Package ‘boostr’

February 19, 2015

Title A modular framework to bag or boost any estimation procedure.

Description boostr provides a modular framework that return the focus of ensemble learning back to 'learning' (instead of programming).

Version 1.0.0

Author Steven Pollack <steven@pollackphoto.net>

Maintainer Steven Pollack <steven@pollackphoto.net>

Depends R (>= 3.0.2)

Imports foreach, iterators, stringr

Suggests knitr, xtable, mlbench, nnet, class, e1071, randomForest

License GPL-2

LazyData true

VignetteBuilder knitr

NeedsCompilation no

Repository CRAN

Date/Publication 2014-05-17 11:02:44

R topics documented:

`adaboostAggregator` *Aggregator for the Adaboost.M1 algorithm*

Description

Implements a (parallelized) version of the aggregator described in the Adaboost.M1 algorithm.

Usage

```
adaboostAggregator(estimators, alpha, ..., .parallelPredict = FALSE)
```

Arguments

`estimators`	a list of estimators which must produce output in the same response-space. This is usually the output of some reweighter function.
`...`	this does nothing – meant to swallow auxillary output from reweighter function.
`.parallelPredict`	a boolean indicating if prediction should be carried out in parallel.
`alpha`	a vector (or list) of length equal to the length of `estimators`. Each entry of `alpha` acts as a prediction weight for the corresponding estimator.

Value

a function whose sole argument is `newdata` and whose output is the aggregated predictions of the boosted ensemble, `estimators`.

For internal bookkeeping, this function is inherits from the '`aggregator`' class.

See Also

Other adaboost: `adaboostReweighter`

Other aggregators: `arcfsAggregator`; `arcx4Aggregator`, `vanillaAggregator`, `weightedAggregator`; `boost`, `boost.function`, `boost.list`

`adaboostReweighter` *Reweighter function for the Adaboost.M1 algorithm*

Description

Implements a slightly modified version of the reweighter described in the Adaboost.M1 algorithm.

Usage

```
adaboostReweighter(prediction, response, weights, ...)
```

Arguments

`prediction`	a vector of predictions.
`response`	a vector whose i^{th} component is the true response for the i^{th} component of `prediction`.
`weights`	a vector of weights. They don't necessarily need to sum to 1.
`...`	implemented to allow reweighter to accept its output as its input.

Details

The modification of the reweighter comes in to play when

$$\epsilon = 0$$

. This is when the esimator correctly classifies every observation in the learning set. Consequently, we're supposed to define $\alpha = log\left(\frac{1-\epsilon}{\epsilon}\right)$ However, this is $+\infty$, which is not a number R is used to working with. To work around this, we have to create a conditional statement that sets `alpha <- log(.Machine$double.xmax)` and let the algorithm proceed as originally described. The effect of this modification is the following:

1. the update that's supposed to be made to `weights`, which is a function of `alpha`, effectively keeps `weights` as they were before.
2. if you pair this reweighter with `adaboostAggregator` then the estimator associated to this very large `alpha` now has tremendous weight inside the weighted sum in the aggregator. This isn't, necessarily, a bad thing – the estimator classified every observation in `data` correctly.

Value

For internal bookkeeping, this function is inherits from the '`reweighter`' class. It returns a named list with components

`weights`	the updated weights calculated from the input weights, `prediction` and `response`.
`alpha`	performance measure of `estimator` to be used by `adaboostAggregator`.

See Also

Other adaboost: adaboostAggregator

Other reweighters: arcfsReweighter; arcx4Reweighter; boost, boost.function, boost.list; vanillaBagger

---

addDots *Extend a function's signature to include '...'*

Description

modifies the input function to accept extra arguments, if it doesn't already.

Usage

```
addDots(func, .verbose = FALSE)
```

Arguments

`func`	the function whose signature is to be modified.
`.verbose`	logical flag indicating whether warnings should be displayed or not.

Value

a function with the same body as `func` but whose signature now includes `...`.

Examples

```
## Not run:
f <- function(x,y) x^2 + y^2

g <- addDots(f)
g

h <- addDots(g, .verbose=TRUE)

## End(Not run)
```

`arcfsAggregator` *Aggregator for the arc-fs algorithm.*

Description

A (parallelized) implementation of the aggregator described in the arc-fs algorithm.

Usage

```
arcfsAggregator(estimators, beta, ..., .parallelPredict = FALSE,
  .parallelTally = FALSE, .rngSeed = 1234)
```

Arguments

`estimators`	a list of estimators which must produce output in the same response-space. This is usually the output of some reweighter function.
`...`	this does nothing – meant to swallow auxillary output from reweighter function.
`.parallelPredict`	a boolean indicating if prediction should be carried out in parallel.
`beta`	a vector of scalar weights associated to each estimator in `estimators`
`.parallelTally`	a boolean indicating if vote tallying should be performed in parallel. Unless you have more than 1,000 votes / observation, you probably won't see much performance gain by parallelizing this step.
`.rngSeed`	the RNG seed sent to `predictClassFromWeightedVote` in the case of a tie.

Details

By default, this function will perform its predictions in sequence across the estimators in `estimators`. To predict in parallel, change `.parallelPredict` to `TRUE`.

Value

a function whose sole argument is `newdata` and whose output is the aggregated predictions of the boosted ensemble, `estimators`.

For internal bookkeeping, this function is inherits from the '`aggregator`' class.

Note

In accord with the arc-fs algorithm, there is the assumption that the estimators in `estimators` are classifiers. More aptly, their output is either of factor or character-type.

See Also

Other aggregators: `adaboostAggregator`; `arcx4Aggregator`, `vanillaAggregator`, `weightedAggregator`; `boost`, `boost.function`, `boost.list`

Other arc-fs: `arcfsReweighter`

arcfsReweighter *Reweighter for the arc-fs algorithm.*

Description

A slightly modified implementation of the reweighter described in the arc-fs algorithm.

Usage

```
arcfsReweighter(prediction, response, weights, ...)
```

Arguments

`prediction`	a vector of predictions.
`response`	a vector whose i^{th} component is the true response for the i^{th} component of `prediction`.
`weights`	a vector of weights. They don't necessarily need to sum to 1.
`...`	implemented to allow reweighter to accept its output as its input.

