Package 'ParBayesianOptimization'

October 20, 2026

Title Parallel Bayesian Optimization of Hyperparameters

Version 1.4.2

Description Fast, flexible framework for implementing Bayesian optimization of model hyperparameters according to the methods described in Snoek et al. (2012) <doi:10.48550/arXiv.1206.2944>. The package allows the user to run scoring function in parallel, save intermediary results, and tweak other aspects of the process to fully utilize the computing resources available to the user.

License GPL-2

URL https://github.com/novica/ParBayesianOptimization

BugReports https://github.com/novica/ParBayesianOptimization/issues

Depends R (>= 4.1)

Imports cli, data.table (>= 1.11.8), dbscan, DiceKriging, foreach, ggplot2 (>= 3.3.0), graphics, lhs, patchwork, stats, utils

Suggests knitr, quarto, testthat (>= 3.1.8), withr, xgboost (>= 3.0.0)

VignetteBuilder quarto

Config/roxygen2/version 8.1.0

Config/testthat/edition 3

Encoding UTF-8

NeedsCompilation no

Author Samuel Wilson [aut, cph], Novica Nakov [aut, cre, cph] (ORCID: <https://orcid.org/0009-0005-7773-7718>)

Maintainer Novica Nakov <nnovica@gmail.com>

Repository CRAN

Date/Publication 2026-10-20 16:40:02 UTC

Contents

addIterations 2
bayesOpt 4
changeSaveFile 8
getBestPars 9
getLocalOptimums 10
plot.bayesOpt 12
print.bayesOpt 13
updateGP 13

Index **15**

`addIterations` *Run Additional Optimization Iterations*

Description

Use this function to continue optimization of a bayesOpt object.

Usage

```
addIterations(
  optObj,
  iters.n = 1,
  iters.k = 1,
  otherHalting = list(timeLimit = Inf, minUtility = 0),
  bounds = optObj$bounds,
  acq = optObj$optPars$acq,
  kappa = optObj$optPars$kappa,
  eps = optObj$optPars$eps,
  gsPoints = optObj$optPars$gsPoints,
  convThresh = optObj$optPars$convThresh,
  acqThresh = optObj$optPars$acqThresh,
  errorHandling = "stop",
  saveFile = optObj$saveFile,
  parallel = FALSE,
  plotProgress = FALSE,
  verbose = 1,
  ...
)
```

Arguments

`optObj`	an object of class `bayesOpt`.
`iters.n`	The total number of additional times to sample the scoring function.

iters.k	integer that specifies the number of times to sample FUN at each Epoch (optimization step). If running in parallel, good practice is to set iters.k to some multiple of the number of cores you have designated for this process. Must belower than, and preferrably some multiple of iters.n.
otherHalting	Same as bayesOpt()
bounds	Same as bayesOpt()
acq	Same as bayesOpt()
kappa	Same as bayesOpt()
eps	Same as bayesOpt()
gsPoints	Same as bayesOpt()
convThresh	Same as bayesOpt()
acqThresh	Same as bayesOpt()
errorHandling	Same as bayesOpt()
saveFile	Same as bayesOpt()
parallel	Same as bayesOpt()
plotProgress	Same as bayesOpt()
verbose	Same as bayesOpt()
...	Same as bayesOpt()

Details

By default, this function uses the original parameters used to create optObj, however the parameters (including the bounds) can be customized. If new bounds are used which cause some of the prior runs to fall outside of the bounds, these samples are removed from the optimization procedure, but will remain in scoreSummary. FUN should return the same elements and accept the same inputs as the original, or this function may fail.

Value

An object of class bayesOpt having run additional iterations.

Examples

```
scoringFunction <- function(x) {
  a <- exp(-(2-x)^2)*1.5
  b <- exp(-(4-x)^2)*2
  c <- exp(-(6-x)^2)*1
  return(list(Score = a+b+c))
}

bounds <- list(x = c(0,8))

Results <- bayesOpt(
    FUN = scoringFunction
  , bounds = bounds
  , initPoints = 3
```

```
  , iters.n = 1
  , gsPoints = 10
)
Results <- addIterations(Results,iters.n=1)
```

---

bayesOpt *Bayesian Optimization with Gaussian Processes*

Description

Maximizes a user defined function within a set of bounds. After the function is sampled a predetermined number of times, a Gaussian process is fit to the results. An acquisition function is then maximized to determine the most likely location of the global maximum of the user defined function. This process is repeated for a set number of iterations.

