Package ‘caret’

February 16, 2014

Version 6.0-24

Date 2014-02-15

Title Classification and Regression Training

Author Max Kuhn. Contributions from Jed Wing, Steve Weston, Andre Williams, Chris Keefer, Allan Engelhardt, Tony Cooper, Zachary Mayer and the R Core Team

Description Misc functions for training and plotting classification and regression models

Maintainer Max Kuhn <Max.Kuhn@pfizer.com>

Depends R (>= 2.10), stats, lattice (>= 0.20), ggplot2

URL http://caret.r-forge.r-project.org/

Imports car, reshape2, foreach, methods, plyr

Suggests e1071, earth (>= 2.2-3), ellipse, fastICA, gam, ipred,kernlab, klaR, MASS, mda, mgcv, mlbench, nnet, party (>= 0.9-999992), pls, pROC, proxy, randomForest, RANN, spls, affy

License GPL-2

NeedsCompilation yes

Repository CRAN

Date/Publication 2014-02-16 08:25:18

R topics documented:

Alternate Affy Gene Expression Summary Methods. 3
as.table.confusionMatrix 4
avNNet.default 6
bag.default 7
bagEarth 10
bagFDA 12
BloodBrain 13
R topics documented:

BoxCoxTrans.default ........................................... 14
calibration .................................................... 16
caretFuncs ..................................................... 18
caretSBF ....................................................... 20
cars ........................................................... 22
classDist ....................................................... 22
calibration ...................................................... 24
calibration ...................................................... 27
cox2 .......................................................... 29
createDataPartition ........................................... 29
dhfr ............................................................ 31
diff.resamples ............................................... 32
dotPlot ........................................................ 34
dotplot.diff.resamples ....................................... 35
downSample .................................................. 36
dummyVars ...................................................... 37
featurePlot ..................................................... 40
filterVarImp ................................................... 42
findCorrelation ............................................... 43
findLinearCombos ............................................. 44
format.bagEarth .............................................. 45
GermanCredit .................................................. 46
histogram.train ............................................... 47
icr.formula ................................................... 48
knn3 .......................................................... 50
knnreg ........................................................ 52
lattice.rfe .................................................... 53
lift ............................................................. 55
maxDissim ..................................................... 57
mdrr .......................................................... 59
modelLookup ................................................... 60
nearZeroVar ................................................... 62
normalize.AffyBatch.normalize2Reference ............. 64
normalize2Reference .......................................... 65
nullModel ...................................................... 66
nullModel ...................................................... 68
oneSE ......................................................... 68
panel.lift2 .................................................... 71
panel.needle .................................................. 72
pcaNNet.default .............................................. 73
plot.rfe ....................................................... 75
plot.train ..................................................... 76
plot.varImp.train ........................................... 78
plotClassProbs ................................................ 79
plotObsVsPred ................................................ 80
plSda ........................................................ 82
postResample ................................................ 85
pottery ........................................................ 87
Generate Expression Values from Probes

Description

Generate an expression from the probes

Usage

generateExprVal.method.trimMean(probes, trim = 0.15)

Arguments

probes a matrix of probe intensities with rows representing probes and columns representing samples. Usually pm(probeset) where probeset is a of class ProbeSet

trim the fraction (0 to 0.5) of observations to be trimmed from each end of the data before the mean is computed.
Value

A list containing entries:

exprs  The expression values.
se.exprs  The standard error estimate.

See Also

generateExprSet-methods

Examples

## Not run:
# first, let affy/expresso know that the method exists
express.summary.stat.methods <- c(express.summary.stat.methods, "trimMean")

example not run, as it would take a while
RawData <- ReadAffy(celfile.path=FilePath)

expresso(RawData,  
bgcorrect.method="rma",  
normalize.method="quantiles",  
normalize.param = list(type= "pmonly"),  
pmcorrect.method="pmonly",  
summary.method="trimMean")

step1 <- bg.correct(RawData, "rma")
step2 <- normalize.AffyBatch.quantiles(step1)
step3 <- computeExprSet(step2, "pmonly", "trimMean")

## End(Not run)

---

as.table.confusionMatrix

Save Confusion Table Results

Description

Conversion functions for class confusionMatrix

Usage

## S3 method for class 'confusionMatrix'
as.matrix(x, what = "xtabs", ...)

## S3 method for class 'confusionMatrix'
as.table(x, ...)

Arguments

x             an object of class \texttt{confusionMatrix}
what          data to convert to matrix. Either "xtabs", "overall" or "classes"
...            not currently used

Details

For \texttt{as.table}, the cross-tabulations are saved. For \texttt{as.matrix}, the three object types are saved in matrix format.

Value

A matrix or table

Author(s)

Max Kuhn

See Also

\texttt{confusionMatrix}

Examples

\vspace{1em}
\vspace{1em}

\begin{verbatim}
### 2 class example

lvs <- c("normal", "abnormal")
truth <- factor(rep(lvs, times = c(86, 258)),
    levels = rev(lvs))
pred <- factor(
    c(
        rep(lvs, times = c(54, 32)),
        rep(lvs, times = c(27, 231))),
    levels = rev(lvs))

xtab <- table(pred, truth)

results <- confusionMatrix(xtab)
as.table(results)
as.matrix(results)
as.matrix(results, what = "overall")
as.matrix(results, what = "classes")

### 3 class example

xtab <- confusionMatrix(iris$Species, sample(iris$Species))
as.matrix(xtab)
\end{verbatim}
Description

Aggregate several neural network models.

Usage

## Default S3 method:
```
avnNet(x, y, repeats = 5, bag = FALSE, allowParallel = TRUE, ...)
```

## S3 method for class 'formula'
```
avnNet(formula, data, weights, ..., 
repeats = 5, bag = FALSE, allowParallel = TRUE, 
subset, na.action, contrasts = NULL)
```

## S3 method for class 'avnNet'
```
predict(object, newdata, type = c("raw", "class", "prob"), ...)
```

Arguments

- `formula` A formula of the form `class ~ x1 + x2 + ...`
- `x` matrix or data frame of x values for examples.
- `y` matrix or data frame of target values for examples.
- `weights` (case) weights for each example – if missing defaults to 1.
- `repeats` the number of neural networks with with different random number seeds
- `bag` a logical for bagging for each repeat
- `allowParallel` if a parallel backend is loaded and available, should the function use it?
- `data` Data frame from which variables specified in `formula` are preferentially to be taken.
- `subset` An index vector specifying the cases to be used in the training sample. (NOTE: If given, this argument must be named.)
- `na.action` A function to specify the action to be taken if NAs are found. The default action is for the procedure to fail. An alternative is `na.omit`, which leads to rejection of cases with missing values on any required variable. (NOTE: If given, this argument must be named.)
- `contrasts` a list of contrasts to be used for some or all of the factors appearing as variables in the model formula.
- `object` an object of class `avnNet` as returned by `avnNet`.
- `newdata` matrix or data frame of test examples. A vector is considered to be a row vector comprising a single case.
- `type` Type of output, either: `raw` for the raw outputs, `code` for the predicted class or `prob` for the class probabilities.
- `...` arguments passed to `nnet`
Details
Following Ripley (1996), the same neural network model is fit using different random number seeds. All of the resulting models are used for prediction. For regression, the output from each network are averaged. For classification, the model scores are first averaged, then translated to predicted classes. Bagging can also be used to create the models. If a parallel backend is registered, the foreach package is used to train the networks in parallel.

Value
For avNNNet, an object of "avNNNet" or "avNNNet.formula". Items of interest in the output are:

- **model**: a list of the models generated from nnet
- **repeats**: an echo of the model input
- **names**: if any predictors had only one distinct value, this is a character string of the remaining columns. Otherwise a value of NULL

Author(s)
These are heavily based on the nnet code from Brian Ripley.

References

See Also
nnet, preProcess

Examples
```r
data(BloodBrain)
## Not run:
modelFit <- avNNNet(bbbDescr, logBBB, size = 5, linout = TRUE, trace = FALSE)
modelFit

predict(modelFit, bbbDescr)
## End(Not run)
```

Description
bag provides a framework for bagging classification or regression models. The user can provide their own functions for model building, prediction and aggregation of predictions (see Details below).
Usage

```r
bag(x, ...)
```

```r
## Default S3 method:
bag(x, y, B = 10, vars = ncol(x), bagControl = NULL, ...)
```

```r
bagControl(fit = NULL,
           predict = NULL,
           aggregate = NULL,
           downSample = FALSE,
           oob = TRUE,
           allowParallel = TRUE)
```

```r
ldabag
plsbag
nbBag
ctreeBag
svmbag
nnetBag
```

```r
## S3 method for class 'bag'
predict(object, newdata = NULL, ...)
```

Arguments

- **x**
  - a matrix or data frame of predictors
- **y**
  - a vector of outcomes
- **B**
  - the number of bootstrap samples to train over.
- **bagControl**
  - a list of options.
- **...**
  - arguments to pass to the model function
- **fit**
  - a function that has arguments `x`, `y` and `...` and produces a model object that can later be used for prediction. Example functions are found in `ldabag`, `plsbag`, `nbBag`, `ctreeBag`, `svmbag` and `nnetBag`.
- **predict**
  - a function that generates predictions for each sub-model. The function should have arguments `object` and `x`. The output of the function can be any type of object (see the example below where posterior probabilities are generated. Example functions are found in `ldabag`, `plsbag`, `nbBag`, `svmbag` and `nnetBag`).
- **aggregate**
  - a function with arguments `x` and `type`. The function that takes the output of the predict function and reduces the bagged predictions to a single prediction per sample. the type argument can be used to switch between predicting classes or class probabilities for classification models. Example functions are found in `ldabag`, `plsbag`, `nbBag`, `svmbag` and `nnetBag`.
- **downSample**
  - a logical: for classification, should the data set be randomly sampled so that each class has the same number of samples as the smallest class?
- **oob**
  - a logical: should out-of-bag statistics be computed and the predictions retained?
allowParallel  if a parallel backend is loaded and available, should the function use it?
vars an integer. If this argument is not NULL, a random sample of size vars is taken of the predictors in each bagging iteration. If NULL, all predictors are used.
object an object of class `bag`.
newdata a matrix or data frame of samples for prediction. Note that this argument must have a non-null value

Details

The function is basically a framework where users can plug in any model in to assess the effect of bagging. Examples functions can be found in `ldaBag`, `plsBag`, `nbBag`, `svmBag` and `nnetBag`. Each has elements fit, pred and aggregate.

One note: when vars is not NULL, the sub-setting occurs prior to the fit and predict functions are called. In this way, the user probably does not need to account for the change in predictors in their functions.

When using bag with `train`, classification models should use type = "prob" inside of the predict function so that `predict.train(object, newdata, type = "prob")` will work.

If a parallel backend is registered, the `foreach` package is used to train the models in parallel.

Value

`bag` produces an object of class `bag` with elements

fits a list with two sub-objects: the fit object has the actual model fit for that bagged samples and the vars object is either NULL or a vector of integers corresponding to which predictors were sampled for that model
control a mirror of the arguments passed into `bagControl`
call the call
B the number of bagging iterations
dims the dimensions of the training set

Author(s)

Max Kuhn

Examples

```r
## A simple example of bagging conditional inference regression trees:
data(BloodBrain)

## treebag <- bag(bbbDescr, logBBB, B = 10,
##                bagControl = bagControl(fit = ctreeBag$fit,
##                predict = ctreeBag$pred,
##                aggregate = ctreeBag$aggregate))
```
## bagEarth

**Bagged Earth**

### Description

A bagging wrapper for multivariate adaptive regression splines (MARS) via the earth function

### Usage

```r
## S3 method for class 'formula'
bagEarth(formula, data = NULL, B = 50,
         summary = mean, keepX = TRUE,
         ..., subset, weights, na.action = na.omit)
## Default S3 method:
bagEarth(x, y, weights = NULL, B = 50,
         summary = mean, keepX = TRUE, ...)
```

### Arguments

- **formula**
  - A formula of the form `y ~ x1 + x2 + ...`

- **x**
  - matrix or data frame of ’x’ values for examples.

- **y**
  - matrix or data frame of numeric values outcomes.

- **weights**
  - (case) weights for each example - if missing defaults to 1.

- **data**
  - Data frame from which variables specified in ’formula’ are preferentially to be taken.

- **subset**
  - An index vector specifying the cases to be used in the training sample. (NOTE: If given, this argument must be named.)
**na.action**  A function to specify the action to be taken if 'NA's are found. The default action is for the procedure to fail. An alternative is na.omit, which leads to rejection of cases with missing values on any required variable. (NOTE: If given, this argument must be named.)

**B**  the number of bootstrap samples

**summary**  a function with a single argument specifying how the bagged predictions should be summarized

**keepX**  a logical: should the original training data be kept?

...  arguments passed to the earth function

**Details**

The function computes a Earth model for each bootstrap sample.

**Value**

A list with elements

- **fit**  a list of B Earth fits
- **B**  the number of bootstrap samples
- **call**  the function call
- **x**  either NULL or the value of x, depending on the value of keepX
- **oob**  a matrix of performance estimates for each bootstrap sample

**Author(s)**

Max Kuhn (bagEarth.formula is based on Ripley’s nnet.formula)

**References**


**See Also**

earth, predict.bagEarth

**Examples**

```r
## Not run:
library(mda)
library(earth)
data(trees)
fit1 <- earth(trees[, -3], trees[, 3])
fit2 <- bagEarth(trees[, -3], trees[, 3], B = 10)

## End(Not run)
```
Description

A bagging wrapper for flexible discriminant analysis (FDA) using multivariate adaptive regression splines (MARS) basis functions

Usage

`bagFDA(x, ...)`

## S3 method for class 'formula'
`bagFDA(formula, data = NULL, B = 50, keepX = TRUE, ...`, subset, weights, na.action = na.omit)

## Default S3 method:
`bagFDA(x, y, weights = NULL, B = 50, keepX = TRUE, ...)`

Arguments

- `formula` A formula of the form `y ~ x1 + x2 + ...`
- `x` matrix or data frame of `x` values for examples.
- `y` matrix or data frame of numeric values outcomes.
- `weights` (case) weights for each example - if missing defaults to 1.
- `data` Data frame from which variables specified in `formula` are preferentially to be taken.
- `subset` An index vector specifying the cases to be used in the training sample. (NOTE: If given, this argument must be named.)
- `na.action` A function to specify the action to be taken if `NA's` are found. The default action is for the procedure to fail. An alternative is `na.omit`, which leads to rejection of cases with missing values on any required variable. (NOTE: If given, this argument must be named.)
- `B` the number of bootstrap samples
- `keepX` a logical: should the original training data be kept?
- ... arguments passed to the `mars` function

Details

The function computes a FDA model for each bootstrap sample.

Value

A list with elements

- `fit` a list of `B` FDA fits
- `B` the number of bootstrap samples
call x the function call either NULL or the value of x, depending on the value of keepX

oob a matrix of performance estimates for each bootstrap sample

Author(s)

Max Kuhn (bagFDA.formula is based on Ripley’s nnet.formula)

References


See Also

fda, predict.bagFDA

Examples

library(mlbench)
library(earth)
data(Glass)

set.seed(36)
inTrain <- sample(1:dim(Glass)[1], 150)

trainData <- Glass[ inTrain, ]
 testData <- Glass[-inTrain, ]

baggedFit <- bagFDA(Type ~ ., trainData)
confusionMatrix(predict(baggedFit, testData[, -10]),
 testData[, 10])

BloodBrain

Blood Brain Barrier Data

Description

Mente and Lombardo (2005) develop models to predict the log of the ratio of the concentration of a compound in the brain and the concentration in blood. For each compound, they computed three sets of molecular descriptors: MOE 2D, rule-of-five and Charge Polar Surface Area (CPSA). In all, 134 descriptors were calculated. Included in this package are 208 non-proprietary literature compounds. The vector logBBB contains the concentration ratio and the data frame bbbDescr contains the descriptor values.
**Usage**

```r
data(BloodBrain)
```

**Value**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbbDescr</td>
<td>data frame of chemical descriptors</td>
</tr>
<tr>
<td>logBBB</td>
<td>vector of assay results</td>
</tr>
</tbody>
</table>

**Source**


---

**BoxCoxTrans.default**  
*Box-Cox and Exponential Transformations*

**Description**

These classes can be used to estimate transformations and apply them to existing and future data

**Usage**

```r
BoxCoxTrans(y, ...)  
expotrans(y, ...)
```

```r
## Default S3 method:  
BoxCoxTrans(y, x = rep(1, length(y)),  
fudge = 0.2, numUnique = 3, na.rm = FALSE, ...)
## Default S3 method:  
expotrans(y, na.rm = TRUE, init = 0,  
lim = c(-4, 4), method = "Brent",  
umUnique = 3, ...)
## S3 method for class 'BoxCoxTrans'  
predict(object, newdata, ...)
## S3 method for class 'expotrans'  
predict(object, newdata, ...)
```

**Arguments**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>a numeric vector of data to be transformed. For BoxCoxTrans, the data must be strictly positive.</td>
</tr>
<tr>
<td>x</td>
<td>an optional dependent variable to be used in a linear model.</td>
</tr>
<tr>
<td>fudge</td>
<td>a tolerance value: lambda values within +/-fudge will be coerced to 0 and within 1+/-fudge will be coerced to 1.</td>
</tr>
<tr>
<td>numUnique</td>
<td>how many unique values should y have to estimate the transformation?</td>
</tr>
</tbody>
</table>
na.rm  a logical value indicating whether NA values should be stripped from \( y \) and \( x \) before the computation proceeds.

init, lim, method  
initial values, limits and optimization method for \texttt{optim}.

...  for \texttt{BoxCoxTrans}: options to pass to \texttt{boxcox}, \texttt{plotit} should not be passed through. For \texttt{predict.BoxCoxTrans}, additional arguments are ignored.

object  an object of class \texttt{BoxCoxTrans} or \texttt{expTrans}.

newdata  a numeric vector of values to transform.

Details

BoxCoxTrans function is basically a wrapper for the \texttt{boxcox} function in the MASS library. It can be used to estimate the transformation and apply it to new data.

\texttt{expTrans} estimates the exponential transformation of Manly (1976) but assumes a common mean for the data. The transformation parameter is estimated by directly maximizing the likelihood.

If any\( (y \leq 0) \) or if \texttt{length(unique(y))} < \texttt{numUnique}, lambda is not estimated and no transformation is applied.

Value

Both functions returns a list of class of either \texttt{BoxCoxTrans} or \texttt{expTrans} with elements

- \texttt{lambda}estimated transformation value
- \texttt{fudge}value of fudge
- \texttt{n}number of data points used to estimate lambda
- \texttt{summary}the results of \texttt{summary(y)}
- \texttt{ratio}max\( (y)/min(y)\)
- \texttt{skewness}sample skewness statistic

BoxCoxTrans also returns:

- \texttt{fudge}value of fudge

The predict functions returns numeric vectors of transformed values

Author(s)

Max Kuhn

References


See Also

\texttt{boxcox}, \texttt{preProcess}, \texttt{optim}
**calibration**

**Examples**

```r
data(BloodBrain)

ratio <- exp(logBBB)
bc <- BoxCoxTrans(ratio)
bc

predict(bc, ratio[1:5])

ratio[5] <- NA
bc2 <- BoxCoxTrans(ratio, bbbDescr$tpsa, na.rm = TRUE)
bc2

manly <- expotrans(ratio)
manly
```

---

**calibration**  
*Probability Calibration Plot*

**Description**

For classification models, this function creates a 'calibration plot' that describes how consistent model probabilities are with observed event rates.

**Usage**

```r
calibration(x, ...)
```

```r
# S3 method for class 'formula'
calibration(x, data = NULL,  
    class = NULL,  
    cuts = 11, subset = TRUE,  
    lattice.options = NULL, ...)
```

```r
# S3 method for class 'calibration'
xyplot(x, data, ...)

panel.calibration(...)
```

**Arguments**

- `x`  
  a lattice formula (see `xyplot` for syntax) where the left-hand side of the formula is a factor class variable of the observed outcome and the right-hand side specifies one or model columns corresponding to a numeric ranking variable for a model (e.g. class probabilities). The classification variable should have two levels.
For calibration, formula, a data frame (or more precisely, anything that is a valid envir argument in eval, e.g., a list or an environment) containing values for any variables in the formula, as well as groups and subset if applicable. If not found in data, or if data is unspecified, the variables are looked for in the environment of the formula. This argument is not used for xyplot.calibration.

class: a character string for the class of interest
cuts: the number of splits of the data are used to create the plot. By default, it uses as many cuts as there are rows in data
subset: An expression that evaluates to a logical or integer indexing vector. It is evaluated in data. Only the resulting rows of data are used for the plot.
lattice.options: A list that could be supplied to lattice.options

Details

To construct the calibration plot, the following steps are used for each model:

1. The data are split into cuts - 1 roughly equal groups by their class probabilities
2. the number of samples with true results equal to class are determined
3. the event rate is determined for each bin

xyplot.calibration produces a plot of the observed event rate by the mid-point of the bins.

This implementation uses the lattice function xyplot, so plot elements can be changed via panel functions, trellis.par.set or other means. calibration uses the panel function panel.calibration by default, but it can be changed by passing that argument into xyplot.calibration.

The following elements are set by default in the plot but can be changed by passing new values into xyplot.calibration: xlab = "Bin Midpoint", ylab = "Observed Event Percentage", type = "o", ylim = extendrange(c(0, 100)), xlim = extendrange(c(0, 100)) and panel = panel.calibration

Value

calibration.formula returns a list with elements:

data: the data used for plotting
cuts: the number of cuts
class: the event class
probNames: the names of the model probabilities

xyplot.calibration returns a lattice object
Author(s)

Max Kuhn, some lattice code and documentation by Deepayan Sarkar

See Also

xyplot, trellis.par.set

Examples

```r
## Not run:
data(mdrri)
mdrrDescr <- mdrrDescr[, nearZeroVar(mdrrDescr)]
mdrrDescr <- mdrrDescr[, -findCorrelation(cor(mdrrDescr), .5)]

inTrain <- createDataPartition(mdrrClass)
trainX <- mdrrDescr[inTrain[[1]], ,]
trainY <- mdrrClass[inTrain[[1]]]
testX <- mdrrDescr[-inTrain[[1]], ,]
testY <- mdrrClass[-inTrain[[1]]]

library(MASS)

ldaFit <- lda(trainX, trainY)
qdaFit <- qda(trainX, trainY)

testProbs <- data.frame(obs = testY,
                        lda = predict(ldaFit, testX)$posterior[,1],
                        qda = predict(qdaFit, testX)$posterior[,1])

calibration(obs ~ lda + qda, data = testProbs)
calPlotData <- calibration(obs ~ lda + qda, data = testProbs)
calPlotData
xyplot(calPlotData, auto.key = list(columns = 2))

## End(Not run)
```

---

caretFuncs Backwards Feature Selection Helper Functions

Description

Ancillary functions for backwards selection
Usage

pickSizeTolerance(x, metric, tol = 1.5, maximize)
pickSizeBest(x, metric, maximize)
pickVars(y, size)

caretFuncs
lmFuncs
rffuncs
treebagFuncs
ldafuncs
nbFuncs
gamFuncs
lrFuncs

Arguments

x a matrix or data frame with the performance metric of interest
metric a character string with the name of the performance metric that should be used to choose the appropriate number of variables
maximize a logical; should the metric be maximized?
tol a scalar to denote the acceptable difference in optimal performance (see Details below)
y a list of data frames with variables overall and var
size an integer for the number of variables to retain

Details

This page describes the functions that are used in backwards selection (aka recursive feature elimination). The functions described here are passed to the algorithm via the functions argument of rfeControl. See rfeControl for details on how these functions should be defined.

The 'pick' functions are used to find the appropriate subset size for different situations. pickBest will find the position associated with the numerically best value (see the maximize argument to help define this).

pickSizeTolerance picks the lowest position (i.e. the smallest subset size) that has no more of an X percent loss in performances. When maximizing, it calculates (O-X)/O*100, where X is the set of performance values and O is max(X). This is the percent loss. When X is to be minimized, it uses (X-O)/O*100 (so that values greater than X have a positive "loss"). The function finds the smallest subset size that has a percent loss less than tol.

Both of the 'pick' functions assume that the data are sorted from smallest subset size to largest.

