:

`id` (`character(1)`)
Identifier for the new instance.

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

`param_classes` (`character()`)
Supported parameter classes that the optimizer can optimize, as given in the `paradox::ParamSet` `$class` field.

`properties` (`character()`)
Set of properties of the optimizer. Must be a subset of `bbotk_reflections$optimizer_properties`.

`packages` (`character()`)
Set of required packages. A warning is signaled by the constructor if at least one of the packages is not installed, but loaded (not attached) later on-demand via `requireNamespace()`.

`label` (`character(1)`)
Label for this object. Can be used in tables, plot and text output instead of the ID.

`man` (`character(1)`)
String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

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

Usage:

```
Optimizer$format(...)
```

Arguments:

`...` (ignored).

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

Usage:

```
Optimizer$print()
```

Returns: (`character()`).

Method `help()`**:** Opens the corresponding help page referenced by field `$man`.

Usage:

```
Optimizer$help()
```

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.

See Also

OptimizerAsync, OptimizerBatch

`OptimizerAsync` *Asynchronous Optimizer*

Description

The OptimizerAsync implements the asynchronous optimization algorithm. The optimization is performed asynchronously on a set of workers.

Details

OptimizerAsync is the abstract base class for all asynchronous optimizers. It provides the basic structure for asynchronous optimization algorithms. The public method `$optimize()` is the main entry point for the optimization and runs in the main process. The method starts the optimization process by starting the workers and pushing the necessary objects to the workers. Optionally, a set of points can be created, e.g. an initial design, and pushed to the workers. The private method `$.optimize()` is the actual optimization algorithm that runs on the workers. Usually, the method proposes new points, evaluates them, and updates the archive.

Optimization

The `rush::rush_plan()` function defines the number of workers and their type. There are three types of workers:

- "local": Workers are started as local processes with **processx**. See `$start_local_workers()` in rush::Rush for more details.
- "remote": Workers are started with **mirai** on local or remote machines. `mirai::daemons()` must be created before starting the optimization. See `$start_remote_workers()` in rush::Rush for more details.
- "script": Workers are started by the user with a custom script. See `$create_worker_script()` in rush::Rush for more details.

The workers are started when the `$optimize()` method is called. The main process waits until at least one worker is running. The optimization starts directly after the workers are running. The main process prints the evaluation results and other log messages from the workers. The optimization is terminated when the terminator criterion is satisfied. The result is assigned to the OptimInstanceAsync field. The main loop periodically checks the status of the workers. If all workers crash the optimization is terminated.

Debug Mode

The debug mode runs the optimization loop in the main process. This is useful for debugging the optimization algorithm. The debug mode is enabled by setting `options(bbotk.debug = TRUE)`.

Tiny Logging

The tiny logging mode is enabled by setting the option `bbotk.tiny_logging` to `TRUE`. In the tiny logging mode, the evaluated points are printed in a compact format and the currently best performing point is shown. Deactivated depending parameters are not printed.

Super class

`bbotk::Optimizer` -> `OptimizerAsync`

Methods

Public methods:

- `OptimizerAsync$optimize()`
- `OptimizerAsync$clone()`

Method `optimize()`**:** Performs the optimization on a OptimInstanceAsyncSingleCrit or OptimInstanceAsyncMultiCrit until termination. The single evaluations will be written into the ArchiveAsync. The result will be written into the instance object.

Usage:

```
OptimizerAsync$optimize(inst)
```

Arguments:

`inst` (OptimInstanceAsyncSingleCrit | OptimInstanceAsyncMultiCrit).

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

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

OptimizerAsyncDesignPoints, OptimizerAsyncGridSearch, OptimizerAsyncRandomSearch

---

OptimizerBatch *Batch Optimizer*

Description

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

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 class

bbotk::Optimizer -> OptimizerBatch

Methods

Public methods:

- OptimizerBatch$optimize()
- OptimizerBatch$clone()

Method optimize(): Performs the optimization and writes optimization result into OptimInstanceBatch. The optimization result is returned but the complete optimization path is stored in ArchiveBatch of OptimInstanceBatch.

Usage:

```
OptimizerBatch$optimize(inst)
```

Arguments:

inst (OptimInstanceBatch).

