Package 'DMwR'

February 19, 2015

Type Package

Title Functions and data for ``Data Mining with R''

Version 0.4.1

Depends R(>= 2.10), methods, graphics, lattice (>= 0.18-3), grid (>= 2.10.1)

Imports xts (>= 0.6-7), quantmod (>= 0.3-8), zoo (>= 1.6-4), abind (>= 1.1-0), rpart (>= 3.1-46), class (>= 7.3-1), ROCR (>= 1.0)

Date 2013-08-08

Author Luis Torgo

Maintainer Luis Torgo <ltorgo@dcc.fc.up.pt>

Description This package includes functions and data accompanying the book ``Data Mining with R, learning with case studies'' by Luis Torgo, CRC Press 2010.

License GPL (>= 2)

LazyLoad yes

LazyData yes

NeedsCompilation no

Repository CRAN

Date/Publication 2013-08-08 19:46:37

R topics documented:

DMwR-package . . . 3
algae . . . 4
algae.sols . . . 4
bestScores . . . 5
bootRun-class . . . 6
bootSettings-class . . . 7
bootstrap . . . 8
centralImputation . . . 10
centralValue . . . 11
class.eval . . . 12

compAnalysis . . . 14
compExp-class . . . 16
CRchart . . . 18
crossValidation . . . 19
cvRun-class . . . 21
cvSettings-class . . . 22
dataset-class . . . 23
dist.to.knn . . . 24
dsNames . . . 25
experimentalComparison . . . 26
expSettings-class . . . 28
getFoldsResults . . . 29
getSummaryResults . . . 30
getVariant . . . 31
growingWindowTest . . . 33
GSPC . . . 35
hldRun-class . . . 35
hldSettings-class . . . 36
holdOut . . . 37
join . . . 40
kNN . . . 42
knneigh.vect . . . 43
knnImputation . . . 44
learner-class . . . 46
learnerNames . . . 47
LinearScaling . . . 47
lofactor . . . 48
loocv . . . 49
loocvRun-class . . . 52
loocvSettings-class . . . 53
manyNAs . . . 54
mcRun-class . . . 55
mcSettings-class . . . 56
monteCarlo . . . 57
outliers.ranking . . . 59
PRcurve . . . 62
prettyTree . . . 64
rankSystems . . . 65
reachability . . . 67
regr.eval . . . 68
ReScaling . . . 70
resp . . . 71
rpartXse . . . 72
rt.prune . . . 73
runLearner . . . 74
sales . . . 76
SelfTrain . . . 76
sigs.PR . . . 79

slidingWindowTest 80
SMOTE 82
SoftMax 84
statNames 85
statScores 86
subset-methods 87
task-class 88
test.algae 89
tradeRecord-class 90
trading.signals 91
trading.simulator 92
tradingEvaluation 96
ts.eval 99
unscale 101
variants 102

Index **104**

`DMwR-package` *Functions and data for the book "Data Mining with R"*

Description

This package includes functions and data accompanying the book "Data Mining with R, learning with case studies" by Luis Torgo, published by CRC Press (ISBN: 9781439810187)

Author(s)

Luis Torgo

Maintainer: Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

algae *Training data for predicting algae blooms*

Description

This data set contains observations on 11 variables as well as the concentration levels of 7 harmful algae. Values were measured in several European rivers. The 11 predictor variables include 3 contextual variables (season, size and speed) describing the water sample, plus 8 chemical concentration measurements.

Usage

```
algae
```

Format

A data frame with 200 observations and 18 columns.

Source

ERUDIT http://www.erudit.de/ - European Network for Fuzzy Logic and Uncertainty Modelling in Information Technology.

algae.sols *The solutions for the test data set for predicting algae blooms*

Description

This data set contains the values of the 7 harmful algae for the 140 test observations in the test set `test.algae`.

Usage

```
algae.sols
```

Format

A data frame with 140 observations and 7 columns.

Source

ERUDIT http://www.erudit.de/ - European Network for Fuzzy Logic and Uncertainty Modelling in Information Technology.

bestScores *Obtain the best scores from an experimental comparison*

Description

This function can be used to obtain the learning systems that obtained the best scores on an experimental comparison. This information will be shown for each of the evaluation statistics involved in the comparison and also for all data sets that were used.

Usage

```
bestScores(compRes, maxs = rep(F, dim(compRes@foldResults)[2]))
```

Arguments

compRes	A compExp object with the results of your experimental comparison.
maxs	A vector of booleans with as many elements are there are statistics measured in the experimental comparison. A True value means the respective statistic is to be maximized, while a False means minimization. Defaults to all False values.

Details

This is a handy function to check what were the best performers in a comparative experiment for each data set and each evaluation metric. The notion of "best performance" depends on the type of evaluation metric, thus the need of the second parameter. Some evaluation statistics are to be maximized (e.g. accuracy), while others are to be minimized (e.g. mean squared error). If you have a mix of these types on your experiment then you can use the maxs parameter to inform the function of which are to be maximized (minimized).

Value

The function returns a list with named components. The components correspond to the data sets used in the experimental comparison. For each component you get a data.frame, where the rows represent the statistics. For each statistic you get the name of the best performer (1st column of the data frame) and the respective score on that statistic (2nd column).

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

experimentalComparison, rankSystems, statScores

Examples

```
## Estimating several evaluation metrics on different variants of a
## regression tree and of a SVM, on  two data sets, using one repetition
## of  10-fold CV
data(swiss)
data(mtcars)

## First the user defined functions
cv.rpartXse <- function(form, train, test, ...) {
    require(DMwR)
    t <- rpartXse(form, train, ...)
    p <- predict(t, test)
    mse <- mean((p - resp(form, test))^2)
    c(nmse = mse/mean((mean(resp(form, train)) - resp(form, test))^2),
        mse = mse)
}

## run the experimental comparison
results <- experimentalComparison(
               c(dataset(Infant.Mortality ~ ., swiss),
                 dataset(mpg ~ ., mtcars)),
               c(variants('cv.rpartXse',se=c(0,0.5,1))),
               cvSettings(1,10,1234)
                                  )
## get the best scores for dataset and statistic
bestScores(results)
```

`bootRun-class` *Class "bootRun"*

Description

This is the class of the objects storing the results of a bootstrap experiment.

Objects from the Class

Objects can be created by calls of the form `bootRun(...)`. The objects contain information on the learner evaluated in the holdout experiment, the predictive task that was used, the holdout settings, and the results of the experiment.

Slots

`learner`: Object of class `"learner"`

`dataset`: Object of class `"task"`

`settings`: Object of class `"bootSettings"`

`foldResults`: Object of class `"matrix"` with the results of the experiment. The rows represent the different repetitions of the experiment while the columns the different statistics evaluated on each iteration.

Methods

summary `signature(object = "bootRun")`: method used to obtain a summary of the results of the holdout experiment.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

bootSettings, cvRun,loocvRun,mcRun,hldRun,compExp

Examples

```
showClass("bootRun")
```

---

bootSettings-class *Class "bootSettings"*

Description

This class of objects contains the information describing a bootstrap experiment, i.e. its settings.

Objects from the Class

Objects can be created by calls of the form `bootSettings(...)`. The objects contain information on the random number generator seed and on the number of repetitons of the boostrap process.

Slots

`bootSeed`**:** Object of class `"numeric"`with the random number generator seed (defaulting to 1234).

`bootReps`**:** Object of class `"numeric"` indicating the number of repetitions of the bootstrap experiment (defaulting to 50).

Extends

Class `"expSettings"`, directly.

Methods

show `signature(object = "bootSettings")`: method used to show the contents of a bootSettings object.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

bootRun, mcSettings, loocvSettings, cvSettings,hldSettings, expSettings

Examples

```
showClass("bootSettings")
```

---

bootstrap *Runs a bootstrap experiment*

Description

Function that performs a bootstrap experiment of a learning system on a given data set. The function is completely generic. The generality comes from the fact that the function that the user provides as the system to evaluate, needs in effect to be a user-defined function that takes care of the learning, testing and calculation of the statistics that the user wants to estimate using the bootstrap method.

Usage

```
bootstrap(sys, ds, sets, itsInfo = F, verbose = T)
```

Arguments

sys	sys is an object of the class learner representing the system to evaluate.
ds	ds is an object of the class dataset representing the data set to be used in the evaluation.
sets	sets is an object of the class cvSettings representing the cross validation experimental settings to use.
itsInfo	Boolean value determining whether the object returned by the function should include as an attribute a list with as many components as there are iterations in the experimental process, with each component containing information that the user-defined function decides to return on top of the standard error statistics. See the Details section for more information.
verbose	A boolean value controlling the level of output of the function execution, defaulting to T

Details

The idea of this function is to carry out a bootstrap experiment of a given learning system on a given data set. The goal of this experiment is to estimate the value of a set of evaluation statistics by means of the bootstrap method. Bootstrap estimates are obtained by averaging over a set of k scores each obtained in the following way: i) draw a random sample with replacement with the same size as the original data set; ii) obtain a model with this sample; iii) test it and obtain the estimates for this run on the observations of the original data set that were not used in the sample obtained in step i). This process is repeated k times and the average scores are the bootstrap estimates.

It is the user responsibility to decide which statistics are to be evaluated on each iteration and how they are calculated. This is done by creating a function that the user knows it will be called by this hold out routine at each repetition of the learn+test process. This user-defined function must assume that it will receive in the first 3 arguments a formula, a training set and a testing set, respectively. It should also assume that it may receive any other set of parameters that should be passed towards the learning algorithm. The result of this user-defined function should be a named vector with the values of the statistics to be estimated obtained by the learner when trained with the given training set, and tested on the given test set. See the Examples section below for an example of these functions.

If the `itsInfo` parameter is set to the value TRUE then the `hldRun` object that is the result of the function will have an attribute named itsInfo that will contain extra information from the individual repetitions of the hold out process. This information can be accessed by the user by using the function `attr()`, e.g. attr(returnedObject,'itsInfo'). For this information to be collected on this attribute the user needs to code its user-defined functions in a way that it returns the vector of the evaluation statistics with an associated attribute named itInfo (note that it is "itInfo" and not "itsInfo" as above), which should be a list containing whatever information the user wants to collect on each repetition. This apparently complex infra-structure allows you to pass whatever information you which from each iteration of the experimental process. A typical example is the case where you want to check the individual predictions of the model on each test case of each repetition. You could pass this vector of predictions as a component of the list forming the attribute itInfo of the statistics returned by your user-defined function. In the end of the experimental process you will be able to inspect/use these predictions by inspecting the attribute itsInfo of the `bootRun` object returned by the `bootstrap()` function. See the Examples section on the help page of the function `holdout()` for an illustration of this potentiality.

Value

The result of the function is an object of class `bootRun`.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`experimentalComparison`, `bootRun`,`bootSettings`, `monteCarlo`, `holdOut`, `loocv`, `crossValidation`

Examples

```
## Estimating the mean absolute error and the normalized mean squared
## error of rpart on the swiss data, using one repetition of 10-fold CV
data(swiss)

## First the user defined function (note: can have any name)
user.rpart <- function(form, train, test, ...) {
    require(rpart)
    model <- rpart(form, train, ...)
    preds <- predict(model, test)
    regr.eval(resp(form, test), preds,
              stats=c('mae','nmse'), train.y=resp(form, train))
}

## Now the evaluation
eval.res <- bootstrap(learner('user.rpart',pars=list()),
                      dataset(Infant.Mortality ~ ., swiss),
                      bootSettings(1234,10))  # bootstrap with 10 repetitions

## Check a summary of the results
summary(eval.res)

## Plot them
## Not run:
plot(eval.res)

## End(Not run)
```

---

centralImputation *Fill in NA values with central statistics*

Description

This function fills in any NA value in all columns of a data frame with the statistic of centrality (given by the function `centralvalue()`) of the respective column.

Usage

```
centralImputation(data)
```

Arguments

`data`	The data frame

Value

A new data frame with no NA values

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

knnImputation, centralValue, complete.cases, na.omit

Examples

```
data(algae)
cleanAlgae <- centralImputation(algae)
summary(cleanAlgae)
```

---

centralValue *Obtain statistic of centrality*

Description

This function obtains a statistic of centrality of a variable given a sample of its values.

Usage

```
centralValue(x, ws = NULL)
```

Arguments

x	A vector of values (the sample).
ws	A vector of case weights (defaulting to NULL, i.e. no case weights).

Details

If the variable is numeric it returns de median of the given sample, if it is a factor it returns the mode. In other cases it tries to convert to a factor and then returns the mode.

Value

A number if the variable is numeric. A string with the name of the most frequent nominal value, otherwise.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

mean, median

Examples

```
# An example with numerical data
x <- rnorm(100)
centralValue(x)
# An example with nominal data
y <-
factor(sample(1:10,200,replace=TRUE),levels=1:10,labels=paste('v',1:10,sep=''))
centralValue(y)
```

---

class.eval *Calculate Some Standard Classification Evaluation Statistics*

Description

This function is able to calculate a series of classification evaluation statistics given two vectors: one with the true target variable values, and the other with the predicted target variable values.

Usage

```
class.eval(trues, preds,
          stats=if (is.null(benMtrx)) c('err') else c('err','totU'),
          benMtrx=NULL,
          allCls=levels(factor(trues)))
```

Arguments

trues	A vector or factor with the true values of the target variable.
preds	A vector or factor with the predicted values of the target variable.
stats	A vector with the names of the evaluation statistics to calculate. Possible values are "acc", "err" or "totU". This latter requires that the parameter benMtrx contains a matrix with cost/benefits for all combinations of possible predictions and true values, i.e. with dimension NC x NC, where NC is the number of classes of the classification task being handled.
benMtrx	A matrix with numeric values representing the benefits (positive values) and costs (negative values) for all combinations of predicted and true values of the nominal target variable of the task. In this context, the matrix should have the dimensions NC x NC, where NC is the number of possible class values of the

classification task. Benefits (positive values) should be on the diagonal of the matrix (situations where the true and predicted values are equal, i.e. the model predicted the correct class and thus should be rewarded for that), whilst costs (negative values) should be on all positions outside of the diagonal of the matrix (situations where the predicted value is different from the true class value and thus the model should incur on a cost for this wrong prediction).

`allCls` A vector with the possible values of the nominal target variable, i.e. a vector with the classes of the problem. The default of this parameter is to infer these values from the given vector of true class values. However, if this is a small vector (e.g. you are evaluating your model on a small test set), it may happen that not all possible class values occur in this vector and this will potentially create problems in the sub-sequent calculations. Moreover, even if the vector is not small, for highly unbalanced classification tasks, this problem may still occur. In these contexts, it is safer to specifically indicate the possible class values through this parameter.

Details

The classification evaluation statistics available through this function are "acc", "err" (that is actually the complement of "acc") and "totU".

Both "acc" and "err" are related to the proportion of accurate predictions. They are calculated as:

"acc": sum(I(t_i == p_i))/N, where t's are the true values and p's are the predictions, while I() is an indicator function given 1 if its argument is true and 0 otherwise. Note that "acc" is a value in the interval [0,1], 1 corresponding to all predictions being correct.

"err": = 1 - acc

Regards "totU" this is a metric that takes into consideration not only the fact that the predictions are correct or not, but also the costs or benefits of these predictions. As mentioned above it assumes that the user provides a fully specified matrix of costs and benefits, with benefits corresponding to correct predictions, i.e. where t_i == p_i, while costs correspond to erroneous predictions. These matrices are NC x NC square matrices, where NC is the number of possible values of the nominal target variable (i.e. the number of classes). The diagonal of these matrices corresponds to the correct predictions (t_i == p_i) and should have positive values (benefits). The positions outside of the diagonal correspond to prediction errors and should have negative values (costs). The "totU" measures the total Utility (sum of the costs and benefits) of the predictions of a classification model. It is calculated as:

"totU": sum(CB[t_i,p_i]) where CB is a cost/benefit matrix and CB[t_i,p_i] is the entry on this matrix corresponding to predicting class p_i for a true value of t_i.

Value

A named vector with the calculated statistics.

Note

1. In case you require "totU" to be calculated you must supply a cost/benefit matrix through parameter `benMtrx`.

2. If not all possible class values are present in the vector of true values in parameter `trues`, you should provide a vector with all the possible class values in parameter `allCls`.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`regr.eval`

Examples

```
## Calculating several statistics of a classification tree on the Iris data
data(iris)
idx <- sample(1:nrow(iris),100)
train <- iris[idx,]
test <- iris[-idx,]
tree <- rpartXse(Species ~ .,train)
preds <- predict(tree,test,type='class')
## Calculate the accuracy and error rate
class.eval(test$Species,preds)
## Now trying calculating the utility of the predictions
cbM <- matrix(c(10,-20,-20,-20,20,-10,-20,-10,20),3,3)
class.eval(test$Species,preds,"totU",cbM)
```

`compAnalysis` *Analyse and print the statistical significance of the differences between a set of learners.*

Description

This function analyses and shows the statistical significance results of comparing the estimated average evaluation scores of a set of learners. When you run the `experimentalComparison()` function to compare a set of learners over a set of problems you obtain estimates of their performances across these problems. This function allows you to test whether the observed differences in these estimated performances are statistically significant with a certain confidence level.

Usage

```
compAnalysis(comp, against = dimnames(comp@foldResults)[[3]][1],
             stats = dimnames(comp@foldResults)[[2]],
             datasets = dimnames(comp@foldResults)[[4]], show = T)
```

Arguments

comp	This is a compExp object (type "class?compExp" for details) that contains the results of an experimental comparison obtained through the experimentalComparison() function.
against	When you carry out this type of analysis you have to select against which learner all others will be compared to. By default this will be the first system in the alternatives you have supplied when running the experiments. This parameter allows you to specify the identifier of any other learner as the one to compare against.
stats	By default the analysis will be carried out across all evaluation statistics estimated in the experimental comparison. This parameter allows you to supply a vector with the names of the subset of statistics you wish to analyse.
datasets	By default the analysis will be carried out across all problems you have used in the experimental comparison. This parameter allows you to supply a vector with the names of the subset of problems you wish to analyse.
show	By default this function shows a table with the results of the analysis and will silently return a data structure (see section Value) with these results. If you set this parameter to False the function will not show any thing, simply returning that data structure.

