Package ‘caret’

November 20, 2018

Version 6.0-81

Title Classification and Regression Training

Author Max Kuhn. Contributions from Jed Wing, Steve Weston, Andre Williams, Chris Keefer, Allan Engelhardt, Tony Cooper, Zachary Mayer, Brenton Kenkel, the R Core Team, Michael Benesty, Reynald Lescarbeau, Andrew Ziem, Luca Scrucia, Yuan Tang, Can Candan, and Tyler Hunt.

Description Misc functions for training and plotting classification and regression models.

Maintainer Max Kuhn <mxkuhn@gmail.com>

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

URL https://github.com/topepo/caret/

BugReports https://github.com/topepo/caret/issues

Imports foreach, methods, plyr, ModelMetrics (>= 1.1.0), nlme, reshape2, stats, stats4, utils, grDevices, recipes (>= 0.1.4), withr (>= 2.0.0)

Suggests BradleyTerry2, e1071, earth (>= 2.2-3), fastICA, gam (>= 1.15), ipred, kernlab, knitr, klaR, MASS, ellipse, mda, mgcv, mlbench, MLmetrics, nnet, party (>= 0.9-99992), pls, pROC, proxy, randomForest, RANN, spls, subselect, pamr, superpc, Cubist, testthat (>= 0.9.1), rpart, dplyr

License GPL (>= 2)

RoxygenNote 6.1.0.9000

VignetteBuilder knitr

NeedsCompilation yes

Repository CRAN

Date/Publication 2018-11-20 10:10:14 UTC
### R topics documented:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>as.matrix.confusionMatrix</td>
<td>4</td>
</tr>
<tr>
<td>avNNet</td>
<td>5</td>
</tr>
<tr>
<td>bag</td>
<td>7</td>
</tr>
<tr>
<td>bagEarth</td>
<td>10</td>
</tr>
<tr>
<td>bagFDA</td>
<td>12</td>
</tr>
<tr>
<td>BloodBrain</td>
<td>14</td>
</tr>
<tr>
<td>BoxCoxTrans</td>
<td>14</td>
</tr>
<tr>
<td>calibration</td>
<td>16</td>
</tr>
<tr>
<td>caretSBF</td>
<td>19</td>
</tr>
<tr>
<td>cars</td>
<td>20</td>
</tr>
<tr>
<td>classDist</td>
<td>20</td>
</tr>
<tr>
<td>confusionMatrix</td>
<td>22</td>
</tr>
<tr>
<td>confusionMatrix.train</td>
<td>25</td>
</tr>
<tr>
<td>cox2</td>
<td>27</td>
</tr>
<tr>
<td>createDataPartition</td>
<td>27</td>
</tr>
<tr>
<td>defaultSummary</td>
<td>30</td>
</tr>
<tr>
<td>densityplot.rfe</td>
<td>32</td>
</tr>
<tr>
<td>dhfr</td>
<td>34</td>
</tr>
<tr>
<td>diff.resamples</td>
<td>34</td>
</tr>
<tr>
<td>dotPlot</td>
<td>37</td>
</tr>
<tr>
<td>dotplot.diff.resamples</td>
<td>38</td>
</tr>
<tr>
<td>downSample</td>
<td>39</td>
</tr>
<tr>
<td>dummyVars</td>
<td>40</td>
</tr>
<tr>
<td>extractPrediction</td>
<td>43</td>
</tr>
<tr>
<td>featurePlot</td>
<td>46</td>
</tr>
<tr>
<td>filterVarImp</td>
<td>47</td>
</tr>
<tr>
<td>findCorrelation</td>
<td>48</td>
</tr>
<tr>
<td>findLinearCombos</td>
<td>50</td>
</tr>
<tr>
<td>format.bagEarth</td>
<td>51</td>
</tr>
<tr>
<td>gafs.default</td>
<td>52</td>
</tr>
<tr>
<td>gafsControl</td>
<td>55</td>
</tr>
<tr>
<td>gafs_initial</td>
<td>58</td>
</tr>
<tr>
<td>GermanCredit</td>
<td>60</td>
</tr>
<tr>
<td>getSamplingInfo</td>
<td>60</td>
</tr>
<tr>
<td>ggplot.rfe</td>
<td>61</td>
</tr>
<tr>
<td>ggplot.train</td>
<td>62</td>
</tr>
<tr>
<td>histogram.train</td>
<td>64</td>
</tr>
<tr>
<td>icr.formula</td>
<td>66</td>
</tr>
<tr>
<td>index2vec</td>
<td>68</td>
</tr>
<tr>
<td>knn3</td>
<td>68</td>
</tr>
<tr>
<td>knnreg</td>
<td>70</td>
</tr>
<tr>
<td>learning_curve_dat</td>
<td>72</td>
</tr>
<tr>
<td>lift</td>
<td>74</td>
</tr>
<tr>
<td>maxDissim</td>
<td>76</td>
</tr>
<tr>
<td>mdrd</td>
<td>79</td>
</tr>
<tr>
<td>modelLookup</td>
<td>80</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------</td>
</tr>
<tr>
<td>nearZeroVar</td>
<td>81</td>
</tr>
<tr>
<td>negPredValue</td>
<td>83</td>
</tr>
<tr>
<td>nullModel</td>
<td>87</td>
</tr>
<tr>
<td>oil</td>
<td>89</td>
</tr>
<tr>
<td>oneSE</td>
<td>89</td>
</tr>
<tr>
<td>panel.lift2</td>
<td>91</td>
</tr>
<tr>
<td>panel.needle</td>
<td>93</td>
</tr>
<tr>
<td>pcaNNet</td>
<td>94</td>
</tr>
<tr>
<td>pickSizeBest</td>
<td>96</td>
</tr>
<tr>
<td>plot.gafs</td>
<td>98</td>
</tr>
<tr>
<td>plot.varImp.train</td>
<td>99</td>
</tr>
<tr>
<td>plotClassProbs</td>
<td>100</td>
</tr>
<tr>
<td>plotObsVsPred</td>
<td>102</td>
</tr>
<tr>
<td>plsda</td>
<td>104</td>
</tr>
<tr>
<td>pottery</td>
<td>106</td>
</tr>
<tr>
<td>prcomp.resamples</td>
<td>107</td>
</tr>
<tr>
<td>predict.bagEarth</td>
<td>109</td>
</tr>
<tr>
<td>predict.gafs</td>
<td>110</td>
</tr>
<tr>
<td>predict.knn3</td>
<td>111</td>
</tr>
<tr>
<td>predict.knnreg</td>
<td>112</td>
</tr>
<tr>
<td>predictors</td>
<td>113</td>
</tr>
<tr>
<td>preProcess</td>
<td>113</td>
</tr>
<tr>
<td>print.confusionMatrix</td>
<td>117</td>
</tr>
<tr>
<td>print.train</td>
<td>118</td>
</tr>
<tr>
<td>recall</td>
<td>119</td>
</tr>
<tr>
<td>resampleHist</td>
<td>122</td>
</tr>
<tr>
<td>resamples</td>
<td>123</td>
</tr>
<tr>
<td>resampleSummary</td>
<td>125</td>
</tr>
<tr>
<td>rfe</td>
<td>126</td>
</tr>
<tr>
<td>rfeControl</td>
<td>130</td>
</tr>
<tr>
<td>Sacramento</td>
<td>133</td>
</tr>
<tr>
<td>safs</td>
<td>134</td>
</tr>
<tr>
<td>safs_initial</td>
<td>137</td>
</tr>
<tr>
<td>sbf</td>
<td>139</td>
</tr>
<tr>
<td>sbfControl</td>
<td>142</td>
</tr>
<tr>
<td>scat</td>
<td>145</td>
</tr>
<tr>
<td>segmentationData</td>
<td>146</td>
</tr>
<tr>
<td>SLC14_1</td>
<td>146</td>
</tr>
<tr>
<td>spatialSign</td>
<td>150</td>
</tr>
<tr>
<td>summary.bagEarth</td>
<td>151</td>
</tr>
<tr>
<td>tecator</td>
<td>152</td>
</tr>
<tr>
<td>thresholder</td>
<td>153</td>
</tr>
<tr>
<td>train</td>
<td>155</td>
</tr>
<tr>
<td>trainControl</td>
<td>161</td>
</tr>
<tr>
<td>train_model_list</td>
<td>165</td>
</tr>
<tr>
<td>update.safs</td>
<td>197</td>
</tr>
<tr>
<td>update.train</td>
<td>199</td>
</tr>
<tr>
<td>varImp</td>
<td>200</td>
</tr>
</tbody>
</table>
Description
Conversion functions for class confusionMatrix

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

Arguments
- `x` an object of class confusionMatrix
- `what` data to convert to matrix. Either "xtabs", "overall" or "classes"
- `...` not currently used

Details
For as.table, the cross-tabulations are saved. For as.matrix, the three object types are saved in matrix format.

Value
A matrix or table

Author(s)
Max Kuhn

Examples
R class example
```r
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)

---

**avNNet**  
*Neural Networks Using Model Averaging*

**Description**

Aggregate several neural network models

**Usage**

```r
avNNet(x, ...)
```

```r
## S3 method for class 'formula'
avNNet(formula, data, weights, ..., repeats = 5,
        bag = FALSE, allowParallel = TRUE, seeds = sample.int(1e+05,
        repeats), subset, na.action, contrasts = NULL)

## Default S3 method:
avNNet(x, y, repeats = 5, bag = FALSE,
        allowParallel = TRUE, seeds = sample.int(1e+05, repeats), ...)