Author(s)

Max Kuhn
See Also

caretSBF, rfe

Examples

## For picking subset sizes:
## Minimize the RMSE
example <- data.frame(RMSE = c(1.2, 1.1, 1.05, 1.01, 1.01, 1.03, 1.00),
  Variables = 1:7)
## Percent Loss in performance (positive)
example$PctLoss <- (example$RMSE - min(example$RMSE))/min(example$RMSE)*100

xyplot(RMSE ~ Variables, data= example)
xyplot(PctLoss ~ Variables, data= example)

absoluteBest <- pickSizeBest(example, metric = "RMSE", maximize = FALSE)
within5Pct <- pickSizeTolerance(example, metric = "RMSE", maximize = FALSE)

cat("numerically optimal: ",
  example$RMSE[absoluteBest],
  "RMSE in position",
  absoluteBest, "\n")
cat("Accepting a 1.5 pctl loss: ",
  example$RMSE[within5Pct],
  "RMSE in position",
  within5Pct, "\n")

## Example where we would like to maximize
example2 <- data.frame(Rsquared = c(0.4, 0.6, 0.94, 0.95, 0.95, 0.95, 0.95),
  Variables = 1:7)
## Percent Loss in performance (positive)
example2$PctLoss <- (max(example2$Rsquared) - example2$Rsquared)/max(example2$Rsquared)*100

xyplot(Rsquared ~ Variables, data= example2)
xyplot(PctLoss ~ Variables, data= example2)

absoluteBest2 <- pickSizeBest(example2, metric = "Rsquared", maximize = TRUE)
within5Pct2 <- pickSizeTolerance(example2, metric = "Rsquared", maximize = TRUE)

cat("numerically optimal: ",
  example2$Rsquared[absoluteBest2],
  "R^2 in position",
  absoluteBest2, "\n")
cat("Accepting a 1.5 pct loss: ",
  example2$Rsquared[within5Pct2],
  "R^2 in position",
  within5Pct2, "\n")

---

caretSBF Selection By Filtering (SBF) Helper Functions
Description

Ancillary functions for univariate feature selection

Usage

anovascores(x, y)
gamscores(x, y)
caretSBF
lmSBF
rfSBF
treebagSBF
ldaSBF
nbSBF

Arguments

x a matrix or data frame of numeric predictors
y a numeric or factor vector of outcomes

Details

More details on these functions can be found at http://caret.r-forge.r-project.org/featureselection.html.

This page documents the functions that are used in selection by filtering (SBF). The functions described here are passed to the algorithm via the functions argument of sbfControl.

See sbfControl for details on how these functions should be defined.

anovaScores and gamsScores are two examples of univariate filtering functions. anovaScores fits a simple linear model between a single feature and the outcome, then the p-value for the whole model F-test is returned. gamsScores fits a generalized additive model between a single predictor and the outcome using a smoothing spline basis function. A p-value is generated using the whole model test from summary.gam and is returned.

If a particular model fails for lm or gam, a p-value of 1 is returned.

Author(s)

Max Kuhn

See Also

sbfControl, sbf, summary.gam
**classDist**

**cars**  
*Kelly Blue Book resale data for 2005 model year GM cars*

**Description**


**Usage**

```r
data(cars)
```

**Value**

cars  
data frame of the suggested retail price (column `Price`) and various characteristics of each car (columns `Mileage`, `Cylinder`, `Doors`, `Cruise`, `Sound`, `Leather`, `Buick`, `Cadillac`, `Chevy`, `Pontiac`, `Saab`, `Saturn`, `convertible`, `coupe`, `hatchback`, `sedan` and `wagon`)

**Source**


---

**classDist**  
*Compute and predict the distances to class centroids*

**Description**

This function computes the class centroids and covariance matrix for a training set for determining Mahalanobis distances of samples to each class centroid.

**Usage**

```r
classDist(x, ...)
```

### S3 method:
```
classDist(x, y, groups = 5, pca = FALSE, keep = NULL, ...)
```

### S3 method for class 'classDist'
```
predict(object, newdata, trans = log, ...)
```
classDist

Arguments

- **x**: a matrix or data frame of predictor variables
- **y**: a numeric or factor vector of class labels
- **groups**: an integer for the number of bins for splitting a numeric outcome
- **pca**: a logical: should principal components analysis be applied to the dataset prior to splitting the data by class?
- **keep**: an integer for the number of PCA components that should be used to predict new samples (NULL uses all within a tolerance of $\sqrt{\text{Machine\\$double.\eps}}$)
- **object**: an object of class `classDist`
- **newdata**: a matrix or data frame. If `vars` was previously specified, these columns should be in `newdata`
- **trans**: an optional function that can be applied to each class distance. `trans = NULL` will not apply a function
- **...**: optional arguments to pass (not currently used)

Details

For factor outcomes, the data are split into groups for each class and the mean and covariance matrix are calculated. These are then used to compute Mahalanobis distances to the class centers (using `predict.classDist`). The function will check for non-singular matrices.

For numeric outcomes, the data are split into roughly equal sized bins based on `groups`. Percentiles are used to split the data.

Value

- **for `classDist`**, an object of class `classDist` with elements:
  - **values**: a list with elements for each class. Each element contains a mean vector for the class centroid and the inverse of the class covariance matrix
  - **classes**: a character vector of class labels
  - **pca**: the results of `prcomp` when `pca = TRUE`
  - **call**: the function call
  - **p**: the number of variables
  - **n**: a vector of samples sizes per class

For `predict.classDist`, a matrix with columns for each class. The columns names are the names of the class with the prefix `dist...`. In the case of numeric `y`, the class labels are the percentiles. For example, of `groups = 9`, the variable names would be `dist.11.11`, `dist.22.22`, etc.

Author(s)

Max Kuhn
confusionMatrix

References

Forina et al. CAIMAN brothers: A family of powerful classification and class modeling techniques. Chemometrics and Intelligent Laboratory Systems (2009) vol. 96 (2) pp. 239-245

See Also

mahalanobis

Examples

trainSet <- sample(1:150, 100)
distData <- classDist(iris[trainSet, 1:4], iris$Species[trainSet])
newDist <- predict(distData, iris[-trainSet, 1:4])
splom(newDist, groups = iris$Species[-trainSet])

confusionMatrix(data, ...)  

## Default S3 method:
confusionMatrix(data, reference, positive = NULL,  
dnn = c("Prediction", "Reference"),  
prevalence = NULL, ...)

## S3 method for class 'table'
confusionMatrix(data, positive = NULL, prevalence = NULL, ...)

Arguments

data a factor of predicted classes (for the default method) or an object of class table.
reference a factor of classes to be used as the true results
positive an optional character string for the factor level that corresponds to a "positive" result (if that makes sense for your data). If there are only two factor levels, the first level will be used as the "positive" result.
confusionMatrix

- **dnn**: a character vector of dimnames for the table
- **prevalence**: a numeric value or matrix for the rate of the "positive" class of the data. When data has two levels, prevalence should be a single numeric value. Otherwise, it should be a vector of numeric values with elements for each class. The vector should have names corresponding to the classes.

... options to be passed to `table`. NOTE: do not include `dnn` here

### Details

The functions require that the factors have exactly the same levels.

For two-class problems, the sensitivity, specificity, positive predictive value and negative predictive value is calculated using the `positive` argument. Also, the prevalence of the "event" is computed from the data (unless passed in as an argument), the detection rate (the rate of true events also predicted to be events) and the detection prevalence (the prevalence of predicted events).

Suppose a 2x2 table with notation

<table>
<thead>
<tr>
<th>Reference</th>
<th>Predicted</th>
<th>Event</th>
<th>No Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>No Event</td>
<td>C</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

The formulas used here are:

- **Sensitivity** = $A/(A + C)$
- **Specificity** = $D/(B + D)$
- **Prevalence** = $(A + C)/(A + B + C + D)$
- **PPV** = $(sensitivity*Prevalence)/(((sensitivity*Prevalence)+((1-specificity)*(1-Prevalence))))$
- **NPV** = $(specificity*(1-Prevalence))/(((1-sensitivity)*Prevalence)+((specificity)*(1-Prevalence)))$
- **Detection Rate** = $A/(A + B + C + D)$
- **Detection Prevalence** = $(A + B)/(A + B + C + D)$
- **Balanced Accuracy** = $(Sensitivity + Specificity)/2$

See the references for discussions of the first five formulas.

For more than two classes, these results are calculated comparing each factor level to the remaining levels (i.e. a "one versus all" approach).

The overall accuracy and unweighted kappa statistic are calculated. A p-value from McNemar’s test is also computed using `mcnemar.test` (which can produce `NA` values with sparse tables).

The overall accuracy rate is computed along with a 95 percent confidence interval for this rate (using `binom.test`) and a one-sided test to see if the accuracy is better than the "no information rate," which is taken to be the largest class percentage in the data.
Value

- a list with elements
  - table: the results of table on data and reference
  - positive: the positive result level
  - overall: a numeric vector with overall accuracy and Kappa statistic values
  - byClass: the sensitivity, specificity, positive predictive value, negative predictive value, prevalence, detection rate, detection prevalence and balanced accuracy for each class. For two class systems, this is calculated once using the positive argument

Author(s)

Max Kuhn

References


See Also

as.table.confusionMatrix, as.matrix.confusionMatrix, sensitivity, specificity, posPredValue, negPredValue, print.confusionMatrix, binom.test

Examples

```
# 2 class example

lvs <- c("normal", "abnormal")
truth <- factor(rep(lvs, times = c(86, 258)),
              levels = rev(lvs))
pred <- factor(
             c(
              rep(lvs, times = c(54, 32)),
              rep(lvs, times = c(27, 231))),
              levels = rev(lvs))

xtab <- table(pred, truth)

confusionMatrix(xtab)

confusionMatrix(pred, truth)
```
confusionMatrix(xtab, prevalence = 0.25)

############################
## 3 class example

confusionMatrix(iris$Species, sample(iris$Species))

newPrior <- c(.05, .8, .15)
names(newPrior) <- levels(iris$Species)

confusionMatrix(iris$Species, sample(iris$Species))

### Estimate a Resampled Confusion Matrix ###

**Description**

Using a `train`, `rfe`, `sbf` object, determine a confusion matrix based on the resampling procedure.

**Usage**

```r
## S3 method for class 'train'
confusionMatrix(data, norm = "overall",
                dnn = c("Prediction", "Reference"), ...)

## S3 method for class 'rfe'
confusionMatrix(data, norm = "overall",
                dnn = c("Prediction", "Reference"), ...)

## S3 method for class 'sbf'
confusionMatrix(data, norm = "overall",
                dnn = c("Prediction", "Reference"), ...)
```

**Arguments**

- `data`: an object of class `train`, `rfe`, `sbf` that did not use out-of-bag resampling or leave-one-out cross-validation.
- `norm`: a character string indicating how the table entries should be normalized. Valid values are "none", "overall" or "average".
- `dnn`: a character vector of dimnames for the table.
- `...`: not used here.
Details

When `train` is used for tuning a model, it tracks the confusion matrix cell entries for the hold-out samples. These can be aggregated and used for diagnostic purposes. For `train`, the matrix is estimated for the final model tuning parameters determined by `train`. For `rfe`, the matrix is associated with the optimal number of variables.

There are several ways to show the table entries. Using `norm = "none"` will show the frequencies of samples on each of the cells (across all resamples). `norm = "overall"` first divides the cell entries by the total number of data points in the table, then averages these percentages. `norm = "average"` takes the raw, aggregate cell counts across resamples and divides by the number of resamples (i.e. to yield an average count for each cell).

Value

A list of class `confusionMatrix.train`, `confusionMatrix.rfe` or `confusionMatrix.sbf` with elements

- **table**: the normalized matrix
- **norm**: an echo of the call
- **text**: a character string with details about the resampling procedure (e.g. "Bootstrapped (25 reps) Confusion Matrix")

Author(s)

Max Kuhn

See Also

`confusionMatrix`, `train`, `rfe`, `sbf`, `trainControl`

Examples

```r
data(iris)
TrainData <- iris[,1:4]
TrainClasses <- iris[,5]

knnFit <- train(TrainData, TrainClasses,
               method = "knn",
               preProcess = c("center", "scale"),
               tuneLength = 10,
               trControl = trainControl(method = "cv"))

confusionMatrix(knnFit)
confusionMatrix(knnFit, "average")
confusionMatrix(knnFit, "none")
```
**Description**

From Sutherland, O'Brien, and Weaver (2003): "A set of 467 cyclooxygenase-2 (COX-2) inhibitors has been assembled from the published work of a single research group, with in vitro activities against human recombinant enzyme expressed as IC50 values ranging from 1 nM to >100 uM (53 compounds have indeterminate IC50 values)."

The data are in the Supplimental Data file for the article.

A set of 255 descriptors (MOE2D and QikProp) were generated. To classify the data, we used a cutoff of $2^{2.5}$ to determine activity.

**Usage**

data(cox2)

**Value**

- **cox2descr** the descriptors
- **cox2IC50** the IC50 data used to determine activity
- **cox2Class** the categorical outcome ("Active" or "Inactive") based on the $2^{2.5}$ cutoff

**Source**


---

**createDataPartition** *Data Splitting functions*

**Description**

A series of test/training partitions are created using `createDataPartition` while `createResample` creates one or more bootstrap samples. `createFolds` splits the data into k groups while `createTimeSlices` creates cross-validation sample information to be used with time series data.
Usage

createDataPartition(y, 
  times = 1, 
  p = 0.5, 
  list = TRUE, 
  groups = min(5, length(y)))
cREATEsampler(y, times = 10, list = TRUE) 
createtimeslices(y, initialWindow, horizon = 1, fixedWindow = TRUE)

Arguments

y a vector of outcomes. For createTimeSlices, these should be in chronological order.
times the number of partitions to create
p the percentage of data that goes to training
list logical - should the results be in a list (TRUE) or a matrix with the number of rows equal to floor(p * length(y)) and times columns.
groups for numeric y, the number of breaks in the quantiles (see below)
k an integer for the number of folds.
returnTrain a logical. When true, the values returned are the sample positions corresponding to the data used during training. This argument only works in conjunction with list = TRUE
initialWindow The initial number of consecutive values in each training set sample
horizon The number of consecutive values in test set sample
fixedWindow A logical: if FALSE, the training set always start at the first sample.

Details

For bootstrap samples, simple random sampling is used.
For other data splitting, the random sampling is done within the levels of y when y is a factor in an attempt to balance the class distributions within the splits.
For numeric y, the sample is split into groups sections based on percentiles and sampling is done within these subgroups. For createDataPartition, the number of percentiles is set via the groups argument. For createFolds and createMultifolds, the number of groups is set dynamically based on the sample size and k. For smaller samples sizes, these two functions may not do stratified splitting and, at most, will split the data into quartiles.
Also, for createDataPartition, very small class sizes (<= 3) the classes may not show up in both the training and test data.
For multiple k-fold cross-validation, completely independent folds are created. The names of the list objects will denote the fold membership using the pattern "FoldRepj" meaning the ith section (of k) of the jth cross-validation set (of times). Note that this function calls createFolds with list = TRUE and returnTrain = TRUE.
Hyndman and Athanasopoulos (2013) discuss rolling forecasting origin techniques that move the training and test sets in time. `createTimeSlices` can create the indices for this type of splitting.

**Value**

A list or matrix of row position integers corresponding to the training data

**Author(s)**

Max Kuhn, `createTimeSlices` by Tony Cooper

**References**

[http://caret.r-forge.r-project.org/splitting.html](http://caret.r-forge.r-project.org/splitting.html)

Hyndman and Athanasopoulos (2013), Forecasting: principles and practice. [https://www.otexts.org/fpp](https://www.otexts.org/fpp)

**Examples**

```r
data(oil)
createDataPartition(oilType, 2)

x <- rgamma(50, 3, .5)
inA <- createDataPartition(x, list = FALSE)

plot(density(x[inA]))
rug(x[inA])

points(density(x[-inA]), type = "l", col = 4)
rug(x[-inA], col = 4)

createResample(oilType, 2)

createFolds(oilType, 10)
createFolds(oilType, 5, FALSE)

createFolds(rnorm(20))

createTimeSlices(1:9, 5, 1, fixedWindow = FALSE)
createTimeSlices(1:9, 5, 1, fixedWindow = TRUE)
createTimeSlices(1:9, 5, 3, fixedWindow = TRUE)
createTimeSlices(1:9, 5, 3, fixedWindow = FALSE)
```

---

**dhfr**

*Dihydrofolate Reductase Inhibitors Data*
Description

Sutherland and Weaver (2004) discuss QSAR models for dihydrofolate reductase (DHFR) inhibition. This data set contains values for 325 compounds. For each compound, 228 molecular descriptors have been calculated. Additionally, each sample is designated as "active" or "inactive". The data frame dhfr contains a column called Y with the outcome classification. The remainder of the columns are molecular descriptor values.

Usage

data(dhfr)

Value

dhfr        data frame of chemical descriptors and the activity values

Source


---

**diff.resamples**

**Inferential Assessments About Model Performance**

Description

Methods for making inferences about differences between models

Usage

```r
## S3 method for class 'resamples'
diff(x, models = x$models, metric = x$metrics,
    test = t.test,
    confLevel = 0.95, adjustment = "bonferroni",
    ...)
```

```r
## S3 method for class 'diff.resamples'
summary(object, digits = max(3,getOption("digits") - 3), ...)
```

Arguments

- `x` an object generated by `resamples`
- `models` a character string for which models to compare
- `metric` a character string for which metrics to compare
- `test` a function to compute differences. The output of this function should have scalar outputs called `estimate` and `p.value`
**diff.resamples**

- **object**
  - a object generated by `diff.resamples`
- **adjustment**
  - any p-value adjustment method to pass to `p.adjust`
- **confLevel**
  - confidence level to use for `dotplot.diff.resamples`. See Details below.
- **digits**
  - the number of significant differences to display when printing
- **...**
  - further arguments to pass to `test`

**Details**


For each metric, all pair-wise differences are computed and tested to assess if the difference is equal to zero.

When a Bonferroni correction is used, the confidence level is changed from `confLevel` to $1 - ((1 - confLevel)/p)$ where $p$ is the number of pair-wise comparisons are being made. For other correction methods, no such change is used.

**Value**

An object of class "diff.resamples" with elements:

- **call**
  - the call
- **difs**
  - a list for each metric being compared. Each list contains a matrix with differences in columns and resamples in rows
- **statistics**
  - a list of results generated by `test`
- **adjustment**
  - the p-value adjustment used
- **models**
  - a character string for which models were compared.
- **metrics**
  - a character string of performance metrics that were used

or...

An object of class "summary.diff.resamples" with elements:

- **call**
  - the call
- **table**
  - a list of tables that show the differences and p-values

**Author(s)**

Max Kuhn

**References**


See Also

resamples, dotplot.diff.resamples, densityplot.diff.resamples, bwplot.diff.resamples, levelplot.diff.resamples

Examples

## Not run:
#load(url("http://caret.r-forge.r-project.org/exampleModels.RData"))

resamps <- resamples(list(CART = rpartFit,
                           CondInfTree = ctreeFit,
                           MARS = earthFit))

difs <- diff(resamps)

difs

summary(difs)

## End(Not run)

---

dotPlot

Create a dotplot of variable importance values

Description

A lattice dotplot is created from an object of class varImp.train.

Usage

dotPlot(x, top = min(20, dim(x$importance)[1]), ...)

Arguments

x

an object of class varImp.train

top

the number of predictors to plot

... options passed to dotplot

Value

an object of class trellis.

Author(s)

Max Kuhn

See Also

varImp, dotplot
Examples

```r
data(iris)
TrainData <- iris[,1:4]
TrainClasses <- iris[,5]

knnFit <- train(TrainData, TrainClasses, "knn")

knnImp <- varImp(knnFit)
dotPlot(knnImp)
```

Description

Lattice functions for visualizing resampling result differences between models

Usage

```r
## S3 method for class 'diff.resamples'
densityplot(x, data, metric = x$metric, ...)

## S3 method for class 'diff.resamples'
bwplot(x, data, metric = x$metric, ...)

## S3 method for class 'diff.resamples'
levelplot(x, data = NULL, metric = x$metric[1], what = "pvalues", ...)

## S3 method for class 'diff.resamples'
dotplot(x, data = NULL, metric = x$metric[1], ...)
```

Arguments

- `x` an object generated by `diff.resamples`
- `data` Not used
- `what` levelplot only: display either the "pvalues" or "differences"
- `metric` a character string for which metrics to plot. Note: dotplot and levelplot require exactly two models whereas the other methods can plot more than two.
- `...` further arguments to pass to either `densityplot`, `dotplot` or `levelplot`
Details

densityplot and bwplot display univariate visualizations of the resampling distributions. levelplot displays the matrix of pair-wide comparisons. dotplot shows the differences along with their associated confidence intervals.

Value

a lattice object

Author(s)

Max Kuhn

See Also

resamples, diff.resamples, bwplot, densityplot, xyplot, splom

Examples

```r
## Not run:
#load(url("http://caret.r-forge.r-project.org/exampleModels.RData"))

resamps <- resamples(list(CART = rpartFit,
                         CondInfTree = ctreeFit,
                         MARS = earthFit))
difs <- diff(resamps)
dotplot(difs)
densityplot(difs,  
            metric = "RMSE",  
            auto.key = TRUE,  
            pch = "|")
bwplot(difs,  
       metric = "RMSE")
levelplot(difs, what = "differences")

## End(Not run)
```
dummyVars

Usage

downSample(x, y, list = FALSE, yname = "Class")

upSample(x, y, list = FALSE, yname = "Class")

Arguments

- **x**: a matrix or data frame of predictor variables
- **y**: a factor variable with the class memberships
- **list**: should the function return list(x, y) or bind x and y together? If TRUE, the output will be coerced to a data frame.
- **yname**: if list = FALSE, a label for the class column

Details

Simple random sampling is used to down-sample for the majority class(es). Note that the minority class data are left intact and that the samples will be re-ordered in the down-sampled version.

For up-sampling, all of the original data are left intact and additional samples are added to the minority classes with replacement.

Value

Either a data frame or a list with elements x and y.

Author(s)

Max Kuhn

Examples

```r
## A ridiculous example...
data(oil)
table(oilType)
downSample(fattyAcids, oilType)

upSample(fattyAcids, oilType)
```

dummyVarsCreate A Full Set of Dummy Variables

Description

dummyVars creates a full set of dummy variables (i.e. less than full rank parameterization)
Usage

dummyVars(formula, ...)

## Default S3 method:
dummyVars(formula, data, sep = ".", levelsOnly = FALSE,
fullRank = FALSE, ...)

## S3 method for class 'dummyVars'
predict(object, newdata, na.action = na.pass, ...)

contrdummy(n, ...) ## DEPRECATED

contrltfr(n, contrasts = TRUE, sparse = FALSE)

Arguments

formula An appropriate R model formula, see References
data A data frame with the predictors of interest
sep An optional separator between factor variable names and their levels. Use sep = NULL
for no separator (i.e. normal behavior of model.matrix as shown in the Details
section)
levelsOnly A logical; TRUE means to completely remove the variable names from the column names
fullRank A logical; should a full rank or less than full rank parameterization be used? If TRUE, factors are encoded to be consistent with model.matrix and the resulting there are no linear dependencies induced between the columns.
object An object of class dummyVars
newdata A data frame with the required columns
na.action A function determining what should be done with missing values in newdata. The default is to predict NA.
n A vector of levels for a factor, or the number of levels.
contrasts A logical indicating whether contrasts should be computed.
sparse A logical indicating if the result should be sparse.
... additional arguments to be passed to other methods

Details

Most of the contrasts functions in R produce full rank parameterizations of the predictor data. For example, contr.treatment creates a reference cell in the data and defines dummy variables for all factor levels except those in the reference cell. For example, if a factor with 5 levels is used in a model formula alone, contr.treatment creates columns for the intercept and all the factor levels except the first level of the factor. For the data in the Example section below, this would produce:

(Intercept) dayTue dayWed dayThu dayFri daySat daySun
1 1 1 0 0 0 0 0 0
In some situations, there may be a need for dummy variables for all of the levels of the factor. For the same example:

```
   dayMon dayTue dayWed dayThu dayFri daySat daySun
1     0     1     0     0     0     0     0     0
2     0     1     0     0     0     0     0     0
3     0     1     0     0     0     0     0     0
4     0     0     0     1     0     0     0     0
5     0     0     0     1     0     0     0     0
6     0     0     0     0     1     0     0     0
7     0     0     1     0     0     0     0     0
8     0     0     0     1     0     0     0     0
9     1     0     0     0     0     0     0     0
```

Given a formula and initial data set, the class `dummyVars` gathers all the information needed to produce a full set of dummy variables for any data set. It uses `contr.1tfr` as the base function to do this.

### Value

The output of `dummyVars` is a list of class 'dummyVars' with elements:

- **call**: the function call
- **form**: the model formula
- **vars**: names of all the variables in the model
- **facVars**: names of all the factor variables in the model
- **lvs**: levels of any factor variables
- **sep**: `NULL` or a character separator
- **terms**: the `terms.formula` object
- **levelsOnly**: a logical

The `predict` function produces a data frame.

`contr.1tfr` generates a design matrix.