Details

Independently of the experimental methodology you select (e.g. cross validation) all results you obtain with the experimentalComparison() function are estimates of the (unknown) true scores of the learners you are comparing. This function allows you to carry out a statistical test to check the statistical significance of the observed differences among the learners. Namely, the function will carry out a Wilcoxon paired test for checking the significance of the differences among the estimated average scores. The function will print the results of these tests using a set of symbols that correspond to a set of pre-defined confidence levels (essencially the standard 95% and 99% thresholds). All tests are carried out between two learners: the one indicated in the against parameter, which defaults to the first learner in the experiments (named Learn.1 on the tables); and all other learners. For each of the competitors the function will print a symbol beside its average score representing the result of the comparison against the baseline learner. If there is no symbol it means that the difference among the two learners can not be considered statistically significant with 95% confidence. If there is one symbol (either a "+" or a "-") it means the statistical confidence on the difference is between 95% and 99%. A "+" means the competitor has a larger estimated value (this can be good or bad depending on the statistic being estimated) than the baseline, whilst a "-" means the opposite. Finally, two symbols (either "++" or "–") mean that the difference is significant with more than 99% confidence.

Value

Usually this function is used to print the tables with the results of the statistical significance tests. However, the function also returns silently the information on these tables, so that you may further proccess it if you want. This means that if you assign the results of the function to some variable, you will get as a result a list with as many components as there are evaluation statistics in your experiment. For each of these list components, you will get a data frame with the results of the comparison following the same schema as the printed version.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

experimentalComparison,compExp

Examples

```
## Estimating several evaluation metrics on different variants of a
## regression tree on  a data set, using one repetition of 10-fold CV
data(swiss)

## First the user defined functions
cv.rpartXse <- function(form, train, test, ...) {
    require(DMwR)
    t <- rpartXse(form, train, ...)
    p <- predict(t, test)
    mse <- mean((p - resp(form, test))^2)
    c(nmse = mse/mean((mean(resp(form, train)) - resp(form, test))^2),
        mse = mse)
}

results <- experimentalComparison(
               c(dataset(Infant.Mortality ~ ., swiss)),
               c(variants('cv.rpartXse',se=c(0,0.5,1))),
               cvSettings(1,10,1234)
                                 )

## Testing the statistical significance of the differences
compAnalysis(results)

## Comparing against the learner with best NMSE, and only on that statistic
compAnalysis(results,against=bestScores(results)$swiss['nmse','system'],
             stats='nmse')
```

---

compExp-class *Class "compExp"*

Description

This is the main class that holds the results of experimental comparisons of a set of learners over a set of predictive tasks, using some experimental methodology.

Objects from the Class

Objects can be created by calls of the form `compExp(...)`. These objects contain information on the set of learners being compared, the set of predictive tasks being used on the comparison, the experimental settings and the overall results of all experimental comparisons.

Slots

`learners`: Object of class `"list"` : a list of objects of the class learner.

`datasets`: Object of class `"list"` : a list of objects of the class task.

`settings`: Object of class `"expSettings"` : an object belonging to one of the classes in this class union.

`foldResults`: Object of class `"array"` : a numeric array with the overall results of the experiment. This array has 4 dimensions. The first dimension are the different repetitions/iterations of the experiment; the second dimension are the evaluation statistics being estimated; the third dimension are the different learners being compared; while the fourth dimension are the predictive tasks.

Methods

plot `signature(x = "compExp", y = "missing")`: plots the results of the experiments. It can result in an over-cluttered graph if too many learners/datasets/evaluation metrics - use the subset method (see below) to overcome this.

show `signature(object = "compExp")`: shows the contents of an object in a proper way

subset `signature(x = "compExp")`: can be used to obtain a smaller compExp object containing only a subset of the information of the provided object. This method also accepts the arguments "its", "stats", "vars" and "dss". All are vectors of numbers or names corresponding to an indexing of each of the dimensions of the "foldResults" slot. They default to all values of each dimension. See "methods?subset" for further details.

summary `signature(object = "compExp")`: provides a summary of the experimental results.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`experimentalComparison`, `compAnalysis`, `rankSystems`, `bestScores`, `statScores`, `join`

Examples

```
showClass("compExp")
```

CRchart *Plot a Cumulative Recall chart*

Description

A cumulative recall chart plots the cumulative recall score against the rate of positive class predictions of a classification model.

Usage

```
CRchart(preds, trues, ...)
```

Arguments

preds	A vector containing the predictions of the model.
trues	A vector containing the true values of the class label. Must have the same dimension as preds.
...	Further parameters that are passed to the plot() function.

Details

The cumulative recall chart plots the recall against the rate of positive predictions. The latter measure the proportion of cases predicted as positivem while the former measure the proportion of positive cases signaled as such by the model.

The function uses the infra-structure provided by the ROCR package (Sing et al., 2009). This package allows us to obtain several measures of the predictive performance of models. We use it to obtain the recall and the rate of positive predictions of the predictions of a model.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Sing, T., Sander, O., Beerenwinkel, N., and Lengauer, T. (2009). *ROCR: Visualizing the performance of scoring classifiers*. R package version 1.0-4.

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

prediction, performance, CRchart

Examples

```
## A simple example with data in package ROCR
library(ROCR)
data(ROCR.simple)

## Obtaining the Cumulative Recall chart for this problem
## Not run:
CRchart(ROCR.simple$predictions,ROCR.simple$labels)

## End(Not run)
```

`crossValidation` *Run a Cross Validation Experiment*

Description

Function that performs a cross validation experiment of a learning system on a given data set. The function is completely generic. The generality comes from the fact that the function that the user provides as the system to evaluate, needs in effect to be a user-defined function that takes care of the learning, testing and calculation of the statistics that the user wants to estimate through cross validation.

Usage

```
crossValidation(sys, ds, sets, itsInfo = F)
```

Arguments

`sys`	`sys` is an object of the class `learner` representing the system to evaluate.
`ds`	`ds` is an object of the class `dataset` representing the data set to be used in the evaluation.
`sets`	`sets` is an object of the class `cvSettings` representing the cross validation experimental settings to use.
`itsInfo`	Boolean value determining whether the object returned by the function should include as an attribute a list with as many components as there are iterations in the experimental process, with each component containing information that the user-defined function decides to return on top of the standard error statistics. See the Details section for more information.

Details

The idea of this function is to carry out a cross validation experiment of a given learning system on a given data set. The goal of this experiment is to estimate the value of a set of evaluation statistics by means of cross validation. k-Fold cross validation estimates are obtained by randomly partition the given data set into k equal size sub-sets. Then a learn+test process is repeated k times. At each iteration one of the k partitions is left aside as test set and the model is obtained with a training set formed by the remaining k-1 partitions. The process is repeated leaving each time one of the

partitions aside as test set. In the end the average of the k scores obtained on each iteration is the cross validation estimate.

It is the user responsibility to decide which statistics are to be evaluated on each iteration and how they are calculated. This is done by creating a function that the user knows it will be called by this cross validation routine at each iteration of the cross validation process. This user-defined function must assume that it will receive in the first 3 arguments a formula, a training set and a testing set, respectively. It should also assume that it may receive any other set of parameters that should be passed towards the learning algorithm. The result of this user-defined function should be a named vector with the values of the statistics to be estimated obtained by the learner when trained with the given training set, and tested on the given test set. See the Examples section below for an example of these functions.

If the `itsInfo` parameter is set to the value TRUE then the `hldRun` object that is the result of the function will have an attribute named itsInfo that will contain extra information from the individual repetitions of the hold out process. This information can be accessed by the user by using the function `attr()`, e.g. attr(returnedObject,'itsInfo'). For this information to be collected on this attribute the user needs to code its user-defined functions in a way that it returns the vector of the evaluation statistics with an associated attribute named itInfo (note that it is "itInfo" and not "itsInfo" as above), which should be a list containing whatever information the user wants to collect on each repetition. This apparently complex infra-structure allows you to pass whatever information you which from each iteration of the experimental process. A typical example is the case where you want to check the individual predictions of the model on each test case of each repetition. You could pass this vector of predictions as a component of the list forming the attribute itInfo of the statistics returned by your user-defined function. In the end of the experimental process you will be able to inspect/use these predictions by inspecting the attribute itsInfo of the `cvRun` object returned by the `crossValidation()` function. See the Examples section on the help page of the function `holdout()` for an illustration of this potentiality.

Value

The result of the function is an object of class `cvRun`.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`experimentalComparison`, `cvRun`,`cvSettings`, `monteCarlo`, `holdOut`, `loocv`, `bootstrap`

Examples

```
## Estimating the mean absolute error and the normalized mean squared
## error of rpart on the swiss data, using one repetition of 10-fold CV
data(swiss)
```

```
## First the user defined function (note: can have any name)
cv.rpart <- function(form, train, test, ...) {
    require(rpart)
    model <- rpart(form, train, ...)
    preds <- predict(model, test)
    regr.eval(resp(form, test), preds,
              stats=c('mae','nmse'), train.y=resp(form, train))
}

## Now the evaluation
eval.res <- crossValidation(learner('cv.rpart',pars=list()),
                            dataset(Infant.Mortality ~ ., swiss),
                            cvSettings(1,10,1234))

## Check a summary of the results
summary(eval.res)

## Plot them
## Not run:
plot(eval.res)

## End(Not run)
```

---

cvRun-class *Class "cvRun"*

Description

This is the class of the objects holding the results of a cross validation experiment.

Objects from the Class

Objects can be created by calls of the form `cvRun(...)`. The objects contain information on the learner evaluated in the CV experiment, the predictive task that was used, the cross validation settings, and the results of the experiment.

Slots

`learner`: Object of class `"learner"`

`dataset`: Object of class `"task"`

`settings`: Object of class `"cvSettings"`

`foldResults`: Object of class `"matrix"` with the results of the experiment. The rows represent the different iterations of the experiment while the columns the different statistics evaluated on each iteration.

Methods

plot `signature(x = "cvRun", y = "missing")`: method used to visualize the results of the cross validation experiment.

summary `signature(object = "cvRun")`: method used to obtain a summary of the results of the cross validation experiment.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

crossValidation, cvSettings, hldRun,loocvRun,mcRun,bootRun,compExp

Examples

```
showClass("cvRun")
```

---

cvSettings-class *Class "cvSettings"*

Description

This class of objects contains the information describing a cross validation experiment, i.e. its settings.

Objects from the Class

Objects can be created by calls of the form `cvSettings(...)`. These objects include information on the number of repetitions of the experiment, the number of folds, the random number generator seed and whether the sampling should or not be stratefied.

Slots

`cvReps`: Object of class `"numeric"` indicating the number of repetitions of the N folds CV experiment (defaulting to 1).

`cvFolds`: Object of class `"numeric"` with the number of folds on each CV experiment (defaulting to 10).

`cvSeed`: Object of class `"numeric"` with the random number generator seed (defaulting to 1234).

`strat`: Object of class `"logical"` indicating whether the sampling should or not be stratified (defaulting to F).

Extends

Class "expSettings", directly.

Methods

show `signature(object = "cvSettings")`: method used to show the contents of a cvSettings object.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187). http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

cvRun, mcSettings, loocvSettings, hldSettings,bootSettings, expSettings

Examples

```
showClass("cvSettings")
```

---

dataset-class *Class "dataset"*

Description

This is the class of objects that represent all necessary information on a predictive task. This class extends the task class by adding the data frame with the data of the predictive task.

Objects from the Class

Objects can be created by calls of the form `dataset(...)`. The objects include information on the name of the predictive task, the formula and the data frame with the data used in the task.

Slots

`formula`**:** Object of class `"formula"` containing the formula representing the predictive task

`data`**:** Object coercible to class `"data.frame"` containing the data of the problem

`name`**:** Object of class `"character"` containing an internal name of the task

Extends

Class "task", directly.

Methods

show `signature(object = "dataset")`: method used to show the contents of a dataset object.

Author(s)

Luis Torgo (ltorgo@dcc.fc.up.pt)

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`task`, `learner`

Examples

```
showClass("dataset")
```

---

dist.to.knn *An auxiliary function of* `lofactor()`

Description

This function returns an object in which columns contain the indices of the first k neighbors followed by the distances to each of these neighbors.

Usage

```
dist.to.knn(dataset, neighbors)
```

Arguments

`dataset`	A data set that will be internally coerced into a matrix.
`neighbors`	The number of neighbours.

Details

This function is strongly based on the code provided by Acuna et. al. (2009) for the previously available `dprep` package.

Value

A matrix

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Acuna, E., and Members of the CASTLE group at UPR-Mayaguez, (2009). *dprep: Data preprocessing and visualization functions for classification*. R package version 2.1.

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

lofactor

---

dsNames *Obtain the name of the data sets involved in an experimental comparison*

Description

This function produces a vector with the names of the datasets involved in an experimental comparison

Usage

```
dsNames(res)
```

Arguments

res	This is a compExp object (type "class?compExp" for details) that contains the results of an experimental comparison obtained through the experimentalComparison() function.

Value

A vector of strings with the names of the datasets

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

learnerNames, statNames,experimentalComparison

`experimentalComparison` *Carry out Experimental Comparisons Among Learning Systems*

Description

This function can be used to carry out different types of experimental comparisons among learning systems on a set of predictive tasks. This is a generic function that should work with any learning system provided a few assumptions are met. The function implements different experimental methodologies, namely: cross validation, leave one out cross validation, hold-out, monte carlo simulations and bootstrap.

Usage

```
experimentalComparison(datasets, systems, setts, ...)
```

Arguments

`datasets`	This is a list of objects of class `dataset`, containing the data sets that will be used in the comparison.
`systems`	This is a list of objects of class `learner`, containing the learning systems that will be used in the comparison.
`setts`	This is an object belonging to any of the sub-classes of the virtual class `expSettings`. It is the class of this object that determines the type of experimental comparison that will be carried out. See section Details for the possible values.
`...`	Other parameter settings that are to be passed to the functions actually carrying out the experiments (e.g. `crossValidation`, etc.).

Details

The goal of this function is to allow to carry out different types of experimental comparisons between a set of learning systems over a set of predictive tasks. The idea is that all learning system will be compared over the same data partitions for each of the tasks thus ensuring fare comparisons and also allowing for proper statistical tests of significance of the observed differences, to be carried out.

Currently, the function allows for 5 different types of experimental comparisons to be carried out. These different types are in effect, different estimation methods for the target evaluation statistics that are to be used in evaluation the different learners over the tasks. The method to be used is determined by the class of the object provided in the argument `setts`. The following are the possibilities:

"Cross validation": this type of estimates can be obtained by providing in the `setts` argument and object of class `cvSettings`. More details on this type of experiments can be obtained in the help page of the function `crossValidation`.

"Leave one out cross validation": this type of estimates can be obtained by providing in the `setts` argument and object of class `loocvSettings`. More details on this type of experiments can be obtained in the help page of the function `loocv`.

"Hold out": this type of estimates can be obtained by providing in the `setts` argument and object of class `hldSettings`. More details on this type of experiments can be obtained in the help page of the function `holdOut`.

"Monte carlo": this type of estimates can be obtained by providing in the `setts` argument and object of class `mcSettings`. More details on this type of experiments can be obtained in the help page of the function `monteCarlo`.

"Bootstrap": this type of estimates can be obtained by providing in the `setts` argument and object of class `bootSettings`. More details on this type of experiments can be obtained in the help page of the function `bootstrap`.

Value

The result of the function is an object of class `compExp` (type "class?compExp" for details).

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`variants`,`bestScores`,`rankSystems`, `compAnalysis`, `crossValidation`, `loocv`, `holdOut`, `monteCarlo`, `bootstrap`, `compExp`, `cvSettings`, `hldSettings`, `mcSettings`, `loocvSettings`, `bootSettings`

Examples

```
## Estimating several evaluation metrics on different variants of a
## regression tree and of a SVM, on  two data sets, using one repetition
## of 10-fold CV
data(swiss)
data(mtcars)

## First the user defined functions
cv.rpartXse <- function(form, train, test, ...) {
    require(DMwR)
    t <- rpartXse(form, train, ...)
    p <- predict(t, test)
    mse <- mean((p - resp(form, test))^2)
    c(nmse = mse/mean((mean(resp(form, train)) - resp(form, test))^2),
        mse = mse)
}

results <- experimentalComparison(
               c(dataset(Infant.Mortality ~ ., swiss),
                 dataset(mpg ~ ., mtcars)),
               c(variants('cv.rpartXse',se=c(0,0.5,1))),
```

```
                cvSettings(1,10,1234)
                                    )
## Check a summary of the results
summary(results)

bestScores(results)

## Check the statistical significance against the best model in terms of
## nmse for the swiss data set
compAnalysis(results,against='cv.rpartXse.v3',stats='nmse',datasets='swiss')

## Plot them
## Not run:
plot(results)

## End(Not run)
```

`expSettings-class` *Class "expSettings"*

Description

This is a class union formed by the classes cvSettings, mcSettings, hldSettings, loocvSettings and bootSettings

Objects from the Class

A virtual Class: No objects may be created from it.

Methods

No methods defined with class "expSettings" in the signature.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

cvSettings, mcSettings, loocvSettings, hldSettings,bootSettings

Examples

```
showClass("expSettings")
```

getFoldsResults *Obtain the results on each iteration of a learner*

Description

This function allows you to obtain the scores obtained by a learner on the different iterations that form an experimental comparison. These scores are obtained for a particular data set of this comparison.

Usage

```
getFoldsResults(results,learner,dataSet)
```

Arguments

results	This is a compExp object (type "class?compExp" for details) that contains the results of an experimental comparison obtained through the experimentalComparison() function.
learner	This is the string that identifies the learner.
dataSet	The string that identifies the data set for which you want to get the scores.

Value

The result of the function is a matrix with as many columns as there are evaluation statistics in the experimental comparison, and with as many rows as there are iterations in this experiment. The values on this matrix are the scores of the learner for respective statistic on the different iterations of the process.

