# MLInterfaces

April 19, 2010

---

| | |
|---|---|
| `balKfold.xvspec` | *generate a partition function for cross-validation, where the partitions are approximately balanced with respect to the distribution of a response variable* |

---

## Description

generate a partition function for cross-validation, where the partitions are approximately balanced with respect to the distribution of a response variable

## Usage

```
balKfold.xvspec(K)
```

## Arguments

| | |
|---|---|
| `K` | number of partitions to be computed |

## Details

This function returns a closure. The symbol `K` is bound in the environment of the returned function.

## Value

A closure consisting of a function that can be used as a `partitionFunc` for passage in `xvalSpec`.

## Author(s)

VJ Carey <stvjc@channing.harvard.edu>

## Examples

```
## The function is currently defined as
function (K)
function(data, clab, iternum) {
    clabs <- data[[clab]]
    narr <- nrow(data)
    cnames <- unique(clabs)
    ilist <- list()
    for (i in 1:length(cnames)) ilist[[cnames[i]]] <- which(clabs ==
        cnames[i])
    clens <- lapply(ilist, length)
    nrep <- lapply(clens, function(x) ceiling(x/K))
    grpinds <- list()
    for (i in 1:length(nrep)) grpinds[[i]] <- rep(1:K, nrep[[i]])[1:clens[[i]]]
    (1:narr)[-which(unlist(grpinds) == iternum)]
  }
# try it out
data(crabs)
p1c = balKfold.xvspec(5)
inds = p1c( crabs, "sp", 3 )
table(crabs$sp[inds] )
inds2 = p1c( crabs, "sp", 4 )
table(crabs$sp[inds2] )
allc = 1:200
# are test sets disjoint?
intersect(setdiff(allc,inds), setdiff(allc,inds2))
```

---

## `classifierOutput-class` *Class "classifierOutput"*

---

### Description

This class summarizes the output values from different classifiers.

### Objects from the Class

Objects are typically created during the application of a supervised machine learning algorithm to data and are the value returned. It is very unlikely that any user would create such an object by hand.

### Slots

`testOutcomes`: Object of class `"factor"` that lists the actual outcomes in the records on the test set

`testPredictions`: Object of class `"factor"` that lists the predictions of outcomes in the test set

`testScores`: Object of class `"ANY"` – this element will include matrices or vectors or arrays that include information that is typically related to the posterior probability of occupancy of the predicted class or of all classes. The actual contents of this slot can be determined by inspecting the converter element of the learnerSchema used to select the model.

`trainOutcomes`: Object of class `"factor"` that lists the actual outcomes in records on the training set

`trainPredictions`: Object of class `"factor"` that lists the predicted outcomes in the training set

`trainScores`: Object of class `"ANY"` see the description of `testScores` above; the same information is returned, but applicable to the training set records.

`RObject`**:** Object of class `"ANY"` – when the `trainInd` parameter of the `MLearn` call is numeric, this slot holds the return value of the underlying R function that carried out the predictive modeling. For example, if `rpartI` was used as MLearn `method`, `Robject` holds an instance of the `rpart` S3 class, and `plot` and `text` methods can be applied to this. When the `trainInd` parameter of the `MLearn` call is an instance of `xvalSpec`, this slot holds a `list` of results of cross-validatory iterations. Each element of this list has two elements: `test.idx`, giving the numeric indices of the test cases for the associated cross-validation iteration, and `mlans`, which is the `classifierOutput` for the associated iteration. See the example for an illustration of 'digging out' the predicted probabilities associated with each cross-validation iteration executed through an xvalSpec specification.

`embeddedCV`**:** logical value that is TRUE if the procedure in use performs its own cross-validation

`fsHistory`**:** list of features selected through cross-validation process

`learnerSchema`**:** propagation of the learner schema object used in the call

`call`**:** Object of class `"call"` – records the call used to generate the classifierOutput RObject

## Methods

**confuMat** `signature(obj = "classifierOutput")`: Compute the confusion matrix for test records.

**confuMatTrain** `signature(obj = "classifierOutput")`: Compute the confusion matrix for training set. Typically yields optimistically biased information on misclassification rate.

**RObject** `signature(obj = "classifierOutput")`: The R object returned by the underlying classifier. This can then be passed on to specific methods for those objects, when they exist.

