

# Package: CORElearn (via r-universe)

October 1, 2024

**Title** Classification, Regression and Feature Evaluation

**Version** 1.57.3

**Date** 2022-11-18

**Author** Marko Robnik-Sikonja and Petr Savicky

**Maintainer** ``Marko Robnik-Sikonja" <marko.robnik@fri.uni-lj.si>

**Description** A suite of machine learning algorithms written in C++ with the R interface contains several learning techniques for classification and regression. Predictive models include e.g., classification and regression trees with optional constructive induction and models in the leaves, random forests, kNN, naive Bayes, and locally weighted regression. All predictions obtained with these models can be explained and visualized with the 'ExplainPrediction' package. This package is especially strong in feature evaluation where it contains several variants of Relief algorithm and many impurity based attribute evaluation functions, e.g., Gini, information gain, MDL, and DKM. These methods can be used for feature selection or discretization of numeric attributes. The OrdEval algorithm and its visualization is used for evaluation of data sets with ordinal features and class, enabling analysis according to the Kano model of customer satisfaction. Several algorithms support parallel multithreaded execution via OpenMP. The top-level documentation is reachable through ?CORElearn.

**License** GPL-3

**URL** <http://lkm.fri.uni-lj.si/rmarko/software/>

**Imports** cluster,stats,nnet,plotrix,rpart,plot

**Suggests** lattice,MASS,ExplainPrediction

**NeedsCompilation** yes

**Repository** CRAN

**Date/Publication** 2022-11-18 14:10:06 UTC

## Contents

CORElearn-package	3
attrEval	5
auxTest	9
calibrate	10
classDataGen	12
classPrototypes	15
CORElearn-internal	16
CoreModel	17
cvGen	21
destroyModels	23
discretize	24
display.CoreModel	27
getCoreModel	28
getRFsizes	29
getRpartModel	30
helpCore	31
infoCore	37
modelEval	38
noEqualRows	41
ordDataGen	42
ordEval	43
paramCoreIO	47
plot.CoreModel	48
plot.ordEval	50
predict.CoreModel	53
preparePlot	54
regDataGen	55
reliabilityPlot	57
rfAttrEval	58
rfClustering	60
rfOOB	61
rfOutliers	62
rfProximity	63
saveRF	64
testCore	66
versionCore	67
<b>Index</b>	<b>68</b>

## Description

The package CORElearn is an R port of CORElearn data mining system. It provides various classification and regression models as well as algorithms for feature selection and evaluation. Several algorithms support parallel multithreaded execution via OpenMP (see details in function descriptions). It is possible to run many functions outside the R environment. The description and source code is available on the package web site <http://lkm.fri.uni-lj.si/rmarko/software/>.

## Details

The main functions are

- `CoreModel` which constructs classification or regression model.
  - Classification models available:
    - \* random forests with optional local weighing of basic models
    - \* decision tree with optional constructive induction in the inner nodes and/or models in the leaves
    - \* kNN and kNN with Gaussian kernel,
    - \* naive Bayes.
  - Regression models:
    - \* regression trees with optional constructive induction in the inner nodes and/or models in the leaves,
    - \* linear models with pruning techniques
    - \* locally weighted regression
    - \* kNN and kNN with Gaussian kernel.
- `predict.CoreModel` predicts with classification model labels and probabilities of new instances. For regression models it returns the predicted function value.
- `plot.CoreModel` graphically visualizes trees and random forest models
- `modelEval` computes some statistics from predictions
- `attrEval` evaluates the quality of the attributes (dependent variables) with the selected heuristic method. Feature evaluation algorithms are various variants of Relief algorithms (ReliefF, RReliefF, cost-sensitive ReliefF, etc), gain ratio, gini-index, MDL, DKM, information gain, MSE, MAE, etc.
- `ordEval` evaluates ordinal attributes with `ordEval` algorithm and visualizes them with `plot.ordEval`,
- `infoCore` outputs certain information about CORElearn methods,
- `helpCore` prints short description of a given parameter,
- `paramCoreIO` reads/writes parameters for given model from/to file,
- `versionCore` outputs version of the package from underlying C++ library.

Some of the internal structures of the C++ part are described in [CORElearn-internal](#).

For an automatically generated list of functions use `help(package=CORElearn)` or `library(help=CORElearn)`.

ut this feature is currently not supported on all platforms and may interfere with other means of parallelization used in R, like package `paralell`. It is tested to works on Windows, Linux, and Mac.

For certain platforms multithreaded execution is not supported, since current set of compilers at CRAN do not fully support OpenMP. Also note that OpenMP execution may interfere with other means of parallelization on certain platforms. E.g., interference with package `parallel` is reported on Windows which can be prevented by setting parameter `maxThreads=1`. For platforms other than Linux, Windows, and OsX to support multithreading it is possible to recompile the package with appropriate tools and compilers (modify `Makefile` or `Makefile.win` in `src` folder, or consult authors).

### Author(s)

Marko Robnik-Sikonja, Petr Savicky

### References

Marko Robnik-Sikonja, Igor Kononenko: Theoretical and Empirical Analysis of ReliefF and RReliefF. *Machine Learning Journal*, 53:23-69, 2003

Marko Robnik-Sikonja: Improving Random Forests. In J.-F. Boulicaut et al.(Eds): *ECML 2004, LNAI 3210*, Springer, Berlin, 2004, pp. 359-370

Marko Robnik-Sikonja, Koen Vanhoof: Evaluation of ordinal attributes at value level. *Knowledge Discovery and Data Mining*, 14:225-243, 2007

Marko Robnik-Sikonja: Experiments with Cost-sensitive Feature Evaluation. In Lavrac et al.(eds): *Machine Learning, Proceedings of ECML 2003*, Springer, Berlin, 2003, pp. 325-336

Majority of these references are available also from <http://lkm.fri.uni-lj.si/rmarko/papers/>

### See Also

[CoreModel](#), [predict.CoreModel](#), [plot.CoreModel](#), [modelEval](#), [attrEval](#), [ordEval](#), [plot.ordEval](#), [helpCore](#), [paramCoreIO](#), [infoCore](#), [versionCore](#), [CORElearn-internal](#), [classDataGen](#), [regDataGen](#), [ordDataGen](#).

### Examples

```
# load the package
library(CORElearn)
cat(versionCore(),"\n")

# use iris data set
trainIdxs <- sample(x=nrow(iris), size=0.7*nrow(iris), replace=FALSE)
testIdxs <- c(1:nrow(iris))[-trainIdxs]

# build random forests model with certain parameters
# setting maxThreads to 0 or more than 1 forces
# utilization of several processor cores
modelRF <- CoreModel(Species ~ ., iris[trainIdxs,], model="rf",
                     selectionEstimator="MDL",minNodeWeightRF=5,
```

```

rfNoTrees=100, maxThreads=1)

print(modelRF) # simple visualization, test also others with function plot

# prediction on testing set
pred <- predict(modelRF, iris[testIdxs,], type="both")

# compute statistics
mEval <- modelEval(modelRF, iris[["Species"]][testIdxs], pred$class, pred$prob)
print(mEval)

## Not run:
# explain predictions on the level of model and individual instances
require(ExplainPrediction)
explainVis(modelRF, iris[trainIdxs,], iris[testIdxs,], method="EXPLAIN",
           visLevel="model", problemName="iris", fileType="none",
           classValue=1, displayColor="color")
# turn on the history in visualization window to see all instances
explainVis(modelRF, iris[trainIdxs,], iris[testIdxs,], method="EXPLAIN",
           visLevel="instance", problemName="iris", fileType="none",
           classValue=1, displayColor="color")

## End(Not run)
# Clean up, otherwise the memory is still taken
destroyModels(modelRF) # clean up

# evaluate features in given data set with selected method
# instead of formula interface one can provide just
# the name or index of target variable
estReliefF <- attrEval("Species", iris,
                      estimator="ReliefFexpRank", ReliefIterations=30)
print(estReliefF)

# evaluate ordered features with ordEval
profiles <- ordDataGen(200)
est <- ordEval(class ~ ., profiles, ordEvalNoRandomNormalizers=100)
# print(est)

```

---

attrEval

*Attribute evaluation*


---

## Description

The method evaluates the quality of the features/attributes/dependent variables specified by the formula with the selected heuristic method. Feature evaluation algorithms available for classification problems are various variants of Relief and ReliefF algorithms (ReliefF, cost-sensitive ReliefF, ...), and impurity-based algorithms (information gain, gain ratio, gini-index, MDL, DKM, etc). For regression problems there are RREliefF, MSEofMean, MSEofModel, MAEofModel, ... Parallel execution on several cores is supported for speedup.

**Usage**

```
attrEval(formula, data, estimator, costMatrix = NULL,
         outputNumericSplits=FALSE, ...)
```

**Arguments**

formula	Either a formula specifying the attributes to be evaluated and the target variable, or a name of target variable, or an index of target variable.
data	Data frame with evaluation data.
estimator	The name of the evaluation method.
costMatrix	Optional cost matrix used with certain estimators.
outputNumericSplits	Controls of the output contain the best split point for numeric attributes. This is only sensible for impurity based estimators (like information gain, gini, MDL, gain ratio, etc. in classification, and MSEofMean in regression). The default value of parameter <code>binaryEvaluateNumericAttributes = TRUE</code> shall not be modified in this case. If the value of <code>outputNumericSplits = TRUE</code> , the output is a list with attribute evaluations and numeric attributes' splits (instead of a single vector with evaluations). See the returned value description.
...	Additional options used by specific evaluation methods as described in <a href="#">helpCore</a> .

**Details**

The parameter `formula` can be interpreted in three ways, where the formula interface is the most elegant one, but inefficient and inappropriate for large data sets. See also examples below. As formula one can specify:

**an object of class** `formula` used as a mechanism to select features (attributes) and prediction variable (class). Only simple terms can be used and interaction expressed in formula syntax are not supported. The simplest way is to specify just response variable: `class ~ .`. In this case all other attributes in the data set are evaluated. Note that formula interface is not appropriate for data sets with large number of variables.

**a character vector** specifying the name of target variable, all the other columns in data frame `data` are used as predictors.

**an integer** specifying the index of of target variable in data frame `data`, all the other columns are used as predictors.

The optional parameter **costMatrix** can provide nonuniform cost matrix to certain cost-sensitive measures (ReliefFexpC, ReliefFavgC, ReliefFpe, ReliefFpa, ReliefFsmp, GainRatioCost, DKM-cost, ReliefKukar, and MDLsmp). For other measures this parameter is ignored. The format of the matrix is `costMatrix(true class, predicted class)`. By default a uniform costs are assumed, i.e., `costMatrix(i, i) = 0`, and `costMatrix(i, j) = 1`, for `i` not equal to `j`.

The **estimator** parameter selects the evaluation heuristics. For classification problem it must be one of the names returned by `infoCore(what="attrEval")` and for regression problem it must be one of the names returned by `infoCore(what="attrEvalReg")` Majority of these feature evaluation measures are described in the references given below, here only a short description is given. For classification problem they are

- "**ReliefEqualK**" ReliefF algorithm where k nearest instances have equal weight.
- "**ReliefExpRank**" ReliefF algorithm where k nearest instances have weight exponentially decreasing with increasing rank. Rank of nearest instance is determined by the increasing (Manhattan) distance from the selected instance. This is a default choice for methods taking conditional dependencies among the attributes into account.
- "**ReliefBestK**" ReliefF algorithm where all possible k (representing k nearest instances) are tested and for each feature the highest score is returned. Nearest instances have equal weights.
- "**Relief**" Original algorithm of Kira and Rendel (1991) working on two class problems.
- "**InfGain**" Information gain.
- "**GainRatio**" Gain ratio, which is normalized information gain to prevent bias to multi-valued attributes.
- "**MDL**" Acronym for Minimum Description Length, presents method introduced in (Kononenko, 1995) with favorable bias for multi-valued and multi-class problems. Might be the best method among those not taking conditional dependencies into account.
- "**Gini**" Gini-index.
- "**MyopicReliefF**" Myopic version of ReliefF resulting from assumption of no local dependencies and attribute dependencies upon class.
- "**Accuracy**" Accuracy of resulting split.
- "**ReliefFmerit**" ReliefF algorithm where for each random instance the merit of each attribute is normalized by the sum of differences in all attributes.
- "**ReliefFdistance**" ReliefF algorithm where k nearest instances are weighed directly with its inverse distance from the selected instance. Usually using ranks instead of distance as in ReliefFexpRank is more effective.
- "**ReliefFsqrDistance**" ReliefF algorithm where k nearest instances are weighed with its inverse square distance from the selected instance.
- "**DKM**" Measure named after Dietterich, Kearns, and Mansour who proposed it in 1996.
- "**ReliefFexpC**" Cost-sensitive ReliefF algorithm with expected costs.
- "**ReliefFavgC**" Cost-sensitive ReliefF algorithm with average costs.
- "**ReliefFpe**" Cost-sensitive ReliefF algorithm with expected probability.
- "**ReliefFpa**" Cost-sensitive ReliefF algorithm with average probability.
- "**ReliefFsmp**" Cost-sensitive ReliefF algorithm with cost sensitive sampling.
- "**GainRatioCost**" Cost-sensitive variant of GainRatio.
- "**DKMcost**" Cost-sensitive variant of DKM.
- "**ReliefKukar**" Cost-sensitive Relief algorithm introduced by Kukar in 1999.
- "**MDLsmp**" Cost-sensitive variant of MDL where costs are introduced through sampling.
- "**ImpurityEuclid**" Euclidean distance as impurity function on within node class distributions.
- "**ImpurityHellinger**" Hellinger distance as impurity function on within node class distributions.
- "**UniformDKM**" Dietterich-Kearns-Mansour (DKM) with uniform priors.
- "**UniformGini**" Gini index with uniform priors.
- "**UniformInf**" Information gain with uniform priors.

- "**UniformAccuracy**" Accuracy with uniform priors.
- "**EqualDKM**" Dietterich-Kearns-Mansour (DKM) with equal weights for splits.
- "**EqualGini**" Gini index with equal weights for splits.
- "**EqualInf**" Information gain with equal weights for splits.
- "**EqualHellinger**" Two equally weighted splits based Hellinger distance.
- "**DistHellinger**" Hellinger distance between class distributions in branches.
- "**DistAUC**" AUC distance between splits.
- "**DistAngle**" Cosine of angular distance between splits.
- "**DistEuclid**" Euclidean distance between splits.

For regression problem the implemented measures are:

- "**RReliefFequalK**" RReliefF algorithm where k nearest instances have equal weight.
- "**ReliefExpRank**" RReliefF algorithm where k nearest instances have weight exponentially decreasing with increasing rank. Rank of nearest instance is determined by the increasing (Manhattan) distance from the selected instance. This is a default choice for methods taking conditional dependencies among the attributes into account.
- "**RReliefFbestK**" RReliefF algorithm where all possible k (representing k nearest instances) are tested and for each feature the highest score is returned. Nearest instances have equal weights.
- "**RReliefFwithMSE**" A combination of RReliefF and MSE algorithms.
- "**MSEofMean**" Mean Squared Error as heuristic used to measure error by mean predicted value after split on the feature.
- "**MSEofModel**" Mean Squared Error of an arbitrary model used on splits resulting from the feature. The model is chosen with parameter `modelTypeReg`.
- "**MAEofModel**" Mean Absolute Error of an arbitrary model used on splits resulting from the feature. The model is chosen with parameter `modelTypeReg`. If we use median as the model, we get robust equivalent to `MSEofMean`.
- "**RReliefFdistance**" RReliefF algorithm where k nearest instances are weighed directly with its inverse distance from the selected instance. Usually using ranks instead of distance as in `RReliefExpRank` is more effective.
- "**RReliefFsqrDistance**" RReliefF algorithm where k nearest instances are weighed with its inverse square distance from the selected instance.