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

getSummaryResults, experimentalComparison

Examples

```
## Estimating several evaluation metrics on different variants of a
## regression tree on  a data set, using one repetition of 10-fold CV
data(swiss)

## First the user defined functions
cv.rpartXse <- function(form, train, test, ...) {
    require(DMwR)
    t <- rpartXse(form, train, ...)
    p <- predict(t, test)
```

```
    mse <- mean((p - resp(form, test))^2)
    c(nmse = mse/mean((mean(resp(form, train)) - resp(form, test))^2),
        mse = mse)
}

results <- experimentalComparison(
               c(dataset(Infant.Mortality ~ ., swiss)),
               c(variants('cv.rpartXse',se=c(0,0.5,1))),
               cvSettings(1,10,1234)
                                 )

## Get the scores of a specific learner
getFoldsResults(results,'cv.rpartXse.v1','swiss')

## Get the scores of the learner that obtained the best NMSE on the
## swiss data set
getFoldsResults(results,bestScores(results)$swiss['nmse','system'],'swiss')
```

`getSummaryResults` *Obtain a set of descriptive statistics of the results of a learner*

Description

This function provides a set of descriptive statistics for each evaluation metric that is estimated on an experimental comparison. These statistics are obtained for a particular learner, and for one of the prediction problems involved in the experimental comparison.

Usage

```
getSummaryResults(results,learner,dataSet)
```

Arguments

`results`	This is a `compExp` object (type "class?compExp" for details) that contains the results of an experimental comparison obtained through the `experimentalComparison()` function.
`learner`	This is the string that identifies the learner.
`dataSet`	The string that identifies the data set for which you want to get the scores.

Value

The function returns a matrix with the rows representing summary statistics of the scores obtained by the model on the different iterations, and the columns representing the evaluation statistics estimated in the experiment.

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

getFoldsResults, experimentalComparison

Examples

```
## Estimating several evaluation metrics on different variants of a
## regression tree on  a data set, using one repetition of 10-fold CV
data(swiss)

## First the user defined functions
cv.rpartXse <- function(form, train, test, ...) {
    require(DMwR)
    t <- rpartXse(form, train, ...)
    p <- predict(t, test)
    mse <- mean((p - resp(form, test))^2)
    c(nmse = mse/mean((mean(resp(form, train)) - resp(form, test))^2),
        mse = mse)
}

results <- experimentalComparison(
               c(dataset(Infant.Mortality ~ ., swiss)),
               c(variants('cv.rpartXse',se=c(0,0.5,1))),
               cvSettings(1,10,1234)
                                )

## Get the statistics of a specific learner
getSummaryResults(results,'cv.rpartXse.v1','swiss')

## Get the statistics of the learner that obtained the best NMSE on the
## swiss data set
getSummaryResults(results,bestScores(results)$swiss['nmse','system'],'swiss')
```

---

getVariant *Obtain the learner associated with an identifier within a comparison*

Description

The goal of this function is to obtain the `learner` object corresponding to a certain provided identifier in the context of an experimental comparison. This function finds its use after you run an experimental comparison using the infrastructure provided by the `experimentalComparison()` function. This latter function returns an object that contains the results of the several alternative methods that you have decided to compare. Each of these methods has an associated identifier (a string). This function allows you to obtain the learner object (which gives you access to several information necessary to run the associated algorithm), corresponding to its identifier.

Usage

```
getVariant(var, ExpsData)
```

Arguments

var	This is the string that identifies the learner you which to "extract".
ExpsData	This is a compExp object (type "class?compExp" for details) that contains the results of an experimental comparison obtained through the experimentalComparison() function.

Details

Most results analysis functions of the experimental infrastructure provided by the DMwR package use the identifiers generated either by calls to the variants function or names given by the user. Each of these names is associated with a concrete learning algorithm implemented by a R function and also to a set of parameter settings of this function. The function getVariant allows you to obtain all this information, in the form of a learner object, which is associated to an identifier within a compExp object.

Value

The result of this function is an object of class learner (type "class?learner" for details).

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

variants, experimentalComparison

Examples

```
## Estimating several evaluation metrics on different variants of a
## regression tree on  a data set, using one repetition of 10-fold CV
data(swiss)

## First the user defined functions
cv.rpartXse <- function(form, train, test, ...) {
    require(DMwR)
    t <- rpartXse(form, train, ...)
    p <- predict(t, test)
    mse <- mean((p - resp(form, test))^2)
    c(nmse = mse/mean((mean(resp(form, train)) - resp(form, test))^2),
        mse = mse)
}

results <- experimentalComparison(
               c(dataset(Infant.Mortality ~ ., swiss)),
```

```
                c(variants('cv.rpartXse',se=c(0,0.5,1))),
                cvSettings(1,10,1234)
                                )
## Get the best scores
bestScores(results)

# Obtain the settings corresponding to one of the variants
getVariant('cv.rpartXse.v1',results)

# Obtain the settings of the learner that got the best NMSE score on the
# swiss data set
getVariant(bestScores(results)$swiss['nmse','system'],results)
```

`growingWindowTest` *Obtain the predictions of a model using a growing window learning approach.*

Description

This function implements the growing window learning method that is frequently used in time series forecasting. The function allows applying this methodology to any modelling technique. The function returns the predictions of this technique, when applied using a growing window approach, for the given test set.

Usage

```
growingWindowTest(learner, form, train, test, relearn.step = 1, verbose = T)
```

Arguments

`learner`	This is an object of the class `learner` (type "class?learner" for details) representing the system to evaluate.
`form`	A formula describing the prediction problem.
`train`	A data frame with the initial training data. The size of this training set will also determine the size of the sliding window.
`test`	A data frame with the test set for which we want predictions.
`relearn.step`	A number indicating the number of test cases until a new model is re-learned by sliding the training window to cases that are nearest to the current test case.
`verbose`	A boolean determining the level of verbosity of the function.

Details

The growing window is a method frequently used to handle time series prediction problems. The basic motivation is that as time goes by the data gets "old" and thus the models should be re-learned to re-adjust for "fresher" data. This function implements this general idea for any modelling technique.

The function receives an initial training set. Using this initial set a first model is obtained with the supplied modelling technique. This model is applied to obtain predictions for the first `relearn.step` test cases. Afterwards a new model is obtained by adding the more recent training cases to the previous training set. This new training set will have a larger size than the initially provided training set. It will consist of the initial training set, plus the following `relearn.step` observations. This second model is again used to obtain predictions for another set of `relearn.step` test cases. The growing process keeps going until we obtain predictions for all provided test cases.

Value

A vector with the predictions for the test set. Note that if the target variable is a factor this vector will also be a factor.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.liaad.up.pt/~ltorgo/DataMiningWithR

See Also

slidingWindowTest,monteCarlo

Examples

```
data(swiss)

## Obtain the predictions of model rpartXse() for the last 22 rows of
## the swiss data set, when used with a growing window of 25 cases with
## a relearning step of 3

## The base learner used in the experiment
learnAndTest.rpartXse <- function(form, train, test, ...) {
    model <- rpartXse(form, train, ...)
    predict(model, test)
}

preds <- growingWindowTest(learner('learnAndTest.rpartXse',pars=list(se=0.5)),
                           Infant.Mortality ~ .,
                           swiss[1:25,],
                           swiss[26:nrow(swiss),],
                           3)

## Some statistics of these predictions
regr.eval(swiss[26:nrow(swiss),'Infant.Mortality'],preds,stats = c("mae", "mse", "rmse"))
```

GSPC *A set of daily quotes for SP500*

Description

This is a `xts` object containing the daily quotes of the SP500 sotck index from 1970-01-02 till 2009-09-15 (10,022 daily sessions). For each day information is given on the Open, High, Low and Close prices, and also for the Volume and Adjusted close price.

Usage

```
GSPC
```

Format

A `xts` object with a data matrix with 10,022 rows and 6 columns.

Source

Yahoo Finance http://finance.yahoo.com/

hldRun-class *Class "hldRun"*

Description

This is the class of the objects storing the results of a hold out experiment.

Objects from the Class

Objects can be created by calls of the form `hldRun(...)`. The objects contain information on the learner evaluated in the holdout experiment, the predictive task that was used, the holdout settings, and the results of the experiment.

Slots

`learner`: Object of class `"learner"`

`dataset`: Object of class `"task"`

`settings`: Object of class `"hldSettings"`

`foldResults`: Object of class `"matrix"` with the results of the experiment. The rows represent the different repetitions of the experiment while the columns the different statistics evaluated on each iteration.

Methods

plot `signature(x = "hldRun", y = "missing")`: method used to visualize the results of the holdout experiment.

summary `signature(object = "hldRun")`: method used to obtain a summary of the results of the holdout experiment.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`hldSettings`, `cvRun`,`loocvRun`,`mcRun`,`bootRun`,`compExp`

Examples

```
showClass("hldRun")
```

---

hldSettings-class *Class "hldSettings"*

Description

This class of objects contains the information describing a hold out experiment, i.e. its settings.

Objects from the Class

Objects can be created by calls of the form `hldSettings(...)`. The objects contain information on the number of repetitions of the hold out experiment, the percentage of the given data to set as hold out test set, the random number generator seed and information on whether stratified sampling should be used.

Slots

`hldReps`**:** Object of class `"numeric"` indicating the number of repetitions of the N folds CV experiment (defaulting to 1).

`hldSz`**:** Object of class `"numeric"` with the percentage (a number between 0 and 1) of cases to use as hold out (defaulting to 0.3).

`hldSeed`**:** Object of class `"numeric"` with the random number generator seed (defaulting to 1234).

`strat`**:** Object of class `"logical"` indicating whether the sampling should or not be stratefied (defaulting to F).

Extends

Class "expSettings", directly.

Methods

show `signature(object = "hldSettings")`: method used to show the contents of a hldSettings object.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

hldRun, mcSettings, loocvSettings, cvSettings,bootSettings, expSettings

Examples

```
showClass("hldSettings")
```

---

holdOut *Runs a Hold Out experiment*

Description

Function that performs a hold out experiment of a learning system on a given data set. The function is completely generic. The generality comes from the fact that the function that the user provides as the system to evaluate, needs in effect to be a user-defined function that takes care of the learning, testing and calculation of the statistics that the user wants to estimate using the hold out method.

Usage

```
holdOut(sys, ds, sets, itsInfo = F)
```

Arguments

sys	sys is an object of the class learner representing the system to evaluate.
ds	ds is an object of the class dataset representing the data set to be used in the evaluation.
sets	sets is an object of the class cvSettings representing the cross validation experimental settings to use.

`itsInfo` Boolean value determining whether the object returned by the function should include as an attribute a list with as many components as there are iterations in the experimental process, with each component containing information that the user-defined function decides to return on top of the standard error statistics. See the Details section for more information.

Details

The idea of this function is to carry out a hold out experiment of a given learning system on a given data set. The goal of this experiment is to estimate the value of a set of evaluation statistics by means of the hold out method. Hold out estimates are obtained by randomly dividing the given data set in two separate partitions, one that is used for obtaining the prediction model and the other for testing it. This learn+test process is repeated k times. In the end the average of the k scores obtained on each repetition is the hold out estimate.

It is the user responsibility to decide which statistics are to be evaluated on each iteration and how they are calculated. This is done by creating a function that the user knows it will be called by this hold out routine at each repetition of the learn+test process. This user-defined function must assume that it will receive in the first 3 arguments a formula, a training set and a testing set, respectively. It should also assume that it may receive any other set of parameters that should be passed towards the learning algorithm. The result of this user-defined function should be a named vector with the values of the statistics to be estimated obtained by the learner when trained with the given training set, and tested on the given test set. See the Examples section below for an example of these functions.

If the `itsInfo` parameter is set to the value TRUE then the `hldRun` object that is the result of the function will have an attribute named itsInfo that will contain extra information from the individual repetitions of the hold out process. This information can be accessed by the user by using the function `attr()`, e.g. attr(returnedObject,'itsInfo'). For this information to be collected on this attribute the user needs to code its user-defined functions in a way that it returns the vector of the evaluation statistics with an associated attribute named itInfo (note that it is "itInfo" and not "itsInfo" as above), which should be a list containing whatever information the user wants to collect on each repetition. This apparently complex infra-structure allows you to pass whatever information you which from each iteration of the experimental process. A typical example is the case where you want to check the individual predictions of the model on each test case of each repetition. You could pass this vector of predictions as a component of the list forming the attribute itInfo of the statistics returned by your user-defined function. In the end of the experimental process you will be able to inspect/use these predictions by inspecting the attribute itsInfo of the `hldRun` object returned by the `holdOut()` function. See the Examples section for an illustration of this potentiality.

Value

The result of the function is an object of class `hldRun`.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

experimentalComparison, hldRun,hldSettings, monteCarlo, crossValidation, loocv, bootstrap

Examples

```
## Estimating the mean absolute error and the normalized mean squared
## error of rpart on the swiss data, using 10 repetitions of 70%-30%
## Hold Out experiment
data(swiss)

## First the user defined function (note: can have any name)
hld.rpart <- function(form, train, test, ...) {
    require(rpart)
    model <- rpart(form, train, ...)
    preds <- predict(model, test)
    regr.eval(resp(form, test), preds,
              stats=c('mae','nmse'), train.y=resp(form, train))
}

## Now the evaluation
eval.res <- holdOut(learner('hld.rpart',pars=list()),
                            dataset(Infant.Mortality ~ ., swiss),
                            hldSettings(10,0.3,1234))

## Check a summary of the results
summary(eval.res)

## Plot them
## Not run:
plot(eval.res)

## End(Not run)

## An illustration of the use of the itsInfo parameter.
## In this example the goal is to be able to check values predicted on
## each iteration of the experimental process (e.g. checking for extreme
## values)

## We need a different user-defined function that exports this
## information as an attribute
hld.rpart <- function(form, train, test, ...) {
    require(rpart)
    model <- rpart(form, train, ...)
    preds <- predict(model, test)
    eval.stats <- regr.eval(resp(form, test), preds,
                            stats=c('mae','nmse'),
                            train.y=resp(form,train))
    structure(eval.stats,itInfo=list(predictions=preds))
}

## Now lets run the experimental comparison
eval.res <- holdOut(learner('hld.rpart',pars=list()),
```

```
                            dataset(Infant.Mortality ~ ., swiss),
                            hldSettings(10,0.3,1234),
                            itsInfo=TRUE)

## getting the information with the predictions for all 10 repetitions
info <- attr(eval.res,'itsInfo')
## checking the predictions on the 5th repetition
info[[5]]
```

`join` *Merging several* `compExp` *class objects*

Description

This function can be used to join several `compExp` class objects into a single object. The merge is carried out assuming there there is something in common between the objects (e.g. all use the same learners on different data sets), and that the user specifies which property should be used for the merging process.

Usage

```
join(..., by = "datasets")
```

Arguments

`...`	The `compExp` class object names separated by commas
`by`	The dimension of the `compExp` class objects that should be used for the merge. All objects should have the same values on this dimension.

Details

The objects of class `compExp` (type "class?compExp" for details) contain several information on the results of an experimental comparison between several prediction models on several data sets. These experiments are carried out with the function `experimentalComparison()`. One of the "slots" of the objects of class `compExp` contains the actual results of the experiment on the different repetitions that were carried out. This slot is an array with four dimensions: "iterations","statistics","variants","datasets", in this order. This function allows the user the merge several objects of this class according to one of these four dimensions. Example uses of this function is a user that carries out a similar experiment (i.e. with the same experimental settings) on the same data sets twice, each time with a different set of learners being compared. This user might be interested in mergind the two `compExp` objects resulting from these experiments into a single object for comparing the results across all learners. This use should then use this function to join the two objects by "variants". Another example would be a set up where the same experiment with a set of learners was repeated with different sets of data sets. All the resulting objects would be merged by "datasets" to obtain a single results object.

You should note that the merging is only possible if all objects share the same experimental settings. Obviously, it only makes sense to merge several objects into a single one by some dimension "x" if all other dimensions are equal.

Value

The result of this function is a compExp object.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

experimentalComparison, compExp, subset

Examples

```
## Run some experiments with the swiss data and tow different prediction models
data(swiss)

## First the user defined functions for obtaining the two models
cv.rpart <- function(form, train, test, ...) {
    model <- rpartXse(form, train, ...)
    preds <- predict(model, test)
    regr.eval(resp(form, test), preds,
              stats=c('mae','nmse'), train.y=resp(form, train))
}
cv.lm <- function(form, train, test, ...) {
    model <- lm(form, train, ...)
    preds <- predict(model, test)
    regr.eval(resp(form, test), preds,
              stats=c('mae','nmse'), train.y=resp(form, train))
}
## Now the evaluation of the two models, which will be done separately
## just to illustrate the use of the join() function afterward
exp1 <- experimentalComparison(
               c(dataset(Infant.Mortality ~ ., swiss)),
               c(variants('cv.rpart',se=c(0,0.5,1))),
               cvSettings(1,10,1234))
exp2 <- experimentalComparison(
               c(dataset(Infant.Mortality ~ ., swiss)),
               c(variants('cv.lm')),
               cvSettings(1,10,1234))

## Now the examples of the join

## joining the two experiments by variants (i.e. models)
all <- join(exp1,exp2,by='variants')
bestScores(all)   # check the best results
```

```
## an example including also subsetting
bestScores(join(subset(exp1,stats='mae'),subset(exp2,stats='mae'),
                by='variants'))
```

---

kNN *k-Nearest Neighbour Classification*

Description

This function provides a formula interface to the existing knn() function of package class. On top of this type of convinient interface, the function also allows normalization of the given data.

Usage

```
kNN(form, train, test, norm = T, norm.stats = NULL, ...)
```

Arguments

form	An object of the class formula describing the functional form of the classification model.
train	The data to be used as training set.
test	The data set for which we want to obtain the k-NN classification, i.e. the test set.
norm	A boolean indicating whether the training data should be previously normalized before obtaining the k-NN predictions (defaults to TRUE).
norm.stats	This argument allows the user to supply the centrality and spread statistics that will drive the normalization. If not supplied they will default to the statistics used in the function scale(). If supplied they should be a list with two components, each beig a vector with as many positions as there are columns in the data set. The first vector should contain the centrality statistics for each column, while the second vector should contain the spread statistc values.
...	Any other parameters that will be forward to the knn() function of package class.