## S3 method for class 'avNNet'
print(x, ...)

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

**Arguments**

- `x`  
  matrix or data frame of `x` values for examples.
... arguments passed to \texttt{nnet}

\textbf{formula}  A formula of the form \texttt{class ~ x1 + x2 + ...}

\textbf{data}  Data frame from which variables specified in \texttt{formula} are preferentially to be taken.

\textbf{weights}  (case) weights for each example – if missing defaults to 1.

\textbf{repeats}  the number of neural networks with different random number seeds

\textbf{bag}  a logical for bagging for each repeat

\textbf{allowParallel}  if a parallel backend is loaded and available, should the function use it?

\textbf{seeds}  random number seeds that can be set prior to bagging (if done) and network creation. This helps maintain reproducibility when models are run in parallel.

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

\textbf{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 \texttt{na.omit}, which leads to rejection of cases with missing values on any required variable. (NOTE: If given, this argument must be named.)

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

\textbf{y}  matrix or data frame of target values for examples.

\textbf{object}  an object of class \texttt{avNNNet} as returned by \texttt{avNNNet}.

\textbf{newdata}  matrix or data frame of test examples. A vector is considered to be a row vector comprising a single case.

\textbf{type}  Type of output, either: \texttt{raw} for the raw outputs, \texttt{code} for the predicted class or \texttt{prob} for the class probabilities.

\section*{Details}

Following Ripley (1996), the same neural network model is fit using different random number seeds. All 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 \texttt{foreach} package is used to train the networks in parallel.

\section*{Value}

For \texttt{avNNNet}, an object of \texttt{"avNNNet"} or \texttt{"avNNNet.formula"}. Items of interest in \# the output are:

\begin{itemize}
  \item [\textbf{model}] a list of the models generated from \texttt{nnet}
  \item [\textbf{repeats}] an echo of the model input
  \item [\textbf{names}] if any predictors had only one distinct value, this is a character string of the \# remaining columns. Otherwise a value of \texttt{NULL}
\end{itemize}

\section*{Author(s)}

These are heavily based on the \texttt{nnet} code from Brian Ripley.
References


See Also

`nnet`, `preProcess`

Examples

data(BloodBrain)
## Not run:
modelFit <- avNNet(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**

`bag(x, ...)`

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

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

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

## S3 method for class 'bag'
`print(x, ...)`

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

x a matrix or data frame of predictors

arguments to pass to the model function

... 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, 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 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 logical: should out-of-bag statistics be computed and the predictions retained?

allowParallel a parallel backend is loaded and available, should the function use it?

y a vector of outcomes

B the number of bootstrap samples to train over.

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.

bagControl a list of options.

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.

digits minimal number of significant digits.
Format

An object of class list of length 3.

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, nbbBag, 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

## 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))

## An example of pooling posterior probabilities to generate class predictions
data(mdrr)

## remove some zero variance predictors and linear dependencies
Bagged Earth

Description

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

Usage

```r
bagEarth(x, ...)  
```

## Default S3 method:
```r
bagEarth(x, y, weights = NULL, B = 50, 
          summary = mean, keepX = TRUE, ...)  
```

## S3 method for class ‘formula’
```r
bagEarth(formula, data = NULL, B = 50, 
          summary = mean, keepX = TRUE, ..., subset, weights = NULL, 
          na.action = na.omit)  
```

## S3 method for class ‘bagEarth’
```r
print(x, ...)  
```

Arguments

- `x`: matrix or data frame of `x` values for examples.
- `...`: arguments passed to the `earth` function.
- `y`: matrix or data frame of numeric values outcomes.
- `weights`: (case) weights for each example - if missing defaults to 1.
- `B`: the number of bootstrap samples.
- `summary`: a function with a single argument specifying how the bagged predictions should be summarized.
**bagEarth**

KeepX a logical: should the original training data be kept?

**formula**
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 '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.)

**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(x = trees[,-3], y = trees[,3])
set.seed(2189)
fit2 <- bagEarth(x = trees[,-3], y = 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

```r
bagFDA(x, ...)
```

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

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

```
## S3 method for class 'bagFDA'
print(x, ...)
```

Arguments

- `x`: matrix or data frame of `x` values for examples.
- `...`: arguments passed to the `mars` function
- `y`: matrix or data frame of numeric values outcomes.
- `weights`: (case) weights for each example - if missing defaults to 1.
- `B`: the number of bootstrap samples
- `keepX`: a logical: should the original training data be kept?
- `formula`: A formula of the form `y ~ x1 + x2 + ...`
- `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.)

Details

The function computes a FDA model for each bootstrap sample.
Value

A list with elements

- `fit` a list of FDA 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 (bagFDA.formula is based on Ripley’s nnet.formula)

References


See Also

`fda, predict.bagFDA`

Examples

```r
library(mlbench)
library(earth)
data(Glass)

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

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

set.seed(3577)
baggedFit <- bagFDA(Type ~ ., trainData)
confusionMatrix(data = predict(baggedFit, testData[, -10]),
               reference = 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 fame bbbDescr contains the descriptor values.

Value

| bbbDescr | data frame of chemical descriptors |
| logBBB   | vector of assay results            |

Source


BoxCoxTrans  Box-Cox and Exponential Transformations

Description

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

Usage

BoxCoxTrans(y, ...)

## Default S3 method:
BoxCoxTrans(y, x = rep(1, length(y)), fudge = 0.2,
   numUnique = 3, na.rm = FALSE, ...)

## S3 method for class 'BoxCoxTrans'
print(x, newdata, digits = 3, ...)

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

- **y**: a numeric vector of data to be transformed. For BoxCoxTrans, the data must be strictly positive.
- **...**: for BoxCoxTrans: options to pass to `boxcox`. `plotit` should not be passed through. For `predict.BoxCoxTrans`, additional arguments are ignored.
- **x**: an optional dependent variable to be used in a linear model.
- **fudge**: a tolerance value: 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 transformation?
- **na.rm**: a logical value indicating whether NA values should be stripped from `y` and `x` before the computation proceeds.
- **newdata**: a numeric vector of values to transform.
- **digits**: minimal number of significant digits.
- **object**: an object of class `BoxCoxTrans` or `expotrans`.

Details

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

`expotrans` 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 <= 0)` or if `length(unique(y)) < numUnique`, lambda is not estimated and no transformation is applied.

Value

Both functions returns a list of class of either `BoxCoxTrans` or `expotrans` with elements

- **lambda**: estimated transformation value
- **fudge**: value of fudge
- **n**: number of data points used to estimate lambda
- **summary**: the results of `summary(y)`
- **ratio**: `max(y)/min(y)`
- **skewness**: sample skewness statistic

`BoxCoxTrans` also returns:

- **fudge**: value of fudge

The `predict` functions returns numeric vectors of transformed values

Author(s)

Max Author
References


See Also

boxcox, preProcess, optim

Examples

data(BloodBrain)

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

predict(bc, ratio[1:5])

tests[5] <- NA
bc2 <- BoxCoxTrans(ratio, bbbDescr$tps, na.rm = TRUE)
bcc2

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

calibration(x, ...)

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

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

## S3 method for class 'calibration'
print(x, ...)

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

## S3 method for class 'calibration'
`ggplot(data, ..., bwidth = 2, dwidth = 3)`

### 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.

- **...**: options to pass through to `xyplot` or the panel function (not used in `calibration.formula`).

- **data**: 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`. For `ggplot.calibration`, `data` should be an object of class "calibration".

- **class**: a character string for the class of interest

- **cuts**: If a single number this indicates 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`. If a vector, these are the actual cuts that will be used.

- **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`

- **bwidth, dwidth**: a numeric value for the confidence interval bar width and dodge width, respectively. In the latter case, a dodge is only used when multiple models are specified in the formula.

### Details

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

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

1. The data are split into `cuts` 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`.

For the `ggplot` method, confidence intervals on the estimated proportions (from `binom.test`) are also shown.

### 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(mdrr)
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])
```
caretSBF

Description
Ancillary functions for univariate feature selection

Usage
caretSBF
anovaScores(x, y)
gamScores(x, y)

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

Format
An object of class list of length 5.

Details
More details on these functions can be found at http://topepo.github.io/caret/feature-selection-using-univariate.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 gamScores 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. gamScores 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

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

Description

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
classDist(x, ...)

