Package 'randomUniformForest'

February 20, 2015

Type Package

Title Random Uniform Forests for Classification, Regression and Unsupervised Learning

Version 1.1.5

Date 2015-02-16

Author Saip Ciss

Maintainer Saip Ciss <saip.ciss@wanadoo.fr>

Description Ensemble model, for classification, regression and unsupervised learning, based on a forest of unpruned and randomized binary decision trees. Each tree is grown by sampling, with replacement, a set of variables at each node. Each cut-point is generated randomly, according to the continuous Uniform distribution. For each tree, data are either bootstrapped or subsampled. The unsupervised mode introduces clustering, dimension reduction and variable importance, using a three-layer engine. Random Uniform Forests are mainly aimed to lower correlation between trees (or trees residuals), to provide a deep analysis of variable importance and to allow native distributed and incremental learning.

License BSD_3_clause + file LICENSE

LazyData yes

Depends R (>= 3.0.0)

Imports methods, Rcpp (>= 0.11.1), parallel, doParallel, iterators, foreach (>= 1.4.2), ggplot2, pROC, gtools, cluster, MASS

Suggests R.rsp

VignetteBuilder R.rsp

LinkingTo Rcpp

NeedsCompilation yes

Repository CRAN

Date/Publication 2015-02-16 21:29:00

R topics documented:

randomUniformForest-package
Random Uniform Forests for Classification, Regression and Unsupervised Learning

Description

Ensemble model for classification, regression and unsupervised learning, based on a forest of unpruned and randomized binary decision trees. Unlike *Breiman's Random Forests*, each tree is grown by sampling, *with replacement*, a set of variables before splitting each node. Each cut-point is generated randomly, according to the *continuous Uniform distribution between two random points of each candidate variable or using its whole current support*. Optimal random node is, then, selected among many full random ones by maximizing Information Gain (classification) or minimizing a distance (regression), 'L2' (or 'L1'). Unlike *Extremely Randomized Trees*, data are either *bootstrapped or sub-sampled for each tree*. From the theoretical side, Random Uniform Forests are aimed to lower correlation between trees and to offer a deep analysis of variable importance. The unsupervised mode introduces clustering and dimension reduction, using a three-layer engine: dissimilarity matrix, Multidimensional Scaling (or spectral decomposition) and k-means (or hierarchical clustering). From the practical side, Random Uniform Forests are designed to provide a complete analysis of (un)supervised problems and to allow native distributed and incremental learning.

Details

Package:	randomUniformForest
Type:	Package
Version:	1.1.5
Date:	2015-02-16
License:	BSD_3_clause

Installation: install.packages("randomUniformForest")
Usage: library(randomUniformForest)

Author(s)

Saip Ciss

Maintainer: Saip Ciss <saip.ciss@wanadoo.fr>

References

Amit, Y., Geman, D., 1997. Shape Quantization and Recognition with Randomized Trees. *Neural Computation* 9, 1545-1588.

Biau, G., Devroye, L., Lugosi, G., 2008. Consistency of random forests and other averaging classifiers. *The Journal of Machine Learning Research* 9, 2015-2033.

Bousquet, O., Boucheron, S., Lugosi, G., 2004. *Introduction to Statistical Learning Theory*, in: Bousquet, O., Luxburg, U. von, Ratsch, G. (Eds.), Advanced Lectures on Machine Learning, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 169-207.

Breiman, L, 1996. Heuristics of instability and stabilization in model selection. *The Annals of Statistics* 24, no. 6, 2350-2383.

Breiman, L., 1996. Bagging predictors. *Machine learning* 24, 123-140.

Breiman, L., 2001. Statistical Modeling: The Two Cultures (with comments and a rejoinder by the author). *Statistical Science* 16, no. 3, 199-231.

Breiman, L., 2001. Random Forests, *Machine Learning* 45(1), 5-32.

Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C., 1984. *Classification and Regression Trees*. New York: Chapman and Hall.

Ciss, S., 2014. PhD thesis: *Forets uniformement aleatoires et detection des irregularites aux cotisations sociales*. Universite Paris Ouest Nanterre, France. In french.
English title : *Random Uniform Forests and Irregularity Detection in social Security contributions*.
Link : https://www.dropbox.com/s/q7hbgeafrdd8qtc/Saip_Ciss_These.pdf?dl=0

Ciss, S., 2015a. Random Uniform Forests. Preprint. hal-01104340.

Ciss, S., 2015b. Variable Importance in Random Uniform Forests. Preprint. hal-01104751.

Ciss, S., 2015c. Generalization Error and Out-of-bag Bounds in Random (Uniform) Forests. Preprint. hal-01110524.

Cox, T. F., Cox, M. A. A., 2001. *Multidimensional Scaling. Second edition*. Chapman and Hall.

Devroye, L., Gyorfi, L., Lugosi, G., 1996. *A probabilistic theory of pattern recognition*. New York: Springer.

Dietterich, T.G., 2000. *Ensemble Methods in Machine Learning*, in : Multiple Classifier Systems, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 1-15.

Efron, B., 1979. Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics* 7, 1-26.

Gower, J. C., 1966. Some distance properties of latent root and vector methods used in multivariate analysis. *Biometrika* 53, 325-328.

Gyorfi, L., 2002. *A distribution-free theory of nonparametric regression*. Springer Science & Business Media.

Hastie, T., Tibshirani, R., Friedman, J.J.H., 2001. *The elements of statistical learning*. New York: Springer.

Ho, T.K., 1998. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, 832-844.

Lin, Y., Jeon, Y., 2002. Random Forests and Adaptive Nearest Neighbors. *Journal of the American Statistical Association* 101-474.

Ng, A. Y., Jordan, M. I., Weiss, Y., 2002. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 2, 849-856.

Vapnik, V.N., 1995. *The nature of statistical learning theory*. Springer-Verlag New York, Inc., New York, NY, USA.

Examples

```
## Presenting some of the core functions
## Not run

## 1 - Classification: iris data set (assess whole data)

## load data (included in R):
# data(iris)
# XY = iris
# p = ncol(XY)
# X = XY[,-p]
# Y = XY[,p]

## Train a model : using formula
# iris.ruf = randomUniformForest(Species ~., XY, threads = 1)

## or using matrix
## iris.ruf = randomUniformForest(X, as.factor(Y), threads = 1)

## Assess model : Out-of-bag (OOB) evaluation
# iris.ruf

## Variable Importance : base assessment
# summary(iris.ruf)

## Variable Importance : deeper assessment (explain the modelling)
# iris.importance = importance(iris.ruf, Xtest = X, maxInteractions = p - 1)

## Visualize : details of Variable Importance
## (tile windows vertically, using the R menu, to see all plots)
# plot(iris.importance, Xtest = X)

## Analyse : get an interpretation of the model results
# iris.ruf.analysis = clusterAnalysis(iris.importance, X, components = 3,
# clusteredObject = iris.ruf, OOB = TRUE)

## Dimension reduction, clustering and visualization : OOB evaluation
# iris.clust.ruf = clusteringObservations(iris.ruf, X, importanceObject = iris.importance)

## 2 - Regression: Boston Housing (assess a test set)

## load data :
# install.packages("mlbench") ##if not installed
# data(BostonHousing, package = "mlbench")

# XY = BostonHousing
# p = ncol(XY)
# X = XY[,-p]
# Y = XY[,p]

## get random training and test sets :
## reproducibility :
```

```
# set.seed(2015)

# train_test = init_values(X, Y, sample.size = 1/2)
# Xtrain = train_test$xtrain
# Ytrain = train_test$ytrain
# Xtest = train_test$xtest
# Ytest = train_test$ytest

## Train a model :
# boston.ruf = randomUniformForest(Xtrain, Ytrain)

## Assess (quickly) the model :
# boston.ruf
# plot(boston.ruf)
# summary(boston.ruf)

## Predict the test set :
# boston.pred.ruf = predict(boston.ruf, Xtest)

## or predict quantiles
# boston.predQunatile_97.5.ruf = predict(boston.ruf, Xtest, type = "quantile",
# whichQuantile = 0.975)

## or prediction intervals
# boston.predConfInt_95.ruf = predict(boston.ruf, Xtest, type = "confInt", conf = 0.95)

## Assess predictions :
# statsModel = model.stats(boston.pred.ruf, Ytest, regression = TRUE)

## Avoiding overfitting : under the i.i.d. assumption, OOB error
## is expected to be an upper bound of MSE. Convergence is first needed.
## Convergence needs low correlation between trees residuals.

# boston.ruf

## The easy way; reduce correlation(decreasing 'mtry' value) + post-processing
# bostonNew.ruf = randomUniformForest(Xtrain, Ytrain, mtry = 4)

## (predict and) Post-process :
# bostonNew.predAll.ruf = predict(bostonNew.ruf, Xtest, type = "all")

# bostonNew.postProcessPred.ruf = postProcessingVotes(bostonNew.ruf,
# predObject = bostonNew.predAll.ruf)

## Assess new predictions :
# statsModel = model.stats(bostonNew.postProcessPred.ruf, Ytest, regression = TRUE)

## Convergence : grow more trees
# bostonNew.moreTrees.ruf = rUniformForest.grow(bostonNew.ruf, Xtrain, ntree = 100)
```

`as.supervised` *Conversion of an unsupervised model into a supervised one*

Description

Turn an unsupervised object of class unsupervised into a supervised one of class RandomUniformForest, allowing prediction of next unlabelled datasets, full analysis of variable importance in the unsupervised case and incremental unsupervised learning.

Usage

```
as.supervised(object, X, ...)
```

Arguments

`object`	an object of class unsupervised
`X`	the dataset previously learned in the unsupervised mode.
`...`	allow all options of `randomUniformForest` to be passed to the as.supervised() function, e.g. ntree, mtry, nodesize, ...

Details

The function get clusters labels and send them to the randomUniformForest classifier in order to be learnt with the data. The resulting object can be used for any coming dataset for fast prediction or clustering. Note that main argument of the method is that the model which generates the unsupervised model also generates the supervised one. The process to clustering, in both unsupervised and supervised cases, is filtered by the dissimilarity matrix in conjunction with the MDS function.

Value

An object of class randomUniformForest. See `randomUniformForest`.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

See Also

`modifyClusters`, `mergeClusters` `clusteringObservations`

Examples

```
# see unsupervised.randomUniformForest() function.
```

autoMPG *Auto MPG Data Set*

Description

Revised from CMU StatLib library, data concerns city-cycle fuel consumption.

Usage

```
autoMPG
```

Format

A matrix containing 398 observations and 8 attributes.

Source

http://archive.ics.uci.edu/ml/datasets/Auto+MPG

References

Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tenth International Conference of Machine Learning, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.

bCI *Bootstrapped Prediction Intervals for Ensemble Models*

Description

Bootstrapped prediction (and confidence) interval(s) for problems where response vector has a very frequent value (e.g. 0) or for drawing tighter prediction interval (for each response) than those provided by random Uniform Forests.

Usage

```
bCI(X, f = mean, R = 100, conf = 0.95, largeValues = TRUE, skew = TRUE, threads = "auto")
```

Arguments

X	a matrix of base model outputs for observations of the test sample. Each row contains outputs of all base models for the current response.
f	a function to approximate. Can be arbitrary but must take outputs of base models (vector) and return a scalar. By default, 'f' is the mean() function, since a random (Uniform) Forest output is, for each observation, an average of all trees outputs. bCi() will, then, compute 'f(X)' for each response, where 'f' is the function over model parameter and 'X' are the outputs of base models.
R	number of replications required to compute prediction intervals.
conf	value of 'conf' (greater than 0 and lesser than 1) is the desired level of confidence for all prediction intervals.
largeValues	if TRUE, assume that quantile estimates (bounds of each prediction interval) need to be enough large, so that bounds will be realistic. If FALSE, tightest bounds will be computed, leading to small confidence interval for a parameter, e.g. mean of the response values, but not suitable prediction intervals. If one needs only confidence interval, set option to FALSE.
skew	if TRUE (and largeValues = TRUE), assume that bounds admit fat tails in one or another direction (but not both, since then bounds will be large). Useful for exponential type distributions where data distribution is not really continuous, typically when one value is, by far, the most frequent and is (or close to) the smallest one. Note that if computed lower bound goes under the minimum of the distribution, one has to adjust it, since bCI function always assumes that only distribution of response is known, not its value (or estimate) for any single case. If FALSE, bCI() function will consider, for each response, a Gaussian neighbourhood depending on unique outputs of base models (number and variance). So, bounds computed are expected to be tighter. Note that prediction intervals must, in the same time, be globally consistent with the confidence level of confidence interval. One can assess this, by using outsideConfIntLevels() function (see examples) in order to choose right instance of the option. For example, outsideConfIntLevels() function on the OOB classifier is one way to do it.
threads	compute model in parallel for computers with many cores. Default value is "auto", letting model running on all logical cores minus 1. One can set 'threads' to any values >= 1, depending on the number of cores (include logical).

Details

Main objective of the bCI() function is to provide more realistic prediction intervals in the same time than an acceptable confidence interval for a parameter, e.g. mean, for ensemble models. Since trees outputs are little correlated and random (conditional expectation) for each response, random Uniform Forests will provide prediction intervals larger than those really needed. bCI() function attempts to reduce them, especially when one value of the response variable is overrepresented, assuming either a Gaussian neighbourhood or an exponential type (e.g. large variance) around each estimated bound of each response. Confidence interval is simply the mean of prediction intervals bounds. Model assumes that if prediction intervals are tight, confidence interval will be (reciprocity is usually not valid) and options give different ways to achieve it. Note that bCI() function is designed for random Uniform Forests, but could work (not tested) with any ensemble model based on little correlated outputs of base models.

Value

a matrix of estimates, lower and upper bounds.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

Examples

```
# n = 100;  p = 10
# Simulate 'p' gaussian vectors with random parameters between -10 and 10.
# X <- simulationData(n,p)

## make a rule to create response vector
# epsilon1 = runif(n,-1,1)
# epsilon2 = runif(n,-1,1)
# Y = 2*(X[,1]*X[,2] + X[,3]*X[,4]) + epsilon1*X[,5] + epsilon2*X[,6]

## suppose Y has many zeros and a reduced scale
# Y[Y <= mean(Y)] = 0
# Y[Y > 0] = Y[Y > 0]/100

## randomize and make train and test sample
# twoSamples <- cut(sample(1:n,n), 2, labels = FALSE)

# Xtrain = X[which(twoSamples == 1),]
# Ytrain = Y[which(twoSamples == 1)]
# Xtest = X[which(twoSamples == 2),]
# Ytest = Y[which(twoSamples == 2)]

## compute a model and predict
## NOTE : in regression, the model uses subsampling
# rUF.model <- randomUniformForest(Xtrain, Ytrain, bagging = TRUE, logX = TRUE,
# ntree = 50, threads = 1, OOB = FALSE)

######### PART ONE : get classical predictions intervals

## CLASSIC PREDICTION INTERVALS by random Uniform Forest :
## get 99% predictions interval by the standard predictions interval of the model
# rUF.ClassicPredictionsInterval <- predict(rUF.model, Xtest, type = "confInt", conf = 0.99)

## check mean squared error
# sum((rUF.ClassicPredictionsInterval$Estimate - Ytest)^2)/length(Ytest)

## PROBABILITY (estimate) OF BEING OUT OF THE BOUNDS OF PREDICTION INTERVALS
## using all test sample, meaning that we compute the probability that any response
## goes outside its predicted interval when assessing the test set

# outsideConfIntLevels(rUF.ClassicPredictionsInterval, Ytest, conf = 0.99)

## get average of prediction intervals (we will assess it against the one derived
## in the second part)
```

```
# mean(abs(rUF.ClassicPredictionsInterval[,3] - rUF.ClassicPredictionsInterval[,2]))

######### PART TWO : get bootstrapped predictions intervals

## BOOTSTRAPPED PREDICTION INTERVALS by random Uniform Forest :
##  get all votes
# rUF.predictionsVotes <- predict(rUF.model, Xtest, type = "votes")

##  get 99% predictions interval by bCI:
# bCI.predictionsInterval <- bCI(rUF.predictionsVotes, conf = 0.99, R = 100, threads = 1)

## since we know that lower bound can not be lower than 0, we set it explicitly
## (it is already the case in standard prediction interval)
# bCI.predictionsInterval[bCI.predictionsInterval[,2] < 0, 2] = 0

##  PROBABILITY (estimate) OF BEING OUT OF THE BOUNDS OF PREDICTION INTERVALS
## (bounds are expected to be, at most, slightly worst than in the standard case)
# outsideConfIntLevels(bCI.predictionsInterval, Ytest, conf = 0.99)

##  get average of prediction intervals
## (expected to be much tighter than the standard one)
# mean(abs(bCI.predictionsInterval[,3] - bCI.predictionsInterval[,2]))

######### PART THREE : COMPARISON OF CONFIDENCE INTERVALS

## CONFIDENCE INTERVAL FOR THE EXPECTATION OF Y:
## mean of Y
# mean(Ytest)

## estimate of confidence interval by bCI:
# colMeans(bCI.predictionsInterval[,2:3])

## estimate by standard confidence interval:
# colMeans(rUF.ClassicPredictionsInterval[,2:3])

## using bCI(), get closer confidence interval for expectation of Y, at the expense
## of realistic prediction intervals:
## get 99% predictions interval by bCI, assuming very tight bounds
# bCI.UnrealisticPredictionsInterval <- bCI(rUF.predictionsVotes,
# conf = 0.99, R = 200, largeValues = FALSE, threads = 1)

##  assess it
## (we do not want predictions interval...)
# outsideConfIntLevels(bCI.UnrealisticPredictionsInterval, Ytest, conf = 0.99)

## get (bayesian?) mean and confidence interval for the expectation of Y
## (...but we want good confidence interval)
# colMeans(bCI.UnrealisticPredictionsInterval)

## much closer than the standard one, with almost the same mean
# colMeans(rUF.ClassicPredictionsInterval)

######## PART FOUR : A REAL CASE
```

```
#### Regression : "Concrete Compressive Strength" data
## (http://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength)

# data(ConcreteCompressiveStrength)
# ConcreteCompressiveStrength.data = ConcreteCompressiveStrength

# n <- nrow(ConcreteCompressiveStrength.data)
# p <- ncol(ConcreteCompressiveStrength.data)

# set.seed(2014)
# trainTestIdx <- cut(sample(1:n, n), 2, labels= FALSE)

## train examples
# Concrete.data.train <- ConcreteCompressiveStrength.data[trainTestIdx == 1, -p]
# Concrete.responses.train <- ConcreteCompressiveStrength.data[trainTestIdx == 1, p]

## model
# Concrete.ruf <- randomUniformForest(Concrete.data.train, Concrete.responses.train,
# threads = 2)
# Concrete.ruf

## get test data
# Concrete.data.test <- ConcreteCompressiveStrength.data[trainTestIdx == 2, -p]
# Concrete.responses.test <- ConcreteCompressiveStrength.data[trainTestIdx == 2, p]

## assessing predictions intervals :

## 1- with the OOB classifier, what can we expect from the bounds
## when going toward the test set :

## first, compute the prediction intervals of the OOB classifier
# OOBpredictionIntervals = bCI(Concrete.ruf$forest$OOB.votes, conf = 0.95)

## main interest: how could we rely on the OOB evaluation to get useful informations
## about what we will predict when going toward the test set
# outsideConfIntLevels(OOBpredictionIntervals, Concrete.responses.train, conf = 0.95)

## 2- so, we know now that our 95% confidence interval must be tempered
## when giving estimates for the test set: in other words, the confidence level we ask
## will probably not be the one we will get from the test set.
## Let us ask for prediction intervals from the forest classifier:

# predictionIntervals = bCI(predict(Concrete.ruf, Concrete.data.test, type= "votes"))

## a- main interest: after getting the test responses we can now look, if the OOB classifier
## was right, i.e. out-of-bounds probabilities for the test set with the forest classifier
## should see their sum being over 5% and close to (the sum of) the OOB ones
##(but this latter would probably require more trees than the default value we let)
# outsideConfIntLevels(predictionIntervals, Concrete.responses.test, conf = 0.95)

## b- let us now look to the confidence interval for the mean of test responses:
## two arguments here : the forest classifier has no bias (under the i.i.d assumption);
```

```
## average of prediction intervals lead to confidence interval for the true mean

# colMeans(predictionIntervals)

## comparing to the true mean
# mean(Concrete.responses.test)

## and to the one of the standard case of the forest classifier
# colMeans(predict(Concrete.ruf, Concrete.data.test, type= "confInt", conf= 0.95))
```

`biasVarCov` *Bias-Variance-Covariance Decomposition*

Description

Bias-Variance-Covariance decomposition for Mean Squared Error (MSE) or test error in binary classification, between a response vector and its estimate, over the test sample. For every estimate, based on training examples, MSE on the test sample, between values of the response and values of the estimate, can be decomposed in noise (variance of the response), squared bias (between response and estimate), variance (of the estimate) and covariance (of the response and the estimate). Same decomposition arrives for binary classification with responses in $\{0, 1\}$.

Usage

```
biasVarCov(predictions, target, regression = FALSE, idx = 1:length(target))
```

Arguments

`predictions`	vector of predictions for the test sample.
`target`	response vector for the test sample.
`regression`	if TRUE, decomposition is done for regression. If FALSE, for classification.
`idx`	not currently used.

Value

a list with the following components :

`MSE, predError`	mean squared error or test error for the test sample.
`squaredBias`	squared bias.
`predictionsVar`	variance of the estimate.
`predictionsTargetCov`	covariance between estimate and response.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

Examples

```
n = 100;  p = 10
# simulate 'p' gaussian vectors with random parameters between -10 and 10.
X <- simulationData(n,p)

# make a rule to create response vector
epsilon1 = runif(n,-1,1)
epsilon2 = runif(n,-1,1)
rule = 2*(X[,1]*X[,2] + X[,3]*X[,4]) + epsilon1*X[,5] + epsilon2*X[,6]

# Classification
Y <- as.factor(ifelse(rule > mean(rule), 1, 0))

# define random train and test sample of (X,Y)
trainTestIdx <- cut(sample(1:nrow(X), nrow(X)), 2, labels= FALSE)

# training sample
Xtrain = X[trainTestIdx == 1,]
Ytrain = Y[trainTestIdx == 1]

# test sample
Xtest = X[trainTestIdx == 2,]
Ytest = Y[trainTestIdx == 2]

# learning model
synthDataset.ruf <- randomUniformForest(Xtrain, Ytrain, threads = 1,
ntree = 20, BreimanBounds = FALSE)
synthDataset.pred.ruf <- predict(synthDataset.ruf, Xtest)

# test error
testError = 1 - sum(synthDataset.pred.ruf == Ytest)/length(Ytest)
testError

# test error decomposition
testError.dec <- biasVarCov(synthDataset.pred.ruf, Ytest, regression = FALSE)

# Regression
Y = rule

# training sample
Xtrain = X[trainTestIdx == 1,]
Ytrain = Y[trainTestIdx == 1]

# test sample
Xtest = X[trainTestIdx == 2,]
Ytest = Y[trainTestIdx == 2]

# learning model
synthDataset.ruf <- randomUniformForest(Xtrain, Ytrain, threads = 1,
ntree = 20, BreimanBounds = FALSE)
synthDataset.pred.ruf <- predict(synthDataset.ruf, Xtest)
```

```
# MSE
MSE <- sum((synthDataset.pred.ruf - Ytest)^2)/length(Ytest)

# test error decomposition
MSE.dec <- biasVarCov(synthDataset.pred.ruf, Ytest, regression = TRUE)
```

`breastCancer` *Breast Cancer Wisconsin (Original) Data Set*

Description

Original Wisconsin Breast Cancer Database

Usage

```
breastCancer
```

Format

A matrix containing 699 observations and 10 attributes with missing values.

