Package ‘bbotk’

September 13, 2021

Title Black-Box Optimization Toolkit

Version 0.4.0

Description Provides a common framework for optimization of black-box
functions for other packages, e.g. 'mlr3tuning' or 'mlr3fselect'. It
offers various optimization methods e.g. grid search, random search,
genewed simulated annealing and iterated racing.

License LGPL-3


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

Depends paradox (>= 0.7.0), R (>= 3.1.0)

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

Suggests adagio, GenSA, irace, knitr, nloptr, progressr, rmarkdown,
testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

Config/testthat/parallel false

Encoding UTF-8

Language en-US

NeedsCompilation yes

RoxygenNote 7.1.2

Collate 'Archive.R' 'ArchiveBest.R' 'Objective.R' 'ObjectiveRFun.R'
'ObjectiveRFunDt.R' 'OptimInstance.R'
'OptimInstanceMultiCrit.R' 'OptimInstanceSingleCrit.R'
'mlr_optimizers.R' 'Optimizer.R' 'OptimizerCmaes.R'
'OptimizerDesignPoints.R' 'OptimizerGenSA.R'
'OptimizerGridSearch.R' 'OptimizerIrace.R' 'OptimizerNLOptr.R'
'OptimizerRandomSearch.R' 'Progressor.R' 'mlr_terminators.R'
'TerminatorEvals.R' 'TerminatorNone.R'
R topics documented:

'TerminatorPerfReached.R' 'TerminatorRunTime.R'
'assertions.R' 'bb_optimize.R' 'bbotk_reflections.R'
'bibentries.R' 'helper.R' 'helper_irace.R' 'sugar.R' 'zzz.R'

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

Maintainer  Marc Becker <marcbecker@posteo.de>

Repository  CRAN

Date/Publication  2021-09-13 15:50:08 UTC

R topics documented:

bbotk-package .................................................. 3
Archive ............................................................ 4
ArchiveBest ........................................................ 6
bb_optimize ........................................................ 8
is_dominated ....................................................... 10
mlr_optimizers ................................................... 11
mlr_optimizers_cmaes ............................................ 11
mlr_optimizers_design_points .................................. 13
mlr_optimizers_gensa ........................................... 15
mlr_optimizers_grid_search .................................... 17
mlr_optimizers_irace ........................................... 19
mlr_optimizers_nloptr .......................................... 21
mlr_optimizers_random_search ................................. 23
mlr_terminators ................................................ 25
mlr_terminators_clock_time ................................... 26
mlr_terminators_combo ......................................... 28
mlr_terminators_evals ......................................... 30
mlr_terminators_none .......................................... 31
mlr_terminators_perf_reached ................................. 33
mlr_terminators_run_time ..................................... 34
mlr_terminators_stagnation ................................... 36
mlr_terminators_stagnation_batch ............................ 37
Objective ......................................................... 39
ObjectiveRFun .................................................... 41
ObjectiveRFunDt ................................................ 43
opt ................................................................. 45
OptimInstance .................................................... 46
OptimInstanceMultiCrit ......................................... 49
OptimInstanceSingleCrit ....................................... 50
Optimizer ........................................................ 52
Description

Provides a common framework for optimization of black-box functions for other packages, e.g. 'mlr3tuning' or 'mlr3fselect'. It offers various optimization methods e.g. grid search, random search, generalized simulated annealing and iterated racing.

Author(s)

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

Authors:

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

Other contributors:

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

See Also

Useful links:

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

Logging object for objective function evaluations

Description

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

S3 Methods

- `as.data.table(archive)`
  
  `Archive -> 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.

Public fields

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

- `codomain (paradox::ParamSet)`
  
  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.

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

- `store_x_domain (logical(1))`
  
  Determines if x values, should be stored in `$data$x_domain` as list items. The trafo will be applied if defined in `search_space`.