## Default 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(.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. If **trans** = NULL will not apply a function

**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

```r
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 Create a confusion matrix

Description

Calculates a cross-tabulation of observed and predicted classes with associated statistics.

Usage

```r
confusionMatrix(data, ...) 
```

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

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

Arguments

data a factor of predicted classes (for the default method) or an object of class table.

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

reference a factor of classes to be used as the true results
**confusionMatrix**

- **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. When `mode = "prec_recall"`, `positive` is the same value used for `relevant` for functions `precision`, `recall`, and `F_meas.table`.

- **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.

- **mode**: a single character string either "sens_spec", "prec_recall", or "everything"

**Details**

The functions requires 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></th>
<th>Event</th>
<th>No Event</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Event</strong></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td><strong>No Event</strong></td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

The formulas used here are:

- \( Sensitivity = \frac{A}{A + C} \)
- \( Specificity = \frac{D}{B + D} \)
- \( Prevalence = \frac{A + C}{A + B + C + D} \)
- \( PPV = \frac{\text{Sensitivity} \times \text{Prevalence}}{(\text{Sensitivity} \times \text{Prevalence}) + ((1 - \text{Specificity}) \times (1 - \text{Prevalence}))} \)
- \( NPV = \frac{\text{Specificity} \times (1 - \text{Prevalence})}{((1 - \text{Sensitivity}) \times \text{Prevalence}) + ((\text{Specificity}) \times (1 - \text{Prevalence}))} \)
- \( DetectionRate = \frac{A}{A + B + C + D} \)
- \( DetectionPrevalence = \frac{A + B}{A + B + C + D} \)
- \( BalancedAccuracy = \frac{(\text{Sensitivity} + \text{Specificity})}{2} \)

- \( Precision = \frac{A}{A + B} \)
- \( Recall = \frac{A}{A + C} \)

- \( F1 = \frac{(1 + \beta^2) \times \text{precision} \times \text{recall}}{(\beta^2 \times \text{precision}) + \text{recall}} \)

where \( \beta = 1 \) for this function.
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, precision, recall, F1, prevalence, detection rate, detection prevalence and balanced accuracy for each class. For two class systems, this is calculated once using the positive argument.

Note

If the reference and data factors have the same levels, but in the incorrect order, the function will reorder them to the order of the data and issue a warning.

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

classification(iris$Species, sample(iris$Species))

newPrior <- c(0.05, 0.8, 0.15)
names(newPrior) <- levels(iris$Species)

classification(iris$Species, sample(iris$Species))
```

---

**confusionMatrix.train**  Estimate a Resampled Confusion Matrix

**Description**

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

**Usage**

```
## S3 method for class 'train'
classification(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.
confusionMatrix.train

- **norm**: A character string indicating how the table entries should be normalized. Valid values are "none", "overall" or "average".

- **dnn**: A character vector of dimension names 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 aggregated counts of samples on each of the cells (across all resamples). For `norm = "average"`, the average number of cell counts across resamples is computed (this can help evaluate how many holdout samples there were on average). The default is `norm = "overall"`, which is equivalent to "average" but in percentages.

**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)
```
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)."

Details

The data are in the Supplemental 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

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 split for series data. groupKFold splits the data based on a grouping factor.
Usage

createDataPartition(y, times = 1, p = 0.5, list = TRUE, 
groups = min(5, length(y)))

createFolds(y, k = 10, list = TRUE, returnTrain = FALSE)

createMultiFolds(y, k = 10, times = 5)

createTimeSlices(y, initialWindow, horizon = 1, fixedWindow = TRUE, 
skip = 0)

groupKFold(group, k = length(unique(group)))

createResample(y, times = 10, list = 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 logical, if FALSE, all training samples start at 1
skip integer, how many (if any) resamples to skip to thin the total amount
group a vector of groups whose length matches the number of rows in the overall data set.

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 (\(\leq 3\)) 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 "Foldi.Repj" 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.

For Group k-fold cross-validation, the data are split such that no group is contained in both the modeling and holdout sets. One or more group could be left out, depending on the value of k.

**Value**

A list or matrix of row position integers corresponding to the training data. For createTimeSlices subsamples are named by the end index of each training subsample.

**Author(s)**

Max Kuhn, createTimeSlices by Tony Cooper

**References**


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

**Examples**

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(21))
createTimeSlices(1:9, 5, 1, fixedWindow = FALSE)
createTimeSlices(1:9, 5, 1, fixedWindow = TRUE)
defaultSummary

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

defaultSummary(data, lev = NULL, model = NULL)

postResample(pred, obs)

twoClassSummary(data, lev = NULL, model = NULL)

mnLogLoss(data, lev = NULL, model = NULL)

multiClassSummary(data, lev = NULL, model = NULL)

prSummary(data, lev = NULL, model = NULL)

Arguments

data a data frame with columns obs and pred for the observed and predicted outcomes. For metrics that rely on class probabilities, such as 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 from the method argument of train.

pred A vector of numeric data (could be a factor)

obs A vector of numeric data (could be a factor)
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 $\sqrt{\text{mean}((\text{pred} - \text{obs})^2 \cdot 2}$.

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",

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

. Mean absolute error is calculated using $\text{mean}(|\text{pred} - \text{obs}|)$.

defaultSummary is the default function to compute performance metrics in train. It is a wrapper around postResample. The first argument is data, which is data.frame with columns named obs and pred for the observed and predicted outcome values (either numeric data for regression or character values for classification). The second argument is lev, a character string that has the outcome factor levels or NULL for a regression model. The third parameter is model, which can be used if a summary metric is specific to a model function. If other columns from the data are required to compute the summary statistics, but should not be used in the model, the recipe method for train can be used.

twoClassSummary computes sensitivity, specificity and the area under the ROC curve. mnlLogLoss computes the minus log-likelihood of the multinomial distribution (without the constant term):

$$-\logLoss = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{C} y_{ij} \log(p_{ij})$$

where the $y$ values are binary indicators for the classes and $p$ are the predicted class probabilities.

prSummary (for precision and recall) computes values for the default 0.50 probability cutoff as well as the area under the precision-recall curve across all cutoffs and is labelled as "AUC" in the output. If assumes that the first level of the factor variables corresponds to a relevant result but the lev argument can be used to change this.

multiClassSummary computes some overall measures of for performance (e.g. overall accuracy and the Kappa statistic) and several averages of statistics calculated from "one-versus-all" configurations. For example, if there are three classes, three sets of sensitivity values are determined and the average is reported with the name ("Mean_Sensitivity"). The same is true for a number of statistics generated by confusionMatrix. With two classes, the basic sensitivity is reported with the name "Sensitivity".

To use twoClassSummary and/or mnlLogLoss, the classProbs argument of trainControl should be TRUE. multiClassSummary can be used without class probabilities but some statistics (e.g. overall log loss and the average of per-class area under the ROC curves) will not be in the result set.

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, Zachary Mayer

References


See Also

trainControl

Examples

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

classes <- c("class1", "class2")
set.seed(1)
dat <- data.frame(obs = factor(sample(classes, 50, replace = TRUE)),
  pred = factor(sample(classes, 50, replace = TRUE)),
  class1 = runif(50))
dat$class2 <- 1 - dat$class1

defaultSummary(dat, lev = classes)
twoClassSummary(dat, lev = classes)
prSummary(dat, lev = classes)
mnLogLoss(dat, lev = classes)

densityplot.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'
densityplot(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,
    method = "cv",
    verbose = FALSE,
    returnResamp = "all")
```
diff.resamples

lmProfile <- rfe(x, y, 
sizes = subsets,
  rfeControl = ctrl)
xyplot(lmProfile)
stripplot(lmProfile)
histogram(lmProfile)
densityplot(lmProfile)
## End(Not run)

---

### 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".

**Details**

The data frame `dhfr` contains a column called `y` with the outcome classification. The remainder of the columns are molecular descriptor values.

**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", ...)

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

compare_models(a, b, metric = a$metric[1])
```

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`
- `confLevel`: confidence level to use for `dotplot.diff.resamples`. See Details below.
- `adjustment`: any p-value adjustment method to pass to `p.adjust`
- `...`: further arguments to pass to `test`
- `object`: a object generated by `diff.resamples`
- `digits`: the number of significant differences to display when printing
- `a`, `b`: two objects of class `train`, `sbf` or `rfe` with a common set of resampling indices in the control object.

Details

The ideas and methods here are based on Hothorn et al. (2005) and Eugster et al. (2008).

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)` here `p` is the number of pair-wise comparisons are being made. For other correction methods, no such change is used.

`compare_models` is a shorthand function to compare two models using a single metric. It returns the results of `t.test` on the differences.

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

...or (for compare_models) an object of class htest resulting from t.test.

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://topepo.github.io/caret/exampleModels.RData"))

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

difs <- diff(resamps)

difs

summary(difs)

compare_models(rpartFit, ctreeFit)

## End(Not run)
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

Grab 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

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

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

knnImp <- varImp(knnFit)
dotPlot(knnImp)
dotplot.diff.resamples

Lattice Functions for Visualizing Resampling Differences

Description

Lattice functions for visualizing resampling result differences between models

Usage

## 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
- `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

## Not run:
#load(url("http://topepo.github.io/caret/exampleModels.RData"))

resamps <- resamples(list(CART = rpartFit,
downSample

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)

downSample

\textit{Down- and Up-Sampling Imbalanced Data}

\textbf{Description}

downSample will randomly sample a data set so that all classes have the same frequency as the minority class. upSample samples with replacement to make the class distributions equal.

