Package 'compboost'

October 28, 2018

Type Package

Title C++ Implementation of Component-Wise Boosting

Version 0.1.0

Maintainer Daniel Schalk <daniel.schalk@stat.uni-muenchen.de>

Description C++ implementation of component-wise boosting implementation of component-wise boosting written in C++ to obtain high runtime performance and full memory control. The main idea is to provide a modular class system which can be extended without editing the source code. Therefore, it is possible to use R functions as well as C++ functions for custom base-learners, losses, logging mechanisms or stopping criteria.

License MIT + file LICENSE

Encoding UTF-8

LazyData true

RoxygenNote 6.1.0

Imports Rcpp (>= 0.11.2), methods, glue, R6, checkmate

LinkingTo Rcpp, RcppArmadillo

Suggests RcppArmadillo (>= 0.9.100.5.0), ggplot2, testthat, rpart, mboost, knitr, rmarkdown, titanic, mlr, gridExtra

RcppModules baselearner_module, compboost_module, loss_module, baselearner_module, baselearner_factory_module, baselearner_list_module, logger_module, optimizer_module, data_module

VignetteBuilder knitr

NeedsCompilation yes

Author Daniel Schalk [aut, cre] (<https://orcid.org/0000-0003-0950-1947>), Janek Thomas [aut] (<https://orcid.org/0000-0003-4511-6245>), Bernd Bischl [aut] (<https://orcid.org/0000-0001-6002-6980>)

Repository CRAN

Date/Publication 2018-10-28 19:10:10 UTC

R topics documented:

`BaselearnerCustom` *Create custom base-learner factory by using R functions.*

Description

`BaselearnerCustom` creates a custom base-learner factory by setting custom `R` functions. This factory object can be registered within a base-learner list and then used for training.

Format

`S4` object.

Usage

```
BaselearnerCustom$new(data_source, data_target, instantiateData, train,
  predict, extractParameter)
```

Arguments

`data_source` [`Data` **Object**] Data object which contains the source data.

`data_target` [`Data` **Object**] Data object which gets the transformed source data.

`instantiateData` [`function`] `R` function to transform the source data. For details see the `Details`.

`train` [`function`] `R` function to train the base-learner on the target data. For details see the `Details`.

`predict` [`function`] `R` function to predict on the object returned by `train`. For details see the `Details`.

`extractParameter` [`function`] `R` function to extract the parameter of the object returned by `train`. For details see the `Details`.

Details

The function must have the following structure:

`instantiateData(X) { ... return (X.trafo) }` With a matrix argument `X` and a matrix as return object.

`train(y, X) { ... return (SEXP) }` With a vector argument `y` and a matrix argument `X`. The target data is used in `X` while `y` contains the response. The function can return any `R` object which is stored within a `SEXP`.

`predict(model, newdata) { ... return (prediction) }` The returned object of the `train` function is passed to the `model` argument while `newdata` contains a new matrix used for predicting.

`extractParameter() { ... return (parameters) }` Again, `model` contains the object returned by `train`. The returned object must be a matrix containing the estimated parameter. If no parameter should be estimated one can return `NA`.

For an example see the `Examples`.

This class is a wrapper around the pure `C++` implementation. To see the functionality of the `C++` class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classblearnerfactory_1_1_custom_blearner_factory.html.

Fields

This class doesn't contain public fields.

Methods

`getData()` Get the data matrix of the target data which is used for modeling.

`transformData(X)` Transform a data matrix as defined within the factory. The argument has to be a matrix with one column.

`summarizeFactory()` Summarize the base-learner factory object.

Examples

```
# Sample data:
data.mat = cbind(1, 1:10)
y = 2 + 3 * 1:10

# Create new data object:
data.source = InMemoryData$new(data.mat, "my.data.name")
data.target = InMemoryData$new()

instantiateDataFun = function (X) {
  return(X)
}
# Ordinary least squares estimator:
trainFun = function (y, X) {
  return(solve(t(X) %*% X) %*% t(X) %*% y)
}
predictFun = function (model, newdata) {
  return(as.matrix(newdata %*% model))
}
extractParameter = function (model) {
  return(as.matrix(model))
}

# Create new custom linear base-learner factory:
custom.lin.factory = BaselearnerCustom$new(data.source, data.target,
  instantiateDataFun, trainFun, predictFun, extractParameter)

# Get the transformed data:
custom.lin.factory$getData()

# Summarize factory:
custom.lin.factory$summarizeFactory()

# Transform data manually:
custom.lin.factory$transformData(data.mat)
```

`BaselearnerCustomCpp` *Create custom cpp base-learner factory by using cpp functions and external pointer.*

Description

`BaselearnerCustomCpp` creates a custom base-learner factory by setting custom `C++` functions. This factory object can be registered within a base-learner list and then used for training.

Format

`S4` object.

Usage

```
BaselearnerCustomCpp$new(data_source, data_target, instantiate_data_ptr,
  train_ptr, predict_ptr)
```

Arguments

`data_source` [`Data` **Object**] Data object which contains the source data.

`data_target` [`Data` **Object**] Data object which gets the transformed source data.

`instantiate_data_ptr` [`externalptr`] External pointer to the `C++` instantiate data function.

`train_ptr` [`externalptr`] External pointer to the `C++` train function.

`predict_ptr` [`externalptr`] External pointer to the `C++` predict function.

Details

For an example see the extending compboost vignette or the function `getCustomCppExample`.

This class is a wrapper around the pure `C++` implementation. To see the functionality of the `C++` class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classblearnerfactory_1_1_custom_cpp_blearner_factory.html.

Fields

This class doesn't contain public fields.

Methods

`getData()` Get the data matrix of the target data which is used for modeling.

`transformData(X)` Transform a data matrix as defined within the factory. The argument has to be a matrix with one column.

`summarizeFactory()` Summarize the base-learner factory object.