### Author(s)

`contr.1tfr` is a small modification of `contr.treatment` by Max Kuhn
References

http://cran.r-project.org/doc/manuals/R-intro.html#Formulae-for-statistical-models

See Also

model.matrix, contrasts, formula

Examples

```r
when <- data.frame(time = c("afternoon", "night", "afternoon", "morning", "morning", "morning", "morning", "afternoon", "afternoon"),
                   day = c("Mon", "Mon", "Mon", "Wed", "Wed", "Fri", "Sat", "Sat", "Fri"))

levels(when$time) <- c("morning", "afternoon", "night")
levels(when$day)  <- c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")

## Default behavior:
model.matrix(~ day, when)
mainEffects <- dummyVars(~ day + time, data = when)
mainEffects
predict(mainEffects, when[1:3,])

when2 <- when
when2[1, 1] <- NA
predict(mainEffects, when2[1:3,])
predict(mainEffects, when2[1:3,], na.action = na.omit)

interactionModel <- dummyVars(~ day + time + day:time,
                              data = when,
                              sep = ".")
predict(interactionModel, when[1:3,])

noNames <- dummyVars(~ day + time + day:time,
                      data = when,
                      levelsOnly = TRUE)
predict(noNames, when)
```

featurePlot  
Wrapper for Lattice Plotting of Predictor Variables

Description

A shortcut to produce lattice graphs
Usage

```r
defeaturePlot(x, y,
    plot = if(is.factor(y)) "strip" else "scatter",
    labels = c("Feature", ""),
    ...
)
```

Arguments

- **x**: a matrix or data frame of continuous feature/probe/spectra data.
- **y**: a factor indicating class membership.
- **plot**: the type of plot. For classification: box, strip, density, pairs or ellipse. For regression, pairs or scatter
- **labels**: a bad attempt at pre-defined axis labels
- **...**: options passed to lattice calls.

Details

This function "stacks" data to get it into a form compatible with lattice and creates the plots

Value

An object of class "trellis". The ‘update’ method can be used to update components of the object and the ‘print’ method (usually called by default) will plot it on an appropriate plotting device.

Author(s)

Max Kuhn

Examples

```r
x <- matrix(rnorm(50*5), ncol=5)
y <- factor(rep(c("A", "B"), 25))
trellis.par.set(theme = col.whitebg(), warn = FALSE)
featurePlot(x, y, "ellipse")
featurePlot(x, y, "strip", jitter = TRUE)
featurePlot(x, y, "box")
featurePlot(x, y, "pairs")
```
filterVarImp | Calculation of filter-based variable importance

**Description**

Specific engines for variable importance on a model by model basis.

**Usage**

```
filterVarImp(x, y, nonpara = FALSE, ...)
```

**Arguments**

- `x` A matrix or data frame of predictor data
- `y` A vector (numeric or factor) of outcomes
- `nonpara` should nonparametric methods be used to assess the relationship between the features and response
- `...` options to pass to either `lm` or `loess`

**Details**

The importance of each predictor is evaluated individually using a “filter” approach.

For classification, ROC curve analysis is conducted on each predictor. For two class problems, a series of cutoffs is applied to the predictor data to predict the class. The sensitivity and specificity are computed for each cutoff and the ROC curve is computed. The trapezoidal rule is used to compute the area under the ROC curve. This area is used as the measure of variable importance. For multi–class outcomes, the problem is decomposed into all pair-wise problems and the area under the curve is calculated for each class pair (i.e class 1 vs. class 2, class 2 vs. class 3 etc.). For a specific class, the maximum area under the curve across the relevant pair–wise AUC’s is used as the variable importance measure.

For regression, the relationship between each predictor and the outcome is evaluated. An argument, `nonpara`, is used to pick the model fitting technique. When `nonpara = FALSE`, a linear model is fit and the absolute value of the $t$–value for the slope of the predictor is used. Otherwise, a loess smoother is fit between the outcome and the predictor. The $R^2$ statistic is calculated for this model against the intercept only null model.

**Value**

A data frame with variable importances. Column names depend on the problem type. For regression, the data frame contains one column: "Overall" for the importance values.

**Author(s)**

Max Kuhn
**findCorrelation**

**Examples**

```r
data(mdr)
filterVarImp(mdrDescr[, 1:5], mdrClass)

data(BloodBrain)
filterVarImp(bbbDescr[, 1:5], logBBB, nonpara = FALSE)
apply(bbbDescr[, 1:5],
2,
  function(x, y) summary(lm(y~x))$coefficients[2,3],
  y = logBBB)
filterVarImp(bbbDescr[, 1:5], logBBB, nonpara = TRUE)
```

**Description**

This function searches through a correlation matrix and returns a vector of integers corresponding to columns to remove to reduce pair-wise correlations.

**Usage**

```r
findCorrelation(x, cutoff = .90, verbose = FALSE)
```

**Arguments**

- `x`: A correlation matrix
- `cutoff`: A numeric value for the pairwise absolute correlation cutoff
- `verbose`: A boolean for printing the details

**Details**

The absolute values of pair-wise correlations are considered. If two variables have a high correlation, the function looks at the mean absolute correlation of each variable and removes the variable with the largest mean absolute correlation.

There are several function in the `subselect` package (`leaps`, `genetic`, `anneal`) that can also be used to accomplish the same goal.

**Value**

A vector of indices denoting the columns to remove. If no correlations meet the criteria, `numeric(0)` is returned.

**Author(s)**

Original R code by Dong Li, modified by Max Kuhn
findLinearCombos

Determine linear combinations in a matrix

Description

Enumerate and resolve the linear combinations in a numeric matrix

Usage

findLinearCombos(x)

Arguments

x          a numeric matrix

Details

The QR decomposition is used to determine if the matrix is full rank and then identify the sets of columns that are involved in the dependencies. To "resolve" them, columns are iteratively removed and the matrix rank is rechecked. The trim.matrix function in the subselect package can also be used to accomplish the same goal.

Value

a list with elements:

linearCombos If there are linear combinations, this will be a list with elements for each dependency that contains vectors of column numbers.
remove        a list of column numbers that can be removed to counter the linear combinations
Author(s)

Kirk Mettler and Jed Wing (enumLC) and Max Kuhn (findLinearCombos)

See Also

trim.matrix

Examples

testData1 <- matrix(0, nrow=20, ncol=8)
testData1[,1] <- 1
testData1[,2] <- round(rnorm(20), 1)
testData1[,3] <- round(rnorm(20), 1)
testData1[,4] <- round(rnorm(20), 1)
testData1[,5] <- 0.5 * testData1[,2] - 0.25 * testData1[,3] - 0.25 * testData1[,4]
testData1[1:4,6] <- 1
testData1[5:10,7] <- 1
testData1[11:20,8] <- 1

findLinearCombos(testData1)

testData2 <- matrix(0, nrow=6, ncol=6)
testData2[,1] <- c(1, 1, 1, 1, 1, 1)
testData2[,2] <- c(1, 1, 1, 0, 0, 0)
testData2[,3] <- c(0, 0, 0, 1, 1, 1)
testData2[,4] <- c(1, 0, 0, 1, 0, 0)
testData2[,5] <- c(0, 1, 0, 0, 1, 0)
testData2[,6] <- c(0, 0, 1, 0, 0, 1)

findLinearCombos(testData2)

format.bagEarth

Format 'bagEarth' objects

Description

Return a string representing the 'bagEarth' expression.

Usage

## S3 method for class 'bagEarth'
format(x, file = "", cat = TRUE, ...)

Arguments

x An bagEarth object. This is the only required argument.

file A connection, or a character string naming the file to print to. If "" (the default),
the output prints to the standard output connection. See link[base](cat).

cat a logical; should the equation be printed?

Arguments to format.earth.
Value

A character representation of the bagged earth object.

See Also

earth

Examples

```r
a <- bagEarth(Volume ~ ., data = trees, B= 3)
cat(format(a))

# yields:
# 
# 31.61075
# + 6.587273 * pmax(0, Girth - 14.2)
# - 3.229363 * pmax(0, 14.2 - Girth)
# - 0.3167140 * pmax(0, 79 - Height)
# +  
# 22.80225
# + 5.309866 * pmax(0, Girth - 12)
# - 2.378058 * pmax(0, 12 - Girth)
# + 0.793045 * pmax(0, Height - 80)
# - 0.3411915 * pmax(0, 80 - Height)
# +  
# 31.39772
# + 6.18193 * pmax(0, Girth - 14.2)
# - 3.660456 * pmax(0, 14.2 - Girth)
# + 0.6489774 * pmax(0, Height - 80)
# )/3
```

---

GermanCredit

| German Credit Data |

Description

Data from Dr. Hans Hofmann of the University of Hamburg.

These data have two classes for the credit worthiness: good or bad. There are predictors related to attributes, such as: checking account status, duration, credit history, purpose of the loan, amount of the loan, savings accounts or bonds, employment duration, Installment rate in percentage of disposable income, personal information, other debtors/guarantors, residence duration, property, age, other installment plans, housing, number of existing credits, job information, Number of people being liable to provide maintenance for, telephone, and foreign worker status.

Many of these predictors are discrete and have been expanded into several 0/1 indicator variables

Usage

data(GermanCredit)
Source

UCI Machine Learning Repository

---

**histogram.train** | **Lattice functions for plotting resampling results**

---

**Description**

A set of lattice functions are provided to plot the resampled performance estimates (e.g. classification accuracy, RMSE) over tuning parameters (if any).

**Usage**

```r
## S3 method for class 'train'
histogram(x, data = NULL, metric = x$metric, ...)
```

```r
## S3 method for class 'train'
densityplot(x, data = NULL, metric = x$metric, ...)
```

```r
## S3 method for class 'train'
xyplot(x, data = NULL, metric = x$metric, ...)
```

```r
## S3 method for class 'train'
stripplot(x, data = NULL, metric = x$metric, ...)
```

**Arguments**

- `x`  
  An object produced by `train`

- `data`  
  This argument is not used

- `metric`  
  A character string specifying the single performance metric that will be plotted

- `...`  
  Arguments to pass to either `histogram`, `densityplot`, `xyplot` or `stripplot`

**Details**

By default, only the resampling results for the optimal model are saved in the `train` object. The function `trainControl` can be used to save all the results (see the example below).

If leave-one-out or out-of-bag resampling was specified, plots cannot be produced (see the method argument of `trainControl`)

For `xyplot` and `stripplot`, the tuning parameter with the most unique values will be plotted on the x-axis. The remaining parameters (if any) will be used as conditioning variables. For `densityplot` and `histogram`, all tuning parameters are used for conditioning.

Using `horizontal = FALSE` in `stripplot` works.

**Value**

A lattice plot object
Author(s)
Max Kuhn

See Also
train, trainControl, histogram, densityplot, xyplot, stripplot

Examples

## Not run:

```r
library(mlbench)
data(BostonHousing)

library(rpart)
rpartFit <- train(medv ~ ., data = BostonHousing, "rpart",
  tuneLength = 9, trControl = trainControl(method = "boot",
  returnResamp = "all"))

densityplot(rpartFit, adjust = 1.25)

xyplot(rpartFit, metric = "Rsquared",
  type = c("p", "a"))

stripplot(rpartFit, horizontal = FALSE, jitter = TRUE)

## End(Not run)
```

---

**icr.formula**

Independent Component Regression

Description

Fit a linear regression model using independent components

Usage

```r
## S3 method for class 'formula'
icr(formula, data, weights, ..., subset, na.action, contrasts = NULL)
## Default S3 method:
```
icr(x, y, ...)

## S3 method for class 'icr'
predict(object, newdata, ...)

### Arguments

- **formula**: A formula of the form \( \text{class} \sim x_1 + x_2 + \ldots \)
- **data**: Data frame from which variables specified in formula are preferentially to be taken.
- **weights**: (case) weights for each example – if missing defaults to 1.
- **subset**: An index vector specifying the cases to be used in the training sample. (NOTE: If given, this argument must be named.)
- **na.action**: A function to specify the action to be taken if NAs are found. The default action is for the procedure to fail. An alternative is na.omit, which leads to rejection of cases with missing values on any required variable. (NOTE: If given, this argument must be named.)
- **contrasts**: a list of contrasts to be used for some or all of the factors appearing as variables in the model formula.
- **...**: arguments passed to fastICA
- **x**: matrix or data frame of \( x \) values for examples.
- **y**: matrix or data frame of target values for examples.
- **object**: an object of class icr as returned by icr.
- **newdata**: matrix or data frame of test examples.

### Details

This produces a model analogous to Principal Components Regression (PCR) but uses Independent Component Analysis (ICA) to produce the scores. The user must specify a value of \( n\).comp to pass to fastICA.

The function preProcess to produce the ICA scores for the original data and for newdata.

### Value

For icr, a list with elements

- **model**: the results of \( \text{lm} \) after the ICA transformation
- **ica**: pre-processing information
- **n.comp**: number of ICA components
- **names**: column names of the original data

### Author(s)

Max Kuhn
See Also

fastICA, preProcess, lm

Examples

data(BloodBrain)

icrFit <- icr(bbbDescr, logBBB, n.comp = 5)

icrFit

predict(icrFit, bbbDescr[1:5,])

knn

k-Nearest Neighbour Classification

Description

$k$-nearest neighbour classification that can return class votes for all classes.

Usage

## S3 method for class 'formula'
knn3(formula, data, subset, na.action, k = 5, ...)

## S3 method for class 'matrix'
knn3(x, y, k = 5, ...)

## S3 method for class 'data.frame'
knn3(x, y, k = 5, ...)

knn3Train(train, test, cl, k=1, l=0, prob = TRUE, use.all=TRUE)

Arguments

formula a formula of the form lhs ~ rhs where lhs is the response variable and rhs a set of predictors.
data optional data frame containing the variables in the model formula.
subset optional vector specifying a subset of observations to be used.
na.action function which indicates what should happen when the data contain NAs.
k number of neighbours considered.
x a matrix of training set predictors
y  a factor vector of training set classes

... additional parameters to pass to knnTrain. However, passing prob = FALSE will be over-ridden.

train matrix or data frame of training set cases.

test matrix or data frame of test set cases. A vector will be interpreted as a row vector for a single case.

c1 factor of true classifications of training set

l minimum vote for definite decision, otherwise doubt. (More precisely, less than k-1 dissenting votes are allowed, even if k is increased by ties.)

prob If this is true, the proportion of the votes for each class are returned as attribute prob.

use.all controls handling of ties. If true, all distances equal to the kth largest are included. If false, a random selection of distances equal to the kth is chosen to use exactly k neighbours.

Details

knn3 is essentially the same code as ipredknn and knnTrain is a copy of knn. The underlying C code from the class package has been modified to return the vote percentages for each class (previously the percentage for the winning class was returned).

Value

An object of class knn3. See predict.knn3.

Author(s)

knn by W. N. Venables and B. D. Ripley and ipredknn by Torsten.Hothorn<Torsten.Hothorn@rzmail.uni-erlangen.de>, modifications by Max Kuhn and Andre Williams

Examples

irisFit1 <- knn3(Species ~ ., iris)

irisFit2 <- knn3(as.matrix(iris[, -5]), iris[,5])

data(iris3)

train <- rbind(iris3[1:25,,1], iris3[1:25,,2], iris3[1:25,,3])

test <- rbind(iris3[26:50,,1], iris3[26:50,,2], iris3[26:50,,3])

c1 <- factor(c(rep("s",25), rep("c",25), rep("v",25)))

knn3Train(train, test, c1, k = 5, prob = TRUE)
knnreg

\textit{k-Nearest Neighbour Regression}

\textbf{Description}

$k$-nearest neighbour classification that can return the average value for the neighbours.

\textbf{Usage}

```r
## Default S3 method:
knnreg(x, ...)

## S3 method for class 'formula'
knnreg(formula, data, subset, na.action, k = 5, ...)

## S3 method for class 'matrix'
knnreg(x, y, k = 5, ...)

## S3 method for class 'data.frame'
knnreg(x, y, k = 5, ...)  
knnregTrain(train, test, y, k = 5, use.all=TRUE)
```

\textbf{Arguments}

- \texttt{formula} a formula of the form \texttt{lhs ~ rhs} where \texttt{lhs} is the response variable and \texttt{rhs} a set of predictors.
- \texttt{data} optional data frame containing the variables in the model formula.
- \texttt{subset} optional vector specifying a subset of observations to be used.
- \texttt{na.action} function which indicates what should happen when the data contain NAs.
- \texttt{k} number of neighbours considered.
- \texttt{x} a matrix or data frame of training set predictors.
- \texttt{y} a numeric vector of outcomes.
- \texttt{...} additional parameters to pass to \texttt{knnregTrain}.
- \texttt{train} matrix or data frame of training set cases.
- \texttt{test} matrix or data frame of test set cases. A vector will be interpreted as a row vector for a single case.
- \texttt{use.all} controls handling of ties. If true, all distances equal to the kth largest are included. If false, a random selection of distances equal to the kth is chosen to use exactly k neighbours.
Details

knnreg is similar to ipredknn and knnregTrain is a modification of knn. The underlying C code from the class package has been modified to return average outcome.

Value

An object of class knnreg. See predict.knnreg.

Author(s)

knn by W. N. Venables and B. D. Ripley and ipredknn by Torsten.Hothorn <Torsten.Hothorn@rzmail.unierlangen.de>, modifications by Max Kuhn and Chris Keefer

Examples

data(BloodBrain)

inTrain <- createDataPartition(logBBB, p = .8)[[1]]

trainX <- bbbDescr[inTrain,]
trainY <- logBBB[inTrain]

testX <- bbbDescr[-inTrain,]
testY <- logBBB[-inTrain]

fit <- knnreg(trainX, trainY, k = 3)

plot(testY, predict(fit, testX))

lattice.rfe

Lattice functions for plotting resampling results of recursive feature selection

Description

A set of lattice functions are provided to plot the resampled performance estimates (e.g. classification accuracy, RMSE) over different subset sizes.

Usage

## S3 method for class 'rfe'
histogram(x, data = NULL, metric = x$metric, ...)

## S3 method for class 'rfe'
densityplot(x, data = NULL, metric = x$metric, ...)

## S3 method for class 'rfe'
xyplot(x, data = NULL, metric = x$metric, ...)
## S3 method for class 'rfe'

`stripplot(x, data = NULL, metric = x$metric, ...)`

### Arguments

- `x`  
  An object produced by `rfe`
- `data`  
  This argument is not used
- `metric`  
  A character string specifying the single performance metric that will be plotted
- `...`  
  Arguments to pass to either `histogram`, `densityplot`, `xyplot` or `stripplot`

### Details

By default, only the resampling results for the optimal model are saved in the `rfe` object. The function `rfeControl` can be used to save all the results using the `returnResamp` argument.

If leave-one-out or out-of-bag resampling was specified, plots cannot be produced (see the method argument of `rfeControl`)

### Value

A lattice plot object

### Author(s)

Max Kuhn

### See Also

- `rfe`, `rfeControl`, `histogram`, `densityplot`, `xyplot`, `stripplot`

### Examples

```r
## Not run:
library(mlbench)
n <- 100
p <- 40
sigma <- 1
set.seed(1)
sim <- mlbench.friedman1(n, sd = sigma)
x <- cbind(sim$x, matrix(rnorm(n * p), nrow = n))
y <- sim$y
colnames(x) <- paste("var", 1:ncol(x), sep = "")

normalization <- preProcess(x)
x <- predict(normalization, x)
x <- as.data.frame(x)
subsets <- c(10, 15, 20, 25)
ctrl1 <- rfeControl(
  functions = lmFuncs,
)
```
For classification models, this function creates a 'lift plot' that describes how well a model ranks samples for one class.

Usage

lift(x, ...)

## S3 method for class 'formula'
lift(x, data = NULL, class = NULL, subset = TRUE, lattice.options = NULL, labels = NULL, ...)

## S3 method for class 'lift'
xyplot(x, data, plot = "gain", values = NULL, ...)

Arguments

x

A lattice formula (see xyplot for syntax) where the left-hand side of the formula is a factor class variable of the observed outcome and the right-hand side specifies one or model columns corresponding to a numeric ranking variable for a model (e.g., class probabilities). The classification variable should have two levels.

data

For lift.formula, a data frame (or more precisely, anything that is a valid envir argument in eval, e.g., a list or an environment) containing values for any variables in the formula, as well as groups and subset if applicable. If not found in data, or if data is unspecified, the variables are looked for in the environment of the formula. This argument is not used for xyplot.lift.
class a character string for the class of interest
subset An expression that evaluates to a logical or integer indexing vector. It is evaluated in data. Only the resulting rows of data are used for the plot.
lattice.options A list that could be supplied to lattice.options
labels A named list of labels for keys. The list should have an element for each term on the right-hand side of the formula and the names should match the names of the models.
plot Either "gain" (the default) or "lift". The former plots the number of samples called events versus the event rate while the latter shows the event cut-off versus the lift statistic.
values A vector of numbers between 0 and 100 specifying reference values for the percentage of samples found (i.e. the y-axis). Corresponding points on the x-axis are found via interpolation and line segments are shown to indicate how many samples must be tested before these percentages are found. The lines use either the plot.line or superpose.line component of the current lattice theme to draw the lines (depending on whether groups were used. These values are only used when type = "gain".

Options to pass through to xyplot or the panel function (not used in lift.formula).

Details

lift.formula is used to process the data and xyplot.lift is used to create the plot.

To construct data for the lift and gain plots, the following steps are used for each model:

1. The data are ordered by the numeric model prediction used on the right-hand side of the model formula
2. Each unique value of the score is treated as a cut point
3. The number of samples with true results equal to class are determined
4. The lift is calculated as the ratio of the percentage of samples in each split corresponding to class over the same percentage in the entire data set

lift with plot = "gain" produces a plot of the cumulative lift values by the percentage of samples evaluated while plot = "lift" shows the cut point value versus the lift statistic.

This implementation uses the lattice function xyplot, so plot elements can be changed via panel functions, trellis.par.set or other means. lift uses the panel function panel.lift2 by default, but it can be changes using update.trellis (see the examples in panel.lift2).

The following elements are set by default in the plot but can be changed by passing new values into xyplot.lift: xlab = "% Samples Tested", ylab = "% Samples Found", type = "S", ylim = extendrange(c(0, 100)) and xlim = extendrange(c(0, 100)).

Value

lift.formula returns a list with elements:

data the data used for plotting
maxDissim

- **cuts**: the number of cuts
- **class**: the event class
- **probNames**: the names of the model probabilities
- **pct**: the baseline event rate

`xyplot.lift` returns a **lattice** object

**Author(s)**
Max Kuhn, some **lattice** code and documentation by Deepayan Sarkar

**See Also**
- `xyplot`, `trellis.par.set`

**Examples**

```r
set.seed(1)
simulated <- data.frame(obs = factor(rep(letters[1:2], each = 100)),
                          perfect = sort(runif(200), decreasing = TRUE),
                          random = runif(200))

lift1 <- lift(obs ~ random, data = simulated)
lift1
xyplot(lift1)

lift2 <- lift(obs ~ random + perfect, data = simulated)
lift2
xyplot(lift2, auto.key = list(columns = 2))

xyplot(lift2, auto.key = list(columns = 2), value = c(10, 30))

xyplot(lift2, plot = "lift", auto.key = list(columns = 2))
```

**Description**
Functions to create a sub-sample by maximizing the dissimilarity between new samples and the existing subset.

**Usage**

```r
maxDissim(a, b, n = 2, obj = minDiss, useNames = FALSE,
          randomFrac = 1, verbose = FALSE, ...)
minDiss(u)
sumDiss(u)
```

**Maximum Dissimilarity Sampling**

Functions to create a sub-sample by maximizing the dissimilarity between new samples and the existing subset.
Arguments

- **a**: a matrix or data frame of samples to start
- **b**: a matrix or data frame of samples to sample from
- **n**: the size of the sub-sample
- **obj**: an objective function to measure overall dissimilarity
- **useNames**: a logical: should the function return the row names (as opposed to the row index)
- **randomFrac**: a number in (0, 1] that can be used to sub-sample from the remaining candidate values
- **verbose**: a logical; should each step be printed?
- **...**: optional arguments to pass to `dist`
- **u**: a vector of dissimilarities

Details

Given an initial set of `m` samples and a larger pool of `n` samples, this function iteratively adds points to the smaller set by finding which of the `n` samples is most dissimilar to the initial set. The argument **obj** measures the overall dissimilarity between the initial set and a candidate point. For example, maximizing the minimum or the sum of the `m` dissimilarities are two common approaches.

This algorithm tends to select points on the edge of the data mainstream and will reliably select outliers. To select more samples towards the interior of the data set, set **randomFrac** to be small (see the examples below).

Value

A vector of integers or row names (depending on **useNames**) corresponding to the rows of **b** that comprise the sub-sample.

Author(s)

Max Kuhn <max.kuhn@pfizer.com>

References


See Also

dist

Examples

```r
example <- function(pct = 1, obj = minDiss, ...) {
  tmp <- matrix(rnorm(200 * 2), nrow = 200)
```
## Multidrug Resistance Reversal (MDRR) Agent Data

```r
## start with 15 data points
start <- sample(1:dim(tmp)[1], 15)
base <- tmp[start,]
pool <- tmp[-start,]

## select 9 for addition
newsamp <- maxDissim(
  base, pool, n = 9,
  randomFrac = pct, obj = obj, ...)

allsamp <- c(start, newsamp)

plot(
  tmp[-newSamp,],
  xlim = extendrange(tmp[,1]), ylim = extendrange(tmp[,2]),
  col = "darkgrey",
  xlab = "variable 1", ylab = "variable 2")
points(base, pch = 16, cex = .7)

for(i in seq(along = newsamp))
  points(
    pool[newSamp[i],1],
    pool[newSamp[i],2],
    pch = paste(i), col = "darkred")

par(mfrow=c(2,2))
set.seed(414)
example(1, minDiss)
title("No Random Sampling, Min Score")

set.seed(414)
example(.1, minDiss)
title("10 Pct Random Sampling, Min Score")

set.seed(414)
example(1, sumDiss)
title("No Random Sampling, Sum Score")

set.seed(414)
example(.1, sumDiss)
title("10 Pct Random Sampling, Sum Score")
```
Description

Svetnik et al (2003) describe these data: "Bakken and Jurs studied a set of compounds originally
discussed by Klopman et al., who were interested in multidrug resistance reversal (MDRR) agents.
The original response variable is a ratio measuring the ability of a compound to reverse a leukemia
cell's resistance to adriamycin. However, the problem was treated as a classification problem, and
compounds with the ratio >4.2 were considered active, and those with the ratio <= 2.0 were con-
sidered inactive. Compounds with the ratio between these two cutoffs were called moderate and
removed from the data for twoclass classification, leaving a set of 528 compounds (298 actives
and 230 inactives). (Various other arrangements of these data were examined by Bakken and Jurs,
but we will focus on this particular one.) We did not have access to the original descriptors, but
we generated a set of 342 descriptors of three different types that should be similar to the original
descriptors, using the DRAGON software."