**show** `signature(object = "classifierOutput")`: A print method that provides a summary of the output of the classifier.

**testScores** `signature(object = "classifierOutput")`: ...

**testPredictions** `signature(object = "classifierOutput")`: Print the predicted classes for each sample/individual in the test set.

**trainPredictions** `signature(object = "classifierOutput")`: Print the predicted classes for each sample/individual in the training set.

**fsHistory** `signature(object = "classifierOutput")`: ...

## Author(s)

V. Carey

## Examples

```
showClass("classifierOutput")
library(golubEsets)
data(Golub_Train) # now cross-validate a neural net
set.seed(1234)
xv5 = xvalSpec("LOG", 5, balKfold.xvspec(5))
m2 = MLearn(ALL.AML~., Golub_Train[1000:1050,], nnetI, xv5,
   size=5, decay=.01, maxit=1900 )
testScores(RObject(m2)[[1]]$mlans)
alls = lapply(RObject(m2), function(x) testScores(x$mlans))
```

## `clusteringOutput-class` *container for clustering outputs in uniform structure*

### Description

container for clustering outputs in uniform structure

### Objects from the Class

Objects can be created by calls of the form `new("clusteringOutput", ...)`.

### Slots

`partition`: Object of class `"integer"`, labels for observations as clustered

`silhouette`: Object of class `"silhouette"`, structure from Rousseeuw cluster package measuring cluster membership strength per observation

`prcomp`: Object of class `"prcompObj"` a wrapped instance of stats package prcomp output

`call`: Object of class `"call"` for auditing

`learnerSchema`: Object of class `"learnerSchema"`, a formal object indicating the package, function, and other attributes of the clustering algorithm employed to generate this object

`RObject`: Object of class `"ANY"`, the unaltered output of the function called according to learnerSchema

`converter`: converter propagated from call

`distFun`: distfun propagated from call

### Methods

**RObject** `signature(x = "clusteringOutput")`: extract the unaltered output of the R function or method called according to learnerSchema

**plot** `signature(x = "clusteringOutput", y = "ANY")`: a 4-panel plot showing features of the clustering, including the scree plot for a principal components transformation and a display of the partition in PC1xPC2 plane. For a clustering method that does not have a native plot procedure, such as kmeans, the parameter y should be bound to a data frame or matrix with feature data for all records; an image plot of robust feature z-scores (z=(x-median(x))/mad(x)) and the cluster indices is produced in the northwest panel.

**show** `signature(object = "clusteringOutput")`: concise report

### Author(s)

VJ Carey <stvjc@channing.harvard.edu>

### Examples

```
showClass("clusteringOutput")
```

---

`confuMat-methods` *Compute the confusion matrix for a classifier.*

---

## Description

This function will compute the confusion matrix for a classifier's output

## Methods

**obj = "classifOutput"** Typically, an instance of class "`classifierOutput`" is built on a training subset of the input data. The model is then used to predict the class of samples in the test set. When the true class labels for the test set are available the confusion matrix is the cross-tabulation of the true labels of the test set against the predictions from the classifier.

**obj = "classifierOutput", type="character"** For instances of classifierOutput, it is possible to specify the `type` of confusion matrix desired. The default is `test`, which tabulates classes from the test set against the associated predictions. If `type` is `train`, the training class vector is tabulated against the predictions on the training set.

## Examples

```
library(golubEsets)
data(Golub_Merge)
smallG <- Golub_Merge[101:150,]
k1 <- MLearn(ALL.AML~., smallG, knnI(k=1), 1:30)
confuMat(k1)
confuMat(k1, "train")
```

---

`fs.absT` *support for feature selection in cross-validation*

---

## Description

support for feature selection in cross-validation

## Usage

```
fs.absT(N)
fs.probT(p)
fs.topVariance(p)
```

## Arguments

| | |
|---|---|
| `N` | number of features to retain; features are ordered by descending value of abs(two-sample t stat.), and the top N are used. |
| `p` | cumulative probability (in (0,1)) in the distribution of absolute t statistics above which we retain features |

## Details

This function returns a function that will be used as a parameter to `xvalSpec` in applications of `MLearn`.

### Value

a function is returned, that will itself return a formula consisting of the selected features for application of MLearn.