Active bindings

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

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

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

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

Public methods:

• Archive$new()
• Archive$add_evals()
• Archive$best()
• Archive$format()
• Archive/print()  
• Archive$clear()
• Archive$clone()

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

Usage:
Archive$new(search_space, codomain, check_values = TRUE, store_x_domain = TRUE)

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

codomain  (paradox::ParamSet)
  Specifies codomain of function. Most importantly the tags of each output "Parameter" de-
  fine whether it should be minimized or maximized. The default is to minimize each com-
  ponent.

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

store_x_domain  (logical(1))
  Determines if x values, should be stored in $data$x_domain as list items. The trafo will be
  applied if defined in search_space.

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

Usage:
Archive$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.

oxss_trafoed  (list())
  Transformed point(s) in the domain space. Not stored and needed if store_x_domain =
  FALSE.

ydt  (data.table::data.table())
  Optimal outcome.
Method best(): Returns the best scoring evaluation. For single-crit optimization, the solution
that minimizes / maximizes the objective function. For multi-crit optimization, the Pareto set /
front.

Usage:
Archive$best(batch = NULL)
Arguments:
batch (integer())
   The batch number(s) to limit the best results to. Default is all batches.
Returns: data.table::data.table().

Method format(): Helper for print outputs.

Usage:
Archive$format()

Method print(): Printer.

Usage:
Archive$print()
Arguments:
... (ignored).

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

Usage:
Archive$clear()

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

Usage:
Archive$clone(deep = FALSE)
Arguments:
deep Whether to make a deep clone.

---

| ArchiveBest | Minimal logging object for objective function evaluations |

Description

The ArchiveBest stores no data but records the best scoring evaluation passed to $add_evals().
The Archive API is fully implemented but many parameters are ignored and some methods do
nothing. The archive still works with TerminatorClockTime, TerminatorEvals, TerminatorNone
and TerminatorEvals.

Super class

bbotk::Archive -> ArchiveBest
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:
- ArchiveBest$new()
- ArchiveBest$add_evals()
- ArchiveBest$best()
- ArchiveBest$clone()

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

Usage:
ArchiveBest$new(
   search_space,
   codomain,
   check_values = FALSE,
   store_x_domain = 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))
  ignored.
- store_x_domain (logical(1))
  Determines if x values, should be stored in $data$x_domain as list items. The trafo will be applied if defined in search_space.

Method add_evals(): Stores the best result in ydt.

Usage:
ArchiveBest$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.
MetaDescription

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,  
)
bb_optimize

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" - OptimInstanceSingleCrit | OptimInstanceMultiCrit
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

```r
# 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)
```

---

**is_dominated**

`Calculate which points are dominated`

**Description**

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

**Usage**

```r
is_dominated(ymat)
```

**Arguments**

- `ymat` (matrix())
  - A numeric matrix. Each column (!) contains one point.
Dictionary of Optimizer

Description

A simple mlr3misc::Dictionary storing objects of class Optimizer. Each optimizer has an associated help page, see mlr_optimizer_[id]. This dictionary can get populated with additional optimizer by add-on packages. For a more convenient way to retrieve and construct optimizer, see opt()/opts().

Format

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

Methods

See mlr3misc::Dictionary.

See Also

Sugar functions: opt(), opts()

Examples

opt("random_search", batch_size = 10)

mlr_optimizers_cmaes
Optimization via Covariance Matrix Adaptation Evolution Strategy

Description

OptimizerCmaes 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.

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

Soptimize() 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 -> OptimizerCmaes

Methods

Public methods:

• OptimizerCmaes$new()
• OptimizerCmaes$clone()

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

Usage:
OptimizerCmaes$new()

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

Usage:
OptimizerCmaes$clone(deep = FALSE)

Arguments:
dee{p} Whether to make a deep clone.