Examples

```
# Sample data:
data.mat = cbind(1, 1:10)
y = 2 + 3 * 1:10

# Create new data object:
data.source = InMemoryData$new(data.mat, "my.data.name")
data.target = InMemoryData$new()

# Source the external pointer exposed by using XPtr:
Rcpp::sourceCpp(code = getCustomCppExample(silent = TRUE))

# Create new linear base-learner:
custom.cpp.factory = BaselearnerCustomCpp$new(data.source, data.target,
  dataFunSetter(), trainFunSetter(), predictFunSetter())
```

```
# Get the transformed data:
custom.cpp.factory$getData()

# Summarize factory:
custom.cpp.factory$summarizeFactory()

# Transform data manually:
custom.cpp.factory$transformData(data.mat)
```

`BaselearnerPolynomial` *Base-learner factory to make polynomial regression*

Description

`BaselearnerPolynomial` creates a polynomial base-learner factory object which can be registered within a base-learner list and then used for training.

Format

`S4` object.

Usage

```
BaselearnerPolynomial$new(data_source, data_target, degree, intercept)
BaselearnerPolynomial$new(data_source, data_target, blearner_type, degree, intercept)
```

Arguments

`data_source` [`Data` **Object**] Data object which contains the source data.

`data_target` [`Data` **Object**] Data object which gets the transformed source data.

`degree` [`integer(1)`] This argument is used for transforming the source data. Each element is taken to the power of the `degree` argument.

`intercept` [`logical(1)`] Indicating whether an intercept should be added or not. Default is set to TRUE.

Details

The polynomial base-learner factory takes any matrix which the user wants to pass the number of columns indicates how much parameter are estimated. Note that the intercept isn't added by default. To get an intercept add a column of ones to the source data matrix.

This class is a wrapper around the pure `C++` implementation. To see the functionality of the `C++` class visit `https://schalkdaniel.github.io/compboost/cpp_man/html/classblearnerfactory_1_1_polynomial_blearner_factory.html`.

Fields

This class doesn't contain public fields.

Methods

`getData()` Get the data matrix of the target data which is used for modeling.

`transformData(X)` Transform a data matrix as defined within the factory. The argument has to be a matrix with one column.

`summarizeFactory()` Summarize the base-learner factory object.

Examples

```
# Sample data:
data.mat = cbind(1:10)

# Create new data object:
data.source = InMemoryData$new(data.mat, "my.data.name")
data.target1 = InMemoryData$new()
data.target2 = InMemoryData$new()

# Create new linear base-learner factory:
lin.factory = BaselearnerPolynomial$new(data.source, data.target1,
  degree = 2, intercept = FALSE)
lin.factory.int = BaselearnerPolynomial$new(data.source, data.target2,
  degree = 2, intercept = TRUE)

# Get the transformed data:
lin.factory$getData()
lin.factory.int$getData()

# Summarize factory:
lin.factory$summarizeFactory()

# Transform data manually:
lin.factory$transformData(data.mat)
lin.factory.int$transformData(data.mat)
```

BaselearnerPSpline *Base-learner factory to do non-parametric B or P-spline regression*

Description

`BaselearnerPSpline` creates a spline base-learner factory object which can be registered within a base-learner list and then used for training.

Format

S4 object.

Usage

```
BaselearnerPSpline$new(data_source, data_target, degree, n_knots, penalty,
  differences)
```

Arguments

`data_source` [`Data` **Object**] Data object which contains the source data.

`data_target` [`Data` **Object**] Data object which gets the transformed source data.

`degree` [`integer(1)`] Degree of the spline functions to interpolate the knots.

`n_knots` [`integer(1)`] Number of **inner knots**. To prevent weird behavior on the edges the inner knots are expanded by $\text{degree} - 1$ additional knots.

`penalty` [`numeric(1)`] Positive numeric value to specify the penalty parameter. Setting the penalty to 0 ordinary B-splines are used for the fitting.

`differences` [`integer(1)`] The number of differences which are penalized. A higher value leads to smoother curves.

Details

The data matrix of the source data is restricted to have just one column. The spline bases are created for this single feature. Multidimensional splines are not supported at the moment.

This class is a wrapper around the pure `C++` implementation. To see the functionality of the `C++` class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classblearnerfactory_1_1_p_spline_blearner_factory.html.

Fields

This class doesn't contain public fields.

Methods

`getData()` Get the data matrix of the target data which is used for modeling.

`transformData(X)` Transform a data matrix as defined within the factory. The argument has to be a matrix with one column.

`summarizeFactory()` Summarize the base-learner factory object.

Examples

```
# Sample data:
data.mat = cbind(1:10)
y = sin(1:10)

# Create new data object:
data.source = InMemoryData$new(data.mat, "my.data.name")
data.target = InMemoryData$new()

# Create new linear base-learner:
spline.factory = BaselearnerPSpline$new(data.source, data.target,
```

```
  degree = 3, n_knots = 4, penalty = 2, differences = 2)

# Get the transformed data:
spline.factory$getData()

# Summarize factory:
spline.factory$summarizeFactory()

# Transform data manually:
spline.factory$transformData(data.mat)
```

`BlearnerFactoryList` *Base-learner factory list to define the set of base-learners*

Description

`BlearnerFactoryList` creates an object in which base-learner factories can be registered. This object can then be passed to compboost as set of base-learner which is used by the optimizer to get the new best base-learner.

Format

`S4` object.

Usage

```
BlearnerFactoryList$new()
```

Details

This class is a wrapper around the pure `C++` implementation. To see the functionality of the `C++` class visit `https://schalkdaniel.github.io/compboost/cpp_man/html/classblearnerlist_1_1_baselearner_factory_list.html`.

Fields

This class doesn't contain public fields.

Methods

`registerFactory(BaselearnerFactory)` Takes a object of the class `BaseLearnerFactory` and adds this factory to the set of base-learner.

`printRegisteredFactories()` Get all registered factories.

`clearRegisteredFactories()` Remove all registered factories. Note that the factories are not deleted, just removed from the map.

`getModelFrame()` Get each target data matrix parsed to one big matrix.

`getNumberOfRegisteredFactories()` Get the number of registered factories.