The data and R code are in the Supplimental Data file for the article.

Usage

data(mdrr)

Value

mdrrDescr    the descriptors
mdrrClass    the categorical outcome ("Active" or "Inactive")

Source

Forest: A Classification and Regression Tool for Compound Classification and QSAR Modeling,

modelLookup

Description

These function show information about models and packages that are accessible via train

Usage

modelLookup(model = NULL)

getModelInfo(model = NULL, regex = TRUE, ...)

checkInstall(pkg)
modelLookup

Arguments

model  a character string associated with the method argument of train. If no value is passed, all models are returned. For getModelInfo, regular expressions can be used.
regex  a logical: should a regular expressions be used? If FALSE, a simple match is conducted against the whole name of the model.
pkg    a character string of package names.
...    options to pass to grepl

Details

modelLookup is good for getting information related to the tuning parameters for a model. getModelInfo will return all the functions and metadata associated with a model. Both of these functions will only search within the models bundled in this package.

checkInstall will check to see if packages are installed. If they are not and the session is interactive, an option is given to install the packages using install.packages using that functions default arguments (the missing packages are listed if you would like to install them with other options). If the session is not interactive, an error is thrown.

Value

modelLookup produces a data frame with columns

model a character string for the model code
parameter the tuning parameter name
label a tuning parameter label (used in plots)
forReg a logical; can the model be used for regression?
forClass a logical; can the model be used for classification?
probModel a logical; does the model produce class probabilities?

getModelInfo returns a list containing one or more lists of the standard model information.
checkInstall returns not value.

Note

The column seq is no longer included in the output of modelLookup.

Author(s)

Max Kuhn

See Also

train, install.packages, grepl
Examples

```r
modelLookup()
modelLookup("gbm")

getModelInfo("pls")
getModelInfo("pls")
getModelInfo("pls", regex = FALSE)

## Not run:
checkInstall(getModelInfo("pls")$library)

## End(Not run)
```

### Description

`nearZeroVar` diagnoses predictors that have one unique value (i.e. are zero variance predictors) or predictors that have both of the following characteristics: they have very few unique values relative to the number of samples and the ratio of the frequency of the most common value to the frequency of the second most common value is large. `checkConditionalX` looks at the distribution of the columns of `x` conditioned on the levels of `y` and identifies columns of `x` that are sparse within groups of `y`.

#### Usage

```r
nearZeroVar(x, freqCut = 95/5, uniqueCut = 10, saveMetrics = FALSE)
checkConditionalX(x, y)
checkResamples(index, x, y)
```

#### Arguments

- **x**: a numeric vector or matrix, or a data frame with all numeric data
- **freqCut**: the cutoff for the ratio of the most common value to the second most common value
- **uniqueCut**: the cutoff for the percentage of distinct values out of the number of total samples
- **saveMetrics**: a logical. If false, the positions of the zero- or near-zero predictors is returned. If true, a data frame with predictor information is returned.
- **y**: a factor vector with at least two levels
- **index**: a list. Each element corresponds to the training set samples in `x` for a given resample
Details

For example, an example of near zero variance predictor is one that, for 1000 samples, has two distinct values and 999 of them are a single value.

To be flagged, first the frequency of the most prevalent value over the second most frequent value (called the “frequency ratio”) must be above freqCut. Secondly, the “percent of unique values,” the number of unique values divided by the total number of samples (times 100), must also be below uniqueCut.

In the above example, the frequency ratio is 999 and the unique value percentage is 0.0001.

Checking the conditional distribution of x may be needed for some models, such as naive Bayes where the conditional distributions should have at least one data point within a class.

Value

For nearZeroVar: if saveMetrics = FALSE, a vector of integers corresponding to the column positions of the problematic predictors. If saveMetrics = TRUE, a data frame with columns:

- freqRatio: the ratio of frequencies for the most common value over the second most common value
- percentUnique: the percentage of unique data points out of the total number of data points
- zeroVar: a vector of logicals for whether the predictor has only one distinct value
- nzv: a vector of logicals for whether the predictor is a near zero variance predictor

For checkResamples or checkConditionalX, a vector of column indicators for predictors with empty conditional distributions in at least one class of y.

Author(s)

Max Kuhn, with speed improvements to nearZeroVar by Allan Engelhardt

Examples

nearZeroVar(iris[, -5], saveMetrics = TRUE)

data(BloodBrain)
nearZeroVar(bbbDescr)

set.seed(1)
classes <- factor(rep(letters[1:3], each = 30))
x <- data.frame(x1 = rep(c(0, 1), 45),
                x2 = c(rep(0, 10), rep(1, 80)))
lapply(x, table, y = classes)
checkConditionalX(x, classes)

folds <- createFolds(classes, k = 3, returnTrain = TRUE)
x$x3 <- x$x1
x$x3[folds[[1]]] <- 0
Quantile Normalization to a Reference Distribution

Description
Quantile normalization based upon a reference distribution. This function normalizes a matrix of data (typically Affy probe level intensities).

Usage
normalize.AffyBatch.normalize2Reference(
  abatch,
  type = c("separate", "pmonly", "mmonly", "together"),
  ref = NULL)

Arguments
  abatch   An {AffyBatch}
  type     A string specifying how the normalization should be applied. See details for more.
  ref      A vector of reference values. See details for more.

Details
This method is based upon the concept of a quantile-quantile plot extended to n dimensions. No special allowances are made for outliers. If you make use of quantile normalization either through rma or expresso please cite Bolstad et al, Bioinformatics (2003).

The type argument should be one of "separate", "pmonly", "mmonly", "together" which indicates whether to normalize only one probe type (PM,MM) or both together or separately.

The function uses the data supplied in ref to use as the reference distribution. In other words, the PMs in abatch will be normalized to have the same distribution as the data in ref. If ref is NULL, the normalizing takes place using the average quantiles of the PM values in abatch (just as in normalize.AffyBatch.quantile).

Value
A normalized AffyBatch.

Author(s)
Max Kuhn, adapted from Ben Bolstad, <bolstad@stat.berkeley.edu>
normalize2Reference

References

Bolstad, B (2001) Probe Level Quantile Normalization of High Density Oligonucleotide Array Data. Unpublished manuscript


See Also

normalize

Examples

# first, let affy/expresso know that the method exists
# normalize.AffyBatch.methods <- c(normalize.AffyBatch.methods, "normalize2Reference")

# example not run, as it would take a while
# RawData <- ReadAffy(celfile.path=FilePath)

# Batch1Step1 <- bg.correct(RawData, "rma")
# Batch1Step2 <- normalize.AffyBatch.quantiles(Batch1Step1)
# referencePM <- pm(Batch1Step2)[,1]
# Batch1Step3 <- computeExprSet(Batch1Step2, "pmonly", "medianpolish")

# Batch2Step1 <- bg.correct(RawData2, "rma")
# Batch2Step2 <- normalize.AffyBatch.normalize2Reference(Batch2Step1, ref = referencePM)
# Batch2Step3 <- computeExprSet(Batch2Step2, "pmonly", "medianpolish")

normalize2Reference(data, refData = NULL, ties = TRUE)

Arguments

data numeric matrix. Missing values are allowed.
refData A vector of reference values.
ties logical. If TRUE, ties in each column of A are treated in careful way. Tied values will be normalized to the mean of the corresponding pooled quantiles.
Details

This function is intended to normalize single channel or A-value microarray intensities between arrays. Each quantile of each column is set to either: quantiles of the reference distribution (refData supplied) or the mean of that quantile across arrays (refData is NULL). The intention is to make all the normalized columns have the same empirical distribution. This will be exactly true if there are no missing values and no ties within the columns: the normalized columns are then simply permutations of one another.

If there are ties amongst the intensities for a particular array, then with ties=FALSE the ties are broken in an unpredictable order. If ties=TRUE, all the tied values for that array will be normalized to the same value, the average of the quantiles for the tied values.

Value

A matrix of the same dimensions as A containing the normalized values.

Author(s)

Max Kuhn, adapted from Gordon Smyth

References


See Also

normalize.AffyBatch.normalize2Reference

nullModel

Fit a simple, non-informative model

Description

Fit a single mean or largest class model

Usage

nullModel(x, ...)

## Default S3 method:
nullModel(x = NULL, y, ...)

## S3 method for class 'nullModel'
predict(object, newdata = NULL, type = NULL, ...)
nullModel

Arguments

x
An optional matrix or data frame of predictors. These values are not used in the model fit

y
A numeric vector (for regression) or factor (for classification) of outcomes

... Optional arguments (not yet used)

object An object of class nullModel

cdata A matrix or data frame of predictors (only used to determine the number of predictions to return)

type Either "raw" (for regression), "class" or "prob" (for classification)

Details

nullModel emulates other model building functions, but returns the simplest model possible given a training set: a single mean for numeric outcomes and the most prevalent class for factor outcomes. When class probabilities are requested, the percentage of the training set samples with the most prevalent class is returned.

Value

The output of nullModel is a list of class nullModel with elements

call the function call

value the mean of y or the most prevalent class

levels when y is a factor, a vector of levels. NULL otherwise

pct when y is a factor, a data frame with a column for each class (NULL otherwise). The column for the most prevalent class has the proportion of the training samples with that class (the other columns are zero).

n the number of elements in y

predict.nullModel returns a either a factor or numeric vector depending on the class of y. All predictions are always the same.

Examples

outcome <- factor(sample(letters[1:2],
          size = 100,
          prob = c(.1,.9),
          replace = TRUE))

useless <- nullModel(y = outcome)

useless

predict(useless, matrix(NA, nrow = 10))
**Fatty acid composition of commercial oils**

**Description**

Fatty acid concentrations of commercial oils were measured using gas chromatography. The data is used to predict the type of oil. Note that only the known oils are in the data set. Also, the authors state that there are 95 samples of known oils. However, we count 96 in Table 1 (pgs. 33-35).

**Usage**

```r
data(oil)
```

**Value**

- `fattyAcids` data frame of fatty acid compositions: Palmitic, Stearic, Oleic, Linoleic, Linolenic, Eicosanoic and Eicosenoic. When values fell below the lower limit of the assay (denoted as <X in the paper), the limit was used.
- `oilType` factor of oil types: pumpkin (A), sunflower (B), peanut (C), olive (D), soybean (E), rapeseed (F) and corn (G).

**Source**


---

**Selecting tuning Parameters**

**Description**

Various functions for setting tuning parameters

**Usage**

```r
best(x, metric, maximize)
onese(x, metric, num, maximize)
tolerance(x, metric, tol = 1.5, maximize)
```
Arguments

- **x**: a data frame of tuning parameters and model results, sorted from least complex models to the most complex
- **metric**: a string that specifies what summary metric will be used to select the optimal model. By default, possible values are "RMSE" and "Rsquared" for regression and "Accuracy" and "Kappa" for classification. If custom performance metrics are used (via the summaryFunction argument in `trainControl`), the value of **metric** should match one of the arguments. If it does not, a warning is issued and the first metric given by the summaryFunction is used.
- **maximize**: a logical: should the metric be maximized or minimized?
- **num**: the number of resamples (for oneSE only)
- **tol**: the acceptable percent tolerance (for tolerance only)

Details

These functions can be used by `train` to select the "optimal" model form a series of models. Each requires the user to select a metric that will be used to judge performance. For regression models, values of "RMSE" and "Rsquared" are applicable. Classification models use either "Accuracy" or "Kappa" (for unbalanced class distributions).

More details on these functions can be found at [http://caret.r-forge.r-project.org/training.html#custom](http://caret.r-forge.r-project.org/training.html#custom).

By default, `train` uses `best`.

`best` simply chooses the tuning parameter associated with the largest (or lowest for "RMSE") performance.

`oneSE` is a rule in the spirit of the "one standard error" rule of Breiman et al (1984), who suggest that the tuning parameter associated with the best performance may overfit. They suggest that the simplest model within one standard error of the empirically optimal model is the better choice. This assumes that the models can be easily ordered from simplest to most complex (see the Details section below).

`tolerance` takes the simplest model that is within a percent tolerance of the empirically optimal model. For example, if the largest Kappa value is 0.5 and a simpler model within 3 percent is acceptable, we score the other models using \((x - 0.5)/0.5 \times 100\). The simplest model whose score is not less than 3 is chosen (in this case, a model with a Kappa value of 0.35 is acceptable).

User-defined functions can also be used. The argument `selectionFunction` in `trainControl` can be used to pass the function directly or to pass the function by name.

Value

an row index

Note

In many cases, it is not very clear how to order the models on simplicity. For simple trees and other models (such as PLS), this is straightforward. However, for others it is not.
For example, many of the boosting models used by *caret* have parameters for the number of boosting iterations and the tree complexity (others may also have a learning rate parameter). In this implementation, we order models on number of iterations, then tree depth. Clearly, this is arguable (please email the author for suggestions though).

For MARS models, they are orders on the degree of the features, then the number of retained terms.

RBF SVM models are ordered first by the cost parameter, then by the kernel parameter while polynomial models are ordered first on polynomial degree, then cost and scale.

Neural networks are ordered by the number of hidden units and then the amount of weight decay.

$\text{SkS}$–nearest neighbor models are ordered from most neighbors to least (i.e. smoothest to model jagged decision boundaries).

Elastic net models are ordered first on the L1 penalty, then by the L2 penalty.

**Author(s)**

Max Kuhn

**References**


**See Also**

`train`, `trainControl`

**Examples**

```r
## Not run:
# simulate a PLS regression model
test <- data.frame(ncomp = 1:5,
                   RMSE = c(3, 1.1, 1.02, 1, 2),
                   RMSESD = .4)

best(test, "RMSE", maximize = FALSE)
oneSE(test, "RMSE", maximize = FALSE, num = 10)
tolerance(test, "RMSE", tol = 3, maximize = FALSE)

### usage example
data(BloodBrain)
marsGrid <- data.frame(degree = 1, nprune = (1:10) * 3)

set.seed(1)
marsFit <- train(bbbDescr, logBBB,
               method = "earth",
               tuneGrid = marsGrid,
               trControl = trainControl(method = "cv",
                                       number = 10,
                                       selectionFunction = "tolerance"))
```
# around 18 terms should yield the smallest CV RMSE

## End(Not run)

---

**panel.lift2**

### Lattice Panel Functions for Lift Plots

**Description**

Two panel functions that be used in conjunction with `lift`.

**Usage**

```r
panel.lift(x, y, ...)
panel.lift2(x, y, pct = 0, values = NULL, ...)
```

**Arguments**

- `x` the percentage of searched to be plotted in the scatterplot
- `y` the percentage of events found to be plotted in the scatterplot
- `pct` the baseline percentage of true events in the data
- `values` A vector of numbers between 0 and 100 specifying reference values for the percentage of samples found (i.e. the y-axis). Corresponding points on the x-axis are found via interpolation and line segments are shown to indicate how many samples must be tested before these percentages are found. The lines use either the `plot.line` or `superpose.line` component of the current lattice theme to draw the lines (depending on whether groups were used)
- `...` options to pass to `panel.xyplot`

**Details**

`panel.lift` plots the data with a simple (black) 45 degree reference line.

`panel.lift2` is the default for `lift` and plots the data points with a shaded region encompassing the space between to the random model and perfect model trajectories. The color of the region is determined by the lattice `reference.line` information (see example below).

**Author(s)**

Max Kuhn

**See Also**

`lift, panel.xyplot, xplot, trellis.par.set`
Examples

```r
set.seed(1)
simulated <- data.frame(obs = factor(rep(letters[1:2], each = 100)),
                         perfect = sort(runif(200), decreasing = TRUE),
                         random = runif(200))

regionInfo <- trellis.par.get("reference.line")
regionInfo$col <- "lightblue"
trellis.par.set("reference.line", regionInfo)

lift2 <- lift(obs ~ random + perfect, data = simulated)

xyplot(lift2, auto.key = list(columns = 2))
```

## use a different panel function

```r
xyplot(lift2, panel = panel.lift)
```

---

### panel.needle

**Needle Plot Lattice Panel**

**Description**

A variation of `panel.dotplot` that plots horizontal lines from zero to the data point.

**Usage**

```r
panel.needle(x, y, horizontal = TRUE,
pch, col, lty, lwd,
col.line, levels.fos,
groups = NULL,
...)
```

**Arguments**

- `x, y` variables to be plotted in the panel. Typically `y` is the 'factor'
- `horizontal` logical. If `FALSE`, the plot is 'transposed' in the sense that the behaviours of `x` and `y` are switched. `x` is now the 'factor'. Interpretation of other arguments change accordingly. See documentation of `bwplot` for a fuller explanation.
- `pch, col, lty, lwd, col.line` graphical parameters
- `levels.fos` locations where reference lines will be drawn
- `groups` grouping variable (affects graphical parameters)
- `...` extra parameters, passed to `panel.xyplot` which is responsible for drawing the foreground points (`panel.dotplot` only draws the background reference lines).
Details

Creates (possibly grouped) needleplot of x against y or vice versa.

Author(s)

Max Kuhn, based on panel.dotplot by Deepayan Sarkar

See Also

dotplot

Description

Run PCA on a dataset, then use it in a neural network model.

Usage

```r
## Default S3 method:
pcaNNet(x, y, thresh = 0.99, ...)
## S3 method for class 'formula'
pcaNNet(formula, data, weights, ..., thresh = .99, subset, na.action, contrasts = NULL)

## S3 method for class 'pcaNNet'
predict(object, newdata, type = c("raw", "class"), ...)
```

Arguments

- `formula`: A formula of the form `class ~ x1 + x2 + ...`
- `x`: matrix or data frame of x values for examples.
- `y`: matrix or data frame of target values for examples.
- `weights`: (case) weights for each example – if missing defaults to 1.
- `thresh`: a threshold for the cumulative proportion of variance to capture from the PCA analysis. For example, to retain enough PCA components to capture 95 percent of variation, set `thresh = .95`
- `data`: Data frame from which variables specified in `formula` are preferentially to be taken.
- `subset`: An index vector specifying the cases to be used in the training sample. (NOTE: If given, this argument must be named.)
- `na.action`: A function to specify the action to be taken if NAs are found. The default action is for the procedure to fail. An alternative is `na.omit`, which leads to rejection of cases with missing values on any required variable. (NOTE: If given, this argument must be named.)
contrasts          a list of contrasts to be used for some or all of the factors appearing as variables in the model formula.
object             an object of class pcaNNet as returned by pcaNNet.
newdata            matrix or data frame of test examples. A vector is considered to be a row vector comprising a single case.
type               Type of output
...                arguments passed to nnet

Details
The function first will run principal component analysis on the data. The cumulative percentage of variance is computed for each principal component. The function uses the thresh argument to determine how many components must be retained to capture this amount of variance in the predictors.
The principal components are then used in a neural network model.
When predicting samples, the new data are similarly transformed using the information from the PCA analysis on the training data and then predicted.
Because the variance of each predictor is used in the PCA analysis, the code does a quick check to make sure that each predictor has at least two distinct values. If a predictor has one unique value, it is removed prior to the analysis.

Value
For pcaNNet, an object of "pcaNNet" or "pcaNNet.formula". Items of interest in the output are:
pc                the output from preprocess
model             the model generated from nnet
names             if any predictors had only one distinct value, this is a character string of the remaining columns. Otherwise a value of NULL

Author(s)
These are heavily based on the nnet code from Brian Ripley.

References

See Also
nnet, preprocess

Examples
data(BloodBrain)
modelFit <- pcaNNet(bbbDescr[, 1:10], logBBB, size = 5, linout = TRUE, trace = FALSE)
modelFit

predict(modelFit, bbbDescr[, 1:10])
**Description**

These functions plot the resampling results for the candidate subset sizes evaluated during the recursive feature elimination (RFE) process.

**Usage**

```r
# S3 method for class 'rfe'
plot(x, metric = x$metric, ...)  

# S3 method for class 'rfe'
ggplot(data = NULL, metric = data$metric[1], output = "layered", ...)
```

**Arguments**

- `x` an object of class `rfe`.
- `metric` What measure of performance to plot. Examples of possible values are "RMSE", "Rsquared", "Accuracy" or "Kappa". Other values can be used depending on what metrics have been calculated.
- `...` plot only: specifications to be passed to `xyplot`. The function automatically sets some arguments (e.g. axis labels) but passing in values here will over-ride the defaults.
- `data` an object of class `rfe`.
- `output` either "data", "ggplot" or "layered". The first returns a data frame while the second returns a simple `ggplot` object with no layers. The third value returns a plot with a set of layers.

**Details**

These plots show the average performance versus the subset sizes.

**Value**

a lattice or `ggplot` object

**Author(s)**

Max Kuhn

**References**

Kuhn (2008), “Building Predictive Models in R Using the caret” (http://www.jstatsoft.org/v28/i05/)
See Also

rfe, xyplot, ggplot

Examples

```r
## Not run:
data(BloodBrain)

x <- scale(bbbDescr[, -nearZeroVar(bbbDescr)])
x <- x[, -findCorrelation(cor(x), .8)]
x <- as.data.frame(x)

set.seed(1)
lmProfile <- rfe(x, log BBB,
                 sizes = c(2:25, 30, 35, 40, 45, 50, 55, 60, 65),
                 rfeControl = rfeControl(functions = lmFuncs,
                                          number = 200))

plot(lmProfile)
plot(lmProfile, metric = "Rsquared")
ggplot(lmProfile)

## End(Not run)
```

---

**plot.train**  
*Plot Method for the train Class*

**Description**

This function takes the output of a `train` object and creates a line or level plot using the `lattice` or `ggplot2` libraries.

**Usage**

```r
## S3 method for class 'train'
plot(x,  
     plotType = "scatter",
     metric = x$metric[1],
     digits =getOption("digits") - 3,
     xTrans = NULL,
     ...
)

## S3 method for class 'train'
ggplot(data = NULL,  
        metric = data$metric[1],
        plotType = "scatter",
        output = "layered",
        ...
)
```
plot.train

Arguments

- **x**: an object of class `train`.
- **metric**: What measure of performance to plot. Examples of possible values are "RMSE", "Rsquared", "Accuracy" or "Kappa". Other values can be used depending on what metrics have been calculated.
- **plotType**: a string describing the type of plot ("scatter", "level" or "line" (plot only))
- **digits**: an integer specifying the number of significant digits used to label the parameter value.
- **xTrans**: a function that will be used to scale the x-axis in scatter plots.
- **data**: an object of class `train`.
- **output**: either "data", "ggplot" or "layered". The first returns a data frame while the second returns a simple `ggplot` object with no layers. The third value returns a plot with a set of layers.

Details

If there are no tuning parameters, or none were varied, an error is produced.

If the model has one tuning parameter with multiple candidate values, a plot is produced showing the profile of the results over the parameter. Also, a plot can be produced if there are multiple tuning parameters but only one is varied.

If there are two tuning parameters with different values, a plot can be produced where a different line is shown for each value of one of the other parameter. For three parameters, the same line plot is created within conditioning panels/facets of the other parameter.

Also, with two tuning parameters (with different values), a levelplot (i.e. un-clustered heatmap) can be created. For more than two parameters, this plot is created inside conditioning panels/facets.

Author(s)

Max Kuhn

References

Kuhn (2008), “Building Predictive Models in R Using the caret” (http://www.jstatsoft.org/v28/i05/)

See Also

`train, levelplot, xyplot, stripplot, ggplot`
Examples

```r
## Not run:
library(klaR)
rdafit <- train(Species ~ .,
    data = iris,
    method = "rda",
    control = trainControl(method = "cv"))
plot(rdaFit)
plot(rdaFit, plotType = "level")

ggplot(rdaFit) + theme_bw()

## End(Not run)
```

---

**plot.varImp.train**  
*Plotting variable importance measures*

**Description**

This function produces lattice plots of objects with class "varImp.train". More info will be forthcoming.

**Usage**

```r
## S3 method for class 'varImp.train'
plot(x, top = dim(x$importance)[1], ...)
```

**Arguments**

- `x`  
an object with class varImp.
- `top`  
a scalar numeric that specifies the number of variables to be displayed (in order of importance)
- `...`  
arguments to pass to the lattice plot function (dotplot and panel.needle)

**Details**

For models where there is only one importance value, such as regression models, a "Pareto-type" plot is produced where the variables are ranked by their importance and a needle-plot is used to show the top variables.