### Note

The functions fs.absT and fs.probT are two examples of approaches to embedded feature selection that make sense for two-sample prediction problems. For selection based on linear models or other discrimination measures, you will need to create your own selection helper, following the code in these functions as examples.

fs.topVariance performs non-specific feature selection based on the variance. Argument p is the variance percentile beneath which features are discarded.

### Author(s)

VJ Carey <stvjc@channing.harvard.edu>

### See Also

MLearn

### Examples

```
# we will demonstrate this procedure with the crabs data.
# first, create the closure to pick 3 features
demFS = fs.absT(3)
# run it on the entire dataset with features excluding sex
demFS(sp~.-sex, crabs)
# emulate cross-validation by excluding last 50 records
demFS(sp~.-sex, crabs[1:150,])
# emulate cross-validation by excluding first 50 records -- different features retained
demFS(sp~.-sex, crabs[51:200,])
```

## fsHistory *extract history of feature selection for a cross-validated machine learner*

### Description

extract history of feature selection for a cross-validated machine learner

### Usage

```
fsHistory(x)
```

### Arguments

| | |
|---|---|
| x | instance of classifierOutput |

### Details

returns a list of names of selected features

### Value

a list; the names of variables are made 'syntactic'

### Author(s)

Vince Carey <stvjc@channing.harvard.edu>

### Examples

```
data(iris)
iris2 = iris[ iris$Species %in% levels(iris$Species)[1:2], ]
iris2$Species = factor(iris2$Species) # drop unused levels
x1 = MLearn(Species~., iris2, ldaI, xvalSpec("LOG", 3,
   balKfold.xvspec(3), fs.absT(3)))
fsHistory(x1)
```

---

## `learnerSchema-class` *Class "learnerSchema" – convey information on a machine learning function to the MLearn wrapper*

---

### Description

conveys information about machine learning functions in CRAN packages, for example, to MLearn wrapper

### Objects from the Class

Objects can be created by calls of the form `new("learnerSchema", ...)`.

### Slots

`packageName`: Object of class `"character"` string naming the package in which the function to be used is defined.

`mlFunName`: Object of class `"character"` string naming the function to be used

`converter`: Object of class `"function"` function with parameters obj, data, trainInd, that will produce a classifierOutput instance

### Methods

**MLearn** `signature(formula = "formula", data = "ExpressionSet", method = "learnerSchema", trainInd = "numeric")`: execute desired learner passing a formula and ExpressionSet

**MLearn** `signature(formula = "formula", data = "data.frame", method = "learnerSchema", trainInd = "numeric")`: execute desired learner passing a formula

**show** `signature(object = "learnerSchema")`: concise display

### Author(s)

Vince Carey <stvjc@channing.harvard.edu>

## Examples

```
showClass("learnerSchema")
```

---

`MLearn` *revised MLearn interface for machine learning*

---

## Description

revised MLearn interface for machine learning, emphasizing a schematic description of external learning functions like knn, lda, nnet, etc.

## Usage

```
MLearn( formula, data, .method, trainInd, ... )
makeLearnerSchema(packname, mlfunname, converter)
```

## Arguments

| | |
|---|---|
| `formula` | standard model formula |
| `data` | data.frame or ExpressionSet instance |
| `.method` | instance of learnerSchema |
| `trainInd` | obligatory numeric vector of indices of data to be used for training; all other data are used for testing, or instance of the xvalSpec class |
| `...` | additional named arguments passed to external learning function |
| `packname` | character – name of package harboring a learner function |
| `mlfunname` | character – name of function to use |
| `converter` | function – with parameters (obj, data, trainInd) that tells how to convert the material in obj [produced by [packname::mlfunname] ] into a classifierOutput instance. |

## Details

The purpose of the MLearn methods is to provide a uniform calling sequence to diverse machine learning algorithms. In R package, machine learning functions can have parameters `(x, y, ...)` or `(formula, data, ...)` or some other sequence, and these functions can return lists or vectors or other sorts of things. With MLearn, we always have calling sequence `MLearn(formula, data, .method, trainInd, ...)`, and `data` can be a `data.frame` or `ExpressionSet`. `MLearn` will always return an S4 instance of `classifierObject` or `clusteringObject`.

At this time (1.13.x), NA values in predictors trigger an error.

To obtain documentation on the older (pre bioc 2.1) version of the MLearn method, please use help(MLearn-OLD).