Details

This function is essentially a convenience function that provides a formula-based interface to the already existing knn() function of package class. On top of this type of interface it also incorporates some facilities in terms of normalization of the data before the k-nearest neighbour classification algorithm is applied. This algorithm is based on the distances between observations, which are known to be very sensitive to different scales of the variables and thus the usefulness of normalization.

Value

The return value is the same as in the knn() function of package class. This is a factor of classifications of the test set cases.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

knn, knn1, knn.cv

Examples

```
## A small example with the IRIS data set
data(iris)

## Split in train + test set
idxs <- sample(1:nrow(iris),as.integer(0.7*nrow(iris)))
trainIris <- iris[idxs,]
testIris <- iris[-idxs,]

## A 3-nearest neighbours model with no normalization
nn3 <- kNN(Species ~ .,trainIris,testIris,norm=FALSE,k=3)

## The resulting confusion matrix
table(testIris[,'Species'],nn3)

## Now a 5-nearest neighbours model with normalization
nn5 <- kNN(Species ~ .,trainIris,testIris,norm=TRUE,k=5)

## The resulting confusion matrix
table(testIris[,'Species'],nn5)
```

---

knneigh.vect *An auxiliary function of* lofactor()

Description

Function that returns the distance from a vector x to its k-nearest-neighbors in the matrix data

Usage

```
knneigh.vect(x, data, k)
```

Arguments

x	An observation.
data	A data set that will be internally coerced into a matrix.
k	The number of neighbours.

Details

This function is strongly based on the code provided by Acuna et. al. (2009) for the previously available dprep package.

Value

A vector.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Acuna, E., and Members of the CASTLE group at UPR-Mayaguez, (2009). *dprep: Data preprocessing and visualization functions for classification*. R package version 2.1.

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

lofactor

---

knnImputation *Fill in NA values with the values of the nearest neighbours*

Description

Function that fills in all NA values using the k Nearest Neighbours of each case with NA values. By default it uses the values of the neighbours and obtains an weighted (by the distance to the case) average of their values to fill in the unknows. If meth='median' it uses the median/most frequent value, instead.

Usage

```
knnImputation(data, k = 10, scale = T, meth = "weighAvg",
              distData = NULL)
```

Arguments

data	A data frame with the data set
k	The number of nearest neighbours to use (defaults to 10)
scale	Boolean setting if the data should be scale before finding the nearest neighbours (defaults to T)
meth	String indicating the method used to calculate the value to fill in each NA. Available values are 'median' or 'weighAvg' (the default).
distData	Optionally you may sepecify here a data frame containing the data set that should be used to find the neighbours. This is usefull when filling in NA values on a test set, where you should use only information from the training set. This defaults to NULL, which means that the neighbours will be searched in data

Details

This function uses the k-nearest neighbours to fill in the unknown (NA) values in a data set. For each case with any NA value it will search for its k most similar cases and use the values of these cases to fill in the unknowns.

If meth='median' the function will use either the median (in case of numeric variables) or the most frequent value (in case of factors), of the neighbours to fill in the NAs. If meth='weighAvg' the function will use a weighted average of the values of the neighbours. The weights are given by exp(-dist(k,x) where dist(k,x) is the euclidean distance between the case with NAs (x) and the neighbour k.

Value

A data frame without NA values

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

centralImputation, centralValue, complete.cases, na.omit

Examples

```
data(algae)
cleanAlgae <- knnImputation(algae)
summary(cleanAlgae)
```

learner-class *Class "learner"*

Description

Objects of the class learner represent learning systems that can be used in the routines designed to carry out experimental comparisons within the DMwR package.

Objects from the Class

Objects can be created by calls of the form `learner( ...)`. The objects contain information on the R function implementing the learning algorithm, and also a list of arguments with respective values, that should be used when calling that function.

Slots

`func`: A character string containing the name of the R function that implements the learning algorithm used by the learner object.

`pars`: A named list containing the parameters and respective values to be used when calling the learner (defaulting to the empty list).

Methods

show `signature(object = "learner")`: method used to show the contents of a learner object.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

task, dataset, runLearner

Examples

```
showClass("learner")
```

`learnerNames` *Obtain the name of the learning systems involved in an experimental comparison*

Description

This function produces a vector with the names of the learners that were evaluated in an experimental comparison.

Usage

```
learnerNames(res)
```

Arguments

`res`	This is a `compExp` object (type "class?compExp" for details) that contains the results of an experimental comparison obtained through the `experimentalComparison()` function.

Value

A vector of strings with the names of the learners

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`dsNames`, `statNames`,`experimentalComparison`

`LinearScaling` *Normalize a set of continuous values using a linear scaling*

Description

Function for normalizing the range of values of a continuous variable using a linear scaling within the range of the variable.

Usage

```
LinearScaling(x, mx = max(x, na.rm = T), mn = min(x, na.rm = T))
```

Arguments

x	A vector with numeric values
mx	The maximum value of the continuous variable being normalized (defaults to the maximum of the values in x).
mn	The minimum value of the continuous variable being normalized (defaults to the minimum of the values in x).

Details

The linear scaling normalization consist in transforming the value x into

(x - MIN) / (MAX - MIN)

Value

An object with the same dimensions as x but with the values normalized

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

scale, SoftMax, ReScaling

Examples

```
## A simple example with the algae data set
summary(LinearScaling(algae[,'NO3']))
summary(algae[,'NO3'])
```

---

lofactor *An implementation of the LOF algorithm*

Description

This function obtain local outlier factors using the LOF algorithm. Namely, given a data set it produces a vector of local outlier factors for each case.

Usage

```
lofactor(data, k)
```

Arguments

data	A data set that will be internally coerced into a matrix.
k	The number of neighbours that will be used in the calculation of the local outlier factors.

Details

This function re-implements the code previously made available in the dprep package (Acuna et. al., 2009) that was removed from CRAN. This code in turn is an implementation of the LOF method by Breunig et. al. (2000). See this reference to understand the full details on how these local outlier factors are calculated for each case in a data set.

Value

The function returns a vector of local outlier factors (numbers). This vector has as many values as there are rows in the original data set.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Acuna, E., and Members of the CASTLE group at UPR-Mayaguez, (2009). *dprep: Data preprocessing and visualization functions for classification*. R package version 2.1.

Breunig, M., Kriegel, H., Ng, R., and Sander, J. (2000). *LOF: identifying density-based local outliers*. In ACM Int. Conf. on Management of Data, pages 93-104.

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

Examples

```
data(iris)
lof.scores <- lofactor(iris[,-5],10)
```

---

loocv *Run a Leave One Out Cross Validation Experiment*

Description

Function that performs a leave one out cross validation (loocv) experiment of a learning system on a given data set. The function is completely generic. The generality comes from the fact that the function that the user provides as the system to evaluate, needs in effect to be a user-defined function that takes care of the learning, testing and calculation of the statistics that the user wants to estimate through loocv.

Usage

```
loocv(sys, ds, sets, itsInfo = F, verbose = F)
```

Arguments

sys	sys is an object of the class learner representing the system to evaluate.
ds	ds is an object of the class dataset representing the data set to be used in the evaluation.
sets	sets is an object of the class cvSettings representing the cross validation experimental settings to use.
itsInfo	Boolean value determining whether the object returned by the function should include as an attribute a list with as many components as there are iterations in the experimental process, with each component containing information that the user-defined function decides to return on top of the standard error statistics. See the Details section for more information.
verbose	A boolean value controlling the level of output of the function execution, defaulting to F

Details

The idea of this function is to carry out a leave one out cross validation (LOOCV) experiment of a given learning system on a given data set. The goal of this experiment is to estimate the value of a set of evaluation statistics by means of LOOCV. This type of estimates are obtained by carrying out N repetitions of a lear+test cycle, where N is the size of the given data set. On each repetition one of the N observations is left out to serve as test set, while the remaining N-1 cases are used to obtain the model. The process is repeated N times by leaving aside each of the N given observations. The LOOCV estimates are obtained by averaging the N scores obtained on the different repetitions.

It is the user responsibility to decide which statistics are to be evaluated on each iteration and how they are calculated. This is done by creating a function that the user knows it will be called by this LOOCV routine at each iteration of the process. This user-defined function must assume that it will receive in the first 3 arguments a formula, a training set and a testing set, respectively. It should also assume that it may receive any other set of parameters that should be passed towards the learning algorithm. The result of this user-defined function should be a named vector with the values of the statistics to be estimated obtained by the learner when trained with the given training set, and tested on the given test set. See the Examples section below for an example of these functions.

If the itsInfo parameter is set to the value TRUE then the hldRun object that is the result of the function will have an attribute named itsInfo that will contain extra information from the individual repetitions of the hold out process. This information can be accessed by the user by using the function attr(), e.g. attr(returnedObject,'itsInfo'). For this information to be collected on this attribute the user needs to code its user-defined functions in a way that it returns the vector of the evaluation statistics with an associated attribute named itInfo (note that it is "itInfo" and not "itsInfo" as above), which should be a list containing whatever information the user wants to collect on each repetition. This apparently complex infra-structure allows you to pass whatever information you which from each iteration of the experimental process. A typical example is the case where you want to check the individual predictions of the model on each test case of each repetition. You could pass this vector of predictions as a component of the list forming the attribute itInfo of the statistics returned by your user-defined function. In the end of the experimental process you will be able to

inspect/use these predictions by inspecting the attribute itsInfo of the loocvRun object returned by the loocv() function. See the Examples section on the help page of the function holdout() for an illustration of this potentiality.

Value

The result of the function is an object of class loocvRun.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

experimentalComparison, loocvRun,loocvSettings, monteCarlo, holdOut, crossValidation, bootstrap

Examples

```
## Estimating the mean absolute error and the normalized mean squared
## error of rpart on the swiss data, using one repetition of 10-fold CV
data(swiss)

## First the user defined function (note: can have any name)
user.rpart <- function(form, train, test, ...) {
    require(rpart)
    model <- rpart(form, train, ...)
    preds <- predict(model, test)
    regr.eval(resp(form, test), preds,
              stats=c('mae','nmse'), train.y=resp(form, train))
}

## Now the evaluation
eval.res <- loocv(learner('user.rpart',pars=list()),
                  dataset(Infant.Mortality ~ ., swiss),
                  loocvSettings(1234))

## Check a summary of the results
summary(eval.res)

## Plot them
## Not run:
plot(eval.res)

## End(Not run)
```

loocvRun-class *Class "loocvRun"*

Description

This is the class of the objects holding the results of a leave one out cross validation experiment.

Objects from the Class

Objects can be created by calls of the form `loocvRun(...)`. These objects contain information on the learner evaluated in the LOOCV experiment, the predictive task that was used, the leave one out cross validation settings, and the results of the experiment.

Slots

`learner`: Object of class `"learner"`

`dataset`: Object of class `"task"`

`settings`: Object of class `"loocvSettings"`

`foldResults`: Object of class `"matrix"` with the results of the experiment. The rows represent the different iterations of the experiment while the columns the different statistics evaluated on each iteration.

Methods

summary `signature(object = "loocvRun")`: method used to obtain a summary of the results of the leave one out cross validation experiment.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

loocvSettings, hldRun,cvRun,mcRun,bootRun,compExp

Examples

```
showClass("loocvRun")
```

loocvSettings-class *Class "loocvSettings"*

Description

This class of objects contains the information describing a leave one out cross validation experiment, i.e. its settings.

Objects from the Class

Objects can be created by calls of the form `loocvSettings(...)`. These objects contain infoprmation on the random number generator seed and also whether the execution of the experiments should be verbose.

Slots

`loocvSeed`**:** Object of class `"numeric"` with the random number generator seed (defaulting to 1234).

`verbose`**:** Object of class `"logical"` indicating whether the execution of the experiments should be verbose (defaulting to F).

Extends

Class `"expSettings"`, directly.

Methods

show `signature(object = "loocvSettings")`: method used to show the contents of a loocvSettings object.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`loocvRun`, `mcSettings`, `cvSettings`, `hldSettings`,`bootSettings`, `expSettings`

Examples

```
showClass("loocvSettings")
```

manyNAs *Find rows with too many NA values*

Description

Small utility function to obtain the number of the rows in a data frame that have a "large" number of unknown values. "Large" can be defined either as a proportion of the number of columns or as the number in itself.

Usage

```
manyNAs(data, nORp = 0.2)
```

Arguments

data	A data frame with the data set.
nORp	A number controlling when a row is considered to have too many NA values (defaults to 0.2, i.e. 20% of the columns). If no rows satisfy the constraint indicated by the user, a warning is generated.

Value

A vector with the IDs of the rows with too many NA values. If there are no rows with many NA values and error is generated.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

complete.cases, na.omit

Examples

```
data(algae)
manyNAs(algae)
```

mcRun-class *Class "mcRun"*

Description

This is the class of the objects holding the results of a monte carlo experiment.

Objects from the Class

Objects can be created by calls of the form `mcRun(...)`. The objects contain information on the learner evaluated in the monte carlo experiment, the predictive task that was used, the monte carlo settings, and the results of the experiment.

Slots

`learner`: Object of class `"learner"`

`dataset`: Object of class `"task"`

`settings`: Object of class `"mcSettings"`

`foldResults`: Object of class `"matrix"` with the results of the experiment. The rows represent the different iterations of the experiment while the columns the different statistics evaluated on each iteration.

Methods

plot `signature(x = "mcRun", y = "missing")`: method used to visualize the results of the monte carlo experiment.

summary `signature(object = "mcRun")`: method used to obtain a summary of the results of the monte carlo experiment.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

mcSettings, hldRun,loocvRun,cvRun,bootRun,compExp

Examples

```
showClass("mcRun")
```

mcSettings-class *Class "mcSettings"*

Description

This class of objects contains the information describing a monte carlo experiment, i.e. its settings.

Objects from the Class

Objects can be created by calls of the form `mcSettings(...)`. These objects contain information on the number of repetitions of the experiments, the data used for training the models on each repetition, the data used for testing these models, and the random number generator seed.

Slots

`mcReps`**:** Object of class `"numeric"` indicating the number of repetitions of the monte carlo experiment (defaulting to 10).

`mcTrain`**:** Object of class `"numeric"`. If it is a value between 0 and 1 it is interpreted as a percentage of the available data set, otherwise it is interpreted as the number of cases to use. It defaults to 0.25.

`mcTest`**:** Object of class `"numeric"` If it is a value between 0 and 1 it is interpreted as a percentage of the available data set, otherwise it is interpreted as the number of cases to use. It defaults to 0.25.

`mcSeed`**:** Object of class `"numeric"` with the random number generator seed (defaulting to 1234).

Extends

Class `"expSettings"`, directly.

Methods

show `signature(object = "mcSettings")`: method used to show the contents of a mcSettings object.

Author(s)

Luis Torgo `<ltorgo@dcc.fc.up.pt>`

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`mcRun`, `cvSettings`, `loocvSettings`, `hldSettings`,`bootSettings`, `expSettings`

Examples

```
showClass("mcSettings")
```

`monteCarlo` *Run a Monte Carlo experiment*

Description

This function performs a Monte Carlo experiment with the goal of estimating the performance of a learner on a data set. This is a generic function in the sense that it can be used with any learner, data set and performance metrics. This is achieved by requiring the user to supply a function that takes care of the learning, testing and evaluation of the learner. This function is called for each iteration of the Monte Carlo experiment.

Usage

```
monteCarlo(learner, data.set, mcSet, itsInfo = F, verbose = T)
```

Arguments

`learner`	This is an object of the class `learner` (type "class?learner" for details) representing the system to evaluate.
`data.set`	This is an object of the class `dataset` (type "class?dataset" for details) representing the data set to be used in the evaluation.
`mcSet`	This is an object of the class `mcSettings` (type "class?mcSettings" for details) with the experimental settings of the Monte Carlo experiment.
`itsInfo`	Boolean value determining whether the object returned by the function should include as an attribute a list with as many components as there are iterations in the experimental process, with each component containing information that the user-defined function decides to return on top of the standard error statistics. See the Details section for more information.
`verbose`	A boolean value controlling the level of output of the function execution, defaulting to `TRUE`

Details

This function estimates a set of evaluation statistics through a Monte Carlo experiment. The user supplies a learning system and a data set, together with the experiment settings. These settings should specify, among others, the size of the training (TR) and testing sets (TS) and the number of repetitions (R) of the train+test cycle. The function randomly selects a set of R numbers in the interval [TR+1,NDS-TS+1], where NDS is the size of the data set. For each of these R numbers the previous TR observations of the data set are used to learn a model and the subsequent TS observations for testing it and obtaining the wanted evaluation statistics. The resulting R estimates of the evaluation statistics are averaged at the end of this process resulting in the Monte Carlo estimates of these metrics.

This function is particularly adequate for obtaining estimates of performance for time series prediction problems. The reason is that the experimental repetitions ensure that the order of the rows in the original data set are never swaped. If this order is related to time stamps, as is the case in time series, this is an important issue to ensure that a prediction model is never tested on past observations of the time series.

If the `itsInfo` parameter is set to the value TRUE then the `hldRun` object that is the result of the function will have an attribute named itsInfo that will contain extra information from the individual repetitions of the hold out process. This information can be accessed by the user by using the function `attr()`, e.g. attr(returnedObject,'itsInfo'). For this information to be collected on this attribute the user needs to code its user-defined functions in a way that it returns the vector of the evaluation statistics with an associated attribute named itInfo (note that it is "itInfo" and not "itsInfo" as above), which should be a list containing whatever information the user wants to collect on each repetition. This apparently complex infra-structure allows you to pass whatever information you which from each iteration of the experimental process. A typical example is the case where you want to check the individual predictions of the model on each test case of each repetition. You could pass this vector of predictions as a component of the list forming the attribute itInfo of the statistics returned by your user-defined function. In the end of the experimental process you will be able to inspect/use these predictions by inspecting the attribute itsInfo of the `mcRun` object returned by the `monteCarlo()` function. See the Examples section on the help page of the function `holdout()` for an illustration of this potentiality.