When there is more than one importance value per predictor, the same plot is produced within conditioning panels for each class. The top predictors are sorted by their average importance.

**Value**

- a lattice plot object
plotClassProbs 

Author(s)
Max Kuhn

plotClassProbs  

Plot Predicted Probabilities in Classification Models

Description
This function takes an object (preferably from the function extractProb) and creates a lattice plot. If the call to extractProb included test data, these data are shown, but if unknowns were also included, these are not plotted.

Usage
plotClassProbs(object,

plotType = "histogram",

useObjects = FALSE,

...) 

Arguments

object an object (preferably from the function extractProb). There should be columns for each level of the class factor and columns named obs, pred, model (e.g. "rpart", "nnet" etc), dataType (e.g. "Training", "Test" etc) and optionally objects (for giving names to objects with the same model type).

plotType either "histogram" or "densityplot"

useObjects a logical; should the object name (if any) be used as a conditioning variable?

... parameters to pass to histogram or densityplot

Value
A lattice object. Note that the plot has to be printed to be displayed (especially in a loop).

Author(s)
Max Kuhn

Examples
## Not run:
data(mdrr)
set.seed(90)
inTrain <- createDataPartition(mdrrClass, p = .5)[[1]]

trainData <- mdrrDescr[inTrain,1:20]
testData <- mdrrDescr[-inTrain,1:20]
plotObsVsPred

Plot Observed versus Predicted Results in Regression and Classification Models

Description

This function takes an object (preferably from the function extractPrediction) and creates a lattice plot. For numeric outcomes, the observed and predicted data are plotted with a 45 degree reference line and a smoothed fit. For factor outcomes, a dotplot plot is produced with the accuracies for the different models.

If the call to extractPrediction included test data, these data are shown, but if unknowns were also included, they are not plotted.

Usage

plotObsVsPred(object, equalRanges = TRUE, ...)

trainY <- mdrrClass[inTrain]
testY <- mdrrClass[-inTrain]

ctrl <- trainControl(method = "cv")
nbFit1 <- train(trainData, trainY, "nb",
    trControl = ctrl,
    tuneGrid = data.frame(usekernel = TRUE, fL = 0))
nbFit2 <- train(trainData, trainY, "nb",
    trControl = ctrl,
    tuneGrid = data.frame(usekernel = FALSE, fL = 0))

models <- list(para = nbFit2, nonpara = nbFit1)
predProbs <- extractProb(models, testX = testData, testY = testY)

plotClassProbs(predProbs, useObjects = TRUE)
plotClassProbs(predProbs,
    subset = object == "para" & dataType == "Test")
plotClassProbs(predProbs,
    useObjects = TRUE,
    plotType = "densityplot",
    auto.key = list(columns = 2))

## End(Not run)
**plotObsVsPred**

**Arguments**

- **object**
  - An object (preferably from the function `extractPrediction`). There should be columns named `obs`, `pred`, `model` (e.g. "rpart", "nnet" etc) and `dataType` (e.g. "Training", "Test" etc).

- **equalRanges**
  - A logical; should the x- and y-axis ranges be the same?

- **...**
  - Parameters to pass to `xyplot` or `dotplot`, such as `auto.key`.

**Value**

A lattice object. Note that the plot has to be printed to be displayed (especially in a loop).

**Author(s)**

Max Kuhn

**Examples**

```r
## Not run:
# regression example
data(BostonHousing)

rpartFit <- train(BostonHousing[1:100, -c(4, 14)],
                  BostonHousing$medv[1:100],
                  "rpart", tuneLength = 9)

plsFit <- train(BostonHousing[1:100, -c(4, 14)],
                 BostonHousing$medv[1:100],
                 "pls")

predVals <- extractPrediction(list(rpartFit, plsFit),
                              testX = BostonHousing[101:200, -c(4, 14)],
                              testY = BostonHousing$medv[101:200],
                              unkX = BostonHousing[201:300, -c(4, 14)])

plotObsVsPred(predVals)

# classification example
data(Satellite)

numSamples <- dim(Satellite)[1]
set.seed(716)

varIndex <- 1:numSamples

trainSamples <- sample(varIndex, 150)

varIndex <- (1:numSamples)[-trainSamples]

testSamples <- sample(varIndex, 100)

varIndex <- (1:numSamples)[-c(testSamples, trainSamples)]

unkSamples <- sample(varIndex, 50)

trainX <- Satellite[trainSamples, -37]
```
Partial Least Squares and Sparse Partial Least Squares Discriminant Analysis

Description

`plsda` is used to fit standard PLS models for classification while `splsda` performs sparse PLS that embeds feature selection and regularization for the same purpose.

Usage

```r
plsda(x, ...)  
## Default S3 method:
plsda(x, y, ncomp = 2, probMethod = "softmax", prior = NULL, ...)  
## S3 method for class 'plsda'
predict(object, newdata = NULL, ncomp = NULL, type = "class", ...)

splsda(x, ...)  
## Default S3 method:
splsda(x, y, probMethod = "softmax", prior = NULL, ...)  
## S3 method for class 'splsda'
predict(object, newdata = NULL, type = "class", ...)
```
Arguments

- **x**: a matrix or data frame of predictors
- **y**: a factor or indicator matrix for the discrete outcome. If a matrix, the entries must be either 0 or 1 and rows must sum to one
- **ncomp**: the number of components to include in the model. Predictions can be made for models with values less than ncomp.
- **probMethod**: either "softmax" or "Bayes" (see Details)
- **prior**: a vector or prior probabilities for the classes (only used for probMethod = "Bayes")
- **...**: arguments to pass to plsr or spls. For splsda, this is the method for passing tuning parameters specifications (e.g. K, eta or kappa)
- **object**: an object produced by plsda
- **newdata**: a matrix or data frame of predictors
- **type**: either "class", "prob" or "raw" to produce the predicted class, class probabilities or the raw model scores, respectively.

Details

If a factor is supplied, the appropriate indicator matrix is created.

A multivariate PLS model is fit to the indicator matrix using the plsr or spls function.

Two prediction methods can be used.

The **softmax function** transforms the model predictions to "probability-like" values (e.g. on [0, 1] and sum to 1). The class with the largest class probability is the predicted class.

Also, **Bayes rule** can be applied to the model predictions to form posterior probabilities. Here, the model predictions for the training set are used along with the training set outcomes to create conditional distributions for each class. When new samples are predicted, the raw model predictions are run through these conditional distributions to produce a posterior probability for each class (along with the prior). This process is repeated ncomp times for every possible PLS model. The **NaiveBayes function** is used with usekernel = TRUE for the posterior probability calculations.

Value

For plsda, an object of class "plsda" and "mvr". For splsda, an object of class splsda.

The predict methods produce either a vector, matrix or three-dimensional array, depending on the values of type and ncomp. For example, specifying more than one value of ncomp with type = "class" with produce a three-dimensional array but the default specification would produce a factor vector.

See Also

plsr, spls
Examples

```r
## Not run:
data(mdrr)
s.set.seed(1)
inTrain <- sample(seq(along = mdrrClass), 450)
nzv <- nearZeroVar(mdrrDescr)
filteredDescr <- mdrrDescr[, -nzv]
training <- filteredDescr[inTrain,]
test <- filteredDescr[-inTrain,]
trainMDRR <- mdrrClass[inTrain]
testMDRR <- mdrrClass[-inTrain]

preProcValues <- preProcess(training)

trainDescr <- predict(preProcValues, training)
testDescr <- predict(preProcValues, test)

useBayes <- plsda(trainDescr, trainMDRR, ncomp = 5,
                   probMethod = "Bayes")
useSoftmax <- plsda(trainDescr, trainMDRR, ncomp = 5)

confusionMatrix(predict(useBayes, testDescr),
                 testMDRR)

confusionMatrix(predict(useSoftmax, testDescr),
                 testMDRR)

histogram(~predict(useBayes, testDescr, type = "prob")[,"Active",]
          | testMDRR, xlab = "Active Prob", xlim = c(-1,1))

histogram(~predict(useSoftmax, testDescr, type = "prob")[,"Active",]
          | testMDRR, xlab = "Active Prob", xlim = c(-1,1))

## different sized objects are returned
length(predict(useBayes, testDescr))
dim(predict(useBayes, testDescr, ncomp = 1:3))
dim(predict(useBayes, testDescr, type = "prob"))
dim(predict(useBayes, testDescr, type = "prob", ncomp = 1:3))

## Using spls:
## (As of 11/09, the spls package now has a similar function with
## the same name. To avoid conflicts, use caret::splsda to
## get this version)
splsFit <- caret::splsda(trainDescr, trainMDRR,
                         K = 5, eta = .9,
                         probMethod = "Bayes")

confusionMatrix(caret::predict.splsda(splsFit, testDescr),
                testMDRR)
```
postResample

Calculates performance across resamples

Description

Given two numeric vectors of data, the mean squared error and R-squared are calculated. For two factors, the overall agreement rate and Kappa are determined.

Usage

postResample(pred, obs)
defaultSummary(data, lev = NULL, model = NULL)
twoClassSummary(data, lev = NULL, model = NULL)
R2(pred, obs, formula = "corr", na.rm = FALSE)
RMSE(pred, obs, na.rm = FALSE)

getTrainPerf(x)

Arguments

pred A vector of numeric data (could be a factor)
obs A vector of numeric data (could be a factor)
data a data frame or matrix with columns obs and pred for the observed and predicted outcomes. For twoClassSummary, columns should also include predicted probabilities for each class. See the classProbs argument to trainControl
lev a character vector of factors levels for the response. In regression cases, this would be NULL.
model a character string for the model name (as taken form the method argument of train.
formula which $R^2$ formula should be used? Either "corr" or "traditional". See Kvalseth (1985) for a summary of the different equations.
na.rm a logical value indicating whether NA values should be stripped before the computation proceeds.
x an object of class train.
Details

`postResample` is meant to be used with `apply` across a matrix. For numeric data the code checks to see if the standard deviation of either vector is zero. If so, the correlation between those samples is assigned a value of zero. NA values are ignored everywhere.

Note that many models have more predictors (or parameters) than data points, so the typical mean squared error denominator (n - p) does not apply. Root mean squared error is calculated using $\sqrt{\text{mean}((\text{pred} - \text{obs})^2)}$. Also, $R^2$ is calculated wither using as the square of the correlation between the observed and predicted outcomes when `form = "corr"`. when `form = "traditional"`,

$$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y}_i)^2}$$

For `defaultSummary` is the default function to compute performance metrics in `train`. It is a wrapper around `postResample`.

twoClassSummary computes sensitivity, specificity and the area under the ROC curve. To use this function, the `classProbs` argument of `trainControl` should be `TRUE`.

Other functions can be used via the `summaryFunction` argument of `trainControl`. Custom functions must have the same arguments as `defaultSummary`.

The function `getTrainPerf` returns a one row data frame with the resampling results for the chosen model. The statistics will have the prefix "Train" (i.e. "TrainROC"). There is also a column called "method" that echoes the argument of the call to `trainControl` of the same name.

Value

A vector of performance estimates.

Author(s)

Max Kuhn

References


See Also

`trainControl`

Examples

```r
predicted <- matrix(rnorm(50), ncol = 5)
observed <- rnorm(10)
apply(predicted, 2, postResample, obs = observed)
```
Description

Measurements of 58 pottery samples.

Usage

data(pottery)

Value

pottery 11 elemental composition measurements
potteryClass factor of pottery type: black carbon containing bulks (A) and clayey (B)

Source


-----------------

### prcomp.resamples

*Principal Components Analysis of Resampling Results*

Description

Performs a principal components analysis on an object of class `resamples` and returns the results as an object with classes `prcomp.resamples` and `prcomp`.

Usage

```r
## S3 method for class 'resamples'
prcomp(x, metric = x$metrics[1], ...)

cluster(x, ...)
## S3 method for class 'resamples'
cluster(x, metric = x$metrics[1], ...)

## S3 method for class 'prcomp.resamples'
plot(x, what = "scree", dims = max(2, ncol(x$rotation)), ...)
```
prcomp.resamples

Arguments

x For prcomp, an object of class resamples and for plot.prcomp.resamples, an object of class plot.prcomp.resamples

metric a performance metric that was estimated for every resample

what the type of plot: "scree" produces a bar chart of standard deviations, "cumulative" produces a bar chart of the cumulative percent of variance, "loadings" produces a scatterplot matrix of the loading values and "components" produces a scatterplot matrix of the PCA components

dims The number of dimensions to plot when what = "loadings" or what = "components"

... For prcomp.resamples, options to pass to prcomp, for plot.prcomp.resamples, options to pass to Lattice objects (see Details below) and, for cluster.resamples, options to pass to hclust.

Details

The principal components analysis treats the models as variables and the resamples are realizations of the variables. In this way, we can use PCA to "cluster" the assays and look for similarities. Most of the methods for prcomp can be used, although custom print and plot methods are used.

The plot method uses lattice graphics. When what = "scree" or what = "cumulative", barchart is used. When what = "loadings" or what = "components", either xyplot or splom are used (the latter when dims > 2). Options can be passed to these methods using ...

When what = "loadings" or what = "components", the plots are put on a common scale so that later components are less likely to be over-interpreted. See Geladi et al (2003) for examples of why this can be important.

For clustering, hclust is used to determine clusters of models based on the resampled performance values.

Value

For prcomp.resamples, an object with classes prcomp.resamples and prcomp. This object is the same as the object produced by prcomp, but with additional elements:

metric the value for the metric argument
call the call

For plot.prcomp.resamples, a Lattice object (see Details above)

Author(s)

Max Kuhn

References

predict.bagEarth

Predicted values based on bagged Earth and FDA models

**Description**

Predicted values based on bagged Earth and FDA models

**Usage**

```r
# S3 method for class 'bagEarth'
predict(object, newdata = NULL, type = "response", ...)
# S3 method for class 'bagFDA'
predict(object, newdata = NULL, type = "class", ...)
```

**Arguments**

- `object`: Object of class inheriting from `bagEarth`
- `newdata`: An optional data frame or matrix in which to look for variables with which to predict. If omitted, the fitted values are used (see note below).

**Examples**

```r
## Not run:
#load(url("http://caret.r-forge.r-project.org/exampleModels.RData"))

resamps <- resamples(list(CART = rpartFit,
                          CondInfTree = ctreeFit,
                          MARS = earthFit))
resampPCA <- prcomp(resamps)

resampPCA

plot(resampPCA, what = "scree")
plot(resampPCA, what = "components")
plot(resampPCA, what = "components", dims = 2, auto.key = list(columns = 3))

clustered <- cluster(resamps)
plot(clustered)

## End(Not run)
```
type

The type of prediction. For bagged earth regression model, type = "response" will produce a numeric vector of the usual model predictions. earth also allows the user to fit generalized linear models. In this case, type = "response" produces the inverse link results as a vector. In the case of a binomial generalized linear model, type = "response" produces a vector of probabilities, type = "class" generates a factor vector and type = "prob" produces a 2 column matrix with probabilities for both classes (averaged across the individual models). Similarly, for bagged fda models, type = "class" generates a factor vector and type = "probs" outputs a matrix of class probabilities.

... not used

Value

a vector of predictions

Note

If the predictions for the original training set are needed, there are two ways to calculate them. First, the original data set can be predicted by each bagged earth model. Secondly, the predictions form each bootstrap sample could be used (but are more likely to overfit). If the original call to bagEarth or bagFDA had keepX = TRUE, the first method is used, otherwise the values are calculated via the second method.

Author(s)

Max Kuhn

See Also

bagEarth

Examples

```r
## Not run:
data(trees)
## out of bag predictions vs just re-predicting the training set
fit1 <- bagEarth(Volume ~ ., data = trees, keepX = TRUE)
fit2 <- bagEarth(Volume ~ ., data = trees, keepX = FALSE)
hist(predict(fit1) - predict(fit2))

## End(Not run)
```
predict.knn3

Predictions from k-Nearest Neighbors

Description

Predict the class of a new observation based on k-NN.

Usage

## S3 method for class 'knn3'
predict(object, newdata, type=c("prob", "class"), ...)

Arguments

- object: object of class knn3.
- newdata: a data frame of new observations.
- type: return either the predicted class or the proportion of the votes for the winning class.
- ...: additional arguments.

Details

This function is a method for the generic function `predict` for class knn3. For the details see knn3. This is essentially a copy of `predict.ipredknn`.

Value

Either the predicted class or the proportion of the votes for each class.

Author(s)

predict.ipredknn by Torsten.Hothorn <Torsten.Hothorn@rzmail.uni-erlangen.de>

predict.knnreg

Predictions from k-Nearest Neighbors Regression Model

Description

Predict the outcome of a new observation based on k-NN.

Usage

## S3 method for class 'knnreg'
predict(object, newdata, ...)


predict.train

Arguments

object  object of class knnreg.
newdata  a data frame or matrix of new observations.
...  additional arguments.

Details

This function is a method for the generic function predict for class knnreg. For the details see knnreg. This is essentially a copy of predict.ipredknn.

Value

a numeric vector

Author(s)

Max Kuhn, Chris Keefer, adapted from knn and predict.ipredknn

---

predict.train  Extract predictions and class probabilities from train objects

Description

These functions can be used for a single train object or to loop through a number of train objects to calculate the training and test data predictions and class probabilities.

Usage

```r
# S3 method for class 'list'
predict(object, ...)

# S3 method for class 'train'
predict(object, newdata = NULL, type = "raw", na.action = na.omit, ...)

extractPrediction(models,
                   testX = NULL, testY = NULL,
                   unkX = NULL,
                   unkOnly = !is.null(unkX) & is.null(testX),
                   verbose = FALSE)

extractProb(models,
            testX = NULL, testY = NULL,
            unkX = NULL,
            unkOnly = !is.null(unkX) & is.null(testX),
            verbose = FALSE)
```

**predict.train**

Arguments

- **object**
  For `predict.train`, an object of class `train`. For `predict.list`, a list of objects of class `train`.

- **newdata**
  an optional set of data to predict on. If `NULL`, then the original training data are used.

- **type**
  either "raw" or "prob", for the number/class predictions or class probabilities, respectively. Class probabilities are not available for all classification models.

- **models**
  a list of objects of the class `train`. The objects must have been generated with `fitBest = FALSE` and `returnData = TRUE`.

- **na.action**
  the method for handling missing data.

- **testX**
  an optional set of data to predict.

- **testY**
  an optional outcome corresponding to the data given in `testX`.

- **unkX**
  another optional set of data to predict without known outcomes.

- **unkOnly**
  a logical to bypass training and test set predictions. This is useful if speed is needed for unknown samples.

- **verbose**
  a logical for printing messages.

- **...**
  additional arguments to be passed to other methods.

Details

These functions are wrappers for the specific prediction functions in each modeling package. In each case, the optimal tuning values given in the `tuneValue` slot of the `finalModel` object are used to predict.

To get simple predictions for a new data set, the `predict` function can be used. Limits can be imposed on the range of predictions. See `trainControl` for more information.

To get predictions for a series of models at once, a list of `train` objects can be passed to the `predict` function and a list of model predictions will be returned.

The two extraction functions can be used to get the predictions and observed outcomes at once for the training, test and/or unknown samples at once in a single data frame (instead of a list of just the predictions). These objects can then be passed to `plotObsVsPred` or `plotClassProbs`.

Value

For `predict.train`, a vector of predictions if `type = "raw"` or a data frame of class probabilities for `type = "probs"`. In the latter case, there are columns for each class.

For `predict.list`, a list results. Each element is produced by `predict.train`.

For `extractPrediction`, a data frame with columns:

- **obs**
  the observed training and test data

- **pred**
  predicted values

- **model**
  the type of model used to predict

- **object**
  the names of the objects within `models`. If `models` is an un-named list, the values of `object` will be "Object1", "Object2" and so on.
predictors

dataType "Training", "Test" or "Unknown" depending on what was specified

For extractProb, a data frame. There is a column for each class containing the probabilities. The remaining columns are the same as above (although the pred column is the predicted class)

Author(s)
Max Kuhn

References
Kuhn (2008), “Building Predictive Models in R Using the caret” (http://www.jstatsoft.org/v28/i05/)

See Also
plotObsVsPred, plotClassProbs, trainControl

Examples

```r
## Not run:

knnFit <- train(Species ~ ., data = iris, method = "knn",
                 trControl = trainControl(method = "cv"))

rdaFit <- train(Species ~ ., data = iris, method = "rda",
                 trControl = trainControl(method = "cv"))

predict(knnFit)
predict(knnFit, type = "prob")

bothModels <- list(knn = knnFit,
                   tree = rdaFit)

predict(bothModels)

extractPrediction(bothModels, testX = iris[1:10, -5])
extractProb(bothModels, testX = iris[1:10, -5])

## End(Not run)
```

predictors               List predictors used in the model

Description

This class uses a model fit to determine which predictors were used in the final model.
Usage

predictors(x, ...)

## Default S3 method:
predictors(x, ...)

## S3 method for class 'formula'
predictors(x, ...)

## S3 method for class 'list'
predictors(x, ...)

## S3 method for class 'rfe'
predictors(x, ...)

## S3 method for class 'sbf'
predictors(x, ...)

## S3 method for class 'terms'
predictors(x, ...)

## S3 method for class 'train'
predictors(x, ...)

Arguments

x       a model object, list or terms
...
not currently used

Details

For `randomForest`, `cforest`, `ctree`, `rpart`, `ipredbagg`, `bagging`, `earth`, `fda`, `pamr.train`, `superpc.train`, `bagEarth` and `bagFDA`, an attempt was made to report the predictors that were actually used in the final model.

The `predictors` function can be called on the model object (as opposed to the `train`) object and the package will try to find the appropriate coed (if it exists).

In cases where the predictors cannot be determined, NA is returned. For example, `nnet` may return missing values from `predictors`.

Value

a character string of predictors or NA.
**Pre-Processing of Predictors**

**Description**

Pre-processing transformation (centering, scaling etc) can be estimated from the training data and applied to any data set with the same variables.

**Usage**

```r
preProcess(x, ...)  
```  
```r  
## Default S3 method: 
preProcess(x,  
  method = c("center", "scale"),  
  thresh = 0.95,  
  pcaComp = NULL,  
  na.remove = TRUE,  
  k = 5,  
  knnSummary = mean,  
  outcome = NULL,  
  fudge = .2,  
  numUnique = 3,  
  verbose = FALSE,  
  ...)  
```  
```r  
## S3 method for class 'preProcess' 
predict(object, newdata, ...)  
```  
**Arguments**

- **x** a matrix or data frame. All variables must be numeric.
- **method** a character vector specifying the type of processing. Possible values are "Box-Cox", "YeoJohnson", "expoTrans", "center", "scale", "range", "knnImpute", "bagImpute", "medianImpute", "ica" and "spatialSign" (see Details below)
- **thresh** a cutoff for the cumulative percent of variance to be retained by PCA
- **pcaComp** the specific number of PCA components to keep. If specified, this over-rides thresh
- **na.remove** a logical; should missing values be removed from the calculations?
- **object** an object of class preProcess
- **newdata** a matrix or data frame of new data to be pre-processed
- **k** the number of nearest neighbors from the training set to use for imputation
- **knnSummary** function to average the neighbor values per column during imputation
outcome a numeric or factor vector for the training set outcomes. This can be used to help estimate the Box-Cox transformation of the predictor variables (see Details below)

fudge a tolerance value: Box-Cox transformation lambda values within +/-fudge will be coerced to 0 and within 1+/fudge will be coerced to 1

numUnique how many unique values should \( y \) have to estimate the Box-Cox transformation?

verbose a logical: prints a log as the computations proceed

Details

The Box-Cox, Yeo-Johnson and exponential transformations have been "repurposed" here: they are being used to transform the predictor variables. The Box-Cox transformation was developed for transforming the response variable while another method, the Box-Tidwell transformation, was created to estimate transformations of predictor data. However, the Box-Cox method is simpler, more computationally efficient and is equally effective for estimating power transformations. The Yeo-Johnson transformation is similar to the Box-Cox model but can accommodate predictors with zero and/or negative values (while the predictors values for the Box-Cox transformation must be strictly positive.) The exponential transformation of Manly (1976) can also be used for positive or negative data.

The "range" transformation scales the data to be within \([0, 1]\). If new samples have values larger or smaller than those in the training set, values will be outside of this range.

The operations are applied in this order: Box-Cox/Yeo-Johnson transformation, centering, scaling, range, imputation, PCA, ICA then spatial sign. This is a departure from versions of \texttt{caret} prior to version 4.76 (where imputation was done first) and is not backwards compatible if bagging was used for imputation.

If PCA is requested but centering and scaling are not, the values will still be centered and scaled. Similarly, when ICA is requested, the data are automatically centered and scaled.

k-nearest neighbor imputation is carried out by finding the k closest samples (Euclidian distance) in the training set. Imputation via bagging fits a bagged tree model for each predictor (as a function of all the others). This method is simple, accurate and accepts missing values, but it has much higher computational cost. Imputation via medians takes the median of each predictor in the training set, and uses them to fill missing values. This method is simple, fast, and accepts missing values, but treats each predictor independently, and may be inaccurate.