Details

As per Leo Breiman's suggestions, a slight modification to the arc-fs algorithm has been made in the case where ϵ – the misclassification measure – exceed 0.5, or becomes 0. Should this happen β is set to $-\infty$ and a warning is produced. At this point you are advised to restart the algorithm with equal probabilities or stop boosting, at that iteration.

Value

For internal bookkeeping, this function is inherits from the '`reweighter`' class. It returns a named list with components

`weights`	the updated weights calculated from the input weights, `prediction` and `response`.
`beta`	scalar weights to be used by `arcfsAggregator`.

See Also

Other arc-fs: `arcfsAggregator`

Other reweighters: `adaboostReweighter`; `arcx4Reweighter`; `boost`, `boost.function`, `boost.list`; `vanillaBagger`

arcx4Aggregator *Stock aggregators*

Description

Parallelized implementations of weighted and unweighted "classification by voting" procedures.

Usage

```
arcx4Aggregator(estimators, ..., .parallelPredict = FALSE,
  .parallelTally = FALSE, .rngSeed = 1234)

vanillaAggregator(estimators, ..., .parallelPredict = FALSE,
  .parallelTally = FALSE, .rngSeed = 1234)

weightedAggregator(estimators, weights, ..., .parallelPredict = FALSE,
  .parallelTally = FALSE, .rngSeed = 1234)
```

Arguments

`weights`	a vector of scalar weights associated to each estimator in `estimators`
`estimators`	a list of estimators which must produce output in the same response-space. This is usually the output of some reweighter function.
`...`	this does nothing – meant to swallow auxillary output from reweighter function.
`.parallelPredict`	a boolean indicating if prediction should be carried out in parallel.
`.parallelTally`	a boolean indicating if vote tallying should be performed in parallel. Unless you have more than 1,000 votes / observation, you probably won't see much performance gain by parallelizing this step.
`.rngSeed`	the RNG seed sent to `predictClassFromVote` or `predictClassFromWeightedVote`. in the case of a tie.

Details

`arcx4Aggregator` is just `vanillaAggregator` by another name.

If performing regression and your estimators produce `NA`'s, you can have `weighted.mean` remove the `NA`'s by passing `na.rm=TRUE` to `weightedAggregator`'s function call.

Value

a function whose sole argument is `newdata` and whose output is the aggregated predictions of the boosted ensemble, `estimators`.

For internal bookkeeping, this function is inherits from the '`aggregator`' class.

Note

It's assumed that the estimators in `estimators` are classifiers. More aptly, their output is either of factor or character-type.

See Also

`predictClassFromWeightedVote`; `predictClassFromVote`

Other aggregators: `adaboostAggregator`; `arcfsAggregator`; `boost`, `boost.function`, `boost.list`

`arcx4Reweighter` *Reweighter for the arc-x4 algorithm.*

Description

An implementation of the reweighter described in the arc-x4 algorithm.

Usage

```
arcx4Reweighter(prediction, response, weights, m, ...)
```

Arguments

`prediction`	a vector of predictions.
`response`	a vector whose i^{th} component is the true response for the i^{th} component of `prediction`.
`weights`	a vector of weights. They don't necessarily need to sum to 1.
`...`	implemented to allow reweighter to accept its output as its input.
`m`	a vector length equal to `nrow(data)` enumerating each time the i^{th} entry in `data` has been misclassified by all the estimators previously built.

Value

For internal bookkeeping, this function is inherits from the '`reweighter`' class. It returns a named list with components

`weights`	the updated weights calculated from the input weights, `prediction` and `response`.
`m`	the updated count of misclassifications.

Note

If you're going to use this reweighter with `boost` you'll want to initialize `m` to 0 by including `.reweighterArgs=list(m=0)` inside your `metadata` list.

See Also

Other reweighters: `adaboostReweighter`; `arcfsReweighter`; `boost`, `boost.function`, `boost.list`; `vanillaBagger`

boost — *Boost an Estimation Procedure with a Reweighter and an Aggregator.*

Description

Boost an *estimation procedure* and analyze individual estimator performance using a *reweighter*, *aggregator*, and some *performance analyzer*.

Usage

```
boost(x, B, reweighter, aggregator, data, .procArgs = NULL, metadata = NULL,
  initialWeights = rep.int(1, nrow(data))/nrow(data),
  analyzePerformance = defaultOOBPerformanceAnalysis,
  .boostBackendArgs = NULL)

## S3 method for class 'list'
boost(x, B, reweighter, aggregator, data, .procArgs = NULL,
  metadata = NULL, initialWeights = rep.int(1, nrow(data))/nrow(data),
  analyzePerformance = defaultOOBPerformanceAnalysis,
  .boostBackendArgs = NULL)

## S3 method for class 'function'
boost(x, B, reweighter, aggregator, data, .procArgs = NULL,
  metadata = NULL, initialWeights = rep.int(1, nrow(data))/nrow(data),
  analyzePerformance = defaultOOBPerformanceAnalysis,
  .boostBackendArgs = NULL)
```

Arguments

`B`	number of iterations of boost to perform.
`x`	a list with entries '`train`' and '`predict`' or a function that satisfies the definition of an estimation procedure given below. The list input will invoke a call to `buildEstimationProcedure`. Function input will invoke a call to `wrapProcedure`, unless the function inherits from '`estimationProcedure`'. In either event, metadata may be required to properly wrap `x`. See the appropriate help documentation.
`reweighter`	A reweighter, as defined below. If the function does not inherit from '`reweighter`', a call to `wrapReweighter` will be made. See `wrapReweighter` to determine what metadata, if any, you may need to pass for the wrapper to be `boostr` compatible
`aggregator`	An aggregator, as defined below. If the function does not inherit from '`aggregator`' a call to `wrapAggregator` will be made to build a boostr compatible wrapper. See `wrapAggregator` to determine if any metadata needs to be passed in for this to be successful.