**randomForestI** randomForest. Note, that to obtain the default performance of randomForestB, you need to set mtry and sampsize parameters to sqrt(number of features) and table([training set response factor]) respectively, as these were not taken to be the function's defaults. Note you can use xvalSpec("NOTEST") as trainInd, to use all the samples; the RObject() result will print the misclassification matrix estimate along with OOB error rate estimate.

**knnI(k=1,l=0)** knn; special support bridge required, defined in MLint

**knn.cvI(k=1,l=0)** knn.cv; special support bridge required, defined in MLint. This option uses the embedded leave-one-out cross-validation of `knn.cv`, and thereby achieves high performance. You can have more general cross-validation using `knnI` with an `xvalSpec`, but it will be slower. When using this learner schema, you should use the numerical `trainInd` setting with `1:N` where `N` is the number of samples.

**dldaI** diagDA; special support bridge required, defined in MLint

**nnetI** nnet

**rpartI** rpart

**ldaI** lda

**svmI** svm

**qdaI** qda

**logisticI(threshold)** glm – with binomial family, expecting a dichotomous factor as response variable, not bulletproofed against other responses yet. If response probability estimate exceeds threshold, predict 1, else 0

**adaI** ada

**BgbmI** gbm, forcing the Bernoulli loss function.

**blackboostI** blackboost – you MUST supply a family parameter relevant for mboost package procedures

**lvqI** lvqtest after building codebook with lvqinit and updating with olvq1. You will need to write your own detailed schema if you want to tweak tuning parameters.

**naiveBayesI** naiveBayes

**baggingI** bagging

**sldaI** slda

**rdaI** rda – you must supply the alpha and delta parameters to use this. Typically cross-validation is used to select these. See `rdacvI` below.

**rdacvI** rda.cv. This interface is complicated. The typical use includes cross-validation internal to the rda.cv function. That process searches a tuning parameter space and delivers an ordering on parameters. The interface selects the parameters by looking at all parameter configurations achieving the smallest min+1SE cv.error estimate, and taking the one among them that employed the -most- features (agnosticism). A final run of rda is then conducted with the tuning parameters set at that 'optimal' choice. The bridge code can be modified to facilitate alternative choices of the parameters in use. `plotXvalRDA` is an interface to the plot method for objects of class rdacv defined in package rda. You can use xvalSpec("NOTEST") with this procedure to use all the samples to build the discriminator.