Source

http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Original)

References

Wolberg, W.H., and Mangasarian, O.L. (1990). Multisurface method of pattern separation for medical diagnosis applied to breast cytology. In Proceedings of the National Academy of Sciences, 87, 9193–9196.

Zhang, J. (1992). Selecting typical instances in instance-based learning. In Proceedings of the Ninth International Machine Learning Conference (pp. 470–479). Aberdeen, Scotland: Morgan Kaufmann.

`CarEvaluation` *Car Evaluation Data Set*

Description

Derived from simple hierarchical decision model, this database may be useful for testing constructive induction and structure discovery methods.

Usage

```
carEvaluation
```

Format

A matrix containing 1728 observations and 6 attributes.

Source

http://archive.ics.uci.edu/ml/datasets/Car+Evaluation

References

M. Bohanec and V. Rajkovic: Knowledge acquisition and explanation for multi-attribute decision making. In 8th Intl Workshop on Expert Systems and their Applications, Avignon, France. pages 59-78, 1988.

B. Zupan, M. Bohanec, I. Bratko, J. Demsar: Machine learning by function decomposition. ICML-97, Nashville, TN. 1997 (to appear)

`clusterAnalysis` *Cluster (or classes) analysis of importance objects.*

Description

Provides a full analysis of clustered objects in a compact and granular representation. More precisely, observations, features and clusters are analysed in a same scheme, leading to unify and interpret all results of the unsupervised mode in one way. The function prints at most 5 tables and is designed to be an extension of importance object and also works in the supervised mode.

Usage

```
clusterAnalysis(object, X,
components = 2,
maxFeatures = 2,
clusteredObject = NULL,
categorical = NULL,
OOB = FALSE)
```

Arguments

`object`	an object of class 'importance'.
`X`	the original data frame (or matrix) used to compute 'importance'.
`components`	number of components to use for assessing cluster (or class). It has the same purpose than eigenvectors in PCA. Note that the number must not excess the value of 'maxInteractions' in `importance.randomUniformForest`.
`maxFeatures`	number of components to print for the visibility of analysis.

clusteredObject
: in the case of unsupervised learning, the clustered object coming from unsupervised.randomUniformFo
or it could be class predictions or labels of the training set (whose, for both, size should be equal to the number of rows of 'X') or an object of class randomUniformForest from which the function will search either OOB predictions or test set predictions, the former having priority.

categorical
: which variables (by names or subscripts) should be considered as categorical? It is currently recommended to let the algorithm find it.

OOB
: if TRUE and clusteredObject contains an OOB object, compute the analysis using the OOB classifier. Useful when there are no test labels.

Details

'clusterAnalysis( )' function is designed to provide a straightforward interpretation of a clustering problem or, in classification, the relation found by the model between features, observations and labels. The main ingredient of the function comes from an object of class importance, which gives the local importance and its derivatives (see Ciss, 2015b). For each observation, one records the features, over all trees, that comes in first, second, ..., and so on position in the terminal nodes where falls the observation. One also records the frequency (number of times divided by number of trees) at which each of the recorded features appears. This gives the first outputted table ('featuresAndObs' see below), with all observations (in rows) and in columns their class or cluster, their position (from first one to last one) and frequency. Note that a position is a local component , named 'localVariable', e.g. 'localVariable1' means that for the i-th observation, the recorded variable was the one that had the highest frequency in the terminal node where the observation has been dropped.

The resulted table is then aggregated, providing in rows, the cluster (or the class) predicted by the model and in columns a component (with the same purpose than an eigenvector), that aggregates variables recorded for all observations of the dataset. We call the table 'clusterAnalysis' since it connects clusters (or class) with dependent variables on each component. The table is granular since one may choose (using options) both number of components and number of variables that will be displayed allowing an easy viewing on variables that matter. In the table, variables in each component are displayed in an decreased order of influence and are usually co-dependent.

The importance (between 0 and 1) of each component is displayed in the third table 'componentAnalysis' allowing the user to define how many components are needed to explain the clusters. More means more variables and the assistance of object of class importance, friends and methods (plot, print, summary) will be needed. Less means easy, but possibly partial, interpretation. Note that the 'clusterAnalysis( )' function is, partially, the summarized view of importance.randomUniformForest.

The three resulted tables comes directly from the modelling and, in classification, do not use training labels but predictions (like clusters are predictions computed by the model). One may want to to know how the modelling is relying on the true nature of data. That is the main interest of the 'clusterAnalysis( )' function. The latter unifies modelling and observed points. The last tables, named 'numericalFeaturesAnalysis' and/or 'categoricalFeaturesAnalysis', aggregates (by a sum, mean, standard deviation or most frequent state) the values of each feature for each predicted cluster or class.
1 - If the clustering scheme is pertinent, then the table must show differences between clusters, by looking to the aggregated values of features.
2 - It must provide insights, especially in a cost-benefit analysis, about the features one needs to take care in order to have more knowledge of each cluster (or class).

3 - In a general case and in Random Uniform Forests, the modelling may not match, even is the clusters are well separated, either the true distribution (number of cases per cluster or class) or the true effects (relations between covariates, relations in each cluster,...) present in the data. The 'clusterAnalysis( )' function, unifying all provided tables, must ensure that:
3.a - true, or at least main or possible, effects are found,
3.b - the approximated distribution generated is enough close to the true one.

From the points above, the main argument is that Random Uniform Forests are almost stochastic. Hence random effects are first expected. If repeated modelling lead to similar results (clusters and within relations), then results are, at least, self-consistent and clusterAnalysis( ) provides a way to look how the modelling enlighten the data.

Value

A list of 3 to 5 data frames with the following components:

`featuresAndObs` a data frame with on rows the observations and on columns a local 'component' (localVariable) representing a meta-variable that accumulates dependent covariates by their names. Components are sorted by level of interactions (from highest to the lowest) then after for each local component is quantified by the influence (named LocalVariableFrequency, from 0 to 1) it gets on each observation over all trees. Note that the data frame is called from 'object' and the dimension depends to the value of 'maxInteractions' when calling `importance.randomUniformForest` .

`clusterAnalysis`
from the data frame above, an aggregated data frame giving a more or less, but compact, granular view of components elements (i.e. variables). Note that for each component, within variables have a co-dependence. Each row is a cluster.

`componentAnalysis`
from the data frame on top, an aggregated data frame giving a more or less, but compact, granular view of component frequencies similar to a "percentage of variance explained" of an eigen value. Each row is a cluster.

`numericalFeaturesAnalysis`
from the data frame on top and 'X', an aggregated data frame giving sum of values of each numerical variable within each cluster.

`categoricalFeaturesAnalysis`
from the data frame on top and the data, an aggregated data frame giving the most frequent value of each categorical variable within each cluster.

Author(s)

Saip Ciss `<saip.ciss@wanadoo.fr>`

References

Ciss, S., 2015b. Variable Importance in Random Uniform Forests. hal-1104751 https://hal.archives-ouvertes.fr/hal-01104751

See Also

unsupervised.randomUniformForest, importance.randomUniformForest

Examples

```
## not run
## load iris data
# data(iris)

## A - Clustering
## run unsupervised modelling, removing labels and committing 4 clusters
# iris.rufUnsupervised = unsupervised.randomUniformForest(iris[,-5], mtry = 1, nodesize = 2)

## as supervised
# iris.rufUnsup2sup = as.supervised(iris.rufUnsupervised, iris[,-5], mtry = 1, nodesize = 2)

## importance
# iris.ruf.importance = importance(iris.rufUnsup2sup, Xtest = iris[,-5], maxInteractions = 4)

## cluster analysis
# iris.ruf.clusterAnalysis = clusterAnalysis(iris.ruf.importance, iris[,-5], components = 3)

## or
# iris.ruf.clusterAnalysis = clusterAnalysis(iris.ruf.importance, iris[,-5], components = 3,
# clusteredObject = iris.rufUnsupervised)

## B - Classification (using OOB evaluation)
# iris.ruf = randomUniformForest(Species ~., data = iris, mtry = 1, nodesize = 2, threads = 1)

## importance
# iris.ruf.importance = importance(iris.ruf, Xtest = iris[,-5], maxInteractions = 4)

## analysis
# iris.ruf.clusterAnalysis = clusterAnalysis(iris.ruf.importance, iris[,-5], components = 3,
# OOB = TRUE, clusteredObject = iris.ruf)
```

---

clusteringObservations

Cluster observations of a (supervised) randomUniformForest object

---

Description

Provide a clustering scheme for observations (in training or test sample) in a (supervised) randomUniformForest object. Observations are clustered using their labels or predictions. The rest of the process is the same than in the unsupervised case. clusteringObservations() may be needed when one wants to know how the classifier can have some troubles in the prediction task, especially when a link between features and decision rule has to be made.

Usage

```
clusteringObservations(object, X,
OOB = TRUE,
predObject = NULL,
importanceObject = NULL,
baseModel = c("proximity", "proximityThenDistance",  "importanceThenDistance"),
MDSmetric = c("metricMDS", "nonMetricMDS"),
endModel = c("MDS", "MDSkMeans", "MDShClust", "SpectralkMeans"),
...)
```

Arguments

object	an object of class randomUniformForest.
X	the dataset to predict.
OOB	set it to TRUE, if one wants to assess the OOB classifier.
predObject	full prediction object expected for test set, if no predictions in 'object'.
importanceObject	object of class importance, if 'baseModel' equals 'importanceThenDistance'.
baseModel	see unsupervised.randomUniformForest.
MDSmetric	see unsupervised.randomUniformForest.
endModel	see unsupervised.randomUniformForest.
...	others parameters of unsupervised.randomUniformForest.

Details

clusteringObservations() function leads to many modelling paradigms, depending on what is called.
- To summarize, the first level is a clustering scheme of the learning task, e.g. 'OOB = FALSE'. In this case, the modelling links the learning with the true labels, providing a picture on how the classifier is "seeing" the training sample. In particular, this representation provides insights on how the classification task might be difficult.
- The second level calls the OOB point of view, e.g. how the classifier is generalizing its current knowledge. In this case, one will simply visualize where errors will probably appear the most.
- The third level involves the prediction task if 'predObject' is provided or if a test set is provided when doing the supervised task (option 'xtest' in randomUniformForest). In this case, the classifier will map its own predictions on the low dimensional space.

In the three levels, there is at least one layer (the dissimilarity matrix), in addition, involved. If one overrides the default 'endModel = MDS' option, then more layers will be added, leading to a more complex object.

Value

An object of class unsupervised, embedding the supervised model.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

Examples

```
## not run
## Vehicle dataset :
# data(Vehicle, package = "mlbench")
# XY = Vehicle
# Y = XY[,"Class"]
# X = XY[, -ncol(XY)]

## 1- learn supervised model and visualize only true labels:

# vehicle.ruf = randomUniformForest(X, as.factor(Y), ntree = 200)

## compute, for more details, importance
# vehicle.importance = importance(vehicle.ruf, Xtest = X)

## clustering object
## 'OOB = FALSE' ensures that OOB classifier will not be called
## One may note that class 'opel' and 'saab' are hard to separate
# vehicle.clusterObject = clusteringObservations(vehicle.ruf, X, OOB = FALSE,
# importanceObject = vehicle.importance, clusters = 4)

# vehicle.clusterObject

## 2 - visualize the OOB point of view
# vehicle.clusterObjectOOB = clusteringObservations(vehicle.ruf, X, OOB = TRUE,
# importanceObject = vehicle.importance, clusters = 4)

## note that true labels are left to their numeric values in order to not modify
## the internal structure of the supervised model
#  vehicle.clusterObjectOOB

## 2.1 use 'sparseProximities = TRUE' (providing a better clustering)
# vehicle.clusterObjectOOB.2 = clusteringObservations(vehicle.ruf, X, OOB = TRUE,
# importanceObject = vehicle.importance, clusters = 4, sparseProximities  = TRUE)

# vehicle.clusterObjectOOB.2

## 3 - learn supervised model, predict and visualize predictions
## set training and test sets

# set.seed(2014)
# train_test = init_values(X, Y, sample.size = 1/2)
# X1 = train_test$xtrain
# Y1 = train_test$ytrain
# X2 = train_test$xtest
# Y2 = train_test$ytest

## learn and predict
# vehicle.new.ruf = randomUniformForest(X1, as.factor(Y1), xtest = X2, ntree = 200)

## compute importance for the test set
# vehicle.new.importance = importance(vehicle.new.ruf, Xtest = X2)
```

```
## clustering predictions : no satisfaction.
## Clustering seems to the reside in a higher dimension
# vehicle.new.clusterObject = clusteringObservations(vehicle.new.ruf, X2,
# importanceObject = vehicle.new.importance, OOB = FALSE, clusters = 4, sparsePrxoimities = TRUE)

## 3.1 - ask help : let's kMeans talk, i.e. can the problem have a dimension as small as 2 ?
## note that the clustering process generates its own predictions and the labels of clusters
## generated have, usually, no direct links with the true labels.
## More precisely cluster 1 is usually not label 1 of the supervised case.
## For the same reasons, importanceObject must be computed in another manner

# vehicle.more.clusterObject = clusteringObservations(vehicle.new.ruf, X2,
# endModel = "MDSkMeans", clusters = 4, OOB = FALSE)

## 3.2 - Clustering is achieved, but what is the cost for the predictions ?
## compare with true test labels : first let's look to the supervised model

# stats.new = model.stats(as.factor(vehicle.new.ruf$predictionObject$majority.vote),
# as.factor(Y2))

## for the clustering model, pairing true test labels and cluster labels
## is done as a shortcut. Finally, even if clustering is achievable
## it does not necessarily correspond to the true labels.

# clusterPredictions = mergeOutliers(vehicle.more.clusterObject)

# 'bus'
# table(clusterPredictions[which(as.numeric(Y2) == 1)])
# hence 'bus' corresponds to the cluster 4 (most frequent values)

# 'opel'
# table(clusterPredictions[which(as.numeric(Y2) == 2)])
# hence 'opel' corresponds to the cluster 3 (most frequent values)

# 'saab'
# table(clusterPredictions[which(as.numeric(Y2) == 3)])
# hence 'saab' corresponds to the cluster 3 (most frequent values)
# or 1, since the former is already taken. We can see that
# the clustering is still having troubles to separate 'saab' and 'opel'
# since both could reside in the same cluster (3).

# 'van'
# table(clusterPredictions[which(as.numeric(Y2) == 4)])
# hence 'van' corresponds to the cluster 2 (most frequent values)

## 3.2 - ask spectral clustering
# vehicle.more.clusterObject = clusteringObservations(vehicle.new.ruf, X2,
# endModel = "SpectralkMeans", clusters = 4, OOB = FALSE)
# clusterPredictions = mergeOutliers(vehicle.more.clusterObject)

## looks better, but finding true classes remains difficult
## one may use more dimensions : 'metricDimension' for the number to compute,
```

```
## 'coordinates' for the dimensions to be clustered.
# vehicle.more.clusterObject = clusteringObservations(vehicle.new.ruf, X2,
# endModel = "SpectralkMeans", clusters = 4, OOB = FALSE, metricDimension = 10, coordinates = 1:5)

## then plot pair of coordinates and choose the one that looks good and use it for a new modelling
# plot(vehicle.more.clusterObject, coordinates = c(1,2))
# plot(vehicle.more.clusterObject, coordinates = c(2,3))

## remove coordinates that are useless
# vehicle.more.clusterObject = rm.coordinates(vehicle.more.clusterObject, c(1,4,5))
# vehicle.more.clusterObject
# plot(vehicle.more.clusterObject)
# clusterPredictions = mergeOutliers(vehicle.more.clusterObject)

## new model should looks better in predicting Vehicle labels...
```

`combineUnsupervised` *Combine Unsupervised Learning objects*

Description

Combine unsupervised learning objects in order to achieve incremental learning. Only the MDS (spectral) points are used before calling a clustering algorithm on all. Note that the function is currently highly experimental with a lack of applications.

Usage

```
combineUnsupervised(...)
```

Arguments

`...`	(enumeration of) objects of class unsupervised, coming from `unsupervised.randomUniformForest`, that needs to be combined.

Value

An object of class unsupervised, which is a list with the following components:

`proximityMatrix`	the resulted dissimilarity matrix.
`MDSModel`	the resulted Multidimensional scaling model.
`unsupervisedModel`	the resulted unsupervised model with clustered observations in unsupervisedModel$cluster.
`largeDataLearningModel`	if the dataset is large, the resulted model that learned a sample of the MDS points, and predicted others points.

gapStatistics	if K-means algorithm has been called, the results of the gap statistic. Otherwise NULL.
rUFObject	Random Uniform Forests object.
nbClusters	Number of clusters found.
params	options of the model.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

See Also

update.unsupervised, modifyClusters, mergeClusters, splitClusters, clusteringObservations, as.supervised

Examples

```
## not run
## Wine Quality Data Set
## http://archive.ics.uci.edu/ml/datasets/Wine+Quality

# data(wineQualityRed)
# X = wineQualityRed[, -ncol(wineQualityRed)]

## 1 - run unsupervised analysis on the first half of dataset

# subset.1 = 1:floor(nrow(X)/2)
# wineQualityRed.model.1 = unsupervised.randomUniformForest(X, subset = subset.1, depth = 5)

## assess roughly the model and visualize
#  wineQualityRed.model.1

# plot(wineQualityRed.model.1)

## 2 - run unsupervised analysis on the second half of dataset
# wineQualityRed.model.2 = unsupervised.randomUniformForest(X, subset = -subset.1, depth = 5)


## 2.1 if less clusters (than in 1) are got, split the one with the highest number of cases
## it is the second cluster in our case
# wineQualityRed.model.2 = splitClusters(wineQualityRed.model.2, 2)

## roughly assess and, eventually, merge and split again (with different seeds) in order
## to be confident about the new clustering
# wineQualityRed.model.2

## 3 - combine
# wineQualityRed.combinedModel =
# combineUnsupervised(wineQualityRed.model.1, wineQualityRed.model.2)

## visualize and plot
```

```
# wineQualityRed.combinedModel
# plot(wineQualityRed.combinedModel)

## compare with the full data and same modelling
# wineQualityRed.model = unsupervised.randomUniformForest(X, depth = 5)

## or increase depth (more computation and default option) for a more detailed model
# wineQualityRed.model = unsupervised.randomUniformForest(X)
```

`ConcreteCompressiveStrength`
Concrete Compressive Strength Data Set

Description

Concrete is the most important material in civil engineering. The concrete compressive strength is a highly nonlinear function of age and ingredients.

Usage

```
ConcreteCompressiveStrength
```

Format

A matrix containing 1030 observations and 9 attributes.

Source

http://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength

References

I-Cheng Yeh, "Modeling of strength of high performance concrete using artificial neural networks," Cement and Concrete Research, Vol. 28, No. 12, pp. 1797-1808 (1998).

`fillNA2.randomUniformForest`
Missing values imputation by randomUniformForest

Description

Impute missing values using randomUniformForest. Each variable containing missing values is, in turn, considered as a responses vector, where non-missing values are training responses and missing values, responses to predict.

Usage

```
fillNA2.randomUniformForest(X, Y = NULL, ntree = 100,
mtry = 1,
nodesize = 10,
categoricalvariablesidx = NULL,
NAgrep = "",
maxClasses = floor(0.01*min(3000, nrow(X))+2),
threads = "auto",
...)
```

Arguments

X	A data frame or matrix of predictors.
Y	response vector, e.g. training labels. Note that it is strongly recommended to use the default values in order to not creating bias, for example if test set needs also to be imputed.
ntree	number of trees to grow for each predictor with missing values. Default value usually works, unless one wants a better level of accuracy.
mtry	number of predictors to try when growing the forest. Default value means full randomized trees will be grown, relying more on convergence of random forests (to achieve a good estimation) and less to the data.
nodesize	smallest number of observations in each leaf node.
categoricalvariablesidx	see randomUniformForest.
NAgrep	which symbol(s) (in case of a data frame), e.g. "?", in addition to "NA" (that is automatically matched) have to be considered as missing values. One can put many symbols, for example 'NAgrep = c(" ", "?")'.
maxClasses	maximal number of classes for a categorical variable. Note that the function will look first the structure of the dataset to detect categorical variables. One has to take care of categories with many classes but not defined as factors. Hence in case of doubt, use a large value of 'maxClasses' is preferable.
threads	how many logical cores to complete data. Default values will let algorithm use all available cores.
...	not currently used.

Details

Algorithm uses randomUniformForest to complete matrix. At the first step, all missing values are identified and rough imputation is done using most frequent (or median) values for each predictor with missing values. Then, these predictors are, one by one, considered as response vectors, where training data are the ones with non missing values (or roughly fixed) and test sample are data whose values are really missing. There is only one iteration (over all predictors) since this version already requires computing time. Hence, it is strongly recommended to use it only if others imputation models do not work or if speed is not mandatory.

Value

a matrix or a data frame containing completed values.

Note

The function will try to render exactly the same structure than the one in the original dataset. Note also that '""' (void character) will be automatically considered as a true category; hence if it is really a missing value one should replace it, for example, by '"?"' and type 'NAgrep = "?"' in the options of the function. See examples.

Note also that rufImpute( ) is an alias of the function.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

Examples

```
## not run

## A - usual case
# get same example as rfImpute() function from randomForest package
# data(iris)
# iris.na <- iris
# set.seed(111)
## artificially drop some data values.
# for (i in 1:4) iris.na[sample(150, sample(20)), i] <- NA

## imputation
# iris.imputed <- fillNA2.randomUniformForest(iris.na, threads = 1)

## model with imputation
# iris.NAfixed.ruf <- randomUniformForest(Species ~ ., iris.imputed,
# BreimanBounds = FALSE, threads = 1)
# iris.NAfixed.ruf

## Compare with true data (OOB evaluation)
# iris.ruf <- randomUniformForest(Species ~ ., iris, BreimanBounds = FALSE, threads = 1)
# iris.ruf

## B - hard case : titanic dataset
## see http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic.html
## for more informations

# URL = "http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/"
# dataset = "titanic3.csv"
# titanic3 = read.csv(paste(URL, dataset, sep =""))

## 1309 observations, 14 variables, 2 classes
# XY = titanic3
# Y = XY,"survived"]
# X = XY,-which(colnames(XY) == "survived")]
```

```
## remove name :
# XX = X = X[,-2]

## first imputation : not working good because missing values are "" and NA
# X.imputed.1 = rufImpute(X)

# head(X.imputed.1)
# head(X)

## 1 - one first has to replace all factors by characters in order to handle ""
# for (j in 1:ncol(X))
# { if (is.factor(X[,j])) { XX[,j] = as.character(X[,j]) } }

## 2 - replace "" by "?"
# XX[which(XX == "", arr.ind  = TRUE)] = "?"

## 3 - impute by including "?" in missing values,
## increasing 'maxClasses' to be sure to match all categorical variables
## setting categorical variables explicitly (they may be integers, factors or characters)
## Note : integers may also be viewed as numerical values

# str(XX) ##gives the type of all variables

# categoricalVariables = vector();  i = 1
# for (j in 1:ncol(X))
# {
# if (class(X[,j]) != "numeric") { categoricalVariables[i] = j; i = i + 1 }
# }

# X.imputed.1 = rufImpute(XX, NAgrep = "?", maxClasses = 1200,
# categorical = categoricalVariables)

# Take a random sample and compare
# idx = sample(nrow(X), 20)
# X[idx,]
# X.imputed.1[idx,]

## modify eventually some numeric values like 'age' to match cases
## use more trees and less randomization to possibly increase accuracy
## at the risk of less consistency.

# X.imputed.2 = rufImpute(XX, NAgrep = "?", maxClasses = 1200, mtry = 4, nodesize = 5,
# ntree = 200, categorical = categoricalVariables)

## 4- assess the imputed matrix : OOB evaluation
## - base model : omit missing values is not possible (too many ones). Roughly impute instead.
# titanic.baseModel.ruf = randomUniformForest(X, as.factor(Y), na.action = "fastImpute",
# categorical = categoricalVariables)

# titanic.baseModel.ruf

## - imputed model
```

```
# titanic.imputedModel.ruf = randomUniformForest(X.imputed.1, as.factor(Y),
# categorical = categoricalVariables)

# titanic.imputedModel.ruf

## roughly (and internal) imputation works better in this case and one should investigate
## models and data to understand possible reasons (one influential feature, many categories, ...)
```

---

generic.cv *Generic k-fold cross-validation*

Description

Performs k-fold cross-validation 'n' times for any specified algorithm, using two of many metrics(test error, AUC, precision,...)