Examples

```
# Sample data:
data.mat = cbind(1:10)

# Create new data object:
data.source = InMemoryData$new(data.mat, "my.data.name")
data.target1 = InMemoryData$new()
data.target2 = InMemoryData$new()

lin.factory = BaselearnerPolynomial$new(data.source, data.target1, 1, TRUE)
poly.factory = BaselearnerPolynomial$new(data.source, data.target2, 2, TRUE)

# Create new base-learner list:
my.bl.list = BlearnerFactoryList$new()

# Register factories:
my.bl.list$registerFactory(lin.factory)
my.bl.list$registerFactory(poly.factory)

# Get registered factories:
my.bl.list$printRegisteredFactories()

# Get all target data matrices in one big matrix:
my.bl.list$getModelFrame()

# Clear list:
my.bl.list$clearRegisteredFactories()

# Get number of registered factories:
my.bl.list$getNumberOfRegisteredFactories()
```

---

boostLinear *Wrapper to boost linear models for each feature.*

Description

This wrapper function automatically initializes the model by adding all numerical features of a dataset within a linear base-learner. Categorical features are dummy encoded and inserted using linear base-learners without intercept. After initializing the model `boostLinear` also fits as many iterations as given by the user through `iters`.

Usage

```
boostLinear(data, target, optimizer = OptimizerCoordinateDescent$new(),
  loss, learning.rate = 0.05, iterations = 100, trace = -1,
  intercept = TRUE, data.source = InMemoryData,
  data.target = InMemoryData)
```

Arguments

data	[data.frame] A data frame containing the data on which the model should be built.
target	[character(1)] Character indicating the target variable. Note that the loss must match the data type of the target.
optimizer	[S4 Optimizer] Optimizer to select features. This should be an initialized S4 Optimizer object exposed by Rcpp (for instance OptimizerCoordinateDescent$new()).
loss	[S4 Loss] Loss used to calculate the risk and pseudo residuals. This object must be an initialized S4 Loss object exposed by Rcpp (for instance LossQuadratic$new()).
learning.rate	[numeric(1)] Learning rate which is used to shrink the parameter in each step.
iterations	[integer(1)] Number of iterations that are trained.
trace	[integer(1)] Integer indicating how often a trace should be printed. Specifying trace = 10, then every 10th iteration is printed. If no trace should be printed set trace = 0. Default is -1 which means that we set trace at a value that 40 iterations are printed.
intercept	[logical(1)] Internally used by BaselearnerPolynomial. This logical value indicates if each feature should get an intercept or not (default is TRUE).
data.source	[S4 Data] Uninitialized S4 Data object which is used to store the data. At the moment just in memory training is supported.
data.target	[S4 Data] Uninitialized S4 Data object which is used to store the data. At the moment just in memory training is supported.

Details

The returned object is an object of the Compboost class which then can be used for further analyses (see ?Compboost for details).

Value

Usually a model of class Compboost. This model is an R6 object which can be used for retraining, predicting, plotting, and anything described in ?Compboost.

Examples

```
mod = boostLinear(data = iris, target = "Sepal.Length", loss = LossQuadratic$new())
mod$getBaselearnerNames()
mod$getEstimatedCoef()
```

```
table(mod$getSelectedBaselearner())
mod$predict()
mod$plot("Sepal.Width_linear")
```

`boostSplines` *Wrapper to boost p spline models for each feature.*

Description

This wrapper function automatically initializes the model by adding all numerical features of a dataset within a spline base-learner. Categorical features are dummy encoded and inserted using linear base-learners without intercept. After initializing the model `boostSpline` also fits as many iterations as given by the user through `iters`.

Usage

```
boostSplines(data, target, optimizer = OptimizerCoordinateDescent$new(),
  loss, learning.rate = 0.05, iterations = 100, trace = -1,
  degree = 3, n.knots = 20, penalty = 2, differences = 2,
  data.source = InMemoryData, data.target = InMemoryData)
```

Arguments

`data`	[`data.frame`] A data frame containing the data on which the model should be built.
`target`	[`character(1)`] Character indicating the target variable. Note that the loss must match the data type of the target.
`optimizer`	[`S4 Optimizer`] Optimizer to select features. This should be an initialized `S4 Optimizer` object exposed by Rcpp (for instance `OptimizerCoordinateDescent$new()`).
`loss`	[`S4 Loss`] Loss used to calculate the risk and pseudo residuals. This object must be an initialized `S4 Loss` object exposed by Rcpp (for instance `LossQuadratic$new()`).
`learning.rate`	[`numeric(1)`] Learning rate which is used to shrink the parameter in each step.
`iterations`	[`integer(1)`] Number of iterations that are trained.
`trace`	[`integer(1)`] Integer indicating how often a trace should be printed. Specifying `trace = 10`, then every 10th iteration is printed. If no trace should be printed set `trace = 0`. Default is -1 which means that we set `trace` at a value that 40 iterations are printed.
`degree`	[`integer(1)`] Polynomial degree of the splines used for modeling. Note that the number of parameter increases with the degrees.

`n.knots` [integer(1)]
Number of equidistant "inner knots". The real number of used knots also depends on the polynomial degree.

`penalty` [numeric(1)]
Penalty term for p-splines. If penalty equals 0, then ordinary b-splines are fitted. The higher penalty, the higher the smoothness.

`differences` [integer(1)]
Number of differences that are used for penalization. The higher this value is, the more function values of neighbor knots are forced to be more similar which results in a smoother curve.

`data.source` [S4 Data]
Uninitialized `S4 Data` object which is used to store the data. At the moment just in memory training is supported.

`data.target` [S4 Data]
Uninitialized `S4 Data` object which is used to store the data. At the moment just in memory training is supported.

Details

The returned object is an object of the `Compboost` class which then can be used for further analyses (see `?Compboost` for details).

Value

Usually a model of class `Compboost`. This model is an `R6` object which can be used for retraining, predicting, plotting, and anything described in `?Compboost`.

Examples

```
mod = boostSplines(data = iris, target = "Sepal.Length", loss = LossQuadratic$new())
mod$getBaselearnerNames()
mod$getEstimatedCoef()
table(mod$getSelectedBaselearner())
mod$predict()
mod$plot("Sepal.Width_spline")
```

`Compboost` *Compboost API*

Description

`Compboost` wraps the `S4` class system exposed by `Rcpp` to make defining objects, adding objects, the training and taking predictions, and plotting much easier. As already mentioned, the `Compboost` class is just a wrapper and hence compatible with the most `S4` classes. This together defines the compboost API.

Format

R6Class object.