`data`	a data.frame of matrix to act as the learning set. The columns are assumed to be ordered such that the response variable in the first column and the remaining columns as the predictors. As a convenience, `boostBackend` comes with a switch, `.formatData` (defaulted to `TRUE`) which will look for an argument named `formula` inside `.procArgs` and use the value of `formula` to format `data`. If you don't want this to happen, or if the data is already properly formatted, include `.formatData=FALSE` in `metadata`.
`.procArgs`	a named list of arguments to pass to the estimation procedure. If `x` is a list, `.procArgs` is a named list of lists with entries `.trainArgs` and `.predictArgs` and each list is a named list of arguments to pass to `x$train` and `x$predict`, respectively. If `x` is a function, `.procArgs` is a named list of arguments to pass to `x`, in addition to `data` and `weights`. See 'Examples' below.
`initialWeights`	a vector of weights used for the first iteration of the ensemble building phase of Boost.
`analyzePerformance`	a function which accepts an estimator's predictions and the true responses to said predictions (among other arguments) and returns a list of values. If no function is provided, `defaultOOBPerformanceAnalysis` is used. See `wrapPerformanceAnalyzer` for metadata that may need to be passed to make `analyzePerformance` compatible with the boostr framework.
`metadata`	a named list of arguments to be passed to `wrapProcedure`, `buildEstimationProcedure`, `wrapReweighter`, `wrapAggregator`, and/or `wrapPerformanceAnalyzer`.
`.boostBackendArgs`	a named list of additional arguments to pass to `boostBackend`.

Details

This function is a designed to be an interface between the user and `boostBackend` when `x`, `reweighter`, `aggregator` and/or `analyzePerformance` are valid input to the Boost algorithm, but do not have `boostr` compatible signatures. Hence, `boost` calls the appropriate wrapper function (with the relevant information from `metadata`) to convert user supplied functions into `boostr` compatible functions.

Value

a '`boostr`' object which is returned from `boostBackend`. This object is a function of a single input

`newdata`	a data.frame or matrix whose columns should probably be in the same order as the columns of the data each of the constituent estimators was trained on.

The return value of this function is a prediction for each row in `newdata`.

See `boostBackend` for more details on "`boostr`" objects.

See Also

Other aggregators: `adaboostAggregator`; `arcfsAggregator`; `arcx4Aggregator`, `vanillaAggregator`, `weightedAggregator`

Other performance analyzers: `defaultOOBPerformanceAnalysis`

Other reweighters: `adaboostReweighter`; `arcfsReweighter`; `arcx4Reweighter`; `vanillaBagger`

Examples

```
### Demonstrate simple call with just list(train=svm)

library(foreach)
library(iterators)
library(e1071)

svmArgs <- list(formula=Species~., cost=100)
boost(x=list(train=svm),
      reweighter=arcfsReweighter,
      aggregator=arcfsAggregator,
      data=iris,
      .procArgs=list(.trainArgs=svmArgs),
      B=2)

### Demonstrate call with train and predict and custom
### reweighters and aggregators

df <- within(iris, {
  Setosa <- as.factor(2*as.numeric(Species == "setosa")-1)
  Species <- NULL
})

# custom predict function
newPred <- function(obj, new) {
  predict(obj, new)
}

predMetadata <- c(modelName="obj", predictionSet="new")

# custom reweighter
testReweighterMetadata <- list(
                            reweighterInputWts="w",
                            reweighterInputResponse="truth",
                            reweighterInputPreds="preds",
                            reweighterOutputWts="w")

testReweighter <- function(preds, truth, w) {

  wrongPreds <- (preds != truth)
  err <- mean(wrongPreds)
  if (err != 0) {
    new_w <- w / err^(!wrongPreds)
  } else {
    new_w <- runif(n=length(w), min=0, max=1)
  }


  list(w=new_w, alpha=rnorm(1))
}

# custom aggregator
```

```
testAggregatorMetadata <- c(.inputEnsemble="ensemble")

testAggregator <- function(ensemble) {
  weights <- runif(min=0, max=1, n=length(ensemble))
  function(x) {
    preds <- foreach(estimator = iter(ensemble),
                     .combine = rbind) %do% {
                       matrix(as.character(estimator(x)), nrow=1)
                     }

    as.factor(predictClassFromWeightedVote(preds, weights))
  }
}

# collect all the relevant metadata
metadata <- c(predMetadata, testReweighterMetadata, testAggregatorMetadata)

# set additional procedure arguments
procArgs <- list(
              .trainArgs=list(
                formula=Setosa ~ .,
                cost=100)
              )

#test boost when irrelevant metadata is passed in.
boostedSVM <- boost(list(train=svm, predict=newPred),
                    B=3,
                    reweighter=testReweighter,
                    aggregator=testAggregator,
                    data=df,
                    metadata=metadata,
                    .procArgs=procArgs,
                    .boostBackendArgs=list(
                      .reweighterArgs=list(fakeStuff=77))
                    )

### Demonstrate customizing 'metadata' for estimation procedure
library(class)

testkNNProcMetadata <- list(learningSet="traindata", predictionSet="testdata")

testkNNProc <- function(formula, traindata, k) {
  df <- model.frame(formula=formula, data=traindata)
  function(testdata, prob=FALSE) {
    df2 <- tryCatch(model.frame(formula=formula, data=testdata)[, -1],
                    error = function(e) testdata
    )
    knn(train=df[, -1], test=df2, cl=df[, 1], prob=prob, k=k)
  }
}

testKNNProcArgs <- list(formula=Setosa ~ ., k = 5)
```

```
metadata <- testkNNProcMetadata
boostBackendArgs <- list(.reweighterArgs=list(m=0))

boostedKNN <- boost(x=testkNNProc, B=3,
      reweighter=arcx4Reweighter,
      aggregator=arcx4Aggregator,
      data=df,
      metadata=metadata,
      .boostBackendArgs=boostBackendArgs,
      .procArgs=testKNNProcArgs)

### Demonstrate using an alternative performance analyzer

testPerfAnalyzer2 <- function(pred, truth, oob, zeta) {
  list(e=mean(pred != truth), z=zeta)
}

testPerfAnalyzer2Metadata <- list(analyzerInputPreds="pred",
                                  analyzerInputResponse="truth",
                                  analyzerInputOObObs="oob")

metadata <- c(metadata, testPerfAnalyzer2Metadata)

boostedkNN <- boost(testkNNProc,
                    B=3,
                    reweighter=vanillaBagger,
                    aggregator=vanillaAggregator,
                    data=df,
                    .procArgs=testKNNProcArgs,
                    metadata=metadata,
                    .boostBackendArgs = list(
                      .analyzePerformanceArgs = list(zeta="77"),
                      .reweighterArgs=list(fakeStuff=77)),
                    analyzePerformance=testPerfAnalyzer2)
```

boostBackend *Boost an estimation procedure with a reweighter and aggregator.*

Description

Perform the Boost algorithm on proc with reweighter and aggregator and monitor estimator performance with analyzePerformance.