Returns: data.table::data.table.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

OptimizerBatchDesignPoints, OptimizerBatchGridSearch, OptimizerBatchRandomSearch

---

`shrink_ps` *Shrink a ParamSet towards a point.*

Description

Shrinks a paradox::ParamSet towards a point. Boundaries of numeric values are shrinked to an interval around the point of half of the previous length, while for discrete variables, a random (currently not chosen) level is dropped.

Note that for `paradox::p_lgl()`s the value to be shrinked around is set as the `default` value instead of dropping a level. Also, a tag `shrinked` is added.

Note that the returned paradox::ParamSet has lost all its original `defaults`, as they may have become infeasible.

If the paradox::ParamSet has a trafo, `x` is expected to contain the transformed values.

Usage

```
shrink_ps(param_set, x, check.feasible = FALSE)
```

Arguments

`param_set`	(paradox::ParamSet) The paradox::ParamSet to be shrinked.
`x`	(data.table::data.table) data.table::data.table with one row containing the point to shrink around.
`check.feasible`	(`logical(1)`) Should feasibility of the parameters be checked? If feasibility is not checked, and invalid values are present, no shrinking will be done. Must be turned off in the case of the paradox::ParamSet having a trafo. Default is `FALSE`.

Value

paradox::ParamSet

Examples

```
library(paradox)
library(data.table)
param_set = ps(
  x = p_dbl(lower = 0, upper = 10),
  x2 = p_int(lower = -10, upper = 10),
  x3 = p_fct(levels = c("a", "b", "c")),
  x4 = p_lgl()
)
x = data.table(x1 = 5, x2 = 0, x3 = "b", x4 = FALSE)
shrink_ps(param_set, x = x)
```

---

terminated_error *Termination Error*

Description

Error class for termination.

Usage

```
terminated_error(optim_instance)
```

Arguments

optim_instance	OptimInstance OptimInstance that terminated.

Value

A 'terminated_error' condition (inherits from 'error' and 'condition').

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.

Technical details

`Terminator` subclasses can overwrite `.status()` to support progress bars via the package **progressr**. The method must return the maximum number of steps (`max_steps`) and the currently achieved number of steps (`current_steps`) as a named integer vector.

Public fields

`id (character(1))`
: Identifier of the object. Used in tables, plot and text output.

Active bindings

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

`label (character(1))`
: Label for this object. Can be used in tables, plot and text output instead of the ID.

`man (character(1))`
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

`properties (character())`
: Set of properties of the terminator. Must be a subset of `bbotk_reflections$terminator_properties`.

`unit (character())`
: Unit of steps.

Methods

Public methods:

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

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

Usage:

```
Terminator$new(
  id,
  param_set = ps(),
  properties = character(),
  unit = "percent",
  label = NA_character_,
  man = NA_character_
)
```

Arguments:

`id` (`character(1)`)
Identifier for the new instance.

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

`properties` (`character()`)
Set of properties of the terminator. Must be a subset of `bbotk_reflections$terminator_properties`.

`unit` (`character()`)
Unit of steps.

`label` (`character(1)`)
Label for this object. Can be used in tables, plot and text output instead of the ID.

`man` (`character(1)`)
String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

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

Usage:

```
Terminator$format(with_params = FALSE, ...)
```

Arguments:

`with_params` (`logical(1)`)
Add parameter values to format string.

`...` (ignored).

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

Usage:

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

Arguments:

`...` (ignored).

Method `status()`**:** Returns how many progression steps are made (`current_steps`) and the amount steps needed for termination (`max_steps`).

Usage:

```
Terminator$status(archive)
```

Arguments:

`archive` (Archive).

Returns: named `integer(2)`.

Method `remaining_time()`: Returns remaining runtime in seconds. If the terminator is not time-based, the reaming runtime is `Inf`.

Usage:

```
Terminator$remaining_time(archive)
```

Arguments:

`archive` (Archive).

Returns: `integer(1)`.

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`, `mlr_terminators_clock_time`, `mlr_terminators_combo`, `mlr_terminators_evals`, `mlr_terminators_none`, `mlr_terminators_perf_reached`, `mlr_terminators_run_time`, `mlr_terminators_stagnation`, `mlr_terminators_stagnation_batch`, `mlr_terminators_stagnation_hypervolume`

---

trafo_xs *Calculate the transformed x-values*

Description

Transforms a given `list()` to a list with transformed x values.

Usage

```
trafo_xs(xs, search_space)
```

Arguments

`xs`	(`list()`) List of x-values.
`search_space`	paradox::ParamSet Search space.

Value

`list()` with transformed x values.

`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