Usage

```
cboost = Compboost$new(data, target, optimizer = OptimizerCoordinateDescent$new(), loss,
  learning.rate = 0.05)

cboost$addLogger(logger, use.as.stopper = FALSE, logger.id, ...)

cbboost$addBaselearner(features, id, bl.factory, data.source = InMemoryData,
  data.target = InMemoryData, ...)

cbboost$train(iteration = 100, trace = TRUE)

cboost$getCurrentIteration()

cboost$predict(newdata = NULL)

cboost$getInbagRisk()

cboost$getSelectedBaselearner()

cboost$getEstimatedCoef()

cboost$plot(blearner.type = NULL, iters = NULL, from = NULL, to = NULL, length.out = 1000)

cboost$getBaselearnerNames()

cboost$prepareData(newdata)
```

Arguments

For Compboost$new():

`data` [`data.frame`]
: Data used for training.

`target` [`character(1)`]
: Character naming the target. It is necessary that target is available as column in data.

`optimizer` [`S4 Optimizer`]
: Optimizer used for the fitting process given as initialized `S4 Optimizer` class. Default is the `OptimizerCoordinateDescent`.

`loss` [`S4 Loss`]
: Loss as initialized `S4 Loss` which is used to calculate pseudo residuals and the empirical risk. Note that the loss needs match the data type of the target variable. See the details for possible choices.

`learning.rage [numeric(1)]`
: Learning rate used to shrink estimated parameter in each iteration. The learning rate remains constant during the training and has to be between 0 and 1.

For cboost$addLogger():

`logger [S4 Logger]`
: Logger which are registered within a logger list. The objects must be given as uninitialized `S4 Logger` class. See the details for possible choices.

`use.as.stopper [logical(1)]`
: Logical indicating whether the new logger should also be used as stopper. Default value is `FALSE`.

`logger.id [character(1)]`
: Id of the new logger. This is necessary to e.g. register multiple risk logger.

`...`
: Further arguments passed to the constructor of the `S4 Logger` class specified in `logger`. For possible arguments see details or the help pages (e.g. `?LoggerIteration`) of the `S4` classes.

For cboost$addBaselearner():

`features [character()]`
: Vector of column names which are used as input data matrix for a single base-learner. Note that not every base-learner supports the use of multiple features (e.g. the spline base-learner).

`id [character(1)]`
: Id of the base-learners. This is necessary since it is possible to define multiple learners with the same underlying data.

`bl.factory [S4 Factory]`
: Uninitialized base-learner factory represented as `S4 Factory` class. See the details for possible choices.

`data.source [S4 Data]`
: Data source object. At the moment just in memory is supported.

`data.target [S4 Data]`
: Data target object. At the moment just in memory is supported.

`...`
: Further arguments passed to the constructor of the `S4 Factory` class specified in `bl.factory`. For possible arguments see the help pages (e.g. `?BaselearnerPSplineFactory`) of the `S4` classes.

For cboost$train():

`iteration [integer(1)]`
: Set the algorithm at `iteration`. Note: This argument is ignored if this is the first training and an iteration logger is already specified. For further uses the algorithm automatically continues training if `iteration` is set to an value larger than the already trained iterations.

`trace [integer(1)]`
: Integer indicating how often a trace should be printed. Specifying `trace = 10`, then every 10th iteration is printed. If no trace should be printed set `trace = 0`. Default is -1 which means that we set `trace` at a value that 40 iterations are printed.

For cboost$predict():

`newdata [data.frame()]`
Data to predict on. If `NULL` predictions on the training data are returned.

For cboost$plot():

`blearner.type [character(1)]`
Character name of the base-learner to plot the additional contribution to the response.

`iters [integer()]`
Integer vector containing the iterations the user wants to illustrate.

`from [numeric(1)]`
Lower bound for plotting (should be smaller than `to`).

`to [numeric(1)]`
Upper bound for plotting (should be greater than `from`).

`length.out [integer(1)]`
Number of equidistant points between `from` and `to` used for plotting.

Details

Loss
Available choices for the loss are:

- `LossQuadratic` (Regression)
- `LossAbsolute` (Regression)
- `LossBinomial` (Binary Classification)
- `LossCustom` (Custom)
- `LossCustomCpp` (Custom)

(For each loss also take a look at the help pages (e.g. `?LossBinomial`) and the `C++` documentation for details about the underlying formulas)

Logger
Available choices for the logger are:

- `LoggerIteration`: Log current iteration. Additional arguments:
  `max_iterations` **[**`integer(1) ]` Maximal number of iterations.
- `LoggerTime`: Log already elapsed time. Additional arguments:
  `max_time` **[**`integer(1) ]` Maximal time for the computation.
  `time_unit` **[**`character(1) ]` Character to specify the time unit. Possible choices are `minutes`, `seconds`, or `microseconds`.
- `LoggerInbagRisk`:
  `used_loss` **[**`S4 Loss ]` Loss as initialized `S4 Loss` which is used to calculate the empirical risk. See the details for possible choices.
  `eps_for_break` **[**`numeric(1) ]` This argument is used if the logger is also used as stopper. If the relative improvement of the logged inbag risk falls above this boundary the stopper breaks the algorithm.

- `LoggerOobRisk`:

  `used_loss` **[**`S4 Loss` **]** Loss as initialized `S4 Loss` which is used to calculate the empirical risk. See the details for possible choices.

  `eps_for_break` **[**`numeric(1)` **]** This argument is used if the logger is also used as stopper. If the relative improvement of the logged inbag risk falls above this boundary the stopper breaks the algorithm.

  `oob_data` **[**`list` **]** A list which contains data source objects which corresponds to the source data of each registered factory. The source data objects should contain the out of bag data. This data is then used to calculate the new predictions in each iteration.

  `oob_response` **[**`vector` **]** Vector which contains the response for the out of bag data given within `oob_data`.

Note:

- Even if you do not use the logger as stopper you have to define the arguments such as `max_time`.
- We are aware of that the style guide here is not consistent with the `R6` arguments. Nevertheless, using _ as word separator is due to the used arguments within `C++`.

Fields

`data` **[**`data.frame` **]** Data used for training the algorithm.

`response` **[**`vector` **]** Response given as vector.

`target` **[**`character(1)` **]** Name of the Response.

`id` **[**`character(1)` **]** Value to identify the data. By default name of `data`, but can be overwritten.

`optimizer` **[**`S4 Optimizer` **]** Optimizer used within the fitting process.

`loss` **[**`S4 Loss` **]** Loss used to calculate pseudo residuals and empirical risk.

`learning.rate` **[**`numeric(1)` **]** Learning rate used to shrink the estimated parameter in each iteration.