Value

The result of the function is an object of class `mcRun`.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`experimentalComparison`, `mcRun`, `mcSettings`, `slidingWindowTest`, `growingWindowTest`, `crossValidation`, `holdOut`, `loocv`, `bootstrap`

Examples

```
## The following is an example of a possible approach to a time series
## problem, although in this case the used data is clearly not a time
## series being selected only for illustration purposes

data(swiss)


## The base learner used in the experiment
mc.rpartXse <- function(form, train, test, ...) {
```

```
    model <- rpartXse(form, train, ...)
    preds <- predict(model, test)
    regr.eval(resp(form, test), preds,
              stats=c('mae','nmse'), train.y=resp(form, train))
}

## Estimate the MAE and NMSE of the learner rpartXse when asked to
## obtain predictions for a test set with 10 observations given a
## training set with 20 observations. The predictions for the 10
## observations are obtained using a sliding window learn+test approach
## (see the help of function slidingWindowTest() ) with a
## model re-learning step of 5 observations.
## Estimates are obtained by repeating 10 times the train+test process

x <- monteCarlo(learner("slidingWindowTest",
                        pars=list(learner=learner("mc.rpartXse",pars=list(se=1)),
                                  relearn.step=5
                                 )
                          ),
                        dataset(Infant.Mortality ~ ., swiss),
                        mcSettings(10,20,10,1234)
                    )

summary(x)
```

`outliers.ranking` *Obtain outlier rankings*

Description

This function uses hierarchical clustering to obtain a ranking of outlierness for a set of cases. The ranking is obtained on the basis of the path each case follows within the merging steps of a agglomerative hierarchical clustering method. See the references for further technical details on how these rankings are obtained.

Usage

```
outliers.ranking(data, test.data = NULL, method = "sizeDiff",
                 method.pars = NULL,
                 clus = list(dist = "euclidean",alg = "hclust",
                             meth = "ward"),
                 power = 1, verb = F)
```

Arguments

`data`	The data set to be ranked according to outlyingness. This parameter can also be the distance matrix of your additional data set, in case you wish to calculate these distances "outside" of this function.

test.data	If a data set is provided in this argument, then the rankings are obtained for these cases and not for the cases provided in the argument data. The clustering process driving the obtention of the rankings is carried out on the union of the two sets of data (data and test.data), but the resulting outlier ranking factors are only for the observations belonging to this set. This parameter defaults to NULL.
method	The method used to obtain the outlier ranking factors (see the Details section). Defaults to "sizeDiff".
method.pars	A list with the parameter values specific to the method selected for obtaining the outlier ranks (see the Details section).
clus	This is a list that provides several parameters of the clustering process that drives the calculation of the outlier raking factors. If the parameter data is not a distance function, then this list should contain a component named dist with a value that should be one of the possible values of the parameter method the the function dist() (see the help of this function for further details). The list should also contain a component named alg with the name of the clustering algorithm that should be used. Currently, valid names are either "hclust" (the default) or "diana". Finally, in case the clustering algorithm is "hclust" then the list should also contain a component named meth with the name of the agglomerative method to use in the hierarchical clustering algorithm. This should be a valid value of the parameter method of the function hclust() (check its help page for further details).
power	Integer value. It allows to raise the distance matrix to some power with the goal of "amplifying" the distance values (defaults to 1).
verb	Boolean value that determines the level of verbosity of the function (default to FALSE).

Details

This function produces outlier ranking factors for a set of cases. The methodology used for obtaining these factors is described in Section 4.4.1.3 of the book Data Mining with R (Torgo, 2010) and more details can be obtained in Torgo (2007). The methodology is based on the simple idea of using the information provided by an agglomerative hierarchical clustering algorithm to infer the degree of outlyingness of the observations. The basic assumption is that outliers should offer "more resistance" to being clustered, i.e. being merged on large groups of observations.

The function was written to be used with the outcome of the hclust() R function that implements several agglomerative clustering methods. Although in theory the methodology could be used with any other agglomerative hierarchical clustering algorithm, the fact is that the code of this implementation strongly depends on the data structures produced by the hclust() function. As such if you wish to change the function to be able to use other clustering algorithms you should ensure that the data structures it produces are compatible with the requirements of our function. Specifically, your clustering algorithm should produce a list with a component named merge that should be a matrix describing the merging steps of the clustering process (see the help page of the hclust() function for a full description of this data structure). This is the only data structure that is required by our function and that is used from the object returned by clustering algorithm. The diana() clustering algorithm also produces this type of information and thus can also be used with our function by providing the value "diana" on the component alg of the list forming the parameter clus.

There are essentially two ways of using this function. The first consists in giving it a data set on the parameter `data` and the function will rank these observations according to their outlyingness. The other consists in specifying two sets of data. One is the set for which you want the outlyingness factors that should be given on the parameter `test.data`. The second set is provided on the `data` parameter and it is used to increase the ammount of data used in the clustering process to improve the statistical reliability of the process.

In the first way of using this function that was described above the user can either supply the data set or the respective distance matrix. If the data set is provided then the user should specify the type of distance metric it should be used to calculate the distances between the observations. This is done by including a distance calculation method in the "dist" component of the list provided in parameter `clus`. This method should be a valid value of the parameter `method` of the R function `dist()` (see its help for details).

This function currently implements three different methods for obtaing outlier ranking factors from the clustering process. These are: "linear", "sigmoid" and "sizeDiff" (the default). Irrespectively, of this method the outlyingness factor of observation X is obtained by: OF_H(X) = max_i of_i(X), where i represents the different merging steps of the clustering process and it goes from 1 to N-1, where N is the size of the data set to be clustered. The three methods differ in the way they calculate of_i(X) for each merging step. In the "linear" method of_i(X) = i / (N-1) * p(|g|), where g is the group to which X belongs at the merging step i (each merging step involves two groups), and |g| is the size of that group. The function p() is a penalization factor depending on the size of the group. The larger this size the smaller the value of p(), p(s) = I(s < thr) * (1 - (s-1) / (N-2)), where I() is an indicator function and thr is a threshold defined as perc*N. The user should set the value of perc by including a component named "sz.perc" in the list provided in the parameter `method.pars`. In the "sigmoid" method of_i(X) = exp(-2 * (i - (N-1))^2 / (N-1)^2) * p(|g|), where the p() function has the same meaning as in the "linear" method but this time is defined as p(s) = I(s < 2*thr) * (1 - exp(-4 * (s-2*thr)^2 / (2*thr)^2)). Again thr is perc*N and the user must set the value of perc by including a component named "sz.perc" in the list provided in the parameter `method.pars`. Finally, the method "sizeDiff" defines of_i(X) = max (0, (|g_y,i| - |g_x,i|) / (|g_y,i| + |g_x,i|)), where g_y,i and g_x,i are the two groups involved in the merge at step i, and g_x,i is the group which X belongs to. Note that if X belongs to the larger of the two groups this will get X a value of of_i() equals to zero.

Value

The result of this function is a list with four components. Component `rank.outliers` contains a vector with as many positions as there are cases to rank, where position i of the vector contains the rank order of the observation i. Component `prob.outliers` is another vector with the same size this time containing the outlyingness factor (the value of OF_H(X) described in the Details section) of each observation. Component `h` contains the object returned by the clustering process. Finally, component `dist` contains the distance matrix used i nthe clustering process.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

Torgo, L. (2007) : *Resource-bounded Fraud Detection*, in Progress in Artificial Intelligence, 13th Portuguese Conference on Artificial Intelligence, EPIA 2007, Neves et. al (eds.). LNAI, Springer.

Examples

```
## Some examples with algae frequencies in water samples
data(algae)

## Trying to obtain a reanking of the 200 samples
o <- outliers.ranking(algae)

## As you may have observed the function complained about some problem
## with the dist() function. The problem is that the algae data frame
## contains columns (the first 3) that are factors and the dist() function
## assumes all numeric data.
## We can solve the problem by calculating the distance matrix "outside"
## using the daisy() function that handles mixed-mode data, as show in
## the code below that requires the R package "cluster" to be available
## dm <- daisy(algae)
## o <- outliers.ranking(dm)

## Now let us check the outlier ranking factors ordered by decreasing
## score of outlyingness
o$prob.outliers[o$rank.outliers]

## Another example with detection of fraudulent transactions
data(sales)

## trying to obtain the outlier ranks for the set of transactions of a
## salesperson regarding one particular product, taking into
## consideration the overall existing transactions of that product
s <- sales[sales$Prod == 'p1',c(1,3:4)]  # transactions of product p1
tr <- na.omit(s[s$ID != 'v431',-1])      # all except salesperson v431
ts <- na.omit(s[s$ID == 'v431',-1])

o <- outliers.ranking(data=tr,test.data=ts,
         clus=list(dist='euclidean',alg='hclust',meth='average'))
# The outlyingness factor of the transactions of this salesperson
o$prob.outliers
```

PRcurve *Plot a Precision/Recall curve*

Description

Precision/recall (PR) curves are visual representations of the performance of a classification model in terms of the precision and recall statistics. The curves are obtained by proper interpolation of the values of the statistics at different working points. These working points can be given by different cut-off limits on a ranking of the class of interest provided by the model.

Usage

```
PRcurve(preds, trues, ...)
```

Arguments

preds	A vector containing the predictions of the model.
trues	A vector containing the true values of the class label. Must have the same dimension as preds.
...	Further parameters that are passed to the plot() function.

Details

This function uses the infra-structure provided by the ROCR package (Sing et al., 2009). This package allows us to obtain several measures of the predictive performance of models. We use it to obtain the precision and recall of the predictions of a model. We then calculate the interpolated precision to avoid the saw-tooth effect that we would get with the standard plots produced by the ROCR package. The interpolated precision for a value r of recall is the the highest precision value for any recall level greather or equal to r.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Sing, T., Sander, O., Beerenwinkel, N., and Lengauer, T. (2009). *ROCR: Visualizing the performance of scoring classifiers*. R package version 1.0-4.

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

prediction, performance, CRchart

Examples

```
## A simple example with data in package ROCR
library(ROCR)
data(ROCR.simple)
pred <- prediction(ROCR.simple$predictions,ROCR.simple$labels)
perf <- performance(pred,"prec","rec")
## The plot obtained with the standard ROCR functions
## Not run:
plot(perf)

## End(Not run)

## Now our Precision/Recall curve avoiding the saw-tooth effect
## Not run:
```

```
PRcurve(ROCR.simple$predictions,ROCR.simple$labels)

## End(Not run)
```

---

prettyTree *Visual representation of a tree-based model*

Description

This function plots a tree-based model, i.e. a rpart object

Usage

```
prettyTree(t, compress = F, branch = 0.2, margin = 0.1, uniform = T,
           all = T, cex = 0.8, font = 10, use.n = T, fwidth = 0.5,
           fheight = 0.45, center = 0, ...)
```

Arguments

t	A rpart object
compress	A boolean parameter passed to plot.rpart(). See the help page of this function for further details. Defaults to F.
branch	A numeric parameter passed to plot.rpart(). See the help page of this function for further details. Defaults to 0.2.
margin	A numeric parameter passed to plot.rpart(). See the help page of this function for further details. Defaults to 0.1.
uniform	A boolean parameter passed to plot.rpart(). See the help page of this function for further details. Defaults to T.
all	A boolean parameter passed to text.rpart(). See the help page of this function for further details. Defaults to T.
cex	A number controling the character size. Defaults to 0.8.
font	A number setting the base font size in points. Defaults to 10.
use.n	A boolean parameter passed to text.rpart(). See the help page of this function for further details. Defaults to T.
fwidth	A numeric parameter passed to text.rpart(). See the help page of this function for further details. Defaults to 0.5.
fheight	A numeric parameter passed to text.rpart(). See the help page of this function for further details. Defaults to 0.45.
center	A numeric parameter controlling drawing of ellipses. Defaults to 0.
...	Further parameters passed both to plot.rpart() and text.rpart()

Details

This function achieves the same functionallity as applying the function `plot()` and then the function `text()` to a `rpart` object: it essentially obtains a graphical representation of a tree-based model. The basic differences are related to visual formatting of the trees.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Therneau, T. M. and Atkinson, B.; port by Brian Ripley. (2010). *rpart: Recursive Partitioning*. R package version 3.1-46.

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`plot.rpart`, `text.rpart`, `rpartXse`, `rpart`

Examples

```
data(iris)
tree <- rpartXse(Species ~ ., iris)
## Not run:
prettyTree(tree)
prettyTree(tree,all=F,use.n=F,branch=0)

## End(Not run)
```

---

rankSystems *Provide a ranking of learners involved in an experimental comparison.*

Description

Given a `compExp` object resulting from an experimental comparison, this function provides a ranking (by default the top 5) of the learners involved in the comparison. The rankings are provided by data set and for each evaluation metric.

Usage

```
rankSystems(compRes, top = 5, maxs = rep(F, dim(compRes@foldResults)[2]))
```

Arguments

compRes	An object of class compExp with the results of the experimental comparison.
top	The number of learners to include in the rankings (defaulting to 5)
maxs	A vector of booleans with as many elements are there are statistics measured in the experimental comparison. A True value means the respective statistic is to be maximized, while a False means minimization. Defaults to all False values.

Value

The function returns a named list with as many components as there are data sets in the comparison. For each data set you will get another named list, with as many elements as there evaluation statistics. For each of these components you have a data frame with N lines, where N is the size of the requested rank. Each line includes the name of the learner in the respective rank position and the score he got on that particular data set / evaluation metric.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

experimentalComparison, bestScores, statScores

Examples

```
## Estimating several evaluation metrics on different variants of a
## regression tree and of a SVM, on  two data sets, using one repetition of 10-fold CV
data(swiss)
data(mtcars)

## First the user defined functions
cv.rpartXse <- function(form, train, test, ...) {
    require(DMwR)
    t <- rpartXse(form, train, ...)
    p <- predict(t, test)
    mse <- mean((p - resp(form, test))^2)
    c(nmse = mse/mean((mean(resp(form, train)) - resp(form, test))^2),
        mse = mse)
}

## run the experimental comparison
results <- experimentalComparison(
               c(dataset(Infant.Mortality ~ ., swiss),
                 dataset(mpg ~ ., mtcars)),
               c(variants('cv.rpartXse',se=c(0,0.5,1))),
```

```
                cvSettings(1,10,1234)
                                    )
## get the top 3 best performing systems
rankSystems(results,top=2)
```

`reachability` *An auxiliary function of* `lofactor()`

Description

This function computes the reachability measure for each instance of a dataset. This result is used later to compute the Local Outlyingness Factor.

Usage

```
reachability(distdata, k)
```

Arguments

`distdata`	The matrix of distances.
`k`	The number of neighbors.

Details

This function is strongly based on the code provided by Acuna et. al. (2009) for the previously available `dprep` package.

Value

A vector.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Acuna, E., and Members of the CASTLE group at UPR-Mayaguez, (2009). *dprep: Data preprocessing and visualization functions for classification*. R package version 2.1.

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`lofactor`

`regr.eval` *Calculate Some Standard Regression Evaluation Statistics*

Description

This function is able to calculate a series of regression evaluation statistics given two vectors: one with the true target variable values, and the other with the predicted target variable values.

Usage

```
regr.eval(trues, preds,
          stats = if (is.null(train.y)) c("mae","mse","rmse","mape")
                  else c("mae","mse","rmse","mape","nmse","nmae"),
          train.y = NULL)
```

Arguments

`trues`	A numeric vector with the true values of the target variable.
`preds`	A numeric vector with the predicted values of the target variable.
`stats`	A vector with the names of the evaluation statistics to calculate. Possible values are "mae", "mse", "rmse", "mape", "nmse" or "nmae". The two latter require that the parameter `train.y` contains a numeric vector of target variable values (see below).
`train.y`	In case the set of statistics to calculate include either "nmse" or "nmae", this parameter should contain a numeric vector with the values of the target variable on the set of data used to obtain the model whose performance is being tested.

Details

The regression evaluation statistics calculated by this function belong to two different groups of measures: absolute and relative. The former include "mae", "mse", and "rmse" and are calculated as follows:

"mae": mean absolute error, which is calculated as sum(|t_i - p_i|)/N, where t's are the true values and p's are the predictions, while N is supposed to be the size of both vectors.

"mse": mean squared error, which is calculated as sum((t_i - p_i)^2)/N

"rmse": root mean squared error that is calculated as sqrt(mse)

The remaining measures ("mape", "nmse" and "nmae") are relative measures, the two later comparing the performance of the model with a baseline. They are unit-less measures with values always greater than 0. In the case of "nmse" and "nmae" the values are expected to be in the interval [0,1] though occasionaly scores can overcome 1, which means that your model is performing worse than the baseline model. The baseline used in our implementation is a constant model that always predicts the average target variable value, estimated using the values of this variable on the training data (data used to obtain the model that generated the predictions), which should be given in the parameter `train.y`. The relative error measure "mape" does not require a baseline. It simply calculates the average percentage difference between the true values and the predictions.

These measures are calculated as follows:

"mape": sum(|(t_i - p_i) / t_i|)/N

"nmse": sum((t_i - p_i)^2) / sum((t_i - AVG(Y))^2), where AVG(Y) is the average of the values provided in vector `train.y`

"nmae": sum(|t_i - p_i|) / sum(|t_i - AVG(Y)|)

Value

A named vector with the calculated statistics.

Note

In case you require either "nmse" or "nmae" to be calculated you must supply a vector of numeric values through the parameter `train.y`, otherwise the function will return an error message. The average of these values will be used as the baseline model against which your model predictions will be compared to.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

class.eval

Examples

```
## Calculating several statistics of a regression tree on the Swiss data
data(swiss)
idx <- sample(1:nrow(swiss),as.integer(0.7*nrow(swiss)))
train <- swiss[idx,]
test <- swiss[-idx,]
library(rpart)
model <- rpart(Infant.Mortality ~ .,train)
preds <- predict(model,test)
## calculate mae and rmse
regr.eval(test[,'Infant.Mortality'],preds,stats=c('mae','rmse'))
## calculate all statistics
regr.eval(test[,'Infant.Mortality'],preds,train.y=train[,'Infant.Mortality'])
```

`ReScaling` *Re-scales a set of continuous values into a new range using a linear scaling*

Description

Function for normalizing the range of values of a continuous variable into a new interval using a linear scaling.