A warning is thrown if both PCA and ICA are requested. ICA, as implemented by the \texttt{fastICA} package automatically does a PCA decomposition prior to finding the ICA scores.

The function will throw an error of any variables in \( x \) has less than two unique values.

Value

\texttt{preProcess} results in a list with elements

\begin{itemize}
  \item \texttt{call} the function call
  \item \texttt{dim} the dimensions of \( x \)
  \item \texttt{bc} Box-Cox transformation values, see \texttt{BoxCoxTrans}
\end{itemize}
mean a vector of means (if centering was requested)
std a vector of standard deviations (if scaling or PCA was requested)
rotation a matrix of eigenvectors if PCA was requested
method the value of method
thresh the value of thresh
ranges a matrix of min and max values for each predictor when method includes "range" (and NULL otherwise)
numComp the number of principal components required of capture the specified amount of variance
ica contains values for the W and K matrix of the decomposition
median a vector of medians (if median imputation was requested)

Author(s)
Max Kuhn, median imputation by Zachary Mayer

References

http://caret.r-forge.r-project.org/preprocess.html

Kuhn (2008), Building predictive models in R using the caret (http://www.jstatsoft.org/v28/i05/)


See Also

BoxCoxTrans, expoTrans boxcox, prcomp, fastICA, spatialSign

Examples

data(BloodBrain)
# one variable has one unique value
## Not run:
preProc <- preProcess(bbbDescr)

preProc <- preProcess(bbbDescr[1:100,-3])
training <- predict(preProc, bbbDescr[1:100,-3])
test <- predict(preProc, bbbDescr[101:208,-3])

## End(Not run)
print.confusionMatrix  Print method for confusionMatrix

Description

A print method for confusionMatrix

Usage

```r
## S3 method for class 'confusionMatrix'
print(x, digits = max(3,getOption("digits") - 3),
      printStats = TRUE, ...)
```

Arguments

- `x`: an object of class confusionMatrix
- `digits`: number of significant digits when printed
- `printStats`: a logical: if TRUE then table statistics are also printed
- `...`: optional arguments to pass to print.table

Value

`x` is invisibly returned

Author(s)

Max Kuhn

See Also

- confusionMatrix

print.train  Print Method for the train Class

Description

Print the results of a train object.
Usage

```r
## S3 method for class 'train'
print(x, 
    digits = min(3,getOption("digits") - 3),
    printCall = FALSE,
    details = FALSE,
    selectCol = FALSE,
    ...
)
```

Arguments

- `x`: an object of class `train`.
- `digits`: an integer specifying the number of significant digits to print.
- `printCall`: a logical to print the call at the top of the output.
- `details`: a logical to show print or summary methods for the final model. In some cases (such as `gbm`, `knn`, `lvq`, naive Bayes and bagged tree models), no information will be printed even if `details = TRUE`.
- `selectCol`: a logical to a column with a star next to the final model.
- `...`: options passed to the generic print method.

Details

The table of complexity parameters used, their resampled performance and a flag for which rows are optimal.

Value

A data frame with the complexity parameter(s) and performance (invisibly).

Author(s)

Max Kuhn

See Also

- `train`

Examples

```r
## Not run:
data(iris)
TrainData <- iris[,1:4]
TrainClasses <- iris[,5]

library(klaR)
rdafit <- train(TrainData, TrainClasses, method = "rda",
               control = trainControl(method = "cv"))
print(rdaFit)
```
resampleHist

## End(Not run)

---

| resampleHist | Plot the resampling distribution of the model statistics |

### Description

Create a lattice histogram or densityplot from the resampled outcomes from a `train` object.

### Usage

```r
code
resampleHist(object, type = "density", ...)
```

### Arguments

- **object**: an object resulting from a call to `train`
- **type**: a character string. Either "hist" or "density"
- **...**: options to pass to histogram or densityplot

### Details

All the metrics from the object are plotted, but only for the final model. For more comprehensive plots functions, see `histogram.train, densityplot.train, xyplot.train, stripplot.train`.

For the plot to be made, the `returnResamp` argument in `trainControl` should be either "final" or "all".

### Value

A object of class `trellis`

### Author(s)

Max Kuhn

### See Also

`train, histogram, densityplot, histogram.train, densityplot.train, xyplot.train, stripplot.train`
Examples

```r
## Not run:
data(iris)
TrainData <- iris[,1:4]
TrainClasses <- iris[,5]

knnFit <- train(TrainData, TrainClasses, "knn")
resampleHist(knnFit)

## End(Not run)
```

---

resamples  
Collation and Visualization of Resampling Results

Description

These functions provide methods for collection, analyzing and visualizing a set of resampling results from a common data set.

Usage

```
resamples(x, ...)
```

```
## Default S3 method:
resamples(x, modelNames = names(x), ...)
```

```
## S3 method for class 'resamples'
summary(object, ...)
```

```
## S3 method for class 'resamples'
sort(x, decreasing = FALSE, metric = x$metric[1], FUN = mean, ...)
modelCor(x, metric = x$metric[1], ...)
```

Arguments

- **x**: a list of two or more objects of class `train, sbf` or `rfe` with a common set of resampling indices in the control object. For `sort.resamples`, it is an object generated by `resamples`.
- **modelNames**: an optional set of names to give to the resampling results
- **object**: an object generated by `resamples`
- **metric**: a character string for the performance measure used to sort or computing the between-model correlations
- **decreasing**: logical. Should the sort be increasing or decreasing?
resamples

FUN

a function whose first argument is a vector and returns a scalar, to be applied to each model's performance measure.

... only used for sort and modelCor and captures arguments to pass to sort or FUN.

Details


The results from `train` can have more than one performance metric per resample. Each metric in the input object is saved.

`resamples` checks that the resampling results match; that is, the indices in the object `trainObject$control$index` are the same. Also, the argument `trainControl$returnResamp` should have a value of "final" for each model.

The summary function computes summary statistics across each model/metric combination.

Value

For `resamples`: an object with class "resamples" with elements

call the call
values a data frame of results where rows correspond to resampled data sets and columns indicate the model and metric
models a character string of model labels
metrics a character string of performance metrics
methods a character string of the `train` method argument values for each model

For `sort.resamples` a character string in the sorted order is generated. `modelCor` returns a correlation matrix.

Author(s)

Max Kuhn

References


See Also

`train`, `trainControl`, `diff.resamples`, `xyplot.resamples`, `densityplot.resamples`, `bwplot.resamples`, `splom.resamples`
resampleSummary

Summary of resampled performance estimates

Description

This function uses the out-of-bag predictions to calculate overall performance metrics and returns the observed and predicted data.

Usage

resampleSummary(obs, resampled, index = NULL, keepData = TRUE)
rfe

Backwards Feature Selection

Description

A simple backwards selection, a.k.a. recursive feature selection (RFE), algorithm

Arguments

- **obs**: A vector (numeric or factor) of the outcome data
- **resampled**: For bootstrapping, this is either a matrix (for numeric outcomes) or a data frame (for factors). For cross-validation, a vector is produced.
- **index**: The list to index of samples in each cross-validation fold (only used for cross-validation).
- **keepData**: A logical for returning the observed and predicted data.

Details

The mean and standard deviation of the values produced by postResample are calculated.

Value

A list with:

- **metrics**: A vector of values describing the bootstrap distribution.
- **data**: A data frame or NULL. Columns include obs, pred and group (for tracking cross-validation folds or bootstrap samples)

Author(s)

Max Kuhn

See Also

postResample

Examples

```r
resampleSummary(rnorm(10), matrix(rnorm(50), ncol = 5))
```
Usage

rfe(x, ...)

## Default S3 method:
rfe(x, y,
   sizes = 2^c(2:4),
   metric = ifelse(is.factor(y), "Accuracy", "RMSE"),
   maximize = ifelse(metric == "RMSE", FALSE, TRUE),
   rfeControl = rfeControl(),
   ...)

rfeIter(x, y,
   testX, testY,
   sizes,
   rfeControl = rfeControl(),
   label = "",
   seeds = NA,
   ...)

## S3 method for class 'rfe'
predict(object, newdata, ...)

Arguments

x a matrix or data frame of predictors for model training. This object must have unique column names.
y a vector of training set outcomes (either numeric or factor)
testX a matrix or data frame of test set predictors. This must have the same column names as x
testY a vector of test set outcomes
sizes a numeric vector of integers corresponding to the number of features that should be retained
metric a string that specifies what summary metric will be used to select the optimal model. By default, possible values are "RMSE" and "Rsquared" for regression and "Accuracy" and "Kappa" for classification. If custom performance metrics are used (via the functions argument in rfeControl), the value of metric should match one of the arguments.
maximize a logical: should the metric be maximized or minimized?
rfeControl a list of options, including functions for fitting and prediction. The web page http://caret.r-forge.r-project.org/ has more details and examples related to this function.
object an object of class rfe
newdata a matrix or data frame of new samples for prediction
label an optional character string to be printed when in verbose mode.
seeds is an optional vector of integers for the size. The vector should have length of 
length(sizes)+1

... options to pass to the model fitting function (ignored in predict.rfe)

Details

More details on this function can be found at http://caret.r-forge.r-project.org/featureselection.html.

This function implements backwards selection of predictors based on predictor importance ranking. The predictors are ranked and the less important ones are sequentially eliminated prior to modeling. The goal is to find a subset of predictors that can be used to produce an accurate model. The web page http://caret.r-forge.r-project.org/ has more details and examples related to this function.

rfe can be used with "explicit parallelism", where different resamples (e.g. cross-validation group) can be split up and run on multiple machines or processors. By default, rfe will use a single processor on the host machine. As of version 4.99 of this package, the framework used for parallel processing uses the foreach package. To run the resamples in parallel, the code for rfe does not change; prior to the call to rfe, a parallel backend is registered with foreach (see the examples below).

rfeIter is the basic algorithm while rfe wraps these operations inside of resampling. To avoid selection bias, it is better to use the function rfe than rfeIter.

Value

A list with elements

finalVariables a list of size length(sizes) + 1 containing the column names of the “surviving” predictors at each stage of selection. The first element corresponds to all the predictors (i.e. size = ncol(x))

def a data frame with columns for the test set outcome, the predicted outcome and the subset size.

Author(s)

Max Kuhn

See Also

rfeControl

Examples

## Not run:
data(BloodBrain)

x <- scale(bbbDescr[, -nearZeroVar(bbbDescr)])

x <- x[, -findCorrelation(cor(x), .8)]

x <- as.data.frame(x)
set.seed(1)
lmProfile <- rfe(x, logBBB,
    sizes = c(2:25, 30, 35, 40, 45, 50, 55, 60, 65),
    rfeControl = rfeControl(functions = lmFuncs, 
                   number = 200))

set.seed(1)
lmProfile2 <- rfe(x, logBBB,
    sizes = c(2:25, 30, 35, 40, 45, 50, 55, 60, 65),
    rfeControl = rfeControl(functions = lmFuncs, 
                   rerank = TRUE, 
                   number = 200))

xyplot(lmProfile$results$RMSE + lmProfile2$results$RMSE ~ 
    lmProfile$results$Variables,
    type = c("g", "p", "l"),
    auto.key = TRUE)

rfProfile <- rfe(x, logBBB,
    sizes = c(2, 5, 10, 20),
    rfeControl = rfeControl(functions = rffuncs))

bagProfile <- rfe(x, logBBB,
    sizes = c(2, 5, 10, 20),
    rfeControl = rfeControl(functions = treebagFuncs))

set.seed(1)
svmProfile <- rfe(x, logBBB,
    sizes = c(2, 5, 10, 20),
    rfeControl = rfeControl(functions = caretFuncs, 
                   number = 200),
    ## pass options to train()
    method = "svmRadial")

## classification

data(mdrr)
mdrrDescr <- mdrrDescr[, -nearZeroVar(mdrrDescr)]
mdrrDescr <- mdrrDescr[, -findCorrelation(cor(mdrrDescr), .8)]

set.seed(1)
inTrain <- createDataPartition(mdrrClass, p = .75, list = FALSE)[,1]

train <- mdrrDescr[ inTrain, ]
test <- mdrrDescr[-inTrain, ]
trainClass <- mdrrClass[ inTrain]
testClass <- mdrrClass[-inTrain]

set.seed(2)
ldaProfile <- rfe(train, trainClass,
    sizes = c(1:10, 15, 30),
    rfeControl = rfeControl(functions = ldaFuncs, method = "cv"))
plot(ldaProfile, type = c("o", "g"))
postResample(predict(ldaProfile, test), testClass)

## End(Not run)

############################################################
# Parallel Processing Example via multicore

## Not run:
library(doMC)

## Note: if the underlying model also uses foreach, the
## number of cores specified above will double (along with
## the memory requirements)
registerDoMC(cores = 2)

set.seed(1)
lmProfile <- rfe(x, logBBB,
    sizes = c(2:25, 30, 35, 40, 45, 50, 55, 60, 65),
    rfeControl = rfeControl(functions = lmFuncs,
                              number = 200))

## End(Not run)

rfeControl

Controlling the Feature Selection Algorithms

Description

This function generates a control object that can be used to specify the details of the feature selection algorithms used in this package.

Usage

rfeControl(functions = NULL,
    rerank = FALSE,
    method = "boot",
    saveDetails = FALSE,
    number = ifelse(method %in% c("cv", "repeatedcv"), 10, 25),
    repeats = ifelse(method %in% c("cv", "repeatedcv"), 1, number),
    verbose = FALSE,
    returnResamp = "final",
    p = .75,
    index = NULL,
    timingSamps = 0,
    seeds = NA,
    allowParallel = TRUE)
Arguments

functions: a list of functions for model fitting, prediction and variable importance (see Details below)
rerank: a logical: should variable importance be re-calculated each time features are removed?
method: The external resampling method: boot, cv, LOOCV or LGOCV (for repeated training/test splits)
number: Either the number of folds or number of resampling iterations
repeats: For repeated k-fold cross-validation only: the number of complete sets of folds to compute
saveDetails: a logical to save the predictions and variable importances from the selection process
verbose: a logical to print a log for each external resampling iteration
returnResamp: A character string indicating how much of the resampled summary metrics should be saved. Values can be “final”, “all” or “none”
p: For leave-group out cross-validation: the training percentage
index: a list with elements for each external resampling iteration. Each list element is the sample rows used for training at that iteration.
timingSamps: the number of training set samples that will be used to measure the time for predicting samples (zero indicates that the prediction time should not be estimated).
seeds: an optional set of integers that will be used to set the seed at each resampling iteration. This is useful when the models are run in parallel. A value of NA will stop the seed from being set within the worker processes while a value of NULL will set the seeds using a random set of integers. Alternatively, a list can be used. The list should have B+1 elements where B is the number of resamples. The first B elements of the list should be vectors of integers of length P where P is the number of subsets being evaluated (including the full set). The last element of the list only needs to be a single integer (for the final model). See the Examples section below.
allowParallel: if a parallel backend is loaded and available, should the function use it?

Details

More details on this function can be found at [http://caret.r-forge.r-project.org/featureselection.html](http://caret.r-forge.r-project.org/featureselection.html).

Backwards selection requires function to be specified for some operations.

The fit function builds the model based on the current data set. The arguments for the function must be:

- x the current training set of predictor data with the appropriate subset of variables
- y the current outcome data (either a numeric or factor vector)
- first a single logical value for whether the current predictor set has all possible variables
- last similar to first, but TRUE when the last model is fit with the final subset size and predictors.
rfeControl

- ... optional arguments to pass to the fit function in the call to rfe

The function should return a model object that can be used to generate predictions.

The pred function returns a vector of predictions (numeric or factors) from the current model. The arguments are:

- object the model generated by the fit function
- x the current set of predictor set for the held-back samples

The rank function is used to return the predictors in the order of the most important to the least important. Inputs are:

- object the model generated by the fit function
- x the current set of predictor set for the training samples
- y the current training outcomes

The function should return a data frame with a column called var that has the current variable names. The first row should be the most important predictor etc. Other columns can be included in the output and will be returned in the final rfe object.

The selectSize function determines the optimal number of predictors based on the resampling output. Inputs for the function are:

- x a matrix with columns for the performance metrics and the number of variables, called "Variables"
- metric a character string of the performance measure to optimize (e.g. "RMSE", "Rsquared", "Accuracy" or "Kappa")
- maximize a single logical for whether the metric should be maximized

This function should return an integer corresponding to the optimal subset size. caret comes with two examples functions for this purpose: pickSizeBest and pickSizeTolerance.

After the optimal subset size is determined, the selectVar function will be used to calculate the best rankings for each variable across all the resampling iterations. Inputs for the function are:

- y a list of variables importance for each resampling iteration and each subset size (generated by the user-defined rank function). In the example, each each of the cross-validation groups the output of the rank function is saved for each of the subset sizes (including the original subset). If the rankings are not recomputed at each iteration, the values will be the same within each cross-validation iteration.
- size the integer returned by the selectSize function

This function should return a character string of predictor names (of length size) in the order of most important to least important

Examples of these functions are included in the package: lmfuncs, rffuncs, treebagFuncs and nbfuncs.

Model details about these functions, including examples, are at http://caret.r-forge.r-project.org/featureselection.html.
**Value**

A list

**Author(s)**

Max Kuhn

**See Also**

rfe, lmFuncs, rFuncs, treebagFuncs, nbFuncs, pickSizeBest, pickSizeTolerance

**Examples**

```r
## Not run:
subsetSizes <- c(2, 4, 6, 8)
set.seed(123)
seeds <- vector(mode = "list", length = 51)
for(i in 1:50) seeds[[i]] <- sample.int(1000, length(subsetSizes) + 1)
seeds[[51]] <- sample.int(1000, 1)

set.seed(1)
rfMod <- rfe(bbbDescr, logBBB,
  sizes = subsetSizes,
  rfeControl = rfeControl(functions = rffuncs,
    seeds = seeds,
    number = 50))

## End(Not run)
```

---

**sbf**

*Selection By Filtering (SBF)*

**Description**

Model fitting after applying univariate filters

**Usage**

```r
sbf(x, ...)

## Default S3 method:
sbf(x, y, sbfControl = sbfControl(), ...)

## S3 method for class 'formula'
sbf(form, data, ..., subset, na.action, contrasts = NULL)

## S3 method for class 'sbf'
predict(object, newdata = NULL, ...)
```
Arguments

x a data frame containing training data where samples are in rows and features are in columns.

y a numeric or factor vector containing the outcome for each sample.

form A formula of the form y \sim x_1 + x_2 + \ldots

data Data frame from which variables specified in formula are preferentially to be taken.

subset An index vector specifying the cases to be used in the training sample. (NOTE: If given, this argument must be named.)

na.action A function to specify the action to be taken if NAs are found. The default action is for the procedure to fail. An alternative is na.omit, which leads to rejection of cases with missing values on any required variable. (NOTE: If given, this argument must be named.)

contrasts a list of contrasts to be used for some or all of the factors appearing as variables in the model formula.

sbfControl a list of values that define how this function acts. See sbfControl. (NOTE: If given, this argument must be named.)

object an object of class sbf

newdata a matrix or data frame of predictors. The object must have non-null column names

... for sbf: arguments passed to the classification or regression routine (such as randomForest). For predict.sbf: augment cannot be passed to the prediction function using predict.sbf as it uses the function originally specified for prediction.

Details

More details on this function can be found at http://caret.r-forge.r-project.org/featureselection.html.

This function can be used to get resampling estimates for models when simple, filter-based feature selection is applied to the training data.

For each iteration of resampling, the predictor variables are univariately filtered prior to modeling. Performance of this approach is estimated using resampling. The same filter and model are then applied to the entire training set and the final model (and final features) are saved.

sbf can be used with "explicit parallelism", where different resamples (e.g. cross-validation group) can be split up and run on multiple machines or processors. By default, sbf will use a single processor on the host machine. As of version 4.99 of this package, the framework used for parallel processing uses the foreach package. To run the resamples in parallel, the code for sbf does not change; prior to the call to sbf, a parallel backend is registered with foreach (see the examples below).

The modeling and filtering techniques are specified in sbfControl. Example functions are given in ImSBF.
Value

for `sbf`, an object of class `sbf` with elements:

- `pred`: if `sbfControl$saveDetails` is `TRUE`, this is a list of predictions for the hold-out samples at each resampling iteration. Otherwise it is `NULL`.
- `variables`: a list of variable names that survived the filter at each resampling iteration.
- `results`: a data frame of results aggregated over the resamples.
- `fit`: the final model fit with only the filtered variables.
- `optVariables`: the names of the variables that survived the filter using the training set.
- `call`: the function call.
- `control`: the control object.
- `resample`: if `sbfControl$returnResamp` is "all", a data frame of the resampled performance measures. Otherwise, `NULL`.
- `metrics`: a character vector of names of the performance measures.
- `dots`: a list of optional arguments that were passed in.

For `predict.sbf`, a vector of predictions.

Author(s)

Max Kuhn

See Also

- `sbfControl`

Examples

```r
# Not run:
data(BloodBrain)

# Use a GAM is the filter, then fit a random forest model
RFwithGAM <- sbf(bbbDescr, logBBB,
                   sbfControl = sbfControl(functions = rfSBF,
                                            verbose = FALSE,
                                            method = "cv"))
RFwithGAM

predict(RFwithGAM, bbbDescr[1:10,])

# classification example with parallel processing

# library(doMC)

# Note: if the underlying model also uses foreach, the
# number of cores specified above will double (along with
# the memory requirements)
# registerDoMC(cores = 2)
```
sbfControl

Control Object for Selection By Filtering (SBF)

Description

Controls the execution of models with simple filters for feature selection

Usage

sbfControl(functions = NULL,
    method = "boot",
    saveDetails = FALSE,
    number = ifelse(method %in% c("cv", "repeatedcv"), 10, 25),
    repeats = ifelse(method %in% c("cv", "repeatedcv"), 1, number),
    verbose = FALSE,
    returnResamp = "final",
    p = 0.75,
    index = NULL,
    timingSamps = 0,
    seeds = NA,
    allowParallel = TRUE)

Arguments

functions a list of functions for model fitting, prediction and variable filtering (see Details below)

method The external resampling method: boot, cv, LOOCV or LGOCV (for repeated training/test splits)

number Either the number of folds or number of resampling iterations

repeats For repeated k-fold cross-validation only: the number of complete sets of folds to compute

data(mdrr)
mdrrDescr <- mdrrDescr[, -nearZeroVar(mdrrDescr)]
mdrrDescr <- mdrrDescr[, -findCorrelation(cor(mdrrDescr), .8)]

set.seed(1)
filteredNB <- sbf(mdrrDescr, mdrrClass,
    sbfControl = sbfControl(functions = nbSBF,
        verbose = FALSE,
        method = "repeatedcv",
        repeats = 5))

confusionMatrix(filteredNB)

## End(Not run)
saveDetails a logical to save the predictions and variable importances from the selection process
verbose a logical to print a log for each external resampling iteration
returnResamp A character string indicating how much of the resampled summary metrics should be saved. Values can be “final” or “none”
p For leave-group out cross-validation: the training percentage
index a list with elements for each external resampling iteration. Each list element is the sample rows used for training at that iteration.
timingSamps the number of training set samples that will be used to measure the time for predicting samples (zero indicates that the prediction time should not be estimated).
seeds an optional set of integers that will be used to set the seed at each resampling iteration. This is useful when the models are run in parallel. A value of NA will stop the seed from being set within the worker processes while a value of NULL will set the seeds using a random set of integers. Alternatively, a vector of integers can be used. The vector should have B+1 elements where B is the number of resamples. See the Examples section below.
allowParallel if a parallel backend is loaded and available, should the function use it?

Details

More details on this function can be found at http://caret.r-forge.r-project.org/featureselection.html.

Simple filter-based feature selection requires function to be specified for some operations.
The fit function builds the model based on the current data set. The arguments for the function must be:
- x the current training set of predictor data with the appropriate subset of variables (i.e. after filtering)
- y the current outcome data (either a numeric or factor vector)
- ... optional arguments to pass to the fit function in the call to sbf

The function should return a model object that can be used to generate predictions.
The pred function returns a vector of predictions (numeric or factors) from the current model. The arguments are:
- object the model generated by the fit function
- x the current set of predictor set for the held-back samples

The score function is used to return a vector of scores with names for each predictor (such as a p-value). Inputs are:
- x the predictors for the training samples
- y the current training outcomes

The function should return a vector, as previously stated. Examples are give by anovaScores for classification and gamScores for regression.
The filter function is used to return a logical vector with names for each predictor (TRUE indicates that the prediction should be retained). Inputs are:
• score the output of the score function
• x the predictors for the training samples
• y the current training outcomes

The function should return a named logical vector.
Examples of these functions are included in the package: `caretSBF`, `lmSBF`, `rfSBF`, `treebagSBF`, `ldaSBF` and `nbSBF`.

The web page http://caret.r-forge.r-project.org/ has more details and examples related to this function.