Examples

if(requireNamespace("adagio"){  search_space = domain = ps(    x1 = p_dbl(-10, 10),    x2 = p_dbl(-5, 5)  )  
codomain = ps(y = p_dbl(tags = "maximize"))  
objective_function = function(xs) {    c(y = -(xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)  }
```
objective = ObjectiveRFun$new(
  fun = objective_function,
  domain = domain,
  codomain = codomain)

instance = OptimInstanceSingleCrit$new(
  objective = objective,
  search_space = search_space,
  terminator = trm("evals", n_evals = 10))

optimizer = opt("cmaes")

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

# returns best scoring evaluation
instance$result

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

---

### Optimization via Design Points

#### Description

OptimizerDesignPoints class that implements optimization w.r.t. fixed design points. We simply search over a set of points fully specified by the user. The points in the design are evaluated in order as given.

In order to support general termination criteria and parallelization, we evaluate points in a batch-fashion of size `batch_size`. Larger batches mean we can parallelize more, smaller batches imply a more fine-grained checking of termination criteria.

#### Dictionary

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

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

#### Parameters

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

Methods

Public methods:

• `OptimizerDesignPoints$new()`
• `OptimizerDesignPoints$clone()`

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

Usage:
`OptimizerDesignPoints$new()`

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

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

Arguments:

depth Whether to make a deep clone.

Examples

```r
library(data.table)
search_space = domain = ps(x = p_dbl(lower = -1, upper = 1))
codomain = ps(y = p_dbl(tags = "minimize"))

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

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

instance = OptimInstanceSingleCrit$new(
  objective = objective,
  search_space = search_space,
  terminator = trm("evals", n_evals = 10))

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

optimizer = opt("design_points", design = design)
```
# Modifies the instance by reference
optimizer$optimize(instance)

# Returns best scoring evaluation
instance$result

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

---

**mlr_optimizers_gensa**  *Optimization via Generalized Simulated Annealing*

---

**Description**

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

**Dictionary**

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

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

**Parameters**

- `smooth` logical(1)
- `temperature` numeric(1)
- `acceptance.param` numeric(1)
- `verbose` logical(1)
- `trace.mat` logical(1)

For the meaning of the control parameters, see `GenSA::GenSA()`. Note that we have removed all control parameters which refer to the termination of the algorithm and where our terminators allow to obtain the same behavior.

**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` $\rightarrow$ OptimizerGenSA
Methods

Public methods:

• OptimizerGenSA$new()
• OptimizerGenSA$clone()

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

Usage:
OptimizerGenSA$new()

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

Usage:
OptimizerGenSA$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

Source


Examples

if(requireNamespace("GenSA")) {

search_space = domain = ps(x = p dbl(lower = -1, upper = 1))

codomain = ps(y = p dbl(tags = "minimize"))

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

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

instance = OptimInstanceSingleCrit$new(
  objective = objective,
  search_space = search_space,
  terminator = trm("evals", n_evals = 10))

optimizer = opt("cmaes")

# Modifies the instance by reference
optimizer$optimize(instance)
# Returns best scoring evaluation
instance$result

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

mlr_optimizers_grid_search

Optimization via Grid Search

Description

OptimizerGridSearch class that implements grid search. The grid is constructed as a Cartesian product over discretized values per parameter, see paradox::generate_design_grid(). The points of the grid are evaluated in a random order.

In order to support general termination criteria and parallelization, we evaluate points in a batch-fashion of size batch_size. Larger batches mean we can parallelize more, smaller batches imply a more fine-grained checking of termination criteria.

Dictionary

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

mlr_optimizers$get("grid_search")
opt("grid_search")

Parameters

resolution integer(1)
Resolution of the grid, see paradox::generate_design_grid().

param_resolutions named integer()
Resolution per parameter, named by parameter ID, see paradox::generate_design_grid().

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

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 progressr as backend; enable with progressr::handlers("progress").

Super class

bbotk::Optimizer -> OptimizerGridSearch
Methods

Public methods:

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

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

Usage:

`OptimizerGridSearch$new()`

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

Usage:

`OptimizerGridSearch$clone(deep = FALSE)`

Arguments:

- `deep` Whether to make a deep clone.

Examples

```r
search_space = domain = ps(x = p_dbl(lower = -1, upper = 1))
codomain = ps(y = p_dbl(tags = "minimize"))

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

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

instance = OptimInstanceSingleCrit$new(
  objective = objective,
  search_space = search_space,
  terminator = trm("evals", n_evals = 10))

optimizer = opt("grid_search")