`model` **[**`S4 Compboost_internal` **]** Internal `S4 Compboost_internal` class on which the main operations are called.

`bl.factory.list` **[**`S4 FactoryList` **]** List of all registered factories represented as `S4 FactoryList` class.

`positive.category` **[**`character(1)` **]** Character containing the name of the positive class in the case of classification.

`stop.if.all.stoppers.fulfilled` **[**`logical(1)` **]** Logical indicating whether all stopper should be used simultaneously or if it is sufficient that the first stopper which is fulfilled breaks the algorithm.

Methods

`addLogger` method to add a logger to the algorithm (Note: This is just possible before the training).

`addBaselearner` method to add a new base-learner factories to the algorithm (Note: This is just possible before the training).

`getCurrentIteration` method to get the current iteration on which the algorithm is set.

`train` method to train the algorithm.

`predict` method to predict on a trained object.

`getSelectedBaselearner` method to get a character vector of selected base-learner.

`getEstimatedCoef` method to get a list of estimated coefficient for each selected base-learner.

`plot` method to plot the `Compboost` object.

`getBaselearnerNames` method to get names of registered factories.

Examples

```
cboost = Compboost$new(mtcars, "mpg", loss = LossQuadratic$new())
cboost$addBaselearner("hp", "spline", BaselearnerPSpline, degree = 3,
  n.knots = 10, penalty = 2, differences = 2)
cboost$train(1000)

table(cboost$getSelectedBaselearner())
cboost$plot("hp_spline")
```

`Compboost_internal` *Main Compboost Class*

Description

This class collects all parts such as the factory list or the used logger and passes them to `C++`. On the `C++` side is then the main algorithm.

Format

`S4` object.

Usage

```
Compboost$new(response, learning_rate, stop_if_all_stopper_fulfilled,
  factory_list, loss, logger_list, optimizer)
```

Arguments

`response` [`numeric`] Vector of the true values which should be modeled.

`learning_rate` [`numeric(1)`] The learning rate which is used to shrink the parameter in each iteration.

`stop_if_all_stopper_fulfilled` [`logical(1)`] Boolean to indicate which stopping strategy is used. If `TRUE` then the algorithm stops if all registered logger stopper are fulfilled.

`factory_list` [`BlearnerFactoryList` **object**] List of base-learner factories from which one base-learner is selected in each iteration by using the

`loss` [`Loss` **object**] The loss which should be used to calculate the pseudo residuals in each iteration.

`logger_list` [`LoggerList` **object**] The list with all registered logger which are used to track the algorithm.

`optimizer` [`Optimizer` **object**] The optimizer which is used to select in each iteration one good base-learner.

Details

This class is a wrapper around the pure `C++` implementation. To see the functionality of the `C++` class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classcboost_1_1_compboost.html.

Fields

This class doesn't contain public fields.

Methods

`train(trace)` Initial training of the model. The integer argument `trace` indicates if the logger progress should be printed or not and if so trace indicates which iterations should be printed.

`continueTraining(trace, logger_list)` Continue the training by using an additional `logger_list`. The retraining is stopped if the first logger says that the algorithm should be stopped.

`getPrediction()` Get the inbag prediction which is done during the fitting process.

`getSelectedBaselearner()` Returns a character vector of how the base-learner are selected.

`getLoggerData()` Returns a list of all logged data. If the algorithm is retrained, then the list contains for each training one element.

`getEstimatedParameter()` Returns a list with the estimated parameter for base-learner which was selected at least once.

`getParameterAtIteration(k)` Calculates the prediction at the iteration `k`.

`getParameterMatrix()` Calculates a matrix where row `i` includes the parameter at iteration `i`. There are as many rows as done iterations.

`isTrained()` This function returns just a boolean value which indicates if the initial training was already done.

`predict(newdata)` Prediction on new data organized within a list of source data objects. It is important that the names of the source data objects matches those one that were used to define the factories.

`predictAtIteration(newdata, k)` Prediction on new data by using another iteration `k`.

`setToIteration(k)` Set the whole model to another iteration `k`. After calling this function all other elements such as the parameters or the prediction are calculated corresponding to `k`.

`summarizeCompboost()` Summarize the `Compboost` object.

Examples

```
# Some data:
df = mtcars
df$mpg.cat = ifelse(df$mpg > 20, 1, -1)
```

```
# # Create new variable to check the polynomial base-learner with degree 2:
# df$hp2 = df[["hp"]]^2

# Data for the baselearner are matrices:
X.hp = as.matrix(df[["hp"]])
X.wt = as.matrix(df[["wt"]])

# Target variable:
y = df[["mpg.cat"]]

data.source.hp = InMemoryData$new(X.hp, "hp")
data.source.wt = InMemoryData$new(X.wt, "wt")

data.target.hp1 = InMemoryData$new()
data.target.hp2 = InMemoryData$new()
data.target.wt1 = InMemoryData$new()
data.target.wt2 = InMemoryData$new()

# List for oob logging:
oob.data = list(data.source.hp, data.source.wt)

# List to test prediction on newdata:
test.data = oob.data

# Factories:
linear.factory.hp = BaselearnerPolynomial$new(data.source.hp, data.target.hp1, 1, TRUE)
linear.factory.wt = BaselearnerPolynomial$new(data.source.wt, data.target.wt1, 1, TRUE)
quadratic.factory.hp = BaselearnerPolynomial$new(data.source.hp, data.target.hp2, 2, TRUE)
spline.factory.wt = BaselearnerPSpline$new(data.source.wt, data.target.wt2, 3, 10, 2, 2)

# Create new factory list:
factory.list = BlearnerFactoryList$new()

# Register factories:
factory.list$registerFactory(linear.factory.hp)
factory.list$registerFactory(linear.factory.wt)
factory.list$registerFactory(quadratic.factory.hp)
factory.list$registerFactory(spline.factory.wt)

# Define loss:
loss.bin = LossBinomial$new()

# Define optimizer:
optimizer = OptimizerCoordinateDescent$new()

## Logger

# Define logger. We want just the iterations as stopper but also track the
# time, inbag risk and oob risk:
log.iterations  = LoggerIteration$new(TRUE, 500)
log.time        = LoggerTime$new(FALSE, 500, "microseconds")
log.inbag       = LoggerInbagRisk$new(FALSE, loss.bin, 0.05)
```

```
log.oob         = LoggerOobRisk$new(FALSE, loss.bin, 0.05, oob.data, y)

# Define new logger list:
logger.list = LoggerList$new()

# Register the logger:
logger.list$registerLogger(" iteration.logger", log.iterations)
logger.list$registerLogger("time.logger", log.time)
logger.list$registerLogger("inbag.binomial", log.inbag)
logger.list$registerLogger("oob.binomial", log.oob)

# Run compboost:
# --------------

# Initialize object:
cboost = Compboost_internal$new(
  response      = y,
  learning_rate = 0.05,
  stop_if_all_stopper_fulfilled = FALSE,
  factory_list = factory.list,
  loss         = loss.bin,
  logger_list  = logger.list,
  optimizer    = optimizer
)

# Train the model (we want to print the trace):
cboost$train(trace = 50)
cboost

# Get estimated parameter:
cboost$getEstimatedParameter()

# Get trace of selected base-learner:
cboost$getSelectedBaselearner()

# Set to iteration 200:
cboost$setToIteration(200)

# Get new parameter values:
cboost$getEstimatedParameter()
```