**ksvmI** ksvm

**hclustI(distMethod, agglomMethod)** hclust – you must explicitly specify distance and agglomeration procedure.

**kmeansI(centers, algorithm)** kmeans – you must explicitly specify centers and algorithm name.

## Value

Instances of classifierOutput or clusteringOutput

## Author(s)

Vince Carey <stvjc@channing.harvard.edu>

## Examples

```
data(crabs)
set.seed(1234)
kp = sample(1:200, size=120)
rf1 = MLearn(sp~CW+RW, data=crabs, randomForestI, kp, ntree=600 )
rf1
nn1 = MLearn(sp~CW+RW, data=crabs, nnetI, kp, size=3, decay=.01 )
nn1
RObject(nn1)
knn1 = MLearn(sp~CW+RW, data=crabs, knnI(k=3,l=2), kp)
knn1
names(RObject(knn1))
dlda1 = MLearn(sp~CW+RW, data=crabs, dldaI, kp )
dlda1
names(RObject(dlda1))
lda1 = MLearn(sp~CW+RW, data=crabs, ldaI, kp )
lda1
names(RObject(lda1))
slda1 = MLearn(sp~CW+RW, data=crabs, sldaI, kp )
slda1
names(RObject(slda1))
svm1 = MLearn(sp~CW+RW, data=crabs, svmI, kp )
svm1
names(RObject(svm1))
ldapp1 = MLearn(sp~CW+RW, data=crabs, ldaI.predParms(method="debiased"), kp )
ldapp1
names(RObject(ldapp1))
qda1 = MLearn(sp~CW+RW, data=crabs, qdaI, kp )
qda1
names(RObject(qda1))
logi = MLearn(sp~CW+RW, data=crabs, glmI.logistic(threshold=0.5), kp, family=binomial ) #
logi
names(RObject(logi))
rp2 = MLearn(sp~CW+RW, data=crabs, rpartI, kp)
rp2
## recode data for RAB
#nsp = ifelse(crabs$sp=="O", -1, 1)
#nsp = factor(nsp)
#ncrabs = cbind(nsp,crabs)
#rab1 = MLearn(nsp~CW+RW, data=ncrabs, RABI, kp, maxiter=10)
#rab1
#
# new approach to adaboost
#
ada1 = MLearn(sp ~ CW+RW, data = crabs, .method = adaI,
    trainInd = kp, type = "discrete", iter = 200)
ada1
confuMat(ada1)
#
lvq.1 = MLearn(sp~CW+RW, data=crabs, lvqI, kp )
lvq.1
nb.1 = MLearn(sp~CW+RW, data=crabs, naiveBayesI, kp )
confuMat(nb.1)
bb.1 = MLearn(sp~CW+RW, data=crabs, baggingI, kp )
confuMat(bb.1)
#
```

```
# new mboost interface -- you MUST supply family for nonGaussian response
#
require(party)
blb.1 = MLearn(sp~CW+RW+FL, data=crabs, blackboostI, kp, family=mboost::Binomial() )
confuMat(blb.1)
#
# ExpressionSet illustration
#
data(sample.ExpressionSet)
X = MLearn(type~., sample.ExpressionSet[100:250,], randomForestI, 1:16, importance=TRUE )
library(randomForest)
library(hgu95av2.db)
opar = par(no.readonly=TRUE)
par(las=2)
plot(getVarImp(X), n=10, plat="hgu95av2", toktype="SYMBOL")
par(opar)
#
# demonstrate cross validation
#
nn1cv = MLearn(sp~CW+RW, data=crabs[c(1:20,101:120),], nnetI, xvalSpec("LOO"), size=3, de
confuMat(nn1cv)
nn2cv = MLearn(sp~CW+RW, data=crabs[c(1:20,101:120),], nnetI,
   xvalSpec("LOG",5, balKfold.xvspec(5)), size=3, decay=.01 )
confuMat(nn2cv)
nn3cv = MLearn(sp~CW+RW+CL+BD+FL, data=crabs[c(1:20,101:120),], nnetI,
   xvalSpec("LOG",5, balKfold.xvspec(5), fsFun=fs.absT(2)), size=3, decay=.01 )
confuMat(nn3cv)
nn4cv = MLearn(sp~.-index-sex, data=crabs[c(1:20,101:120),], nnetI,
   xvalSpec("LOG",5, balKfold.xvspec(5), fsFun=fs.absT(2)), size=3, decay=.01 )
confuMat(nn4cv)
#
# try with expression data
#
library(golubEsets)
data(Golub_Train)
litg = Golub_Train[ 100:150, ]
g1 = MLearn(ALL.AML~. , litg, nnetI, xvalSpec("LOG",5, balKfold.xvspec(5), fsFun=fs.probT
confuMat(g1)
#
# illustrate rda.cv interface from package rda (requiring local bridge)
#
library(ALL)
data(ALL)
#
# restrict to BCR/ABL or NEG
#
bio <- which( ALL$mol.biol %in% c("BCR/ABL", "NEG"))
#
# restrict to B-cell
#
isb <- grep("^B", as.character(ALL$BT))
kp <- intersect(bio,isb)
all2 <- ALL[,kp]
mads = apply(exprs(all2),1,mad)
kp = which(mads>1)  # get around 250 genes
vall2 = all2[kp, ]
vall2$mol.biol = factor(vall2$mol.biol) # drop unused levels
```

```
r1 = MLearn(mol.biol~., vall2, rdacvI, 1:40)
confuMat(r1)
RObject(r1)
plotXvalRDA(r1)  # special interface to plots of parameter space

# illustrate clustering support

cl1 = MLearn(~CW+RW+CL+FL+BD, data=crabs, hclustI(distFun=dist, cutParm=list(k=4)))
plot(cl1)

cl1a = MLearn(~CW+RW+CL+FL+BD, data=crabs, hclustI(distFun=dist, cutParm=list(k=4)),
   method="complete")
plot(cl1a)

cl2 = MLearn(~CW+RW+CL+FL+BD, data=crabs, kmeansI, centers=5, algorithm="Hartigan-Wong")
plot(cl2, crabs[,-c(1:3)])

c3 = MLearn(~CL+CW+RW, crabs, pamI(dist), k=5)
c3
plot(c3, data=crabs[,c("CL", "CW", "RW")])
```

---

## `MLIntInternals` *MLInterfaces infrastructure*

---

### Description

These functions are internal tools for `MLInterfaces`. Users will generally not call these functions directly.