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

# returns best scoring evaluation
instance$result

# allows access of data.table of full path of all evaluations
as.data.table(instance$archive$data)
```
mlr_optimizers_irace  Optimization via Iterated Racing

Description

OptimizerIrace 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(). Note that we have removed all control parameters which refer to the termination of the algorithm. Use TerminatorEvals instead. Other terminators do not work with OptimizerIrace.

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 OptimizerIrace. 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():

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

Progress Bars

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

Super class

bbotk::Optimizer -> OptimizerIrace

Methods

Public methods:
- OptimizerIrace$new()
- OptimizerIrace$clone()

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

Usage:
OptimizerIrace$new()

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

Usage:
OptimizerIrace$clone(deep = FALSE)

Arguments:
- deep Whether to make a deep clone.

Source


Examples

```r
library(data.table)

search_space = domain = ps(
  x1 = p_dbl(-5, 10),
  x2 = p_dbl(0, 15)
)
```
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) {
  a = 1
  b = 5.1 / (4 * (pi ^ 2))
  c = 5 / pi
  r = 6
  s = 10
  t = 1 / (8 * pi)

  data.table(y = (a * ((xdt["x2"] - b * (xdt["x1"] ^ 2L) + c * xdt["x1"] - r) ^ 2) +
              ((s * (1 - t)) * cos(xdt["x1"])) +
              unlist(instances)))
}

objective = ObjectiveRFunDt$new(fun = fun, domain = domain, codomain = codomain)

instance = OptimInstanceSingleCrit$new(
  objective = objective,
  search_space = search_space,
  terminator = trm("evals", n_evals = 1000))

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

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

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

# best scoring configuration
instance$result

# all evaluations
as.data.table(instance$archive)

---

**mlr_optimizers_nloptr**  
*Optimization via Non-linear Optimization*

---

**Description**

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

algorithm character(1)  
eval_g_ineq function()  
xtol_rel numeric(1)  
xtol_abs numeric(1)  
ftol_rel numeric(1)  
ftol_abs numeric(1)  
start_values character(1)

Create random start values or based on center of search space? In the latter case, it is the center of the parameters before a trafo is applied.

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

The termination conditions stopval, maxtime and maxeval of `nloptr::nloptr()` are deactivated and replaced by the `Terminator` subclasses. The x and function value tolerance termination conditions (xtol_rel = $10^{-4}$, xtol_abs = rep(0,0,length(x0)), ftol_rel = 0.0 and ftol_abs = 0.0) are still available and implemented with their package defaults. To deactivate these conditions, set them to -1.

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` $\rightarrow$ `OptimizerNLoptr`

Methods

Public methods:

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

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

Usage:

`OptimizerNLoptr$new()`

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

Usage:

`OptimizerNLoptr$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.
### Source


### Examples

```r
if(requireNamespace("nloptr")) {

  search_space = domain = ps(x = p_dbl(lower = -1, upper = 1))
  codomain = ps(y = p_dbl(tags = "minimize"))

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

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

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

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

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

  # Returns best scoring evaluation
  instance$result

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

---

**mlr_optimizers_random_search**

*Optimization via Random Search*
Description

OptimizerRandomSearch class that implements a simple Random Search.

In order to support general termination criteria and parallelization, we evaluate points in a batch-fashion of size batch_size. Larger batches mean we can parallelize more, smaller batches imply a more fine-grained checking of termination criteria.

Dictionary

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

```r
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 class

`bbotk::Optimizer` -> OptimizerRandomSearch

Methods

Public methods:

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

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

Usage:

```r
OptimizerRandomSearch$new()
```

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

Usage:

```r
OptimizerRandomSearch$clone(deep = FALSE)
```

Arguments:

- deep Whether to make a deep clone.
**mllr_terminators**

**Source**


**Examples**

```r
search_space = domain = ps(x = p_dbl(lower = -1, upper = 1))
codomain = ps(y = p_dbl(tags = "minimize"))

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

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

instance = OptimInstanceSingleCrit$new(
  objective = objective,
  search_space = search_space,
  terminator = trm("evals", n_evals = 10))

optimizer = opt("random_search")

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

# returns best scoring evaluation
instance$result

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

**Dictionary of Terminators**

**Description**

A simple *mlr3misc::Dictionary* storing objects of class *Terminator*. Each terminator has an associated help page, see *mllr_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

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

Methods

See mlr3misc::Dictionary.