There are some additional parameters ... available which are used by specific evaluation heuristics. Their list and short description is available by calling [helpCore](#). See Section on attribute evaluation.

The attributes can also be evaluated via random forest out-of-bag set with function [rfAttrEval](#).

Evaluation and visualization of ordered attributes is covered in function [ordEval](#).

## Value

The method returns a vector of evaluations for the features in the order specified by the formula. In case of parameter `binaryEvaluateNumericAttributes=TRUE` the method returns a list with two components: `attrEval` and `splitPointNum`. The `attrEval` contains a vector of evaluations for the features in the order specified by the formula. The `splitPointNum` contains the split points of numeric attributes which produced the given attribute evaluation scores.



**Author(s)**

Marko Robnik-Sikonja

**References**

Marko Robnik-Sikonja, Igor Kononenko: Theoretical and Empirical Analysis of ReliefF and RReliefF. *Machine Learning Journal*, 53:23-69, 2003

Marko Robnik-Sikonja: Experiments with Cost-sensitive Feature Evaluation. In Lavrac et al.(eds): *Machine Learning, Proceedings of ECML 2003*, Springer, Berlin, 2003, pp. 325-336

Igor Kononenko: On Biases in Estimating Multi-Valued Attributes. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'95)*, pp. 1034-1040, 1995

Some of these references are available also from <http://lkm.fri.uni-lj.si/rmarko/papers/>

**See Also**

[CORElearn](#), [CoreModel](#), [rfAttrEval](#), [ordEval](#), [helpCore](#), [infoCore](#).

**Examples**

```
# use iris data

# run method ReliefF with exponential rank distance
estReliefF <- attrEval(Species ~ ., iris,
                      estimator="ReliefFexpRank", ReliefIterations=30)
print(estReliefF)

# alternatively and more appropriate for large data sets
# one can specify just the target variable
# estReliefF <- attrEval("Species", iris, estimator="ReliefFexpRank",
#                       ReliefIterations=30)

# print all available estimators
infoCore(what="attrEval")
```

---

auxTest

*Test functions for manual usage*

---

**Description**

Test functions for the current state of the development.

**Usage**

```
testTime()
testClassPseudoRandom(s, k, m)
```

**Arguments**

s	Seed.
k	Length of required output.
m	number of streams.

**Details**

testTime() determines the current time. testClassPseudoRandom(s, k, m) tests the functionality of multiple streams of RNGs.

**Value**

Depends on the function.

**Author(s)**

Marko Robnik-Sikonja, Petr Savicky

**See Also**

[CORElearn](#).

**Examples**

```
testTime()
```

---

calibrate

*Calibration of probabilities according to the given prior.*

---

**Description**

Given probability scores predictedProb as provided for example by a call to [predict.CoreModel](#) and using one of available methods given by methods the function calibrates predicted probabilities so that they match the actual probabilities of a binary class 1 provided by correctClass. The computed calibration can be applied to the scores returned by that model.

**Usage**

```
calibrate(correctClass, predictedProb, class1=1,  
          method = c("isoReg", "binIsoReg", "binning", "mdlMerge"),  
          weight=NULL, noBins=10, assumeProbabilities=FALSE)  
  
applyCalibration(predictedProb, calibration)
```

**Arguments**

correctClass	A vector of correct class labels for a binary classification problem.
predictedProb	A vector of predicted class 1 (probability) scores. In calibrate method it should be of the same length as correctClass.
class1	A class value (factor) or an index of the class value to be taken as a class to be calibrated.
method	One of isoReg, binIsoReg, binning, or mdlMerge. See details below.
weight	If specified, should be of the same length as correctClass and gives the weights for all the instances, otherwise a default weight of 1 for each instance is assumed.
noBins	The value of parameter depends on the parameter method and specifies desired or initial number of bins. See details below.
assumeProbabilities	If assumeProbabilities=TRUE the values in predictedProb are expected to be in [0,1] range i.e., probability estimates. assumeProbabilities=FALSE the algorithm can be used as ordinary (isotonic) regression
calibration	The list resulting from a call to calibration and subsequently applied to probability scores returned by the same model.

**Details**

Depending on the specified method one of the following calibration methods is executed.

- "isoReg" isotonic regression calibration based on pair-adjacent violators (PAV) algorithm.
- "binning" calibration into a pre-specified number of bands given by noBins parameter, trying to make bins of equal weight.
- "binIsoReg" first binning method is executed, following by a isotonic regression calibration.
- "mdlMerge" first intervals are merged by a MDL gain criterion into a prespecified number of intervals, following by the isotonic regression calibration.

If model="binning" the parameter noBins specifies the desired number of bins i.e., calibration bands; if model="binIsoReg" the parameter noBins specifies the number of initial bins that are formed by binning before isotonic regression is applied; if model="mdlMerge" the parameter noBins specifies the number of bins formed after first applying isotonic regression. The most similar bins are merged using MDL criterion.

**Value**

A function returns a list with two vector components of the same length:

interval	The boundaries of the intervals. Lower boundary 0 is not explicitly included but should be taken into account.
calProb	The calibrated probabilities for each corresponding interval.

**Author(s)**

Marko Robnik-Sikonja

## References

- I. Kononenko, M. Kukar: *Machine Learning and Data Mining: Introduction to Principles and Algorithms*. Horwood, 2007
- A. Niculescu-Mizil, R. Caruana: Predicting Good Probabilities With Supervised Learning. *Proceedings of the 22nd International Conference on Machine Learning (ICML'05)*, 2005

## See Also

[reliabilityPlot](#), [CORElearn](#), [predict.CoreModel](#).

## Examples

```
# generate data set separately for training the model,
# calibration of probabilities and testing
train <-classDataGen(noInst=200)
cal <-classDataGen(noInst=200)
test <- classDataGen(noInst=200)

# build random forests model with default parameters
modelRF <- CoreModel(class~., train, model="rf", maxThreads=1)

# prediction
predCal <- predict(modelRF, cal, rfPredictClass=FALSE)
predTest <- predict(modelRF, test, rfPredictClass=FALSE)
destroyModels(modelRF) # clean up, model not needed anymore

# calibrate for a chosen class1 and method
class1<-1
calibration <- calibrate(cal$class, predCal$prob[,class1], class1=class1,
                        method="isoReg",assumeProbabilities=TRUE)

# apply the calibration to the testing set
calibratedProbs <- applyCalibration(predTest$prob[,class1], calibration)
# the calibration of probabilities can be visualized with
# reliabilityPlot function
```

---

classDataGen

*Artificial data for testing classification algorithms*

---

## Description

The generator produces classification data with 2 classes, 7 discrete and 3 numeric attributes.

## Usage

```
classDataGen(noInst, t1=0.7, t2=0.9, t3=0.34, t4=0.32,
             p1=0.5, classNoise=0)
```

**Arguments**

noInst	Number of instances to generate.
t1, t2, t3	Parameters, which control the hardness of the discrete attributes.
t4	Parameter, which controls the hardness of the numeric attributes..
p1	Probability of class 1.
classNoise	Proportion of noise in the class variable for classification or virtual class variable for regression.

**Details**

Class probabilities are  $p_1$  and  $1 - p_1$ , respectively. The conditional distribution of attributes under each of the classes depends on parameters  $t_1, t_2, t_3, t_4$  from  $[0,1]$ . Attributes  $a_7$  and  $x_3$  are irrelevant for all values of parameters.

Examples of extreme settings of the parameters.

- Setting satisfying  $t_1 * t_2 = t_3$  implies no difference between the distributions of individual discrete attributes among the two classes. However, if  $t_1 < 1$ , then the joint distribution of them is different for the two classes.
- Setting  $t_1 = 1$  and  $t_2 = t_3$  implies no difference between the joint distribution of the discrete attributes among the two classes.
- Setting  $t_1 = 1, t_2 = 1, t_3 = 0$  implies disjoint supports of the distributions of  $a_1, a_2, a_4, a_5$ , so this allows exact classification.
- Setting  $t_4 = 1$  implies no difference between the distribution of  $x_1, x_2$  between the classes. Setting  $t_4 = 0$  allows correct classification with probability one only using  $x_1$  and  $x_2$ .

For class 1 the attributes have distributions

(a1, a2, a3)	$D_1(t_1, t_2)$
a4, a5, a6	$D_2(t_3)$
a7	irrelevant attribute, probabilities of {a,b,c,d} are (1/2, 1/6, 1/6, 1/6)
x1, x2, x3	independent normal variables with mean 0 and standard deviation 1, $t_4, 1, 1$
x4, x5	independent uniformly distributed variables on $[0,1]$

For class 2 the attributes have distributions

a1, a2, a3	$D_2(t_3)$
(a4, a5, a6)	$D_1(t_1, t_2)$
a7	irrelevant attribute, probabilities of {a,b,c,d} are (1/2, 1/6, 1/6, 1/6)
x1, x2, x3	independent normal variables with mean 0 and st. dev. $t_4, 1, 1$
x4, x5	independent uniformly distributed variables on $[0,1]$

$x_3$  is irrelevant for classification, since it has the same distribution under both classes.

Attributes in a bracket are mutually dependent. Otherwise, the attributes are conditionally independent for each of the two classes. This means that if we consider groups of the attributes such that the attributes in each of the two brackets form a group and each of the remaining attributes forms a

group with one element, then for each class, we have 7 groups, which are conditionally independent for the given class. Note that the splitting into groups differs for class 1 and 2.

Distribution  $D_1(t_1, t_2)$  consists of three dependent attributes. The distribution of individual attributes depends only on  $t_1 * t_2$ . For a given  $t_1 * t_2$ , the level of dependence decreases with  $t_1$  and increases with  $t_2$ . There are two extreme settings: Setting  $t_1 = 1$ ,  $t_2 = t_1 * t_2$  has the largest  $t_1$  and the smallest  $t_2$  and all three attributes are independent. Setting  $t_1 = t_1 * t_2$ ,  $t_2 = 1$  has the smallest  $t_1$  and the largest  $t_2$  and also the largest dependence between attributes.

Distribution  $D_2(t_3)$  is equal to  $D_1(1, t_3)$ , so it contains three independent attributes, whose distributions are the same as in  $D_1(t_1, t_2)$  for every setting satisfying  $t_1 * t_2 = t_3$ .

In other words, if  $t_3 = t_1 * t_2$ , then the distributions  $D_1(t_1, t_2)$  and  $D_2(t_3)$  have the same distributions of individual attributes and may differ only in the dependences. There are no in  $D_2(t_3)$  and there are some in  $D_1(t_1, t_2)$  if  $t_1 < 1$ .

#### *Hardness of the discrete part*

Setting  $t_1 = 1$  and  $t_2 = t_3$  implies no difference between the discrete attributes among the two classes.

Setting satisfying  $t_1 * t_2 = t_3$  implies no difference between the distributions of individual discrete attributes among the two classes. However, there may be a difference in dependences.

Setting  $t_1 = 1$ ,  $t_2 = 1$ ,  $t_3 = 0$  implies disjoint supports of the distributions of  $a_1$ ,  $a_2$ ,  $a_4$ ,  $a_5$ , so this allows exact classification.

#### *Hardness of the continuous part*

Depends monotonically on  $t_4$ . Setting  $t_4 = 1$  implies no difference between the classes. Setting  $t_4 = 0$  allows correct classification with probability one.

### **Value**

The method classDataGen returns a `data.frame` with `noInst` rows and 11 columns. Range of values of the attributes and class are

a1	0,1
a2	0,1
a3	a,b,c,d
a4	0,1
a5	0,1
a6	a,b,c,d
a7	a,b,c,d
x1	numeric
x2	numeric
x3	numeric
class	1,2

For detailed specification of attributes (columns) see details section below.

### **Author(s)**

Petr Savicky

**See Also**

[regDataGen](#), [ordDataGen](#), [CoreModel](#).

**Examples**

```
#prepare a classification data set
classData <-classDataGen(noInst=200)

# build random forests model with certain parameters
modelRF <- CoreModel(class~., classData, model="rf",
                     selectionEstimator="MDL", minNodeWeightRF=5,
                     rfNoTrees=100, maxThreads=1)
print(modelRF)
destroyModels(modelRF) # clean up
```

---

classPrototypes	<i>The typical instances of each class - class prototypes</i>
-----------------	---

---

**Description**

For each class the most typical instances are returned based on the highest predicted probability for each class.

**Usage**

```
classPrototypes(model, dataset, noPrototypes=10)
```

**Arguments**

model	a <a href="#">CoreModel</a> model.
dataset	a dataset from which to get prototypes.
noPrototypes	number of instances of each class to return

**Details**

The function uses `predict.CoreModel(model, dataset)` for prediction of the dataset with model. Based on the returned probabilities, it selects the `noPrototypes` instances with highest probabilities for each class to be typical representatives of that class, i.e., prototypes. The prototypes can be visualized by calling e.g., `plot(model, dataset, rfGraphType="prototypes", noPrototypes = 10)`.

**Value**

A list with the most typical `noPrototypes` instances is returned. The list has the following attributes.

prototypes	vector with indexes of the most typical instances
clustering	vector with class assignments for typical instances in vector instances
levels	the names of the class values.

**Author(s)**

John Adeyanju Alao (as a part of his BSc thesis) and Marko Robnik-Sikonja (thesis supervisor)

**References**

Leo Breiman: Random Forests. *Machine Learning Journal*, 45:5-32, 2001

**See Also**

[predict.CoreModel](#), [plot.CoreModel](#).

**Examples**

```
dataset <- iris
md <- CoreModel(Species ~ ., dataset, model="rf", rfNoTrees=30,maxThreads=1)
typical <- classPrototypes(md, dataset, 10)
destroyModels(md) # clean up
```

---

CORElearn-internal      *Internal structures of CORElearn C++ part*

---

**Description**

The package CORElearn is an R port of CORElearn data mining system. This document is a short description of the C++ part which can also serve as a standalone Linux or Windows data mining system, its organization and main classes and data structures.

**Details**

The C++ part is called from R functions collected in file `Rinterface.R`. The C++ functions called from R and providing interface to R are collected in `Rfront.cpp` and `Rconvert.cpp`. The front end for standalone version is in file `frontend.cpp`. For many parts of the code there are two variants, classification and regression one. Regression part usually has `Reg` somewhere in its name. The main classes are

- `marray`, `mmatrix` are templates for storing vectors and matrixes
- `dataStore` contains data storage and data manipulation methods, of which the most important are
  - `mmatrix<int> DiscData`, `DiscPredictData` contain values of discrete attributes and class for training and prediction (optional). In classification column 0 always stores class values.
  - `mmatrix<double> ContData`, `ContPredictData` contain values of numeric attribute and prediction values for training and prediction (optional). In regression column 0 always stores target values.
  - `marray<attribute> AttrDesc` with information about attributes' types, number of values, min, max, column index in `DiscData` or `ContData`, ...



- estimation, estimationReg evaluate attributes with different purposes: decision/regression tree splitting, binarization, discretization, constructive induction, feature selection, etc. Because of efficiency these classes store its own data in
  - `mMatrix<int> DiscValues` containing discrete attributes and class values,
  - `mMatrix<double> ContValues` containing numeric attribute and prediction values.
- Options stores and handles all the parameters of the system.
- `featureTree`, `regressionTree` build all the models, predict with them, and create output.