Usage

```
boostBackend(B, reweighter, aggregator, proc, data, initialWeights, .procArgs,
  analyzePerformance = defaultOOBPerformanceAnalysis,
  .reweighterArgs = NULL, .aggregatorArgs = NULL,
  .analyzePerformanceArgs = NULL, .subsetFormula = findFormulaIn(.procArgs),
  .formatData = !is.null(.subsetFormula), .storeData = FALSE,
  .calcBoostrPerformance = TRUE)
```

Arguments

`B`	the number of iterations to run.
`reweighter`	a boostr compatible reweighter function.
`aggregator`	a boostr compatible aggregator function.
`proc`	a boostr compatible estimation procedure.
`data`	the learning set to pass to `proc`. `data` is assumed to hold the response variable in its first column.
`initialWeights`	a vector of weights used for the first iteration of the ensemble building phase of Boost.
`.procArgs`	a named list of arguments to pass to `proc` in addition to `data`.
`.reweighterArgs`	a named list of arguments to pass to `reweighter` in addition to `proc`, `data` and `weights`. These are generally initialization values for other parameters that govern the behaviour of `reweighter`.
`.aggregatorArgs`	a named list of arguments to pass to `aggregator` in addition to the output from `reweighter`.
`.storeData`	a boolean indicating whether the data should be stored in the returned `boostr` object under the attribute "`data`".
`.calcBoostrPerformance`	a boolean indicating whether `analyzePerformance` should be used to monitor the performance of the returned `boostr` object on the learning set. A value of `seq.int(nrow(data))` will be passed to `analyzePerformance` as the `oobObs` argument.
`.subsetFormula`	a `formula` object indicating how `data` is to be subsetted. A formula of like "Type ~ ." will rearrange the columns of `data` such that `data[,1] == data$Type`. By default, this value is taken to be the value of the `formula` entry in `.procArgs`. If multiple entries have the substring "formula" in their names, the search will throw an error and you're advised to manually set `.subsetFormula`.
`.formatData`	a boolean indicating whether the data needs to be reformatted via `.subsetFormula` such that the response variable is in the first column and the remaining columns are all predictor variables. This is defaulted to `!is.null(.subsetFormula)`.
`analyzePerformance`	a boostr compatible performance analyzer.
`.analyzePerformanceArgs`	a named list arguments to pass to `analyzePerformance` in addition to `prediction`, `response`, and `oobPbs`.

Details

For the details behind this algorithm, check out the paper at `http://pollackphoto.net/misc/masters_thesis.pdf`

Value

a "`boostr`" object. The returned closure is the output of `aggregator` on the collection of estimators built during the iterative phase of Boost. This is intended to be a new estimator, and hence accepts the argument `newdata`. However, the estimator also has attributes

`ensembleEstimators`	An ordered list whose components are the trained estimators.
`reweighterOutput`	An ordered list whose components are the output of `reweighter` at each iteration.
`performanceOnLearningSet`	The performance of the returned boostr object on the learning set, as measure by `analyzePerformance`. This is only calculated if `.calcBoostrPerformance=TRUE`
`estimatorPerformance`	An ordered list whose components are the output of `analyzePerformance` at each iteration.
`oobVec`	A row-major matrix whose ij-th entry indicates if observation j was used to train estimator i.
`reweighter`	The reweighter function used.
`reweighterArgs`	Any additional arguments passed to `boostBackend` for `reweighter`.
`aggregator`	The aggregator function used.
`aggregatorArgs`	Any additional arguments passed to `boostBackend` for `aggregator`.
`estimationProcedure`	The estimation procedure used.
`estimationProcedureArgs`	Any additional arguments passed to `boostBackend` for `proc`.
`data`	The learning set. Only stored if `.storeData = TRUE`.
`analyzePerformance`	The performance analyzer used.
`analyzePerformanceArgs`	Any additional arguments passed to `boostBackend` for `analyzePerformance`.
`subsetFormula`	The value of `.subsetFormula`.
`formatData`	The value of `.formatData`.
`storeData`	The value of `.storeData`.
`calcBoostrPerformance`	The value of `.calcBoostrPerformance`
`initialWeights`	The initial weights used.

The attributes can be accessed through the appropropriate `extraction function`.

Note

`wrapReweighter`, `wrapAggregator`, `wrapPerformanceAnalyzer`, `wrapProcedure`, and `buildEstimationProcedure` are all Wrapper Generators designed to allow user implemented functions inside the `boostBackend`. These functions are intelligently called from inside `boost`. Thus, to minimize any sources of frustration, the recommended use of `boostBackend` is through `boost`.

References

Steven Pollack. (2014). Boost: a practical generalization of AdaBoost (Master's Thesis). http://pollackphoto.net/misc/masters_thesis.pdf

Examples

```
## Not run:
df <- within(iris, {
              Setosa <- factor(2*as.numeric(Species == "setosa") - 1)
              Species <- NULL
             })

form <- formula(Setosa ~ . )
df <- model.frame(formula=form, data=df)

# demonstrate arc-fs algorithm using boostr convenience functions

glmArgs <- list(.trainArgs=list(formula=form, family="binomial"))

# format prediction to yield response in {-1,1} instead of {0,1}
glm_predict <- function(object, newdata) {
  2*round(predict(object, newdata, type='response')) - 1
  }

Phi_glm <- buildEstimationProcedure(train=glm, predict=glm_predict)

phi <- boostBackend(B=3, data=df,
                     reweighter=adaboostReweighter,
                     aggregator=adaboostAggregator,
                     proc=Phi_glm,
                     .procArgs=glmArgs)

## End(Not run)
```

---

boostr — *Boost (or bag) an estimation procedure with any reweighter or aggregator.*

Description

boostr provides a modular framework that return the focus of ensemble learning back to 'learning' (instead of programming).

`boostWithArcFs` *Boostr implemented versions of arc-fs, arc-x4 and AdaBoost.*

Description

Perform the Boost algorithm for the algorithms arc-fs, arc-x4, and AdaBoost.