\textbf{Usage}

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

\textbf{Arguments}

\begin{itemize}
  \item \texttt{x} \hspace{1cm} a matrix or data frame of predictor variables
  \item \texttt{y} \hspace{1cm} a factor variable with the class memberships
  \item \texttt{list} \hspace{1cm} should the function return \texttt{list(x, y)} or bind \texttt{x} and \texttt{y} together? If \texttt{TRUE}, the output will be coerced to a data frame.
  \item \texttt{yname} \hspace{1cm} if \texttt{list = FALSE}, a label for the class column
\end{itemize}

\textbf{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 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)
```

---

**dummyVars**  
*Create 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'
print(x, ...)

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

ccontr.ltrf(n, contrasts = TRUE, sparse = FALSE)

class2ind(x, drop2nd = FALSE)
Arguments

- **formula**: An appropriate R model formula, see References
- **...**: additional arguments to be passed to other methods
- **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.
- **x**: A factor vector.
- **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.
- **drop2nd**: A logical: if the factor has two levels, should a single binary vector be returned?

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:

```r
(Intercept) dayTue dayWed dayThu dayFri daySat daySun
 1  0  0   0   0   0  0  0
 1  0  0   0   0  0  0  0
 1  0  0   0   0  0  0  0
 1  0  1   0   0   0  0
 1  0  1   0   0   0  0
 1  0  0   0   1   0  0
 1  0  0   0   0   1  0
 1  0  0   0   0   1  0
 1  0  0   0   0   1  0
```

In some situations, there may be a need for dummy variables for all the levels of the factor. For the same example:
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.treatment` as the base function to do this.

`class2ind` is most useful for converting a factor outcome vector to a matrix (or vector) of dummy variables.

### 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
- `lvls`: 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.

- `class2ind` returns a matrix (or a vector if `drop2nd = TRUE`).
- `contr.treatment` generates a design matrix.

### Author(s)

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

### References

[https://cran.r-project.org/doc/manuals/R-intro.html#Formulae-for-statistical-models](https://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", "afternoon", "afternoon"),
                     day = c("Mon", "Mon", "Mon", "Wed", "Wed", "Fri", "Sat", "Sat", "Fri"))
levels(when$time) <- list(morning="morning", afternoon="afternoon", night="night")
levels(when$day) <- list(Mon="Mon", Tue="Tue", Wed="Wed", Thu="Thu", Fri="Fri", Sat="Sat", Sun="Sun")

## Default behavior:
mymodel.matrix(~day, when)

mainEffects <- dummyVars(~ day + time, data = when)
mainEffects
predict(mainEffects, when[1:3,])

when2 <- when
when2[, 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)

head(class2ind(iris$Species))
two_levels <- factor(rep(letters[1:2], each = 5))
class2ind(two_levels)
class2ind(two_levels, drop2ind = TRUE)
```

**extractPrediction**

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

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)

## S3 method for class 'train'

predict(object, newdata = NULL, type = "raw",
na.action = na.omit, ...)

Arguments

models a list of objects of the class train. The objects must have been generated with
fitBest = FALSE and returnData = TRUE.
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
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 but, if the train model used a recipe, an error will occur.
type either "raw" or "prob", for the number/class predictions or class probabilities,
respectively. Class probabilities are not available for all classification models
na.action the method for handling missing data
...
only used for sort and modelCor and captures arguments to pass to sort or
FUN.

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 passes 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 passes to plotObsVsPred or plotClassProbs.
extractPrediction

Value

For `predict.train`, a vector of predictions if `type = "raw"` or a data frame of class probabilities for `type = "prob"`. 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
- `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/article/view/v028i05/v28i05.pdf)

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)

---

**featurePlot**

*Wrapper for Lattice Plotting of Predictor Variables*

**Description**

A shortcut to produce lattice graphs

**Usage**

```r
featurePlot(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)
```
filterVarImp

    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 regres-
sion, the data frame contains one column: "Overall" for the importance values.
findCorrelation

Author(s)

Max Kuhn

Examples

data(mdrr)
filterVarImp(mdrrDescr[, 1:5], mdrrClass)

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)

findCorrelation

Determine highly correlated variables

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

findCorrelation(x, cutoff = 0.9, verbose = FALSE, names = FALSE,
exact = ncol(x) < 100)

Arguments

x  A correlation matrix

cutoff  A numeric value for the pair-wise absolute correlation cutoff

verbose  A boolean for printing the details

names  a logical; should the column names be returned (TRUE) or the column index (FALSE)?

exact  a logical; should the average correlations be recomputed at each step? See Details below.
findCorrelation

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.

Using `exact = TRUE` will cause the function to re-evaluate the average correlations at each step while `exact = FALSE` uses all the correlations regardless of whether they have been eliminated or not. The exact calculations will remove a smaller number of predictors but can be much slower when the problem dimensions are "big".

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

Value

A vector of indices denoting the columns to remove (when `names = TRUE`) otherwise a vector of column names. If no correlations meet the criteria, `integer(0)` is returned.

Author(s)

Original R code by Dong Li, modified by Max Kuhn

See Also

`leaps, genetic, anneal, findLinearCombos`

Examples

```r
R1 <- structure(c(1, 0.86, 0.56, 0.32, 0.85, 0.86, 1, 0.01, 0.74, 0.32,
                  0.56, 0.01, 1, 0.65, 0.91, 0.32, 0.74, 0.65, 1, 0.36,
                  0.85, 0.32, 0.91, 0.36, 1),
                 Dim = c(5L, 5L))
colnames(R1) <- rownames(R1) <- paste0("x", 1:ncol(R1))
R1

findCorrelation(R1, cutoff = .6, exact = FALSE)
findCorrelation(R1, cutoff = .6, exact = TRUE)
findCorrelation(R1, cutoff = .6, exact = TRUE, names = FALSE)

R2 <- diag(rep(1, 5))
R2[2, 3] <- R2[3, 2] <- .7
R2[5, 3] <- R2[3, 5] <- -.7
R2[4, 1] <- R2[1, 4] <- -.67

corrDF <- expand.grid(row = 1:5, col = 1:5)
corrDF$correlation <- as.vector(R2)
levelplot(correlation ~ row + col, corrDF)

findCorrelation(R2, cutoff = .65, verbose = TRUE)
```
findCorrelation(R2, cutoff = .99, verbose = TRUE)

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
format.bagEarth

Examples

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

findLinearCombos(testData)
```

format.bagEarth

Format `bagEarth` objects

Description

Return a string representing the ‘bagEarth’ expression.

Usage

```r
## 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 `cat`.
- `cat` a logical; should the equation be printed?
- `...` Arguments to `format.earth`.

Value

A character representation of the bagged earth object.
See Also
default

Examples

a <- bagEarth(Volume ~ ., data = trees, B = 3)
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)
# - 2.378658 * 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

gafs.default Genetic algorithm feature selection

Description

Supervised feature selection using genetic algorithms

Usage

## Default S3 method:
gafs(x, y, iters = 10, popSize = 50,
pCrossOver = 0.8, pMutation = 0.1, elite = 0, suggestions = NULL,
differences = TRUE, gafsControl = gafsControl(), ...)

Arguments

x an object where samples are in rows and features are in columns. This could be
a simple matrix, data frame or other type (e.g. sparse matrix). See Details below
y a numeric or factor vector containing the outcome for each sample
**gafs** conducts a supervised binary search of the predictor space using a genetic algorithm. See Mitchell (1996) and Scrucca (2013) for more details on genetic algorithms.

This function conducts the search of the feature space repeatedly within resampling iterations. First, the training data are split by whatever resampling method was specified in the control function. For example, if 10-fold cross-validation is selected, the entire genetic algorithm is conducted 10 separate times. For the first fold, nine tenths of the data are used in the search while the remaining tenth is used to estimate the external performance since these data points were not used in the search.

During the genetic algorithm, a measure of fitness is needed to guide the search. This is the internal measure of performance. During the search, the data that are available are the instances selected by the top-level resampling (e.g. the nine tenths mentioned above). A common approach is to conduct another resampling procedure. Another option is to use a holdout set of samples to determine the internal estimate of performance (see the holdout argument of the control function). While this is faster, it is more likely to cause overfitting of the features and should only be used when a large amount of training data are available. Yet another idea is to use a penalized metric (such as the AIC statistic) but this may not exist for some metrics (e.g. the area under the ROC curve).

The internal estimates of performance will eventually overfit the subsets to the data. However, since the external estimate is not used by the search, it is able to make better assessments of overfitting. After resampling, this function determines the optimal number of generations for the GA.

Finally, the entire data set is used in the last execution of the genetic algorithm search and the final model is built on the predictor subset that is associated with the optimal number of generations determined by resampling (although the update function can be used to manually set the number of generations).

This is an example of the output produced when `gafsControl(verbose = TRUE)` is used:

```
Fold2  1 0.715 (13)
Fold2  2 0.715->0.737 (13->17, 30.4%) *
Fold2  3 0.737->0.732 (17->14, 24.0%) 
Fold2  4 0.737->0.769 (17->23, 25.0%) *
```
For the second resample (e.g. fold 2), the best subset across all individuals tested in the first generation contained 13 predictors and was associated with a fitness value of 0.715. The second generation produced a better subset containing 17 samples with an associated fitness values of 0.737 (and improvement is symbolized by the *). The percentage listed is the Jaccard similarity between the previous best individual (with 13 predictors) and the new best. The third generation did not produce a better fitness value but the fourth generation did.

The search algorithm can be parallelized in several places:

1. each externally resampled GA can be run independently (controlled by the allowParallel option of gafsControl)
2. within a GA, the fitness calculations at a particular generation can be run in parallel over the current set of individuals (see the genParallel option in gafsControl)
3. if inner resampling is used, these can be run in parallel (controls depend on the function used. See, for example, trainControl)
4. any parallelization of the individual model fits. This is also specific to the modeling function.

It is probably best to pick one of these areas for parallelization and the first is likely to produces the largest decrease in run-time since it is the least likely to incur multiple re-starting of the worker processes. Keep in mind that if multiple levels of parallelization occur, this can effect the number of workers and the amount of memory required exponentially.