Value

a list that echos the specified arguments

Author(s)

Max Kuhn

See Also

`sbf`, `caretSBF`, `lmSBF`, `rfSBF`, `treebagSBF`, `ldaSBF` and `nbSBF`

Examples

```r
## Not run:
data(BloodBrain)

## Use a GAM is the filter, then fit a random forest model
RFwithGAM <- sbf(bbbDescr, logBBB,
                 sbfControl = sbfControl(functions = rfSBF,
                 verbose = FALSE,
                 seeds = sample.int(100000, 11),
                 method = "cv"))
RFwithGAM

## End(Not run)
```

---

**segmentationData**  
*Cell Body Segmentation*

**Description**

Hill, LaPan, Li and Haney (2007) develop models to predict which cells in a high content screen were well segmented. The data consists of 119 imaging measurements on 2019. The original analysis used 1009 for training and 1010 as a test set (see the column called `case`).

The outcome class is contained in a factor variable called `class` with levels "PS" for poorly segmented and "WS" for well segmented.
The raw data used in the paper can be found at the Biomedcentral website. Versions of caret < 4.98 contained the original data. The version now contained in segmentationData is modified. First, several discrete versions of some of the predictors (with the suffix "Status") were removed. Second, there are several skewed predictors with minimum values of zero (that would benefit from some transformation, such as the log). A constant value of 1 was added to these fields: AvgIntenCh2, FiberAlign2Ch3, FiberAlign2Ch4, SpotFiberCountCh4 and TotalIntenCh2.

A binary version of the original data is at http://caret.r-forge.r-project.org/segmentationOriginal.RData.

Usage

data(segmentationData)

Value

segmentationData

data frame of cells

Source


---

**sensitivity**  
*Calculate sensitivity, specificity and predictive values*

Description

These functions calculate the sensitivity, specificity or predictive values of a measurement system compared to a reference results (the truth or a gold standard). The measurement and "truth" data must have the same two possible outcomes and one of the outcomes must be thought of as a "positive" results.

The sensitivity is defined as the proportion of positive results out of the number of samples which were actually positive. When there are no positive results, sensitivity is not defined and a value of *NA* is returned. Similarly, when there are no negative results, specificity is not defined and a value of *NA* is returned. Similar statements are true for predictive values.

The positive predictive value is defined as the percent of predicted positives that are actually positive while the negative predictive value is defined as the percent of negative positives that are actually negative.
Usage

sensitivity(data, ...)  
## Default S3 method:  
sensitivity(data, reference, positive = levels(reference)[1], na.rm = TRUE, ...)  
## S3 method for class 'table'  
sensitivity(data, positive = rownames(data)[1], ...)  
## S3 method for class 'matrix'  
sensitivity(data, positive = rownames(data)[1], ...)

specificity(data, ...)  
## Default S3 method:  
specificity(data, reference, negative = levels(reference)[-1], na.rm = TRUE, ...)  
## S3 method for class 'table'  
specificity(data, negative = rownames(data)[-1], ...)  
## S3 method for class 'matrix'  
specificity(data, negative = rownames(data)[-1], ...)

posPredValue(data, ...)  
## Default S3 method:  
posPredValue(data, reference, positive = levels(reference)[1],  
prevalence = NULL, ...)  
## S3 method for class 'table'  
posPredValue(data, positive = rownames(data)[1], prevalence = NULL, ...)  
## S3 method for class 'matrix'  
posPredValue(data, positive = rownames(data)[1], prevalence = NULL, ...)

negPredValue(data, ...)  
## Default S3 method:  
negPredValue(data, reference, negative = levels(reference)[2],  
prevalence = NULL, ...)  
## S3 method for class 'table'  
negPredValue(data, negative = rownames(data)[-1], prevalence = NULL, ...)  
## S3 method for class 'matrix'  
negPredValue(data, negative = rownames(data)[-1], prevalence = NULL, ...)

Arguments

data for the default functions, a factor containing the discrete measurements. For the 
table or matrix functions, a table or matrix object, respectively.

reference a factor containing the reference values

positive a character string that defines the factor level corresponding to the "positive" results

negative a character string that defines the factor level corresponding to the "negative" results

prevalence a numeric value for the rate of the "positive" class of the data
sensitivity

na.rm a logical value indicating whether NA values should be stripped before the computation proceeds

... not currently used

Details

Suppose a 2x2 table with notation

<table>
<thead>
<tr>
<th>Reference</th>
<th>Predicted Event</th>
<th>No Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>No Event</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

The formulas used here are:

\[
Sensitivity = \frac{A}{A + C} \\
Specificity = \frac{D}{B + D} \\
Prevalence = \frac{A + C}{A + B + C + D} \\
PPV = \frac{\text{sensitivity} \times \text{Prevalence}}{\left(\text{sensitivity} \times \text{Prevalence}\right) + \left(1 - \text{specificity}\right) \times \left(1 - \text{Prevalence}\right)} \\
NPV = \frac{\text{specificity} \times (1 - \text{Prevalence})}{\left(1 - \text{sensitivity}\right) \times \text{Prevalence} + \left(1 - \text{Prevalence}\right) \times \text{specificity}}
\]

See the references for discussions of the statistics.

Value

A number between 0 and 1 (or NA).

Author(s)

Max Kuhn

References


See Also

*confusionMatrix*
sensitivity

Examples

```r
## Not run:  
##############################################################################  
## 2 class example

lvs <- c("normal", "abnormal")
truth <- factor(rep(lvs, times = c(86, 258)),
               levels = rev(lvs))
pred <- factor(
   c(
     rep(lvs, times = c(54, 32)),
     rep(lvs, times = c(27, 231))),
   levels = rev(lvs))

xtab <- table(pred, truth)

sensitivity(pred, truth)
sensitivity(xtab)
posPredValue(pred, truth)
posPredValue(pred, truth, prevalence = 0.25)

specificity(pred, truth)
negPredValue(pred, truth)
negPredValue(xtab)
negPredValue(pred, truth, prevalence = 0.25)

prev <- seq(0.001, .99, length = 20)
npvVals <- ppvVals <- prev * NA
for(i in seq(along = prev))
  {
    ppvVals[i] <- posPredValue(pred, truth, prevalence = prev[i])
    npvVals[i] <- negPredValue(pred, truth, prevalence = prev[i])
  }

plot(prev, ppvVals,
     ylim = c(0, 1),
     type = "l",
     ylab = "",
     xlab = "Prevalence (i.e. prior)"
points(prev, npvVals, type = "l", col = "red")
abline(h=sensitivity(pred, truth), lty = 2)
abline(h=specificity(pred, truth), lty = 2, col = "red")
legend(.5, .5,
       c("ppv", "npv", "sens", "spec"),
       col = c("black", "red", "black", "red"),
       lty = c(1, 1, 2, 2))

##############################################################################  
## 3 class example

library(MASS)
```
spatialSign

Compute the multivariate spatial sign

Description

Compute the spatial sign (a projection of a data vector to a unit length circle). The spatial sign of a vector \( \mathbf{w} \) is \( \mathbf{w} / \text{norm}(\mathbf{w}) \).

Usage

## Default S3 method:
spatialSign(x)
## S3 method for class 'matrix'
spatialSign(x)
## S3 method for class 'data.frame'
spatialSign(x)

Arguments

- `x` an object full of numeric data (which should probably be scaled). Factors are not allowed. This could be a vector, matrix or data frame.

Value

A vector, matrix or data frame with the same dim names of the original data.

Author(s)

Max Kuhn

---

```r
fit <- lda(Species ~ ., data = iris)
model <- predict(fit)$class
irisTabs <- table(model, iris$Species)

## When passing factors, an error occurs with more than two levels
sensitivity(model, iris$Species)

## When passing a table, more than two levels can be used
sensitivity(irisTabs, "versicolor")
specificity(irisTabs, c("setosa", "virginica"))

## End(Not run)
```
**summary.bagEarth**

**Summarize a bagged earth or FDA fit**

**Description**

The function shows a summary of the results from a bagged earth model.

**Usage**

```r
## S3 method for class 'bagEarth'
summary(object, ...)

## S3 method for class 'bagFDA'
summary(object, ...)
```

**Arguments**

- `object`:
  - an object of class "bagEarth" or "bagFDA"
- `...`:
  - optional arguments (not used)

**Details**

The out-of-bag statistics are summarized, as well as the distribution of the number of model terms and number of variables used across all of the bootstrap samples.

**References**


**Examples**

```r
spatialSign(rnorm(5))
spatialSign(matrix(rnorm(12), ncol = 3))
# should fail since the fifth column is a factor
try(spatialSign(iris), silent = TRUE)
spatialSign(iris[,-5])

trellis.par.set(caretTheme())
featurePlot(iris[,-5], iris[,5], "pairs")
featurePlot(spatialSign(scale(iris[,-5])), iris[,5], "pairs")
```
Value

- a list with elements
  - modelInfo: a matrix with the number of model terms and variables used
  - oobStat: a summary of the out-of-bag statistics
  - bmarsCall: the original call to bagEarth

Author(s)

Max Kuhn

Examples

```r
## Not run:
data(trees)
fit <- bagEarth(trees[, -3], trees[3])
summary(fit)
## End(Not run)
```

Description

"These data are recorded on a Tecator Infratec Food and Feed Analyzer working in the wavelength range 850 - 1050 nm by the Near Infrared Transmission (NIT) principle. Each sample contains finely chopped pure meat with different moisture, fat and protein contents. If results from these data are used in a publication we want you to mention the instrument and company name (Tecator) in the publication. In addition, please send a preprint of your article to Karin Thente, Tecator AB, Box 70, S-263 21 Hoganas, Sweden

The data are available in the public domain with no responsibility from the original data source. The data can be redistributed as long as this permission note is attached."

"For each meat sample the data consists of a 100 channel spectrum of absorbances and the contents of moisture (water), fat and protein. The absorbance is \(-\log_{10}\) of the transmittance measured by the spectrometer. The three contents, measured in percent, are determined by analytic chemistry."

Included here are the training, monitoring and test sets.

Usage

data(tecator)

Value

- absorp: absorbance data for 215 samples. The first 129 were originally used as a training set
- endpoints: the percentages of water, fat and protein
Examples

data(tecator)

splom(~endpoints)

# plot 10 random spectra
set.seed(1)
inSubset <- sample(1:dim(endpoints)[1], 10)

absorpSubset <- absorp[inSubset,
endpointSubset <- endpoints[inSubset, 3]

newOrder <- order(absorpSubset[,1])
absorpSubset <- absorpSubset[newOrder,
endpointSubset <- endpointSubset[newOrder]

plotColors <- rainbow(10)

plot(absorpSubset[1,],
   type = "n",
   ylim = range(absorpSubset),
   xlim = c(0, 105),
   xlab = "Wavelength Index",
   ylab = "Absorption")

for(i in 1:10)
{
   points(absorpSubset[i,], type = "l", col = plotColors[i], lwd = 2)
   text(105, absorpSubset[i,100], endpointSubset[i, 1:100], col = plotColors[i])
} 

title("Predictor Profiles for 10 Random Samples")

train

Fit Predictive Models over Different Tuning Parameters

Description

This function sets up a grid of tuning parameters for a number of classification and regression routines, fits each model and calculates a resampling based performance measure.

Usage

train(x, ...)

## Default S3 method:
train(x, y,
   method = "rf",
   preProcess = NULL,
   ...,
weights = NULL,
metric = ifelse(is.factor(y), "Accuracy", "RMSE"),
maximize = ifelse(metric == "RMSE", FALSE, TRUE),
trControl = trainControl(),
tuneGrid = NULL,
tuneLength = 3)

## S3 method for class 'formula'
train(form, data, ..., weights, subset, na.action, contrasts = NULL)

Arguments

x a data frame containing training data where samples are in rows and features are in columns.
y a numeric or factor vector containing the outcome for each sample.
form A formula of the form y ~ x1 + x2 + ...
data Data frame from which variables specified in formula are preferentially to be taken.
weights a numeric vector of case weights. This argument will only affect models that allow case weights.
subset An index vector specifying the cases to be used in the training sample. (NOTE: If given, this argument must be named.)
na.action A function to specify the action to be taken if NAs are found. The default action is for the procedure to fail. An alternative is na.omit, which leads to rejection of cases with missing values on any required variable. (NOTE: If given, this argument must be named.)
contrasts a list of contrasts to be used for some or all of the factors appearing as variables in the model formula.
method a string specifying which classification or regression model to use. Possible values are found using names(getModelInfo()). See http://caret.r-forge.r-project.org/bytag.html. A list of functions can also be passed for a custom model function. See http://caret.r-forge.r-project.org/custom_models.html for details.
... arguments passed to the classification or regression routine (such as randomForest). Errors will occur if values for tuning parameters are passed here.
preProcess a string vector that defines an pre-processing of the predictor data. Current possibilities are "BoxCox", "YeoJohnson", "expoTrans", "center", "scale", "range", "knnImpute", "bagImpute", "medianImpute", "pca", "ica" and "spatialSign". The default is no pre-processing. See preProcess and trainControl on the procedures and how to adjust them.
metric a string that specifies what summary metric will be used to select the optimal model. By default, possible values are "RMSE" and "R^2" for regression and "Accuracy" and "Kappa" for classification. If custom performance metrics are used (via the summaryFunction argument in trainControl), the value of metric should match one of the arguments. If it does not, a warning is issued.
and the first metric given by the `summaryFunction` is used. (NOTE: If given, this argument must be named.)

- **maximize**
  
  A logical: should the metric be maximized or minimized?

- **trControl**
  
  A list of values that define how this function acts. See `trainControl` and [http://caret.r-forge.r-project.org/training.html#custom](http://caret.r-forge.r-project.org/training.html#custom). (NOTE: If given, this argument must be named.)

- **tuneGrid**
  
  A data frame with possible tuning values. The columns are named the same as the tuning parameters. Use `getModelInfo` to get a list of tuning parameters for each model or see [http://caret.r-forge.r-project.org/modellist.html](http://caret.r-forge.r-project.org/modellist.html). (NOTE: If given, this argument must be named.)

- **tuneLength**
  
  An integer denoting the number of levels for each tuning parameters that should be generated by `train`. (NOTE: If given, this argument must be named.)

**Details**

`train` can be used to tune models by picking the complexity parameters that are associated with the optimal resampling statistics. For particular model, a grid of parameters (if any) is created and the model is trained on slightly different data for each candidate combination of tuning parameters. Across each data set, the performance of held-out samples is calculated and the mean and standard deviation is summarized for each combination. The combination with the optimal resampling statistic is chosen as the final model and the entire training set is used to fit a final model.

More details on this function can be found at [http://caret.r-forge.r-project.org/training.html](http://caret.r-forge.r-project.org/training.html).

A variety of models are currently available and are enumerated by tag (i.e. their model characteristics) at [http://caret.r-forge.r-project.org/bytag.html](http://caret.r-forge.r-project.org/bytag.html).

**Value**

A list is returned of class `train` containing:

- **method**
  
  The chosen model.

- **modelType**
  
  An identifier of the model type.

- **results**
  
  A data frame the training error rate and values of the tuning parameters.

- **bestTune**
  
  A data frame with the final parameters.

- **call**
  
  The (matched) function call with dots expanded.

- **dots**
  
  A list containing any ... values passed to the original call.

- **metric**
  
  A string that specifies what summary metric will be used to select the optimal model.

- **control**
  
  The list of control parameters.

- **preProcess**
  
  Either `NULL` or an object of class `preProcess`.

- **finalModel**
  
  An fit object using the best parameters.

- **trainingData**
  
  A data frame.
resample

A data frame with columns for each performance metric. Each row corresponds to each resample. If leave-one-out cross-validation or out-of-bag estimation methods are requested, this will be NULL. The returnResamp argument of trainControl controls how much of the resampled results are saved.

perfNames

a character vector of performance metrics that are produced by the summary function

maximize

a logical recycled from the function arguments.

ylimits

the range of the training set outcomes.

times

a list of execution times: everything is for the entire call to train, final for the final model fit and, optionally, prediction for the time to predict new samples (see trainControl)

Author(s)

Max Kuhn (the guts of train.formula were based on Ripley’s nnet.formula)

References

http://caret.r-forge.r-project.org/training.html

Kuhn (2008), “Building Predictive Models in R Using the caret” (http://www.jstatsoft.org/v28/i05/)

See Also

trainControl, update.train, modelLookup, createFolds

Examples

## Not run:

###################################################################
## Classification Example

data(iris)
TrainData <- iris[,1:4]
TrainClasses <- iris[,5]

knnFit1 <- train(TrainData, TrainClasses,
    method = "knn",
    preProcess = c("center", "scale"),
    tuneLength = 10,
    trControl = trainControl(method = "cv"))

knnFit2 <- train(TrainData, TrainClasses,
    method = "knn",
    preProcess = c("center", "scale"),
    tuneLength = 10,
    trControl = trainControl(method = "boot"))
library(MASS)
nnetFit <- train(TrainData, TrainClasses, 
    method = "nnet", 
    preProcess = "range", 
    tuneLength = 2, 
    trace = FALSE, 
    maxit = 100)

library(mlbench)
data(BostonHousing)

lmFit <- train(medv ~ . + rm:lmstat, 
    data = BostonHousing, 
    "lm")

library(rpart)
rpartFit <- train(medv ~ ., 
    data = BostonHousing, 
    "rpart", 
    tuneLength = 9)

library(party)
earthFit <- train(medv ~ ., 
    data = BostonHousing, 
    "earth", 
    tuneGrid = marsGrid, 
    metric = "MAD", 
    maximize = FALSE, 
    trControl = robustControl)

library(earth)
earthFit <- train(medv ~ ., 
    data = BostonHousing, 
    "earth", 
    tuneGrid = marsGrid, 
    metric = "MAD", 
    maximize = FALSE, 
    trControl = robustControl)
## trainControl

Control parameters for train

### Description

Control the computational nuances of the `train` function

### Usage

```r
trainControl(method = "boot",
             number = ifelse(method %in% c("cv", "repeatedcv"), 10, 25),
             repeats = ifelse(method %in% c("cv", "repeatedcv"), 1, number),
             p = 0.75,
             initialWindow = NULL,
             horizon = 1,
             fixedWindow = TRUE,
             verboseIter = FALSE,
             returnData = TRUE,
             returnResamp = "final",
             savePredictions = FALSE,
             classProbs = FALSE,
             summaryFunction = defaultSummary,
             selectionFunction = "best",
             preProcOptions = list(thresh = 0.95, ICAcomp = 3, k = 5),
             index = NULL,
             indexOut = NULL,
             timingSamps = 0,
             predictionBounds = rep(FALSE, 2),
             seeds = NA,
             allowParallel = TRUE)
```
### Arguments

**method**
- The resampling method: `boot`, `boot632`, `cv`, `repeatedcv`, `LOOCV`, `LGOCV` (for repeated training/test splits), `none` (only fits one model to the entire training set) or `oob` (only for random forest, bagged trees, bagged earth, bagged flexible discriminant analysis, or conditional tree forest models)

**number**
- Either the number of folds or number of resampling iterations

**repeats**
- For repeated k-fold cross-validation only: the number of complete sets of folds to compute

**verboseIter**
- A logical for printing a training log.

**returnData**
- A logical for saving the data

**returnResamp**
- A character string indicating how much of the resampled summary metrics should be saved. Values can be “final”, “all” or “none”

**savePredictions**
- A logical to save the hold-out predictions for each resample

**p**
- For leave-group out cross-validation: the training percentage

**initialWindow**, **horizon**, **fixedWindow**
- possible arguments to `createTimeSlices`

**classProbs**
- A logical; should class probabilities be computed for classification models (along with predicted values) in each resample?

**summaryFunction**
- a function to compute performance metrics across resamples. The arguments to the function should be the same as those in `defaultSummary`

**selectionFunction**
- the function used to select the optimal tuning parameter. This can be a name of the function or the function itself. See `best` for details and other options.

**preProcOptions**
- A list of options to pass to `preProcess`. The type of pre-processing (e.g. center, scaling etc) is passed in via the `preProc` option in `train`.

**index**
- a list with elements for each resampling iteration. Each list element is the sample rows used for training at that iteration.

**indexOut**
- a list (the same length as `index`) that dictates which sample are held-out for each resample. If `NULL`, then the unique set of samples not contained in `index` is used.

**timingSamps**
- the number of training set samples that will be used to measure the time for predicting samples (zero indicates that the prediction time should not be estimated).

**predictionBounds**
- a logical or numeric vector of length 2 (regression only). If logical, the predictions can be constrained to be within the limit of the training set outcomes. For example, a value of `c(TRUE, FALSE)` would only constrain the lower end of predictions. If numeric, specific bounds can be used. For example, if `c(10, NA)`, values below 10 would be predicted as 10 (with no constraint in the upper side).

**seeds**
- an optional set of integers that will be used to set the seed at each resampling iteration. This is useful when the models are run in parallel. A value of `NA` will stop the seed from being set within the worker processes while a value of `NULL` will set the seeds using a random set of integers. Alternatively, a list can be used.
trainControl

The list should have B+1 elements where B is the number of resamples. The first B elements of the list should be vectors of integers of length M where M is the number of models being evaluated. The last element of the list only needs to be a single integer (for the final model). See the Examples section below and the Details section.

allowParallel if a parallel backend is loaded and available, should the function use it?

Details

When setting the seeds manually, the number of models being evaluated is required. This may not be obvious as train does some optimizations for certain models. For example, when tuning over PLS model, the only model that is fit is the one with the largest number of components. So if the model is being tuned over \texttt{comp in 1:10}, the only model fit is \texttt{ncomp = 10}. However, if the vector of integers used in the seeds arguments is longer than actually needed, no error is thrown.

Using method = "none" and specifying model than one model in \texttt{train}'s tuneGrid or tuneLength arguments will result in an error.

Value

An echo of the parameters specified

Author(s)

Max Kuhn

Examples

```r
## Not run:

## Do 5 repeats of 10-Fold CV for the iris data. We will fit
## a KNN model that evaluates 12 values of k and set the seed
## at each iteration.

set.seed(123)
seeds <- vector(mode = "list", length = 51)
for(i in 1:50) seeds[[i]] <- sample.int(1000, 22)

## For the last model:
seeds[[51]] <- sample.int(1000, 1)

ctrl <- trainControl(method = "repeatedcv",
                      repeats = 5,
                      seeds = seeds)

set.seed(1)
mod <- train(Species ~ ., data = iris,
             method = "knn",
             tuneLength = 12,
             trControl = ctrl)

## End(Not run)
```
twoClassSim

**Two-Class Simulations**

**Description**

This function simulates data with two classes, truly important predictors and irrelevant predictions.

**Usage**

```r
twoClassSim(n = 100, intercept = -5,
linearVars = 10, noiseVars = 0, corrVars = 0,
corrType = "AR1", corrValue = 0, mislabel = 0)
```

**Arguments**

- `n`: The number of simulated data points
- `intercept`: The intercept, which controls the class balance. The default value produces a roughly balanced data set when the other defaults are used.
- `linearVars`: The number of linearly important effects. See Details below.
- `noiseVars`: The number of uncorrelated irrelevant predictors to be included.
- `corrVars`: The number of correlated irrelevant predictors to be included.
- `corrType`: The correlation structure of the correlated irrelevant predictors. Values of "AR1" and "exch" are available (see Details below)
- `corrValue`: The correlation value.
- `mislabel`: The proportion of data that is possibly mislabeled. See Details below.

**Details**

The data are simulated in sets. First, two multivariate normal predictors (denoted here as A and B) are created with a correlation our about 0.65. They change the log-odds using main effects and an interaction:

\[ \text{intercept} - 4A + 4B + 2AB \]

The intercept is a parameter for the simulation and can be used to control the amount of class imbalance.

The second set of effects are linear with coefficients that alternate signs and have values between 2.5 and 0.025. For example, if there were six predictors in this set, their contribution to the log-odds would be

\[ -2.50C + 2.05D -1.60E + 1.15F -0.70G + 0.25H \]

The third set is a nonlinear function of a single predictor ranging between [0, 1] called J here:

\[ (J^3) + 2\exp(-6(J-0.3)^2) \]
The fourth set of informative predictors are copied from one of Friedman’s systems and use two more predictors (K and L):

\[ 2\sin(KL) \]

All of these effects are added up to model the log-odds. This is used to calculate the probability of a sample being in the first class and a random uniform number is used to actually make the assignment of the actual class. To mislabel the data, the probability is reversed (i.e. \( p = 1 - p \)) before the random number generation.

The user can also add non-informative predictors to the data. These are random standard normal predictors and can be optionally added to the data in two ways: a specified number of independent predictors or a set number of predictors that follow a particular correlation structure. The only two correlation structure that have been implemented are:

- **compound-symmetry** (aka exchangeable) where there is a constant correlation between all the predictors
- **auto-regressive 1** [AR(1)]. While there is no time component to these data, this structure can be used to add predictors of varying levels of correlation. For example, if there were 4 predictors and \( r \) was the correlation parameter, the between predictor correlation matrix would be

\[
\begin{pmatrix}
1 & \text{sym} & & \\
1 & 1 & & \\
1 & r & 1 & \\
1 & r^2 & r & 1 \\
1 & r^3 & r^2 & r \\
1 & r^4 & r^3 & r^2 & 1
\end{pmatrix}
\]

**Value**

A data frame with columns:

- **Class**: A factor with levels "Class1" and "Class2"
- TwoFactor1, TwoFactor2: Correlated multivariate normal predictors (denoted as A and B above)
- Linear1, ...: Optional uncorrelated standard normal predictors (C through H above)
- Noise1, ...: Optional uncorrelated standard normal predictions
- Corr1, ...: Optional correlated multivariate normal predictors (each with unit variances)

**Author(s)**

Max Kuhn

**Examples**

```r
example <- twoClassSim(100, linearVars = 1)
splom(~example[, 1:6], groups = example$Class)
```
Description

update allows a user to over-ride the tuning parameter selection process by specifying a set of tuning parameters or to update the model object to the latest version of this package.