Usage

```
generic.cv(X, Y,
nTimes = 1,
k = 10,
seed = 2014,
regression = TRUE,
genericAlgo = NULL,
specificPredictFunction = NULL,
metrics = c("none", "AUC", "precision", "F-score", "L1", "geometric mean",
"geometric mean (precision)"))
```

Arguments

X	a matrix or dataframe of observations
Y	a vector (a factor for classification) for the observed data.
nTimes	number of times that k-fold cross-validation need to be performed.
k	how many folds ?
seed	the seed for reproducibility.
regression	if TRUE, performs regression.
genericAlgo	wrapper function to embed the algorithm that one needs to assess. One can eventually add options. NULL is only for convenience. Wrapper function is needed to assess cross-validation.
specificPredictFunction	if the assessed model does not support the R generic method 'predict', one has to define here, with a function, how predictions have to be generated.
metrics	One of many other metrics one can call with the standard one, test error (or MSE for regression).

Value

a list with the following components :

testError	the values of test error.
avgError	mean of test error.
stdDev	standard deviation of test error.
metric	values of the other chosen metric.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

Examples

```
## not run
# data(iris)
# Y <- iris$Species
# X <- iris[,-which(colnames(iris) == "Species")]

## 10-fold cross-validation for the randomUniformForest algorithm:

## create the wrapper function (setting 'threads = 1' since data are small)
# genericAlgo.ruf <- function(X, Y) randomUniformForest(X, Y,
# OOB = FALSE, importance = FALSE, threads = 1)

## run
# rUF.10cv.iris <- generic.cv(X, as.factor(Y),
# genericAlgo = genericAlgo.ruf, regression = FALSE)

## 10-fold cross-validation for the randomForest algorithm:

## create the wrapper function
# require(randomForest) || install.packages("randomForest")
# genericAlgo.rf <- function(X, Y) randomForest(X, Y)

## run
# RF.10cv.iris <- generic.cv(X, as.factor(Y),
# genericAlgo = genericAlgo.rf, regression = FALSE)

## 10-fold cross-validation for Gradient Boosting Machines algorithm (gbm package)

## create the wrapper function
# require(gbm) || install.packages("gbm")
# genericAlgo.gbm <- function(X, Y) gbm.fit(X, Y, distribution = "multinomial",
# n.trees = 500, shrinkage = 0.05, interaction.depth = 24, n.minobsinnode = 1)

## create a wrapper for the prediction function of gbm
# nClasses = length(unique(Y))
# specificPredictFunction.gbm <- function(model, newdata)
# {
# modelPrediction = predict(model, newdata, 500)
```

```
# predictions = matrix(modelPrediction, ncol = nClasses )
# colnames(predictions) = colnames(modelPrediction)
# return(as.factor(apply(predictions, 1, function(Z) names(which.max(Z)))))
# }

## run
# gbm.10cv.iris <- generic.cv(X, Y, genericAlgo = genericAlgo.gbm,
# specificPredictFunction = specificPredictFunction.gbm, regression = FALSE)

## 10-fold cross-validation for CART algorithm (rpart package):

# genericAlgo.CART <- function(X, Y)
#{
# ZZ = data.frame(Y, X)
# if (is.factor(Y)) { modelObject = rpart(Y ~., data = ZZ, method = "class", ...) }
# else {  modelObject = rpart(Y ~., data = ZZ, ...) }
# return(modelObject)
#}

# specificPredictFunction.CART <- function(model, newdata)
# predict(model, data.frame(newdata), type= "vector")

# CART.10cv.iris <- generic.cv(X, as.factor(Y), genericAlgo = genericAlgo.CART,
# specificPredictFunction = specificPredictFunction.CART, regression = FALSE)
```

`getTree.randomUniformForest` *Extract a tree from a forest*

Description

get the structure of a tree from a randomUniformForest object.

Usage

```
getTree.randomUniformForest(object, whichTree, labelVar = TRUE)
```

Arguments

`object`	an object of class randomUniformForest.
`whichTree`	which tree have to be printed ?
`labelVar`	if TRUE, names of the variables will be printed, rather than their positions.

Value

a data frame representing the raw tree structure

Note

Trees in randomUniformForest package have almost the same structure than trees in the randomForest package.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

Examples

```
data(iris)
iris.ruf <- randomUniformForest(Species ~ ., data = iris,
threads = 1, ntree = 20, BreimanBounds = FALSE)

# get the 10th tree
OneTree <- getTree.randomUniformForest(iris.ruf, 10)
```

---

importance.randomUniformForest

Variable Importance for random Uniform Forests

Description

Compute object that leads to a full analysis of features (importance, dependency, interactions, selection, ...).

Usage

```
## S3 method for class 'randomUniformForest'
importance(object,
maxVar = 30,
maxInteractions = 3,
Xtest = NULL,
predObject = NULL,
...)
## S3 method for class 'importance'
plot(x,
nGlobalFeatures = 30,
nLocalFeatures = 5,
Xtest = NULL,
whichFeature = NULL,
whichOrder = if (ncol(x$globalVariableImportance) > 5)
{
if (nrow(x$localVariableImportance$obsVariableImportance) > 1000)
"first" else "all"
}
else
```

```
{  if (nrow(x$localVariableImportance) > 1000) "first" else "all" },
    outliersFilter = FALSE,
formulaInput = NULL,
border = NA,
...)
## S3 method for class 'importance'
print(x, ...)
```

Arguments

x, object	an object of class randomUniformForest (for the 'print' and 'plot' method, an object of class 'Importance').
nGlobalFeatures, nLocalFeatures	for the 'plot' method, number of global and local features to show.
maxVar	maximum number of features to display.
maxInteractions	maximum order of interactions. Default value is 3, meaning function will compute interactions for each variable at first order (current variable is supposed to be the most important), second order (current variable is supposed to be the second most important, but the first is unknown) and third order (current variable is supposed to be the third most important, but the first and the second are unknown) and so on, depending on the value of 'maxInteractions'.
Xtest	current matrix used to compute 'object' model. Can be either training or test matrix. If it is the latter, please read below.
whichFeature	for the 'plot' method, the feature (by its name or position) that one need to be assessed. It will be used in partial dependence. Useful only if the feature is not an important one for the model.
whichOrder	for the 'plot' method, the order(s) at which some of the computation, e.g. partial dependence, has to be done. At "first" order, computation is done considering only each feature as the most important. At "second", the feature is considered at the second most important (and the first is unknown). At "all", each feature is assessed considering it is the most important, then the second most (and the first is unknown), then the third most (and the first and the second are unknown), and so on until 'maxInteractions' value is reached. Then one has to first look to this latter value to know at which maximum order a feature will be assessed. Influence of the variable will be a combination of all orders of local importance, depending on the level of interactions specified by 'maxInteractions'.
outliersFilter	for the 'plot' method, do outliers of a feature need to be removed ? if TRUE, observations above 0.95 quantile and below 0.05 quantile will be removed.
formulaInput	for the 'plot' method, if one uses formula, it has to be copy there in order to match figures. Not recommended, since formula can lead to unexpected effects.
border	for the 'plot' method, if positive value, draw borders around barplots.
predObject	if 'object' was computed without an evaluation of test data, one must provide test data in 'Xtest' option and a full prediction object (calling option type = "all", when using predict() function. See examples).
...	others options currently not used.

Details

'Importance' retrieves (or computes) global importance from 'object' which is "variable importance from the model and for training data based on most predictive (by score) and discriminant(by class and class.frequency) features".

Then it computes, either for train or test data and for at least two orders :
1 - 'interactions', which show a kind of dependence between two features, for all pairs of variables. Interactions shows links between variables by using informations at the leaf level. Most important (highly predictive) features usually have more interactions, but a low predictive feature can have many interactions that provide insights about the big picture. Interactions are currently computed and plotted at the first and second order.

2 - 'overall interactions' that leads to "variable importance based on interaction". It shows how response values can be explained, using more than just predictive features and it is an aggregated function of interactions.

3 - 'partial dependence', that leads to show how response values are evolving according to one feature, and knowing the distribution of all others. See `partialDependenceOverResponses`.

4 - 'variable importance over labels' (in classification), that shows, on the same figure, how each important feature is matching labels from the point of view of the model. More precisely, this object acts a conditional importance. First a label is selected. Then importance is assessed conditionally. Hence features take influence, once the class is known (or predicted), meaning that there possibly exists more influential features (that lead to choose a class rather than another). In regression, the plotted object is called 'Dependence on most important predictors'.

5 - 'Importance( )' function allows a deeper analysis, depending on options and functions like `partialDependenceBetweenPredictors`, `partialImportance` or `clusterAnalysis`.

6 - To neutralize bias induced by correlated variables, it is strongly recommended :
a - to use `randomUniformForest` with 'mtry = 1', e.g. as a purely random forest (nodes are built and chosen completely at random)
b - then, to compute importance and use the resulted object as a benchmark of Variable Importance. If features have influence in the purely random case, then they also will have one in the weaker random case... The reciprocal is not valid.

Summarized, importance object tells (or leads to, using others functions) "which, how, when and where" features affect the response and interact each other. Note that importance strongly depends of hyper-parameters of the model (mtry, depth, nodesize, ...) and one has to take care. Categorical features are treated like numeric ones (default option) except if 'categorical' option is specified in `randomUniformForest`. 'categorical' option leads to a better assessment but might hurt accuracy. One possible reason is that the cross-entropy function (optimization criterion) would be more efficient for variables with high variability.

Value

an object of class importance.

Note

Please note that 'plot' method of importance object will produce many plots (5) and a prompt entering a loop which will wait user input to plot one by one partial dependencies for most important features. One has to first tile windows in R, to get the big picture. There is currently not a generic way (at least, I did not find it) to automatically tile windows in R in all platforms.

Author(s)

Saip CISS <saip.ciss@wanadoo.fr>

References

Ciss, S., 2015b. Variable Importance in Random Uniform Forests. hal-1104751 https://hal.archives-ouvertes.fr/hal-01104751

See Also

partialDependenceOverResponses, partialDependenceBetweenPredictors, partialImportance, clusterAnalysis

Examples

```
## not run
## 1 - Synthetic data:
## generate data

# set.seed(2014)
# n = 1000; p = 100
# X = simulationData(n, p)
# X = fillVariablesNames(X)
# epsilon1 = runif(n,-1,1); epsilon2 = runif(n,-1,1)
# rule = 2*(X[,1]*X[,2] + X[,3]*X[,4]) + epsilon1*X[,5] + epsilon2*X[,6]
# Y = as.factor(ifelse(rule > mean(rule), "+","-"))

## X1, X2, X3, X4 are the most important ones since they generate the labels
## Other ones are noise.

## run model
# synth.model.ruf = randomUniformForest(X, as.factor(Y))

## a - summary of the model provides (by default) global variable importance
## (most predictive and most discriminant ones (in the case of classification)
# summary(synth.model.ruf)

## b - get details of importance : local variable importance and partial dependencies
## we choose 'maxInteractions' between covariates to be equal to 2
# synth.importance.ruf = importance(synth.model.ruf, Xtest = X, maxInteractions = 2)

## show interactions, variable importance based on interactions and variable importance
## conditionally to each label
# synth.importance.ruf
```

```
## c - plot details :
## - adds partial dependence, showing for which values of a variable, the class is changing,
## - displays interactions, showing how influential variables are covering all the possible
## interactions
## - displays variable importance based on interactions, showing their relative influence
## - displays variable importance over labels, showing how each variable is influencing a class
## and how it is discriminant in the separation between two or more classes.

# plot(synth.importance.ruf, Xtest = X, nLocalFeatures = 6)

## d - complete analysis with other tools, for example clusterAnalysis() :
## synth.Analysis.ruf = clusterAnalysis(synth.importance.ruf, X, components = 3,
## clusteredObject = synth.model.ruf, OOB = TRUE)

## or clusteringObservations() : table and plot
# synth.Analysis2.ruf = clusteringObservations(synth.model.ruf, X, OOB = TRUE,
# importanceObject = synth.importance.ruf)

## 2 - Importance for Classification and Regression (with formula)
####  Classification

# data(iris)
# iris.ruf <- randomUniformForest(Species ~ ., data = iris, threads = 1)

## global importance :
# summary(iris.ruf)

# iris.ruf.importance <- importance(iris.ruf, Xtest = iris, threads = 1)

## get importance summary
# iris.ruf.importance

## visualizing all in one
# plot(iris.ruf.importance, Xtest = iris)

#### Regression

# data(airquality)
# airquality.data = airquality

## impute NA
# airquality.NAimputed <- fillNA2.randomUniformForest(airquality.data)

## compute model
# ozone.ruf <- randomUniformForest(Ozone ~ ., data = airquality.NAimputed, threads = 1)

# ozone.ruf
# summary(ozone.ruf)
# ozone.ruf.importance <- importance(ozone.ruf, Xtest = airquality.NAimputed, threads = 1)

## visualization: in case of formula, 'formulaInput' is needed for the 'plot' method
# plot(ozone.ruf.importance, Xtest = airquality.NAimputed, formulaInput = ozone.ruf$formula)
```

```
## 3 - Importance for Classification and Regression without formula (more usual and recommended)

#### Classification: "car evaluation" data
## (http://archive.ics.uci.edu/ml/datasets/Car+Evaluation)

# data(carEvaluation)
# car.data <- carEvaluation
# n <- nrow(car.data)
# p <- ncol(car.data)

# trainTestIdx <- cut(sample(1:n, n), 2, labels= FALSE)

## training examples
# car.data.train <- car.data[trainTestIdx == 1, -p]
# car.class.train <- as.factor(car.data[trainTestIdx == 1, p])

## test data
# car.data.test <- car.data[trainTestIdx == 2, -p]
# car.class.test <- as.factor(car.data[trainTestIdx == 2, p])

## compute model : train then test in the same function
## option 'categorical' may be used for categorical variables and should be consistent
## with Variable Importance but may lead to less accuracy.

# car.ruf <- randomUniformForest(car.data.train, car.class.train,
# xtest = car.data.test, ytest = car.class.test, categorical = "all")
# car.ruf

## global importance: note that 'safety' does not appear to be an important feature
## in the barplot but in the table, it is the most important of unacceptable (unacc) cars.
# summary(car.ruf)

## interactions and local importance tell most of the story...
# car.ruf.importance <- importance(car.ruf, Xtest = car.data.train, threads = 1)

## ...that can be used to explain train data
# plot(car.ruf.importance, Xtest = car.data.train)

## or explain test data
# car.ruf.importance.test <- importance(car.ruf, Xtest = car.data.test, threads = 1)
# plot(car.ruf.importance.test, Xtest = car.data.test)

#### Regression : "Concrete Compressive Strength" data
## (http://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength)

# data(ConcreteCompressiveStrength)
# ConcreteCompressiveStrength.data = ConcreteCompressiveStrength

# n <- nrow(ConcreteCompressiveStrength.data)
# p <- ncol(ConcreteCompressiveStrength.data)

# trainTestIdx <- cut(sample(1:n, n), 2, labels= FALSE)
```

```
## train examples
# Concrete.data.train <- ConcreteCompressiveStrength.data[trainTestIdx == 1, -p]
# Concrete.responses.train <- ConcreteCompressiveStrength.data[trainTestIdx == 1, p]

## test data
# Concrete.data.test <- ConcreteCompressiveStrength.data[trainTestIdx == 2, -p]
# Concrete.responses.test <- ConcreteCompressiveStrength.data[trainTestIdx == 2, p]

## model
# Concrete.ruf <- randomUniformForest(Concrete.data.train, Concrete.responses.train,
# featureselectionrule = "L1", threads = 1)
# Concrete.ruf

## predictions : option ' type = "all" ' is needed to manually assess importance of a test set
# Concrete.ruf.pred <- predict(Concrete.ruf, Concrete.data.test, type = "all")

## more interactions
# Concrete.ruf.importance <- importance(Concrete.ruf, Xtest = Concrete.data.test,
# maxInteractions = 6, predObject = Concrete.ruf.pred, threads = 1)

## or more features to plot
# plot(Concrete.ruf.importance, nLocalFeatures = 7, Xtest = Concrete.data.test)
```

`init_values` *Training and validation samples from data*

Description

Draw training and test samples from data. Samples can be accessed by subsctioting original data or by their own references.

Usage

```
init_values(X, Y = NULL, sample.size = 0.5,
data.splitting = "ALL",
unit.scaling = FALSE,
scaling = FALSE,
regression = FALSE)
```

Arguments

`X`	a matrix or dataframe to be splitted in training and validation sample
`Y`	a response vector for the observed data.
`sample.size`	size of the needed training sample in proportion of the nulber of observations in original data.
`data.splitting`	not currently used.

unit.scaling	if TRUE, scale all data in X between 0 and 1, if they are all positive, or between -1 and 1.
scaling	if TRUE, centers and scales data, so each variable willhave mean 0 abd variance 1.
regression	if TRUE and scaling = TRUE, Y will also be scaled.

Value

a list with the following components :

xtrain	a matrix or data frame representing the training sample.
ytrain	a response vector representing the training responses according to the training sample.
xtest	a matrix or data frame representing the validation sample.
ytest	a response vector representing the validation responses according to the validation sample.
train_idx	subscripts of the training sample.
test_idx	subscripts of the validation sample.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

Examples

```
data(iris)
Y <- iris$Species
X <- iris[,-which(colnames(iris) == "Species")]
trainingAndValidationsamples <- init_values(X, Y, sample.size = 0.5)

Xtrain = trainingAndValidationsamples$xtrain
Ytrain = trainingAndValidationsamples$ytrain
Xvalid = trainingAndValidationsamples$xtest
Yvalid = trainingAndValidationsamples$ytest
```

---

internalFunctions *All internal functions*

Description

Internal functions for random Uniform Forests

mergeClusters *Merge two arbitrary, but adjacent, clusters*

Description

merge, on the flight, two adjacent clusters in order to allow better clustering scheme, if needed, and to avoid new computation of the unsupervised mode.

Usage

```
mergeClusters(object, whichOnes)
```

Arguments

object	an object of class unsupervised.
whichOnes	which clusters are needed to be merged ? Should be two adjacent ones.

Value

An object of class unsupervised.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

See Also

modifyClusters

Examples

```
## not run

## Crabs data
# data(crabs, package = "MASS")

## and more about
# ?crabs

## model : commit 4 clusters
# crabs.rufUnsupervised = unsupervised.randomUniformForest(crabs,
# categoricalvariablesidx = "all", nodesize = 5, threads = 1, clusters = 4)

## visualize clusters and merge adjacent clusters
# plot(crabs.rufUnsupervised)

## we can first merge clusters 1 and 4
## note that clusters may change if run again
# crabs.rufUnsupervisedNew = mergeClusters(crabs.rufUnsupervised, c(1,4))
```

```
## one can assess the fitting, comparing old and new model
# crabs.rufUnsupervised
# crabs.rufUnsupervisedNew

## visualize new model
# plot(crabs.rufUnsupervisedNew)

## merge new clusters 1 and 2 and look if it will be better
# crabs.rufUnsupervisedNewest = mergeClusters(crabs.rufUnsupervisedNew, c(1,2))

# crabs.rufUnsupervisedNewest
# plot(crabs.rufUnsupervisedNewest)

## NOTE : mergeClusters() provides choice on how to choose and assess clusters
## using simply visualization.
```

`model.stats` *Common statistics for a vector (or factor) of predictions and a vector (or factor) of responses*

Description

Given a vector of predictions and a vector of responses, provide some statistics and plots like AUC, AUPR, confusion matrix, F1-score, geometric mean, residuals, mean squared and mean absolute error.

Usage

```
model.stats(predictions, responses, regression = FALSE, OOB = FALSE, plotting = TRUE)
```

Arguments

`predictions`	a vector (or factor, if classification) of predictions.
`responses`	a vector (or factor, if classification) of responses of the same length than 'predictions'.
`regression`	if FALSE, considered arguments are treated as a classification task.
`OOB`	if TRUE, expects 'prediction' to be an object of class randomUniformForest (with option 'OOB' enabled) in order to assess OOB predictions.
`plotting`	if TRUE, displays graphics. Set it to FALSE in the case of a regression with large datasets.

Value

print and plot metrics.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

Examples

```
## not run
## Classification : synthetic data

# set.seed(2014)
# n = 1000
# p = 100
# X = simulationData(n, p)
# X = fillVariablesNames(X)
# epsilon1 = runif(n,-1,1)
# epsilon2 = runif(n,-1,1)
# rule = 2*(X[,1]*X[,2] + X[,3]*X[,4]) + epsilon1*X[,5] + epsilon2*X[,6]
# Y = as.factor(ifelse(rule > mean(rule), "+","-"))

# training and test sets

# train_test = init_values(X, Y, sample.size = 1/2)
# X1 = train_test$xtrain
# Y1 = train_test$ytrain
# X2 = train_test$xtest
# Y2 = train_test$ytest

# train model
# synth.ruf = randomUniformForest(X1, as.factor(Y1))

# evaluates OOB predictions
# statsOOB.pred.synth.ruf = model.stats(synth.ruf, as.factor(Y1), OOB = TRUE)

# predict
# pred.synth.ruf = predict(synth.ruf, X2)

# statistics : produces also two plots
# stats.pred.synth.ruf = model.stats(pred.synth.ruf, as.factor(Y2))

# or, trick, do all in two lines
# synth.ruf = randomUniformForest(X1, as.factor(Y1), xtest = X2, ytest = as.factor(Y2))
# stats.pred.synth.ruf = model.stats(synth.ruf, as.factor(Y2))

## regression : synthetic data
# Y = rule
# Y1 = Y[train_test$train_idx]
# Y2 = Y[train_test$test_idx]

# synth.ruf = randomUniformForest(X1, Y1)
# statsOOB.pred.synth.ruf = model.stats(synth.ruf, Y1, OOB = TRUE, regression = TRUE)
# pred.synth.ruf = predict(synth.ruf, X2)
# stats.pred.synth.ruf = model.stats(pred.synth.ruf, Y2, regression = TRUE)
```

modifyClusters *Change number of clusters (and clusters shape) on the fly*

Description

Modify on the fly the number of clusters in the unsupervised mode of Random Uniform Forests. Once the unsupervised model is built, one can change the clustering scheme, adding or merging clusters without computing again the model. Average silhouette coefficient or variance between clusters are automatically adjusted in order to assess the new scheme.

Usage

```
modifyClusters(object, decreaseBy = NULL, increaseBy = NULL, seed = 2014)
```

Arguments

object	an object of class unsupervised.
decreaseBy	decrease the current number of clusters by the value of 'decreaseBy'.
increaseBy	decrease the current number of clusters by the value of 'increaseBy'.
seed	see unsupervised.randomUniformForest.