### Usage

```
getGrid(x)
```

### Arguments

| | |
|---|---|
| `x` | a vector or matrix or ExpressionSet |

### Details

Forthcoming.

### Value

Functions with ‘new’ as prefix are constructor helpers.

### Author(s)

VJ Carey <stvjc@channing.harvard.edu>

## `planarPlot-methods` *Methods for Function planarPlot in Package 'MLInterfaces'*

### Description

show the classification boundaries on the plane dictated by two genes in an ExpressionSet

### Methods

**clo = "classifierOutput", eset = "ExpressionSet", classifLab = "character"** uses two genes in the ExpressionSet to exhibit the decision boundaries in the plane

**clo = "classifierOutput", eset = "data.frame", classifLab = "character"** uses two columns in the data.frame to exhibit the decision boundaries in the plane

### Examples

```
library(ALL)
library(hgu95av2.db)
data(ALL)
#
# restrict to BCR/ABL or NEG
#
bio <- which( ALL$mol.biol %in% c("BCR/ABL", "NEG"))
#
# restrict to B-cell
#
isb <- grep("^B", as.character(ALL$BT))
kp <- intersect(bio,isb)
all2 <- ALL[,kp]
#
# sample 2 genes at random
#
set.seed(1234)
ng <- nrow(exprs(all2))
pick <- sample(1:ng, size=2, replace=FALSE)
gg <- all2[pick,]
sym <- unlist(mget(featureNames(gg), hgu95av2SYMBOL))
featureNames(gg) <- sym
gg$class = factor(ifelse(all2$mol.biol=="NEG", "NEG", "POS"))

cl1 <- which( gg$class == "NEG" )
cl2 <- which( gg$class != "NEG" )
#
# create balanced training sample
#
trainInds <- c( sample(cl1, size=floor(length(cl1)/2) ),
      sample(cl2, size=floor(length(cl2)/2)) )
#
# run rpart
#
tgg <- MLearn(class~., gg, rpartI, trainInds, minsplit=4 )
opar <- par(no.readonly=TRUE)
par(mfrow=c(2,2))
planarPlot( tgg, gg, "class" )
```

```
title("rpart")
points(exprs(gg)[1,trainInds], exprs(gg)[2,trainInds], col=ifelse(gg$class[trainInds]=="N
#
# run nnet
#
ngg <- MLearn( class~., gg, nnetI, trainInds, size=8 )
planarPlot( ngg, gg, "class" )
points(exprs(gg)[1,trainInds], exprs(gg)[2,trainInds], col=ifelse(gg$class[trainInds]=="N
title("nnet")
#
# run knn
#
kgg <- MLearn( class~.,  gg, knnI(k=3,l=1), trainInds)
planarPlot( kgg, gg, "class" )
points(exprs(gg)[1,trainInds], exprs(gg)[2,trainInds], col=ifelse(gg$class[trainInds]=="N
title("3-nn")
#
# run svm
#
sgg <- MLearn( class~., gg, svmI, trainInds )
planarPlot( sgg, gg, "class" )
points(exprs(gg)[1,trainInds], exprs(gg)[2,trainInds], col=ifelse(gg$class[trainInds]=="N
title("svm")
par(opar)
```

## `raboostCont-class` *Class "raboostCont" ~~~*

### Description

~~ A concise (1-5 lines) description of what the class is. ~~

### Objects from the Class

Objects can be created by calls of the form `new("raboostCont", ...)`. ~~ describe objects here ~~

### Slots

`.Data`: Object of class `"list"` ~~

`formula`: Object of class `"formula"` ~~

`call`: Object of class `"call"` ~~

### Extends

Class `"list"`, from data part. Class `"vector"`, by class "list", distance 2.

### Methods

`Predict` is an S4 method that can apply to instances of this class.