**Author(s)**

Marko Robnik-Sikonja

**See Also**

[CORElearn](#), [CoreModel](#), [predict.CoreModel](#), [modelEval](#), [attrEval](#), [ordEval](#), [plot.ordEval](#), [helpCore](#), [paramCoreIO](#), [infoCore](#), [versionCore](#).

---

CoreModel

*Build a classification or regression model*

---

**Description**

Builds a classification or regression model from the data and formula with given parameters. Classification models available are

- random forests, possibly with local weighing of basic models (parallel execution on several cores),
- decision tree with constructive induction in the inner nodes and/or models in the leaves,
- kNN and weighted kNN with Gaussian kernel,
- naive Bayesian classifier.

Regression models:

- regression trees with constructive induction in the inner nodes and/or models in the leaves,
- linear models with pruning techniques,
- locally weighted regression,
- kNN and weighted kNN with Gaussian kernel.

Function `cvCoreModel` applies cross-validation to estimate predictive performance of the model.

**Usage**

```
CoreModel(formula, data,
           model=c("rf", "rfNear", "tree", "knn", "knnKernel", "bayes", "regTree"),
           costMatrix=NULL, ...)
cvCoreModel(formula, data,
            model=c("rf", "rfNear", "tree", "knn", "knnKernel", "bayes", "regTree"),
            costMatrix=NULL, folds=10, stratified=TRUE, returnModel=TRUE, ...)
```

## Arguments

<code>formula</code>	Either a formula specifying the attributes to be evaluated and the target variable, or a name of target variable, or an index of target variable.
<code>data</code>	Data frame with training data.
<code>model</code>	The type of model to be learned.
<code>costMatrix</code>	Optional misclassification cost matrix used with certain models.
<code>folds</code>	An integer, specifying the number of folds to use in cross-validation of model.
<code>stratified</code>	A boolean specifying if cross-validation is to be stratified for classification problems, i.e. shall all folds have the same distribution of class values.
<code>returnModel</code>	If TRUE the function <code>cvCoreModel</code> estimates predictive performance using cross-validation and returns the model build on the whole data set. If <code>returnModel=FALSE</code> the function only evaluates the model using cross-validation.
<code>...</code>	Options for building the model. See <a href="#">helpCore</a> .

## Details

The parameter `formula` can be interpreted in three ways, where the formula interface is the most elegant one, but inefficient and inappropriate for large data sets. See also examples below. As formula one can specify:

**an object of class** `formula` used as a mechanism to select features (attributes) and prediction variable (class). Only simple terms can be used and interaction expressed in formula syntax are not supported. The simplest way is to specify just response variable: `class ~ .`. In this case all other attributes in the data set are evaluated. Note that formula interface is not appropriate for data sets with large number of variables.

**a character vector** specifying the name of target variable, all the other columns in data frame `data` are used as predictors.

**an integer** specifying the index of of target variable in data frame `data`, all the other columns are used as predictors.

Parameter **model** controls the type of the constructed model. There are several possibilities:

"rf" random forests classifier as defined by (Breiman, 2001) with some extensions,

"rfNear" random forests classifier with basic models weighted locally (Robnik-Sikonja, 2005),

"tree" decision tree with constructive induction in the inner nodes and/or models in the leaves,

"knn" k nearest neighbors classifier,

"knnKernel" weighted k nearest neighbors classifier with distance taken into account through Gaussian kernel,

"bayes" naive Bayesian classifier,

"regTree" regression trees with constructive induction in inner nodes and/or models in leaves controlled by `modelTypeReg` parameter. Models used in leaves of the regression tree can also be used as stand-alone regression models using option `minNodeWeightTree=Inf` (see examples below):

- linear models with pruning techniques

- locally weighted regression
- kNN and kNN with Gaussian kernel.

There are many additional parameters ... available which are used by different models. Their list and description is available by calling [helpCore](#). Evaluation of attributes is covered in function [attrEval](#).

The optional parameter **costMatrix** can provide nonuniform cost matrix for classification problems. For regression problem this parameter is ignored. The format of the matrix is `costMatrix(true class, predicted class)`. By default uniform costs are assumed, i.e., `costMatrix(i, i) = 0`, and `costMatrix(i, j) = 1`, for `i` not equal to `j`.

### Value

The created model is not returned as a R structure. It is stored internally in the package memory space and only its pointer (index) is returned. The maximum number of models that can be stored simultaneously is a parameter of the initialization function `initCore` and defaults to 16384. Models, which are not needed, may be deleted in order to free the memory using function `destroyModels`. By referencing the returned model, any of the stored models may be used for prediction with [predict.CoreModel](#). What the function actually returns is a list with components:

<code>modelID</code>	index of internally stored model,
<code>terms</code>	description of prediction variables and response,
<code>class.lev</code>	class values for classification problem, null for regression problem,
<code>model</code>	the type of model used, see parameter <code>model</code> ,
<code>formula</code>	the formula parameter passed.

The function `cvCoreModel` evaluates the model using cross-validation and function [modelEval](#) to return these additional components:

<code>avgs</code>	A vector with average values of each evaluation metric obtained from <a href="#">modelEval</a> .
<code>stds</code>	A vector with standard deviations of each evaluation metric from <a href="#">modelEval</a> .
<code>evalList</code>	A list, where each component is an evaluation metric from <a href="#">modelEval</a> . Each component contains results of cross-validated runs.

In case `returnModel=FALSE` the function only returns the above three components and keeps no model.

### Author(s)

Marko Robnik-Sikonja, Petr Savicky

### References

- Marko Robnik-Sikonja, Igor Kononenko: Theoretical and Empirical Analysis of ReliefF and RReliefF. *Machine Learning Journal*, 53:23-69, 2003
- Leo Breiman: Random Forests. *Machine Learning Journal*, 45:5-32, 2001
- Marko Robnik-Sikonja: Improving Random Forests. In J.-F. Boulicaut et al.(Eds): *ECML 2004, LNAI 3210*, Springer, Berlin, 2004, pp. 359-370

Marko Robnik-Sikonja: CORE - a system that predicts continuous variables. *Proceedings of ERK'97*, Portoroz, Slovenia, 1997

Marko Robnik-Sikonja, Igor Kononenko: Discretization of continuous attributes using ReliefF. *Proceedings of ERK'95*, B149-152, Ljubljana, 1995

Majority of these references are available from <http://lkm.fri.uni-lj.si/rmarko/papers/>

## See Also

[CORElearn](#), [predict.CoreModel](#), [modelEval](#), [attrEval](#), [helpCore](#), [paramCoreIO](#).

## Examples

```
# use iris data set
trainIdxs <- sample(x=nrow(iris), size=0.7*nrow(iris), replace=FALSE)
testIdxs <- c(1:nrow(iris))[-trainIdxs]

# build random forests model with certain parameters
# setting maxThreads to 0 or more than 1 forces
# utilization of several processor cores
modelRF <- CoreModel(Species ~ ., iris[trainIdxs,], model="rf",
                     selectionEstimator="MDL",minNodeWeightRF=5,
                     rfNoTrees=100, maxThreads=1)
print(modelRF) # simple visualization, test also others with function plot
# prediction on testing set
pred <- predict(modelRF, iris[testIdxs,], type="both")
mEval <- modelEval(modelRF, iris[["Species"]][testIdxs], pred$class, pred$prob)
print(mEval) # evaluation of the model
# visualization of individual predictions and the model
## Not run:
require(ExplainPrediction)
explainVis(modelRF, iris[trainIdxs,], iris[testIdxs,], method="EXPLAIN",
           visLevel="model", problemName="iris", fileType="none",
           classValue=1, displayColor="color")
# turn on the history in visualization window to see all instances
explainVis(modelRF, iris[trainIdxs,], iris[testIdxs,], method="EXPLAIN",
           visLevel="instance", problemName="iris", fileType="none",
           classValue=1, displayColor="color")

## End(Not run)
destroyModels(modelRF) # clean up

# build decision tree with naive Bayes in the leaves
# more appropriate for large data sets one can specify just the target variable
modelDT <- CoreModel("Species", iris, model="tree", modelType=4)
print(modelDT)
destroyModels(modelDT) # clean up

# build regression tree similar to CART
instReg <- regDataGen(200)
modelRT <- CoreModel(response~., instReg, model="regTree", modelTypeReg=1)
```

```

print(modelRT)
destroyModels(modelRT) # clean up

# build kNN kernel regressor by preventing tree splitting
modelKernel <- CoreModel(response~., instReg, model="regTree",
                        modelTypeReg=7, minNodeWeightTree=Inf)
print(modelKernel)
destroyModels(modelKernel) # clean up

## Not run:
# A more complex example
# Test accuracy of random forest predictor with 20 trees on iris data
# using 10-fold cross-validation.
ncases <- nrow(iris)
ind <- ceiling(10*(1:ncases)/ncases)
ind <- sample(ind,length(ind))
pred <- rep(NA,ncases)
fit <- NULL
for (i in unique(ind)) {
  # Delete the previous model, if there is one.
  fit <- CoreModel(Species ~ ., iris[ind!=i,], model="rf",
                  rfNoTrees=20, maxThreads=1)
  pred[ind==i] <- predict(fit, iris[ind==i,], type="class")
  if (!is.null(fit)) destroyModels(fit) # dispose model no longer needed
}
table(pred,iris$Species)

## End(Not run)
# a simpler way to estimate performance using cross-validation
model <- cvCoreModel(Species ~ ., iris, model="rf", rfNoTrees=20,
                    folds=10, stratified=TRUE, returnModel=TRUE,
                    maxThreads=1)

model$avgs

```

---

cvGen

*Cross-validation and stratified cross-validation*


---

## Description

Generate indices for cross-validation and stratified cross-validation

## Usage

```

cvGen(n, k)
cvGenStratified(classVal,k)
gatherFromList(lst)

```

**Arguments**

n	The number of instances in a data set.
k	The number of folds in cross-validation.
classVal	A vector of factors representing class values.
lst	A list of lists from which we collect results of the same components.

**Details**

The functions `cvGen` and `cvGenStratified` generate indices of instances from a data set which can be used in cross-validation. The function `cvGenStratified` generates the same distribution of class values in each fold. The function `gatherFromList` is an auxiliary function helping in collection of results, see the example below.

**Value**

The functions `cvGen` and `cvGenStratified` return a vector of indices indicating fold membership i.e. from 1:k. The function `codegatherFromList` returns a list with components containing elements of the same name.

**Author(s)**

Marko Robnik-Sikonja

**See Also**

[CORElearn](#).

**Examples**

```
data <- iris
folds <- 10
foldIdx <- cvGen(nrow(data), k=folds)
evalCore<-list()
for (j in 1:folds) {
  dTrain <- data[foldIdx!=j,]
  dTest <- data[foldIdx==j,]
  modelCore <- CoreModel(Species~., dTrain, model="rf")
  predCore <- predict(modelCore, dTest)
  evalCore[[j]] <- modelEval(modelCore, correctClass=dTest$Species,
    predictedClass=predCore$class, predictedProb=predCore$prob )
  destroyModels(modelCore)
}
results <- gatherFromList(evalCore)
sapply(results, mean)
```

---

destroyModels	<i>Destroy single model or all CORElearn models</i>
---------------	---

---

### Description

Destroys internal representation of a given model or all constructed models. As side effect the memory used by the model(s) is freed.

### Usage

```
destroyModels(model=NULL)
```

### Arguments

model	The model structure as returned by <a href="#">CoreModel</a> . The default value of NULL represents all generated models.
-------	---

### Details

The function destroys the model structure as returned by [CoreModel](#). Subsequent work with this model is no longer possible. If parameter `model=NULL` (default value) all generated models are destroyed and memory used by their internal representation is freed.

### Value

There is no return value.

### Author(s)

Marko Robnik-Sikonja, Petr Savicky

### See Also

[CORElearn](#), [CoreModel](#).

### Examples

```
# use iris data set

# build random forests model with certain parameters
model <- CoreModel(Species ~ ., iris, model="rf",
                  selectionEstimator="MDL", minNodeWeightRF=5,
                  rfNoTrees=100, maxThreads=1)

# prediction
pred <- predict(model, iris, rfPredictClass=FALSE)
# print(pred)

# destruction of model's internal representation
```

```
destroyModels(model)
```

---

 discretize

*Discretization of numeric attributes*


---

### Description

The method `discretize` returns discretization bounds for numeric attributes and two auxiliary functions. Discretization can be obtained with one of the three discretization methods: greedy search using given feature evaluation heuristics, equal width of intervals, or equal number of instances in each interval. The attributes and target variable are specified using formula interface, target variable name or index. Feature evaluation algorithms available for classification problems are various variants of Relief and ReliefF algorithms, gain ratio, gini-index, MDL, DKM, information gain, etc. For regression problems there are RREliefF, MSEofMean, MSEofModel, MAEofMode, etc.

### Usage

```
discretize(formula, data, method=c("greedy", "equalFrequency", "equalWidth"),
           estimator, discretizationLookahead=3, discretizationSample=0,
           maxBins=0, equalDiscBins=4, ...)
```

```
applyDiscretization(data, boundsList, noDecimalsInValueName=2)
```

```
intervalMidPoint(data, boundsList,
                 midpointMethod=c("equalFrequency", "equalWidth"))
```

### Arguments

<code>formula</code>	Either a formula specifying the attributes to be evaluated and the target variable, or a name of target variable, or an index of target variable.
<code>data</code>	Data frame with data.
<code>method</code>	Three discretization methods are available. With <code>method="greedy"</code> greedy search using given feature evaluation heuristics is selected, while <code>"equalFrequency"</code> and <code>"equalWidth"</code> select equal frequency (the same number of instances in each interval) and equal width discretization, respectively.
<code>estimator</code>	The name of the evaluation method.
<code>discretizationLookahead</code>	Discretization is performed with a greedy algorithm which adds a new boundary, until there is no improvement in evaluation function for <code>discretizationLookahead</code> number of times (0=try all possibilities). Candidate boundaries are chosen from a random sample of boundaries, whose size is <code>discretizationSample</code> .



discretizationSample	Maximal number of points to try discretization (0=all sensible). Binarization of multivalued discrete features with $k$ values is performed exhaustively, if $2^k - 1$ is at most discretizationSample. Otherwise binarization is done greedily starting from the best separation of a single value. For ReliefF-type measures, binarization of numeric features is performed with discretizationSample randomly chosen splits. For other measures, the split is searched exhaustively among all possible splits.
maxBins	The maximal number of discrete bins for numeric attributes used for greedy discretization (0=don't care). This shall be an integer vector of length equal to the number of numeric attributes or an integer which applies to all numeric attributes. The default value of 0 means that the number of bins will be determined greedily taking into account discretizationLookahead.
.	.
equalDiscBins	The number of bins used in equal frequency and equal width discretization. This shall be an integer vector of length equal to the number of numeric attributes or an integer which applies to all numeric attributes. The default value is 4.
...	Additional options used by specific evaluation methods as described in <a href="#">helpCore</a> .
boundsList	A list of numeric bounds which is applied to numeric attributes in data to produce discrete attributes of type factor. Numeric bounds can be obtained by calling discretize function.
.	.
noDecimalsInValueName	With how many decimal places will the numeric feature values be presented in description (i.e., levels) of feature values. The default value is 2, but will be increased if this is necessary to avoid the same description of feature values.
midPointMethod	Two methods to determine the middle points of discretization intervals are available. The "equalFrequency" method select the middle point so that each half-interval contains equal number of instances. The "equalWidth" methods sets middle point to be equally distant from the boundaries.