`getCustomCppExample` *Get example C++ script to define a custom cpp logger*

Description

This function can be used to print the trace of the parameters of a trained compboost object.

Usage

```
getCustomCppExample(example = "blearner", silent = FALSE)
```

Arguments

example	[character(1)] Character value indicating if an example for the base-learner or for the loss should be returned. The values, for example, has to be one of blearner or loss.
silent	[logical(1)] Logical value indicating if the example code should be printed to the screen.

Value

This function returns a string which can be compiled using Rcpp::sourceCpp(code = getCustomCppExample) to define a new custom cpp logger.

InMemoryData *In memory data class to store data in RAM*

Description

InMemoryData creates an data object which can be used as source or target object within the base-learner factories of compboost. The convention to initialize target data is to call the constructor without any arguments.

Format

S4 object.

Usage

```
InMemoryData$new()
InMemoryData$new(data.mat, data.identifier)
```

Arguments

data.mat [matrix] Matrix containing the source data. This source data is later transformed to obtain the design matrix a base-learner uses for training.

data.identifier [character(1)] The name for the data specified in data.mat. Note that it is important to have the same data names for train and evaluation data.

Details

The data.mat needs to suits the base-learner. For instance, the spline base-learner does just take a one column matrix since there are just one dimensional splines till now. Additionally, using the polynomial base-learner the data.mat is used to control if a intercept should be fitted or not by adding a column containing just ones. It is also possible to add other columns to estimate multiple features simultaneously. Anyway, this is not recommended in terms of unbiased features selection.

The data.mat and data.identifier of a target data object is set automatically by passing the source and target object to the desired factory. getData() can then be used to access the transformed data of the target object.

This class is a wrapper around the pure C++ implementation. To see the functionality of the C++ class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classdata_1_1_in_memory_data.html.

Fields

This class doesn't contain public fields.

Methods

getData() method extract the data.mat from the data object.

getIdentifier() method to extract the used name from the data object.

Examples

```
# Sample data:
data.mat = cbind(1:10)

# Create new data object:
data.obj = InMemoryData$new(data.mat, "my.data.name")

# Get data and identifier:
data.obj$getData()
data.obj$getIdentifier()
```

LoggerInbagRisk *Logger class to log the inbag risk*

Description

This class logs the inbag risk for a specific loss function. It is also possible to use custom losses to log performance measures. For details see the use case or extending compboost vignette.

Format

S4 object.

Usage

```
LoggerInbagRisk$new(use_as_stopper, used_loss, eps_for_break)
```

Arguments

`use_as_stopper` [`logical(1)`] Boolean to indicate if the logger should also be used as stopper.

`used_loss` [`Loss` **object**] The loss used to calculate the empirical risk by taking the mean of the returned defined loss within the loss object.

`eps_for_break` [`numeric(1)`] This argument is used if the loss is also used as stopper. If the relative improvement of the logged inbag risk falls above this boundary the stopper returns `TRUE`.

Details

This logger computes the risk for the given training data $\mathcal{D} = \{(x^{(i)},\ y^{(i)}) \mid i \in \{1, \ldots, n\}\}$ and stores it into a vector. The empirical risk $\mathcal{R}$ for iteration m is calculated by:

$$\mathcal{R}_{\text{emp}}^{[m]} = \frac{1}{n} \sum_{i=1}^{n} L(y^{(i)}, \hat{f}^{[m]}(x^{(i)}))$$

Note:

- If $m = 0$ than $\hat{f}$ is just the offset.
- The implementation to calculate $\mathcal{R}_{\text{emp}}^{[m]}$ is done in two steps:
  1. Calculate vector `risk_temp` of losses for every observation for given response $y^{(i)}$ and prediction $\hat{f}^{[m]}(x^{(i)})$.
  2. Average over `risk_temp`.

This procedure ensures, that it is possible to e.g. use the AUC or any arbitrary performance measure for risk logging. This gives just one value for `risk_temp` and therefore the average equals the loss function. If this is just a value (like for the AUC) then the value is returned.

This class is a wrapper around the pure `C++` implementation. To see the functionality of the `C++` class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classlogger_1_1_inbag_risk_logger.html.

Fields

This class doesn't contain public fields.

Methods

`summarizeLogger()` Summarize the logger object.

Examples

```
# Used loss:
log.bin = LossBinomial$new()

# Define logger:
log.inbag.risk = LoggerInbagRisk$new(FALSE, log.bin, 0.05)

# Summarize logger:
log.inbag.risk$summarizeLogger()
```

LoggerIteration *Logger class to log the current iteration*

Description

This class seems to be useless, but it gives more control about the algorithm and doesn't violate the idea of object programming here. Additionally, it is quite convenient to have this class instead of tracking the iteration at any stage of the fitting within the compboost object as another vector.

Format

S4 object.

Usage

```
LoggerIterationWrapper$new(use_as_stopper, max_iterations)
```

Arguments

`use_as_stopper` [`logical(1)`] Boolean to indicate if the logger should also be used as stopper.

`max_iterations` [`integer(1)`] If the logger is used as stopper this argument defines the maximal iterations.