Value

An object of class unsupervised.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

See Also

mergeClusters

Examples

```
## not run
## load iris data
# data(iris)

## run unsupervised modelling, removing labels and committing 4 clusters
# iris.rufUnsupervised = unsupervised.randomUniformForest(iris[,-5], threads = 1, clusters = 4)

## view a summary
# iris.rufUnsupervised

## plot clusters
# plot(iris.rufUnsupervised)
```

```
## modify clusters, decreasing them by  one
# iris.rufUnsupervisedNew = modifyClusters(iris.rufUnsupervised, decreaseBy = 1)

## assess fitting comparing average Silhouette before and after
# iris.rufUnsupervisedNew

## plot to see the new clusters
# plot(iris.rufUnsupervisedNew)
```

`partialDependenceBetweenPredictors` *Partial Dependence between Predictors and effect over Response*

Description

Computes partial dependence between two predictors, and their effects on response values.

Usage

```
partialDependenceBetweenPredictors(Xtest, importanceObject, features,
whichOrder = c("first", "second", "all"),
perspective = FALSE,
outliersFilter = FALSE,
maxClasses = max(10, which.is.factor(Xtest[,features, drop = FALSE],
count = TRUE)),
bg = "grey")
```

Arguments

`Xtest`	a matrix or data frame specifying test (or train) data.
`importanceObject`	an object of class importance.
`features`	features that one needs to see dependence with responses (either train responses or predicted values).
`whichOrder`	at which order, partial dependence does it need to be computed ?
`perspective`	if TRUE, plot dependence in 3D with animation or not. Note that model will lead to interpolation. If one needs extrapolation, 'outputperturbationsampling' option can help?
`outliersFilter`	filter outliers ?
`maxClasses`	for variables with discrete values that need to be treated as categorical for a better visualization and that have more than 'maxClasses' unique values. Or for categorical variables that one knows to be categorical but whose data are stored as a R matrix.
`bg`	background color for the plot. Type 'bg = "none"' to get a white background.

Details

partial dependence shows how response values are evolving depending to a pair of variables and knowing the distribution of all others covariates. Note that it is essential to first have a view on variable importance object. Steps can be given as this :
1- get importance object (and plot it), that shows almost all objects that could explain the link between features and response.
2- compute dependence between two target features and response values. Link this point with step 1 to obtain precise effects on response feature.

Note that the function shows both the dependence between the pair of variables and the effect over the whole response values.

Value

A matrix containing values of the two features and values of expected conditional response, for regression. For classification, responses column is whether or not the values of the two features share the same class. partialDependenceBetweenPredictors() function also returns a set of figures representing how the dependency between the two features is affecting classes of the problem (or responses values in case of regression). It also returns a measure of dependence between the two predictors at first (one of the two feature is supposed to be the most important one in the data) and second order (one of the two features is supposed to be the second most important one).

Note

Please note that many plots (4 or 5, depending on task and option) will be produced. One has to first tile windows in R, to get the big picture. There is currently not a generic way (at least, I did not find it) to automatically tile windows in R in all platforms.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

See Also

partialImportance

Examples

```
## not run

#### Classification: "car evaluation" data
## (http://archive.ics.uci.edu/ml/datasets/Car+Evaluation)
# data(carEvaluation)
# car.data <- carEvaluation

# n <- nrow(car.data)
# p <- ncol(car.data)

# trainTestIdx <- cut(sample(1:n, n), 2, labels= FALSE)

## train examples
```

```
# car.data.train <- car.data[trainTestIdx == 1, -p]
# car.class.train <- as.factor(car.data[trainTestIdx == 1, p])

## test data
# car.data.test <- car.data[trainTestIdx == 2, -p]
# car.class.test <- as.factor(car.data[trainTestIdx == 2, p])

## compute model : train then test in the same function.
## use 'categorical' option to better handle  variable importance
# car.ruf <- randomUniformForest(car.data.train, car.class.train,
# xtest = car.data.test, ytest = car.class.test, categorical = "all")
# car.ruf

## get for example two most important features, using the table
## we choose "buying" and "safety"
# summary(car.ruf)

## compute importance with deepest level of interactions to get enough points
# car.ruf.importance <- importance.randomUniformForest(car.ruf,
# Xtest = car.data.train, maxInteractions = 6)

## compute and plot partial dependence between "buying" and "safety" on train data
## is interaction leading to the same class (underlying structure)?
## is dependence linear or, for categorical variables, how is the effect of cross-tabulating
## the variables over each class ?
## for which values of the pair is the dependence most effective ?

# pDbetweenPredictors.car.buyingAndSafety <- partialDependenceBetweenPredictors(car.data.train,
# car.ruf.importance, c("buying", "safety"), whichOrder = "all")

## Interpretation :
## 1 - if "safety" is 'low', with average frequency of (around) 63 percent the
## label of the evaluated car would be the same than the one coming from any buying category
## meaning that, the buying price will not have influence.
## If safety is 'high', the evaluated car will depend on the buying price...
## In the same sense, if the buying price is 'very high', the label of the evaluated car
## will also depend to the safety category most of the time.
## For a high (or lower) buying price, it will be much less dependent.
## But:
## 2 - the "heatmap..." states that confidence (more data) will be greater for cases
## where buying price is very high (with safety low) or high (with safety high)
## 3 - Hence looking to the "dependence between predictors",  shows that if buying price is high,
## it means that safety must be also high before one draws a conclusion.
## 4 - Looking "Variable Importance based on interactions" tells which variable dominates
## the other when a car is evaluated.

#### Regression : "Concrete Compressive Strength" data
## (http://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength)

# data(ConcreteCompressiveStrength)
# ConcreteCompressiveStrength.data = ConcreteCompressiveStrength

# n <- nrow(ConcreteCompressiveStrength.data)
```

```
# p <- ncol(ConcreteCompressiveStrength.data)

# set.seed(2015)
# trainTestIdx <- cut(sample(1:n, n), 2, labels= FALSE)

## train examples
# concrete.data.train <- ConcreteCompressiveStrength.data[trainTestIdx == 1, -p]
# concrete.responses.train <- ConcreteCompressiveStrength.data[trainTestIdx == 1, p]

## test data
# concrete.data.test <- ConcreteCompressiveStrength.data[trainTestIdx == 2, -p]
# concrete.responses.test <- ConcreteCompressiveStrength.data[trainTestIdx == 2, p]

## model
# concrete.ruf <- randomUniformForest(concrete.data.train, concrete.responses.train,
# featureselectionrule = "L1")
# concrete.ruf

## Assessing test set only
## importance at the deepest level of interactions
# concrete.ruf.importance <- importance.randomUniformForest(concrete.ruf,
# Xtest = concrete.data.test, maxInteractions = 8)

## compute and plot partial dependence between "Age" and "Cement",
## without 3D representation and with filter upon outliers

# pDbetweenPredictors.concrete.cementAndAge <-
# partialDependenceBetweenPredictors(concrete.data.test,
# concrete.ruf.importance, c("Age", "Cement"), whichOrder = "all", outliersFilter = TRUE)

## compute and plot partial dependence between "Age" and "Cement",
## with 3D representation (slower)

# pDbetweenPredictors.concrete.cementAndAge <-
# partialDependenceBetweenPredictors(concrete.data.test,
# concrete.ruf.importance, c("Age", "Cement"), whichOrder = "all", perspective = TRUE)
```

`partialDependenceOverResponses`

Partial Dependence Plots and Models

Description

Computes partial dependence between expected conditional response and all values of its target feature, knowing the distribution of all others features in the data (e.g. marginal effect of the target feature over the response)

Usage

```
partialDependenceOverResponses(Xtest, importanceObject,
```

```
whichFeature = NULL,
whichOrder = c("first", "second", "all"),
outliersFilter = FALSE,
plotting = TRUE,
followIdx = FALSE,
maxClasses = if (is.null(whichFeature)) { 10 } else { max(10,
which.is.factor(Xtest[, whichFeature, drop = FALSE], count = TRUE)) },
bg = "lightgrey")
```

Arguments

Xtest	a matrix or data frame specifying test (or train) data.
importanceObject	an object of class importance.
whichFeature	feature that one needs to see dependence with responses (either train responses or predicted values).
whichOrder	at which order, partial dependence does it need to be computed ?
outliersFilter	filter outliers ?
plotting	plot partial dependence ?
followIdx	not currently used.
maxClasses	for variables with discrete values that need to be treated as categorical for a better visualization and that have more than 'maxClasses' unique values. Or for categorical variables that one knows to be categorical but whose data are stored as a R matrix.
bg	background color for the plot. Type 'bg = "none"' to get a white background.

Details

Partial dependence shows how expected conditional response is evolving with its target feature, knowing the distribution of all others features. For example, if one wants to know how response is moving, on average, depending on one target feature and for all possible values of the others features. For Classification, partial dependence shows the dependence of each important variable over classes. One key advantage of partial dependence is to allow extrapolation for ensemble models. Note that this version is inspired by Friedman (2001) ideas, but uses a different implementation.

Value

a matrix containing feature values and expected conditional responses.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

References

Friedman, J.H., 2001. *Greedy function approximation: A gradient boosting machine*. Ann. Statist. 29, 1189-1232.

See Also

partialDependenceBetweenPredictors, partialImportance

Examples

```
## not run
## NOTE:  please remove comments to run
#### Classification: "car evaluation" data (http://archive.ics.uci.edu/ml/datasets/Car+Evaluation)
# data(carEvaluation)
# car.data <- carEvaluation

# n <- nrow(car.data)
# p <- ncol(car.data)

# trainTestIdx <- cut(sample(1:n, n), 2, labels= FALSE)

## train examples
# car.data.train <- car.data[trainTestIdx == 1, -p]
# car.class.train <- as.factor(car.data[trainTestIdx == 1, p])

## test data
# car.data.test <- car.data[trainTestIdx == 2, -p]
# car.class.test <- as.factor(car.data[trainTestIdx == 2, p])

## compute model : train then test in the same function
# car.ruf <- randomUniformForest(car.data.train, car.class.train,
# xtest = car.data.test, ytest = car.class.test, threads = 2)
# car.ruf

## compute importance
# car.ruf.importance <- importance.randomUniformForest(car.ruf,
# Xtest = car.data.train, threads = 2)

## plot partial dependence, at all orders, between classes and 'safety' feature on train data.
## Note that data are mainly categorical, ordinal and characters (e.g. v-high, high, med, low
## for 'safety'). Model deals internally with characters and compute them as numerical values
## (e.g.  "v-high, high, med, low" leads to 4,1,3,2.)

## how safety affects labels, knowing all others features ?
# pD.car.safety <- partialDependenceOverResponses(car.data.train, car.ruf.importance,
# whichFeature = "safety", whichOrder = "all")

## What's happen at first order ?
# pD.1rstOrder.car.safety <- partialDependenceOverResponses(car.data.train, car.ruf.importance,
# whichFeature = "safety", whichOrder = "first")

## plot partial dependence, at first order, default,
## between classes and 'buying' feature on train data
# pD.1rstOrder.car.buying <- partialDependenceOverResponses(car.data.train, car.ruf.importance,
# whichFeature = "buying")

## plot partial dependence, at second order, between classes and 'buying' feature on train data.
```

```
## Second order means 'buying' feature is supposed to be the second most important feature
## and first one is unknown.
# pD.2ndOrder.car.buying <- partialDependenceOverResponses(car.data.train, car.ruf.importance,
# whichFeature = "buying", whichOrder = "second")

## if one wants to assess test set, e.g. test responses are unkown
# car.ruf <- randomUniformForest(car.data.train, car.class.train,
# xtest = car.data.test, threads = 2)
# car.ruf

## compute importance object deeper (increasing level of interactions) and on test data
# car.ruf.importance <- importance.randomUniformForest(car.ruf, Xtest = car.data.test,
# maxInteractions = 6, threads = 2)

# pD.1rstOrder.car.safety <- partialDependenceOverResponses(car.data.test, car.ruf.importance,
# whichFeature = "safety")

# pD.2ndOrder.car.buying <- partialDependenceOverResponses(car.data.test, car.ruf.importance,
# whichFeature = "buying", whichOrder = "second")

# pD.allOrders.car.priceOfMaintenance <- partialDependenceOverResponses(car.data.test,
# car.ruf.importance, whichFeature = "priceOfMaintenance", whichOrder = "all")

# pD.2ndOrder.car.priceOfMaintenance <- partialDependenceOverResponses(car.data.test,
# car.ruf.importance, whichFeature = "priceOfMaintenance", whichOrder = "second")

#### Regression : "Concrete Compressive Strength" data
## (http://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength)

# data(ConcreteCompressiveStrength)
# ConcreteCompressiveStrength.data = ConcreteCompressiveStrength

# n <- nrow(ConcreteCompressiveStrength.data)
# p <- ncol(ConcreteCompressiveStrength.data)

# trainTestIdx <- cut(sample(1:n, n), 2, labels= FALSE)

## train examples
# Concrete.data.train <- ConcreteCompressiveStrength.data[trainTestIdx == 1, -p]
# Concrete.responses.train <- ConcreteCompressiveStrength.data[trainTestIdx == 1, p]

## model
# Concrete.ruf <- randomUniformForest(Concrete.data.train, Concrete.responses.train,
# featureselectionrule = "L1", threads = 2)
# Concrete.ruf

## importance object only for train examples
# Concrete.ruf.importance <- importance.randomUniformForest(Concrete.ruf,
# Xtest = Concrete.data.train, maxInteractions = 4, threads = 2)

## partial dependence only for train examples :
## at all orders
# pD.Concrete.Cement <- partialDependenceOverResponses(Concrete.data.train,
```

```
# Concrete.ruf.importance, whichFeature = "Cement", whichOrder = "all")

# pD.Concrete.Age <- partialDependenceOverResponses(Concrete.data.train,
# Concrete.ruf.importance, whichFeature = "Age", whichOrder = "all")

## at first order
# pD.1rstOrder.Concrete.Water <- partialDependenceOverResponses(Concrete.data.train,
# Concrete.ruf.importance, whichFeature = "Water")
```

`partialImportance` *Partial Importance for random Uniform Forests*

Description

Describe what features are the most important for one specifically class (in case of classification) or explain features that are affecting, the most, variability of the response (for regression), either for train or test sample.

Usage

```
partialImportance(X, object,
whichClass = NULL,
threshold = NULL,
thresholdDirection = c("low", "high"),
border = NA,
nLocalFeatures = 5)
```

Arguments

`X`	a matrix or data frame specifying test (or train) data.
`object`	an object of class importance.
`whichClass`	for classification only. The index of the class that needs to be computed.
`threshold`	for regression only. The threshold point. Partial importance will compute importance below or above this point (see below).
`thresholdDirection`	where importance does it need to be fitted ? "low" will be lower than 'threshold' and "high" will be above.
`border`	visualization option. Draw border around barplots ?
`nLocalFeatures`	how many features does it need to be assessed ?

Details

Partial importance must be used in conjunction with importance object, since it explains what features have influence on a class (or on the variability of the response) but not how (that can be lower, higher or all values of one or many features). Partial Importance produces final (and local) rules that lead to the decision (observed or predicted class, observed or predicted value in cas of regression).

Value

A vector containing relative influence of the most important features in a decreasing order.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

Examples

```
## not run
## NOTE : please remove comments to run
#### Classification: "car evaluation" data (http://archive.ics.uci.edu/ml/datasets/Car+Evaluation)

# data(carEvaluation)
# car.data = carEvaluation

# n <- nrow(car.data)
# p <- ncol(car.data)

# trainTestIdx <- cut(sample(1:n, n), 2, labels= FALSE)

## train examples
# car.data.train <- car.data[trainTestIdx == 1, -p]
# car.class.train <- as.factor(car.data[trainTestIdx == 1, p])

## test data
# car.data.test <- car.data[trainTestIdx == 2, -p]
# car.class.test <- as.factor(car.data[trainTestIdx == 2, p])

## compute model : train and predict in the same function
# car.ruf <- randomUniformForest(car.data.train, car.class.train, xtest = car.data.test)
# car.ruf

## compute importance object deeper (increasing level of interactions) on test data
# car.ruf.importance <- importance(car.ruf, Xtest = car.data.test,
# maxinteractions = 6, threads = 2)

## get partial importance for the three most important features
## in test set for "unacceptable" cars
# car.ruf.partialImportance.unacc <- partialImportance(car.data.test,
# car.ruf.importance, whichClass = "unacc")

## get partial importance for the three most important features in test set for "acceptable" cars
# car.ruf.partialImportance.acc <- partialImportance(car.data.test, car.ruf.importance,
# whichClass = "acc")

#### Regression : "Concrete Compressive Strength" data
## (http://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength)

# data(ConcreteCompressiveStrength.data)
# ConcreteCompressiveStrength.data = ConcreteCompressiveStrength
```

```
# n <- nrow(ConcreteCompressiveStrength.data)
# p <- ncol(ConcreteCompressiveStrength.data)

# trainTestIdx <- cut(sample(1:n, n), 2, labels= FALSE)

## train examples
# concrete.data.train <- ConcreteCompressiveStrength.data[trainTestIdx == 1, -p]
# concrete.responses.train <- ConcreteCompressiveStrength.data[trainTestIdx == 1, p]

## test data
# concrete.data.test <- ConcreteCompressiveStrength.data[trainTestIdx == 2, -p]
# concrete.responses.test <- ConcreteCompressiveStrength.data[trainTestIdx == 2, p]

## model
# concrete.ruf <- randomUniformForest(concrete.data.train, concrete.responses.train,
# featureselectionrule = "L1")
# concrete.ruf

## importance on train data
# concrete.ruf.importance <- importance.randomUniformForest(concrete.ruf,
# Xtest = concrete.data.train, maxInteractions = 8, threads = 2)

## partial importance for concrete compressive strength higher than 40.
# concrete.ruf.partialImportance <- partialImportance(concrete.data.train,
# concrete.ruf.importance, threshold = 40, thresholdDirection = "high")

## partial importance for concrete compressive strength lower than 30.
# concrete.ruf.partialImportance <- partialImportance(concrete.data.train,
# concrete.ruf.importance, threshold = 30, thresholdDirection = "low")
```

---

plotTree *Plot a Random Uniform Decision Tree*

Description

plot the tree structure, showing nodes, variables, cut-points and predictions.

Usage

```
plotTree(treeStruct,
rowNum = 1,
height.increment = 1,
maxDepth = 100,
fullTree = FALSE,
xlim = NULL,
ylim= NULL,
center = TRUE)
```

Arguments

treeStruct	a data frame same at the output of getTree.randomUniformForest() function.
rowNum	internal option not currently used.
height.increment	internal option not currently used.
maxDepth	internal option not currently used.
fullTree	if TRUE, draw full tree, but if the tree is deep or large, one will not see all. if FALSE, only parts of the tree will be shown but visualization will be clearer. Note that using fullTree = TRUE, simultaneously with 'xlim' and 'ylim', one can see the whole tree (but not the details).
xlim	a vector of two values, giving the width of the tree to show. One can, then, explore tree structure more in details.
ylim	a vector of two values, giving the depth of the tree to show. One can, then, explore tree structure more in details.
center	if TRUE, the tree will be centered.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

Examples

```
# not run
data(airquality)
ozone.ruf <- randomUniformForest(Ozone ~ ., data = airquality,
ntree = 20, BreimanBounds = FALSE, threads = 1)

OneTree <- getTree.randomUniformForest(ozone.ruf, 10)

# plotTree(OneTree) ##only a part is visible

## full tree :
# plotTree(OneTree, fullTree = TRUE, xlim = c(1,55), ylim = c(0, 11))
```

---

postProcessingVotes *Post-processing for Regression*

Description

Post-processing use OOB votes and predicted values to build more accurate estimates of Response values. Note that post-processing can not ensure that new estimate will have a lower error. It works for many cases but not all.

Usage

```
postProcessingVotes(object,
nbModels = 1,
idx = 1,
granularity = 1,
predObject = NULL,
swapPredictions = FALSE,
X = NULL,
imbalanced = FALSE)
```

Arguments

object	a randomUniformForest object with OOB data.
nbModels	how many models to build for new estimates. Usually one is enough.
idx	how many values to choose in OOB model for each new predicted value. Usually one is enough.
granularity	degree of precision needed for each old estimate value. Usually one is enough.
predObject	if current model is built with full sample, then using an old model 'predObject' (a randomUniformForest object) that have OOB data can help to reduce error. Must be used with 'swapPredictions = TRUE'
swapPredictions	set it to TRUE if two models, current one without OOB data and old one with OOB data, have to be used for trying to reduce prediction error.
X	not currently used.
imbalanced	not currently used.

Value

a vector of predicted values.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

Examples

```
# Note that post-processing works better with enough trees, at least 100, and enough data
n = 200;  p = 20
# Simulate 'p' gaussian vectors with random parameters between -10 and 10.
X <- simulationData(n,p)

# give names to features
X <- fillVariablesNames(X)

# Make a rule to create response vector
epsilon1 = runif(n,-1,1)
epsilon2 = runif(n,-1,1)
```

```
# a rule with many noise (only four variables are significant)
Y = 2*(X[,1]*X[,2] + X[,3]*X[,4]) + epsilon1*X[,5] + epsilon2*X[,6]

# randomize then  make train and test sample
twoSamples <- cut(sample(1:n,n), 2, labels = FALSE)

Xtrain = X[which(twoSamples == 1),]
Ytrain = Y[which(twoSamples == 1)]
Xtest = X[which(twoSamples == 2),]
Ytest = Y[which(twoSamples == 2)]

# compute an accurate model (in this case bagging and log data works best) and predict
rUF.model <- randomUniformForest(Xtrain, Ytrain, xtest = Xtest, ytest = Ytest,
bagging = TRUE, logX = TRUE, ntree = 60, threads = 2)

# get mean squared error
rUF.model

# post process
newEstimate <- postProcessingVotes(rUF.model)

# get mean squared error
sum((newEstimate - Ytest)^2)/length(Ytest)

## regression do not use all data but sub-samples.
## compare, when using full sample. But then, we do not have OOB data

# rUF.model.fullsample <- randomUniformForest(Xtrain, Ytrain, xtest = Xtest, ytest = Ytest,
# subsamplerate = 1, bagging = TRUE, logX = TRUE, ntree = 60, threads = 2)
# rUF.model.fullsample

## Nevertheless we can use old model with OOB data to fit a new estimate
## newEstimate.fullsample <- postProcessingVotes(rUF.model.fullsample,
# predObject = rUF.model, swapPredictions = TRUE)

## get mean squared error
# sum((newEstimate.fullsample - Ytest)^2)/length(Ytest)
```

`predict.randomUniformForest` *Predict method for random Uniform Forests objects*

Description

Prediction of test data with random Uniform Forests. Many options are allowed, the default one rendering exactly the same type of variable than the one of training labels.

Usage

```
## S3 method for class 'randomUniformForest'
```

```
predict(object, X,
type = c("response", "prob", "votes", "confInt",
"ranking", "quantile", "truemajority", "all"),
classcutoff = c(0,0),
conf = 0.95,
whichQuantile = NULL,
rankingIDs = NULL,
threads = "auto",
parallelpackage = "doParallel",
...)
```

Arguments

object	an object of class randomUniformForest, as one created by the randomUniformForest( ) function.
X	a data frame or matrix containing new data (without response values).
type	one of response, prob, votes, prediction intervals, ranking, quantile, true majority or raw outputs, indicating respectively the type of output: 'predicted values', 'matrix of class probabilities', 'matrix of vote counts', 'prediction interval for each response', 'ranked responses' in case of recommendation scheme (note that it is currently experimental and it, first, requires classification modelling), 'quantile regression', 'predicted values based on nodes votes instead of tree votes', or lastly, 'raw outputs' of a random uniform forest. Note that "confInt" and "quantile" use an estimate of the conditional expectation for each tree parameter (the randomness), which contains many duplicates, due to bootstrap and the nature of ensemble learning, to produce estimates. These ones are usually loose since the forest replicates the distribution of Y (knowing X) to take decisions while one does not need all the X values to have one prediction. A more accurate option is to use the dedicated bCI function. Note also that option "quantile" is better handled when using default value of the 'nodesize' option in the randomUniformForest function.
classcutoff	see randomUniformForest.
conf	if type == "confInt", value of 'conf' (greater than 0 and lesser than 1) is the desired level of confidence for any prediction interval.
whichQuantile	if type == "quantile", value of 'whichQuantile' is the desired quantile (greater than 0 and lesser than 1).
rankingIDs	experimental. If 'type = "ranking"', vector of ID if one wants to rank responses instead of classify them. Ranking usually involves many same users (or objects, or items) whose choices have to be ranked, e.g., by a recommendation engine. For example, if 'user1' is present four times in test sample, its ID gives to the model a way to identify him (or it) and to order response values, beginning by the most relevant. Then, for many users, the process is repeated giving, finally, predictions from most to least relevant, taking into account all users. random Uniform Forests approach currently uses class probabilities to order predictions values. Note that optimizing AUC is one of the methods leading to good ranking methods if choices are binary, and NDCG is a way to assess model.
threads	see randomUniformForest.

parallelpackage
: see randomUniformForest.