Usage

```
bayesOpt(
  FUN,
  bounds,
  saveFile = NULL,
  initGrid = NULL,
  initPoints = NULL,
  iters.n = 3,
  iters.k = 1,
  otherHalting = list(timeLimit = Inf, minUtility = 0),
  acq = c("ucb", "ei", "eips", "poi"),
  kappa = 2.576,
  eps = 0,
  parallel = FALSE,
  gsPoints = pmax(100, length(bounds)^3),
  convThresh = 1e+08,
  acqThresh = 1,
  errorHandling = "stop",
  plotProgress = FALSE,
  verbose = 1,
  ...
)
```

Arguments

FUN	the function to be maximized. This function should return a named list with at least 1 component. The first component must be named Score and should contain the metric to be maximized. You may return other named scalar elements that you wish to include in the final summary table.
bounds	named list of lower and upper bounds for each FUN input. The names of the list should be arguments passed to FUN. Use "L" suffix to indicate integers.

`saveFile` character filepath (including file name and extension, .RDS) that specifies the location to save results as they are obtained. A `bayesOpt` object is saved to the file after each epoch.

`initGrid` user specified points to sample the scoring function, should be a `data.frame` or `data.table` with identical column names as bounds. Provide exactly one of `initGrid` and `initPoints`.

`initPoints` Number of points to initialize the process with. Points are chosen with latin hypercube sampling within the bounds supplied. Must be greater than the number of `FUN` inputs. Provide exactly one of `initGrid` and `initPoints`.

`iters.n` The total number of times FUN will be run after initialization.

`iters.k` integer that specifies the number of times to sample FUN at each Epoch (optimization step). If running in parallel, good practice is to set `iters.k` to some multiple of the number of cores you have designated for this process. Must be lower than, and preferrably some multiple of `iters.n`.

`otherHalting` A list of other halting specifications. The process will stop if any of the following is true. These checks are only performed in between optimization steps:

- The elapsed seconds is greater than the list element `timeLimit`.
- The utility expected from the Gaussian process is less than the list element `minUtility`.

`acq` acquisition function type to be used. One of "ucb" (the default), "ei", "eips" or "poi".

- `ucb` Upper Confidence Bound
- `ei` Expected Improvement
- `eips` Expected Improvement Per Second
- `poi` Probability of Improvement

`kappa` tunable parameter kappa of the upper confidence bound. Adjusts exploitation/exploration. Increasing kappa will increase the importance that uncertainty (unexplored space) has, therefore incentivising exploration. This number represents the standard deviations above 0 of your upper confidence bound. Default is 2.56, which corresponds to the ~99th percentile.

`eps` tunable parameter epsilon of ei, eips and poi. Adjusts exploitation/exploration. This value is added to y_max after the scaling, so should between -0.1 and 0.1. Increasing eps will make the "improvement" threshold for new points higher, therefore incentivising exploitation.

`parallel` should the process run in parallel? If TRUE, several criteria must be met:

- A parallel backend must be registered
- Objects required by `FUN` must be loaded into each cluster.
- Packages required by `FUN` must be loaded into each cluster. See vignettes.
- `FUN` must be thread safe.

`gsPoints` integer that specifies how many initial points to try when searching for the optimum of the acquisition function. Increase this for a higher chance to find global optimum, at the expense of more time.

`convThresh` convergence threshold passed to `factr` when the `optim` function (L-BFGS-B) is called. Lower values will take longer to converge, but may be more accurate.

acqThresh	number 0-1. Represents the minimum percentage of the global optimal utility required for a local optimum to be included as a candidate parameter set in the next scoring function. If 1.0, only the global optimum will be used as a candidate parameter set. If 0.5, only local optimums with 50 percent of the utility of the global optimum will be used.
errorHandling	If FUN returns an error, how to proceed. All errors are stored in scoreSummary. Can be one of 3 options: "stop" stops the function running and returns results. "continue" keeps the process running. Passing an integer will allow the process to continue until that many errors have occured, after which the results will be returned.
plotProgress	Should the progress of the Bayesian optimization be printed? Top graph shows the score(s) obtained at each iteration. The bottom graph shows the estimated utility of each point. This is useful to display how much utility the Gaussian Process is assuming still exists. If your utility is approaching 0, then you can be confident you are close to an optimal parameter set.
verbose	Whether or not to print progress to the console. If 0, nothing will be printed. If 1, progress will be printed. If 2, progress and information about new parameter-score pairs will be printed.
...	Other parameters passed to DiceKriging::km(). All FUN inputs and scores are scaled from 0-1 before being passed to km. FUN inputs are scaled within bounds, and scores are scaled by 0 = min(scores), 1 = max(scores).

Value

An object of class bayesOpt containing information about the process.