See Also

Sugar functions: \texttt{trm()}, \texttt{trms()}


Examples

\begin{verbatim}
trm("evals", n_evals = 10)
\end{verbatim}

\begin{verbatim}
mlr_terminators_clock_time

\textit{Terminator that stops according to the clock time}
\end{verbatim}

Description

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

Dictionary

This Terminator can be instantiated via the dictionary mlr_terminators or with the associated sugar function \texttt{trm()}:

\begin{verbatim}
mlr_terminators$get("clock_time")
trm("clock_time")
\end{verbatim}

Parameters

\begin{verbatim}
stop_time POSIXct(1)

Terminator stops after this point in time.
\end{verbatim}

Super class

\texttt{bbotk::Terminator} \rightarrow \texttt{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


Examples

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

```r
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:

```r
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:*
deepe Whether to make a deep clone.

**See Also**

Examples

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

### 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

$$B = n_{\text{vals}} + k \times D$$

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()`:

```r
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` $\rightarrow$ `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**


**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**  
Terminator that never stops.

**Description**

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

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

```r
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:

```r
TerminatorNone$new()
```

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

Usage:

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

Arguments:

archive (Archive).

Returns: logical(1).

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

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**

**Examples**
TerminatorPerfReached$new()
trm("perf_reached")

---

**mlr_terminators_run_time**

*Terminator that stops according to the run time*

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

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

```r	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`
mlr_terminators_run_time

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:
depth 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


Examples

trm("run_time", secs = 1800)
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

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

TerminatorStagnation_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


Examples

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

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

Technical details

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

Public fields

- id (character(1)).
- properties (character()).
- domain (paradox::ParamSet)
  - Specifies domain of function, hence its input parameters, their types and ranges.
- codomain (paradox::ParamSet)
  - Specifies codomain of function, hence its feasible values.
- 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

- xdim (integer(1))
  - Dimension of domain.
- ydim (integer(1))
  - Dimension of codomain.

Methods

Public methods:

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

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

**Usage:**

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

**Arguments:**

- `id` (character(1))
- `properties` (character())
- `domain` (paradox::ParamSet)
  - Specifies domain of function. The paradox::ParamSet should describe all possible input parameters of the objective function. This includes their id, their types and the possible range.
- `codomain` (paradox::ParamSet)
  - Specifies codomain of function. Most importantly the tags of each output "Parameter" define whether it should be minimized or maximized. The default is to minimize each component.
- `constants` (paradox::ParamSet)
  - Changeable constants or parameters that are not subject to tuning can be stored and accessed here.
- `check_values` (logical(1))
  - Should points before the evaluation and the results be checked for validity?

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

**Usage:**

```r
Objective$format()
```

**Returns:** character().

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

**Usage:**

```r
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:**

```r
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:

```r
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:

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

Arguments:

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

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

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

Usage:

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

Arguments:

- `deep` Whether to make a deep clone.

---

**ObjectiveRFun**  
*Objective interface with custom R function*

**Description**

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

**Super class**

`bbotk::Objective -> ObjectiveRFun`
Active bindings

fun (function)
   Objective function.

Methods

Public methods:

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

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

Usage:
ObjectiveRFun$new(
   fun,  
   domain,  
   codomain = NULL,  
   id = "function",  
   properties = character(),  
   constants = ps(),  
   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" de- 
   fine whether it should be minimized or maximized. The default is to minimize each com- 
   ponent.

id (character(1)).
properties (character()).
constants (paradox::ParamSet)  
   Changeable constants or parameters that are not subject to tuning can be stored and accessed  
   here.