## Details

In method `discretize` the parameter `formula` can be interpreted in three ways, where the formula interface is the most elegant one, but inefficient and inappropriate for large data sets. See [CoreModel](#) for details.

The **estimator** parameter selects the evaluation heuristics. For classification problem it must be one of the names returned by `infoCore(what="attrEval")` and for regression problem it must be one of the names returned by `infoCore(what="attrEvalReg")`. For details see their description in [attrEval](#).

If the number of supplied vector in `maxBins` and `equalDiscBins` is shorter than the number of numeric attributes, the vector is coerced to the required length.

There are some additional parameters ... available which are used by specific evaluation heuristics. Their list and short description is available by calling [helpCore](#). See Section on attribute evaluation.

The function `applyDiscretization` takes the discretization bounds obtain with function `discretize` and transforms numeric features in a data set into discrete features.

The function `intervalMidPoint` takes discretization bounds provided by function `discretize` and returns middle points of discretization intervals for numeric attributes. The middle points are computed from the data; for lowest/highest interval the minimum/maximum of the values in the data for particular attribute are implicitly taken as an additional left/right boundary point.

### Value

The method `discretize` returns a list of discretization bounds for numeric attributes. One component of a list contains bounds for one attribute. If an attribute has all values equal, value NA is returned. If an attribute has all values equal to NA, it is skipped in the returned list.

The function `applyDiscretization` returns a data set where all numeric attributes are replaced with their discrete versions.

The function `intervalMidPoint` returns a list of vectors where each vector contains middle point of discretized intervals.

### Author(s)

Marko Robnik-Sikonja

### References

Marko Robnik-Sikonja, Igor Kononenko: Theoretical and Empirical Analysis of ReliefF and RReliefF. *Machine Learning Journal*, 53:23-69, 2003

Marko Robnik-Sikonja, Igor Kononenko: Discretization of continuous attributes using ReliefF. *Proceedings of ERK'95*, Portoroz, Slovenia, 1995.

Some of these references are available also from <http://lkm.fri.uni-lj.si/rmarko/papers/>

### See Also

[CORElearn](#), [CoreModel](#), [attrEval](#), [helpCore](#), [infoCore](#).

### Examples

```
# use iris data
# run method using estimator ReliefF with exponential rank distance
discBounds <- discretize(Species ~ ., iris, method="greedy",
                        estimator="ReliefFexpRank")

print(discBounds)
discreteIris <- applyDiscretization(iris, discBounds)
prototypePoints <- intervalMidPoint(iris, discBounds,
                                   midPointMethod="equalFrequency")

regData <- regDataGen(200)
discretize(response ~ ., regData, method="greedy", estimator="RReliefFequalK",
           maxBins=2)

# print all available estimators
```

```
#infoCore(what="attrEval")
#infoCore(what="attrEvalReg")
```

---

display.CoreModel      *Displaying decision and regression trees*

---

## Description

The method `display` prints the tree models returned by `CoreModel()` function. Depending of parameter `format` the output is prepared for either screen or in dot format.

## Usage

```
## S3 method for class 'CoreModel'
display(x, format=c("screen", "dot"))
```

## Arguments

<code>x</code>	The model structure as returned by <a href="#">CoreModel</a> .
<code>format</code>	The type of output, i.e., prepared for screen display or in dot language

## Details

The tree based models returned by function [CoreModel](#) are visualized. Only tree based models supported, including the trees which include other prediction models in their leaves. Tree based models available are decision trees (obtained by using parameter `model="tree"` in `CoreModel`), and regression trees (with `model="regTree"`).

Models in the leaves of decision trees can be set using parameter `modelType` in [CoreModel](#). At the moment naive Bayes and kNN are available, for details see [helpCore](#).

Models in the leaves of regression trees can be set using parameter `modelTypeReg` in [CoreModel](#). At the moment kNN, kernel regression, and several types of linear models are available, for details see [helpCore](#).

The output in dot language can be used with `graphViz` visualization software to create model visualization in various formats.

## Value

The method invisibly returns a printed character vector.

## Author(s)

Marko Robnik-Sikonja

## See Also

[CoreModel](#), [plot.CoreModel](#).

## Examples

```
# decision tree
dataset <- CO2
md <- CoreModel(Plant ~ ., dataset, model="tree")
display(md)
destroyModels(md) #clean up

# regression tree
dataset <- CO2
mdr <- CoreModel(uptake ~ ., dataset, model="regTree")
display(mdr, format="dot")
destroyModels(mdr) # clean up
```

---

getCoreModel	<i>Conversion of model to a list</i>
--------------	--------------------------------------

---

## Description

Function converts given model from internal structures in C++ to R's data structures.

## Usage

```
getCoreModel(model)
```

## Arguments

model            The model structure as returned by [CoreModel](#).

## Details

The function converts the model referenced by model from C++ internal structures to R's lists. Currently it is implemented only for random forests models.

## Value

For random forest a resulting list contains first all the information on the forest level, followed by the list of trees. For each tree the nodes are recursively nested with indication of node type (leaf or internal node) and than required information for that data type.

## Author(s)

Marko Robnik-Sikonja

## See Also

[CoreModel](#), [CORElearn](#).

**Examples**

```
# uses iris data set

# build random forests model with certain parameters,
# do not make too many and too large trees
modelRF <- CoreModel(Species ~ ., iris, model="rf",
                     selectionEstimator="MDL", minNodeWeightRF=50,
                     rfNoTrees=5, maxThreads=1)
print(modelRF)

# get the structure of the forest
forest <- getCoreModel(modelRF)
# forest

destroyModels(modelRF) # clean up
```

---

`getRFsizes`*Get sizes of the trees in RF*

---

**Description**

Get numerical characteristics of the trees in a RF model related to the size and depth.

**Usage**

```
getRFsizes(model, type=c("size", "sumdepth"))
```

**Arguments**

<code>model</code>	The model structure as returned by <a href="#">CoreModel</a> .
<code>type</code>	The required characteristics.

**Details**

Size is the number of leaves. The sum of depths means the sum of the depth of all leaves.

**Value**

Numerical vector of the length equal to the number of trees in RF.

**Author(s)**

Petr Savicky

**See Also**

[CoreModel](#), [CORElearn](#).

## Examples

```
# uses iris data set

# build random forests model with certain parameters,
# do not make too many and too large trees
modelRF <- CoreModel(Species ~ ., iris, model="rf",
                    selectionEstimator="MDL", minNodeWeightRF=50,
                    rfNoTrees=50, maxThreads=1)

getRFsizes(modelRF)

destroyModels(modelRF) # clean up
```

---

getRpartModel

*Conversion of a CoreModel tree into a rpart.object*

---

## Description

The function converts a given CoreModel model (decision or regression tree) into a rpart.object prepared for visualization with plot function.

## Usage

```
getRpartModel(model, dataset)
```

## Arguments

model	A tree model produced by <a href="#">CoreModel</a>
dataset	A data set which was used in learning of the model.

## Details

The conversion creates rpart.object and copies CORElearn internal structures contained in memory controlled by dynamic link library written in C++.

An alternative visualization is accessible via function [display](#), which outputs tree structure formatted for screen or in dot format.

## Value

Function returns a [rpart.object](#).

## Author(s)

Initial version by John Adeyanju Alao, improvements by Marko Robnik-Sikonja.

## See Also

[CoreModel](#), [plot.CoreModel](#), [rpart.object](#), [display](#)

**Examples**

```
# plot a decision tree directly
dataset <- CO2
md<-CoreModel(Plant ~ ., dataset, model="tree")
plot(md, dataset)

# or indirectly
rpm <- getRpartModel(md, dataset)
# set angle to tan(0.5)=45 (degrees) and length of branches at least 5
plot(rpm, branch=0.5, minbranch=5, compress=TRUE)
# pretty=0 prints full names of attributes,
# numbers to 3 decimals, try to make a dendrogram more compact
text(rpm, pretty=0, digits=3)
destroyModels(md) # clean up

# an alternative is to use fancier rpart.plot package
# rpart.plot(rpm) # rpart.plot has many parameters controlling the output
# but it cannot plot models in tree leaves
```

helpCore

*Description of parameters.***Description**

The behavior of CORElearn is controlled by several parameters. This is a short overview.

**Details**

There are many different parameters available. Some are general and can be used in many learning, or feature evaluation algorithms. All the values actually used by the classifier / regressor can be written to file (or read from it) using [paramCoreIO](#). The parameters for the methods are split into several groups and documented below.

**Attribute/feature evaluation**

The parameters in this group may be used inside model construction via [CoreModel](#) and feature evaluation in [attrEval](#). See [attrEval](#) for description of relevant evaluation methods.

Parameters `attrEvaluationInstances`, `binaryEvaluation`, `binarySplitNumericAttributes` are applicable to all attribute evaluation methods. In models which need feature evaluation (e.g., trees, random forests) they affect the selection of splits in the nodes. Other parameters may be used only in context sensitive measures, i.e., ReliefF in classification and RReliefF in regression and their variants.

**binaryEvaluation** type: logical, default value: FALSE

Shall we treat all attributes as binary and binarize them before evaluation if necessary. If TRUE, then for all multivalued discrete and all numeric features a search for the best binarization is performed. The evaluation of the best binarization found is reported. If FALSE, then multivalued discrete features are evaluated "as is" with multivalued versions of estimators. With

ReliefF-type measures, numeric features are also evaluated "as is". For evaluation of numeric features with other (non-ReliefF-type) measures, they are first binarized or discretized. The choice between binarization and discretization is controlled by `binaryEvaluateNumericAttributes`. Due to performance reasons it is recommended that `binaryEvaluation=FALSE` is used. See also `discretizationSample`.

**binaryEvaluateNumericAttributes** type: logical, default value: TRUE

ReliefF like measures can evaluate numeric attributes intrinsically, others have to discretize or binarize them before evaluation; for those measures this parameter selects binarization (default) or discretization (computationally more demanding).

**multiclassEvaluation** type: integer, default value: 1, value range: 1, 4

multi-class extension for two-class-only evaluation measures (1-average of all-pairs, 2-best of all-pairs, 3-average of one-against-all, 4-best of one-against-all).

**attrEvaluationInstances** type: integer, default value: 0, value range: 0, Inf

number of instances for attribute evaluation (0=all available).

**minNodeWeightEst** type: numeric, default value: 2, value range: 0, Inf

minimal number of instances (weight) in resulting split to take it in consideration.

**ReliefIterations** type: integer, default value: 0, value range: -2, Inf

number of iterations for all variants of Relief (0=DataSize, -1=ln(DataSize) -2=sqrt(DataSize)).

**numAttrProportionEqual** type: numeric, default value: 0.04, value range: 0, 1

used in ramp function, proportion of numerical attribute's range to consider two values equal.

**numAttrProportionDifferent** type: numeric, default value: 0.1, value range: 0, 1

used in ramp function, proportion of numerical attribute's range to consider two values different.

**kNearestEqual** type: integer, default value: 10, value range: 0, Inf

number of neighbors to consider in equal k-nearest attribute evaluation.

**kNearestExpRank** type: integer, default value: 70, value range: 0, Inf

number of neighbors to consider in exponential rank distance attribute evaluation.

**quotientExpRankDistance** type: numeric, default value: 20, value range: 0, Inf

quotient in exponential rank distance attribute evaluation.

### Decision/regression tree construction

There are several parameters controlling a construction of the tree model. Some are described here, but also attribute evaluation, stop building, model, constructive induction, discretization, and pruning options described in this document are applicable. Splits in trees are always binary, however, the option `binaryEvaluation` has influence on the feature selection for the split. Namely, selecting the best feature for the split is done with the given value of `binaryEvaluation`. If `binaryEvaluation=FALSE`, the features are first evaluated and the best one is finally binarized. If `binaryEvaluation=TRUE`, the features are binarized before selection. In this case, a search for the best binarization for all considered features is performed and the best binarizations found are used for splits. The latter option is computationally more intensive, but typically does not produce better trees.

**selectionEstimator** type: character, default value: "MDL", possible values: all from `attrEval`, section classification

estimator for selection of attributes and binarization in classification.



**selectionEstimatorReg** type: character, default value: "RReliefFexpRank", possible values: all from `attrEval`, section regression estimator for selection of attributes and binarization in regression.

**minReliefEstimate** type: numeric, default value: 0, value range: -1, 1 for all variants of Relief attribute estimator: the minimal evaluation of attribute to consider the attribute useful in further processing.

**minInstanceWeight** type: numeric, default value: 0.05, value range: 0, 1 minimal weight of an instance to use it further in splitting.

### Stop tree building

During tree construction the node is recursively split, until certain condition is fulfilled.

**minNodeWeightTree** type: numeric, default value: 5, value range: 0, Inf minimal number of instances (weight) of a leaf in the decision or regression tree model.

**minNodeWeightRF** type: numeric, default value: 2, value range: 0, Inf minimal number of instances (weight) of a leaf in the random forest tree.

**relMinNodeWeight** type: numeric, default value: 0, value range: 0, 1 minimal proportion of training instances in a tree node to split it further.

**majorClassProportion** type: numeric, default value: 1, value range: 0, 1 proportion of majority class in a classification tree node to stop splitting it.

**rootStdDevProportion** type: numeric, default value: 0, value range: 0, 1 proportion of root's standard deviation in a regression tree node to stop splitting it.

### Models in the tree leaves

In leaves of the tree model there can be various prediction models controlling prediction. For example instead of classification with majority of class values one can use naive Bayes in classification, or a linear model in regression, thereby expanding expressive power of the tree model.

**modelType** type: integer, default value: 1, value range: 1, 4 type of models used in classification tree leaves (1=majority class, 2=k-nearest neighbors, 3=k-nearest neighbors with kernel, 4=naive Bayes).

**modelTypeReg** type: integer, default value: 5, value range: 1, 8 type of models used in regression tree leaves (1=mean predicted value, 2=median predicted value, 3=linear by MSE, 4=linear by MDL, 5=linear reduced as in M5, 6=kNN, 7=Gaussian kernel regression, 8=locally weighted linear regression).

**kInNN** type: integer, default value: 10, value range: 0, Inf number of neighbors in k-nearest neighbors models (0=all).

**nnKernelWidth** type: numeric, default value: 2, value range: 0, Inf kernel width in k-nearest neighbors models.

**bayesDiscretization** type: integer, default value: 2, value range: 1, 3 type of discretization for naive Bayesian models (1=greedy with selection estimator, 2=equal frequency, 3=equal width).

**discretizationIntervals** type: integer, default value: 4, value range: 1, Inf number of intervals in equal frequency or equal width discretizations.

### Constructive induction aka. feature construction

The expressive power of tree models can be increased by incorporating additional types of splits. Operator based constructive induction is implemented in both classification and regression. The best construct is searched with beam search. At each step new constructs are evaluated with selected feature evaluation measure. With different types of operators one can control expressions in the interior tree nodes.

**constructionMode** type: integer, default value: 15, value range: 1, 15  
sum of constructive operators (1=single attributes, 2=conjunction, 4=addition, 8=multiplication); all=1+2+4+8=15

**constructionDepth** type: integer, default value: 0, value range: 0, Inf  
maximal depth of the tree for constructive induction (0=do not do construction, 1=only at root, ...).

**noCachedInNode** type: integer, default value: 5, value range: 0, Inf  
number of cached attributes in each node where construction was performed.

**constructionEstimator** type: character, default value: "MDL", possible values: all from [attrEval](#), section classification  
estimator for constructive induction in classification.

**constructionEstimatorReg** type: character, default value: "RReliefFexpRank", possible values: all from [attrEval](#), section regression  
estimator for constructive induction in regression.