- FUN The scoring function.
- bounds The bounds originally supplied.
- iters The total iterations that have been run.
- initPars The initialization parameters.
- optPars The optimization parameters.
- GauProList A list containing information on the Gaussian Processes used in optimization.
- scoreSummary A data.table with results from the execution of FUN at different inputs. Includes information on the epoch, iteration, function inputs, score, and any other information returned by FUN.
- stopStatus Information on what caused the function to stop running. Possible explenations are time limit, minimum utility not met, errors in FUN, iters.n was reached, or the Gaussian Process encountered an error.
- elapsedTime The total time in seconds the function was executing.

Vignettes

It is highly recommended to read the GitHub for examples. There are also several vignettes available from the official CRAN Listing.

References

Jasper Snoek, Hugo Larochelle, Ryan P. Adams (2012) *Practical Bayesian Optimization of Machine Learning Algorithms*

Examples

```
# Example 1 - Optimization of a continuous single parameter function
scoringFunction <- function(x) {
  a <- exp(-(2-x)^2)*1.5
  b <- exp(-(4-x)^2)*2
  c <- exp(-(6-x)^2)*1
  return(list(Score = a+b+c))
}

bounds <- list(x = c(0,8))

Results <- bayesOpt(
    FUN = scoringFunction
  , bounds = bounds
  , initPoints = 3
  , iters.n = 2
  , gsPoints = 10
)


# Example 2 - Hyperparameter Tuning in xgboost
if (requireNamespace('xgboost', quietly = TRUE) &&
    utils::packageVersion('xgboost') >= '3.0.0') {
  library("xgboost")

  data(agaricus.train, package = "xgboost")

  Folds <- list(
      Fold1 = as.integer(seq(1,nrow(agaricus.train$data),by = 3))
    , Fold2 = as.integer(seq(2,nrow(agaricus.train$data),by = 3))
    , Fold3 = as.integer(seq(3,nrow(agaricus.train$data),by = 3))
  )

  scoringFunction <- function(max_depth, min_child_weight, subsample) {

    dtrain <- xgb.DMatrix(agaricus.train$data,label = agaricus.train$label)

    Pars <- list(
        booster = "gbtree"
      , eta = 0.01
      , max_depth = max_depth
      , min_child_weight = min_child_weight
      , subsample = subsample
      , objective = "binary:logistic"
      , eval_metric = "auc"
    )
```

```
  xgbcv <- xgb.cv(
       params = Pars
     , data = dtrain
     , nrounds = 100
     , folds = Folds
     , prediction = TRUE
     , showsd = TRUE
     , early_stopping_rounds = 5
     , maximize = TRUE
     , verbose = 0
  )

  return(
    list(
        Score = max(xgbcv$evaluation_log$test_auc_mean)
      , nrounds = xgbcv$early_stop$best_iteration
    )
  )
}

bounds <- list(
    max_depth = c(2L, 10L)
  , min_child_weight = c(1, 100)
  , subsample = c(0.25, 1)
)

ScoreResult <- bayesOpt(
    FUN = scoringFunction
  , bounds = bounds
  , initPoints = 4
  , iters.n = 2
  , iters.k = 1
  , acq = "ei"
  , gsPoints = 10
  , parallel = FALSE
  , verbose = 1
)
}
```

---

changeSaveFile *Change Save File Location*

Description

Use this to change the saveFile parameter in a pre-existing bayesOpt object.

Usage

```
changeSaveFile(optObj, saveFile = NULL)
```

Arguments

optObj	An object of class bayesOpt
saveFile	A filepath stored as a character. Must include the filename and extension as a .RDS.

Value

The same optObj with the updated saveFile.

Examples

```
scoringFunction <- function(x) {
  a <- exp(-(2-x)^2)*1.5
  b <- exp(-(4-x)^2)*2
  c <- exp(-(6-x)^2)*1
  return(list(Score = a+b+c))
}

bounds <- list(x = c(0,8))

Results <- bayesOpt(
    FUN = scoringFunction
  , bounds = bounds
  , initPoints = 3
  , iters.n = 2
  , gsPoints = 10
  , saveFile = tempfile(fileext = ".RDS")
)
Results <- changeSaveFile(Results,saveFile = tempfile(fileext = ".RDS"))
```

---

getBestPars *Get the Best Parameter Set*

Description

Returns the N parameter sets which resulted in the maximum scores from FUN.

Usage

```
getBestPars(optObj, N = 1)
```

Arguments

optObj	An object of class bayesOpt
N	The number of parameter sets to return

Value

A list containing the FUN inputs which resulted in the highest returned Score. If N > 1, a data.table is returned. Each row is a result from FUN, with results ordered by descending Score.

Examples

```
scoringFunction <- function(x) {
  a <- exp(-(2-x)^2)*1.5
  b <- exp(-(4-x)^2)*2
  c <- exp(-(6-x)^2)*1
  return(list(Score = a+b+c))
}

bounds <- list(x = c(0,8))

Results <- bayesOpt(
    FUN = scoringFunction
  , bounds = bounds
  , initPoints = 3
  , iters.n = 2
  , gsPoints = 10
)
print(getBestPars(Results))
```

---

getLocalOptimums *Get Local Optimums of acq From a bayesOpt Object*

Description

Returns all local optimums of the acquisition function, no matter the utility.