Usage

## S3 method for class 'train'
update(object, param = NULL, ...)

Arguments

- **object**: an object of class `train`
- **param**: a data frame or named list of all tuning parameters
- **...**: not currently used

Details

If the model object was created with version 5.17-7 or earlier, the underlying package structure was different. To make old `train` objects consistent with the new structure, use `param = NULL` to get the same object back with updates.

To update the model parameters, the training data must be stored in the model object (see the option `returnData` in `trainControl`). Also, all tuning parameters must be specified in the `param` slot. All other options are held constant, including the original pre-processing (if any), options passed in using code... and so on. When printing, the verbiage "The tuning parameter was set manually." is used to describe how the tuning parameters were created.

Value

- a new `train` object

Author(s)

Max Kuhn

See Also

- `train`, `trainControl`
### Examples

```r
## Not run:
data(iris)
TrainData <- iris[,1:4]
TrainClasses <- iris[,5]

knnFit1 <- train(TrainData, TrainClasses,
    method = "knn",
    preProcess = c("center", "scale"),
    tuneLength = 10,
    trControl = trainControl(method = "cv"))

update(knnFit1, list(.k = 3))

## End(Not run)
```

---

**varImp**

*Calculation of variable importance for regression and classification models*

---

### Description

A generic method for calculating variable importance for objects produced by `train` and method specific methods.

### Usage

```r
## S3 method for class 'train'
varImp(object, useModel = TRUE, nonpara = TRUE, scale = TRUE, ...)

## S3 method for class 'earth'
varImp(object, value = "gcv", ...)

## S3 method for class 'fda'
varImp(object, value = "gcv", ...)

## S3 method for class 'rpart'
varImp(object, surrogates = FALSE, competes = TRUE, ...)

## S3 method for class 'randomForest'
varImp(object, ...)

## S3 method for class 'gbm'
varImp(object, numTrees, ...)

## S3 method for class 'classbagg'
varImp(object, ...)
```
## varImp

### S3 method for class 'regbagg'
```
varImp(object, ...)
```

### S3 method for class 'pamrtrained'
```
varImp(object, threshold, data, ...)
```

### S3 method for class 'lm'
```
varImp(object, ...)
```

### S3 method for class 'mvr'
```
varImp(object, estimate = NULL, ...)
```

### S3 method for class 'bagEarth'
```
varImp(object, ...)
```

### S3 method for class 'bagFDA'
```
varImp(object, ...)
```

### S3 method for class 'RandomForest'
```
varImp(object, ...)
```

### S3 method for class 'rfe'
```
varImp(object, drop = FALSE, ...)
```

### S3 method for class 'dsa'
```
varImp(object, cuts = NULL, ...)
```

### S3 method for class 'multinom'
```
varImp(object, ...)
```

### S3 method for class 'cubist'
```
varImp(object, weights = c(0.5, 0.5), ...)
```

### S3 method for class 'JRip'
```
varImp(object, ...)
```

### S3 method for class 'PART'
```
varImp(object, ...)
```

### S3 method for class 'C5.0'
```
varImp(object, ...)
```

### S3 method for class 'nnet'
```
varImp(object, ...)
```

### S3 method for class 'glmnet'
```
varImp(object, lambda = NULL, ...)
```
## S3 method for class 'plsda'

```r
varImp(object, ...)
```

### Arguments

- **object**: an object corresponding to a fitted model
- **useModel**: use a model based technique for measuring variable importance? This is only used for some models (lm, pls, rf, rpart, gbm, pam and mars)
- **nonpara**: should nonparametric methods be used to assess the relationship between the features and response (only used with useModel = FALSE and only passed to filterVarImp).
- **scale**: should the importance values be scaled to 0 and 100?
- **...**: parameters to pass to the specific varImp methods
- **numTrees**: the number of iterations (trees) to use in a boosted tree model
- **threshold**: the shrinkage threshold (pamr models only)
- **data**: the training set predictors (pamr models only)
- **value**: the statistic that will be used to calculate importance: either gcv, nsubsets, or rss
- **surrogates**: should surrogate splits contribute to the importance calculation?
- **competes**: should competing splits contribute to the importance calculation?
- **estimate**: which estimate of performance should be used? See mvrVal
- **drop**: a logical: should variables not included in the final set be calculated?
- **cuts**: the number of rule sets to use in the model (for partDSA only)
- **weights**: a numeric vector of length two that weighs the usage of variables in the rule conditions and the usage in the linear models (see details below).
- **lambda**: a single value of the penalty parameter

### Details

For models that do not have corresponding varImp methods, see filterVarImp.

Otherwise:

- **Linear Models**: the absolute value of the t-statistic for each model parameter is used.
- **Random Forest**: varImp.randomForest and varImp.RandomForest are wrappers around the importance functions from the randomForest and party packages, respectively.
- **Partial Least Squares**: the variable importance measure here is based on weighted sums of the absolute regression coefficients. The weights are a function of the reduction of the sums of squares across the number of PLS components and are computed separately for each outcome. Therefore, the contribution of the coefficients are weighted proportionally to the reduction in the sums of squares.
- **Recursive Partitioning**: The reduction in the loss function (e.g. mean squared error) attributed to each variable at each split is tabulated and the sum is returned. Also, since there may be candidate variables that are important but are not used in a split, the top competing variables are also tabulated at each split. This can be turned off using the maxcompete argument in rpart.control. This
method does not currently provide class–specific measures of importance when the response is a factor.

**Bagged Trees**: The same methodology as a single tree is applied to all bootstrapped trees and the total importance is returned

**Boosted Trees**: \texttt{varImp} is a wrapper around the function from that package (see the \texttt{gbm} package vignette)

**Multivariate Adaptive Regression Splines**: MARS models include a backwards elimination feature selection routine that looks at reductions in the generalized cross-validation (GCV) estimate of error. The \texttt{varImp} function tracks the changes in model statistics, such as the GCV, for each predictor and accumulates the reduction in the statistic when each predictor's feature is added to the model. This total reduction is used as the variable importance measure. If a predictor was never used in any of the MARS basis functions in the final model (after pruning), it has an importance value of zero. Prior to June 2008, the package used an internal function for these calculations. Currently, the \texttt{varImp} is a wrapper to the \texttt{evimp} function in the \texttt{earth} package. There are three statistics that can be used to estimate variable importance in MARS models. Using \texttt{varImp} (\texttt{object, value = "gcv"}) tracks the reduction in the generalized cross-validation statistic as terms are added. However, there are some cases when terms are retained in the model that result in an increase in GCV. Negative variable importance values for MARS are set to zero. Alternatively, using \texttt{varImp} (\texttt{object, value = "rss"}) monitors the change in the residual sums of squares (RSS) as terms are added, which will never be negative. Also, the option \texttt{varImp} (\texttt{object, value = "nsubsets"}), which counts the number of subsets where the variable is used (in the final, pruned model).

**Nearest shrunken centroids**: The difference between the class centroids and the overall centroid is used to measure the variable influence (see \texttt{pamr\_predict}). The larger the difference between the class centroid and the overall center of the data, the larger the separation between the classes. The training set predictions must be supplied when an object of class \texttt{pamr\_trained} is given to \texttt{varImp}.

**Cubist**: The Cubist output contains variable usage statistics. It gives the percentage of times where each variable was used in a condition and/or a linear model. Note that this output will probably be inconsistent with the rules shown in the output from \texttt{summary.cubist}. At each split of the tree, Cubist saves a linear model (after feature selection) that is allowed to have terms for each variable used in the current split or any split above it. Quinlan (1992) discusses a smoothing algorithm where each model prediction is a linear combination of the parent and child model along the tree. As such, the final prediction is a function of all the linear models from the initial node to the terminal node. The percentages shown in the Cubist output reflects all the models involved in prediction (as opposed to the terminal models shown in the output). The variable importance used here is a linear combination of the usage in the rule conditions and the model.

**PART and JRip**: For these rule-based models, the importance for a predictor is simply the number of rules that involve the predictor.

**C5.0**: C5.0 measures predictor importance by determining the percentage of training set samples that fall into all the terminal nodes after the split. For example, the predictor in the first split automatically has an importance measurement of 100 percent since all samples are affected by this split. Other predictors may be used frequently in splits, but if the terminal nodes cover only a handful of training set samples, the importance scores may be close to zero. The same strategy is applied to rule-based models and boosted versions of the model. The underlying function can also return the number of times each predictor was involved in a split by using the option \texttt{metric = "usage"}.

**Neural Networks**: The method used here is based on Gevrey et al (2003), which uses combinations
of the absolute values of the weights. For classification models, the class-specific importances will be the same.

**Recursive Feature Elimination:** Variable importance is computed using the ranking method used for feature selection. For the final subset size, the importances for the models across all resamples are averaged to compute an overall value.

**Feature Selection via Univariate Filters,** the percentage of resamples that a predictor was selected is determined. In other words, an importance of 0.50 means that the predictor survived the filter in half of the resamples.

**Value**

A data frame with class c("varImp.train", "data.frame") for varImp.train or a matrix for other models.

**Author(s)**

Max Kuhn

**References**


---

**xyplot.resamples**  
Lattice Functions for Visualizing Resampling Results

**Description**

Lattice functions for visualizing resampling results across models

**Usage**

```r
## S3 method for class 'resamples'
xyplot(x, data = NULL, what = "scatter", models = NULL,
       metric = x$metric[,], units = "min", ...)

## S3 method for class 'resamples'
dotplot(x, data = NULL, models = x$models,
        metric = x$metric, conf.level = 0.95, ...)

## S3 method for class 'resamples'
densityplot(x, data = NULL, models = x$models, metric = x$metric, ...)
```
Arguments

x an object generated by resamples

data Not used

models a character string for which models to plot. Note: xyplot requires one or two models whereas the other methods can plot more than two.

metric a character string for which metrics to use as conditioning variables in the plot. splom requires exactly one metric when variables = "models" and at least two when variables = "metrics".

variables either "models" or "metrics"; which variable should be treated as the scatter plot variables?

panelRange a common range for the panels. If NULL, the panel ranges are derived from the values across all the models

what for xyplot, the type of plot. Valid options are: "scatter" (for a plot of the resampled results between two models), "BlandAltman" (a Bland-Altman, aka MA plot between two models), "tTime" (for the total time to run train versus the metric), "mTime" (for the time to build the final model) or "pTime" (the time to predict samples - see the timingSamps options in trainControl, rfeControl, or sbfControl)

units either "sec", "min" or "hour"; which what is either "tTime", "mTime" or "pTime", how should the timings be scaled?

conf.level the confidence level for intervals about the mean (obtained using t.test)

... further arguments to pass to either histogram, densityplot, xyplot, dotplot or splom

Details


dotplot plots the average performance value (with two-sided confidence limits) for each model and metric.

densityplot and bwplot display univariate visualizations of the resampling distributions while splom shows the pair-wise relationships.

Value

a lattice object
Author(s)

Max Kuhn

References


See Also

resamples, dotplot, bwplot, densityplot, xyplot, splom

Examples

```r
## Not run:
# load(url("http://caret.r-forge.r-project.org/exampleModels.RData"))

resamps <- resamples(list(CART = rpartFit,
                          CondInfTree = ctreeFit,
                          MARS = earthFit))

dotplot(resamps,
        scales = list(x = list(relation = "free")),
        between = list(x = 2))

bwplot(resamps,
       metric = "RMSE")

densityplot(resamps,
            auto.key = list(columns = 3),
            pch = "|

xyplot(resamps,
       models = c("CART", "MARS"),
       metric = "RMSE")

splom(resamps, metric = "RMSE")
splom(resamps, variables = "metrics")

parallelplot(resamps, metric = "RMSE")

## End(Not run)
```
Index

*Topic **datasets**
  BloodBrain, 13
  cars, 22
  cox2, 29
  dhfr, 31
  GermanCredit, 46
  mdr, 59
  oil, 68
  pottery, 87
  segmentationData, 117
  tecator, 124
*Topic **graphs**
  panelNneedle, 72
*Topic **hplot**
  calibration, 16
  dotPlot, 34
  dotplot.diff.resamples, 35
  featurePlot, 40
  histogram.train, 47
  lattice.rfe, 53
  lift, 55
  panel.lift2, 71
  plot.rfe, 75
  plot.train, 76
  plot.varImp.train, 78
  plotClassProbs, 79
  plotObsVsPred, 80
  prcomp.resamples, 87
  resampleHist, 101
  xyplot.resamples, 140
*Topic **manip**
  Alternate Affy Gene Expression
    Summary Methods, 3
    classDist, 22
    findCorrelation, 43
    findLinearCombos, 44
    normalize.AffyBatch.normalize2Reference, 64
    oneSE, 68
  predict.train, 92
  sensitivity, 118
  spatialSign, 122
  summary.bagEarth, 123
*Topic **models**
  bag.default, 7
  caretFuncs, 18
  caretSBF, 20
  diff.resamples, 32
  dummyVars, 37
  filterVarImp, 42
  format.bagEarth, 45
  normalize2Reference, 65
  nullModel, 66
  plsda, 82
  predictors, 94
  resamples, 102
  rfe, 105
  sbf, 112
  train, 125
  twoClassSim, 133
  update.train, 135
  varImp, 136
*Topic **multivariate**
  icr.formula, 48
  knn3, 50
  knnreg, 52
  predict.knn3, 91
  predict.knnreg, 91
*Topic **neural**
  avNNet.default, 6
  pcaNNet.default, 73
*Topic **print**
  print.train, 99
*Topic **regression**
  bagEarth, 10
  bagFDA, 12
  predict.bagEarth, 89
*Topic **utilities**

143
INDEX

as.table.confusionMatrix, 4
BoxCoxTrans.default, 14
confusionMatrix, 24
confusionMatrix.train, 27
createDataPartition, 29
downSample, 36
maxDissim, 57
modelLookup, 60
nearZeroVar, 62
postResample, 85
preProcess, 96
print.confusionMatrix, 99
resampleSummary, 104
rfeControl, 109
sbfControl, 115
trainControl, 130

absorp(tecator), 124
Alternate Affy Gene Expression
    Summary Methods, 3
anneal, 43, 44
anovaScores, 116
anovaScores(caretSBF), 20
as.matrix.confusionMatrix, 26
as.matrix.confusionMatrix
    (as.table.confusionMatrix), 4
as.table.confusionMatrix, 4, 26
avNNNet (avNNNet.default), 6
avNNNet.default, 6

bag (bag.default), 7
bag.default, 7
bagControl (bag.default), 7
bagEarth, 10, 45, 90, 95
bagFDA, 12, 95
bagging, 95
barchart, 88, 89
bbbDescr (BloodBrain), 13
best, 131
best (oneSE), 68
binom.test, 23, 26
BloodBrain, 13
boxcox, 15, 98
BoxCoxTrans, 97, 98
BoxCoxTrans (BoxCoxTrans.default), 14
BoxCoxTrans.default, 14
bwplot, 36, 142
bwplot.diff.resamples, 34

bwplot.diff.resamples
    (dotplot.diff.resamples), 35
bwplot.resamples, 103
bwplot.resamples (xyplot.resamples), 140
calibration, 16
caretFuncs, 18
caretSbf, 20, 117
cars, 22
cforest, 95
checkConditionalX (nearZeroVar), 62
checkInstall (modelLookup), 60
checkResamples (nearZeroVar), 62
classDist, 22
classify (prcomp.resamples), 87
classDist, 22
classify (prcomp.resamples), 87
classificationMatrix, 5, 24, 28, 99, 120
classificationMatrix.rfe
    (classificationMatrix.train), 27
classificationMatrix.sbf
    (classificationMatrix.train), 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
classificationMatrix.train, 27
contr.dummy (dummyVars), 37
contr.ltrf (dummyVars), 37
contr.treatment, 38, 39
contrasts, 38, 40
cox2, 29
cox2Class (cox2), 29
cox2Descr (cox2), 29
cox2IC50 (cox2), 29
createDataPartition, 29
createFolds, 128
createFolds (createDataPartition), 29
createMultiFolds (createDataPartition), 29
createResample (createDataPartition), 29
createTimeSlices, 131
createTimeSlices (createDataPartition), 29
ctree, 95
ctreeBag (bag.default), 7
defaultSummary, 131
defaultSummary (postResample), 85
densityplot, 35, 36, 47, 48, 54, 79, 101, 141, 142
densityplot.diff.resamples, 34
densityplot.diff.resamples
    (dotplot.diff.resamples), 35
densityplot.resamples, 103
INDEX

densityplot.resamples (xyplot.resamples), 140
densityplot.rfe (lattice.rfe), 53
densityplot.train, 101
densityplot.train (histogram.train), 47
dhfr, 31
diff.resamples, 32, 35, 36, 103
dist, 58
dotPlot, 34
dotplot, 34, 35, 73, 78, 81, 141, 142
dotplot.diff.resamples, 33, 34, 35
dotplot.resamples (xyplot.resamples), 140
downSample, 36
dummyVars, 37
earth, 11, 46, 90, 95
endpoints (tecator), 124
evimp, 139
expoTrans, 98
expoTrans (BoxCoxTrans.default), 14
extractPrediction, 80, 81
extractPrediction (predict.train), 92
extractProb, 79
extractProb (predict.train), 92

fastICA, 49, 50, 97, 98
fattyAcids (oil), 68
fda, 13, 90, 95
featurePlot, 40
filterVarImp, 42
findCorrelation, 43
findLinearCombas, 44, 44
format.bagEarth, 45
format.earth, 45
formula, 40

gamFuncs (caretFuncs), 18
gamScores, 116
gamScores (caretSBF), 20
generateExprVal.method.trimMean (Alternate Affy Gene Expression Summary Methods.), 3
genetic, 43, 44
GermanCredit, 46
getModelInfo, 127
getModelInfo (modelLookup), 60
getTrainPerf (postResample), 85
ggplot, 76, 77
ggplot.rfe (plot.rfe), 75
ggplot.train (plot.train), 76
grepl, 61
hclust, 88, 89
histogram, 47, 48, 54, 79, 101, 141
histogram.rfe (lattice.rfe), 53
histogram.train, 47, 101
icr (icr.formula), 48
icr.formula, 48
install.packages, 61
ipredbagg, 95
ipredknn, 51, 53
knn, 51, 53, 92
knn3, 50, 91
knn3Train (knn3), 50
knnreg, 52, 92
knnregTrain (knnreg), 52

lattice.options, 17, 56
lattice.rfe, 53
ldaBag (bag.default), 7
ldafuncs (caretFuncs), 18
ldaSBF, 117
ldaSBF (caretSBF), 20
leaps, 43, 44
levelplot, 35, 77
levelplot.diff.resamples, 34
levelplot.diff.resamples (dotplot.diff.resamples), 35
lift, 55, 71
lm, 42, 49, 50
lmFuncs, 111, 112
lmFuncs (caretFuncs), 18
lmSBF, 113, 117
lmSBF (caretSBF), 20
loess, 42
logBBB (BloodBrain), 13
lrfuncs (caretFuncs), 18

mahalanobis, 24
maxDissim, 57
mcnemar.test, 25
mdrr, 59
mdrrClass (mdrr), 59
mdrrDescr (mdrr), 59
minDiss (maxDissim), 57
model.matrix, 38, 40
modelCor (resamples), 102
modelLookup, 60, 128
mvrVal, 138

NaiveBayes, 83
nbBag (bag.default), 7
nbFuncs, 111, 112
nbFuncs (caretFuncs), 18
nbSBF, 117
nbSBF (caretSBF), 20
nearZeroVar, 62
negPredValue, 26
negPredValue (sensitivity), 118
nnet, 6, 7, 74, 95
nnetBag (bag.default), 7
normalize.AffyBatch.normalize2Reference, 64
normalize2Reference, 65
nullModel, 66

oil, 68
oilType (oil), 68
oneSE, 68
optim, 15

p.adjust, 33
pamr.train, 95
panel. calibration, 17
panel. calibration (calibration), 16
panel.dotplot, 73
panel.lift (panel.lift2), 71
panel.lift2, 56, 71
panel.needle, 72, 78
panel.xyplot, 71
parallelplot.resamples
(parallelplot.resamples), 140
pcaNNet (pcaNNet.default), 73
pcaNNet.default, 73
pickSizeBest, 111, 112
pickSizeBest (caretFuncs), 18
pickSizeTolerance, 111, 112
pickSizeTolerance (caretFuncs), 18
pickVars (caretFuncs), 18
plot.pcomp.resamples
(plot.pcomp.resamples), 87
plot.rfe, 75
plot.train, 76
plot.varImp.train, 78
plotClassProbs, 79, 93, 94
plotObsVsPred, 80, 93, 94
plsBag (bag.default), 7
plsda, 82
plsr, 83
posPredValue, 26
posPredValue (sensitivity), 118
postResample, 85, 105
pottery, 87
potteryClass (pottery), 87
prcomp, 23, 88, 98
pcomp.resamples, 87
predict, 91, 92
predict.avNNet (avNNet.default), 6
predict.bag (bag.default), 7
predict.bagEarth, 11, 89
predict.bagFDA, 13
predict.bagFDA (predict.bagEarth), 89
predict.BoxCoxTrans
(BoxCoxTrans.default), 14
predict.classDist (classDist), 22
predict.dummyVars (dummyVars), 37
predictexpoTrans
(BoxCoxTrans.default), 14
predict.icr (icr.formula), 48
predict.ipredknn, 91, 92
predict.knn3, 51, 91
predict.knnreg, 53, 91
predict.list (predict.train), 92
predict.nullModel (nullModel), 66
predict.pcaNNet (pcaNNet.default), 73
predict.plsda (plsda), 82
predict.preProcess (preProcess), 96
predict.rfe (rfe), 105
predict.sbf (sbf), 112
predict.splsda (plsda), 82
predict.train, 92
predictors, 94
preProcess, 7, 15, 49, 50, 74, 96, 126, 127, 131
print.bagEarth (bagEarth), 10
print.bagFDA (bagFDA), 12
print.confusionMatrix, 26, 99
print.train, 99
ProbeSet, 3

R2 (postResample), 85
randomForest, 95, 113, 126
resampleHist, 101
INDEX

resamples, 34, 36, 87–89, 102, 142
resampleSummary, 104
rfe, 20, 27, 28, 54, 73, 76, 102, 105, 112
rfeControl, 19, 20, 54, 106, 107, 109, 141
rfeIter (rfe), 105
rFFuncs, 111, 112
rFFuncs (caretFuncs), 18
rfSBF, 117
rfSBF (caretSBF), 20
RMSE (postResample), 85
rpart, 95
sbf, 21, 27, 28, 102, 112, 117
sbfControl, 21, 113, 114, 115, 141
segmentationData, 117
sensitivity, 26, 118
sort.resamples (resamples), 102
spatialSign, 98, 122
specificity, 26
specificity (sensitivity), 118
splom, 36, 88, 89, 141, 142
splom.resamples, 103
splom.resamples (xyplot.resamples), 140
spls, 83
splsda (plsdas), 82
striplot, 47, 48, 54, 77
striplot.rfe (lattice.rfe), 53
striplot.train, 101
striplot.train (histogram.train), 47
sumDiss (maxDissim), 57
summary.bagEarth, 123
summary.bagFDA (summary.bagEarth), 123
summary.cubist, 139
summary.diff.resamples (diff.resamples), 32
summary.gam, 21
summary.resamples (resamples), 102
superpc.train, 95
svmBag (bag.default), 7
t.test, 141
table, 24
tecator, 124
terms.formula, 39
tolerance (oneSE), 68
train, 9, 27, 28, 47, 48, 60, 61, 69, 70, 76, 77, 85, 86, 93, 99–103, 125, 127, 130–132, 135
trainControl, 28, 47, 48, 69, 70, 85, 86, 93, 94, 101, 103, 126–128, 130, 135, 141
treebagFuncs, 111, 112
treebagFuncs (caretFuncs), 18
treebagSBF, 117
treebagSBF (caretSBF), 20
trellis.par.set, 17, 18, 56, 57, 71
trim.matrix, 44, 45
twoClassSim, 133
twoClassSummary (postResample), 85
update.train, 128, 135
update.trellis, 56
upSample (downSample), 36
varImp, 34, 136
xyplot, 16–18, 36, 47, 48, 54–57, 71, 75–77, 81, 88, 89, 141, 142
xyplot.calibration (calibration), 16
xyplot.lift (lift), 55
xyplot.resamples, 103, 140
xyplot.rfe (lattice.rfe), 53
xyplot.train, 101
xyplot.train (histogram.train), 47