**beamSize** type: integer, default value: 20, value range: 1, Inf  
size of the beam in search for best feature in constructive induction.

**maxConstructSize** type: integer, default value: 3, value range: 1, Inf  
maximal size of constructs in constructive induction.

### Attribute discretization and binarization

Some algorithms cannot deal with numeric attributes directly, so we have to discretize them. Also the tree models use binary splits in nodes. The discretization algorithm evaluates split candidates and forms intervals of values. Note that setting `discretizationSample=1` will force random selection of splitting point, which will speed-up the algorithm and may be perfectly acceptable for random forest ensembles.

CORElearn builds binary trees so multivalued discrete attributes have to be binarized i.e., values have to be split into two a subset, one going left and the other going right in a node. The method used depends on the parameters and the number of attribute values. Possible methods are exhaustive (if the number of attribute values is less or equal `maxValues4Exhaustive`), greedy ((if the number of attribute values is less or equal `maxValues4Greedy`) and random ((if the number of attribute values is more than `maxValues4Exhaustive`). Setting `maxValues4Greedy=2` will always randomly select splitting point.

**discretizationLookahead** type: integer, default value: 3, value range: 0, Inf  
Discretization is performed with a greedy algorithm which adds a new boundary, until there is no improvement in evaluation function for `discretizationLookahead` number of times (0=try all possibilities). Candidate boundaries are chosen from a random sample of boundaries, whose size is `discretizationSample`.

- discretizationSample** type: integer, default value: 50, value range: 0, Inf  
Maximal number of points to try discretization (0=all sensible). For ReliefF-type measures, binarization of numeric features is performed with discretizationSample randomly chosen splits. For other measures, the split is searched among all possible splits.
- maxValues4Exhaustive** type: integer, default value: 7, value range: 2, Inf  
Maximal number of values of a discrete attribute to try finding split exhaustively. If the attribute has more values the split will be searched greedily or selected randomly based on the value of parameter maxValues4Greedy.
- maxValues4Greedy** type: integer, default value: 30, value range: 2, Inf  
Maximal number of values of a discrete attribute to try finding split greedily. If the attribute has more values the split will be selected randomly. Setting this parameter to 2 will force random but balanced selection of splits which may be acceptable for random forest ensembles and will greatly speed-up tree construction.

### Tree pruning

After the tree is constructed, to reduce noise it is beneficial to prune it.

- selectedPruner** type: integer, default value: 1, value range: 0, 1  
decision tree pruning method used (0=none, 1=with m-estimate).
- selectedPrunerReg** type: integer, default value: 2, value range: 0, 4  
regression tree pruning method used (0=none, 1=MDL, 2=with m-estimate, 3=as in M5, 4=error complexity as in CART (fixed alpha)).
- mdlModelPrecision** type: numeric, default value: 0.1, value range: 0, Inf  
precision of model coefficients in MDL tree pruning.
- mdlErrorPrecision** type: numeric, default value: 0.01, value range: 0, Inf  
precision of errors in MDL tree pruning.
- mEstPruning** type: numeric, default value: 2, value range: 0, Inf  
m-estimate for pruning with m-estimate.
- alphaErrorComplexity** type: numeric, default value: 0, value range: 0, Inf  
alpha for error complexity pruning.

### Prediction

For some models (decision trees, random forests, naive Bayes, and regression trees) one can smooth the output predictions. In classification models output probabilities are smoothed and in case of regression prediction value is smoothed.

- smoothingType** type: integer, default value: 0, value range: 0, 4  
default value 0 means no smoothing (in case classification one gets relative frequencies), value 1 stands for additive smoothing, 2 is pure Laplace's smoothing, 3 is m-estimate smoothing, and 4 means Zadrozny-Elkan type of m-estimate smoothing where smoothingValue is interpreted as  $m \cdot p_c$  and  $p_c$  is the prior probability of the least probable class value; for regression smoothingType has no effect, as the smoothing is controlled solely by smoothingValue.
- smoothingValue** type: numeric, default value: 0, value range: 0, Inf  
additional parameter for some sorts of smoothing; in classification it is needed for additive, m-estimate, and Zadrozny-Elkan type of smoothing; in case of regression trees 0 means no smoothing and values larger than 0 change prediction value towards the prediction of the models in ascendant nodes.

### Random forests

Random forest is quite complex model, whose construction one can control with several parameters. Momentarily only classification version of the algorithm is implemented. Besides parameters in this section one can apply majority of parameters for control of decision trees (except constructive induction and tree pruning).

**rfNoTrees** type: integer, default value: 100, value range: 1, Inf  
number of trees in the random forest.

**rfNoSelAttr** type: integer, default value: 0, value range: -2, Inf  
number of randomly selected attributes in the node (0=sqrt(numOfAttr), -1=log2(numOfAttr)+1, -2=all).

**rfMultipleEst** type: logical, default value: FALSE  
use multiple attribute estimators in the forest? If TRUE the algorithm uses some preselected attribute evaluation measures on different trees.

**rfkNearestEqual** type: integer, default value: 30, value range: 0, Inf  
number of nearest instances for weighted random forest classification (0=no weighing).

**rfPropWeightedTrees** type: numeric, default value: 0, value range: 0, 1  
Proportion of trees where attribute probabilities are weighted with their quality. As attribute weighting might reduce the variance between the models, the default value switches the weighing off.

**rfPredictClass** type: logical, default value: FALSE  
shall individual trees predict with majority class (otherwise with class distribution).

### General tree ensembles

In the same manner as random forests more general tree ensembles can be constructed. Additional options control sampling, tree size and regularization.

**rfSampleProp** type: numeric, default value: 0, value range: 0, 1  
proportion of the training set to be used in learning (0=bootstrap replication).

**rfNoTerminals** type: integer, default value: 0, value range: 0, Inf  
maximal number of leaves in each tree (0=build the whole tree).

**rfRegType** type: integer, default value: 2, value range: 0, 2  
type of regularization (0=no regularization, 1=global regularization, 2=local regularization).

**rfRegLambda** type: numeric, default value: 0, value range: 0, Inf  
regularization parameter lambda (0=no regularization).

### Read data directly from files

In case of very large data sets it is useful to bypass R and read data directly from files as the standalone learning system CORElearn does. Supported file formats are C4.5, M5, and native format of CORElearn. See documentation at <http://lkm.fri.uni-lj.si/rmarko/software/>.

**domainName** type: character,  
name of a problem to read from files with suffixes .dsc, .dat, .names, .data, .cm, and .costs

**dataDirectory** type: character,  
folder where data files are stored.

**NAstring** type: character, default value: "?"  
character string which represents missing and NA values in the data files.

### Miscellaneous

**maxThreads** type: integer, default value: 0, value range: 0, Inf  
maximal number of active threads (0=allow OpenMP to set its defaults).  
As side effect, this parameter changes the number of active threads in all subsequent execution (till maxThreads is set again).

### Author(s)

Marko Robnik-Sikonja, Petr Savicky

### References

B. Zadrozny, C. Elkan. Learning and making decisions when costs and probabilities are both unknown. In Proceedings of the Seventh International Conference on Knowledge Discovery and Data Mining, 2001.

### See Also

[CORElearn](#), [CoreModel](#), [predict.CoreModel](#), [attrEval](#), [ordEval](#), [paramCoreIO](#).

---

infoCore

*Description of certain CORElearn parameters*

---

### Description

Depending on parameter what the function prints some information on CORElearn, for example codes of available classification (or regression) attribute evaluation heuristics. For more complete description of the parameters see [helpCore](#).

### Usage

```
infoCore(what=c("attrEval", "attrEvalReg"))
```

### Arguments

what                   Selects the info to be printed.

### Details

Depending on the parameter what the function some information on CORElearn.

**"attrEval"** Prints codes of all available classification attribute evaluation heuristics. These codes can be used as parameters for attribute evaluation methods in learning. It is internally used for validation of parameters. For more complete information see [attrEval](#).

**"attrEvalReg"** prints codes of all available regression attribute evaluation heuristics. These codes can be used as parameters for attribute evaluation methods in learning. It is internally used for validation of parameters. For more complete information see [attrEval](#).

### Value

For `what="attrEval"` or `"attrEvalReg"` function returns vector of codes for all implemented classification or regression attribute evaluation heuristics, respectively.

### Author(s)

Marko Robnik-Sikonja

### See Also

[attrEval](#), [helpCore](#), [CoreModel](#).

### Examples

```
estClass <- infoCore(what="attrEval")
print(estClass)
infoCore(what="attrEvalReg")
```

---

modelEval

*Statistical evaluation of predictions*

---

### Description

Using predictions of given model produced by [predict.CoreModel](#) and correct labels, computes some statistics evaluating the quality of the model.

### Usage

```
modelEval(model=NULL, correctClass, predictedClass,
           predictedProb=NULL, costMatrix=NULL,
           priorClProb = NULL, avgTrainPrediction = NULL, beta = 1)
```

### Arguments

<code>model</code>	The model structure as returned by <a href="#">CoreModel</a> , or NULL if some other predictions are evaluated.
<code>correctClass</code>	A vector of correct class labels for classification problem and function values for regression problem.
<code>predictedClass</code>	A vector of predicted class labels for classification problem and function values for regression problem.
<code>predictedProb</code>	An optional matrix of predicted class probabilities for classification.

costMatrix	Optional cost matrix can provide nonuniform costs for classification problems.
priorClProb	If model=NULL a vector of prior class probabilities shall be provided in case of classification.
avgTrainPrediction	If model=NULL mean of prediction values on training set shall be provided in case of regression.
beta	For two class problems beta controls the relative importance of precision and recall in F-measure.

## Details

The function uses the model structure as returned by `CoreModel`, `predictedClass` and `predictedProb` returned by `predict.CoreModel`. Predicted values are compared with true values and some statistics are computed measuring the quality of predictions. In classification only one of the `predictedClass` and `predictedProb` can be NULL (one of them is computed from the other under assumption that class label is assigned to the most probable class). Some of the returned statistics are defined only for two class problems, for which the confusion matrix specifying the number of instances of true/predicted class is defined as follows,

true/predicted class	positive	negative
positive	true positive (TP)	false negative (FN)
negative	false positive (FP)	true negative (TN)

Optional cost matrix can provide nonuniform costs for classification problems. For regression problem this parameter is ignored. The costs can be different from the ones used for building the model in `CoreModel` and prediction with the model in `predict.CoreModel`. If no costs are supplied, uniform costs are assumed. The format of the matrix is `costMatrix(true_class, predicted_class)`. By default a uniform costs are assumed, i.e., `costMatrix(i, i) = 0`, and `costMatrix(i, j) = 1`, for `i` not equal to `j`. See the example below.

If a non-CORElearn model is evaluated, one should set `model=NULL`, and a vector of prior of class probabilities `priorClProb` shall be provided in case of classification, and in case of regression `avgTrainPrediction` shall be the mean of prediction values (estimated on a e.g., training set).

## Value

For classification problem function returns list with the components

`accuracy` classification accuracy, for two class problems this would equal

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FN} + \text{FP} + \text{TN}}$$

`averageCost` average classification cost

`informationScore`

information score statistics measuring information contents in the predicted probabilities

`AUC` Area under the ROC curve

`predictionMatrix`

matrix of miss-classifications also confusion matrix

sensitivity sensitivity for two class problems (also called accuracy of the positive class, i.e., acc+, or true positive rate),

$$rmsensitivity = \frac{TP}{TP + FN}$$

specificity specificity for two class problems (also called accuracy of the negative class, i.e., acc-, or true negative rate),

$$specificity = \frac{TN}{TN + FP}$$

brierScore Brier score of predicted probabilities (the original Brier's definition which scores all the classes not only the correct one)

kappa Cohen's kappa statistics measuring randomness of the predictions; for perfect predictions kappa=1, for completely random predictions kappa=0

precision precision for two class problems

$$precision = \frac{TP}{TP + FP}$$

recall recall for two class problems (the same as sensitivity)

F-measure F-measure giving a weighted score of precision and recall for two class problems

$$F = \frac{(1 + \beta^2) \cdot recall \cdot precision}{\beta^2 \cdot recall + precision}$$

G-mean geometric mean of positive and negative accuracy,

$$G = \sqrt{sensitivity \cdot specificity}$$

KS Kolmogorov-Smirnov statistics defined for binary classification problems, reports the distance between the probability distributions of positive class for positive and negative instances, see (Hand, 2005), value 0 means no separation, and value 1 means perfect separation,

$$KS = \max_t |TPR(t) - FPR(t)|$$

see definitions of TPR and FPR below

TPR true positive rate  $TPR = \frac{TP}{TP+FN}$  at maximal value of KS statistics

FPR false positive rate  $FPR = \frac{FP}{FP+TN}$  at maximal value of KS statistics

For regression problem the returned list has components

MSE square root of Mean Squared Error

RMSE Relative Mean Squared Error

MAE Mean Absolute Error

RMAE Relative Mean Absolute Error



**Author(s)**

Marko Robnik-Sikonja

**References**

Igor Kononenko, Matjaz Kukar: *Machine Learning and Data Mining: Introduction to Principles and Algorithms*. Horwood, 2007

David J.Hand: Good practice in retail credit scorecard assesment. *Journal of Operational Research Society*, 56:1109-1117, 2005)

**See Also**

[CORElearn](#), [CoreModel](#), [predict.CoreModel](#).

**Examples**

```
# use iris data

# build random forests model with certain parameters
model <- CoreModel(Species ~ ., iris, model="rf",
                  selectionEstimator="MDL",minNodeWeightRF=5,
                  rfNoTrees=100, maxThreads=1)

# prediction with node distribution
pred <- predict(model, iris, rfPredictClass=FALSE)

# Model evaluation
mEval <- modelEval(model, iris[["Species"]], pred$class, pred$prob)
print(mEval)

# use nonuniform cost matrix
noClasses <- length(levels(iris[["Species"]]))
costMatrix <- 1 - diag(noClasses)
costMatrix[3,1] <- costMatrix[3,2] <- 5 # assume class 3 is more valuable
mEvalCost <- modelEval(model, iris[["Species"]], pred$class, pred$prob,
                      costMatrix=costMatrix)
print(mEvalCost)

destroyModels(model) # clean up
```

---

noEqualRows

*Number of equal rows in two data sets*


---

**Description**

Counts number of equal rows in two data sets. The two data sets shall have equal number of columns.

**Usage**

```
noEqualRows(data1, data2, tolerance=1e-5, countOnce=TRUE)
```

**Arguments**

data1	The first data set.
data2	The second data set.
tolerance	Tolerated difference between two rows.
countOnce	Shall each equal row in data1 be counted just once, or number of rows it is equal to in data2.

**Details**

Rows are compared using column-wise comparisons. The sum of differences up to a given tolerance are tolerated.

**Value**

Integer value giving the count of equal instances.

**Author(s)**

Marko Robnik-Sikonja

**See Also**

[CORElearn](#).

**Examples**

```
# uses two randomly generated data sets
set.seed(12345)
d1 <- classDataGen(100)
d2 <- classDataGen(100)
noEqualRows(d1, d2, tolerance=1e-4)
```

---

ordDataGen

*Artificial data for testing ordEval algorithms*

---

**Description**

The generator produces ordinal data simulating different profiles of attributes: basic, performance, excitement and irrelevant.

**Usage**

```
ordDataGen(noInst, classNoise=0)
```

**Arguments**

noInst	Number of instances to generate.
classNoise	Proportion of randomly determined values in the class variable.