...
: not used currently.

Value

response
: predicted values. Default option that returns values in the same way than original training responses.

prob
: for classification only. Matrix of class probabilities.

votes
: matrix of vote counts. Each row is an observation and each column is tree output.

confInt
: for regression only. Matrix where each row is an observation and each column one of the prediction interval bounds.

quantile
: for regression only. Vector of predicted quantiles for 'conf' (value of the option) level of confidence.

ranking
: a matrix or data frame. Description will be updated soon.

truemajority
: predicted values, using raw outputs of trees (not majority vote). This option makes sense if one set 'nodesize' option greater than 1. Hence, aggregation is participative (at the leaf level) and not representative (at the tree level).

all
: raw outputs of the model. Not useful, unless further computation is needed, for example in case of Post-processing.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

See Also

postProcessingVotes, bCI, model.stats, generic.cv

Examples

```
## same as randomForest example
#  data(iris)
# set.seed(111)
# ind <- sample(2, nrow(iris), replace = TRUE, prob = c(0.8, 0.2))

# iris.ruf <- randomUniformForest(Species ~ ., data = iris[ind == 1,], OOB = FALSE,
# importance = FALSE, threads = 1)
# iris.pred <- predict(iris.ruf, iris[ind == 2,])

# table(observed = iris[ind == 2, "Species"], predicted = iris.pred)

## get all votes : note that aliases of classes are used internally and, for intermediate
## results, are not converted to their true values
# iris.all.votes <- predict(iris.ruf, iris[ind == 2,], type = "votes")

## get class probabilities
# iris.class.prob <- predict(iris.ruf, iris[ind == 2,], type = "prob")
# iris.class.prob
```

randomUniformForest *Random Uniform Forests for Classification, Regression and Unsupervised Learning*

Description

Ensemble model for classification, regression and unsupervised learning, based on a forest of unpruned and randomized binary decision trees. Unlike *Breiman's Random Forests*, each tree is grown by sampling, *with replacement*, a set of variables before splitting each node. Each cut-point is generated randomly, according to the *continuous Uniform distribution between two random points of each candidate variable or using its whole current support*. Optimal random node is, then, selected among many full random ones by maximizing Information Gain (classification) or minimizing a distance (regression), 'L2' (or 'L1'). Unlike *Extremely Randomized Trees*, data are either *bootstrapped or sub-sampled for each tree*. From the theoretical side, Random Uniform Forests are aimed to lower correlation between trees and to offer a deep analysis of variable importance. The unsupervised mode introduces clustering and dimension reduction, using a three-layer engine: dissimilarity matrix, Multidimensional Scaling (or Spectral decomposition) and k-means (or hierarchical clustering). From the practical side, Random Uniform Forests are designed to provide a complete analysis of (un)supervised problems and to allow native distributed and incremental learning.

Usage

```
## S3 method for class 'formula'
randomUniformForest(formula, data = NULL, subset = NULL, ...)
## Default S3 method:
randomUniformForest(X, Y = NULL, xtest = NULL, ytest = NULL,
ntree = 100,
mtry = ifelse(bagging, ncol(X), floor(4/3*ncol(X))),
nodesize = 1,
maxnodes = Inf,
depth = Inf,
depthcontrol = NULL,
regression = ifelse(is.factor(Y), FALSE, TRUE),
replace = ifelse(regression, FALSE, TRUE),
OOB = TRUE,
BreimanBounds = ifelse(OOB, TRUE, FALSE),
subsamplerate = ifelse(regression, 0.7, 1),
importance = TRUE,
bagging = FALSE,
unsupervised = FALSE,
unsupervisedMethod = c("uniform univariate sampling",
"uniform multivariate sampling", "with bootstrap"),
classwt = NULL,
oversampling = 0,
targetclass = -1,
outputperturbationsampling = FALSE,
rebalancedsampling = FALSE,
```

```
featureselectionrule = c("entropy", "gini", "random", "L2", "L1"),
randomcombination = 0,
randomfeature = FALSE,
categoricalvariablesidx = NULL,
na.action = c("fastImpute", "accurateImpute", "omit"),
logX = FALSE,
classcutoff = c(0,0),
subset = NULL,
usesubtrees = FALSE,
threads = "auto",
parallelpackage = "doParallel",
...)
## S3 method for class 'randomUniformForest'
print(x, ...)
## S3 method for class 'randomUniformForest'
summary(object, maxVar = 30, border = NA, ...)
## S3 method for class 'randomUniformForest'
plot(x, threads = "auto", ...)
```

Arguments

maxVar	maximum number of variables to plot and print when summarizing a randomUniformForest object.
border	positive integer value or NA. Change colour of the borders when plotting variable importance. By default, NA, which disables border.
x, object	an object of class randomUniformForest.
data	in case of formula, a data frame, or matrix, containing the variables (including response) and their values.
subset	an index vector indicating which rows should be used.
X, formula	a data frame, or matrix, of predictors, or a formula describing the model to be fitted. Note that, it is strongly recommended to avoid formula when using options or with large samples.
Y	a response vector. If it is a factor, classification is assumed, otherwise regression is computed.
xtest	a data frame or matrix (like X) containing predictors for the test set.
ytest	responses for the test set, if provided.
ntree	number of trees to grow. Default value is 100. Do not set it too small.
mtry	number of variables randomly sampled with replacement as candidates at each split. Default value is floor(4/3*ncol(X)) unless 'bagging' or 'randomfeature' options are specified. One can also set mtry = "random". For regression, increasing 'mtry' value usually leads to better accuracy. Note that mtry = 1 will lead to a *purely uniformly random forest*. 'mtry' has also an effect when assessing variable importance.
nodesize	minimal size of terminal nodes. Default value is 1 (for both classification and regression) and usually produce best results, as it reduces bias when trees are fully

grown. Variance is increased, but that is exactly what Random Uniform Forests need. Random 'nodesize' is allowed, setting option to "random". For each tree 'nodesize' will take a random value between 1 and 50, using increments of 5.

`maxnodes` maximal number of nodes for each tree. Default value is 'Inf', growing trees to maximum size. Random number of nodes is allowed, setting option to "random".

`depth` depth of each tree. By default, Trees are fully grown. Maximum depth, for a balanced tree, is log(n)/log(2). In regression it will usually be the case. Stumps are not allowed, hence smallest depth is 3. Note that 'depth' has an effect when assessing variable importance. Enabling, in conjunction, 'depthcontrol' activates a deeper competition between nodes in order to reduce the loss of accuracy induced by the tree's depth. Random depth is allowed, setting option to "random".

`depthcontrol` an integer, beginning at 1. Let algorithm controls the growth of each tree, letting the optimization criterion depend on the number of nodes as the tree is growing. More precisely, the option activates an internal measure against which algorithm is competing. 'depthcontrol' usually works well in conjunction with 'depth' option for large samples and regression, and is strongly recommended each time one wants to control both speed (lower values) and accuracy (higher values).

`regression` only needed if either classification or regression has to be set explicitly. Otherwise, model checks if 'Y' is a factor (classification), or not (regression) before computing task. If 'Y' is not a factor and one wants to do classification, must be set to FALSE.

`replace` if TRUE, sampling of cases is done with replacement. By default, TRUE for classification, FALSE for regression.

`OOB` if replace is TRUE, then if OOB is TRUE, "Out-of-bag" evaluation is done, resulting in an estimate of generalization (and mean squared) error and bounds. OOB option has an overhead on computing time especially for a large number of trees and large datasets, but it is one of the most useful option.

`BreimanBounds` if TRUE, computes all theoretical properties provided by Breiman (2001), since Random Uniform Forests inherit of Random Forests properties. For classification, it gives the two bounds of prediction error, average correlation between trees, strength and standard deviation of strength. In classification, Breiman's bound should act as an upper bound; if not then overfitting is likely to happen. A special case where the first Breiman's bound does not work is when classes are imbalanced. In such case, the second bound can be used since it overrides imbalanced classes and is the *upper bound of prediction error*. It could be loose but, if there are enough trees and data, will be strongly reduced using options. For regression, model returns an estimate of the forest theoretical prediction error, its upper bound, mean prediction error of a tree, average correlation between trees residuals and expected squared bias. The estimate of theoretical prediction error of the forest (and the upper bound) is not a bound of test (mean squared) error. Note that for multi-class problems or large files, 'BreimanBounds' option requires a lot of computing time.

`subsamplerate` value is the rate of sub-sampling (Bulhmann et Yu, 2002) for training sample. By default, 0.7 for regression (1, e.g. no sub-sampling, in classification). If 'replace' is TRUE, 'subsamplerate' can be set to values greater than 1. For regression, if

only accuracy is needed, setting 'replace' to FALSE and 'subsamplerate' to 1, may improve results in some cases but OOB evaluation (and Breiman's bounds) will be lost.

`importance` should importance of predictors be assessed? By default, TRUE. Note that it is strongly recommended to set it to FALSE for large datasets (then enable it for a subset of the dataset) since it is a intensive task.

`bagging` if TRUE, *Bagging* (Breiman, 1996) of random uniform decision trees is done. Useful to compare "Bagging of random uniform decision trees" and usual "Bagging of trees". For regression, can give sometimes better results than sampling, with replacement, variables at each node.

`unsupervised` unsupervised learning mode, following Breiman's ideas. Note that one has to call the second stage of unsupervised learning, see `unsupervised.randomUniformForest` to obtain a full object that will allow clustering.

`unsupervisedMethod`
method that has to be used to turn unsupervised problem into a supervised one. Note that 'unsupervisedMethod' uses either one (then bootstrap will not happen) or two arguments, the second one always being "with bootstrap" allowing, then, to use bootstrap.

`classwt` for classification only. Prior of the classes. Need not add up to one. Useful for imbalanced classes. Note that if one wants to compute many forests and combine them, with 'classwt' enabled for only few of them, all others forests must have 'classwt' enabled, leading to classwt = rep(1, 'number of classes') for forests that one do not need to compute weights.

`oversampling` for classification, a scalar between 1 and -1 for over or undersampling of minority or majority class, given by the value of 'targetclass'. For example, if set to 'oversampling = -0.3', and 'targetclass = 1', then the first class (assumed to be the majority class) will be undersampled by a proportion of 30 percent. if set to 0.5, and 'targetclass' set to 2, then second class (assumed to be the minority class) will be oversampled by a proportion of 50 percent. In all cases, size of original matrix will not be modified, hence 'oversampling' option is implicitly associated with undersampling. If one wants to force true oversampling or undersampling, 'rebalanced sampling' is the alternative. Note that for both classification and regression, 'oversampling' is also the value that can be used to perturb response values (see 'outputperturbationsampling'). Hence, in classification one should avoid to use 'outputperturbationsampling' and 'oversampling' together.

`targetclass` for classification only. Which class (given by its subscript, e.g. 1 for first class) should be targeted by 'oversampling' or 'outputperturbationsampling' option ?

`outputperturbationsampling`
if TRUE, let model applies a random perturbation over responses vector. For classification, 'targetclass' must be set to the class that will be perturbed. By default 5 percent of the values will be perturbed, but more is allowed (more than 100 percent, with bootstrap) using the 'oversampling' option. 'outputperturbationsampling' is aimed to reduce correlation between trees (residuals) and can be monitored using Breiman's bounds. It also works better with many trees and may help to protect from overfitting, since model does no longer use training labels (or values) but, in regression, a random Gaussian variable with similar (but

possibly different) mean than response but different variance. Note that each tree generates its own Gaussian variable whose parameters vary.

`rebalancedsampling`

for classification only. Can be set to TRUE or to a vector containing the desired sample size for each class. If TRUE, model builds samples where all classes are equally distributed, leading to exactly balanced classes, by either oversampling or undersampling. If vector, size of train data will change, according to the sum of the values in the vector. If the number of minority class cases asked in the vector is greater than the original one, sampling with replacement is done for these cases.

`featureselectionrule`

which optimization criterion should be chosen for growing trees ? By default, model uses "entropy" (in classification) to compute the Information Gain function. If set to "random", model chooses randomly between Gini criterion and entropy for each node of each tree. For regression, sum of squared residuals ("L2") or sum of absolute residuals ("L1") is allowed, or "random".

`randomcombination`

vector containing features index and, eventually, weight(s) for (random) combination of features. For example, if a combination of feature 1 and feature 2 is desired with a weight of 0.2 for the first, then randomcombination = c(1, 2, 0.2). If a combination of feature 3 and feature 4 is needed in the same time with a weight of 0.5, then randomcombination = c(1, 2, 3, 4, 0.2, 0.5). For full random combinations, one just put features in a vector, and then, in our example, randomcombination = c(1, 2, 3, 4). Useful sometimes to both reduce correlation between trees and increase their average individual strength.

`randomfeature` if TRUE, a forest of totally randomized trees (e.g. purely random forest) will be grown. In this case, there is no optimization. Useful as a base result for forest of randomized trees.

`categoricalvariablesidx`

which variables should be considered as categorical? By default, value is NULL, and then categorical variables remain in the same approach as continuous ones. If 'X' is a data frame, option can be set to "all", in which case the model will automatically identify categorical variables and will use a different scheme for growing trees using these variables (see the Details section). If 'X' is a matrix, one has to set a vector containing subscripts of categorical variables. For example, if feature 1 and feature 2 are categorical, then categoricalvariablesidx = c(1,2). Note that using formula will automatically build dummies and throw them to the model. Current engine for categorical features in random uniform forests does not use dummies and is also designed to work with variable that take discrete values. Note that assessing categorical variable increases the computation time.

`na.action` how to deal with NA data? By default, na.action = "fastImpute", using rough replacement with median, mean or most frequent values. If speed is not required, na.action = "accurateImpute", can lead to better results using model to impute NA values. na.action = "omit", just omit NA values and it's the only option when using formula. Note that many, accurate and fast, models (softImpute, missMDA, missForest, Amelia, ...) are available on CRAN to impute missing data. Note that "accurateImpute" calls the `fillNA2.randomUniformForest`

function, hence it is recommended to impute data outside of the model, especially for large datasets.

`logX` applies logarithm transformation to all predictors whose values are strictly positive, and ignores the others.

`classcutoff` for classification only. Change proportion of votes needed to get majority. First value of vector is the name of the class (between quotes) that has to be assessed. Second value is a kind of weight needed to get majority. For example, in a problem with classes "bad" and "good", and 'classcutoff = c("bad", 0.4)', it means that class "bad" needs 'Cte/0.4' times more votes than class "good" to be the majority class when predicting the label of a new observation, where Cte = 0.5 and sum of all votes equals to the number of trees. Note that the option can be monitored with the OOB evaluation.

`usesubtrees` allows to grow trees with an arbitrary depth (default one being half of the maximum depth), that are, in a second step, updated by extending all non pure nodes. The option is mainly designed to reduce memory footprint at the cost of an increasing computing time, for, at least, the same expected accuracy than in the standard case. Note that it seems also more robust when training real life and large datasets, especially when the average depth of trees is far from the theoretical one.

`threads` compute model in parallel for computers with many cores. Default value is 'auto', letting model run on all logical cores minus 1. User can set 'threads' to any value greater than 1. Note that, in Windows, logical cores consume same memory than physical ones, but will not speed up computation linearly with the number of logical cores. Note also that it is strongly recommended to use only one thread for small samples.

`parallelpackage`

which parallel back-end to use for computing parallel tasks ? By default and for ease of use, 'doParallel' is the package retained for now. Should not be modified. It has the great advantage to allow killing task, e.g. pushing the 'Stop' button without freezing R. Note that in this case and, at least, in Windows, one may need to manually terminate processes, using the 'Task manager' in order to avoid them occupying uselessly cores.

`...` not currently used.

Details

Random Uniform Forests have been designed to provide a complete analysis of supervised and unsupervised problems. From pre-processing to predictions, interpretation and summary of results. Hence, many functions and options have been built around the algorithm in order to let the user simply put the data as they come (matrix or data frame), getting basis objects that come with usual (and statistical) measures of assessment for all critical parts.

Random Uniform Forests are inspired by *Bagging* and Breiman's *Random Forests* (tm) but have many differences at theoretical and algorithmic levels. They build many randomized and unpruned binary decision trees and the four main differences with Random Forests are:
- sampling *with replacement* a set of features to generate each candidate node,
- subsampling data, in the case of regression,
- generating *random cut-points according to the Uniform distribution*, i.e. cut-points usually not

belong to data but are virtual points drawn between the minimum and the maximum, or between two random points, of each candidate variable at each node, using the continuous Uniform distribution, since all points are (or will always be converted to) numeric values.
- optimization criterion. Maximizing the Information Gain is preferably used for classification. For regression, sum of squared (or absolute) residuals is computed for each candidate node (region), using for each a random sampled pair of feature and cut-point. Then the metrics are summed for each pair of complementary nodes. The chosen pair is the one that reaches the minimum over all current candidates pairs, and define the optimal (and random) child node. More precisely, *in regression only sums are involved and only in the candidate nodes (not in the current one)*. Note that it could (while not clear) also be the case for Random Forests.

The enumeration above leads to a large and deep tree that is grown using global optimization, for the current partition, to select each node. Sampling features, with replacement, increases the competition between nodes, in order to limit variance, especially in the regression case where prediction error depends more on the model than in the classification case.

Others differences also appear at the node level. Like Random Forests, classification is done by majority vote, and regression by averaging trees outputs but:
- trees can be updated with streaming data (currently disabled for further tests),
- forests with different parameters and data can be combined to form one forest,
- trees are explicitly designed to have an average low bias, while trying to tame the happening of an increasing variance, and are thus optimized to reach a high level of randomness. The forest maintains the bias and reduces variance, since variance of the forest is approximatively (in regression) the product of average correlation between trees and average variance of trees. This leads to a similar scheme for the prediction error bounded by 'average correlation between trees residuals' x 'average variance of trees residuals'. Note that the trend in decreasing correlation can not be obtained in the same time than a decreasing variance. The main work is to *decrease correlation faster than the growth of variance*. Low correlation is mandatory to reach convergence and prevent overfitting, especially in regression where average correlation tends to be high (Ciss, 2015a, 2015c). One may experiment it since the model produces, by default, all Breiman's bounds and their details.

Others main features, thanks to Breiman's ideas, to the ensemble structure and to the Bayesian framework, are:
- some other paradigms of ensemble learning (like Bagging of random uniform decision trees or ensemble of totally randomized trees) using options,
- functions to manipulate and plot trees, see `getTree.randomUniformForest` and friends
- all Breiman's bounds,
- (internal) pre-processing in order to handle (almost) any matrix or data frame,
- post-processing votes in order to lower MSE by reducing bias, see `postProcessingVotes`,
- changing majority vote, using options ('classcutoff')
- *output perturbation sampling*, lowering more the correlation, replacing completely (for regression and for each tree) the training vector of responses by an independent random Gaussian one with similar mean but different variance,
- deep analysis of variable importance and selection, see `importance.randomUniformForest` and `partialImportance`,
- partial dependencies, opening the way to extrapolation, see `partialDependenceOverResponses` and `partialDependenceBetweenPredictors`,
- visualization tools and tables for almost any essential function,
- generic function to assess results, see `model.stats`,
- generic cross-validation function, see `generic.cv`,
- missing values imputation, see `rufImpute`,

- many methods for imbalanced classes, using options ('oversampling', 'rebalancedsampling', 'classwt', 'classcutoff', 'usesubtrees') or reSMOTE
- cost-sensitive learning, using options 'classwt' (which is dual) and friends
- native handling of categorical variables using a randomization mechanism at the node level. More precisely, the algorithm selects for each candidate node, randomly and before the splitting process, two values. The first one keeps its position along the variable while the second replaces, temporarily, all others values of the variable. This leads to a binary variable that can be treated like a numerical one. After the splitting, the variable recovers its original values. Since cut-points are almost virtual and random (a cut-point is not a point of the training sample), one has just to take care that the random splitting would not weaker the variable.
- quantile regression, see predict.randomUniformForest
- (new methods for) prediction and confidence intervals, see bCI,
- unsupervised learning, see unsupervised.randomUniformForest and friends:
- dimension reduction, using MDS or Spectral decomposition, see unsupervised.randomUniformForest,
- dynamic clustering, allowing to split/merge/modify on-the-fly cluster(s), see modifyClusters and friends,
- visualization, allowing to display, manipulate and assess clusters in an easy way, see unsupervised.randomUniformForest plot and print methods,
- variable importance for clusters, inherited from the supervised case,
- cluster analysis, closing unsupervised learning in a compact and granular view, see clusterAnalysis,
- native parallelism, thanks to the *parallel*, *doParallel* and *foreach* packages,
- internal MapReduce paradigm for large datasets that can fit in memory, see rUniformForest.big,
- incremental learning for large datasets that can not fit in memory, see rUniformForest.combine,
- distributed learning, allowing to run many different models on different data (sharing, at least, some features) on many computers and combine them in a single one, in different manners, for predictions. See rUniformForest.combine examples. Note that one has to carefully manage the *i.i.d.* assumption in order to see convergence happen.

In particular, incremental learning is native, since the model uses random cut-points, and one can remove, duplicate, add or modify/update trees at each step of the incremental process, see rm.trees. The model is not allowing results to be exactly reproducible using the set.seed() function. One reason is that many (including essential) options run at the tree (or node) level in order to decrease correlation and use many random seeds internally. Since *convergence is the primal property of Random Forests*, for the same enough large training sample, even if results will slightly vary, one has to consider the OOB estimate and Breiman's upper bound (in classification) as the main guarantees. They are effective only under the *i.i.d.* assumption. If enough data is available, one can derive OOB bounds (Ciss, 2015c), leading the test error to be bounded by the OOB error, the latter itself bounded by the Breiman's bounds.

Note that speed is currently not at the state-of-the-art for small datasets, due to the majority of R code and some constant overhead that seems coming from the parallelism. However some of the critical parts of the algorithm are written in C++, thanks to the Rcpp package. For large datasets the gap is greatly reduced, thanks to shortcuts added to the R code and increased randomness. That is the case when the dimension is getting high, or for regression. A great speed-up can also be achieved with 'depth', 'maxnodes', 'subsamplerate', 'mtry' (large values or with mtry = 1), 'randomfeature' (in combination with mtry), 'rebalancedsampling' (in classification) options or by combining (see rUniformForest.combine) many forests built upon chunks of data. All these tools will usually have a cost, loss in accuracy, depending on the dataset and the task.

Value

An object of class randomUniformForest, which is a list with the following components:

`forest`	list of tree objects, OOB objects (if OOB = TRUE), variable importance objects (if importance = TRUE).
`predictionObject`	if 'xtest' is not NULL, prediction objects.
`errorObject`	statistics about errors of the model.
`forestParams`	almost all parameters of the model.
`classes`	original labels of response vector in case of classification.
`logX`	TRUE, if logarithm transformation has been called.
`y`	training responses.
`variablesNames`	vector of variables names.
`call`	the original call to `randomUniformForest`.