Value

an object of class gafs

Author(s)

Max Kuhn, Luca Scrucca (for GA internals)

References

http://en.wikipedia.org/wiki/Jaccard_index

See Also

gafsControl, predict.gafs, caretGA, rfGA treebagGA

Examples

```r
## Not run:
set.seed(1)
train_data <- twoClassSim(100, noiseVars = 10)
```
### Description

Control the computational nuances of the `gafs` and `safs` functions


The functions component contains the information about how the model should be fit and summarized. It also contains the elements needed for the GA and SA modules (e.g. cross-over, etc).

The elements of functions that are the same for GAs and SAs are:

- `fit`, with arguments `x, y, lev, last, and ...` is used to fit the classification or regression model
- `pred`, with arguments `object` and `x`, predicts new samples
- `fitness_intern`, with arguments `object, x, y, maximize, and p`, summarizes performance for the internal estimates of fitness
- `fitnessExtern`, with arguments `data, lev, and model`, summarizes performance using the externally held-out samples
- `selectIter`, with arguments `x, metric, and maximize`, determines the best search iteration for feature selection.

The elements of functions specific to genetic algorithms are:

- `initial`, with arguments `vars, popSize and ...`, creates an initial population.
- `selection`, with arguments `population, fitness, r, q, and ...`, conducts selection of individuals.
• crossover, with arguments population, fitness, parents and ...., control genetic reproduction.
• mutation, with arguments population, parent and ...., adds mutations.

The elements of functions specific to simulated annealing are:
• initial, with arguments vars, prob, and ...., creates the initial subset.
• perturb, with arguments x, vars, and number, makes incremental changes to the subsets.
• prob, with arguments old, new, and iteration, computes the acceptance probabilities.


holdout can be used to hold out samples for computing the internal fitness value. Note that this is independent of the external resampling step. Suppose 10-fold CV is being used. Within a resampling iteration, holdout can be used to sample an additional proportion of the 90% resampled data to use for estimating fitness. This may not be a good idea unless you have a very large training set and want to avoid an internal resampling procedure to estimate fitness.

The search algorithms can be parallelized in several places:
1. each externally resampled GA or SA can be run independently (controlled by the allowParallel options)
2. within a GA, the fitness calculations at a particular generation can be run in parallel over the current set of individuals (see the genParallel)
3. if inner resampling is used, these can be run in parallel (controls depend on the function used. See, for example, trainControl)
4. any parallelization of the individual model fits. This is also specific to the modeling function.

It is probably best to pick one of these areas for parallelization and the first is likely to produces the largest decrease in run-time since it is the least likely to incur multiple re-starting of the worker processes. Keep in mind that if multiple levels of parallelization occur, this can effect the number of workers and the amount of memory required exponentially.

Usage

gafsControl(functions = NULL, method = "repeatedcv", metric = NULL, maximize = NULL, number = ifelse(grepl("cv", method), 10, 25), repeats = ifelse(grepl("cv", method), 1, 5), verbose = FALSE, returnResamp = "final", p = 0.75, index = NULL, indexOut = NULL, seeds = NULL, holdout = 0, genParallel = FALSE, allowParallel = TRUE)
safsControl(functions = NULL, method = "repeatedcv", metric = NULL, maximize = NULL, number = ifelse(grepl("cv", method), 10, 25), repeats = ifelse(grepl("cv", method), 1, 5), verbose = FALSE, returnResamp = "final", p = 0.75, index = NULL, indexOut = NULL, seeds = NULL, holdout = 0, improve = Inf, allowParallel = TRUE)
Arguments

functions  a list of functions for model fitting, prediction etc (see Details below)
method    The resampling method: boot, boot632, cv, repeatedcv, LOOCV, LGOCV (for repeated training/test splits)
metric    a two-element string that specifies what summary metric will be used to select the optimal number of iterations from the external fitness value and which metric should guide subset selection. If specified, this vector should have names "internal" and "external". See gafs and/or safs for explanations of the difference.
maximize  a two-element logical: should the metrics be maximized or minimized? Like the metric argument, this this vector should have names "internal" and "external".
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
verbose   a logical for printing results
returnResamp A character string indicating how much of the resampled summary metrics should be saved. Values can be “all” or “none”
p        For leave-group out cross-validation: the training percentage
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.
seeds     a vector or integers that can be used to set the seed during each search. The number of seeds must be equal to the number of resamples plus one.
holdout   the proportion of data in [0, 1) to be held-back from x and y to calculate the internal fitness values
genParallel if a parallel backend is loaded and available, should gafs use it tp parallelize the fitness calculations within a generation within a resample?
allowParallel if a parallel backend is loaded and available, should the function use it?
improve   the number of iterations without improvement before safs reverts back to the previous optimal subset

Value
An echo of the parameters specified

Author(s)
Max Kuhn

References

See Also

safs, safs, caretGA, rfGA, treebagGA, caretSA, rfSA, treebagSA

---

**gafs_initial**  
*Ancillary genetic algorithm functions*

**Description**

Built-in functions related to genetic algorithms

These functions are used with the functions argument of the gafsControl function. More information on the details of these functions are at [http://topepo.github.io/caret/feature-selection-using-genetic-algorithms.html](http://topepo.github.io/caret/feature-selection-using-genetic-algorithms.html).

Most of the gafs_* functions are based on those from the GA package by Luca Scrucca. These functions here are small re-writes to work outside of the GA package.

The objects caretGA, rfGA and treebagGA are example lists that can be used with the functions argument of gafsControl.

In the case of caretGA, the ... structure of gafs passes through to the model fitting routine. As a consequence, the train function can easily be accessed by passing important arguments belonging to train to gafs. See the examples below. By default, using caretGA will used the resampled performance estimates produced by train as the internal estimate of fitness.

For rfGA and treebagGA, the randomForest and bagging functions are used directly (i.e. train is not used). Arguments to either of these functions can also be passed to them though the gafs call (see examples below). For these two functions, the internal fitness is estimated using the out-of-bag estimates naturally produced by those functions. While faster, this limits the user to accuracy or Kappa (for classification) and RMSE and R-squared (for regression).

**Usage**

```r
library(caret)
gafs_initial(vars, popSize, ...)
gafs_lrSelection(population, fitness, r = NULL, q = NULL, ...)
gafs_spCrossover(population, fitness, parents, ...)
gafs_raMutation(population, parent, ...)
gafs_nlrSelection(population, fitness, q = 0.25, ...)
gafs_rwSelection(population, fitness, ...)
gafs_tourSelection(population, fitness, k = 3, ...)
gafs_uCrossover(population, parents, ...)
```
Arguments

vars       number of possible predictors
popSize    the population size passed into \texttt{gafs}
...        not currently used
population a binary matrix of the current subsets with predictors in columns and individuals in rows
fitness    a vector of fitness values
r, q, k    tuning parameters for the specific selection operator
parent, parents integer(s) for which chromosomes are altered

Value

The return value depends on the function.

Author(s)

Luca Scrucca, \texttt{gafs\_initial}, \texttt{caretGA}, \texttt{rfGA} and \texttt{treebagGA} by Max Kuhn

References

cran.r-project.org/web/packages/GA/

See Also

\texttt{gafs, gafsControl}

Examples

```r
pop <- gafs\_initial(vars = 10, popSize = 10)
pop

gafs\_lrSelection(population = pop, fitness = 1:10)

gafs\_spCrossover(population = pop, fitness = 1:10, parents = 1:2)
```

```
## Not run:
## Hypothetical examples
lda\_ga <- gafs(x = predictors,
               y = classes,
               gafsControl = gafsControl(functions = caretGA),
               ## now pass arguments to \code{\texttt{\`train\}}
               method = "lda",
```
```r
metric = "Accuracy"
trControl = trainControl(method = "cv", classProbs = TRUE))

rf_ga <- gafs(x = predictors,
              y = classes,
gafsControl = gafsControl(functions = rfGAs),
## these are arguments to `randomForest`
ntree = 1000,
importance = TRUE)

## End(Not run)
```

---

**GermanCredit**

*German Credit Data*

**Description**

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

**Details**

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.

**Source**

UCI Machine Learning Repository

---

**getSamplingInfo**

*Get sampling info from a train model*

**Description**

Placeholder.

**Usage**

```r
getSamplingInfo(method = NULL, regex = TRUE, ...)
```
Arguments

- method: Modeling method.
- regex: Whether to use regex matching.
- ...: additional arguments to passed to grepl.

Details

Placeholder.

Value

A list

---

**ggplot.rfe**

*Plot RFE Performance Profiles*

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'
ggplot(data = NULL, mapping = NULL,
       metric = data$metric[1], output = "layered", ..., 
       environment = NULL)

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

Arguments

- **data**: an object of class `rfe`.
- **mapping**: unused arguments to make consistent with `ggplot2` generic method.
- **metric**: What measure of performance to plot. Examples of possible values are "RMSE", "R squared", "Accuracy" or "Kappa". Other values can be used depending on what metrics have been calculated.
- **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.
- ...: 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.
- **x**: an object of class `rfe`. 
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/article/view/v028i05/v028i05.pdf)

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, logBBB, 
                 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)
```

---

**ggplot.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'
ggplot(data = NULL, mapping = NULL, metric = data$metric[1], plotType = "scatter", output = "layered", nameInStrip = FALSE, highlight = FALSE, ..., environment = NULL)
```

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

Arguments

- **data**: An object of class `train`.
- **mapping, environment**: Unused arguments to make consistent with `ggplot2` generic method.
- **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))
- **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.
- **nameInStrip**: A logical: if there are more than 2 tuning parameters, should the name and value be included in the panel title?
- **highlight**: A logical: if TRUE, a diamond is placed around the optimal parameter setting for models using grid search.
- **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.
- **x**: An object of class `train`.