Usage

```
boostWithArcFs(x, B, data, .procArgs = NULL, metadata = NULL,
  initialWeights = rep.int(1, nrow(data))/nrow(data),
  analyzePerformance = defaultOOBPerformanceAnalysis,
  .boostBackendArgs = NULL)

boostWithArcX4(x, B, data, .procArgs = NULL, metadata = NULL,
  initialWeights = rep.int(1, nrow(data))/nrow(data),
  analyzePerformance = defaultOOBPerformanceAnalysis,
  .boostBackendArgs = NULL)

boostWithAdaBoost(x, B, data, .procArgs = NULL, metadata = NULL,
  initialWeights = rep.int(1, nrow(data))/nrow(data),
  analyzePerformance = defaultOOBPerformanceAnalysis,
  .boostBackendArgs = NULL)
```

Arguments

`x`	a list with entries '`train`' and '`predict`' or a function that satisfies the definition of an estimation procedure given below. The list input will invoke a call to `buildEstimationProcedure`. Function input will invoke a call to `wrapProcedure`, unless the function inherits from '`estimationProcedure`'. In either event, metadata may be required to properly wrap `x`. See the appropriate help documentation.
`B`	number of iterations of boost to perform.
`data`	a data.frame of matrix to act as the learning set. The columns are assumed to be ordered such that the response variable in the first column and the remaining columns as the predictors. As a convenience, `boostBackend` comes with a switch, `.formatData` (defaulted to `TRUE`) which will look for an argument named `formula` inside `.procArgs` and use the value of `formula` to format `data`. If you don't want this to happen, or if the data is already properly formatted, include `.formatData=FALSE` in `metadata`.
`.procArgs`	a named list of arguments to pass to the estimation procedure. If `x` is a list, `.procArgs` is a named list of lists with entries `.trainArgs` and `.predictArgs` and each list is a named list of arguments to pass to `x$train` and `x$predict`, respectively. If `x` is a function, `.procArgs` is a named list of arguments to pass to `x`, in addition to `data` and `weights`. See 'Examples' below.

`initialWeights`	a vector of weights used for the first iteration of the ensemble building phase of Boost.
`analyzePerformance`	a function which accepts an estimator's predictions and the true responses to said predictions (among other arguments) and returns a list of values. If no function is provided, `defaultOOBPerformanceAnalysis` is used. See `wrapPerformanceAnalyzer` for metadata that may need to be passed to make `analyzePerformance` compatible with the boostr framework.
`metadata`	a named list of additional arguments to be passed to `wrapProcedure`, `buildEstimationProcedure`, `wrapPerformanceAnalyzer` and/or `boostBackend`.
`.boostBackendArgs`	a named list of additional arguments to pass to `boostBackend`.

Details

These functions call `boost` with the appropriate reweighters, aggregators, and metadata.

Value

a `"boostr"` object that is the output of `boostBackend`.

`buildEstimationProcedure` *Build a boostr compatible estimation procedure.*

Description

A convenience function which builds a boostr compatible estimation procedure from functions `train` and `predict`.

Usage

```
buildEstimationProcedure(train, predict = stats::predict,
  learningSet = "data", predictionSet = "newdata", modelName = "object")
```

Arguments

`train`	a function that learns from data to produce a model
`predict`	a function that leverages the model from `train` to generate predictions from new data.
`learningSet`	a string indicating the name of the argument in `train`'s signature that passes data inside `train`.
`predictionSet`	a string indicating the name of the argument in `predict`'s signature that indicates the observation to predicate responses for.
`modelName`	a string indicating the name of the argument in `predict`'s signature that passes the model from `train` inside `predict`.

Value

An 'estimationProcedure' object which is compatible with the boostr framework. Meaning, the output is a function factory which accepts arguments

`data`	the data to be passed to `train`.
`.trainArgs`	a list of arguments to be passed to `train`, in addition to `data`. If the order of arguments in `train` is important, you'll need to respect that order inside `.trainArgs`.
`.predictArgs`	a list of arguments to pass to `predict` in addition to `modelName` and `predictionSet`. If the order of these arguments matters, respect that order in `.predictArgs`.

and returns a closure with arguments

`newdata`	the data whose response variable is to be estimated.
`.predictArgs`	a list of arguments to pass to `predict` in addition to `modelName` and `predictionSet`. This is defaulted to the value of `.predictArgs` passed to the parent function, however access to this has been granted as a convenience to the user. Again, if the order of these arguments matters, respect that order in `.predictArgs`.

Estimation Procedures

The examples below demonstrate two typical estimation procedures. For more information, see the Estimation Procedures section in the vignette `vignette(topic = "boostr_user_inputs", package="boostr")`.

Warning

This function makes the fundamental assumption that the design-pattern linking `train` and predict is the common `train-predict` pattern found in the design of `SVM` in the examples. If this is not the case, you'll want to build assemble your procedure manually and call `wrapProcedure` instead.

References

Steven Pollack. (2014). Boost: a practical generalization of AdaBoost (Master's Thesis). http://pollackphoto.net/misc/masters_thesis.pdf

See Also

Other Wrapper Generators: `wrapAggregator`; `wrapPerformanceAnalyzer`; `wrapProcedure`; `wrapReweighter`

Examples

```
## Not run:
 # examples of estimation procedures
 library(class)
 library(e1071)

  kNN <- function(data, formula, k) {
   df <- model.frame(formula=formula, data=data)
   function(newdata) {
     knn(train=df[, -1], test=newdata, cl=df[, 1], k=k)
```

```
    }
  }

  SVM <- function(data, formula, cost) {
    model <- svm(formula, data, cost=cost)
    function(newdata) {
      predict(model, newdata)
    }
  }

## End(Not run)
```

`defaultOOBPerformanceAnalysis`

Perform generic out-of-bag error analysis.

Description

If performing regression, calculate which out-of-bag residuals and MSE. Otherwise, calculate which out-of-bag observations were classified correctly, what the overall misclassification rate is, as well as the confusion matrix.

Usage

```
defaultOOBPerformanceAnalysis(prediction, response, oobObs)
```

Arguments

`prediction`	a vector of predicted responses.
`response`	a vector of true response.
`oobObs`	a vector of indices which values in `predictions` are of out-of-bag observations.

Value

If performing regression, return a list with components:

`oobMSE`	the out-of-bag mean squared error.
`resVec`	a vector of length `nrow(data)` whose entries correspond to observations in `data`. The entry has values `NA` if the observation was not out-of-bag, and the difference between the predicted and true response (the residual) if the observation was out-of-bag.