Usage

```
getLocalOptimums(
  optObj,
  bounds = optObj$bounds,
  acq = optObj$optPars$acq,
  kappa = optObj$optPars$kappa,
  eps = optObj$optPars$eps,
  convThresh = optObj$optPars$convThresh,
  gsPoints = optObj$optPars$gsPoints,
  parallel = FALSE,
  verbose = 1
)
```

Arguments

optObj	an object of class bayesOpt. The following parameters are all defaulted to the options provided in this object, but can be manually specified.
bounds	Same as in bayesOpt()
acq	Same as in bayesOpt()
kappa	Same as in bayesOpt()
eps	Same as in bayesOpt()
convThresh	Same as in bayesOpt()
gsPoints	Same as in bayesOpt()
parallel	Same as in bayesOpt()
verbose	Should warnings be shown before results are returned prematurely?

Details

gsPoints points in the parameter space are randomly initialized, and the L-BFGS-B method is used to find the closest local optimum to each point. dbscan is then used to cluster points together which converged to the same optimum - only unique optimums are returned.

Value

A data table of local optimums, including the utility (gpUtility), the utility relative to the max utility (relUtility), and the steps taken in the L-BFGS-B method (gradCount).

Examples

```
scoringFunction <- function(x) {
  a <- exp(-(2-x)^2)*1.5
  b <- exp(-(4-x)^2)*2
  c <- exp(-(6-x)^2)*1
  return(list(Score = a+b+c))
}

bounds <- list(x = c(0,8))

Results <- bayesOpt(
    FUN = scoringFunction
  , bounds = bounds
  , initPoints = 3
  , iters.n = 2
  , gsPoints = 10
)
print(getLocalOptimums(Results))
```

`plot.bayesOpt` *Plot a* `bayesOpt` *object*

Description

Returns 2 stacked plots - the top shows the results from FUN at each iteration. The bottom shows the utility from each point before the search took place.

Usage

```
## S3 method for class 'bayesOpt'
plot(x, ...)
```

Arguments

`x`	An object of class bayesOpt
`...`	Passed to `patchwork::plot_layout()`, overriding the defaults `ncol = 1` and `guides = "collect"`.

Value

A `patchwork` object, returned invisibly after printing.

Examples

```
scoringFunction <- function(x) {
  a <- exp(-(2-x)^2)*1.5
  b <- exp(-(4-x)^2)*2
  c <- exp(-(6-x)^2)*1
  return(list(Score = a+b+c))
}

bounds <- list(x = c(0,8))

Results <- bayesOpt(
    FUN = scoringFunction
  , bounds = bounds
  , initPoints = 3
  , iters.n = 2
  , gsPoints = 10
)
# This plot will also show in real time with parameter plotProgress = TRUE in bayesOpt()
plot(Results)
```

print.bayesOpt *Print a* bayesOpt *object*

Description

Print a bayesOpt object

Usage

```
## S3 method for class 'bayesOpt'
print(x, ...)
```

Arguments

x	Object of class bayesOpt
...	required to use S3 method

Value

NULL

updateGP *Update Gaussian Processes in a bayesOpt Object*

Description

To save time, Gaussian processes are not updated after the last iteration in addIterations(). The user can do this manually, using this function if they wish. This is not necessary to continue optimization using addIterations.

Usage

```
updateGP(optObj, bounds = optObj$bounds, verbose = 1, ...)
```

Arguments

optObj	an object of class bayesOpt
bounds	The bounds to scale the parameters within.
verbose	Should the user be warned if the GP is already up to date?
...	passed to DiceKriging::km()

Value

An object of class bayesOpt with updated Gaussian processes.

Examples

```
# Create initial object
scoringFunction <- function(x) {
  a <- exp(-(2-x)^2)*1.5
  b <- exp(-(4-x)^2)*2
  c <- exp(-(6-x)^2)*1
  return(list(Score = a+b+c))
}

bounds <- list(x = c(0,8))

Results <- bayesOpt(
    FUN = scoringFunction
  , bounds = bounds
  , initPoints = 3
  , iters.n = 2
  , gsPoints = 10
)

# At this point, the Gaussian Process has not been updated
# with the most recent results. We can update it manually:
Results <- updateGP(Results)
```

Index

addIterations, 2

bayesOpt, 4

changeSaveFile, 8

getBestPars, 9
getLocalOptimums, 10

plot.bayesOpt, 12
print.bayesOpt, 13

updateGP, 13