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 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 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.
histogram.train

Author(s)
Max Kuhn

References
Kuhn (2008), “Building Predictive Models in R Using the caret” (http://www.jstatsoft.org/article/view/v028i05/v28i05.pdf)

See Also
train, levelplot, xyplot, stripplot, ggplot

Examples

## 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)

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

## S3 method for class 'train'

histogram(x, data = NULL, metric = x$metric, ...)

## Not run:
histogram.train

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:

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)
**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 `class ~ x1 + x2 + ...{}`
- `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.
- `x` matrix or data frame of x values for examples.
icr.formula

\begin{itemize}
\item \textbf{y} \hspace{1cm} \text{matrix or data frame of target values for examples.}
\item \textbf{object} \hspace{1cm} \text{an object of class icr as returned by icr.}
\item \textbf{newdata} \hspace{1cm} \text{matrix or data frame of test examples.}
\end{itemize}

\section*{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 \textit{n.comp} to pass to \texttt{fastICA}.

The function \texttt{preProcess} to produce the ICA scores for the original data and for \texttt{newdata}.

\section*{Value}

For \texttt{icr}, a list with elements

\begin{itemize}
\item \textbf{model} \hspace{1cm} \text{the results of \texttt{lm} after the ICA transformation}
\item \textbf{ica} \hspace{1cm} \text{pre-processing information}
\item \textbf{n.comp} \hspace{1cm} \text{number of ICA components}
\item \textbf{names} \hspace{1cm} \text{column names of the original data}
\end{itemize}

\section*{Author(s)}

Max Kuhn

\section*{See Also}

\texttt{fastICA, preProcess, lm}

\section*{Examples}

\begin{verbatim}
data(BloodBrain)

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

icrFit

predict(icrFit, bbbDescr[1:5,])
\end{verbatim}
index2vec  
\textit{Convert indices to a binary vector}

**Description**

The function performs the opposite of which converting a set of integers to a binary vector.

**Usage**

\texttt{index2vec(x, vars, sign = FALSE)}

**Arguments**

- \texttt{x} a vector of integers
- \texttt{vars} the number of possible locations
- \texttt{sign} a logical; when true the data are encoded as -1/+1, and 0/1 otherwise

**Value**

a numeric vector

**Author(s)**

Max Kuhn

**Examples**

\texttt{index2vec(x = 1:2, vars = 5)}
\texttt{index2vec(x = 1:2, vars = 5, sign = TRUE)}

---

**knn3**  
\textit{k-Nearest Neighbour Classification}

**Description**

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

knn3(x, ...)

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

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

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

## S3 method for class 'knn3'
print(x, ...)

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

Arguments

x a matrix of training set predictors

... additional parameters to pass to knn3Train. However, passing prob = FALSE will be over-ridden.

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.

y a factor vector of training set classes

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.

cl 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 knn3Train 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

```r
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)))
kn3Train(train, test, cl, k = 5, prob = TRUE)
```

---

**knnreg**

*k*-Nearest Neighbour Regression

Description

SkS-nearest neighbour regression that can return the average value for the neighbours.

Usage

```r
knnreg(x, ...)

## 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, ...)
```
Arguments

x  a matrix or data frame of training set predictors.
... additional parameters to pass to knnreg.
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.
nb.action function which indicates what should happen when the data contain NAs.
k number of neighbours considered.
y a numeric vector of outcomes.
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.
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.uni-erlangen.de>, modifications by Max Kuhn and Chris Keefer

Examples

data(BloodBrain)
inTrain <- createDataPartition(logBBB, p = .8)[[1]]
```r
trainX <- bbbDescr[inTrain,]
trainY <- logBBB[inTrain]

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

fit <- knnreg(trainX, trainY, k = 3)
plot(testY, predict(fit, testX))
```

### learning_curve_dat

Create Data to Plot a Learning Curve

**Description**

For a given model, this function fits several versions on different sizes of the total training set and returns the results.

**Usage**

```r
learning_curve_dat(dat, outcome = NULL, proportion = (1:10)/10,
                   test_prop = 0, verbose = TRUE, ...)
```

**Arguments**

- `dat`: the training data
- `outcome`: a character string identifying the outcome column name
- `proportion`: the incremental proportions of the training set that are used to fit the model
- `test_prop`: an optional proportion of the data to be used to measure performance.
- `verbose`: a logical to print logs to the screen as models are fit
- `...`: options to pass to `train` to specify the model. These should not include `x`, `y`, `formula`, or `data`. If `trainControl` is used here, do not use `method = "none"`.

**Details**

This function creates a data set that can be used to plot how well the model performs over different sized versions of the training set. For each data set size, the performance metrics are determined and saved. If `test_prop == 0`, the apparent measure of performance (i.e. re-predicting the training set) and the resampled estimate of performance are available. Otherwise, the test set results are also added.

If the model being fit has tuning parameters, the results are based on the optimal settings determined by `train`. 
Value

a data frame with columns for each performance metric calculated by `train` as well as columns:

- **Training_Size**: the number of data points used in the current model fit
- **Data**: which data were used to calculate performance. Values are "Resampling", "Training", and (optionally) "Testing"

In the results, each data set size will have one row for the apparent error rate, one row for the test set results (if used) and as many rows as resamples (e.g. 10 rows if 10-fold CV is used).

Author(s)

Max Kuhn

See Also

`train`

Examples

```r
### Not run:
set.seed(1412)
class_dat <- twoClassSim(1000)

set.seed(29510)
lda_data <- learning_curve_dat(dat = class_dat,
                               outcome = "Class",
                               test_prop = 1/4,
                               ### `train` arguments:
                               method = "lda",
                               metric = "ROC",
                               trControl = trainControl(classProbs = TRUE,
                                                           summaryFunction = twoClassSummary))

ggplot(lda_data, aes(x = Training_Size, y = ROC, color = Data)) +
  geom_smooth(method = loess, span = .8) +
  theme_bw()

### End(Not run)
```
Description

For classification models, this function creates a 'lift plot' that describes how well a model ranks samples for one class.

Usage

```
lift(x, ...)
```

## Default S3 method:
```
lift(x, ...)
```

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

## S3 method for class 'lift'
```
print(x, ...)
```

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

## S3 method for class 'lift'
```
ggplot(data = NULL, mapping = NULL, plot = "gain",
    values = NULL, ..., environment = 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.
- `...` options to pass through to `xyplot` or the panel function (not used in `lift.formula`).
- `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` or `ggplot.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

cuts
If a single value is given, a sequence of values between 0 and 1 are created with length cuts. If a vector, these values are used as the cuts. If NULL, each unique value of the model prediction is used. This is helpful when the data set is large.

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".

mapping, environment
Not used (required for ggplot consistency).

Details

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

To construct data for the 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)).
maxDissim

Value

lift.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
- 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))
```

maxDissim

Maximum Dissimilarity Sampling

Description

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

**Usage**

```r
cmvmaxDissim(a, b, n = 2, obj = minDiss, useNames = FALSE,
nrandomFrac = 1, verbose = FALSE, ...)
```

- `minDiss(u)`
- `sumDiss(u)`

**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 ot 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 with 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)

    ## 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")
```
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 considered 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."

Details

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

Value

<table>
<thead>
<tr>
<th>mdrrDescr</th>
<th>the descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdrrClass</td>
<td>the categorical outcome (&quot;Active&quot; or &quot;Inactive&quot;)</td>
</tr>
</tbody>
</table>

Source

modelLookup

Tools for Models Available in train

Description

These functions show information about models and packages that are accessible via \texttt{train}.

Usage

\begin{verbatim}
modelLookup(model = NULL)

checkInstall(pkg)

getModelInfo(model = NULL, regex = TRUE, ...)
\end{verbatim}

Arguments

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

Details

\texttt{modelLookup} is good for getting information related to the tuning parameters for a model. \texttt{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.

\texttt{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 \texttt{install.packages} using that function's 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

\texttt{modelLookup} produces a data frame with columns

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

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
## Not run:
modelLookup()
modelLookup("gbm")

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

checkInstall(getModelInfo("pls")$library)
## End(Not run)
```

---

**nearZeroVar**

Identification of near zero variance predictors

**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

nearZeroVar(x, freqCut = 95/5, uniqueCut = 10, saveMetrics = FALSE, names = FALSE, foreach = FALSE, allowParallel = TRUE)
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.
names a logical. If false, column indexes are returned. If true, column names are returned.
foreach should the foreach package be used for the computations? If TRUE, less memory should be used.
allowParallel should the parallel processing via the foreach package be used for the computations? If TRUE, more memory will be used but execution time should be shorter.
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.

nzv is the original version of the function.

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
negPredValue

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)
nearZeroVar(bbbDescr, names = TRUE)

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
checkResamples(folds, x, classes)

negPredValue               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.
Usage

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, ...)

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, ...)

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], ...)