Author(s)

Saip Ciss `<saip.ciss@wanadoo.fr>`

References

Amit, Y., Geman, D., 1997. Shape Quantization and Recognition with Randomized Trees. *Neural Computation* 9, 1545-1588.

Biau, G., Devroye, L., Lugosi, G., 2008. Consistency of random forests and other averaging classifiers. *The Journal of Machine Learning Research* 9, 2015-2033.

Breiman, L, 1996. Heuristics of instability and stabilization in model selection. *The Annals of Statistics* 24, no. 6, 2350-2383.

Breiman, L., 1996. Bagging predictors. *Machine learning* 24, 123-140.

Breiman, L., 2001. Random Forests, *Machine Learning* 45(1), 5-32.

Breiman, L., 2001. Statistical Modeling: The Two Cultures (with comments and a rejoinder by the author). *Statistical Science* 16, no. 3, 199-231.

Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C., 1984. *Classification and Regression Trees*. New York: Chapman and Hall.

Ciss, S., 2014. PhD thesis: *Forets uniformement aleatoires et detection des irregularites aux cotisations sociales*. Universite Paris Ouest Nanterre, France. In french.
English title : *Random Uniform Forests and irregularity detection in social security contributions*.
Link : https://www.dropbox.com/s/q7hbgeafrdd8qtc/Saip_Ciss_These.pdf?dl=0

Ciss, S., 2015a. *Random Uniform Forests*. Preprint. hal-01104340.

Ciss, S., 2015b. *Variable Importance in Random Uniform Forests*. Preprint. hal-01104751.

Ciss, S., 2015c. *Generalization Error and Out-of-bag Bounds in Random (Uniform) Forests*. Preprint. hal-01110524.

Dietterich, T.G., 2000. *Ensemble Methods in Machine Learning*, in : Multiple Classifier Systems, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 1-15.

Efron, B., 1979. Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics* 7, 1-26.

Ho, T.K., 1998. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, 832-844.

Vapnik, V.N., 1995. *The nature of statistical learning theory*. Springer-Verlag New York, Inc., New York, NY, USA.

See Also

predict.randomUniformForest, rUniformForest.big, rUniformForest.combine, rUniformForest.grow, importance.randomUniformForest, rm.trees, roc.curve, rufImpute, getTree.randomUniformForest, unsupervised.randomUniformForest, clusteringObservations

Examples

```
## not run
## NOTE : use option 'threads = 1' (disabling parallel processing) to speed up computing
## for small samples, since parallel processing is useful only for computationally
## intensive tasks

###### PART ONE : QUICK GUIDE

#### Classification

# data(iris)
# iris.ruf <- randomUniformForest(Species ~ ., data = iris, threads = 1)

## Regular companions (from 1 to 18):
## 1 -  model, parameters, statistics:
# iris.ruf ## or print(iris.ruf)

## 2 - OOB error:
# plot(iris.ruf, threads = 1)

## 3 - (global) variable importance, some statistics about trees:
# summary(iris.ruf)

#### Regression

## NOTE: when formula is used, missing values are automatically deleted and dummies
## are built for categorical features

# data(airquality)
# ozone.ruf <- randomUniformForest(Ozone ~ ., data = airquality, threads = 1)
# ozone.ruf

## plot OOB error:
# plot(ozone.ruf, threads = 1)

## 4 - Alternative modelling:
## 4.1 bagging like:
```

```
# ozone.bagging <- randomUniformForest(Ozone ~ ., data = airquality,
# bagging = TRUE, threads = 1)

## 4.2 Ensemble of totally randomized trees, e.g. purely random forest:
# ozone.prf <- randomUniformForest(Ozone ~ ., data = airquality,
# randomfeature = TRUE, threads = 1)

## 4.3 Extremely randomized trees like:
# ozone.ETlike <- randomUniformForest(Ozone ~ ., data = airquality,
# subsamplerate = 1, replace = FALSE, bagging = TRUE, mtry = floor((ncol(airquality)-1)/3),
# threads = 1)

## Common case: use X, as a matrix or data frame, and Y, as a response vector,

#### Classification : iris data, training and testing

# data(iris)

## define random training and test sample :
## "Species" is the response vector

# set.seed(2015)
# iris.train_test <- init_values(iris[,-which(colnames(iris) == "Species")], iris$Species,
# sample.size = 1/2)

## training and test samples:
# iris.train = iris.train_test$xtrain
# species.train = iris.train_test$ytrain
# iris.test = iris.train_test$xtest
# species.test = iris.train_test$ytest

## 5 - training and test (or validation) modelling:
# iris.train_test.ruf <- randomUniformForest(iris.train, species.train,
# xtest = iris.test, ytest = species.test, threads = 1)

## 6 - all-in-one results:
# iris.train_test.ruf

## 7 - Alternative modelling: imbalanced classes
## balanced sampling (for example): equal sample size for all labels

# iris.train_test.balancedsampling.ruf <- randomUniformForest(iris.train, species.train,
# xtest = iris.test, ytest = species.test, rebalancedsampling = TRUE, threads = 1)

###### PART TWO : SUMMARIZED CASE STUDY

#### Classification : Wine Quality data
## http://archive.ics.uci.edu/ml/datasets/Wine+Quality
## We use 'red wine quality' file : data have 1599 observations, 12 variables and 6 classes.

# data(wineQualityRed)
# wineQualityRed.data = wineQualityRed
```

```
## class and observations
# Y = wineQualityRed.data[, "quality"]
# X = wineQualityRed.data[, -which(colnames(wineQualityRed.data) == "quality")]

## First look : train model with default parameters (and retrieve estimates)
# wineQualityRed.std.ruf <- randomUniformForest(X, as.factor(Y))
# wineQualityRed.std.ruf

## (global) Variable Importance:
# summary(wineQualityRed.std.ruf)

## But some labels do not have enough observations to assess variable importance
## merging class 3 and 4. Merging class 7 and 8 to get enough labels.
# Y[Y == 3] = 4
# Y[Y == 8] = 7

## make Y as a factor, change names and get a summary
# Y = as.factor(Y)
# levels(Y) = c("3 or 4", "5", "6", "7 or 8")
# table(Y)

## learn a new model to get a better view on variable importance
## NOTE: Y is now a factor, the model will catch the learning task as a classification one
# wineQualityRed.new.ruf <- randomUniformForest(X, Y)
# wineQualityRed.new.ruf

## global variable importance is more consistent
# summary(wineQualityRed.new.ruf)

## plot OOB error (needs some computing)
# plot(wineQualityRed.new.ruf)

## 8 - alternative Modelling: use subtrees (small trees, extended then reassembled)
## may change something, depending on data
# wineQualityRed.new.ruf <- randomUniformForest(X, Y, usesubtrees = TRUE)

## 9 - deep variable importance:
## 9.1 - interactions are granular: use more for consistency, or less to see primary information
## 9.2 - a table is printed with details
# importance.wineQualityRed <- importance(wineQualityRed.new.ruf, Xtest = X, maxInteractions = 6)

## 10 - visualization:
## 10.1 - global importance, interactions, importance based on interactions,
## importance based on labels, partial dependencies for all influential variables
## (loop over the prompt to get others partial dependencies)
## 10.2 - get more points, using option whichOrder = "all", default option.

# plot(importance.wineQualityRed, Xtest = X, whichOrder = "first")

## 11 - Cluster analysis: (if quick answers are needed)
## Note: called 'cluster' since it was first designed for clustering
## 11.1 - choose the granularity : components, maximum features, (as) categorical ones
## 11.2 - get a compact view
```

```
## 11.3 - see how importance is explaining the data
# analysis.wineQualityRed = clusterAnalysis(importance.wineQualityRed, X, components = 3,
# maxFeatures = 3, clusteredObject = wineQualityRed.new.ruf, categorical = NULL, OOB = TRUE)

## 11.4 - interpretation:
## Numerical features average: a good wine has much less volatile acidity,
## much more citric acid, ... than a wine of low quality.
## Most influential features:  while volatile.acidity seems to be important,...
## (Component frequencies:) ..., all variables must be taken into account, since information
## provided by the most important ones does not enough cover the whole available information.

## 11.5 - Complementarity:
## One does not forget to look plot of importance function. clusterAnalysis( )
## is a summarized view of the former and should not have contradictory terms
## but, eventually, complementary ones.

## 12 - Partial importance: (local) variable importance per class
## which features for a very good wine (class 7 or 8) ?
## Note: in classification, partial importance is almost the same than "variable importance over
## labels", being more local but they have different interpretations. The former is exclusive.
# pImportance.wineQualityRed.class7or8 <- partialImportance(X, importance.wineQualityRed,
# whichClass = "7 or 8", nLocalFeatures = 6)

## 13 - Partial dependencies: how response relies to each variable or a pair of ones?
## 13.1 - admit options.
## get it feature after feature, recalling partial dependence and considering feature
## at the first order assuming the feature is the most important,
## at least for the class one need to assess.

# pDependence.wineQualityRed.totalSulfurDioxide <- partialDependenceOverResponses(X,
# importance.wineQualityRed, whichFeature = "total.sulfur.dioxide",
# whichOrder = "first", outliersFilter = TRUE)

## 13.2 - Look for the second order (assuming the feature is the second most important,
## at least for the class one need to assess).
# pDependence.wineQualityRed.totalSulfurDioxide <- partialDependenceOverResponses(X,
# importance.wineQualityRed, whichFeature = "total.sulfur.dioxide",
# whichOrder = "second", outliersFilter = TRUE)

## 13.3 - Look at all orders: no assumptions, simply look the average effect
# pDependence.wineQualityRed.totalSulfurDioxide <- partialDependenceOverResponses(X,
# importance.wineQualityRed, whichFeature = "total.sulfur.dioxide",
# whichOrder = "all", outliersFilter = TRUE)

## see what happens then for "alcohol" (more points using option 'whichOrder = "all"')
# pDependence.wineQualityRed.alcohol <- partialDependenceOverResponses(X,
# importance.wineQualityRed, whichFeature = "alcohol",
# whichOrder = "first", outliersFilter = TRUE)

## 13.4 - Translate interactions into dependence : pair of features
## is interaction leading to the same class (underlying structure)?
## is dependence linear ?
## for which values of the pair is the dependence the most effective ?
```

```
# pDependence.wineQualityRed.sulfatesAndVolatileAcidity <- partialDependenceBetweenPredictors(X,
# importance.wineQualityRed, c("sulphates", "volatile.acidity"),
# whichOrder = "all", outliersFilter = TRUE)

#### Regression : Auto MPG
## http://archive.ics.uci.edu/ml/datasets/Auto+MPG
## 398 observations, 8 variables,
## Variable to predict : "mpg", miles per gallon

# data(autoMPG)
# autoMPG.data = autoMPG

# Y = autoMPG.data[,"mpg"]
# X = autoMPG.data[,-which(colnames(autoMPG.data) == "mpg")]

## remove "car name" which is a variable with unique ID (car models)
# X = X[, -which(colnames(X) == "car name")]

## train the default model and get OOB evaluation
# autoMPG.ruf <- randomUniformForest(X, Y)

## assess variable importance (ask more points with 'maxInteractions' option)
## NOTE: importance strongly depends on 'ntree' and 'mtry' parameters
# importance.autoMPG <- importance(autoMPG.ruf, Xtest = X)

## 14 - Dependence on most important predictors: marginal distribution of the response
## over each variable
# plot(importance.autoMPG, Xtest = X)

## 15 - Extrapolation:
## recalling partial dependencies and getting points
## NOTE : points are the result of the forest classifier and not of the training responses
# pDependence.autoMPG.weight <- partialDependenceOverResponses(X, importance.autoMPG,
# whichFeature = "weight", whichOrder = "all", outliersFilter = TRUE)

## 16 - Visualization again: view as discrete values
# visualize again 'model year' as a discrete variable and not as a continuous one
# pDependence.autoMPG.modelyear <- partialDependenceOverResponses(X, importance.autoMPG,
# whichFeature = "model year", whichOrder = "all", maxClasses = 30)

## 16 - Partial importance for regression: see important variables only for a part
## of response values
## what are the features that lead to a lower consumption (and high mpg)?
# pImportance.autoMPG.high <- partialImportance(X, importance.autoMPG,
# threshold = mean(Y), thresholdDirection = "high", nLocalFeatures = 6)

## 17 - Partial dependencies between covariates:
## look at "weight" and "acceleration" dependence
# pDependence.autoMPG.weightAndAcceleration <-
# partialDependenceBetweenPredictors(X, importance.autoMPG, c("weight", "acceleration"),
# whichOrder = "all", perspective = FALSE, outliersFilter = TRUE)
```

```
## 18 - More visualization: 3D (looking to the prompt to start animation)
## Note: requires some computation
# pDependence.autoMPG.weightAndAcceleration <-
# partialDependenceBetweenPredictors(X, importance.autoMPG, c("weight", "acceleration"),
# whichOrder = "all", perspective = TRUE, outliersFilter = FALSE)

##dtFW
```

---

reSMOTE *REplication of a Synthetic Minority Oversampling TEchnique for highly imbalanced datasets*

Description

Produce new samples of the minority class by creating synthetic data of the original ones by randomization. After that, the new dataset contains all original data + synthetic data labelled with the minority class. Main argument is derived from Breiman's ideas on the construction of synthetic data for the unsupervised mode of random (Uniform) Forests. The new dataset preserves the distribution of the original dataset covariates(at least with default 'samplingMethod' option).

Usage

```
reSMOTE(X, Y,
majorityClass = 0,
increaseMinorityClassTo = NULL,
conditionalTo = NULL,
samplingFromGaussian = FALSE,
samplingMethod = c("uniform univariate sampling",
"uniform multivariate sampling", "with bootstrap"),
seed = 2014)
```

Arguments

X	a matrix (training sample) of numeric values.
Y	labels (vector or factor) of the training sample.
majorityClass	label(s) of the majority class(es).
increaseMinorityClassTo	proportion of minority class observations that one wants to reach.
conditionalTo	the targeted variable (usually the response) that one needs the new (artificial) observations to be sampled. More precisely, synthetic dataset will be generated conditionally to each target value.
samplingFromGaussian	not used for now.
samplingMethod	which method to use for sampling from data? Note that one may use any of the two first methods with the 'with bootstrap' option, for example 'samplingMethod = c("uniform univariate sampling", "with bootstrap")' or 'samplingMethod = "uniform univariate sampling"'.
seed	seed to use when sampling distribution.

Details

The main purpose of the reSMOTE( ) function is to increase AUC. The function is specifically designed for random Uniform Forests with possible conjunction of options 'classwt', 'rebalanced-sampling' and/or 'oversampling', also designed to treat imbalanced classes. Since ensemble models tend to choose majority class too quickly, reSMOTE introduces a perturbation that forces the algorithm to learn data more slowly. Note that the wanted proportion of minority class ('increaseMinorityClassTo' option) plays a key role and is probably model and data dependent. Note also that reSMOTE() function is its in early phase and is mainly designed for very imbalanced datasets and when others options find their limits.

Value

A matrix or a data frame of the new training sample embedding original training sample and synthetic training sample. Labels are put in the last column, with the column name noted "Class" (if Y is a factor) or "Response" (if not).

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

Examples

```
## not run
## A - generate a skewed dataset to learn
# set.seed(2014)
# n = 1000
# p = 100
# X = simulationData(n, p)
# X = fillVariablesNames(X)
# epsilon1 = runif(n,-1,1)
# epsilon2 = runif(n,-1,1)
# rule = 2*(X[,1]*X[,2] + X[,3]*X[,4]) + epsilon1*X[,5] + epsilon2*X[,6]
# Y = as.factor(ifelse(rule > (3*(mean(rule) - 4/3*epsilon1*X[,7])), "-","+"))

## distribution of labels
# distributionOfY = table(Y)

## ratio of majority/minority sample
# max(distributionOfY)/min(distributionOfY)

## generate train and test sets, experiencing concept drift

# set.seed(2014)
# train_test = init_values(X, Y, sample.size = 5/10)
# X1 = train_test$xtrain
# Y1 = train_test$ytrain
# X2 = train_test$xtest
# Y2 = train_test$ytest

# table(Y1)
# table(Y2)
```

```
## learn and predict with base model
# baseModel.ruf = randomUniformForest(X1, Y1, xtest = X2, ytest = Y2)
# baseModel.ruf

## reSMOTE train sample : we choose 'increaseMinorityClassTo = 0.3'.
# X1Y1.resmoted = reSMOTE(X1, Y1, majorityClass = "-",
# conditionalTo = Y1, increaseMinorityClassTo = 0.3)

## learn and predict with reSMOTEd model
# reSMOTEdModel.ruf = randomUniformForest(X1Y1.resmoted[,1:p], X1Y1.resmoted[,"Class"],
# xtest = X2, ytest = Y2)

## compare both models, looking the AUC (and AUPR, if misclassified positives cases
## have a much higher cost than misclassified negative ones)

# baseModel.ruf
# reSMOTEdModel.ruf

## B - German credit dataset
# URL = "http://archive.ics.uci.edu/ml/machine-learning-databases/statlog/german/"
# dataset = "german.data-numeric"
# germanCreditData = read.table(paste(URL, dataset, sep =""))

# Y = germanCreditData[,25]
# X = germanCreditData[,-25]

# set.seed(2014)
# train_test = init_values(X, Y, sample.size = 5/10)
# X1 = train_test$xtrain
# Y1 = train_test$ytrain
# X2 = train_test$xtest
# Y2 = train_test$ytest

# 1 - default modelling: AUC around 0.67; AUPR around 0.56

# germanCredit.baseModel.ruf = randomUniformForest(X1, as.factor(Y1),
# xtest = X2, ytest = as.factor(Y2), ntree = 200, importance = FALSE)
# germanCredit.baseModel.ruf

## 2 - reSMOTEd modelling: AUC around 0.70; AUPR around 0.60;
## Note : reSMOTE is not consistent with OOB evaluation.

# p = ncol(X1)
# X1Y1.resmoted = reSMOTE(as.true.matrix(X1), as.factor(Y1), majorityClass = "1",
# conditionalTo = Y1, increaseMinorityClassTo = 0.6)
# X1.new = X1Y1.resmoted[,1:p]
# Y1.new = X1Y1.resmoted[,"Class"]
# germanCredit.reSMOTEd.ruf2 = randomUniformForest(X1.new, as.factor(Y1.new),
# xtest = X2, ytest = as.factor(Y2), ntree = 200, importance = FALSE)

## 2.1 - try reSMOTE + Random Forests: AUC around 0.70; AUPR around 0.58;
```

```
# library(randomForest)
# germanCredit.reSMOTEd.rf = randomForest(X1.new, as.factor(Y1.new),
# xtest = X2, ytest = as.factor(Y2))
# statsRF = model.stats(as.factor(as.numeric(germanCredit.SMOTEd.rf$test$pred)), as.factor(Y2))

## 3 - use benchmark: SMOTE, implemented in 'unbalanced' R package

# require(unbalanced)
## SMOTE : generate 'perc.over/100' new instance for each rare instance.
## 'perc.under/100' is the number of instances of majority class for each smoted observation.
## Generate smoted observations to have a similar distribution (but more examples)
## of the labels than the one generated by 'reSMOTE()'.

# XX = ubBalance(X1, as.factor(Y1-1), type = "ubSMOTE", perc.over = 400, perc.under = 100)

## 3.1 apply to randomUniformForest: AUC around 0.68; AUPR around 0.55;

# germanCredit.SMOTEd.ruf = randomUniformForest(XX$X, as.factor(XX$Y),
# xtest = X2, ytest = as.factor(Y2-1), ntree = 200, importance = FALSE, BreimanBounds = FALSE)

## 3.2 apply to randomForest: AUC around 0.68; AUPR around 0.57;

# germanCredit.SMOTEd.rf = randomForest(XX$X, as.factor(XX$Y),
# xtest = X2, ytest = as.factor(Y2-1))
# statsModel = model.stats(as.factor(as.numeric(germanCredit.SMOTEd.rf$test$pred)),
# as.factor(Y2))

## 4 - Stabilize : AUC around 0.71; AUPR around 0.62;
## a - generate a large number of minority class examples
## b - use rebalanced sampling in the model : equal size for each class
## so that size of the new training sample equals size of the original training sample
## c - increase minimum number of observations in terminal nodes (nodesize)
## d - increase number of trees.

# X1Y1.resmoted = reSMOTE(as.true.matrix(X1), as.factor(Y1), majorityClass = "1",
# conditionalTo = Y1, increaseMinorityClassTo = 1)
# X1.new = X1Y1.resmoted[,1:p]
# Y1.new = X1Y1.resmoted[,"Class"]

# germanCredit.reSMOTEd.ruf2 = randomUniformForest(X1.new, as.factor(Y1.new),
# xtest = X2, ytest = as.factor(Y2), rebalancedsampling = c(250,250), nodesize = 10,
# ntree = 1000, importance = FALSE, OOB = FALSE)

# 4.1 - Apply to Random Forests : AUC around 0.72; AUPR around 0.62;
# germanCredit.reSMOTEd.rf = randomForest(X1.new, as.factor(Y1.new),
# xtest = X2, ytest = as.factor(Y2), sampsize = c(250,250), nodesize = 10, ntree = 1000)
# statsModel = model.stats(as.factor(as.numeric(germanCredit.reSMOTEd.rf$test$pred)),
# as.factor(Y2))
```

`rm.trees` *Remove trees from a random Uniform Forest*

Description

Remove any number of trees from a random Uniform Forest, especially in case of incremental learning.

Usage

```
rm.trees(rufObject, X = NULL, Y = NULL,
method = c("default", "random", "oldest", "newest", "optimal", "quantile"),
howMany = NULL,
rm.sample = 0.1)
```

Arguments

rufObject	a object of class randomUniformForest.
X	a validation sample, in case of removing 'method' is "optimal".
Y	a vector of responses for validation sample.
method	method to used for removing trees. Note that "optimal" and "quantile" are not fully tested and can lead to unexpected results. "default" work only if OOB evaluation is not missing. Use "random" if only one forest has been set. If ensemble of forests (e.g. incremental learning), use "oldest" or "newest" (but a validation sample may be necessary).
howMany	how many trees to remove ?
rm.sample	if method = "random", 'rm.sample' is the proportion of trees to remove.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

Examples

```
## Classification : "breast cancer" data
## (http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic))

## data(breastCancer)
# breastCancer.data <- breastCancer

## remove ID (first column) and divide data in train and test set
# breastCancer.data = breastCancer.data[,-1]

# n <- nrow(breastCancer.data)
# p <- ncol(breastCancer.data)

# trainTestIdx <- cut(sample(1:n, n), 2, labels= FALSE)

## train examples
# breastCancer.data.train <- breastCancer.data[trainTestIdx == 1, -p]
# breastCancer.class.train <- as.factor(breastCancer.data[trainTestIdx == 1, p])

## rename class in benign (class 2) and malignant (class 4) to have a better view
```

```
# levels(breastCancer.class.train) = c("benign", "malignant")

## test data
# breastCancer.data.test <- breastCancer.data[trainTestIdx == 2, -p]
# breastCancer.class.test <- as.factor(breastCancer.data[trainTestIdx == 2, p])
# levels(breastCancer.class.test) = c("benign", "malignant")

## compute many trees in a fast way, removing OOB evaluation
## breastCancer.ruf <- randomUniformForest(breastCancer.data.train, breastCancer.class.train,
## classwt = c(1, 3.5), maxnodes = 10, OOB = FALSE, ntree = 500, threads = 2)

## predict and see results
## pred.breastCancer.ruf <- predict(breastCancer.ruf, breastCancer.data.test)
# confusion.matrix(as.numeric(pred.breastCancer.ruf), as.numeric(breastCancer.class.test))

## remove some trees at random. Default method can not be used since we don't have OOB data
# breastCancer.ruf <- rm.trees(breastCancer.ruf, method = "random", rm.sample = 0.2)

# breastCancer.ruf # old number of trees is still printed, but trees have been removed

## predict and see results again
# pred.breastCancer.ruf <- predict(breastCancer.ruf, breastCancer.data.test)
# confusion.matrix(as.numeric(pred.breastCancer.ruf), as.numeric(breastCancer.class.test))
```

`roc.curve` *ROC and precision-recall curves for random Uniform Forests*

Description

plot ROC and precision-recall curves for objects of class randomUniformForest and compute F-beta score. It also works for any other model that provides predicted labels (but only for ROC curve).