Details

This class is a wrapper around the pure C++ implementation. To see the functionality of the C++ class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classlogger_1_1_iteration_logger.html.

Fields

This class doesn't contain public fields.

Methods

`summarizeLogger()` Summarize the logger object.

Examples

```
# Define logger:
log.iters = LoggerIteration$new(FALSE, 100)

# Summarize logger:
log.iters$summarizeLogger()
```

`LoggerList` *Logger list class to collect all loggers*

Description

This class is meant to define all logger which should be used to track the progress of the algorithm.

Format

S4 object.

Usage

```
LoggerList$new()
```

Details

This class is a wrapper around the pure C++ implementation. To see the functionality of the C++ class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classloggerlist_1_1_logger_list.html.

Fields

This class doesn't contain public fields.

Methods

`clearRegisteredLogger()` Removes all registered logger from the list. The used logger are not deleted, just removed from the map.

`getNamesOfRegisteredLogger()` Returns the registered logger names as character vector.

`getNumberOfRegisteredLogger()` Returns the number of registered logger as integer.

`printRegisteredLogger()` Prints all registered logger.

`registerLogger(logger.id, logger)` Includes a new `logger` into the logger list with the `logger.id` as key.

Examples

```
# Define logger:
log.iters = LoggerIteration$new(TRUE, 100)
log.time = LoggerTime$new(FALSE, 20, "minutes")

# Create logger list:
logger.list = LoggerList$new()

# Register new loggeR:
logger.list$registerLogger("iteration", log.iters)
logger.list$registerLogger("time", log.time)

# Print registered logger:
logger.list$printRegisteredLogger()

# Important: The keys has to be unique:
logger.list$registerLogger("iteration", log.iters)

# Still just two logger:
logger.list$printRegisteredLogger()

# Remove all logger:
logger.list$clearRegisteredLogger()

# Get number of registered logger:
logger.list$getNumberOfRegisteredLogger()
```

LoggerOobRisk *Logger class to log the out of bag risk*

Description

This class logs the out of bag risk for a specific loss function. It is also possible to use custom losses to log performance measures. For details see the use case or extending compboost vignette.

Format

S4 object.

Usage

```
LoggerOobRisk$new(use_as_stopper, used_loss, eps_for_break, oob_data,
  oob_response)
```

Arguments

`use_as_stopper` [`logical(1)`] Boolean to indicate if the logger should also be used as stopper.

`used_loss` [`Loss` **object**] The loss used to calculate the empirical risk by taking the mean of the returned defined loss within the loss object.

`eps_for_break` [`numeric(1)`] This argument is used if the loss is also used as stopper. If the relative improvement of the logged inbag risk falls above this boundary the stopper returns `TRUE`.

`oob_data` [`list`] A list which contains data source objects which corresponds to the source data of each registered factory. The source data objects should contain the out of bag data. This data is then used to calculate the prediction in each step.

`oob_response` [`numeric`] Vector which contains the response for the out of bag data given within the `list`.

Details

This logger computes the risk for a given new dataset $\mathcal{D}_{\text{oob}} = \{(x^{(i)},\ y^{(i)}) \mid i \in I_{\text{oob}}\}$ and stores it into a vector. The OOB risk $\mathcal{R}_{\text{oob}}$ for iteration m is calculated by:

$$\mathcal{R}_{\text{oob}}^{[m]} = \frac{1}{|\mathcal{D}_{\text{oob}}|} \sum_{(x,y) \in \mathcal{D}_{\text{oob}}} L(y, \hat{f}^{[m]}(x))$$

Note:

- If $m = 0$ than $\hat{f}$ is just the offset.
- The implementation to calculate $\mathcal{R}_{\text{emp}}^{[m]}$ is done in two steps:
  1. Calculate vector `risk_temp` of losses for every observation for given response $y^{(i)}$ and prediction $\hat{f}^{[m]}(x^{(i)})$.
  2. Average over `risk_temp`.

This procedure ensures, that it is possible to e.g. use the AUC or any arbitrary performance measure for risk logging. This gives just one value for $risk_temp$ and therefore the average equals the loss function. If this is just a value (like for the AUC) then the value is returned.

This class is a wrapper around the pure `C++` implementation. To see the functionality of the `C++` class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classlogger_1_1_oob_risk_logger.html.

Fields

This class doesn't contain public fields.

Methods

`summarizeLogger()` Summarize the logger object.

Examples

```
# Define data:
X1 = cbind(1:10)
X2 = cbind(10:1)
data.source1 = InMemoryData$new(X1, "x1")
data.source2 = InMemoryData$new(X2, "x2")

oob.list = list(data.source1, data.source2)

set.seed(123)
y.oob = rnorm(10)

# Used loss:
log.bin = LossBinomial$new()

# Define logger:
log.oob.risk = LoggerOobRisk$new(FALSE, log.bin, 0.05, oob.list, y.oob)

# Summarize logger:
log.oob.risk$summarizeLogger()
```

`LoggerTime` *Logger class to log the elapsed time*

Description

This class just logs the elapsed time. This should be very handy if one wants to run the algorithm for just 2 hours and see how far he comes within that time. There are three time units available for logging:

- minutes
- seconds
- microseconds

Format

S4 object.

Usage

```
LoggerTime$new(use_as_stopper, max_time, time_unit)
```

Arguments

`use_as_stopper` [`logical(1)`] Boolean to indicate if the logger should also be used as stopper.

`max_time` [`integer(1)`] If the logger is used as stopper this argument contains the maximal time which are available to train the model.

`time_unit` [`character(1)`] Character to specify the time unit. Possible choices are `minutes`, `seconds` or `microseconds`

Details

This class is a wrapper around the pure C++ implementation. To see the functionality of the C++ class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classlogger_1_1_time_logger.html.

Fields

This class doesn't contain public fields.

Methods

`summarizeLogger()` Summarize the logger object.

Examples

```
# Define logger:
log.time = LoggerTime$new(FALSE, 20, "minutes")

# Summarize logger:
log.time$summarizeLogger()
```

LossAbsolute *Absolute loss for regression tasks.*

Description

This loss can be used for regression with $y \in \mathrm{R}$.

Format

S4 object.