**Details**

Problem is described by six important and two irrelevant features. The important features correspond to different feature types from the marketing theory: two basic features ( $B_{weak}$  and  $B_{strong}$ ), two performance features ( $P_{weak}$  and  $P_{strong}$ ), two excitement features ( $E_{weak}$  and  $E_{strong}$ ), and two irrelevant features ( $I_{uniform}$  and  $I_{normal}$ ). The values of all features are randomly generated integer values from 1 to 5, indicating for example score assigned to each of the features by the survey's respondent. The dependent variable for each instance (class) is the sum of its features' effects, which we scale to the uniform distribution of integers 1-5, indicating, for example, an overall score assigned by the respondent.

$$C = b_w(B_{weak}) + b_s(B_{strong}) + p_w(P_{weak}) + p_s(P_{strong}) + e_w(E_{weak}) + e_s(E_{strong})$$

**Value**

The method returns a `data.frame` with noInst rows and 9 columns. Range of values of the attributes and class are integers in [1,5]

**Author(s)**

Marko Robnik-Sikonja

**See Also**

[classDataGen](#), [regDataGen](#), [ordEval](#),

**Examples**

```
#prepare a data set
dat <- ordDataGen(200)

# evaluate ordered features with ordEval
est <- ordEval(class ~ ., dat, ordEvalNoRandomNormalizers=100)
# print(est)
plot(est)
```

---

ordEval

*Evaluation of ordered attributes*


---

**Description**

The method evaluates the quality of ordered attributes specified by the formula with ordEval algorithm.

**Usage**

```
ordEval(formula, data, file=NULL, rndFile=NULL,
        variant=c("allNear", "attrDist1", "classDist1"), ...)
```

**Arguments**

formula	Either a formula specifying the attributes to be evaluated and the target variable, or a name of target variable, or an index of target variable.
data	Data frame with evaluation data.
file	Name of file where evaluation results will be written to.
rndFile	Name of file where evaluation of random normalizing attributes will be written to.
variant	Name of the variant of ordEval algorithm. Can be any of "allNear", "attrDist1", or "classDist1".
...	Other options specific to ordEval or common to other context-sensitive evaluation methods (e.g., ReliefF).

**Details**

The parameter `formula` can be interpreted in three ways, where the formula interface is the most elegant one, but inefficient and inappropriate for large data sets. See also examples below. As formula one can specify:

**an object of class** `formula` used as a mechanism to select features (attributes) and prediction variable (class). Only simple terms can be used and interaction expressed in formula syntax are not supported. The simplest way is to specify just response variable: `class ~ .`. In this case all other attributes in the data set are evaluated. Note that formula interface is not appropriate for data sets with large number of variables.

**a character vector** specifying the name of target variable, all the other columns in data frame `data` are used as predictors.

**an integer** specifying the index of of target variable in data frame `data`, all the other columns are used as predictors.

In the data frame `data` take care to supply the ordinal data as factors and to provide equal levels for them (this is not necessary what one gets with [read.table](#)). See example below.

The output can be optionally written to files `file` and `rndFile`, in a format used by visualization methods in [plotOrdEval](#).

The variant of the algorithm actually used is controlled with `variant` parameter which can have values "allNear", "attrDist1", and "classDist1". The default value is "allNear" which takes all nearest neighbors into account in evaluation of attributes. Variant "attrDist1" takes only neighbors with attribute value at most 1 different from current case into account (for each attribute separately). This makes sense when we want to see the thresholds of reinforcement, and therefore observe just small change up or down (it makes sense to combine this with `equalUpDown=TRUE` in [plot.ordEval](#) function). The "classDist1" variant takes only neighbors with class value at most 1 different from current case into account. This makes sense if we want to observe strictly small changes in upward/downward reinforcement and has little effect in practical applications.

There are some additional parameters (note ...) some of which are common with other context-sensitive evaluation methods (e.g., ReliefF). Their list of common parameters is available in [helpCore](#) (see subsection on attribute evaluation therein). The parameters specific to [ordEval](#) are:

**ordEvalNoRandomNormalizers** type: integer, default value: 0, value range: 0, Inf, number of randomly shuffled attributes for normalization of each attribute (0=no normalization). This parameter should be set to a reasonably high value (e.g., 200) in order to produce reliable confidence intervals with [plot.ordEval](#). The parameters [ordEvalBootstrapNormalize](#) and [ordEvalNormalizingPercentile](#) only make sense if this parameter is larger than 0.

**ordEvalBootstrapNormalize** type: logical, default value: FALSE are features used for normalization constructed with bootstrap sampling or random permutation.

**ordEvalNormalizingPercentile** type: numeric, default value: 0.025, value range: 0, 0.5 percentile defines the length of confidence interval obtained with random normalization. Percentile  $t$  forms interval by taking the  $n \cdot t$  and  $n(1 - t)$  random evaluation as the confidence interval boundaries, thereby forming  $100(1 - 2t)\%$  confidence interval ( $t=0.025$  gives 95% confidence interval). The value  $n$  is set by [ordEvalNoRandomNormalizers](#) parameter.

**attrWeights** type: character, a character vector representing a list of attribute weights in the [ordEval](#) distance measure.

Evaluation of attributes without specifics of ordered attributes is covered in function [attrEval](#).

## Value

The method returns a list with following components:

<code>reinfPosAV</code>	a matrix of positive reinforcement for attributes' values,
<code>reinfNegAV</code>	a matrix of negative reinforcement for attributes' values,
<code>anchorAV</code>	a matrix of anchoring for attributes' values,
<code>noAV</code>	a matrix containing count for each value of each attribute,
<code>reinfPosAttr</code>	a vector of positive reinforcement for attributes,
<code>reinfNegAttr</code>	a matrix of negative reinforcement for attributes,
<code>anchorAttr</code>	a matrix of anchoring for attributes,
<code>noVattr</code>	a vector containing count of valid values of each attribute,
<code>rndReinfPosAV</code>	a three dimensional array of statistics for random normalizing attributes' positive reinforcement for attributes' values,
<code>rndReinfNegAV</code>	a three dimensional array of statistics for random normalizing attributes' negative reinforcement for attributes' values,
<code>rndAnchorAV</code>	a three dimensional array of statistics for random normalizing attributes' anchoring for attributes' values,
<code>rndReinfPosAttr</code>	a three dimensional array of statistics for random normalizing attributes' positive reinforcement for attributes,
<code>rndReinfNegAttr</code>	a three dimensional array of statistics for random normalizing attributes' negative reinforcement for attributes,

rndAnchorAttr	a three dimensional array of statistics for random normalizing attributes' anchoring for attributes.
attrNames	the names of attributes
valueNames	the values of attributes
noAttr	number of attributes
ordVal	maximal number of attribute values
variant	the variant of the algorithm used
file	the file to store the results
rndFile	the file to store random normalizations

The statistics used are median, 1st quartile, 3rd quartile, low and high percentile selected by `ordEvalNormalizingPercentile`, mean, standard deviation, and expected probability according to value distribution. With these statistics we can visualize significance of reinforcements using adapted box and whiskers plot.

### Author(s)

Marko Robnik-Sikonja

### References

Marko Robnik-Sikonja, Koen Vanhoof: Evaluation of ordinal attributes at value level. *Knowledge Discovery and Data Mining*, 14:225-243, 2007

Marko Robnik-Sikonja, Igor Kononenko: Theoretical and Empirical Analysis of ReliefF and RReliefF. *Machine Learning Journal*, 53:23-69, 2003

Some of the references are available also from <http://lkm.fri.uni-lj.si/rmarko/papers/>

### See Also

[plot.ordEval](#), [CORElearn](#), [CoreModel](#), [helpCore](#), [infoCore](#).

### Examples

```
#prepare a data set
dat <- ordDataGen(200)

# evaluate ordered features with ordEval
est <- ordEval(class ~ ., dat, ordEvalNoRandomNormalizers=100)
# print(est)
printOrdEval(est)
plot(est)
```

---

paramCoreIO	<i>Input/output of parameters from/to file</i>
-------------	--

---

### Description

All the parameters of the given model are written directly to file, or read from file into model.

### Usage

```
paramCoreIO(model, fileName, io=c("read","write"))
```

### Arguments

model	The model structure as returned by <a href="#">CoreModel</a> .
fileName	Name of the parameter file.
io	Controls weather the parameters will be read or written.

### Details

The function uses the model structure as returned by [CoreModel](#) and reads or writes all its parameters from/to file. If parameter `io="read"` parameters are read from file filename. If parameter `io="write"` parameters are written to file filename.

### Value

Returns invisible list with parameters passed to C function: `list(modelID, filename, io)`.

### Author(s)

Marko Robnik-Sikonja

### See Also

[CORElearn](#), [helpCore](#).

### Examples

```
# use iris data
# build random forests model with certain parameters
modelRF <- CoreModel(Species ~ ., iris, model="rf",
                     selectionEstimator="MDL",minNodeWeightRF=5,
                     rfNoTrees=50, maxThreads=1)

# writes all the used parameters to file
paramCoreIO(modelRF, "parameters.par", io="write")
# and reads them back into the model
paramCoreIO(modelRF, "parameters.par", io="read")
```

```
# clean up for the sake of R package checks
file.remove("parameters.par")

destroyModels(modelRF) # clean up
```

---

plot.CoreModel

*Visualization of CoreModel models*


---

### Description

The method `plot` visualizes the models returned by `CoreModel()` function or summaries obtained by applying these models to data. Different plots can be produced depending on the type of the model.

### Usage

```
## S3 method for class 'CoreModel'
plot(x, trainSet, rfGraphType=c("attrEval", "outliers", "scaling",
  "prototypes", "attrEvalCluster"), clustering=NULL, ...)
```

### Arguments

<code>x</code>	The model structure as returned by <a href="#">CoreModel</a> .
<code>trainSet</code>	The data frame containing training data which produced the model <code>x</code> .
<code>rfGraphType</code>	The type of the graph to produce for random forest models. See details.
<code>clustering</code>	The clustering of the training instances used in some model types. See details.
<code>...</code>	Other options controlling graphical output passed to additional graphical functions.

### Details

The output of function [CoreModel](#) is visualized. Depending on the model type, different visualizations are produced. Currently, classification tree, regression tree, and random forests are supported (models "tree", "regTree", "rf", and "rfNear").

For classification and regression trees (models "tree" and "regTree") the visualization produces a graph representing structure of classification and regression tree, respectively. This process exploits graphical capabilities of [rpart.plot](#) package. Internal structures of `CoreModel` are converted to [rpart.object](#) and then visualized by calling [rpart.plot](#) using default parameters. Any additional parameters are passed on to this function. For further control use the [getRpartModel](#) function and call the function [rpart.plot](#) or [plot.rpart](#) with different parameters. Note that [rpart.plot](#) can only display a single value in a leaf, which is not appropriate for model trees using e.g., linear regression in the leaves. For these cases function [display](#) is a better alternative.

For random forest models (models "rf" and "rfNear") different types of visualizations can be produced depending on the `graphType` parameter:



- "attrEval" the attributes are evaluated with random forest model and the importance scores are then visualized. For details see [rfAttrEval](#).
- "attrEvalClustering" similarly to the "attrEval" the attributes are evaluated with random forest model and the importance scores are then visualized, but the importance scores are generated for each cluster separately. The parameter clustering provides clustering information on the trainSet. If clustering parameter is set to NULL, the class values are used as clustering information and visualization of attribute importance for each class separately is generated. For details see [rfAttrEvalClustering](#).
- "outliers" the random forest proximity measure of training instances in trainSet is visualized and outliers for each class separately can be detected. For details see [rfProximity](#) and [rfOutliers](#).
- "prototypes" typical instances are found based on predicted class probabilities and their values are visualized (see [classPrototypes](#)).
- "scaling" returns a scaling plot of training instances in a two dimensional space using random forest based proximity as the distance (see [rfProximity](#) and a scaling function [cmdscale](#)).

### Value

The method returns no value.

### Author(s)

John Adeyanju Alao (initial implementation) and Marko Robnik-Sikonja (integration, improvements)

### References

Leo Breiman: Random Forests. *Machine Learning Journal*, 45:5-32, 2001

### See Also

[CoreModel](#), [rfProximity](#), [pam](#), [rfClustering](#), [rfAttrEvalClustering](#), [rfOutliers](#), [classPrototypes](#), [cmdscale](#)

### Examples

```
# decision tree
dataset <- iris
md <- CoreModel(Species ~ ., dataset, model="tree")
plot(md, dataset) # additional parameters are passed directly to rpart.plot

# Additional visualizations can be obtained by explicit conversion to rpart.object
#rpm <- getRpartModel(md,dataset)
# and then setting graphical parameters in plot.rpart and text.rpart
#require(rpart)
# E.g., set angle to tan(0.5)=45 (degrees) and length of branches at least 5,
# try to make a dendrogram more compact
#plot(rpm, branch=0.5, minbranch=5, compress=TRUE)
```

```

#(pretty=0) full names of attributes, numbers to 3 decimals,
#text(rpm, pretty=0, digits=3)

destroyModels(md) # clean up

# regression tree
dataset <- CO2
mdr <- CoreModel(uptake ~ ., dataset, model="regTree")
plot(mdr, dataset)
destroyModels(mdr) # clean up

#random forests
dataset <- iris
mdRF <- CoreModel(Species ~ ., dataset, model="rf", rfNoTrees=30, maxThreads=1)
plot(mdRF, dataset, rfGraphType="attrEval")
plot(mdRF, dataset, rfGraphType="outliers")
plot(mdRF, dataset, rfGraphType="scaling")
plot(mdRF, dataset, rfGraphType="prototypes")
plot(mdRF, dataset, rfGraphType="attrEvalCluster", clustering=NULL)
destroyModels(mdRF) # clean up

```

---

plot.ordEval

*Visualization of ordEval results*


---

## Description

The method `plot` visualizes the results of `ordEval` algorithm with an adapted box-and-whiskers plots. The method `printOrdEval` prints summary of the results in a text format.

## Usage

```

plotOrdEval(file, rndFile, ...)

## S3 method for class 'ordEval'
plot(x, graphType=c("avBar", "attrBar", "avSlope"), ...)

printOrdEval(x)

```

## Arguments

<code>x</code>	The object containing results of <code>ordEval</code> algorithm obtained by calling <code>ordEval</code> . If this object is not given, it has to be constructed from files <code>file</code> and <code>rndFile</code> .
<code>file</code>	Name of file where evaluation results of <code>ordEval</code> algorithm were written to.
<code>rndFile</code>	Name of file where evaluation of random normalizing attributes by <code>ordEval</code> algorithm were written to.
<code>graphType</code>	The type of the graph to produce. Can be any of <code>"avBar"</code> , <code>"attrBar"</code> , <code>"avSlope"</code> .
<code>...</code>	Other options controlling graphical output, used by specific graphical methods. See details.

## Details

The output of function `ordEval` either returned directly or stored in files `file` and `rndFile` is read and visualized. The type of graph produced is controlled by `graphType` parameter:

- `avBar` the positive and negative reinforcement of each value of each attribute is visualized as the length of the bar. For each value also a normalizing modified box and whiskers plot is produced above it, showing the confidence interval of the same attribute value under the assumption that the attribute contains no information. If the length of the bar is outside the normalizing whiskers, this is a statistically significant indication that the value is important.
- `attrBar` the positive and negative reinforcement for each attribute is visualized as the length of the bar. This reinforcement is weighted sum of contributions of individual values visualized with `avBar` graph type.
- `avSlope` the positive and negative reinforcement of each value of each attribute is visualized as the slope of the line segment connecting consequent values

The `avBar` and `avSlope` produce several graphs (one for each attribute). In order to see them all on an interactive device use `devAskNewPage`. On some platforms or in RStudio environment the graphical window stores the history and one can browse through recent pages. Alternatively use any of non-interactive devices such as `pdf` or `postscript`. Some support for opening and handling of these devices is provided by function `preparePlot`. The user should take care to call `dev.off` after completion of the operations.