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 sup- 
plied 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.

Examples

# define objective function
fun = function(xs) {
  - (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 object
obfun = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)
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(),
  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.

- **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 NAs in xdt.

Usage:

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

Arguments:

- **xss** (list())
  A list of lists that contains multiple x values, e.g. `list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4)`. 
Returns: `data.table::data.table()` that contains one y-column for single-criteria functions and multiple y-columns for multi-criteria functions, e.g. `data.table(y = 1:2)` or `data.table(y1 = 1:2, y2 = 3:4)`.

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

Usage:
ObjectiveRFunDt$eval_dt(xdt)

Arguments:

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

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

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

Usage:
ObjectiveRFunDt$clone(deep = FALSE)

Arguments:

deep  Whether to make a deep clone.

---

**opt**  
*Syntactic Sugar Optimizer Construction*

**Description**

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

**Usage**

```r
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 `para-
  dox::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.
OptimInstance

Value

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

Examples

opt("random_search", batch_size = 10)

OptimInstance

Optimization Instance with budget and archive

Description

Abstract base class.

Technical details

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

Public fields

- objective (Objective).
- search_space (paradox::ParamSet).
- terminator (Terminator).
- archive (Archive).
- progressor (progressor())
  Stores progressor function.
- 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:

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

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

Usage:
OptimInstance$new(
  objective,
  search_space = NULL,
  terminator,
  keep_evals = "all",
  check_values = TRUE
)

Arguments:
objective (Objective).
search_space (paradox::ParamSet)
  Specifies the search space for the Optimizer. The paradox::ParamSet describes either a
  subset of the domain of the Objective or it describes a set of parameters together with
  a trafo function that transforms values from the search space to values of the domain.
  Depending on the context, this value defaults to the domain of the objective.
terminator (Terminator).
keep_evals (character(1))
  Keep all or only best evaluations in archive?
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 format(): Helper for print outputs.

Usage:
OptimInstance$format()

Method print(): Printer.

Usage:
OptimInstance$print(...)

Arguments:
...
  (ignored).
Method `eval_batch()`: Evaluates all input values in `xdt` by calling the `Objective`. Applies possible transformations to the input values and writes the results to the `Archive`. Before each batch-evaluation, the `Terminator` is checked, and if it is positive, an exception of class `terminated_error` is raised. This function should be internally called by the `Optimizer`.

Usage:
```
OptimInstance$eval_batch(xdt)
```

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

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

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

Arguments:
- `xdt` (data.table::data.table())
  - `x` values as `data.table()` with one row. Contains the value in the `search space` of the `OptimInstance` object. Can contain additional columns for extra information.
- `y` (numeric(1))
  - Optimal outcome.

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

Usage:
```
OptimInstance$objective_function(x)
```

Arguments:
- `x` (numeric())
  - Untransformed points.

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

Method `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.
**OptimInstanceMultiCrit**

*Optimization Instance with budget and archive*

---

**Description**

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

- Automatic storing of results in an **Archive** after evaluation.
- Automatic checking for termination. Evaluations of design points are performed in batches. Before a batch is evaluated, the **Terminator** is queried for the remaining budget. If the available budget is exhausted, an exception is raised, and no further evaluations can be performed from this point on.

**Super class**

`bbotk::OptimInstance` -> `OptimInstanceMultiCrit`