### Author(s)

VJ Carey <stvjc@channing.harvard.edu>

### Examples

```
showClass("raboostCont")
```

---

## RAB *real adaboost (Friedman et al)*

---

### Description

read adaboost ... a demonstration version

### Usage

```
RAB(formula, data, maxiter=200, maxdepth=1)
```

### Arguments

| | |
|---|---|
| `formula` | formula – the response variable must be coded -1, 1 |
| `data` | data |
| `maxiter` | maxiter |
| `maxdepth` | maxdepth – passed to rpart |

### Value

an instance of raboostCont

### Author(s)

Vince Carey <stvjc@channing.harvard.edu>

### References

Friedman et al Ann Stat 28/2 337

### Examples

```
library(MASS)
data(Pima.tr)
data(Pima.te)
Pima.all = rbind(Pima.tr, Pima.te)
tonp = ifelse(Pima.all$type == "Yes", 1, -1)
tonp = factor(tonp)
Pima.all = data.frame(Pima.all[,1:7], mtype=tonp)
fit1 = RAB(mtype~ped+glu+npreg+bmi+age, data=Pima.all[1:200,], maxiter=10, maxdepth=5)
pfit1 = Predict(fit1, newdata=Pima.tr)
table(Pima.tr$type, pfit1)
```

## `varImpStruct-class` *Class "varImpStruct" – collect data on variable importance from various machine learning methods*

### Description

collects data on variable importance

### Objects from the Class

Objects can be created by calls of the form `new("varImpStruct", ...)`. These are matrices of importance measures with separate slots identifying algorithm generating the measures and variable names.

### Slots

`.Data`: Object of class `"matrix"` actual importance measures

`method`: Object of class `"character"` tag

`varnames`: Object of class `"character"` conformant vector of names of variables

### Extends

Class `"matrix"`, from data part. Class `"structure"`, by class `"matrix"`. Class `"array"`, by class `"matrix"`. Class `"vector"`, by class "matrix", with explicit coerce. Class `"vector"`, by class "matrix", with explicit coerce.

### Methods

**plot** `signature(x = "varImpStruct")`: make a bar plot, you can supply arguments `plat` and `toktype` which will use `lookUp(...,plat,toktype)` from the `annotate` package to translate probe names to, e.g., gene symbols.

**show** `signature(object = "varImpStruct")`: simple abbreviated display

**getVarImp** `signature(object = "classifOutput", fixNames="logical")`: extractor of variable importance structure; fixNames parameter is to remove leading X used to make variable names syntactic by randomForest (ca 1/2008). You can set fixNames to false if using hu6800 platform, because all featureNames are syntactic as given.

**report** `signature(object = "classifOutput", fixNames="logical")`: extractor of variable importance data, with annotation; fixNames parameter is to remove leading X used to make variable names syntactic by randomForest (ca 1/2008). You can set fixNames to false if using hu6800 platform, because all featureNames are syntactic as given.

### Examples

```
library(golubEsets)
data(Golub_Merge)
library(hu6800.db)
smallG <- Golub_Merge[1001:1060,]
set.seed(1234)
opar=par(no.readonly=TRUE)
par(las=2, mar=c(10,11,5,5))
rf2 <- MLearn(ALL.AML~., smallG, randomForestI, 1:40, importance=TRUE,
 sampsize=table(smallG$ALL.AML[1:40]), mtry=sqrt(ncol(exprs(smallG))))
plot( getVarImp( rf2, FALSE ), n=10, plat="hu6800", toktype="SYMBOL")
par(opar)
report( getVarImp( rf2, FALSE ), n=10, plat="hu6800", toktype="SYMBOL")
```

## `xvalLoop` *Cross-validation in clustered computing environments*

### Description

Use cross-validation in a clustered computing environment

### Usage

```
xvalLoop( cluster, ... )
```

### Arguments

| | |
|---|---|
| `cluster` | Any S4-class object, used to indicate how to perform clustered computations. |
| `...` | Additional arguments used to inform the clustered computation. |

### Details

Cross-validiation usually involves repeated calls to the same function, but with different arguments. This provides an obvious place for using clustered computers to enhance execution. The method xval is structured to exploit this; `xvalLoop` provides an easy mechanism to change how `xval` performs cross-validation.

The idea is to write an `xvalLoop` method that returns a function. The function is then used to execute the cross-validation. For instance, the default method returns the function `lapply`, so the cross-validation is performed by using `lapply`. A different method might return a function that executed lapply-like functions, but sent different parts of the function to different computer nodes.