Usage

```
roc.curve(X, Y, classes,
positive = classes[2],
ranking.threshold = 0,
ranking.values = 0,
falseDiscoveryRate = FALSE,
plotting = TRUE,
printValues = TRUE,
Beta = 1)
```

Arguments

`X`	a vector (or a factor) of predictions (with two classes) or an object of class randomUniformForest (with OOB option enabled).
`Y`	a vector of numeric (integer) responses, or a factor, with two classes.

classes	a vector (or a factor) of values that designate the class.
positive	convention for the positive class (e.g. the minority class).
ranking.threshold	option currently implemented but not fully tested.
ranking.values	option currently implemented but not fully tested.
falseDiscoveryRate	if TRUE, precision-recall curve is plotted. if FALSE, default value, ROC curve is plotted.
plotting	plotting the ROC curve ?
printValues	display values to screen ?
Beta	'beta' value for F-beta score.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

See Also

importance.randomUniformForest

Examples

```
## Classification : "breast cancer" data
# http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)

data(breastCancer)
breastCancer.data <- breastCancer

# remove ID (first column) and divide data in train and test set
breastCancer.data = breastCancer.data[,-1]

n <- nrow(breastCancer.data)
p <- ncol(breastCancer.data)

trainTestIdx <- cut(sample(1:n, n), 2, labels= FALSE)

# train examples
breastCancer.data.train <- breastCancer.data[trainTestIdx == 1, -p]
breastCancer.class.train <- as.factor(breastCancer.data[trainTestIdx == 1, p])

# rename class in benign (class 2) and malignant (class 4) to have a better view
levels(breastCancer.class.train) = c("benign", "malignant")

# test data
breastCancer.data.test <- breastCancer.data[trainTestIdx == 2, -p]
breastCancer.class.test <- as.factor(breastCancer.data[trainTestIdx == 2, p])

levels(breastCancer.class.test) = c("benign", "malignant")

# compute model : train then test in the same function and assign class weights
```

```
# to better match the distribution (OOB errors and Breiman's bounds can help to choose weights)
# Note that in this case 'recall' (or sensitivity) is the objective,
# e.g. match all possible cases of malignant tumours even if false positive rate increase
#(in this latter case, further steps will reveal the truth). If malignant tumour is not detected,
# then diagnosis error is, by far, more critical.
breastCancer.ruf  <- randomUniformForest(breastCancer.data.train, breastCancer.class.train,
xtest = breastCancer.data.test, ytest = breastCancer.class.test,
classwt = c(1, 3.5), threads = 2, ntree = 40, BreimanBounds = FALSE)

# get a summary of model
breastCancer.ruf

## plot ROC Curve for test data
# roc.curve(breastCancer.ruf, breastCancer.class.test, levels(breastCancer.class.test))

## plot precision-recall curve for test data
# roc.curve(breastCancer.ruf, breastCancer.class.test, levels(breastCancer.class.test),
# falseDiscoveryRate = TRUE)

## associate cut-off and purely random forest as an alternative to find maximum malignant cases
## with a low false positive rate.
## 'classcutoff' option is a bit tricky. Let's take the example we will use below.
## classcutoff = c("benign", 1.25) means that number of votes for class "benign"
## will be weighted by 0.4 (= Cte/1.25, where Cte = 0.5) for each response.
## Hence "benign" will never have majority unless it has 2.5 (1.25/0.5) times more votes
## than "malignant" class and all votes sum to total number of trees.
# breastCancer.cutOff.ruf <- randomUniformForest(breastCancer.data.train, breastCancer.class.train,
# xtest = breastCancer.data.test, ytest = breastCancer.class.test, classcutoff = c("benign", 1.25),
# randomfeature = TRUE, ntree = 50, threads = 2, BreimanBounds = FALSE)

# roc.curve(breastCancer.cutOff.ruf, breastCancer.class.test, levels(breastCancer.class.test))
# roc.curve(breastCancer.cutOff.ruf, breastCancer.class.test, levels(breastCancer.class.test),
# falseDiscoveryRate = TRUE)

## evaluate OOB data, when there is no test set
# breastCancer.ruf <- randomUniformForest(breastCancer.data.train, breastCancer.class.train,
# classwt = c(1, 3.5), threads = 2)
# roc.curve(breastCancer.ruf, breastCancer.class.train, levels(breastCancer.class.train))
```

`rUniformForest.big` *Random Uniform Forests for Classification and Regression with large data sets*

Description

Implements random uniform forests for data sets that are too large too fit in physical memory but enough too fit in virtual memory. data set is randomly (or not) cut in many sub-samples and each one is processed, getting many base (but ensemble) models per sub-sample. At the end, all base models are combined to obtain one ensemble of ensembles model. If data can not reside in physical memory, but can reside in virtual memory (physical memory + swap file) then consider

R packages 'bigmemory', 'data.table' of 'ff' to load data. To save memory (and computing time), subsamples of data (e.g. using an Hadoop environment like) are suited, computing then one forest per sub-sample and combining all trees from all forests using rUniformForest.combine. Note that rUniformForest.big( ) is first designed to compute large files on a small computer, at the expense of accuracy. But, in case of a shifting distribution, model may be more robust than the standard one (at least for regression).

Usage

```
rUniformForest.big(X, Y = NULL,
xtest = NULL,
ytest = NULL,
nforest = 2,
randomCut = FALSE,
reduceDimension = FALSE,
reduceAll = FALSE,
replacement = FALSE,
subsample = FALSE,
ntree = 100,
    nodesize = 1,
maxnodes = Inf,
mtry = ifelse(bagging,ncol(X),floor(4/3*ncol(X))),
regression = ifelse(is.factor(Y),FALSE, TRUE),
    subsamplerate = ifelse(regression, 0.7, 1),
replace = ifelse(regression,FALSE,TRUE),
OOB = TRUE,
BreimanBounds = ifelse(OOB, TRUE, FALSE),
    depth = Inf,
    depthcontrol = NULL,
    importance = TRUE,
bagging = FALSE,
unsupervised = FALSE,
unsupervisedMethod = c("uniform univariate sampling",
"uniform multivariate sampling", "with bootstrap"),
classwt = NULL,
oversampling = 0,
targetclass = -1,
outputperturbationsampling = FALSE,
    rebalancedsampling = FALSE,
featureselectionrule = c("entropy", "gini", "random", "L1", "L2"),
randomcombination = 0,
randomfeature = FALSE,
categoricalvariablesidx = NULL,
na.action = c("fastImpute", "accurateImpute", "omit"),
logX = FALSE,
classcutoff = c(0,0),
subset = NULL,
usesubtrees = FALSE,
threads = "auto",
```

```
parallelpackage = "doParallel")
```

Arguments

X	a large data frame, or matrix, of predictors describing the model to be fitted.
Y	a responses vector. If it is a factor, classification is assumed, otherwise regression is computed.
xtest	an optional data frame or matrix (like X) containing predictors for the test set.
ytest	optional responses for the test set.
nforest	number of forests to compute. Size of each subsample will be 'number of observations in the original data set / nforest'. if 'nforest' is too high, accuracy will be lost. If too low, computing time will be long.
randomCut	should original data be cut randomly ?
reduceDimension	should dimension be reduced in original data (or subsamples, if nforest > 1) ? useful for speed, but can reduce dramatically accuracy.
reduceAll	should subsamples have lower dimension ? It is recommended to use it only to quickly get a base result.
replacement	should sample of data be done with replacement, e.g. sample n observations between n with replacement then divide them in 'nforest' subsamples.
subsample	value of subsample rate (m/n), e.g. sample m observations (m < n) between n, then divide them in 'nforest' subsamples. Note than 'subsample' can be combine with 'replacement' option.
ntree	see randomUniformForest.
mtry	see randomUniformForest.
nodesize	see randomUniformForest.
maxnodes	see randomUniformForest.
depth	see randomUniformForest.
depthcontrol	see randomUniformForest.
regression	see randomUniformForest.
replace	see randomUniformForest.
OOB	see randomUniformForest.
BreimanBounds	see randomUniformForest.
subsamplerate	see randomUniformForest.
importance	see randomUniformForest.
bagging	see randomUniformForest.
unsupervised	see randomUniformForest.
unsupervisedMethod	see randomUniformForest.
classwt	see randomUniformForest.
oversampling	see randomUniformForest.

targetclass see randomUniformForest.

outputperturbationsampling
see randomUniformForest.

rebalancedsampling
see randomUniformForest.

featureselectionrule
see randomUniformForest.

randomcombination
see randomUniformForest.

randomfeature see randomUniformForest.

categoricalvariablesidx
see randomUniformForest.

na.action see randomUniformForest.

logX see randomUniformForest.

classcutoff see randomUniformForest.

subset see randomUniformForest.

usesubtrees see randomUniformForest.

threads see randomUniformForest.

parallelpackage
see randomUniformForest.

Value

an object of class randomUniformForest.

Author(s)

Saip CISS <saip.ciss@wanadoo.fr>

See Also

rUniformForest.combine, rUniformForest.grow

Examples

```
## not run
## Classification: synthetic data
# n = 100;  p = 10 ## for ease of use, we consider small 'n'
## Simulate 'p' gaussian vectors with random parameters between -10 and 10.
#X <- simulationData(n,p)

## Make a rule to create response vector
# epsilon1 = runif(n,-1,1)
# epsilon2 = runif(n,-1,1)
# rule = 2*(X[,1]*X[,2] + X[,3]*X[,4]) + epsilon1*X[,5] + epsilon2*X[,6]

# Y <- as.factor(ifelse(rule > mean(rule), 1, 0))
```

```
# big.ruf <- timer(rUniformForest.big(X, Y, nforest = 2,
# threads = 1, BreimanBounds = FALSE, replacement = TRUE, importance = FALSE))

## elapsing time
# big.ruf$time

## OOB accuracy
# big.ruf$object

## standard model
# std.ruf <- timer(randomUniformForest(X, Y, threads = 1, ntree = 20, BreimanBounds = FALSE))

## elapsing time. Note that for small 'n' standard case will be faster.
# std.ruf$time

## OOB accuracy
#std.ruf$object

## not run
##  regression
# Y = rule
# big.ruf <- timer(rUniformForest.big(X, Y, nforest = 2,
# threads = 2, BreimanBounds = FALSE, subsample = 0.7))
# big.ruf

## classic random uniform forest
# std.ruf <- timer(randomUniformForest(X, Y, threads = 2, BreimanBounds = FALSE))
# std.ruf  # accuracy gap is much larger in case of regression

## but, one can consider a new case, e.g. shifting distribution, to see how it works
# newX <- simulationData(n,p)
# epsilon1 = runif(n,-1,1)
# epsilon2 = runif(n,-1,1)
# newRule = 2*(X[,1]*X[,2] + X[,3]*X[,4]) + epsilon1*X[,5] + epsilon2*X[,6]
# newY = newRule

## predict using standard model
# pred.std.ruf <- predict(std.ruf$object, newX)

## get mean squared error
# sum( (pred.std.ruf - newY)^2 )/length(newY)

## predict using rUniformForest.big
# pred.big.ruf <- predict(big.ruf$object, newX)

## get mean squared error : both errors will be more closer, and for large 'n' (and more trees),
## rUniformForest.big might have lower error
# sum( (pred.big.ruf - newY)^2 )/length(newY)
```

`rUniformForest.combine` *Incremental learning for random Uniform Forests*

Description

Combine many random uniform forests on same or different data and/or same and different parameters to obtain one ensemble model. Function acts as the "Reduce" task of MapReduce paradigm. "Map" task has here no defined length, since rUniformForest.combine() is an ensemble of ensembles of any size, parameters or data that can be continuously merged and evaluated as a single model. Native incremental learning is then achieved adding and removing, but never modifying, trees.

Usage

```
rUniformForest.combine(...)
```

Arguments

...	vector enumerating objects of class randomUniformForest

Details

Incremental learning is assumed as a sub-sampling process if distribution is not switching. Data arrive continuously and random Uniform Forests are trained and updated using chunks of the data. Computing time and memory are constant and only prediction time will increase depending on the size (number of trees) of the model. Hence "split-apply-combine" ideas will matter and chunks just have to be processed on all available workstations. For a shifting distribution, changes can happen in the current processed chunk or between one chunk and another. Then, one needs to choose how the model has to be updated. Note that incremental learning is effective because cut-points are chosen randomly, according to the Uniform distribution on the support of each candidate variable. rUniformForest.combine() acts as "compute everywhere, combine in one place" and there is no need to split data here nor to apply the same model. Data are assumed to come from any source, continuously and by chunks, sharing only the same or some common features. Parameters in any model are allowed to change.

Value

an object of class randomUniformForest

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

See Also

`rm.trees`

Examples

```
## not run
## Classification : synthetic data
## get many forests and combine them

# n = 200;  p = 10
## Simulate 'p' gaussian vectors with random parameters between -10 and 10.
# X <- simulationData(n,p)

## Make a rule to create response vector
# epsilon1 = runif(n,-1,1)
# epsilon2 = runif(n,-1,1)
# rule = 2*(X[,1]*X[,2] + X[,3]*X[,4]) + epsilon1*X[,5] + epsilon2*X[,6]

# Y <- as.factor(ifelse(rule > mean(rule), 1, 0))

## create many subsamples
# manyCuts <- cut(1:n, 5, labels = FALSE)

## compute many different models
# ruf1 <- randomUniformForest(X[which(manyCuts == 1),], Y[which(manyCuts == 1)],
# ntree = 30, mtry = 2, BreimanBounds = FALSE, importance = FALSE, threads = 1)

# ruf2 <- randomUniformForest(X[which(manyCuts == 2),], Y[which(manyCuts == 2)],
# ntree = 40, nodesize = 10, BreimanBounds = FALSE, importance = FALSE, threads = 1)

# ruf3 <- randomUniformForest(X[which(manyCuts == 3),], Y[which(manyCuts == 3)],
# ntree = 50, bagging = TRUE, BreimanBounds = FALSE, importance = FALSE, threads = 1)

## combine them in one ensemble
# ruf.combined <- rUniformForest.combine(ruf1, ruf2, ruf3)
# ruf.combined

## or
# rufObjects <- list(ruf1, ruf2, ruf3)
# ruf.combined <- rUniformForest.combine(rufObjects)

## remove 10 older trees
# ruf.combined <- rm.trees(ruf.combined, method = "oldest", howMany = 10)
# ruf.combined

## compute a new model and update
# ruf4 <- randomUniformForest(X[which(manyCuts == 4),], Y[which(manyCuts == 4)],
# ntree = 40, rebalancedsampling = TRUE, BreimanBounds = FALSE, threads = 1)
# ruf.updateCombined <- rUniformForest.combine(ruf.combined, ruf4)
# ruf.updateCombined

# ruf.pred <- predict(ruf.updateCombined, X[which(manyCuts == 5),])

## confusion matrix
# table(ruf.pred, Y[which(manyCuts == 5)])
```

```
## comparison with offline learning (that will always be better,
## except in the case of a shifting distribution)
# ruf.offline <- randomUniformForest(X[which(manyCuts != 5),], Y[which(manyCuts !=5)],
# threads = 1, ntree = 150, BreimanBounds = FALSE)
# ruf.offline.pred <- predict(ruf.offline, X[which(manyCuts == 5),])

## confusion matrix
# table(ruf.offline.pred, Y[which(manyCuts == 5)])
```

`rUniformForest.grow` *Add trees to a random Uniform Forest*

Description

add more trees to the ensemble model

Usage

```
rUniformForest.grow(object, X, Y = NULL, ntree = 100, threads = "auto")
```

Arguments

`object`	an object of class randomUniformForest.
`X`	current matrix or data frame used to compute 'object' model. One can also choose another matrix with same features and appropriate training responses (see below). All parameters will still remain. To combine heterogeneous models, use randomUniformForest.combine() function.
`Y`	vector of training reponses, if one wants to add trees with another matrix (with same features, but different observations). Otherwise, let it to NULL if same model has to be computed on same data.
`ntree`	number of trees to add to the model.
`threads`	compute model in parallel for computers with many cores. Default value is "auto", letting model running on all logical cores minus 1. User can set 'threads' to any values >= 1, depending on the number of cores (including logical).

Details

rUniformForest.grow allows both to add new trees or new model (by adding trees on a new matrix and training responses) built on the same bases than the former. Note that with formula, only new trees can be added, not new model.

Value

an object of class randomUniformForest, containing new and old trees.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

See Also

rUniformForest.big, rUniformForest.combine

Examples

```
# data(iris)
# iris.rf <- randomUniformForest(Species ~ ., iris, ntree = 50, threads = 1, BreimanBounds = FALSE)
# iris.rf <- rUniformForest.grow(iris.rf, iris, ntree = 20, threads = 1)
# iris.rf
```

---

simulationData *Simulation of Gaussian vector*

Description

Simulate a Gaussian vector with 'p' independent components of length 'n'. Parameters of each component are uniformly random and are taken between -10 and 10, with (absolute) standard deviation equals mean.

Usage

```
simulationData(n, p, distrib = rnorm, colinearity = FALSE)
```

Arguments

n	number of observations to draw.
p	number of variables to draw.
distrib	distribution to use. Currently only Gaussian one is supported.
colinearity	not currently used.

Value

a matrix with 'n' observations and 'p' variables.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

Examples

```
X <- simulationData(100,10)
summary(X)
```

splitClusters *Split a cluster on the fly*

Description

Given one (or many) cluster(s), the function splits it in two new clusters.

Usage

```
splitClusters(object, whichOnes, seed = 2014, ...)
```

Arguments

object	an object of class unsupervised.
whichOnes	a vector defining which cluster(s), given by its label, has to be split ?
seed	see unsupervised.randomUniformForest.
...	not currently used.

Value

An object of class unsupervised.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

See Also

mergeClusters, modifyClusters

Examples

```
## not run
## load iris data
# data(iris)

## run unsupervised modelling, removing labels and committing 2 clusters
# iris.uruf = unsupervised.randomUniformForest(iris[,-5], mtry = 1, nodesize = 2,
# threads = 1, clusters = 2)

## view a summary
# iris.uruf

## plot clusters
# plot(iris.uruf)

## split the cluster which has the highest count (cluster 1, in our case)
# iris.urufSplit = splitClusters(iris.uruf, 1)
```

```
## assess fitting comparing average Silhouette before and after
# iris.urufSplit

## plot to see the new clusters
# plot(iris.urufSplit)
```

---

unsupervised.randomUniformForest

Unsupervised Learning with Random Uniform Forests

Description

Unsupervised mode of Random Uniform Forests is designed to provide, in all cases, clustering, dimension reduction, easy visualization, deep variable importance, relations between observations, variables and clusters. It also comes with two specific points: easy assessment (cluster analysis) and dynamic clustering, allowing to change on-the-fly any clustering shape. A three-layer engine is used: dissimilarity matrix, Multidimensional Scaling (MDS) or Spectral decomposition, and k-means or hierarchical clustering. The unsupervised mode does not require the number of clusters to be known, thanks to the gap statistic, and inherits of main algorithmic properties of the supervised mode, allowing (almost) any type of variable.

Usage

```
## S3 method for class 'randomUniformForest'
unsupervised(object,
baseModel = c("proximity", "proximityThenDistance", "importanceThenDistance"),
endModel = c("MDSkMeans", "MDShClust", "MDS", "SpectralkMeans"),
endModelMetric = NULL,
samplingMethod = c("uniform univariate sampling",
"uniform multivariate sampling", "with bootstrap"),
MDSmetric = c("metricMDS", "nonMetricMDS"),
proximityMatrix = NULL,
sparseProximities = FALSE,
outliersFilter = FALSE,
Xtest = NULL,
predObject = NULL,
metricDimension = 2,
coordinates = c(1,2),
bootstrapReplicates = 100,
clusters = NULL,
maxIters = NULL,
importanceObject = NULL,
maxInteractions = 2,
reduceClusters = FALSE,
maxClusters = 5,
mapAndReduce = FALSE,
```

```
OOB = FALSE,
subset = NULL,
seed = 2014,
uthreads = "auto",
...)
## S3 method for class 'unsupervised'
print(x, ...)
## S3 method for class 'unsupervised'
plot(x, importanceObject = NULL, xlim = NULL, ylim = NULL, coordinates = NULL, ...)
```

Arguments

x, object	an object of class randomUniformForest or a matrix (or data frame) from which the unsupervised mode will begin. If matrix or data frame, a full unsupervised learning object will be modelled, beginning by the computation of a randomUniformForest object (following Breiman's ideas). Note that in this latter case, data must be provided using 'Xtest' option.
xlim	vector of 2 points or NULL. Plotting option for zooming in the graph.
ylim	vector of 2 points or NULL. Plotting option for zooming in the graph.
baseModel	'baseModel' defines the way that algorithm will compute dissimilarity matrix. If 'proximity', a proximities matrix of observations will be computed. It can be square, if there is enough memory, or compressed up to a low dimension matrix, using trees and biggest proximities (currently disabled). By default, the matrix is sent to the Multidimensional scaling (MDS) function. If 'proximitythenDistance', the MDS function computes a distance first before applying scaling. If 'importanceThenDistance', then an instance of the importance.randomUniformForest() function will be used. This latter is an alternative of proximities matrix and is useful for difficult problems or when dimension of the problem is high. Note that for all cases, the matrix outputted can be compressed up to a 2 dimensional matrix without loosing much information. For large data sets, compression automatically happens.
endModel	in all cases, dimension reduction is always done. The MDS process is one of the two engines to achieve the task and cannot be overrided (except if one specifies 'SpectralkMeans' in the option). It will send its output to K-means algorithm, if 'MDSkmeans', or to hierarchical clustering, if 'MDShClust'.
endModelMetric	the metric or algorithm that has to be used when computing 'endModel'. See 'algorithm' in kmeans() function.
samplingMethod	method that has to be used to turn unsupervised problem into a supervised one. 'samplingMethod' uses either one (then bootstrap will not happen) or two arguments, the second one always being "with bootstrap" allowing, then, to use bootstrap. Note that 'samplingMethod' is identical to 'unsupervisedMethod' in randomUniformForest. See Details section for more information.
MDSmetric	one of the metric or non-metric to achieve MDS. See cmdscale() function and isoMDS() one of the MASS R package.
proximityMatrix	proximities matrix, of size 'n x n', can be provided, coming from a previous unsupervised.randomUniformForest object or can be external.

`sparseProximities`
: if TRUE, proximities are filled with 1 (which happens when an observation falls in the same terminal node than another, whatever the number of times is) and 0 (an observation does not fall in any terminal node than another). If FALSE, Breiman's formulation is used. Depending on the data, both can be useful.

`outliersFilter`
: if TRUE, outliers in the MDS output will be filtered to achieve clustering. Currently experimental.

`Xtest`
: if one provides a randomUniformForest object for first argument, then 'Xtest' must be filled with a test sample in order to achieve unsupervised learning. In this case the test sample will be clustered with the same classes as in the randomUniformForest object. For example, one can consider 'Xtest' option as a way to add more training cases independently to the randomUniformForest classifier. The main paradigm of this situation is when only a few cases are labelled while the others are not and can (or do) not be used as a validation set. If first argument of unsupervised.randomUniformForest() function is already a test sample, then 'Xtest' is not mandatory but one can, however, put here another test sample in order to know how the model will react. More precisely, first argument will act, as a training sample, in all the layers of the unsupervised learning model while the sample in 'Xtest' will only have effects on the high-level layers, namely the MDS (or spectral) function and the clustering algorithm.