There are some additional optional parameters ... which are important to all or for some graph types.

- `ciType` The type of the confidence interval in "avBar" and "attrBar" graph types. Can be "two.sided", "upper", "lower", or "none". Together with `ordEvalNormalizingPercentile` parameter in `ordEval`, `ciType`, `ciDisplay`, and `ciDecorate` controls the type, length and display of confidence intervals for each value.
- `ciDisplay` The way how confidence intervals are displayed. Can be "box" or "color". The value "box" displays confidence interval as box and whiskers plot above the actual value with whiskers representing confidence percentiles. The value "color" displays only the upper limit of confidence interval, namely the value (represented with a length of the bar) beyond the confidence interval is displayed with more intensive color or shade.
- `ciDecorate` controls if the reinforcement factors stretching outside the confidence intervals of possible random effects are decorated by being circled with an ellipse. The default value NULL means that there are no decorations, other values are interpreted as colors in the function `draw.ellipse`, e.g., `ciDecorate="red"` draws red ellipses around statistically significant reinforcementnets.
- `equalUpDown` a boolean specifying if upward and downward reinforcement of the same value are to be displayed side by side on the same level; it usually makes sense to set this parameter to TRUE when specifying a single value differences by setting `variant="attrDist1"` in `ordEval` function.
- `graphTitle` specifies text to incorporate into the title.
- `attrIdx` displays plot for a single attribute with specified index.
- `xlabel` label of lower horizontal axis.
- `ylabelLeft` label of the left-hand vertical axis.

- ylabRight label of the right-hand vertical axis.
- colors a vector with four colors specifying colors of reinforcement bars for down, down\_beyond, up, and up\_beyond, respectively. If set to NULL this produces black and white graph with shades of gray. The colors down\_beyond and up\_beyond depict the confidence interval if parameter ciDisplay="color".  
The default values are colors=c("green", "lightgreen", "blue", "lightblue").

### Value

The method returns no value.

### Author(s)

Marko Robnik-Sikonja

### References

Marko Robnik-Sikonja, Koen Vanhoof: Evaluation of ordinal attributes at value level. *Knowledge Discovery and Data Mining*, 14:225-243, 2007

Marko Robnik-Sikonja, Igor Kononenko: Theoretical and Empirical Analysis of ReliefF and RReliefF. *Machine Learning Journal*, 53:23-69, 2003

Some of the references are available also from <http://lkm.fri.uni-lj.si/rmarko/papers/>

### See Also

[ordEval](#), [helpCore](#), [preparePlot](#), [CORElearn](#)

### Examples

```
# prepare a data set
dat <- ordDataGen(200)

# evaluate ordered features with ordEval
oe <- ordEval(class ~ ., dat, ordEvalNoRandomNormalizers=200)
plot(oe)
# printOrdEval(oe)

# the same effect we achieve by storing results to files
tmp <- ordEval(class ~ ., dat, file="profiles.oe",
               rndFile="profiles.oer", ordEvalNoRandomNormalizers=200)
plotOrdEval(file="profiles.oe", rndFile="profiles.oer",
            graphType="attrBar")
# clean up for the sake of R package checks
file.remove("profiles.oe")
file.remove("profiles.oer")
```

---

predict.CoreModel      *Prediction using constructed model*

---

### Description

Using a previously built model and new data, predicts the class value and probabilities for classification problem and function value for regression problem.

### Usage

```
## S3 method for class 'CoreModel'  
predict(object, newdata, ..., costMatrix=NULL,  
        type=c("both", "class", "probability"))
```

### Arguments

object	The model structure as returned by <a href="#">CoreModel</a> .
newdata	Data frame with fresh data.
costMatrix	Optional cost matrix can provide nonuniform costs for classification problems.
type	Controls what will be return value in case of classification.
...	Other model dependent options for prediction. See <a href="#">helpCore</a> .

### Details

The function uses the object structure as returned by [CoreModel](#) and applies it on the data frame newdata. The newdata must be transformable using the formula specified for building the model (with dependent variable removed). If the dependent variable is present in newdata, it is ignored.

Optional cost matrix can provide nonuniform costs for classification problems. For regression problem this parameter is ignored. The costs can be different from the ones used for building the model in [CoreModel](#).

### Value

For regression model a vector of predicted values for given input instances. For classification problem the parameter type controls what is returned. With default value "both" function returns a list with two components class and probabilities containing predicted class values and probabilities for all class values, respectively. With type set to "class" or "probability" the function returns only the selected component as vector or matrix.

### Author(s)

Marko Robnik-Sikonja, Petr Savicky

### See Also

[CORElearn](#), [CoreModel](#), [modelEval](#), [helpCore](#), [paramCoreIO](#).

**Examples**

```
# use iris data set

# build random forests model with certain parameters
modelRF <- CoreModel(Species ~ ., iris, model="rf",
                     selectionEstimator="MDL",minNodeWeightRF=5,rfNoTrees=100)
print(modelRF)

# prediction with node distribution
pred <- predict(modelRF, iris, rfPredictClass=FALSE, type="both")
# print(pred)

destroyModels(modelRF) # clean up
```

---

```
preparePlot
```

```
Prepare graphics device
```

---

**Description**

Based on provided `fileName` opens and sets appropriate graphical device: pdf, postscript, interactive graphical window, or (only on windows) windows metafile,.

**Usage**

```
preparePlot(fileName="Rplot",...)
```

**Arguments**

<code>fileName</code>	Name of the file to store the output to.
<code>...</code>	Further parameters passed to device.

**Details**

The function opens the graphical output device based on `fileName` extension. The extensions `.pdf`, `.ps`, `.jpg`, `.bmp`, `.tif`, `.png`, `.tiff` or none select `pdf`, `postscript`, `jpeg`, `bmp`, `tiff`, `png`, `bitmap` or a default (interactive) graphical device.

On Windows also `.emf` extension is supported which opens `win.metafile` and creates vector graphics in windows enhanced metafile format.

The extension `.tiff` opens `bitmap` device which produces bitmap via `postscript` device. Therefore it requires Ghostscript to be installed and on the executable path.

Some sensible default values are passed to created devices, but further options can be passed via  
`....`

**Value**

A plot device is opened and nothing is returned to the R interpreter.

**Author(s)**

Marko Robnik-Sikonja

**See Also**[CORElearn](#), [plot.ordEval](#), [pdf](#), [postscript](#), [jpeg](#), [bmp](#), [tiff](#), [png](#), [Devices](#)**Examples**

```
# prepare a data set
dat <- ordDataGen(200)
# evaluate ordered features with ordEval
oe <- ordEval(class ~ ., dat, ordEvalNoRandomNormalizers=200)

# the following line if uncommented will create a separate
# postscript file with given filename for each attribute
# preparePlot("myGraph%03d.ps")

plot(oe)
dev.off()
```

regDataGen

*Artificial data for testing regression algorithms***Description**

The generator produces regression data data with 4 discrete and 7 numeric attributes.

**Usage**

```
regDataGen(noInst, t1=0.8, t2=0.5, noise=0.1)
```

**Arguments**

noInst	Number of instances to generate.
t1, t2	Parameters controlling the shape of the distribution.
noise	Parameter controlling the amount of noise. If noise=0, there is no noise. If noise = 1, then the level of the signal and noise are the same.

**Details**

The response variable is derived from  $x_4$ ,  $x_5$ ,  $x_6$  using two different functions. The choice depends on a hidden variable, which determines whether the response value would follow a linear dependency  $f = x_4 - 2x_5 + 3x_6$ , or a nonlinear one  $f = \cos(4\pi x_4)(2x_5 - 3x_6)$ .

Attributes  $a_1$ ,  $a_2$ ,  $x_1$ ,  $x_2$  carry some information on the hidden variables depending on parameters  $t_1$ ,  $t_2$ . Extreme values of the parameters are  $t_1=0.5$  and  $t_2=1$ , when there is no information. On the

other hand, if  $t_1=0$  or  $t_1=1$  then each of the attributes  $a_1$ ,  $a_2$  carries full information. If  $t_2=0$ , then each of  $x_1$ ,  $x_2$  carries full information on the hidden variable.

The attributes  $x_4$ ,  $x_5$ ,  $x_6$  are available with a noise level depending on parameter `noise`. If `noise=0`, there is no noise. If `noise=1`, then the level of the signal and noise are the same.

### Value

Returns a `data.frame` with `noInst` rows and 11 columns. Range of values of the attributes and response are

<code>a1</code>	0,1
<code>a2</code>	a,b,c,d
<code>a3</code>	0,1 (irrelevant)
<code>a4</code>	a,b,c,d (irrelevant)
<code>x1</code>	numeric (gaussian with different sd for each class)
<code>x2</code>	numeric (gaussian with different sd for each class)
<code>x3</code>	numeric (gaussian, irrelevant)
<code>x4</code>	numeric from [0,1]
<code>x5</code>	numeric from [0,1]
<code>x6</code>	numeric from [0,1]
<code>response</code>	numeric

### Author(s)

Petr Savicky

### See Also

[classDataGen](#), [ordDataGen](#), [CoreModel](#),

### Examples

```
#prepare a regression data set
regData <- regDataGen(noInst=200)

# build regression tree similar to CART
modelRT <- CoreModel(response ~ ., regData, model="regTree", modelTypeReg=1)
print(modelRT)

destroyModels(modelRT) # clean up
```



---

reliabilityPlot	<i>Plots reliability plot of probabilities</i>
-----------------	--

---

### Description

Given probability scores `probScore` and true probabilities `trueProb` the methods plots one against the other using a selected boxing method which groups scores and probabilities to show calibration of probabilities in given probability bands.

### Usage

```
reliabilityPlot(probScore, trueProb, titleText="", boxing="equipotent",  
               noBins=10, classValue = 1, printWeight=FALSE)
```

### Arguments

<code>probScore</code>	A vector of predicted probabilities for a given class <code>classValue</code> .
<code>trueProb</code>	A vector of true probabilities for a given <code>classValue</code> , should be of the same length as <code>probScore</code> .
<code>titleText</code>	The text of the graph title.
<code>boxing</code>	One of "unique", "equidistant" or "equipotent", determines the grouping of probabilities. See details below.
<code>noBins</code>	The value of parameter depends on the parameter <code>boxing</code> and specifies the number of bins. See details below.
<code>classValue</code>	A class value (factor) or an index of the class value (integer) for which reliability plot is made.
<code>printWeight</code>	A boolean specifying if box weights are to be printed.

### Details

Depending on the specified `boxing` the probability scores are grouped in one of three possible ways

- "unique" each unique probability score forms its own box.
- "equidistant" forms `noBins` equally wide boxes.
- "equipotent" forms `noBins` boxes with equal number of scores in each box.

The parameter `trueProb` can represent either probabilities (in  $[0, 1]$  range, in most cases these will be 0s or 1s), or the true class values from which the method will form 0 and 1 values corresponding to probabilities for class value `classValue`.

### Value

A function returns a graph containing reliability plot on a current graphical device.

### Author(s)

Marko Robnik-Sikonja

**See Also**

[CORElearn](#), [calibrate](#).

**Examples**

```
# generate data consisting from 3 parts:
# one part for training, one part for calibration, one part for testing
train <- classDataGen(noInst=200)
cal <- classDataGen(noInst=200)
test <- classDataGen(noInst=200)

# build random forests model with default parameters
modelRF <- CoreModel(class~., train, model="rf")
# prediction of calibration and test set
predCal <- predict(modelRF, cal, rfPredictClass=FALSE)
predTest <- predict(modelRF, test, rfPredictClass=FALSE)
destroyModels(modelRF) # no longer needed, clean up

# show reliability plot of uncalibrated test set
class1<-1
par(mfrow=c(1,2))
reliabilityPlot(predTest$prob[,class1], test$class,
                titleText="Uncalibrated probabilities", classValue=class1)

# calibrate for a chosen class1 and method using calibration set
calibration <- calibrate(cal$class, predCal$prob[,class1], class1=1,
                        method="isoReg", assumeProbabilities=TRUE)
calTestProbs <- applyCalibration(predTest$prob[,class1], calibration)
# display calibrated probabilities
reliabilityPlot(calTestProbs, test$class,
                titleText="Calibrated probabilities", classValue=class1)
```

---

rfAttrEval

*Attribute evaluation with random forest*


---

**Description**

The method evaluates the quality of the features/attributes/dependent variables used in the given random forest model.

**Usage**

```
rfAttrEval(model)
rfAttrEvalClustering(model, dataset, clustering=NULL)
```

**Arguments**

model	The model of type rf or rfNear as returned by <a href="#">CoreModel</a> .
dataset	Training instances that produced random forest model.
clustering	A clustering vector of dataset training instances used in model.

**Details**

The attributes are evaluated via provided random forest's out-of-bag sets. Values for each attribute in turn are randomly shuffled and classified with random forest. The difference between average margin of non-shuffled and shuffled instances serves as a quality estimate of the attribute. The function `rfAttrEvalClustering` uses a clustering of the training instances to produce importance score of attributes for each cluster separately. If parameter `clustering` is set to `NULL` the actual class values of the instances are used as clusters thereby producing the evaluation of attributes specific for each of the class values.

**Value**

In case of `rfAttrEval` a vector of evaluations for the features in the order specified by the formula used to generate the provided model. In case of `rfAttrEvalClustering` a matrix is returned, where each row contains evaluations for one of the clusters.

**Author(s)**

Marko Robnik-Sikonja (thesis supervisor) and John Adeyanju Alao (as a part of his BSc thesis)

**References**

Marko Robnik-Sikonja: Improving Random Forests. In J.-F. Boulicaut et al.(Eds): ECML 2004, LNAI 3210, Springer, Berlin, 2004, pp. 359-370 Available also from <http://lkm.fri.uni-lj.si/rmarko/papers/>

Leo Breiman: Random Forests. Machine Learning Journal, 2001, 45, 5-32

**See Also**

[CORElearn](#), [CoreModel](#), [attrEval](#).

**Examples**

```
# build random forests model with certain parameters
modelRF <- CoreModel(Species ~ ., iris, model="rf",
                    selectionEstimator="MDL", minNodeWeightRF=5,
                    rfNoTrees=100, maxThreads=1)
rfAttrEval(modelRF) # feature evaluations

x <- rfAttrEval(modelRF) # feature evaluations for each class
print(x)

destroyModels(modelRF) # clean up
```

---

`rfClustering`*Random forest based clustering*

---

**Description**

Creates a clustering of random forest training instances. Random forest provides proximity of its training instances based on their out-of-bag classification. This information is usually passed to visualizations (e.g., scaling) and attribute importance measures.

**Usage**

```
rfClustering(model, noClusters=4)
```

**Arguments**

<code>model</code>	a random forest model returned by <a href="#">CoreModel</a>
<code>noClusters</code>	number of clusters

**Details**

The method calls [pam](#) function for clustering, initializing its distance matrix with random forest based similarity by calling [rfProximity](#) with argument `model`.

**Value**

An object of class `pam` representing the clustering (see `?pam.object` for details), the most important being a vector of cluster assignments (named `cluster`) to training instances used to generate the model.