An accompanying vignette illustrates the technique in greater detail. An effective division of labor is for experienced cluster programmers to write lapply-like methods for their favored clustering environment. The user then only has to add the cluster object to the list of arguments to `xval` to get clustered calculations.

### Value

A function taking arguments like those for `lapply`

### Examples

```
## Not run:
library(golubEsets)
data(Golub_Merge)
smallG <- Golub_Merge[200:250,]

# Evaluation on one node

lk1 <- xval(smallG, "ALL.AML", knnB, xvalMethod="LOO", group=as.integer(0))
table(lk1,smallG$ALL.AML)
```

```
# Evaluation on several nodes -- a cluster programmer might write the following...

library(snow)
setOldClass("spawnedMPIcluster")

setMethod("xvalLoop", signature( cluster = "spawnedMPIcluster"),
## use the function returned below to evalutae
## the central cross-validation loop in xval
function( cluster, ... ) {
    clusterExportEnv <- function (cl, env = .GlobalEnv)
    {
        unpackEnv <- function(env) {
            for ( name in ls(env) ) assign(name, get(name, env), .GlobalEnv )
            NULL
        }
        clusterCall(cl, unpackEnv, env)
    }
    function(X, FUN, ...) { # this gets returned to xval
        ## send all visible variables from the parent (i.e., xval) frame
        clusterExportEnv( cluster, parent.frame(1) )
        parLapply( cluster, X, FUN, ... )
    }
})

# ... and use the cluster like this...

cl <- makeCluster(2, "MPI")
clusterEvalQ(cl, library(MLInterfaces))

lk1 <- xval(smallG, "ALL.AML", knnB, xvalMethod="LOO", group=as.integer(0), cluster = cl)
table(lk1,smallG$ALL.AML)

## End(Not run)
```

---

`xvalSpec` *container for information specifying a cross-validated machine learning exercise*

---

## Description

container for information specifying a cross-validated machine learning exercise

## Usage

```
xvalSpec( type, niter=0, partitionFunc= function(data, classLab, iternum ) {
(1:nrow(data))[-iternum] },
                   fsFun = function(formula, data) formula )
```

## Arguments

| | |
|---|---|
| `type` | a string, "LOO" indicating leave-one-out cross-validation, or "LOG" indicating leave-out-group, or "NOTEST", indicating the entire dataset is used in a single training run. |

| | |
|---|---|
| `niter` | numeric specification of the number of cross-validation iterations to use. Ignored if `type` is `"LOO"`. |
| `partitionFunc` | function, with parameters data (bound to data.frame), clab (bound to character string), iternum (bound to numeric index into sequence of 1:`niter`). This function's job is to provide the indices of training cases for each cross-validation step. An example is `balKfold.xvspec`, which computes a series of indices that are approximately balanced with respect to frequency of outcome types. |
| `fsFun` | function, with parameters formula, data. The function must return a formula suitable for defining a model on the basis of the main input data. A candidate fsFun is given in example for fsHistory function. |

## Details

If `type == "LOO"`, no other parameters are inspected. If `type == "LOG"` a value for `partitionFunc` must be supplied. We recommend using `balKfold.xvspec(K)`. The values of `niter` and `K` in this usage must be the same. This redundancy will be removed in a future upgrade.

## Value

An instance of `classifierOutput`, with a special structure. The `RObject` return slot is populated with a list of `niter` cross-validation results. Each element of this list is itself a list with two elements: `test.idx` (the indices of the test set for the associated cross-validation iteration, and `mlans`, the `classifierOutput` generated at each iteration. Thus there are `classifierOutput` instances nested within the main `classifierOutput` returned when a `xvalSpec` is used.

## Author(s)

Vince Carey <stvjc@channing.harvard.edu>

## Examples

```
data(crabs)
set.seed(1234)
#
# demonstrate cross validation
#
nn1cv = MLearn(sp~CW+RW, data=crabs, nnetI, xvalSpec("LOG",
   5, balKfold.xvspec(5)), size=3, decay=.01 )
nn1cv
confuMat(nn1cv)
names(RObject(nn1cv)[[1]])
RObject(RObject(nn1cv)[[1]]$mlans)
```

# Index