Details

Loss Function:

$$L(y, f(x)) = |y - f(x)|$$

Gradient:

$$\frac{\delta}{\delta f(x)} L(y, f(x)) = \text{sign}(f(x) - y)$$

Initialization:

$$\hat{f}^{[0]}(x) = \arg\min_{c \in R} \frac{1}{n} \sum_{i=1}^{n} L(y^{(i)}, c) = \text{median}(y)$$

Usage

```
LossAbsolute$new()
LossAbsolute$new(offset)
```

Arguments

`offset` [`numeric(1)`] Numerical value which can be used to set a custom offset. If so, this value is returned instead of the loss optimal initialization.

Details

This class is a wrapper around the pure `C++` implementation. To see the functionality of the `C++` class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classloss_1_1_absolute_loss.html.

Examples

```
# Create new loss object:
absolute.loss = LossAbsolute$new()
absolute.loss
```

`LossBinomial` *0-1 Loss for binary classification derived of the binomial distribution*

Description

This loss can be used for binary classification. The coding we have chosen here acts on $y \in \{-1, 1\}$.

Format

S4 object.

Details

Loss Function:

$$L(y, f(x)) = \log(1 + \exp(-2yf(x)))$$

Gradient:

$$\frac{\delta}{\delta f(x)} L(y, f(x)) = -\frac{y}{1 + \exp(2yf)}$$

Initialization:

$$\hat{f}^{[0]}(x) = \frac{1}{2}\log(p/(1-p))$$

with

$$p = \frac{1}{n}\sum_{i=1}^{n} 1_{\{y^{(i)}=1\}}$$

Usage

```
LossBinomial$new()
LossBinomial$new(offset)
```

Arguments

`offset` [`numeric(1)`] Numerical value which can be used to set a custom offset. If so, this value is returned instead of the loss optimal initialization.

Details

This class is a wrapper around the pure `C++` implementation. To see the functionality of the `C++` class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classloss_1_1_binomial_loss.html.

Examples

```
# Create new loss object:
bin.loss = LossBinomial$new()
bin.loss
```

`LossCustom` *Create LossCustom by using R functions.*

Description

`LossCustom` creates a custom loss by using `Rcpp::Function` to set `R` functions.

Format

`S4` object.

Usage

```
LossCustom$new(lossFun, gradientFun, initFun)
```

Arguments

`lossFun` [`function`] `R` function to calculate the loss. For details see the `Details`.

`gradientFun` [`function`] `R` function to calculate the gradient. For details see the `Details`.

`initFun` [`function`] `R` function to calculate the constant initialization. For details see the `Details`.

Details

The functions must have the following structure:

`lossFun(truth, prediction) { ... return (loss) }` With a vector argument `truth` containing the real values and a vector of predictions `prediction`. The function must return a vector containing the loss for each component.

`gradientFun(truth, prediction) { ... return (grad) }` With a vector argument `truth` containing the real values and a vector of predictions `prediction`. The function must return a vector containing the gradient of the loss for each component.

`initFun(truth) { ... return (init) }` With a vector argument `truth` containing the real values. The function must return a numeric value containing the offset for the constant initialization.

For an example see the `Examples`.

This class is a wrapper around the pure `C++` implementation. To see the functionality of the `C++` class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classloss_1_1_custom_loss.html.

Examples

```
# Loss function:
myLoss = function (true.values, prediction) {
  return (0.5 * (true.values - prediction)^2)
}
# Gradient of loss function:
myGradient = function (true.values, prediction) {
  return (prediction - true.values)
}
# Constant initialization:
myConstInit = function (true.values) {
  return (mean(true.values))
}

# Create new custom quadratic loss:
my.loss = LossCustom$new(myLoss, myGradient, myConstInit)
```

`LossCustomCpp` *Create custom cpp losses by using cpp functions and external pointer.*

Description

`LossCustomCpp` creates a custom loss by using `Rcpp::XPtr` to set `C++` functions.

Format

`S4` object.

Usage

```
LossCustomCpp$new(loss_ptr, grad_ptr, const_init_ptr)
```

Arguments

`loss_ptr` [`externalptr`] External pointer to the C++ loss function.

`grad_ptr` [`externalptr`] External pointer to the C++ gradient function.

`const_init_ptr` [`externalptr`] External pointer to the C++ constant initialization function.

Details

For an example see the extending compboost vignette or the function `getCustomCppExample(example = "loss")`.

This class is a wrapper around the pure C++ implementation. To see the functionality of the C++ class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classloss_1_1_custom_cpp_loss.html.

Examples

```
# Load loss functions:
Rcpp::sourceCpp(code = getCustomCppExample(example = "loss", silent = TRUE))

# Create new custom quadratic loss:
my.cpp.loss = LossCustomCpp$new(lossFunSetter(), gradFunSetter(), constInitFunSetter())
```

LossQuadratic *Quadratic loss for regression tasks.*

Description

This loss can be used for regression with $y \in \mathrm{R}$.

Format

S4 object.

Details

Loss Function:

$$L(y, f(x)) = \frac{1}{2}(y - f(x))^2$$

Gradient:

$$\frac{\delta}{\delta f(x)} L(y, f(x)) = f(x) - y$$

Initialization:

$$\hat{f}^{[0]}(x) = \arg\min c \in \mathrm{R} \arg\min \frac{1}{n} \sum_{i=1}^{n} L\left(y^{(i)}, c\right) = \bar{y}$$

Usage

```
LossQuadratic$new()
LossQuadratic$new(offset)
```

Arguments

`offset` [`numeric(1)`] Numerical value which can be used to set a custom offset. If so, this value is returned instead of the loss optimal initialization.

Details

This class is a wrapper around the pure C++ implementation. To see the functionality of the C++ class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classloss_1_1_quadratic_loss.html.

Examples

```
# Create new loss object:
quadratic.loss = LossQuadratic$new()
quadratic.loss
```

OptimizerCoordinateDescent

Greedy Optimizer

Description

This class defines a new object for the greedy optimizer. The optimizer just calculates for each base-learner the sum of squared errors and returns the base-learner with the smallest SSE.

Format

S4 object.

Usage

```
OptimizerCoordinateDescent$new()
```

Details

This class is a wrapper around the pure C++ implementation. To see the functionality of the C++ class visit https://schalkdaniel.github.io/compboost/cpp_man/html/classoptimizer_1_1_greedy_optimizer.html.

Examples

```
# Define optimizer:
optimizer = OptimizerCoordinateDescent$new()
```

Index