Otherwise, return a list with components:

`oobErr`	overall misclassification rate.
`oobConfMat`	the confusion matrix of out-of-bag predictions against the true class labels.
`errVec`	a vector of length `nrow(data)` whose entries correspond to observations in `data`. The entry has values `NA` if the observation was not out-of-bag, and a 1 or 0 depending whether `estimator` failed to correctly classify the observation.

See Also

Other performance analyzers: `boost`, `boost.function`, `boost.list`

`ensembleEstimators` *Extraction functions for boostr object attributes*

Description

Access the various attributes of "`boostr`" objects through these functions. See `boostBackend` for a description of every `boostr` attribute.

Usage

```
ensembleEstimators(boostrObj)

reweighterOutput(boostrObj)

extractPerformanceOnLearningSet(boostrObj)

extractCalcBoostrPerformance(boostrObj)

estimatorPerformance(boostrObj)

oobVec(boostrObj)

extractReweighter(boostrObj)

reweighterArgs(boostrObj)

extractAggregator(boostrObj)

aggregatorArgs(boostrObj)

extractEstimationProcedure(boostrObj)

estimationProcedureArgs(boostrObj)

extractData(boostrObj)

extractAnalyzePerformance(boostrObj)

analyzePerformanceArgs(boostrObj)

extractSubsetFormula(boostrObj)

extractFormatData(boostrObj)
```

```
extractInitialWeights(boostrObj)
```

Arguments

`boostrObj`	an object of class "`boostr`" – most likely the output of `boost` or `boostBackend`.

Value

The attribute referenced to in the function's title. E.g., `extractEstimationProcedure` returns the stored estimation procedure. `ensembleEstimators` returns the ensemble of estimators built during `boostBackend`.

`isClassConstructor` *check if a function is a(n S3) class constructor*

Description

takes a function and returns a boolean indicating whether its output gets assigned a class.

Usage

```
isClassConstructor(func)
```

Arguments

`func`	any function

Details

The body of `func` is search for one of three idioms:

1. `UseMethod("className")`
2. `class(output) <- classes`
3. `attr(output, "class") <- classes`

If either is found, the assigned class (or classes) are returned as the `classes` attribute of the output. If none are found, a value of `FALSE` is returned (with no attributes).

Value

a boolean. If the return value is `TRUE` the boolean has attribute `classes` which returns the (potential) classes for the output of `func`

Examples

```
isClassConstructor(mean) # FALSE

# simple output
library(randomForest)
isClassConstructor(randomForest) # TRUE

# complicated output (multiple values in "classes")
isClassConstructor(glm) # TRUE
isClassConstructor(lm) # TRUE
```

---

kFoldCV *Generic k-fold Cross Validation wrapper*

Description

A general abstraction of the k-fold cross validation procedure.

Usage

```
kFoldCV(proc, k, data, params,
        .rngSeed = 1234, .chunkSize = 1L, .doSEQ = FALSE)
```

Arguments

proc	the procedure to be k-fold cross validated. proc needs to accept data and newdata in its signature, and must return a numeric vector.
k	the number of folds.
data	a matrix or data.frame from which the folds will be created.
params	a list or data.frame. If params is a list, every combination of the entries in its cells will be used as parameters to be cross validated. If params is a data.frame, each row of arguments will be cross-validated.
.rngSeed	the seed set before randomly generating fold indices.
.chunkSize	the number of parameter combinations to be processed at once (see help for iter).
.doSEQ	logical flag indicating whether cross validation should be run sequentially or with %dopar%.

Details

This function leverages foreach and iter to perform k-fold cross validation in a distributed fashion (provided a parallel backend is registered).

Because the heart of this function is a pair of nested foreach loops one should be careful of "over-parallelization". Meaning, if the routine inside proc is already natively parallel, then by invoking this routine around proc you'll be distributing a distributed computation. This may not yield the speed gains you would expect.

One work around to this – assuming `proc` is parallelized using `foreach` is to call create a wrapper around `proc` that calls `registerDoSEQ`. For example,

`proC <- function(...) {registerDoSEQ(); proc(...)}`

Alternatively, you could run `kFoldCV` sequentially by setting `.doSEQ` to `TRUE`.

For a procedure `proc <- function(data, newdata, arg1, ..., argN){...}` , it may end up that cross-validating a single N-tuple of arguments `c(arg1, ..., argN)` may be very quick. Hence, the time it takes to send off `proc`, the `data` and the appropriate combinations of `params` may overwhelm the actual computation time. In this instance, one should consider changing `.chunkSize` from 1 to `n` (where `n` is any reasonable integer value that would justify the passing of data to a distant node).

Value

a vector whose length is equal to `nrow(params)`, if `params` is a data.frame, or the number of combinations of elements of `params` if it's a list. The i-th component corresponds to the k-fold cross-validated value of `proc` evaluated with parameters from the i-th combination of `params`.

Note

The current implementation of this assumes that entries in `params` are numeric so that `as.matrix(expand.grid(params))` is a numeric matrix with named columns. A work around to passing character parameters would be to translate the character parameter to an integer, and write a wrapper for `proc` that translates the interger back to the appropriate string. See the example below.

Examples

```
# simple example with k-NN where we can build our own wrapper
library(class)
data(iris)
.iris <- iris[, 5:1] # put response as first column

# make a wrapper for class::knn
f <- function(data, newdata, k) {
  preds <- knn(train=data[,-1],
               test=newdata[, -1],
               cl=data[, 1],
               k=k)
  mean(preds==newdata[, 1])
}

params <- list(k=c(1,3,5,7))

accuracy <- kFoldCV(f, 10, .iris, params, .rngSeed=407)

data.frame(expand.grid(params), accuracy=accuracy)

# look at a more complicated example:
# cross validate an svm with different kernels and different models
require(e1071)
g <- function(data, newdata, kernel, cost, gamma, formula) {
  kern <- switch(kernel, "linear", "radial", stop("invalid kernel"))
  form <- switch(formula,
                 as.formula(Species ~ .),
                 as.formula(Species ~ Petal.Length + Petal.Width),
                 as.formula(Petal.Length ~ .),
                 stop('invalid formula'))

   svmWrapper <- function(data, newdata, kernel, cost, gamma, form) {
                   svmObj <- svm(formula=form, data=data, kernel=kernel,
                                 cost=cost, gamma=gamma)
                   predict(svmObj, newdata)
                 }
  preds <- svmWrapper(data, newdata, kernel=kern, cost=cost,
                      gamma=gamma, form=form)

  if (formula != 3) {
    mean(preds == newdata[["Species"]])
  } else {
    mean((preds - newdata[["Petal.Length"]])^2)
  }
}

params <- list(kernel=1:2, cost=c(10,50), gamma=0.01, formula=1)
accuracy <- kFoldCV(g, 10, iris, params)
data.frame(expand.grid(params), metric=accuracy)
```

`makePredictions` *Gather predictions from an ensemble of estimators.*

Description

A parallelized for-loop that goes through each estimator in the given ensemble and collects its predictions in for each row of the given data.