Arguments

data | for the default functions, a factor containing the discrete measurements. For the table or matrix functions, a table or matrix object, respectively.

... | not currently used

reference | a factor containing the reference values
negPredValue

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
positive a character string that defines the factor level corresponding to the "positive" results
na.rm a logical value indicating whether NA values should be stripped before the computation proceeds

Details

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.

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:

\[
Sensitivity = \frac{A}{A + C}
\]

\[
Specificity = \frac{D}{B + D}
\]

\[
Prevalence = \frac{A + C}{A + B + C + D}
\]

\[
PPV = \frac{(sensitivity \times Prevalence)}{(sensitivity \times Prevalence) + ((1 - specificity) \times (1 - Prevalence))}
\]

\[
NPV = \frac{(specificity \times (1 - Prevalence))}{(((1 - sensitivity) \times Prevalence) + (specificity \times (1 - Prevalence)))}
\]

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`

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")
```
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, ...)

---

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)

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)

---
nullModel

Arguments

- **x**: An optional matrix or data frame of predictors. These values are not used in the model fit.
- **...**: Optional arguments (not yet used).
- **y**: A numeric vector (for regression) or factor (for classification) of outcomes.
- **object**: An object of class `nullModel`.
- **newdata**: 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

```r
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).

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


oneSE

Selecting tuning Parameters

Description

Various functions for setting tuning parameters

Usage

- `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.
num
maximize
tol

the number of resamples (for oneSE only)
a logical: should the metric be maximized or minimized?
the acceptable percent tolerance (for tolerance only)

Details

These functions can be used by train to select the "optimal" model from 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://topepo.github.io/caret/model-training-and-tuning.html#custom.

By default, train uses best.

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 over fit. 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

a 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. k–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)
```

Description
Two panel functions that be used in conjunction with lift.
Usage

```r
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 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`, `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))

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

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

## use a different panel function
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 = if (is.null(groups))
  dot.symbol$pch else sup.symbol$pch, col = if (is.null(groups))
  dot.symbol$col else sup.symbol$col, lty = dot.line$lty,
  lwd = dot.line$lwd, col.line = dot.line$col, levels.fos = NULL,
  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`
Neural Networks with a Principal Component Step

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

Usage
pcaNNet(x, ...)

## S3 method for class 'formula'
pcaNNet(formula, data, weights, ..., thresh = 0.99,
    subset, na.action, contrasts = NULL)

## Default S3 method:
pcaNNet(x, y, thresh = 0.99, ...)

## S3 method for class 'pcaNNet'
print(x, ...)

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

Arguments
x matrix or data frame of x values for examples.

... arguments passed to nnet, such as size, decay, etc.

formula A formula of the form class ~ x1 + x2 + ...{}

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.

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

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.
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 `pcaNNNet`, an object of "pcaNNNet" or "pcaNNNet.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**

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

predict(modelFit, bbbDescr[, 1:10])
```
pickSizeBest

**Backwards Feature Selection Helper Functions**

**Description**
Ancillary functions for backwards selection

**Usage**

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

CaretFuncs
LdaFuncs
TreeBagFuncs
GamFuncs
RffFuncs
LmFuncs
NbFuncs
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

**Format**
An object of class list of length 6.
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

`rfeControl`, `rfe`

Examples

```r
## 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$RMSE[absoluteBest],
     "RMSE in position",
     absoluteBest, ":n")
cat("Accepting a 1.5 pct loss:",
     example$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")

Description

Plot the performance values versus search iteration

Usage

## S3 method for class 'gafs'
plot(x, metric = x$control$metric["external"],
     estimate = c("internal", "external"), output = "ggplot", ...)

Arguments

x
metric
estimate
output
... options passed to xyplot

Details

The mean (averaged over the resamples) is plotted against the search iteration using a scatter plot.
When output = "data", the unaveraged data are returned with columns for all the performance metrics and the resample indicator.
plot.varImp.train

Value

Either a data frame, ggplot object or lattice object

Author(s)

Max Kuhn

See Also

gafs, safs, ggplot, xyplot

Examples

```r
## Not run:
set.seed(1)
train_data <- twoClassSim(100, noiseVars = 10)
test_data <- twoClassSim(10, noiseVars = 10)

## A short example
ctrl <- safsControl(functions = rfSA,
  method = "cv",
  number = 3)

rf_search <- safs(x = train_data[, -ncol(train_data)],
  y = train_data$Class,
  iters = 50,
  safsControl = ctrl)

plot(rf_search)
plot(rf_search,
  output = "lattice",
  auto.key = list(columns = 2))

plot_data <- plot(rf_search, output = "data")
summary(plot_data)

## End(Not run)
```

Description

This function produces lattice and ggplot 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], ...)

## S3 method for class 'varImp.train'
ggplot(data, mapping = NULL,
       top = dim(data$importance)[1], ..., environment = NULL)
```

Arguments

- `x, data` 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`)
- `mapping, environment` unused arguments to make consistent with `ggplot2` generic method

Details

For models where there is only one importance value, such a 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. Horizontal bar charts are used for `ggplot`.

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

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.

Usage

```r
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`

Details

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

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:
data(mdrr)
set.seed(90)
inTrain <- createDataPartition(mdrrClass, p = .5)[[1]]

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

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)
```
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.

Usage

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

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`

Details

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

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:
# 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)
umbSamples <- 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]
trainY <- Satellite[trainSamples, 37]
testX <- Satellite[testSamples, -37]
testY <- Satellite[testSamples, 37]
unkX <- Satellite[unkSamples, -37]
```
```
knnFit <- train(trainX, trainY, "knn")
rpartFit <- train(trainX, trainY, "rpart")
```
```
predTargets <- extractPrediction(list(knnFit, rpartFit),
                                 testX = testX,
                                 testY = testY,
                                 unkX = unkX)
```
```
plotObsVsPred(predTargets)
```
### 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, ...)  
## S3 method for class 'plsda'
predict(object, newdata = NULL, ncomp = NULL,  
           type = "class", ...)

## Default S3 method:
plsda(x, y, ncomp = 2, probMethod = "softmax",  
      prior = NULL, ...)
```

**Arguments**

- **x**: a matrix or data frame of predictors
- **...**: 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 `plsd`a
- **newdata**: a matrix or data frame of predictors
- **ncomp**: the number of components to include in the model. Predictions can be made for models with values less than `ncomp`
- **type**: either "class", "prob" or "raw" to produce the predicted class, class probabilities or the raw model scores, respectively
- **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
- **probMethod**: either "softmax" or "Bayes" (see Details)
- **prior**: a vector or prior probabilities for the classes (only used for `probMethod = "Bayes"`)
Details

If a factor is supplied, the appropriate indicator matrix is created. A multivariate PLS model is fit to the indicator matrix using the \texttt{plsr} or \texttt{spls} function. Two prediction methods can be used.

The \texttt{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, \texttt{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 \texttt{ncomp} times for every possible PLS model. The \texttt{NaiveBayes} function is used with \texttt{usekernel = TRUE} for the posterior probability calculations.

Value

For \texttt{plsda}, an object of class "plsda" and "mvr". For \texttt{splsda}, an object of class \texttt{splsda}.

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

See Also

\texttt{plsr}, \texttt{spls}

Examples

```r
# Not run:
data(mdrr)
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"],
    xlab = "Active Prob", xlim = c(-1,1))
histogram(~predict(useSoftmax, testDescr, type = "prob")[,"Active"],
    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)

## End(Not run)

---

### pottery

#### Pottery from Pre-Classical Sites in Italy

**Description**

Measurements of 58 pottery samples.

**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], ...)

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

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.
- **...**: 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.
- **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".

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


See Also

`resamples`, `barchart`, `xyplot`, `splom`, `hclust`

Examples

```r
### Not run:
# load(url("http://topepo.github.io/caret/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)
```
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 = NULL, ...)

## 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).
- `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 two-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 (for regression or `type = "class"`) or a data frame of class probabilities. By default, when the model predicts a number, a vector of numeric predictions is returned. When a classification model is used, the default prediction is a factor vector of classes.

**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 from 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.
**predict.gafs**

*Predict new samples*

### Description

Predict new samples using *safs* and *gafs* objects.

### Usage

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

### Arguments

- **object**: an object of class *safs* or *gafs*
- **newdata**: a data frame or matrix of predictors.
- **...**: not currently used

### Details

Only the predictors listed in `object$optVariables` are required.

### Value

The type of result depends on what was specified in `object$control$functions$predict`.

---

**Author(s)**

Max Kuhn

**See Also**

*bagEarth*

### Examples

```r
## Not run:
data(trees)
## out of bag predictions vs just re-predicting the training set
set.seed(655)
fit1 <- bagEarth(Volume ~ ., data = trees, keepX = TRUE)
set.seed(655)
fit2 <- bagEarth(Volume ~ ., data = trees, keepX = FALSE)
hist(predict(fit1) - predict(fit2))
## End(Not run)
```
predict.knn3

Author(s)
Max Kuhn

See Also
safs, gafs

Examples

```r
## Not run:

set.seed(1)
train_data <- twoClassSim(100, noiseVars = 10)
test_data  <- twoClassSim(10,  noiseVars = 10)

## A short example
ctrl <- safsControl(functions = rfSA,
                      method = "cv",
                      number = 3)

rf_search <- safs(x = train_data[, -ncol(train_data)],
                   y = train_data$Class,
                   iters = 3,
                   safsControl = ctrl)

rf_search

predict(rf_search, train_data)

## End(Not run)
```

---

**predict.knn3**  
*Predictions from k-Nearest Neighbors*

**Description**

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

**Usage**

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

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>

Description

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

Usage

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

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
predictors

Author(s)

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

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, ...)