**Author(s)**

John Adeyanju Alao (as a part of his BSc thesis) and Marko Robnik-Sikonja (thesis supervisor)

**References**

Leo Breiman: Random Forests. *Machine Learning Journal*, 45:5-32, 2001

**See Also**

[CoreModel](#) [rfProximity](#) [pam](#)

**Examples**

```
set<-iris
md<-CoreModel(Species ~ ., set, model="rf", rfNoTrees=30, maxThreads=1)
mdCluster<-rfClustering(md, 5)

destroyModels(md) # clean up
```

---

`rf00B`*Out-of-bag performance estimation for random forests*

---

**Description**

The method returns internal out-of-bag performance evaluation for given random forests `model`.

**Usage**

```
rf00B(model)
```

**Arguments**

`model`            The model of type `rf` or `rfNear` as returned by [CoreModel](#).

**Details**

The method returns random forest performance estimations obtained via its out-of-bag sets. The performance measures returned are classification accuracy, average classification margin, and correlation between trees in the forest. The classification margin is defined as the difference between probability of the correct class and probability of the most probable incorrect class. The correlation between models is estimated as the ratio between classification margin variance and variance of the forest as defined in (Breiman, 2001).

**Value**

The list containing three performance measures computed with out-of-bag instances is returned:

<code>accuracy</code>	the classification accuracy of the forest,
<code>margin</code>	the average margin of classification with the forest,
<code>correlation</code>	the correlation between trees in the forest.

**Author(s)**

Marko Robnik-Sikonja.

**References**

Leo Breiman: Random Forests. *Machine Learning Journal*, 2001, 45, 5-32

**See Also**

[CORElearn](#), [CoreModel](#).

## Examples

```
# build random forests model with certain parameters
modelRF <- CoreModel(Species ~ ., iris, model="rf",
                    selectionEstimator="MDL", minNodeWeightRF=5,
                    rfNoTrees=100, maxThreads=1)
rfOOB(modelRF)

destroyModels(modelRF) # clean up
```

---

rfOutliers

*Random forest based outlier detection*

---

## Description

Based on random forest instance proximity measure detects training cases which are different to all other cases.

## Usage

```
rfOutliers(model, dataset)
```

## Arguments

model	a random forest model returned by <a href="#">CoreModel</a>
dataset	a training set used to generate the model

## Details

Strangeness is defined using the random forest model via a proximity matrix (see [rfProximity](#)). If the number is greater than 10, the case can be considered an outlier according to Breiman 2001.

## Value

For each instance from a dataset the function returns a numeric score of its strangeness to other cases.

## Author(s)

John Adeyanju Alao (as a part of his BSc thesis) and Marko Robnik-Sikonja (thesis supervisor)

## References

Leo Breiman: Random Forests. *Machine Learning Journal*, 45:5-32, 2001

## See Also

[CoreModel](#), [rfProximity](#), [rfClustering](#).

## Examples

```
#first create a random forest tree using CORElearn
dataset <- iris
md <- CoreModel(Species ~ ., dataset, model="rf", rfNoTrees=30,
                maxThreads=1)
outliers <- rfOutliers(md, dataset)
plot(abs(outliers))
#for a nicer display try
plot(md, dataset, rfGraphType="outliers")

destroyModels(md) # clean up
```

---

rfProximity

*A random forest based proximity function*

---

## Description

Random forest computes similarity between instances with classification of out-of-bag instances. If two out-of-bag cases are classified in the same tree leaf the proximity between them is incremented.

## Usage

```
rfProximity(model, outProximity=TRUE)
```

## Arguments

`model` a CORElearn model of type random forest.  
`outProximity` if TRUE, function returns a proximity matrix, else it returns a distance matrix.

## Details

A proximity is transformed into distance with expression  $\text{distance} = \sqrt{1 - \text{proximity}}$ .

## Value

Function returns an M by M matrix where M is the number of training instances. Returned matrix is used as an input to other function (see [rfOutliers](#) and [rfClustering](#)).

## Author(s)

John Adeyanju Alao (as a part of his BSc thesis) and Marko Robnik-Sikonja (thesis supervisor)

## References

Leo Breiman: Random Forests. *Machine Learning Journal*, 45:5-32, 2001

**See Also**

[CoreModel](#), [rfOutliers](#), [cmdscales](#), [rfClustering](#).

**Examples**

```
md <- CoreModel(Species ~ ., iris, model="rf", rfNoTrees=30, maxThreads=1)
pr <- rfProximity(md, outProximity=TRUE)
# visualization
require(lattice)
levelplot(pr)

destroyModels(md) # clean up
```

---

saveRF

*Saves/loads random forests model to/from file*

---

**Description**

saveRF: the internal structure of given random forests model is saved to file. loadRF: the internal structure of random forests model is loaded from given file and a model is created and returned.

**Usage**

```
saveRF(model, fileName)
loadRF(fileName)
```

**Arguments**

model	The model structure as returned by <a href="#">CoreModel</a> .
fileName	Name of the file to save/load the model to/from.

**Details**

The function saveRF saves the internal structure of given random forests model to file. The structures from C++ code are stored to the file with specified file, while internal structures from R are stored to file named fileName.Rda. The model must be a valid structure returned by [CoreModel](#).

The function loadRF loads the internal structure of random forests saved in a specified files and returns access to it.

**Value**

saveRF invisibly returns some debugging information, while loadRF returns a loaded model as a list, similarly to [CoreModel](#).



**Author(s)**

Marko Robnik-Sikonja

**See Also**

[CORElearn](#), [CoreModel](#).

**Examples**

```
# use iris data set

# build random forests model with certain parameters
modelRF <- CoreModel(Species ~ ., iris, model="rf",
                    selectionEstimator="MDL", minNodeWeightRF=5,
                    rfNoTrees=100, maxThreads=1)
print(modelRF)

# prediction with node distribution
pred <- predict(modelRF, iris, rfPredictClass=FALSE, type="both")
# print(pred)

# saves the random forests model to file
saveRF(modelRF, "tempRF.txt")

# restore the model to another model
loadedRF = loadRF("tempRF.txt")

# prediction should be the same
predLoaded <- predict(loadedRF, iris, rfPredictClass=FALSE, type="both")
# print(predLoaded)
# sum of differences should be zero subject to numeric imprecision
sum(pred$probabilities - predLoaded$probabilities)

cat("Are predicted classes of original and retrieved models equal? ",
    all(pred$class == predLoaded$class), "\n" )
# cat("Are predicted probabilities of original and retrieved model equal? ",
#     all(pred$probabilities == predLoaded$probabilities), "\n" )

# clean up the models when no longer needed
destroyModels(modelRF)
destroyModels(loadedRF)

# clean up for the sake of R package checks
file.remove("tempRF.txt")
file.remove("tempRF.txt.Rda")
```

---

`testCore`*Verification of the CORElearn installation*

---

**Description**

Performs a partial check of the classification part of CORElearn.

**Usage**

```
testCoreClass(continue=TRUE)
testCoreAttrEval(continue=TRUE)
testCoreReg(continue=TRUE)
testCoreOrdEval(continue=TRUE)
testCoreNA(continue=TRUE)
testCoreRPORT(continue=TRUE)
testCoreRand(continue=TRUE)
allTests(continue=TRUE, timed=FALSE)
```

**Arguments**

<code>continue</code>	Logical. Whether a warning or an error should be generated when a test fails.
<code>timed</code>	Logical. Whether the time usage should be printed.

**Details**

Functions `testCoreClass()`, `testCoreAttrEval()`, `testCoreReg()` evaluate functions `CoreModel()`, `predict.CoreModel()`, `modelEval()`, and `attrEval()` and perform a partial check of the obtained results.

Function `testNA()` performs a test of consistency NA and NaN between R and CORElearn.

Functions `testCoreRPORT()` and `testCoreRand()` test, whether the `R_PORT` directive is defined in C code and whether R random number generator is used. These tests are mostly used for debugging.

Function `allTests()` calls all the above functions and prints a table of the results. If an error is found, a more detailed information is printed and the continuation of the tests depends on the argument `continue`.

**Value**

The functions have no output value. The result OK or FAILED is printed.

**Author(s)**

Marko Robnik-Sikonja, Petr Savicky

**See Also**

[CORElearn](#).

**Examples**

```
allTests() # run all tests and generate an error, if any of the tests fails
```

---

versionCore	<i>Package version</i>
-------------	------------------------

---

**Description**

Prints package version obtained from C code.

**Usage**

```
versionCore()
```

**Arguments**

None.

**Details**

The function returns the information about the current version obtained from underlying C library `link{CORElearn}`.

**Value**

Character string with information about the version.

**Author(s)**

Marko Robnik-Sikonja, Petr Savicky

**See Also**

[CORElearn](#).

**Examples**

```
# load the package
library(CORElearn)

# print its version
versionCore()
```

# Index

- \* **classif**
  - attrEval, 5
  - calibrate, 10
  - CORElearn-internal, 16
  - CORElearn-package, 3
  - CoreModel, 17
  - destroyModels, 23
  - discretize, 24
  - getCoreModel, 28
  - getRFsizes, 29
  - helpCore, 31
  - infoCore, 37
  - modelEval, 38
  - ordEval, 43
  - paramCoreIO, 47
  - plot.ordEval, 50
  - predict.CoreModel, 53
  - reliabilityPlot, 57
  - rfAttrEval, 58
  - rfOOB, 61
  - saveRF, 64
  - testCore, 66
  - versionCore, 67
- \* **cluster**
  - plot.CoreModel, 48
  - rfClustering, 60
  - rfOutliers, 62
  - rfProximity, 63
- \* **datagen**
  - classDataGen, 12
  - ordDataGen, 42
  - regDataGen, 55
- \* **datasets**
  - CORElearn-package, 3
- \* **data**
  - classDataGen, 12
  - ordDataGen, 42
  - regDataGen, 55
- \* **loess**
  - CORElearn-package, 3
  - CoreModel, 17
  - modelEval, 38
  - predict.CoreModel, 53
- \* **models**
  - calibrate, 10
  - CORElearn-internal, 16
  - CORElearn-package, 3
  - CoreModel, 17
  - destroyModels, 23
  - getCoreModel, 28
  - getRFsizes, 29
  - helpCore, 31
  - infoCore, 37
  - modelEval, 38
  - paramCoreIO, 47
  - predict.CoreModel, 53
  - reliabilityPlot, 57
  - rfAttrEval, 58
  - rfOOB, 61
  - saveRF, 64
  - versionCore, 67
- \* **multivariate**
  - CORElearn-package, 3
  - CoreModel, 17
  - getCoreModel, 28
  - getRFsizes, 29
  - modelEval, 38
  - predict.CoreModel, 53
- \* **nonlinear**
  - attrEval, 5
  - CORElearn-package, 3
  - CoreModel, 17
  - discretize, 24
  - helpCore, 31
  - infoCore, 37
  - modelEval, 38
  - ordEval, 43
  - paramCoreIO, 47

- predict.CoreModel, 53
- rfAttrEval, 58
- rfOOB, 61
- saveRF, 64
- versionCore, 67
- \* **package**
  - CORElearn-package, 3
- \* **regression**
  - attrEval, 5
  - CORElearn-internal, 16
  - CORElearn-package, 3
  - CoreModel, 17
  - destroyModels, 23
  - discretize, 24
  - getCoreModel, 28
  - getRFsizes, 29
  - helpCore, 31
  - infoCore, 37
  - modelEval, 38
  - ordEval, 43
  - paramCoreIO, 47
  - predict.CoreModel, 53
  - saveRF, 64
  - versionCore, 67
- \* **robust**
  - classPrototypes, 15
  - plot.CoreModel, 48
  - rfClustering, 60
  - rfOutliers, 62
  - rfProximity, 63
- \* **tree**
  - CORElearn-internal, 16
  - CORElearn-package, 3
  - CoreModel, 17
  - destroyModels, 23
  - display.CoreModel, 27
  - getCoreModel, 28
  - getRFsizes, 29
  - getRpartModel, 30
  - helpCore, 31
  - infoCore, 37
  - modelEval, 38
  - paramCoreIO, 47
  - plot.CoreModel, 48
  - predict.CoreModel, 53
  - rfAttrEval, 58
  - rfClustering, 60
  - rfOOB, 61
  - saveRF, 64
  - versionCore, 67
- allTests (testCore), 66
- applyCalibration (calibrate), 10
- applyDiscretization (discretize), 24
- attrEval, 3, 4, 5, 17, 19, 20, 25, 26, 31–34, 37, 38, 45, 59
- auxTest, 9
- bitmap, 54
- bmp, 54, 55
- calibrate, 10, 58
- classDataGen, 4, 12, 43, 56
- classPrototypes, 15, 49
- cmdscales, 49, 64
- CORElearn, 9, 10, 12, 17, 20, 22, 23, 26, 28, 29, 37, 41, 42, 46, 47, 52, 53, 55, 58, 59, 61, 65–67
- CORElearn (CORElearn-package), 3
- CORElearn-internal, 16
- CORElearn-package, 3
- CoreModel, 3, 4, 9, 15, 17, 17, 23, 25–31, 37–39, 41, 46–49, 53, 56, 59–62, 64, 65
- cvCoreModel (CoreModel), 17
- cvGen, 21
- cvGenStratified (cvGen), 21
- data.frame, 14, 43, 56
- destroyModels, 23
- dev.off, 51
- devAskNewPage, 51
- Devices, 55
- discretize, 24
- display, 30, 48
- display (display.CoreModel), 27
- display.CoreModel, 27
- gatherFromList (cvGen), 21
- getCoreModel, 28
- getRFsizes, 29
- getRpartModel, 30, 48
- help.Core (helpCore), 31
- helpCore, 3, 4, 6, 8, 9, 17–20, 25–27, 31, 37, 38, 45–47, 52, 53
- infoCore, 3, 4, 9, 17, 26, 37, 46

intervalMidPoint (discretize), 24

jpeg, 54, 55

loadRF (saveRF), 64

modelEval, 3, 4, 17, 19, 20, 38, 53

noEqualRows, 41

ordDataGen, 4, 15, 42, 56

OrdEval (ordEval), 43

ordEval, 3, 4, 8, 9, 17, 37, 43, 43, 45, 50–52

pam, 49, 60

paramCoreIO, 3, 4, 17, 20, 31, 37, 47, 53

pdf, 51, 54, 55

plot.CoreModel, 3, 4, 16, 27, 30, 48

plot.ordEval, 3, 4, 17, 44–46, 50, 55

plot.rpart, 48

plotOrdEval, 44

plotOrdEval (plot.ordEval), 50

png, 54, 55

postscript, 51, 54, 55

predict (predict.CoreModel), 53

predict.CoreModel, 3, 4, 10, 12, 16, 17, 19, 20, 37–39, 41, 53

preparePlot, 51, 52, 54

printOrdEval (plot.ordEval), 50

read.table, 44

regDataGen, 4, 15, 43, 55

reliabilityPlot, 12, 57

rfAttrEval, 8, 9, 49, 58

rfAttrEvalClustering, 49

rfAttrEvalClustering (rfAttrEval), 58

rfClustering, 49, 60, 62–64

rfOOB, 61

rfOutliers, 49, 62, 63, 64

rfProximity, 49, 60, 62, 63

rpart.object, 30, 48

rpart.plot, 48

saveRF, 64

testClassPseudoRandom (auxTest), 9

testCore, 66

testCoreAttrEval (testCore), 66

testCoreClass (testCore), 66

testCoreNA (testCore), 66

testCoreOrdEval (testCore), 66

testCoreRand (testCore), 66

testCoreReg (testCore), 66

testCoreRPORT (testCore), 66

testTime (auxTest), 9

tiff, 54, 55

versionCore, 3, 4, 17, 67