Usage

```
makePredictions(estimators, newdata, .parallel = FALSE)
```

Arguments

`estimators`	a list of functions which take a single (mandatory) argument and returns class label.
`newdata`	the data to feed to each estimator in `estimators`
`.parallel`	a boolean indicating if the predictions should happen in parallel through `estimators`. Defaulted to `FALSE`.

Value

a matrix of predicted responses (either numeric or character, if predictions are factor variables). The columns corresponds to rows in `newdata` so that class-prediction aggregation can be done more effeciently.

---

predictClassFromWeightedVote

Predict a class using (un)weighted voting.

Description

Process a matrix of class predictions and form a column-wise estimate based on weighted voting.

Usage

```
predictClassFromWeightedVote(preds, weights, .parallel = FALSE,
  .rngSeed = 1234)

predictClassFromVote(preds, .parallel = FALSE, .rngSeed = 1234)
```

Arguments

`preds`	is (character) matrix of predicted classes
`weights`	is a vector of length equal to `nrow(preds)`
`.parallel`	is a boolean flag determining whether to work across columns of `preds` in parallel – need to register a parallel backend (e.g. `doParallel`, `doRedis`) for this to actually work.
`.rngSeed`	the value of the RNG seed to be used in the case that ties are to be randomly broken.

Details

Gives the vote from row(i) in `preds` weight equal to `weights[i]`. Ties are broken randomly, but before so, the seed is set to `.rngSeed`.

Value

a character vector of length equal to `ncol(preds)` containing the class estimates per column of `preds`.

`predictResponseFromWeightedAverage`
Predict a numeric response using (un)weighted averaging.

Description

Process a matrix of predicted responses and form a column-wise estimate based on (un)weighted averaging.

Usage

```
predictResponseFromWeightedAverage(preds, weights, .parallel, ...)
```

Arguments

`preds`	is matrix of predicted classes
`weights`	is a vector of length equal to `nrow(preds)`
`.parallel`	is a boolean flag determining whether to work across columns of `preds` in parallel – need to register a parallel backend (e.g. `doParallel`, `doRedis`) for this to actually work.
`...`	additional arguments to pass to `weighted.mean`.

Details

Gives the prediction from row(i) in `preds` weight equal to `weights[i]`. Note that `NA`'s are not removed. To have `weighted.mean` remove the `NA`'s pass `na.rm=TRUE` to the function call.

Value

a vector of length equal to `ncol(preds)` containing the estimated response for each column of `preds`.

`vanillaBagger` *Standard (vanilla) bagging procedure.*

Description

Build an estimator from a simple resampling of data.

Usage

```
vanillaBagger(prediction, response, ...)
```

Arguments

`prediction`	a vector of predictions.
`response`	a vector whose i^{th} component is the true response for the i^{th} component of `prediction`.
`...`	implemented to allow reweighter to accept its output as its input.

Value

a list with component '`weights`': a normalized vector of 1's with length equal to that of `response`.

Note

a "bagger" is just a reweighter who returns uniform weights regardless of the input.

See Also

Other reweighters: `adaboostReweighter`; `arcfsReweighter`; `arcx4Reweighter`; `boost`, `boost.function`, `boost.list`

`wrapAggregator` *Create a boostr compatible wrapper for an aggregator.*

Description

Use provided metadata on a given aggregator to create a boostr compatible wrapper. See section below for details on aggregators.

Usage

```
wrapAggregator(aggregator, .inputEnsemble = "estimators", .verbose = FALSE)
```

Arguments

`aggregator`	a function which satisfies the abstract definition of an aggregator.
`.inputEnsemble`	a string indicating the name of the argument that `aggregator` uses for the ensemble of estimators created during the Boost algorithm.
`.verbose`	a logical flag indicating whether warnings should be output or not.

Value

A function with is also an '`aggregator`' object. The function's signature and output are now compatible with the boostr framework. In particular, the signature of the wrapper is

`estimators`	the list of estimators to be sent to `aggregator`.
`...`	any additional arguments accepted/required by `aggregator`.

The output of this aggregator is an estimator with signature

`newdata`	the data the `aggregator`'s output would use for prediction.

Aggregators

See the Aggregators section in the vignette `vignette(topic = "boostr_user_inputs", package="boostr")` for more details on aggregators.

References

Steven Pollack. (2014). Boost: a practical generalization of AdaBoost (Master's Thesis). `http://pollackphoto.net/misc/masters_thesis.pdf`

See Also

Other Wrapper Generators: `buildEstimationProcedure`; `wrapPerformanceAnalyzer`; `wrapProcedure`; `wrapReweighter`

Examples

```
## Not run:
testAggregator <- function(ensemble) {
 weights <- runif(min=0, max=1, n=length(ensemble))
 function(x) {
   preds <- foreach(estimator = iter(ensemble),
                    .combine = rbind) %do% {
                      matrix(as.character(estimator(x)), nrow=1)
                    }

   as.factor(predictClassFromWeightedVote(preds, weights))
 }
}

wrappedAggregator <- wrapAggregator(testAggregator,
                                    .inputEnsemble="ensemble")

## End(Not run)
```

`wrapPerformanceAnalyzer` *Create a boostr compatible wrapper for a performance analyzer.*

Description

Use provided metadata on a given performance analyzer to create a boostr compatible wrapper.