**Active bindings**

- `result_x_domain` (list())
  - (transformed) x part of the result in the *domain space* of the objective.
- `result_y` (numeric(1))
  - Optimal outcome.

**Methods**

**Public methods:**

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

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

**Usage:**

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

**Arguments:**

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

terminator (Terminator)
  Multi-criteria terminator.

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

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

Usage:
  OptimInstanceMultiCrit$assign_result(xdt, ydt)

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

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

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

Usage:
  OptimInstanceMultiCrit$clone(deep = FALSE)

Arguments:
  deep Whether to make a deep clone.

OptimInstanceSingleCrit

Optimization Instance with budget and archive

Description

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

- Automatic storing of results in an Archive after evaluation.
- Automatic checking for termination. Evaluations of design points are performed in batches. Before a batch is evaluated, the Terminator is queried for the remaining budget. If the available budget is exhausted, an exception is raised, and no further evaluations can be performed from this point on.
Super class

`bbotk::OptimInstance` -> `OptimInstanceSingleCrit`

Methods

Public methods:

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

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

Usage:

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

Arguments:

- `objective` (Objective).
- `search_space` (paradox::ParamSet)
  Specifies the search space for the Optimizer. The paradox::ParamSet describes either a subset of the domain of the Objective or it describes a set of parameters together with a trafo function that transforms values from the search space to values of the domain. Depending on the context, this value defaults to the domain of the objective.
- `terminator` (Terminator).
- `keep_evals` (character(1))
  Keep all or only best evaluations in archive?
- `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 `assign_result()`: The Optimizer object writes the best found point and estimated performance value here. For internal use.

Usage:

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

Arguments:

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

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

Usage:
Optimizer$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

Description

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

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

Progress Bars

Optimizer$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

param_classes (character()).
properties (character()).
packages (character()).

Active bindings

param_set (paradox::ParamSet).

Methods

Public methods:

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

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

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

Arguments:
param_set (paradox::ParamSet).
param_classes (character()).
properties (character()).
packages (character()).

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

*Usage:*
Optimizer$format()

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

*Usage:*
Optimizer$print()

*Returns*: (character()).

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

*Usage:*
Optimizer$optimize(inst)

*Arguments:*
inst (OptimInstance).

*Returns*: data.table::data.table.

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

*Usage:*
Optimizer$clone(deep = FALSE)

*Arguments:*
deeep Whether to make a deep clone.

---

**Progressor**

Wraps progressr::progressor() function and stores current progress.

**Public fields**

progressor (progressr::progressor()).
max_steps (integer(1)).
current_steps (integer(1)).
unit (character(1)).
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

- `param_set`: `paradox::ParamSet`
  Set of control parameters for terminator.
- `properties`: `character()`
  Set of 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(param_set = ps(), properties = character())`

Arguments:

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

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

Usage:
`Terminator$format(with_params = FALSE)`

Arguments:
with_params (logical(1))
    Add parameter values to format string.

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 remaining 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

### Description

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

### Usage