Usage

```
ReScaling(x, t.mn, t.mx, d.mn = min(x,na.rm=T), d.mx = max(x,na.rm=T))
```

Arguments

`x`	A vector with numeric values
`t.mn`	The minimum value in the new scale
`t.mx`	The maximum value in the new scale
`d.mn`	The minimum value of the continuous variable being normalized (defaults to the minimum of the values in `x`).
`d.mx`	The maximum value of the continuous variable being normalized (defaults to the maximum of the values in `x`).

Details

The re-scaling consist in transforming the value x into

sc*x + t.mn - sc*d.mn

where sc = (t.mx-t.mn)/(d.mx-d.mn)

Value

An object with the same dimensions as `x` but with the values normalized

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`scale`, `SoftMax`, `LinearScaling`

Examples

```
## A simple example with the algae data set
summary(LinearScaling(algae[,'NO3']))
summary(ReScaling(LinearScaling(algae[,'NO3']),-10,10))
```

`resp` *Obtain the target variable values of a prediction problem*

Description

This function obtains the values in the column whose name is the target variable of a prediction problem described by a formula.

Usage

```
resp(formula, data)
```

Arguments

`formula`	A formula describing a prediction problem
`data`	The data frame containing the data of the prediction problem

Value

A vector of values

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`as.formula`

Examples

```
data(algae)
tgt <- resp(a1 ~ ., algae)
summary(tgt)
## Not run:
hist(tgt,main='Alga a1')

## End(Not run)
```

`rpartXse` *Obtain a tree-based model*

Description

This function is based on the tree-based framework provided by the `rpart` package (Therneau et. al. 2010). It basically, integrates the tree growth and tree post-pruning in a single function call. The post-pruning phase is essentially the 1-SE rule described in the CART book (Breiman et. al. 1984).

Usage

```
rpartXse(form, data, se = 1, cp = 0, minsplit = 6, verbose = F, ...)
```

Arguments

`form`	A formula describing the prediction problem
`data`	A data frame containg the training data to be used to obtain the tree-based model
`se`	A value with the number of standard errors to use in the post-pruning of the tree using the SE rule (defaults to 1)
`cp`	A value that controls the stopping criteria used to stop the initial tree growth (defaults to 0)
`minsplit`	A value that controls the stopping criteria used to stop the initial tree growth (defaults to 6)
`verbose`	The level of verbosity of the function (defaults to F)
`...`	Any other arguments that are passed to the `rpart()` function

Details

The x-SE rule for tree post-pruning is based on the cross-validation estimates of the error of the sub-trees of the initially grown tree, together with the standard errors of these estimates. These values are used to select the final tree model. Namely, the selected tree is the smallest tree with estimated error less than the B+x*SE, where B is the lowest estimate of error and SE is the standard error of this B estimate.

Value

A `rpart` object

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Therneau, T. M. and Atkinson, B.; port by Brian Ripley. (2010). *rpart: Recursive Partitioning*. R package version 3.1-46.

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). *Classification and regression trees*. Statistics/Probability Series. Wadsworth & Brooks/Cole Advanced Books & Software.

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

rt.prune, rpart, prune.rpart

Examples

```
data(iris)
tree <- rpartXse(Species ~ ., iris)
tree

## A visual representation of the classification tree
## Not run:
prettyTree(tree)

## End(Not run)
```

`rt.prune` *Prune a tree-based model using the SE rule*

Description

This function implements the SE post pruning rule described in the CART book (Breiman et. al., 1984)

Usage

```
rt.prune(tree, se = 1, verbose = T, ...)
```

Arguments

`tree`	An `rpart` object
`se`	The value of the SE threshold (defaulting to 1)
`verbose`	The level of verbosity (defaulting to T)
`...`	Any other arguments passed to the function `prune.rpart()`

Details

The x-SE rule for tree post-pruning is based on the cross-validation estimates of the error of the sub-trees of the initially grown tree, together with the standard errors of these estimates. These values are used to select the final tree model. Namely, the selected tree is the smallest tree with estimated error less than the B+x*SE, where B is the lowest estimate of error and SE is the standard error of this B estimate.

Value

A `rpart` object

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). *Classification and regression trees*. Statistics/Probability Series. Wadsworth & Brooks/Cole Advanced Books & Software.

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`rt.prune`, `rpart`, `prune.rpart`

Examples

```
data(iris)
tree <- rpartXse(Species ~ ., iris)
tree

## A visual representation of the classification tree
## Not run:
prettyTree(tree)

## End(Not run)
```

`runLearner` *Run a Learning Algorithm*

Description

This function can be used to run a learning algorithm whose details are stored in a `learner` class object.

Usage

```
runLearner(l, ...)
```

Arguments

`l`	`l` is an object of class learner containing the information on the learning algorithm.
`...`	`...` represent any other parameters that are passed to the execution of the learning algorithm.

Value

The value returned by the function is the object that results from the execution of the learning algorithm.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

learner

Examples

```
## Run multiple linear regression on the Swiss data set.
data(swiss)
lrn <- learner('lm',pars=list())
runLearner(lrn,Infant.Mortality ~ .,swiss)

## Not run: library(nnet)
lrn2 <- learner('nnet',pars=list(size=4,decay=0.1,linout=TRUE))
runLearner(lrn2,Infant.Mortality ~ .,swiss)

## End(Not run)
```

sales *A data set with sale transaction reports*

Description

This data frame contains 401,146 transaction reports. Each report is made by a salesperson identified by an ID and reports the quantity sold of some product. The data set caontins information on 5 variables: ID (salesperson ID), Prod (product ID), Quant (the sold quantity), Val (the reported value of the transaction) and Insp (a factor containing information on a inspection of the report with possible values 'ok',' fraud' or 'unkn').

Usage

```
sales
```

Format

A data frame with 401,146 rows and 5 columns

Source

Undisclosed

SelfTrain *Self train a model on semi-supervised data*

Description

This function can be used to learn a classification model from semi-supervised data. This type of data includes observations for which the class label is known as well as observation with unknown class. The function implements a strategy known as self-training to be able to cope with this semi-supervised learning problem. The function can be applied to any classification algorithm that is able to obtain class probabilities when asked to classify a set of test cases (see the Details section).

Usage

```
SelfTrain(form, data, learner, predFunc, thrConf = 0.9, maxIts = 10,
          percFull = 1, verbose = F)
```

Arguments

form	A formula describing the prediction problem.
data	A data frame containing the available training set that is supposed to contain some rows for which the value of the target variable is unknown (i.e. equal to NA).
learner	An object of class learner (see class?learner for details), indicating the base classification algorithm to use in the self-training process.
predFunc	A string with the name of a function that will carry out the probabilistic classification tasks that will be necessary during the self training process (see the Details section).
thrConf	A number between 0 and 1, indicating the required classification confidence for an unlabelled case to be added to the labelled data set with the label predicted predicted by the classification algorithm.
maxIts	The maximum number of iterations of the self-training process.
percFull	A number between 0 and 1. If the percentage of labelled cases reaches this value the self-training process is stoped.
verbose	A boolean indicating the verbosity level of the function.

Details

Self-training (e.g. Yarowsky, 1995) is a well-known strategy to handle classification problems where a subset of the available training data has an unknown class label. The general idea is to use an iterative process where at each step we try to augment the set of labelled cases by "asking" the current classification model to label the unlabelled cases and choosing the ones for which the model is more confident on the assigned label to be added to the labeled set. With this extended set of labelled cases a new classification model is learned and the process is repeated until certain termination criteria are met.

This implementation of the self-training algorithm is generic in the sense that it can be used with any baseline classification learner provided this model is able to produce confidence scores for its predictions. The user needs to take care of the learning of the models and of the classification of the unlabelled cases. This is done as follows. The user supplies a learner object (see class?learner for details) in parameter learner to represent the algorithm to be used to obtain the classification models on each iteration of the self-training process. Furthermore, the user should create a function, whose named should be given in the parameter predFunc, that takes care of the classification of the currently unlabelled cases, on each iteration. This function should be written so that it receives as first argument the learned classification model (with the current training set), and a data frame with test cases in the second argument. This user-defined function should return a data frame with two columns and as many rows as there are rows in the given test set. The first column of this data frame should contain the assigned class labels by the provided classification model, for the respective test case. The second column should contain the confidence (a number between 0 and 1) associated to that classification. See the Examples section for an illustration of such user-defined function.

This function implements the iterative process of self training. On each iteration the provided learner is called with the set of labelled cases within the given data set. Unlabelled cases should have the value NA on the column of the target variable. The obtained classification model is then passed to the user-defined "predFunc" function together with the subset of the data that is unlabelled. As

mentioned above this function returns a set of predicted class labels and the respective confidence. Test cases with confidence above the user-specified threshold (parameter `thrConf`) will be added to the labelled training set, with the label assigned by the current model. With this new training set a new classification model is obtained and the overall process repeated.

The self-training process stops if either there are no classifications that reach the required confidence level, if the maximum number of iterations is reached, or if the size of the current labelled training set is alread the target percentage of the given data set.

Value

This function returns a classification model. This will be an object of the same class as the object returned by the base classification learned provided by the user.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

Yarowski, D. (1995). *Unsupervised word sense disambiguation rivaling supervised methods*. In Proceedings of the 33rd Annual Meeting of the association for Computational Linguistics (ACL), pages 189-196.

Examples

```
## Small example with the Iris classification data set
data(iris)

## Dividing the data set into train and test sets
idx <- sample(150,100)
tr <- iris[idx,]
ts <- iris[-idx,]

## Learn a tree with the full train set and test it
stdTree <- rpartXse(Species~ .,tr,se=0.5)
table(predict(stdTree,ts,type='class'),ts$Species)

## Now let us create another training set with most of the target
## variable values unknown
trSelfT <- tr
nas <- sample(100,70)
trSelfT[nas,'Species'] <- NA

## Learn a tree using only the labelled cases and test it
baseTree <- rpartXse(Species~ .,trSelfT[-nas,],se=0.5)
table(predict(baseTree,ts,type='class'),ts$Species)

## The user-defined function that will be used in the self-training process
```

```
f <- function(m,d) {
      l <- predict(m,d,type='class')
      c <- apply(predict(m,d),1,max)
      data.frame(cl=l,p=c)
}

## Self train the same model using the semi-superside data and test the
## resulting model
treeSelfT <- SelfTrain(Species~ .,trSelfT,learner('rpartXse',list(se=0.5)),'f')
table(predict(treeSelfT,ts,type='class'),ts$Species)
```

`sigs.PR` *Precision and recall of a set of predicted trading signals*

Description

This function calculates the values of Precision and Recall of a set of predicted signals, given the set of true signals. The function assumes three types of signals: 'b' (Buy), 's' (Sell) and 'h' (Hold). The function returns the values of Precision and Recall for the buy, sell and sell+buy signals.

Usage

```
sigs.PR(preds, trues)
```

Arguments

`preds`	A factor with the predicted signals (values should be 'b','s', or 'h')
`trues`	A factor with the predicted signals (values should be 'b','s', or 'h')

Details

Precision and recall are two evaluation statistics often used to evaluate predictions for rare events. In this case we are talking about buy and sell opportunities.

Precision is the proportion of the events signaled by a model that actually occurred. Recall is a proportion of events that occurred that the model was able to capture. Ideally, the models should aim to obtain 100% precision and recall. However, it is often the case that there is a trade-off between the two statistics.

Value

A matrix with three rows and two columns. The columns are the values of Precision and Recall, respectively. The rows are the values for the three different events (sell, buy and sell+buy).

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

trading.signals, tradingEvaluation, trading.simulator

Examples

```
## A simple illustrative example use with random signals
ind <- rnorm(sd=0.3,100)
sigs <- trading.signals(ind,b.t=0.1,s.t=-0.1)
indT <- rnorm(sd=0.3,100)
sigsT <- trading.signals(indT,b.t=0.1,s.t=-0.1)
sigs.PR(sigs,sigsT)
```

`slidingWindowTest` *Obtain the predictions of a model using a sliding window learning approach.*

Description

This function implements the sliding window learning method that is frequently used in time series forecasting. The function allows applying this methodology to any modelling technique. The function returns the predictions of this technique, when applied using a sliding window approach, for the given test set.

Usage

```
slidingWindowTest(learner, form, train, test, relearn.step = 1, verbose = T)
```

Arguments

`learner`	This is an object of the class `learner` (type "class?learner" for details) representing the system to evaluate.
`form`	A formula describing the prediction problem.
`train`	A data frame with the initial training data. The size of this training set will also determine the size of the sliding window.
`test`	A data frame with the test set for which we want predictions.
`relearn.step`	A number indicating the number of test cases until a new model is re-learned by sliding the training window to cases that are nearest to the current test case.
`verbose`	A boolean determining the level of verbosity of the function.

Details

The sliding window is a method frequently used to handle time series prediction problems. The basic motivation is that as time goes by the data gets "old" and thus the models should be re-learned to re-adjust for "fresher" data. This function implements this general idea for any modelling technique.

The function receives an initial training set whose size will determine the size of the sliding window. Using this initial set a first model is obtained with the supplied modelling technique. This model is applied to obtain predictions for the first `relearn.step` test cases. Afterwards a new model is obtained with the more recent training cases. This new set will have the same size as the initially provided training set. It is thus as if the training set slided forward `relearn.step` observations. This second model is again used to obtain predictions for another set of `relearn.step` test cases. The sliding process keeps going until we obtain predictions for all provided test cases.

Value

A vector with the predictions for the test set. Note that if the target variable is a factor this vector will also be a factor.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.liaad.up.pt/~ltorgo/DataMiningWithR

See Also

growingWindowTest,monteCarlo

Examples

```
data(swiss)

## Obtain the predictions of model rpartXse() for the last 22 rows of
## the swiss data set, when used with a sliding window of 25 cases with
## a relearning step of 3

## The base learner used in the experiment
learnAndTest.rpartXse <- function(form, train, test, ...) {
    model <- rpartXse(form, train, ...)
    predict(model, test)
}

preds <- slidingWindowTest(learner('learnAndTest.rpartXse',pars=list(se=0.5)),
                           Infant.Mortality ~ .,
                           swiss[1:25,],
                           swiss[26:nrow(swiss),],
                           3)
```

```
## Some statistics of these predictions
regr.eval(swiss[26:nrow(swiss),'Infant.Mortality'],preds,stats = c("mae", "mse", "rmse"))
```

SMOTE *SMOTE algorithm for unbalanced classification problems*

Description

This function handles unbalanced classification problems using the SMOTE method. Namely, it can generate a new "SMOTEd" data set that addresses the class unbalance problem. Alternatively, it can also run a classification algorithm on this new data set and return the resulting model.

Usage

```
SMOTE(form, data, perc.over = 200, k = 5, perc.under = 200,
      learner = NULL, ...)
```

Arguments

`form`	A formula describing the prediction problem
`data`	A data frame containing the original (unbalanced) data set
`perc.over`	A number that drives the decision of how many extra cases from the minority class are generated (known as over-sampling).
`k`	A number indicating the number of nearest neighbours that are used to generate the new examples of the minority class.
`perc.under`	A number that drives the decision of how many extra cases from the majority classes are selected for each case generated from the minority class (known as under-sampling)
`learner`	Optionally you may specify a string with the name of a function that implements a classification algorithm that will be applied to the resulting SMOTEd data set (defaults to NULL).
`...`	In case you specify a learner (parameter `learner`) you can indicate further arguments that will be used when calling this learner.

Details

Unbalanced classification problems cause problems to many learning algorithms. These problems are characterized by the uneven proportion of cases that are available for each class of the problem.

SMOTE (Chawla et. al. 2002) is a well-known algorithm to fight this problem. The general idea of this method is to artificially generate new examples of the minority class using the nearest neighbors of these cases. Furthermore, the majority class examples are also under-sampled, leading to a more balanced dataset.

The parameters `perc.over` and `perc.under` control the amount of over-sampling of the minority class and under-sampling of the majority classes, respectively. `perc.over` will tipically be a number

above 100. With this type of values, for each case in the orginal data set belonging to the minority class, `perc.over/100` new examples of that class will be created. If `perc.over` is a value below 100 than a single case will be generated for a randomly selected proportion (given by `perc.over/100`) of the cases belonging to the minority class on the original data set. The parameter `perc.under` controls the proportion of cases of the majority class that will be randomly selected for the final "balanced" data set. This proportion is calculated with respect to the number of newly generated minority class cases. For instance, if 200 new examples were generated for the minority class, a value of `perc.under` of 100 will randomly select exactly 200 cases belonging to the majority classes from the original data set to belong to the final data set. Values above 100 will select more examples from the majority classes.

The parameter `k` controls the way the new examples are created. For each currently existing minority class example X new examples will be created (this is controlled by the parameter `perc.over` as mentioned above). These examples will be generated by using the information from the `k` nearest neighbours of each example of the minority class. The parameter `k` controls how many of these neighbours are used.

The function can also be used to obtain directely the classification model from the resulting balanced data set. This can be done by including the name of the R function that implements the classifier in the parameter `learner`. You may also include other parameters that will be forward to this learning function. If the `learner` parameter is not `NULL` (the default) the returning value of the function will be the learned model and not the balanced data set. The function that learns the model should have as first parameter the formula describing the classification problem and in the second argument the training set.

Value

The function can return two different types of values depending on the value of the parameter `learner`. If this parameter is `NULL` (the default), the function will return a data frame with the new data set resulting from the application of the SMOTE algorithm. Otherwise the function will return the classification model obtained by the learner specified in the parameter `learner`.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). *Smote: Synthetic minority over-sampling technique*. Journal of Artificial Intelligence Research, 16:321-357.