`predObject`
: one may provide here a full prediction object, coming from the `predict.randomUniformForest` function (with option type='all') and any data set. Two cases are possible. The first one involves a supervised problem (coming with a randomUniformForest at first argument of the unsupervised.randomUniformForest() function) that one wants to cluster the predictions. Hence, one will need to put the dataset using 'Xtest' and its predictions in 'predObject'. The second case arises when one wants to speed up computations or exploit the space with a single supervised randomUniformForest model. Any new dataset (to put in 'Xtest') and its prediction object will be assessed by the unsupervised mode partially (using 'MDS' option) or totally. The output will be able to update the model in a supervised manner.

`metricDimension`
: for the MDS or spectral process, the dimension of the final representation. By default, a 'n x p' matrix (or data frame) used for unsupervised learning will end with a 'n x 2' matrix, with decorrelated components, that will be clustered. Using a low dimension avoids issues with large data sets, especially when combining two unsupervised objects. Usually a 'metricDimension' higher than 2 is useless except for spectral decomposition, where its value should be assessed.

`coordinates`
: MDS (or eigenvectors of the spectral decomposition) coordinates to plot. First and second are usually enough since they concentrate almost all the information and are not correlated. In the Spectral case, coordinates should be chosen with care (using the plot method to assess them) since they change a lot the representation. Note that number of chosen coordinates should be small in order to avoid long computation for large datasets.

`bootstrapReplicates`
: bootstrap replicates to be sent to the gap statistic. This latter is used to automatically find the number of clusters (if 'MDSkmeans' is called). See the clusGap() function in the cluster R package.

clusters	number of clusters to find, if one knows it.
maxIters	number of iterations of the K-means algorithm. See kmeans() function.
importanceObject	if a object of class importance is provided and 'baseModel' option is 'importanceThenDistance', then the dissimilarity matrix will be computed using it. Useful if current data are not enough clean and one has a former trustfully randomUniformForest object. Note that dissimilarity matrix based on importance objects are recommended only when 'proximity' or 'proximityThenDistance' have some issues.
maxInteractions	maximum number of interactions when computing an object of class importance. More means lots of details (and possibly noise). Less means faster computation and possibly less reliable results.
reduceClusters	experimental approach to reduce the number of clusters. One should use the modifyClusters function, a user-friendly and reversible algorithm to modify the number of clusters.
maxClusters	maximum number of clusters to find. Used in conjunction with the clusGap() function from the R package "cluster".
mapAndReduce	for large data sets, this option maps (internally) the whole unsupervised learning process in order to decrease memory footprint and to reduce computation time. To avoid computing a 'n x n' proximity matrix, only a subsample of the data is first used. After then the sampled proximity matrix is sent to the MDS process that generates MDS points. These ones are learned and all others MDS points will be predictions. Note that if the data have more than 10000 rows then 'mapAndReduce' will be partially enabled and will try to preserve as much as possible the loss of accuracy that comes from the prediction task.
OOB	see clusteringObservations
subset	an index vector indicating which rows should be used.
seed	seed value to allow convergence (but not reproducibility since random Uniform forests are stochastic but converge). See Details section for more information.
uthreads	allows multi-core computation using, by default and for the current computer, all logical cores minus 1 for each task that can be done using many cores in parallel.
...	allow all options of randomUniformForest to be passed to the unsupervised.randomUniformForest() function, e.g. ntree, mtry, nodesize, ...

Details

The unsupervised mode of Random Uniform Forests is designed to provide dimension reduction, clustering, visualization and a full analysis of features and observations. The process uses a tree-layer engine built around a randomUniformForest object. One may view the whole task as a kind of PCA. The main purpose of unsupervised Random Uniform Forests is to be highly versatile, allowing to assess datasets.

It can be summarized using the following chain :
Unsupervised randomUniformForest object –> dissimilarity matrix –> multidimensional scaling or

spectral decomposition –> clustering algorithm –> clusters –> that can be turn into a supervised object, see as.supervised and analysed in depth with importance.randomUniformForest, with many details, and clusterAnalysis, which is both compact and granular.

First step involves Breiman's ideas. Since Random Uniform Forests inherit of all properties of Random Forests, they are able to implement the key concepts provided by Breiman for the unsupervised case :
- create a synthetic data set by scrambling independently (for example uniformly) each column of the original dataset,
- merge both synthetic and original dataset and give each one a label,
- run Random (Uniform) Forests to this new dataset and retrieve OOB errors,
- the lower the OOB errors, the more clustering will be easy. If error is too high, say more than 50 percent, then there is (almost) no hope,
- once the forest classifier is built, one can now move to the second step.

The second step used proximities matrix. If data are a 'n x p' matrix (or data frame) then proximities matrix will have a 'n x n' size, meaning that for each observation, one will search, for each tree, which other observation falls in the same terminal node, then increase the proximity of the two observations by one. At the end, all observations and all trees will have been processed, leading to a proximities matrix that will be normalized by the number of trees. Since it can be very large, it is possible (but currently disabled due to the compromise that is needed with large datasets between accuracy and computing time) to use a 'n x B' proximities matrix, where 'B' is the number of trees. A further step is to use a 'n x q' (biggest) proximities matrix, where 'q' can be as small as 2 in order to compress the data to their maximum. Note that proximities matrix has a second mode, for difficult cases, which produces a binary matrix. In our implementation, we considered all cases where n > 10,000 as problems that must be predicted, since the proximities matrix will, otherwise, have size greater than 800 Mo. In this case, learning points imply to learn MDS (or spectral) ones as responses and the data with n <= 10,000 as predictors. Then predict the remaining data.

Once proximities matrix (or dissimilarity matrix using for example '1 - proximities') has been computed, the third step is to enter in the MDS (or spectral) process. MDS is needed for dimension reduction (if not already happened) and mainly to generate decorrelated components in which the points will reside. The two first components are usually enough to get good visualization. Unlike PCA, they are used only to achieve the best possible separation between points. Note that we allow proximities to be transformed into distances since we found that it produces sometimes better results. In the case of spectral decomposition, two eigenvectors are usually enough to get a good clustering scheme, and the algorithm allows to arbitrary choose them, while the default option limits the number of eigenvectors.

The next step concludes the three-layer engine by calling a clustering algorithm, preferably K-means, to partition the MDS (or spectral) points since we use the features only in the early phase of the algorithm. Hence, we manipulate coordinates and points in a new space where MDS (or spectral) is the rule that matters. K-means algorithm is simply a way to provide a measurable way of the clusters which already exist, since clustering is almost done earlier. Note that the number of clusters can be automatically adjusted by the gap statistic (Tibshirani, Walther, Hastie, 2001). If not enough, clusters structure can be instantaneously modified, merged or split (see modifyClusters, mergeClusters, splitClusters) letting the silhouette coefficient (Rousseeuw, 1987) have the last word.

The unsupervised learning is then partially achieved. An object with all tools is generated and can be returned in order to be assessed by the randomUniformForest algorithm that will provide a deeper analysis and visualization tools :

- most influential features,
- interactions,
- partial dependencies,
- dimension reduction,
- visualization,
- cluster analysis,
- ...

The unsupervised model is turned into a supervised one using the `as.supervised` function, with all options one need to call for the supervised case, then doing the analysis by the `importance.randomUniformForest` function to get full details. Moreover, data are now turned into a training sample for the algorithm. If new data become available, one may use the supervised case or the unsupervised one using `update.unsupervised`. The former has the great advantage to have a complexity, when predicting, in O(B*n*log(n)), where 'B' is the number of trees. Indeed, due to the lot of computation involved, the unsupervised case requires much more time than a simple call to K-means algorithm but also provides much more details and more abilities to cluster.

When going toward large datasets, the unsupervised mode becomes hybrid with a fully unsupervised mode for a random subsample of the data and a supervised mode that learns MDS (or spectral) points and predict them for the remaining rows of the sample. This allows to strongly reduce computation time and to make the resulted object recyclable. New data can be learned incrementally (combining or updating objects) and/or separately to the former object.

Note that the whole engine is stochastic, with almost no possibility of exact reproducibility using the set.seed() function. However, since Random Uniform Forests converge, seed has been added to the second part of the first layer, the creation of the synthetic dataset. It means that most of the work to achieve a good clustering representation is devoted to the `randomUniformForest` part, allowing one to assess parameters independently for each layer and look for the ones that have the main effect on the clustering. Moreover, the stochastic nature is the main argument of unsupervised learning in Random Uniform Forests, since it is first expected that random models produce random effects. As a consequence, the model is first self-consistent. Efficiency of results can be assessed independently, using `clusterAnalysis` which makes the bridge between clustering and original data.

Value

An object of class unsupervised, which is a list with the following components:

`proximityMatrix`
: the resulted dissimilarity matrix.

`MDSModel`
: the resulted Multidimensional scaling or spectral decomposition model.

`unsupervisedModel`
: the resulted unsupervised model with clustered observations in unsupervisedModel$cluster.

`largeDataLearningModel`
: if the dataset is large, the resulted model that learned a sample of the MDS points, and predicted others points.

`gapStatistics`
: if K-means algorithm has been called, the results of the gap statistic. Otherwise NULL.

rUFObject	Random Uniform Forests object.
nbClusters	Number of clusters found.
params	options of the model.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

References

Abreu, N., 2011. Analise do perfil do cliente Recheio e desenvolvimento de um sistema promocional. Mestrado em Marketing, ISCTE-IUL, Lisbon

Breiman and Cutler web site : http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

Cox, T. F., Cox, M. A. A., 2001. Multidimensional Scaling. Second edition. *Chapman and Hall*.

Gower, J. C., 1966. Some distance properties of latent root and vector methods used in multivariate analysis. *Biometrika* 53, 325-328.

Kaufman, L., Rousseeuw, P.J., 1990. Finding Groups in Data: An Introduction to Cluster Analysis (1 ed.). New York: John Wiley.

Lloyd, S. P., 1957, 1982. Least squares quantization in PCM. Technical Note, Bell Laboratories. Published in 1982 in *IEEE Transactions on Information Theory* 28, 128-137.

Murtagh, F., Legendre, P., 2013. Ward's hierarchical agglomerative clustering method: which algorithms implement Ward's criterion? *Journal of Classification* (in press).

Ng, A. Y., Jordan, M. I., Weiss, Y., 2002. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 2, 849-856.

Rousseeuw, P. J., 1987. Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis. *Computational and Applied Mathematics* 20, 53-65.

Tibshirani, R., Walther, G., Hastie, T., 2001. Estimating the number of data clusters via the Gap statistic. *Journal of the Royal Statistical Society* B, 63, 411-423.

See Also

modifyClusters, mergeClusters, splitClusters clusteringObservations, as.supervised, update.unsupervised, combineUnsupervised, clusterAnalysis

Examples

```
## not run
#### A - Overview : the famous iris dataset

## load data
# data(iris)

## run unsupervised modelling, removing labels :
## Default options, letting the 'gap statistic' find the number of clusters.

## Note: the stochastic nature of the algorithm acts as an hidden parameter
## whose one has to take care of.
```

```
## default (MDS) clustering
# iris.uruf = unsupervised.randomUniformForest(iris[,-5])

## for iris data, default option is not stable.

## Note: correlation between covariates play a key role on stability and clustering.
## Neutralize correlation requires to choose splitting variable at random.

## Increase 'nodesize' to get more stability
# iris.uruf2 = unsupervised.randomUniformForest(iris[,-5], mtry = 1, nodesize = 2)

## would be better and stable (if repeated)

## Regular companions (from point 1 to point 14):
## 1 - Summary with raw assessment
# iris.uruf2

## 2 - Plot in the new space in 2D
# plot(iris.uruf2)

## 3 - Modify clusters (increasing or decreasing the number of clusters
## and looking the variations of the silhouette coefficient).
## For example, if 4 clusters are found (since we know there are 3) :

# iris.uruf4Clusters = unsupervised.randomUniformForest(iris[,-5], mtry = 1, nodesize = 2,
# clusters = 4)
# iris.uruf3Clusters = modifyClusters(iris.uruf4Clusters, decreaseBy = 1)

## 4 - (or) Merge clusters if there are too many
## merge the second and the third

## Note: one can play with modify/merge/split and 'plot' many times in order to improve
## the clustering
# iris.urufmerge = mergeClusters(iris.urufnsupervised4Clusters, c(1,2))

## 5 - Alternatives : try Spectral clustering
## 'coordinates' specify which eigenvectors have to be sent to the clustering algorithm
## 'metricDimension' states for the maximal number of eigenvectors to use.
## Try 'coordinates = c(2,3)' and assess it by visualization

# iris.urufSpectral = unsupervised.randomUniformForest(iris[,-5], mtry = 1, nodesize = 2,
# endModel = "SpectralkMeans", metricDimension = 3, coordinates = c(2,3))

# plot(iris.urufSpectral)

## or try them all and choose/visualize the best representation
# iris.urufSpectral = unsupervised.randomUniformForest(iris[,-5], mtry = 1, nodesize = 2,
# endModel = "SpectralkMeans", metricDimension = 5, coordinates = c(1:5))

# plot(iris.urufSpectral, coordinates = c(1,2))
# plot(iris.urufSpectral, coordinates = c(2,3))
## ...
```

```
#### B - Full example with details
## Wholesale customers data (UCI machine learning repository)

# URL = "http://archive.ics.uci.edu/ml/machine-learning-databases/00292/"
# datasetName = "Wholesale%20customers%20data.csv"
# wholesaleCustomers = read.csv(paste(URL, datasetName, sep =""))

## modelling : three ways
## - let the algorithm deal with all layers: categorical features, number of clusters,...
## - control the first layer to get stability (proximities matrix) and let the
## algorithm control the others.
## - control all manually.

## first way
# wholesaleCustomers.uruf.1 = unsupervised.randomUniformForest(wholesaleCustomers)

## assess quality of the clustering :
# wholesaleCustomers.uruf.1

## second way (we keep it) : use 'bagging' option to let each feature define a candidate node,
## use 'sparseProximities' that provides a better separation

## third way (seems the best) : spectral clustering, specifying clusters and changing seed
## to stabilize the clustering scheme when modelling is repeated.

# wholesaleCustomers.uruf.3 = unsupervised.randomUniformForest(wholesaleCustomers,
# ntree = 500, sparseProximities = TRUE, endModel = "SpectralkMeans", bagging = TRUE,
# categorical = c("Channel", "Region"), clusters = 3, seed = 2016)

## Speed up computation: less trees + increasing (minimal) node size + do not use logical cores
# wholesaleCustomers.uruf.3 = unsupervised.randomUniformForest(wholesaleCustomers,
# ntree = 100, nodesize = 10, sparseProximities = TRUE, endModel = "SpectralkMeans",
# bagging = TRUE, categorical = c("Channel", "Region"), clusters = 3, seed = 2016)

## Note: 'Channel' and 'Region' are categorical and differ from other products (consumer goods)
## At first, we keep all, seeking links between the context of consumption and the products.

# wholesaleCustomers.uruf.2 = unsupervised.randomUniformForest(wholesaleCustomers,
# ntree = 500, sparseProximities = TRUE, bagging = TRUE, categorical = c("Channel", "Region"))

## Regular companions :
## 6 - Assess the randomUniformForest object (low OOB error is usually better but not a rule)
# wholesaleCustomers.uruf.2$rUF

## 7 - Look for influential variables (before clustering)
# summary(wholesaleCustomers.uruf.2$rUF)

## assess quality of the clustering and remove/add/merge clusters to see if things are better
# wholesaleCustomers.uruf.2
# plot(wholesaleCustomers.uruf.2)

## 8 - Get details : turning first the model into a supervised one
```

```
# wholesaleCustomers.ruf.2 = as.supervised(wholesaleCustomers.uruf.2, wholesaleCustomers,
# ntree = 500, categorical = c("Channel", "Region"), BreimanBounds = FALSE)

## 9 - Assess the 'Learning clusters' process
# wholesaleCustomers.ruf.2

## 10 - Get variable importance : leading to a full analysis and visualization
# wholesaleCustomers.importance = importance(wholesaleCustomers.ruf.2,
# Xtest = wholesaleCustomers, maxInteractions = 8)

## Visualize all: features, interactions, partial dependencies,...
## Note: tile window in the R menu to see all plots. Loop over the prompt to see
## all matched partial dependencies

# plot(wholesaleCustomers.importance, Xtest = wholesaleCustomers, whichOrder = "all")
## For details, type vignette("VariableImportanceInRandomUniformForests",
## package = "randomUniformForest")
## Note : Variable importance strongly depends on the model and the chosen clusters

## 11 - more visualization: (another look on 'variable importance over labels')
## for each cluster, we can see which variables matter and their order.
# featuresCluster1 = partialImportance(wholesaleCustomers,
# wholesaleCustomers.importance, whichClass = 1)
# ...

## 12 - Refining analysis : two possible levels, with and without Region (and Channel)

## 12.a - visualizing clusters and features: keeping Region and Channel
# plot(wholesaleCustomers.uruf.2, importanceObject = wholesaleCustomers.importance)

## 12.b - Removing Region and Channel to better assess others products:
## one may want to lead an analysis independent to the context

## If Region and/or Channel matter:
## - Solution 1: reach stability by first trying and repeating full random models,
## leading to a new model (with variables of interest) and the former one.
## - Solution 2: clustering with all features (as above),
## assessment on variables of interest handled by the importance() function.
## - Solution 3: clustering with all features (as above), clusterAnalysis() function
## is able to provide a granular view. We choose this solution.

## 13 - Cluster analysis: aggregated links between observations, features and clusters
## clusterAnalysis mainly shows what is happening when looking deeper in details

## 13.a - first, look inside clusters
# wholesaleCustomersFinalSummary.ruf = clusterAnalysis(wholesaleCustomers.importance,
# wholesaleCustomers, components = 3, maxFeatures = 4)

## 13.b - Numerical and categorical features aggregation
## same function, more options
## Note : here features are valuable, revenue for each. Hence one can exploit that.

# wholesaleCustomersFinalSummary.ruf = clusterAnalysis(wholesaleCustomers.importance,
```

```
# wholesaleCustomers, components = 3, maxFeatures = 4,
# clusteredObject = wholesaleCustomers.uruf.2, categorical = c("Channel", "Region"))

## 14 - Conciliate analysis
## clusterAnalysis( ) provides both influential features (leading to a better clustering)
## and valuable ones. Due to the purpose of the task, both types can be different
## but should give insights about the next task.
## Suppose one wants to maximize revenues. Where could one put the efforts ?

## First, look to the global point of view (clusters) :
# revenuePerCluster =
# rowSums(wholesaleCustomersFinalSummary.ruf$numericalFeaturesAnalysis[,-c(1,2)])

## Then, look to the local point of view :
## for example, influential features in one cluster might be the ones that one need
## to develop in  another one... avoiding brute force techniques

## Notes:
## 1 - cluster analysis must be consistent with results of importance( ) function.
## If not, then something is going wrong in the modelling.
## 2 - To complete an analysis, one should take care of the whole process.
## For example, changing the clustering may change a lot the analysis.
```

`update.unsupervised` *Update Unsupervised Learning object*

Description

Update unsupervised learning object with new data in order to achieve incremental learning. New MDS (or spectral) points are predicted with new data and learning of MDS (spectral) points of the former unsupervised object. Note that new data are expected to have the same distribution than previous ones.

Usage

```
## S3 method for class 'unsupervised'
update(object, X = NULL,
oldData = NULL,
mapAndReduce = FALSE,
updateModel = FALSE,
...)
```

Arguments

`object`	(vector of) objects of class unsupervised, coming from `unsupervised.randomUniformForest`, that needs to be updated.
`X`	new data frame or matrix with same variables than the ones used to build 'object'.

oldData	former data frame or matrix needed, if 'object' was built from a small dataset (less than 10000 rows).
mapAndReduce	if TRUE, 'X' will be learned by chunks that will be combined to build the forest classifier for the MDS points.
updateModel	if TRUE, the resulted object will also embed a learned model of all the MDS points (hence learning its own predictions). Note that this option is recommended to be enabled since it will allow to assess the resulted model against, for example, any unsupervised algorithm, or itself, based on a subsample of 'X'.
...	all hyper parameters of randomUniformForest.

Value

An object of class unsupervised, which is a list with the following components:

proximityMatrix	the resulted dissimilarity matrix.
MDSModel	the resulted Multidimensional scaling model.
unsupervisedModel	the resulted unsupervised model with clustered observations in unsupervisedModel$cluster.
largeDataLearningModel	if the dataset is large, the resulted model that learned a sample of the MDS points, and predicted others points.
gapStatistics	if K-means algorithm has been called, the results of the gap statistic. Otherwise NULL.
rUFObject	Random Uniform Forests object.
nbClusters	Number of clusters found.
params	options of the model.

Author(s)

Saip Ciss <saip.ciss@wanadoo.fr>

See Also

combineUnsupervised, modifyClusters, mergeClusters, clusteringObservations, as.supervised

Examples

```
## not run
## Water Treatment Plant Data Set
## Data can be download at https://archive.ics.uci.edu/ml/datasets/Water+Treatment+Plant

# URL = "http://archive.ics.uci.edu/ml/machine-learning-databases/water-treatment/"
# dataset = "water-treatment.data"

# X = read.table(paste(URL, dataset, sep= ""), sep = ",")
```

```
## 1- Preprocessing
## first, look at the first column and format date
#  Dates = rm.string(as.character(X[,1]), "D-")
#  DatesAsStringTable = do.call(rbind, strsplit(Dates, "/"))
#  DatesasNumericTable = t(apply(DatesAsStringTable, 1, as.numeric))

##  Then, transform data as a pure R matrix and add new dates
#  XX = as.true.matrix(X)[,-1]
#  XX = cbind(DatesasNumericTable, XX)
#  colnames(XX)[1:3] = c("day", "month", "year")

# Look the new data
# head(XX)
# str(XX)

## and fill missing values,
# X.imputed = fillNA2.randomUniformForest(XX)

## 2 - run unsupervised analysis on the first half of dataset
# subset.1 = 1:floor(nrow(X.imputed)/2)
# WaterTreatment.model.1 = unsupervised.randomUniformForest(X.imputed, subset = subset.1,
# baseModel = "proximityThenDistance",  seed = 2014)

## assess roughly the model and visualize
#  WaterTreatment.model.1

## 3 - update model with the second half of dataset
# WaterTreatment.updated = update.unsupervised(WaterTreatment.model.1,
# X.imputed[-subset.1,], oldData = X.imputed[subset.1,])

# WaterTreatment.updated

## view how MDS points have been learned :
## first component
# WaterTreatment.updated$largeDataLearningModel[[1]]

## second component
# WaterTreatment.updated$largeDataLearningModel[[2]]
```

`wineQualityRed` *Wine Quality Data Set*

Description

Two datasets are included, related to red and white vinho verde wine samples, from the north of Portugal. The goal is to model wine quality based on physicochemical tests.

Usage

```
wineQualityRed
```

Format

A matrix (for red wine) containing 1599 observations and 12 attributes.

Source

http://archive.ics.uci.edu/ml/datasets/Wine+Quality

References

Paulo Cortez, University of Minho, Guimaraes, Portugal, http://www3.dsi.uminho.pt/pcortez A. Cerdeira, F. Almeida, T. Matos and J. Reis, Viticulture Commission of the Vinho Verde Region(CVRVV), Porto, Portugal 2009

P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties.In Decision Support Systems, Elsevier, 47(4):547-553, 2009.

Index