```r
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 `paramo::ParamSet`, or to be set as public field. See `mlr3misc::dictionary_sugar_get()` for more details.

- `.keys` (character(1))
  Keys passed to the respective dictionary to retrieve multiple objects.

### Value

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

### Examples

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

* **Optimizer**
  mlr_optimizers, 11

* **Terminator**
  mlr_terminators, 25
  mlr_terminators_clock_time, 26
  mlr_terminators_combo, 28
  mlr_terminators_evals, 30
  mlr_terminators_none, 31
  mlr_terminators_perf_reached, 33
  mlr_terminators_run_time, 34
  mlr_terminators_stagnation, 36
  mlr_terminators_stagnation_batch, 37
  Terminator, 54

* **datasets**
  mlr_optimizers, 11
  mlr_terminators, 25
  adagio::pureCMAES(), 11, 12
  Archive, 4, 4, 6, 19, 27, 29, 31–33, 35, 36, 38, 46, 48–50, 53, 54, 56
  ArchiveBest, 6, 6
  bb_optimize, 8
  bbotk (bbotk-package), 3
  bbotk-package, 3
  bbotk::Archive, 6
  bbotk::Objective, 41, 43
  bbotk::OptimInstance, 49, 51
  bbotk::Optimizer, 12, 14, 15, 17, 20, 22, 24
  bbotk::Terminator, 26, 28, 30, 32–34, 36, 37
  data.table::data.table, 4, 13, 29, 46, 53
  data.table::data.table(), 4–8, 41, 45, 50, 51
  dictionary, 11, 13, 15, 17, 20, 24, 26, 28, 30, 32–34, 36, 37, 45, 57
  GenSA::GenSA(), 15
  irace::defaultScenario(), 19
  is dominated, 10
  irace::irace(), 19
  mlr3misc::Dictionary, 11, 25, 26
  mlr3misc::dictionary_sugar_get(), 45, 57
  mlr_optimizers, 9, 11, 13, 15, 17, 20, 24, 45
  mlr_optimizers_cmaes, 11
  mlr_optimizers_design_points, 13
  mlr_optimizers_genesa, 15
  mlr_optimizers_grid_search, 17
  mlr_optimizers_irace, 19
  mlr_optimizers_nloptr, 21
  mlr_optimizers_random_search, 23
  mlr_terminators, 25, 26–38, 56, 57
  mlr_terminators_clock_time, 26, 26, 27, 29, 31, 32, 34, 35, 37, 38, 56
  mlr_terminators_combo, 26, 27, 29, 31, 32, 34, 35, 37, 38, 56
  mlr_terminators_evals, 26, 27, 29, 30, 32, 34, 35, 37, 38, 56
  mlr_terminators_none, 26, 27, 29, 31, 34, 35, 37, 38, 56
  mlr_terminators_perf_reached, 26, 27, 29, 31, 32, 33, 35, 37, 38, 56
  mlr_terminators_run_time, 26, 27, 29, 31, 32, 33, 35, 37, 38, 56
  mlr_terminators_stagnation, 26, 27, 29, 31, 32, 34, 35, 36, 38, 56
  mlr_terminators_stagnation_batch, 26, 27, 29, 31, 32, 34, 35, 37, 38, 56
  nloptr::nloptr(), 21, 22
  nloptr::nloptr.print.options(), 22
  Objective, 4, 5, 7–9, 19, 39, 46–51
  ObjectiveRFun, 41
  ObjectiveRFunDt, 43
  opt, 45

58
INDEX

opt(), 11, 13, 15, 17, 20, 24
OptimInstance, 40, 41, 46, 48–50, 52, 53
OptimInstanceMultiCrit, 9, 49
OptimInstanceSingleCrit, 9, 50
Optimizer, 4, 5, 7, 9–11, 13, 15, 17, 20, 24,
35, 46–48, 50, 51, 52
OptimizerCmaes (mlr_optimizers_cmaes), 11
OptimizerDesignPoints
  (mlr_optimizers_design_points), 13
OptimizerGenSA (mlr_optimizers_genSA), 15
OptimizerGridSearch, 31
OptimizerGridSearch
  (mlr_optimizers_grid_search), 17
OptimizerIrace (mlr_optimizers_irace), 19
OptimizerNloptr
  (mlr_optimizers_nloptr), 21
OptimizerRandomSearch
  (mlr_optimizers_random_search), 23
opts (opt), 45
opts(), 11

paradox::generate_design_grid(), 17
paradox::ParamSet, 4, 5, 7, 9, 39, 40, 42,
44–47, 50–53, 55, 57
POSIXct, 4
Progressor, 53

R6, 5, 7, 12, 14, 16, 18, 20, 22, 24, 27, 28, 30,
32, 33, 35, 36, 38, 39, 42, 44, 47, 49,
51, 52, 54, 55
R6::R6Class, 11, 26

Sys.time(), 26

 Terminator, 12, 14, 15, 17, 20, 22, 24–38,
46–52, 54, 54, 57
TerminatorClockTime, 6
TerminatorClockTime
  (mlr_terminators_clock_time), 26
TerminatorCombo, 10
TerminatorCombo
  (mlr_terminators_combo), 28
TerminatorEvals, 6, 19
TerminatorEvals
  (mlr_terminators_evals), 30
TerminatorNone, 6, 10
TerminatorNone (mlr_terminators_none), 31
TerminatorPerfReached
  (mlr_terminators_perf_reached), 33
TerminatorRunTime
  (mlr_terminators_run_time), 34
TerminatorStagnation
  (mlr_terminators_stagnation), 36
TerminatorStagnationBatch
  (mlr_terminators_stagnation_batch), 37
trm, 57
trm(), 25, 26, 28, 30, 32–34, 36, 37
trms (trm), 57
trms(), 25, 26