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

Examples

```
## A small example with a data set created artificially from the IRIS
## data
data(iris)
data <- iris[, c(1, 2, 5)]
data$Species <- factor(ifelse(data$Species == "setosa","rare","common"))
## checking the class distribution of this artificial data set
```

```
table(data$Species)

## now using SMOTE to create a more "balanced problem"
newData <- SMOTE(Species ~ ., data, perc.over = 600,perc.under=100)
table(newData$Species)

## Checking visually the created data
## Not run:
par(mfrow = c(1, 2))
plot(data[, 1], data[, 2], pch = 19 + as.integer(data[, 3]),
     main = "Original Data")
plot(newData[, 1], newData[, 2], pch = 19 + as.integer(newData[,3]),
     main = "SMOTE'd Data")

## End(Not run)

## Now an example where we obtain a model with the "balanced" data
classTree <- SMOTE(Species ~ ., data, perc.over = 600,perc.under=100,
                   learner='rpartXse',se=0.5)
## check the resulting classification tree
classTree
## The tree with the unbalanced data set would be
rpartXse(Species ~ .,data,se=0.5)
```

SoftMax *Normalize a set of continuous values using SoftMax*

Description

Function for normalizing the range of values of a continuous variable using the SoftMax function (Pyle, 199).

Usage

```
SoftMax(x, lambda = 2, avg = mean(x, na.rm = T), std = sd(x, na.rm = T))
```

Arguments

x	A vector with numeric values
lambda	A numeric value entering the formula of the soft max function (see Details). Defaults to 2.
avg	The statistic of centrality of the continuous variable being normalized (defaults to the mean of the values in x).
std	The statistic of spread of the continuous variable being normalized (defaults to the standard deviation of the values in x).

Details

The Soft Max normalization consist in transforming the value x into

1 / [1+ exp((x-AVG(x))/(LAMBDA*SD(X)/2*PI))]

Value

An object with the same dimensions as x but with the values normalized

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Pyle, D. (1999). *Data preparation for data mining*. Morgan Kaufmann.

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

scale, LinearScaling, ReScaling

Examples

```
## A simple example with the algae data set
summary(SoftMax(algae[,'NO3']))
summary(algae[,'NO3'])
```

---

statNames — *Obtain the name of the statistics involved in an experimental comparison*

Description

This function produces a vector with the names of the statistics that were estimated in an experimental comparison

Usage

```
statNames(res)
```

Arguments

res	This is a compExp object (type "class?compExp" for details) that contains the results of an experimental comparison obtained through the experimentalComparison() function.

Value

A vector of strings with the names of the statistics

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

learnerNames, dsNames,experimentalComparison

`statScores` *Obtains a summary statistic of one of the evaluation metrics used in an experimental comparison, for all learners and data sets involved in the comparison.*

Description

Given a `compExp` object this function provides a summary statistic (defaulting to the average score) of the different scores obtained on a single evaluation statistic over all repetitions carried out in the experimental process. This is done for all learners and data sets of the experimental comparison. The function can be handy to obtain things like for instance the maximum score obtained by each learner on a particular statistic over all repetitions of the experimental process.

Usage

```
statScores(compRes, stat, summary = "mean")
```

Arguments

`compRes`	An object of class `compExp` with the results of the experimental comparison.
`stat`	A string with the name of the evaluation metric for which you want to obtain the scores.
`summary`	A string with the name of the function that should be used to aggregate the different repetition results into a single score (defaults to the mean value).

Value

The result of this function is a named list with as many components as there are data sets in the evaluation comparison being used. For each data set (component), we get a named vector with as many elements as there are learners in the experiment. The value for each learner is the result of applying the aggregation function (parameter `summary`) to the different scores obtained by the learner on the evaluation metric specified by the parameter `stat`.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

experimentalComparison, bestScores, rankSystems

Examples

```
## Estimating several evaluation metrics on different variants of a
## regression tree and of a SVM, on  two data sets, using one repetition
## of  10-fold CV
data(swiss)
data(mtcars)

## First the user defined functions
cv.rpartXse <- function(form, train, test, ...) {
    require(DMwR)
    t <- rpartXse(form, train, ...)
    p <- predict(t, test)
    mse <- mean((p - resp(form, test))^2)
    c(nmse = mse/mean((mean(resp(form, train)) - resp(form, test))^2),
        mse = mse)
}

## run the experimental comparison
results <- experimentalComparison(
               c(dataset(Infant.Mortality ~ ., swiss),
                 dataset(mpg ~ ., mtcars)),
               c(variants('cv.rpartXse',se=c(0,0.5,1))),
               cvSettings(1,10,1234)
                                 )

## Get the maximum value of nmse for each learner
statScores(results,'nmse','max')
## Get the interquartile range of the mse score for each learner
statScores(results,'mse','IQR')
```

---

subset-methods *Methods for Function subset in Package 'DMwR'*

Description

The method subset when applied to objects of class compExp can be used to obtain another object of the same class with a subset of the experimental results contained in the orginal object.

Methods

`signature(x = "compExp")` The method has as first argument the object of class `compExp` that you wish to subset. This method also includes four extra arguments that you can use to supply the subsetting criteria.

All subsetting is driven by the dimensions of the array `foldResults` that is one of the slots of the `compExp` object (see "class?compExp" for further details).

Namely, the parameter `its` of this method allows you to supply a vector with the subset of the repetitions of the experimental comparison that are to be "extracted" (this option is seldom used though as it is limited applicability).

The parameter `stats` allows you to indicate a vector with the subset of evaluation statistics in the orginal object. Additionally, you can instead provide a regular expression to be matched against the name of the statistics measured in the experiment to specify the subset you want to select.

The parameter `vars` can be used to provide a vector with the subset of learners (models) that are to be used in the subsetting. Additionally, you can instead provide a regular expression to be matched against the name of the learner variants evaluated in the experiment to specify the subset you want to select.

Finally, the parameter `dss` allows you to indicate a vector with the subset of data sets to be extracted. Additionally, you can instead provide a regular expression to be matched against the name of the datasets used in the experiment to specify the subset you want to select.

`task-class` *Class "task"*

Description

This is an auxiliary class that is part of the representation of the `dataset` class. Objects of the task class do not actually store the data - they simply store information on the name of the predictive task in question as well as the respective formula.

Objects from the Class

Objects can be created by calls of the form `task(...)`. The objects include information on the name to be given to the predictive task and also the associated formula.

Slots

`name`**:** String character with the name of the predictive taks

`formula`**:** R formula describing the task

Methods

show `signature(object = "task")`: method used to show the contents of a task object.

Author(s)

Luis Torgo (ltorgo@dcc.fc.up.pt)

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187). http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

learner, dataset

Examples

```
showClass("task")
```

---

test.algae *Testing data for predicting algae blooms*

Description

This data set contains observations on 11 variables as well as the concentration levels of 7 harmful algae. Values were measured in several European rivers. The 11 predictor variables include 3 contextual variables (season, size and speed) describing the water sample, plus 8 chemical concentration measurements.

Usage

```
test.algae
```

Format

A data frame with 140 observations and 18 columns.

Source

ERUDIT http://www.erudit.de/ - European Network for Fuzzy Logic and Uncertainty Modelling in Information Technology.

`tradeRecord-class` *Class "tradeRecord"*

Description

This is a class that contains the result of a call to the function trading.simulator(). It contains information on the trading performance of a set of signals on a given set of "future" market quotes.

Objects from the Class

Objects can be created by calls of the form `tradeRecord(...)`. These objects contain information on i) the trading variables for each day in the simulation period; ii) on the positions hold during this period; iii) on the value used for transaction costs; iv) on the initial capital for the simulation; v) on the function that implements the trading policy used in the simulation; and vi) on the list of parameters of this function.

Slots

`trading`**:** Object of class `"xts"` containing the information on the trading activities through the testing period. This object has one line for each trading date. For each date it includes information on the closing price of the market ("Close"), on the order given at the end of that day ("Order"), on the money available to the trader at the end of that day ("Money"), on the number of stocks hold by the trader ("N.Stocks"), and on the equity at the end of that day ("Equity").

`positions`**:** Object of class `"matrix"` containing the positions hold by the trader during the simulation period. This is a matrix with seven columns, with as many rows as the number of positions hold by the trader. The columns of this matrix contain the type of position ("pos.type"), the number of stocks of the position ("N.stocks"), the date when the position was opened ("Odate"), the open price ("Oprice"), the closing date ("Cdate"), the closing price ("Cprice") and the percentage return of the position ("result").

`trans.cost`**:** Object of class `"numeric"` with the monetary value of each transaction (market order).

`init.cap`**:** Object of class `"numeric"` with the initial monetary value of the trader.

`policy.func`**:** Object of class `"character"` with the name of the function that should be called at the end of each day to decide what to do, i.e. the trading policy function. This function is called with the vector of signals till the current date, the market quotes till today, the current position of the trader and the currently available money.

`policy.pars`**:** Object of class `"list"` containing a list of extra parameters to be used when calling the trading policy function (these depend on the function defined by the user).

Methods

plot `signature(x = "tradeRecord", y = "ANY")`: provides a graphical representation of the trading results.

show `signature(object = "tradeRecord")`: shows an object in a proper way.

summary `signature(object = "tradeRecord")`: provides a summary of the trading results.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

trading.simulator, tradingEvaluation

Examples

```
showClass("tradeRecord")
```

---

trading.signals *Discretize a set of values into a set of trading signals*

Description

This function transforms a set of numeric values into a set of trading signals according to two thresholds: one that establishes the limit above which any value will be transformed into a buy signal ('b'), and the other that sets the value below which we have a sell signal ('s'). Between the two thresholds we will have a hold signal ('h').

Usage

```
trading.signals(vs, b.t, s.t)
```

Arguments

vs	A vector with numeric values
b.t	A number representing the buy threshold
s.t	A number representing the sell threshold

Value

A factor with three possible values 'b' (buy), 's' (sell) or 'h' (hold)

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).
http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

trading.signals, tradingEvaluation, trading.simulator

Examples

```
trading.signals(rnorm(sd=0.5,100),b.t=0.1,s.t=-0.12)
```

`trading.simulator` *Simulate daily trading using a set of trading signals*

Description

This function can be used to obtain the trading performance of a set of signals by simulating daily trading on a market with these signals according to a user-defined trading policy. The idea is that the user supplies the actual quotes for the simulation period together with the trading signals to use during this period. On top of that the user also supplies a function implementing the trading policy to use. The result is a trading record for this period. This result can then be inspected and used to obtain several trading performance metrics with other functions.

Usage

```
trading.simulator(market, signals,
                  policy.func, policy.pars = list(),
                  trans.cost = 5, init.cap = 1e+06)
```

Arguments

`market`	A `xts` object containing the market quotes for each day of the simulation period. This object should contain at least the Open, High, Low and Close quotes for each day. These quotes (with these exact names) are used within the function and thus are required.
`signals`	A factor with as many signals as there are rows in the `market` `xts` object, i.e. as many signals as there are trading days in the simulation period. The signals should be 'b' for Buy, 's' for Sell and 'h' for Hold (actually this information is solely processed within the user-defined trading policy function which means that the values may be whatever the writer of this function wants).
`policy.func`	A string with the name of the function that will be called at the end of each day of the trading period. This user-defined function implements the trading policy to be used in the simulation. See the Details section for understanding what is the task of this function.
`policy.pars`	A list with parameters that are passed to the user-defined trading policy function when it is called at the end of each day.
`trans.cost`	A number with the cost of each market transaction (defaults to 5 monetary units).
`init.cap`	A number with the initial amount of money available for trading at the start of the simulation period (defaults to 1,000,000 monetary units).

Details

This function can be used to simulate daily trading according to a set of signals. The main parameters of this function are the market quotes for the simulation period and the model signals for this period. Two other parameters are the name of the user-defined trading policy function and its list of parameters. Finally, we can also specify the cost of each transaction and the initial capital available for the trader. The simulator will call the user-provided trading policy function at the end of each daily section, and the function should return the orders that it wants the simulator to carry out. The simulator carries out these orders on the market and records all activity on several data structures. The result of the simulator is an object of class `tradeRecord` containing the information of this simulation. This object can then be used in other functions to obtain economic evaluation metrics or graphs of the trading activity.

The key issue in using this function is to create the user-defined trading policy function. These functions should be written using a certain protocol, that is, they should be aware of how the simulator will call them, and should return the information this simulator is expecting. At the end of each daily session d, the simulator calls the trading policy function with four main arguments plus any other parameters the user has provided in the call to the simulator in the parameter `policy.pars`. These four arguments are (1) a vector with the predicted signals until day d, (2) the market quotes (up to d), (3) the currently opened positions, and (4) the money currently available to the trader. The current positions are stored in a matrix with as many rows as there are open positions at the end of day d. This matrix has four columns: "pos.type" that can be 1 for a long position or -1 for a short position; "N.stocks", which is the number of stocks of the position; "Odate", which is the day on which the position was opened (a number between 1 and d); and "Oprice", which is the price at which the position was opened. The row names of this matrix contain the IDs of the positions that are relevant when we want to indicate the simulator that a certain position is to be closed. All this information is provided by the simulator to ensure the user can define a broad set of trading policy functions. The user-defined functions should return a data frame with a set of orders that the simulator should carry out. This data frame should include the following information (columns): "order", which should be 1 for buy orders and -1 for sell orders; "order.type", which should be 1 for market orders that are to be carried out immediately (actually at next day open price), 2 for limit orders or 3 for stop orders; "val", which should be the quantity of stocks to trade for opening market orders, NA for closing market orders, or a target price for limit and stop orders; "action", which should be "open" for orders that are opening a new position or "close" for orders closing an existing position; and finally, "posID", which should contain the ID of the position that is being closed, if applicable.

Value

An object of class `tradeRecord` (see 'class?tradeRecord' for details).

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

tradingEvaluation, tradeRecord, trading.signals, sigs.PR

Examples

```
## An example partially taken from chapter 3 of my book Data Mining
## with R (Torgo,2010)

## First a trading policy function
## This function implements a strategy to trade on futures with
## long and short positions. Its main ideas are the following:
## - all decisions aretaken at the end of the day, that is, after
## knowing all daily quotes of the current session.
## - if at the end of day d our models issue a sell signal and  we
## currently  do not hold any opened position, we will open a short
## position  by issuing a sell order. When this order is carried out  by
## the market at a price  pr sometime in the  future, we will
## immediately post two other orders. The first is a buy limit order
## with  a limit price of pr - p%, where p% is a target profit margin.
## We will wait 10 days for this target to be reached. If the  order  is
## not carried out by this deadline, we will buy at the closing price
## of  the 10th day. The second order is a buy stop order with a  price
## limit  pr + l%. This order is placed with the goal of limiting our
## eventual  losses with this position. The order will be executed if
## the  market reaches the price pr + l%, thus limiting our possible
## losses  to l%.
## - if the end of the day signal is buy the strategy is more or less
## the inverse
## Not run:
library(xts)
policy.1 <- function(signals,market,opened.pos,money,
                     bet=0.2,hold.time=10,
                     exp.prof=0.025, max.loss= 0.05
                     )
  {
    d <- NROW(market) # this is the ID of today
    orders <- NULL
    nOs <- NROW(opened.pos)
    # nothing to do!
    if (!nOs && signals[d] == 'h') return(orders)

    # First lets check if we can open new positions
    # i) long positions
    if (signals[d] == 'b' && !nOs) {
      quant <- round(bet*money/market[d,'Close'],0)
      if (quant > 0)
        orders <- rbind(orders,
              data.frame(order=c(1,-1,-1),order.type=c(1,2,3),
                         val = c(quant,
                                 market[d,'Close']*(1+exp.prof),
                                 market[d,'Close']*(1-max.loss)
                                ),
```

```
                        action = c('open','close','close'),
                        posID = c(NA,NA,NA)
                       )
                      )

  # ii) short positions
  } else if (signals[d] == 's' && !nOs) {
    # this is the nr of stocks we already need to buy
    # because of currently opened short positions
    need2buy <- sum(opened.pos[opened.pos[,'pos.type']==-1,
                               "N.stocks"])*market[d,'Close']
    quant <- round(bet*(money-need2buy)/market[d,'Close'],0)
    if (quant > 0)
      orders <- rbind(orders,
            data.frame(order=c(-1,1,1),order.type=c(1,2,3),
                       val = c(quant,
                               market[d,'Close']*(1-exp.prof),
                               market[d,'Close']*(1+max.loss)
                              ),
                       action = c('open','close','close'),
                       posID = c(NA,NA,NA)
                      )
                     )
  }

  # Now lets check if we need to close positions
  # because their holding time is over
  if (nOs)
    for(i in 1:nOs) {
      if (d - opened.pos[i,'Odate'] >= hold.time)
        orders <- rbind(orders,
              data.frame(order=-opened.pos[i,'pos.type'],
                         order.type=1,
                         val = NA,
                         action = 'close',
                         posID = rownames(opened.pos)[i]
                        )
                        )
    }

  orders
}

## Now let us play a bit with the SP500 quotes availabe in our package
data(GSPC)

## Let us select the last 3 months as the simulation period
market <- last(GSPC,'3 months')

## now let us generate a set of random trading signals for
## illustration purpose only
ndays <- nrow(market)
aRandomIndicator <- rnorm(sd=0.3,ndays)
```

```
  theRandomSignals <- trading.signals(aRandomIndicator,b.t=0.1,s.t=-0.1)

  ## now lets trade!
  tradeR <- trading.simulator(market,theRandomSignals,
              'policy.1',list(exp.prof=0.05,bet=0.2,hold.time=10))

  ## a few stats on the trading performance
  summary(tradeR)
  tradingEvaluation(tradeR)

## End(Not run)
  ## See the performance graphically
  ## Not run:
    plot(tradeR,market)

## End(Not run)
```

`tradingEvaluation` *Obtain a set of evaluation metrics for a set of trading actions*

Description

This function receives as argument an object of class `tradeRecord` that is the result of a call to the `trading.simulator()` function and produces a set of evaluation metrics of this simulation

Usage

```
tradingEvaluation(t)
```

Arguments

`t`	An object of call `tradeRecord` (see 'class?tradeRecord' for details)

Details

Given the result of a trading simulation this function calculates:

- The number of trades.
- The number of profitable trades.
- The percentage of profitable trades.
- The profit/loss of the simulation (i.e. the final result).
- The return of the simulation.
- The return over the buy and hold strategy.
- The maximum draw down of the simulation.
- The Sharpe Ration score.
- The average percentage return of the profitable trades.