Usage

```
wrapPerformanceAnalyzer(analyzePerformance, analyzerInputPreds = "prediction",
  analyzerInputResponse = "response", analyzerInputOOBObs = "oobObs",
  .verbose = FALSE)
```

Arguments

`analyzePerformance`	a function to analyze the performance of an estimator
`analyzerInputPreds`	a string indicating the name of the argument in `analyzePerformance`'s signature that represents the estimator's predictions.
`analyzerInputResponse`	a string indicating the name of the argument in `analyzePerformance`'s signature that represents the true response associated with the estimator's predictions.
`analyzerInputOOBObs`	a string indiciating the name of the argument in `analyzePerformance`'s signature that represents the vector of indices indicating which observations were out-of-bag.
`.verbose`	a boolean indicating if warnings should be displayed or not.

Details

Since "performance" is a subjective thing, the requirements for a function to be wrappable by `wrapPerformanceAnalyzer` are that they accept predictions, true responses, and a vector of indices for out-of-bag observations. After each iteration of the ensemble building phase in `boostBackend`, these three objects are fed to a performance analyzer. The output of the performance analyze is stored in the `estimatorPerformance` attribute of the object returned by `boostBackend`.

Value

A function (wrapper around `analyzePerformance`) which is also a '`performanceAnalyzer`' object. The function's signature is `(prediction, response, oobObs, ...)` and it's output preserves the output of `analyzePerformance`. Hence, the wrapper is a boostr compatible performance analyzer.

Performance Analyzers

Any function which can accept an estimator's predictions, as well as the true responses can be used as a "performance analyzer" in `boost`. That is, if the signature of a function can be transformed to `(predictions, responses, ...)`, then `wrapPerformanceAnalyzer` can be run on the function, and the results can be used by `boostBackend`. The output of the performance analyzer is entirely preserved, and can be accessed by running `estimatorPerformance` on the resulting 'boostr' object.

At every iteration of the ensemble building phase, `boostBackend` passes `performance Analysis`:

- the newly built estimator's predicted responses for each row in `data`.
- the true response for each row in `data`.
- the indices of the observations in `data` that were not included in the sample of `data` that went into creating the estimator. This variable is passed in as `oobObs`.
- whatever other named arguments were passed from `boost` through the `...`'s.

Hence, an analyzer can accept all three values (and then some) to perform whatever analysis is desired on an individual estimator.

For example, the stock performance analyzer for classification, `defaultOOBPerformanceAnalysis`, has signature `(prediction, response, oobObs)` and calculates an individual, out-of-bag, misclassification vector, the overall out-of-bag error rate, and the confusion matrix for a particular estimator, given the information passed to it.

See Also

Other Wrapper Generators: `buildEstimationProcedure`; `wrapAggregator`; `wrapProcedure`; `wrapReweighter`

`wrapProcedure` *Create a boostr compatible wrapper for an estimation procedure.*

Description

Use provided metadata on a given estimation procedure to create a boostr compatible wrapper. See section below for more details on estimation procedures.

Usage

```
wrapProcedure(proc, learningSet = "data", predictionSet = "newdata")
```

Arguments

`proc`	a function that obeys the definition of an estimation procedure as defined in the white paper. Generally, `proc` must be a function which learns some model and consequently returns an estimator that uses the learned model. See below.
`learningSet`	a string indicating the name of the argument in `proc`'s signature that passes the data to be used inside `proc`.
`predictionSet`	a string indicating the name of the argument in `predict`'s signature that indicates the observation to predicate responses for.

Value

An '`estimationProcedure`' object whose signature and whose output's signature are compatible with boostr. Explicitly, the arguments of the wrapper are

`data`	the data that `proc` will use to build a model.
`...`	any additional arguments necessary for `proc` to make its model.

and the returned closure from the wrapper has arguments

`newdata`	the data that `proc`'s output will predict responses for.
`.estimatorArgs`	a named list of any additional arguments that need to be passed to `proc`'s output.

Estimation Procedures

The examples below demonstrate two typical estimation procedures. For more information, see the Estimation Procedures section in the vignette `vignette(topic = "boostr_user_inputs", package="boostr")`.

References

Steven Pollack. (2014). Boost: a practical generalization of AdaBoost (Master's Thesis). http://pollackphoto.net/misc/masters_thesis.pdf

See Also

Other Wrapper Generators: `buildEstimationProcedure`; `wrapAggregator`; `wrapPerformanceAnalyzer`; `wrapReweighter`

Examples

```
## Not run:
 # examples of estimation procedures
 library(class)
 library(e1071)

  kNN <- function(data, formula, k) {
   df <- model.frame(formula=formula, data=data)
   function(newdata) {
     knn(train=df[, -1], test=newdata, cl=df[, 1], k=k)
   }
  }

  SVM <- function(data, formula, cost) {
   model <- svm(formula, data, cost=cost)
   function(newdata) {
     predict(model, newdata)
   }
  }

## End(Not run)
```

`wrapReweighter` *Create a boostr compatible wrapper for a reweighter.*

Description

Use provided metadata on a given reweighter to create a boostr compatible wrapper.

Usage

```
wrapReweighter(reweighter, reweighterInputPreds = "prediction",
  reweighterInputResponse = "response", reweighterInputWts = "weights",
  reweighterOutputWts = "weights", .verbose = FALSE)
```

Arguments

`reweighter`	a function which satisfies the abstract definition of a reweighter (see description below).
`reweighterInputPreds`	a string indicating the name of the argument `reweighter` uses to represent the input predictions.
`reweighterInputResponse`	a string indicating the name of the argument `reweighter` uses to represent the true responses for the input predictions.
`reweighterInputWts`	a string indicating the name of the argument `reweighter` uses to represent the input weights.
`reweighterOutputWts`	a string indicating the name of the entry in `reweighter`'s output that represents the output weights.
`.verbose`	a boolean indicating if warnings should be displayed or not.

Value

A function (wrapper around `reweighter`) which is a '`reweighter`' object. The wrapper's signature is `(prediction, response, weights, ...)` and its output is a list that names the cell containing its weights '`weight`'. Hence, the wrapper is a boostr compatible reweighter.

Reweighters

See the Reweighters section in the vignette `vignette(topic = "boostr_user_inputs", package="boostr")` for more details on reweighters.

References

Steven Pollack. (2014). Boost: a practical generalization of AdaBoost (Master's Thesis). `http://pollackphoto.net/misc/masters_thesis.pdf`

See Also

Other Wrapper Generators: `buildEstimationProcedure`; `wrapAggregator`; `wrapPerformanceAnalyzer`; `wrapProcedure`

Index