- The average percentage return of the non-profitable trades.
- The average percentage return of all trades.
- The maximum return of all trades.
- The maximum percentage loss of all trades.

Value

A vector of evaluation metric values

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

tradeRecord, trading.simulator, trading.signals

Examples

```
## An example partially taken from chapter 3 of my book Data Mining
## with R (Torgo,2010)

## First a trading policy function
## This function implements a strategy to trade on futures with
## long and short positions. Its main ideas are the following:
## - all decisions aretaken at the end of the day, that is, after
## knowing all daily quotes of the current session.
## - if at the end of day d our models issue a sell signal and  we
## currently  do not hold any opened position, we will open a short
## position  by issuing a sell order. When this order is carried out  by
## the market at a price  pr sometime in the  future, we will
## immediately post two other orders. The first is a buy limit order
## with  a limit price of pr - p%, where p% is a target profit margin.
## We will wait 10 days for this target to be reached. If the  order  is
## not carried out by this deadline, we will buy at the closing price
## of  the 10th day. The second order is a buy stop order with a  price
## limit  pr + l%. This order is placed with the goal of limiting our
## eventual  losses with this position. The order will be executed if
## the  market reaches the price pr + l%, thus limiting our possible
## losses  to l%.
## - if the end of the day signal is buy the strategy is more or less
## the inverse
## Not run:
library(xts)
policy.1 <- function(signals,market,opened.pos,money,
                     bet=0.2,hold.time=10,
```

```
                  exp.prof=0.025, max.loss= 0.05
                  )
{
  d <- NROW(market) # this is the ID of today
  orders <- NULL
  nOs <- NROW(opened.pos)
  # nothing to do!
  if (!nOs && signals[d] == 'h') return(orders)

  # First lets check if we can open new positions
  # i) long positions
  if (signals[d] == 'b' && !nOs) {
    quant <- round(bet*money/market[d,'Close'],0)
    if (quant > 0)
      orders <- rbind(orders,
            data.frame(order=c(1,-1,-1),order.type=c(1,2,3),
                       val = c(quant,
                               market[d,'Close']*(1+exp.prof),
                               market[d,'Close']*(1-max.loss)
                              ),
                       action = c('open','close','close'),
                       posID = c(NA,NA,NA)
                      )
                     )

  # ii) short positions
  } else if (signals[d] == 's' && !nOs) {
    # this is the nr of stocks we already need to buy
    # because of currently opened short positions
    need2buy <- sum(opened.pos[opened.pos[,'pos.type']==-1,
                               "N.stocks"])*market[d,'Close']
    quant <- round(bet*(money-need2buy)/market[d,'Close'],0)
    if (quant > 0)
      orders <- rbind(orders,
            data.frame(order=c(-1,1,1),order.type=c(1,2,3),
                       val = c(quant,
                               market[d,'Close']*(1-exp.prof),
                               market[d,'Close']*(1+max.loss)
                              ),
                       action = c('open','close','close'),
                       posID = c(NA,NA,NA)
                      )
                     )
  }

  # Now lets check if we need to close positions
  # because their holding time is over
  if (nOs)
    for(i in 1:nOs) {
      if (d - opened.pos[i,'Odate'] >= hold.time)
        orders <- rbind(orders,
              data.frame(order=-opened.pos[i,'pos.type'],
                         order.type=1,
```

```
                            val = NA,
                            action = 'close',
                            posID = rownames(opened.pos)[i]
                           )
                          )
      }

    orders
  }

  ## Now let us play a bit with the SP500 quotes availabe in our package
  data(GSPC)

  ## Let us select the last 3 months as the simulation period
  market <- last(GSPC,'3 months')

  ## now let us generate a set of random trading signals for
  ## illustration purpose only
  ndays <- nrow(market)
  aRandomIndicator <- rnorm(sd=0.3,ndays)
  theRandomSignals <- trading.signals(aRandomIndicator,b.t=0.1,s.t=-0.1)

  ## now lets trade!
  tradeR <- trading.simulator(market,theRandomSignals,
              'policy.1',list(exp.prof=0.05,bet=0.2,hold.time=10))

  ## a few stats on the trading performance
  tradingEvaluation(tradeR)

## End(Not run)
```

`ts.eval` *Calculate Some Standard Evaluation Statistics for Time Series Forecasting Tasks*

Description

This function is able to calculate a series of numeric time series evaluation statistics given two vectors: one with the true target variable values, and the other with the predicted target variable values.

Usage

```
ts.eval(trues, preds,
          stats = if (is.null(train.y)) c("mae","mse","rmse","mape")
                  else c("mae","mse","rmse","mape","nmse","nmae","theil"),
          train.y = NULL)
```

Arguments

`trues`	A numeric vector with the true values of the target variable.
`preds`	A numeric vector with the predicted values of the target variable.
`stats`	A vector with the names of the evaluation statistics to calculate. Possible values are "mae", "mse", "rmse", "mape", "nmse", "nmae" or "theil". The three latter require that the parameter `train.y` contains a numeric vector of target variable values (see below).
`train.y`	In case the set of statistics to calculate include either "nmse", "nmae" or "theil", this parameter should contain a numeric vector with the values of the target variable on the set of data used to obtain the model whose performance is being tested.

Details

The evaluation statistics calculated by this function belong to two different groups of measures: absolute and relative. The former include "mae", "mse", and "rmse" and are calculated as follows:

"mae": mean absolute error, which is calculated as sum(|t_i - p_i|)/N, where t's are the true values and p's are the predictions, while N is supposed to be the size of both vectors.

"mse": mean squared error, which is calculated as sum((t_i - p_i)^2)/N

"rmse": root mean squared error that is calculated as sqrt(mse)

The remaining measures ("mape", "nmse", "nmae" and "theil") are relative measures, the three later comparing the performance of the model with a baseline. They are unit-less measures with values always greater than 0. In the case of "nmse", "nmae" and "theil" the values are expected to be in the interval [0,1] though occasionaly scores can overcome 1, which means that your model is performing worse than the baseline model. The baseline used in our implementation for metrics "nmse" and "nmae" is a constant model that always predicts the average target variable value, estimated using the values of this variable on the training data (data used to obtain the model that generated the predictions), which should be given in the parameter `train.y`. The baseline used for calculating the Theil coefficient ("theil") is the model that predicts for time t+1 the value of the time series on time t, i.e. the last known value. The relative error measure "mape" does not require a baseline. It simply calculates the average percentage difference between the true values and the predictions.

These measures are calculated as follows:

"mape": sum(|(t_i - p_i) / t_i|)/N

"nmse": sum((t_i - p_i)^2) / sum((t_i - AVG(Y))^2), where AVG(Y) is the average of the values provided in vector `train.y`

"nmae": sum(|t_i - p_i|) / sum(|t_i - AVG(Y)|)

"theil": sum((t_i - p_i)^2) / sum((t_i - t_[i-1])^2), where t_[i-1] is the last known value of the series when we are trying to forecast the value t_i

Value

A named vector with the calculated statistics.

Note

In case you require either "nmse", "nmae" or "theil" to be calculated you must supply a vector of numeric values through the parameter `train.y`, otherwise the function will return an error message.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`class.eval`

Examples

```
## A few example uses of the function
tr <- rnorm(1000)
true <- rnorm(50)
preds <- rnorm(50)
ts.eval(true,preds)
ts.eval(true,preds,train.y=tr)
ts.eval(true,preds,stats='theil',train.y=tr)
```

---

unscale *Invert the effect of the scale function*

Description

This function can be used to un-scale a set of values. This unscaling is done with the scaling information "hidden" on a scaled data set that should also be provided. This information is stored as an attribute by the function `scale()` when applied to a data frame.

Usage

```
unscale(vals, norm.data, col.ids)
```

Arguments

`vals`	A numeric matrix with the values to un-scale
`norm.data`	A numeric and scaled matrix. This should be an object to which the function `scale()` was applied.
`col.ids`	The columns of the `vals` matrix that are to be un-scaled (defaults to all of them).

Value

An object with the same dimension as the parameter `vals`

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

`scale`

Examples

```
data(algae)
normData <- scale(algae[,4:12])
t <- rpartXse(a1 ~ .,as.data.frame(normData[1:100,]))
normPs <- predict(t,as.data.frame(normData[101:nrow(normData),]))
ps <- unscale(normPs,normData)
## Not run:
 plot(algae[101:nrow(algae),'a1'],ps)

## End(Not run)
```

---

variants *Generate variants of a learning system*

Description

The main goal of this function is to facilitate the generation of different variants of a learning system. The idea is to be able to supply several possible values for a set of parameters of the learner, and then have the function to return a set of `learner` objects, each consisting of one of the different possible combinations of the variants. This function finds its use in the context of experimental comparisons among learning systems, where we may actually be interested in comparing different parameter settings for each of them.

Usage

```
variants(sys, varsRootName = sys, as.is=NULL, ...)
```

Arguments

sys	This is the string representing the name of the function of the base learner from which variants should be generated.
varsRootName	By default the names given to each variant will be formed by concatenating the base name of the learner with the terminations: ".v1", ".v2", and so on. This parameter allows you to supply a different base name.
as.is	This is a vector of parameter names (defaults to NULL that are not to be used as source for system variants. This is useful for systems that have parameters that accept as "legal" values sets (e.g. a vector) and that we do not want the function variants to interprete as source values for generating different system variants.
...	The function then accepts any number of named arguments, each with a set of values. These named arguments are supposed to be the names of arguments of the base learner, while the sets of values are the alternatives that you want to consider in the variants generation (see examples below).

Value

The result of this function is a list of learner objects. Each of these objects represents one of the parameter variants of the learner you have supplied.

Author(s)

Luis Torgo <ltorgo@dcc.fc.up.pt>

References

Torgo, L. (2010) *Data Mining using R: learning with case studies*, CRC Press (ISBN: 9781439810187).

http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR

See Also

learner,experimentalComparison

Examples

```
## Generating several variants of the "rpartXse" learner using different
## values of the paramter "se"

variants('rpartXse',se=c(0,0.5,1))
```

Index

∗Topic **classes**
bootRun-class, 6
bootSettings-class, 7
compExp-class, 16
cvRun-class, 21
cvSettings-class, 22
dataset-class, 23
expSettings-class, 28
hldRun-class, 35
hldSettings-class, 36
learner-class, 46
loocvRun-class, 52
loocvSettings-class, 53
mcRun-class, 55
mcSettings-class, 56
task-class, 88
tradeRecord-class, 90
∗Topic **datasets**
algae, 4
algae.sols, 4
GSPC, 35
sales, 76
test.algae, 89
∗Topic **methods**
subset-methods, 87
∗Topic **models**
bestScores, 5
bootstrap, 8
centralImputation, 10
class.eval, 12
compAnalysis, 14
CRchart, 18
crossValidation, 19
dist.to.knn, 24
dsNames, 25
experimentalComparison, 26
getFoldsResults, 29
getSummaryResults, 30
getVariant, 31
growingWindowTest, 33
holdOut, 37
join, 40
kNN, 42
knneigh.vect, 43
knnImputation, 44
learnerNames, 47
LinearScaling, 47
lofactor, 48
loocv, 49
manyNAs, 54
monteCarlo, 57
outliers.ranking, 59
PRcurve, 62
prettyTree, 64
rankSystems, 65
reachability, 67
regr.eval, 68
ReScaling, 70
resp, 71
rpartXse, 72
rt.prune, 73
runLearner, 74
SelfTrain, 76
sigs.PR, 79
slidingWindowTest, 80
SMOTE, 82
SoftMax, 84
statNames, 85
statScores, 86
trading.signals, 91
trading.simulator, 92
tradingEvaluation, 96
ts.eval, 99
unscale, 101
variants, 102
∗Topic **package**
DMwR-package, 3
∗Topic **univar**

centralValue, 11

algae, 4
algae.sols, 4
as.formula, *71*

bestScores, 5, *17*, *27*, *66*, *87*
bootRun, *8, 9*, *22*, *36*, *52*, *55*
bootRun *(*bootRun-class*)*, 6
bootRun-class, 6
bootSettings, *7*, *9*, *23*, *27, 28*, *37*, *53*, *56*
bootSettings *(*bootSettings-class*)*, 7
bootSettings-class, 7
bootstrap, 8, *20*, *27*, *39*, *51*, *58*

centralImputation, 10, *45*
centralValue, *11*, 11, *45*
class.eval, 12, *69*, *101*
compAnalysis, 14, *17*, *27*
compExp, *7*, *16*, *22*, *27*, *36*, *41*, *52*, *55*
compExp *(*compExp-class*)*, 16
compExp-class, 16
complete.cases, *11*, *45*, *54*
CRchart, *18*, 18, *63*
crossValidation, *9*, 19, *22*, *26, 27*, *39*, *51*, *58*
cvRun, *7*, *20*, *23*, *36*, *52*, *55*
cvRun *(*cvRun-class*)*, 21
cvRun-class, 21
cvSettings, *8*, *20*, *22*, *27, 28*, *37*, *53*, *56*
cvSettings *(*cvSettings-class*)*, 22
cvSettings-class, 22

dataset, *46*, *88, 89*
dataset *(*dataset-class*)*, 23
dataset-class, 23
dist.to.knn, 24
DMwR *(*DMwR-package*)*, 3
DMwR-package, 3
dsNames, 25, *47*, *86*

experimentalComparison, *5*, *9*, *16, 17*, *20*, *25*, 26, *29*, *31, 32*, *39*, *41*, *47*, *51*, *58*, *66*, *86, 87*, *103*
expSettings, *7, 8*, *23*, *37*, *53*, *56*
expSettings *(*expSettings-class*)*, 28
expSettings-class, 28

getFoldsResults, 29, *31*
getSummaryResults, *29*, 30
getVariant, 31

growingWindowTest, 33, *58*, *81*
GSPC, 35

hldRun, *7*, *22*, *37*, *39*, *52*, *55*
hldRun *(*hldRun-class*)*, 35
hldRun-class, 35
hldSettings, *8*, *23*, *27, 28*, *36*, *39*, *53*, *56*
hldSettings *(*hldSettings-class*)*, 36
hldSettings-class, 36
holdOut, *9*, *20*, *27*, 37, *51*, *58*

join, *17*, 40

kNN, 42
knn, *43*
knn.cv, *43*
knn1, *43*
knneigh.vect, 43
knnImputation, *11*, 44

learner, *24*, *75*, *89*, *103*
learner *(*learner-class*)*, 46
learner-class, 46
learnerNames, *25*, 47, *86*
LinearScaling, 47, *70*, *85*
lofactor, *25*, *44*, 48, *67*
loocv, *9*, *20*, *26, 27*, *39*, 49, *58*
loocvRun, *7*, *22*, *36*, *51*, *53*, *55*
loocvRun *(*loocvRun-class*)*, 52
loocvRun-class, 52
loocvSettings, *8*, *23*, *27, 28*, *37*, *51, 52*, *56*
loocvSettings *(*loocvSettings-class*)*, 53
loocvSettings-class, 53

manyNAs, 54
mcRun, *7*, *22*, *36*, *52*, *56*, *58*
mcRun *(*mcRun-class*)*, 55
mcRun-class, 55
mcSettings, *8*, *23*, *27, 28*, *37*, *53*, *55*, *58*
mcSettings *(*mcSettings-class*)*, 56
mcSettings-class, 56
mean, *12*
median, *12*
monteCarlo, *9*, *20*, *27*, *34*, *39*, *51*, 57, *81*

na.omit, *11*, *45*, *54*

outliers.ranking, 59

performance, *18*, *63*

plot,compExp,missing-method (compExp-class), 16
plot,cvRun,missing-method (cvRun-class), 21
plot,hldRun,missing-method (hldRun-class), 35
plot,mcRun,missing-method (mcRun-class), 55
plot,tradeRecord,ANY-method (tradeRecord-class), 90
plot.rpart, *65*
PRcurve, 62
prediction, *18*, *63*
prettyTree, 64
prune.rpart, *73, 74*

rankSystems, *5*, *17*, *27*, 65, *87*
reachability, 67
regr.eval, *14*, 68
ReScaling, *48*, 70, *85*
resp, 71
rpart, *65*, *73, 74*
rpartXse, *65*, 72
rt.prune, *73*, 73, *74*
runLearner, *46*, 74

sales, 76
scale, *48*, *70*, *85*, *102*
SelfTrain, 76
show,bootSettings-method (bootSettings-class), 7
show,compExp-method (compExp-class), 16
show,cvSettings-method (cvSettings-class), 22
show,dataset-method (dataset-class), 23
show,hldSettings-method (hldSettings-class), 36
show,learner-method (learner-class), 46
show,loocvSettings-method (loocvSettings-class), 53
show,mcSettings-method (mcSettings-class), 56
show,task-method (task-class), 88
show,tradeRecord-method (tradeRecord-class), 90
sigs.PR, 79, *94*
slidingWindowTest, *34*, *58*, 80
SMOTE, 82
SoftMax, *48*, *70*, 84
statNames, *25*, *47*, 85
statScores, *5*, *17*, *66*, 86
subset, *41*
subset,compExp-method (subset-methods), 87
subset-methods, 87
summary,bootRun-method (bootRun-class), 6
summary,compExp-method (compExp-class), 16
summary,cvRun-method (cvRun-class), 21
summary,hldRun-method (hldRun-class), 35
summary,loocvRun-method (loocvRun-class), 52
summary,mcRun-method (mcRun-class), 55
summary,tradeRecord-method (tradeRecord-class), 90

task, *23, 24*, *46*
task (task-class), 88
task-class, 88
test.algae, 89
text.rpart, *65*
tradeRecord, *94*, *97*
tradeRecord (tradeRecord-class), 90
tradeRecord-class, 90
trading.signals, *80*, 91, *92*, *94*, *97*
trading.simulator, *80*, *91, 92*, 92, *97*
tradingEvaluation, *80*, *91, 92*, *94*, 96
ts.eval, 99

unscale, 101

variants, *27*, *32*, 102