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.

preProcess

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

preProcess(x, ...)

## Default S3 method:
predict(object, newdata, ...)

Arguments

x a matrix or data frame. Non-numeric predictors are allowed but will be ignored.

... additional arguments to pass to fastICA, such as n.comp

method a character vector specifying the type of processing. Possible values are "Box-
Cox", "YeoJohnson", "expoTrans", "center", "scale", "range", "knnImpute", "bag-
Impute", "medianImpute", "pca", "ica", "spatialSign", "corr", "zv", "nzv", and
"conditionalX" (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?

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

freqCut the cutoff for the ratio of the most common value to the second most common
value. See nearZeroVar.

uniqueCut the cutoff for the percentage of distinct values out of the number of total samples.
See nearZeroVar.

cutoff a numeric value for the pair-wise absolute correlation cutoff. See findCorrelation.

rangeBounds a two-element numeric vector specifying closed interval for range transforma-
tion

object an object of class preProcess

newdata a matrix or data frame of new data to be pre-processed
Details

In all cases, transformations and operations are estimated using the data in \( x \) and these operations are applied to new data using these values; nothing is recomputed when using the \texttt{predict} function.

The Box-Cox (\texttt{method = "BoxCox"}), Yeo-Johnson (\texttt{method = "YeoJohnson"}), and exponential transformations (\texttt{method = "expTrans"}) 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.

\texttt{method = "center"} subtracts the mean of the predictor’s data (again from the data in \( x \)) from the predictor values while \texttt{method = "scale"} divides by the standard deviation.

The "range" transformation scales the data to be within \texttt{rangeBounds}. If new samples have values larger or smaller than those in the training set, values will be outside of this range.

Predictors that are not numeric are ignored in the calculations.

\texttt{method = "zv"} identifies numeric predictor columns with a single value (i.e. having zero variance) and excludes them from further calculations. Similarly, \texttt{method = "nzv"} does the same by applying \texttt{nearZeroVar} exclude "near zero-variance" predictors. The options \texttt{freqCut} and \texttt{uniqueCut} can be used to modify the filter.

\texttt{method = "corr"} seeks to filter out highly correlated predictors. See \texttt{findCorrelation}.

For classification, \texttt{method = "conditionalX"} examines the distribution of each predictor conditional on the outcome. If there is only one unique value within any class, the predictor is excluded from further calculations (see \texttt{checkConditionalX} for an example). When outcome is not a factor, this calculation is not executed. This operation can be time consuming when used within resampling via \texttt{train}.

The operations are applied in this order: zero-variance filter, near-zero variance filter, correlation filter, Box-Cox/Yeo-Johnson/exponential 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 numeric variables in \( x \) has less than two unique values unless either \texttt{method = "zv"} or \texttt{method = "nzv"} are invoked.
Non-numeric data will not be pre-processed and there values will be in the data frame produced by the predict function. Note that when PCA or ICA is used, the non-numeric columns may be in different positions when predicted.

Value

`preProcess` results in a list with elements

- `call`: the function call
- `method`: a named list of operations and the variables used for each
- `dim`: the dimensions of x
- `bc`: Box-Cox transformation values, see `BoxCoxTrans`
- `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)

`predict.preProcess` will produce a data frame.

Author(s)

Max Kuhn, median imputation by Zachary Mayer

References

- [http://topepo.github.io/caret/pre-processing.html](http://topepo.github.io/caret/pre-processing.html)
- Kuhn (2008), Building predictive models in R using the caret ([http://www.jstatsoft.org/article/view/v28i05/v28i05.pdf](http://www.jstatsoft.org/article/view/v28i05/v28i05.pdf))
See Also

BoxCoxTrans, expoTrans boxcox, prcomp, fastICA, spatialSign

Examples

```r
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, mode = x$mode, digits = max(3,
    getOption("digits") - 3), printStats = TRUE, ...)
```

Arguments

- `x` an object of class confusionMatrix
- `mode` a single character string either "sens_spec", "prec_recall", or "everything"
- `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, printCall = FALSE, details = FALSE,
      selectCol = FALSE, showSD = FALSE, ...)  
```

Arguments

- **x**: an object of class `train`.
- **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 whether to add a column with a star next to the selected parameters.
- **showSD**: a logical whether to show the standard deviation of the resampling results within parentheses (e.g. "4.24 (0.493)").
- **...**: options passed to `format`.

Details

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

Value

A matrix with the complexity parameters and performance (invisibly).

Author(s)

Max Kuhn

See Also

`train`
Examples

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

options(digits = 3)

library klaR)

rdaFit <- train(TrainData, TrainClasses, method = "rda",
control = trainControl(method = "cv"))
rdaFit

print(rdaFit, showSD = TRUE)

## End(Not run)
```

Recall

### Calculate recall, precision and F values

#### Description

These functions calculate the recall, precision or F values of a measurement system for finding/retrieving relevant documents compared to reference results (the truth regarding relevance). The measurement and "truth" data must have the same two possible outcomes and one of the outcomes must be thought of as a "relevant" result.

#### Usage

```r
call(data, ...)
```

#### Default S3 method:

```r
call(data, relevant = rownames(data)[1], ...)
```

#### S3 method for class 'table'

```r
call(data, relevant = rownames(data)[1], ...)
```

```r
call(data, reference, relevant = levels(reference)[1], na.rm = TRUE, ...)
```

```r
precision(data, ...)
```

#### Default S3 method:

```r
precision(data, reference, relevant = levels(reference)[1], na.rm = TRUE, ...)
```

#### S3 method for class 'table'

```r
precision(data, relevant = rownames(data)[1], ...)
```
F_meas(data, ...)  

## Default S3 method:  
F_meas(data, reference,  
    relevant = levels(reference)[1], beta = 1, na.rm = TRUE, ...)  

## S3 method for class 'table'  
F_meas(data, relevant = rownames(data)[1], beta = 1,  
    ...)  

### Arguments  
- **data**: for the default functions, a factor containing the discrete measurements. For the table function, a table.  
- **...**: not currently used  
- **relevant**: a character string that defines the factor level corresponding to the "relevant" results  
- **reference**: a factor containing the reference values (i.e. truth)  
- **na.rm**: a logical value indicating whether NA values should be stripped before the computation proceeds  
- **beta**: a numeric value used to weight precision and recall. A value of 1 is traditionally used and corresponds to the harmonic mean of the two values but other values weight recall beta times more important than precision.  

### Details  
The recall (aka sensitivity) is defined as the proportion of relevant results out of the number of samples which were actually relevant. When there are no relevant results, recall is not defined and a value of NA is returned.  
The precision is percentage of predicted truly relevant results of the total number of predicted relevant results and characterizes the "purity in retrieval performance" (Buckland and Gey, 1994)  
The measure "F" is a combination of precision and recall (see below).  

Suppose a 2x2 table with notation  

<table>
<thead>
<tr>
<th>Predicted</th>
<th>Reference</th>
<th>Irrelevant</th>
<th>(\text{relevant} = A)</th>
<th>(\text{Irrelevant} = C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>relevant</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Irrelevant</td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The formulas used here are:  

\[
\text{recall} = \frac{A}{A + C}  
\]

\[
\text{precision} = \frac{A}{A + B}  
\]

\[
F_i = (1 + \beta^2) \cdot \frac{\text{prec} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}  
\]

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`

**Examples**

```
### Data in Table 2 of Powers (2007)

dlvs <- c("Relevant", "Irrelevant")
tbl_2_1_pred <- factor(rep(dlvs, times = c(42, 58)), levels = dlvs)
tbl_2_1_truth <- factor(c(rep(dlvs, times = c(30, 12)),
                         rep(dlvs, times = c(30, 28))),
                        levels = dlvs)
tbl_2_1 <- table(tbl_2_1_pred, tbl_2_1_truth)

precision(tbl_2_1)
precision(data = tbl_2_1_pred, reference = tbl_2_1_truth, relevant = "Relevant")
recall(tbl_2_1)
recall(data = tbl_2_1_pred, reference = tbl_2_1_truth, relevant = "Relevant")


tbl_2_2_pred <- factor(rep(dlvs, times = c(76, 24)), levels = dlvs)
tbl_2_2_truth <- factor(c(rep(dlvs, times = c(56, 20)),
                         rep(dlvs, times = c(12, 12))),
                        levels = dlvs)
tbl_2_2 <- table(tbl_2_2_pred, tbl_2_2_truth)

precision(tbl_2_2)
precision(data = tbl_2_2_pred, reference = tbl_2_2_truth, relevant = "Relevant")
recall(tbl_2_2)
recall(data = tbl_2_2_pred, reference = tbl_2_2_truth, relevant = "Relevant")
```
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
resampleHist(object, type = "density", ...)

Arguments
object  an object resulting form 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

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

knnFit <- train(TrainData, TrainClasses, "knn")
resampleHist(knnFit)
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'
sort(x, decreasing = FALSE, metric = x$metric[1],
    FUN = mean, ...)

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

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

## S3 method for class 'resamples'
summary(object, metric = object$metrics, ...)

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

## S3 method for class 'resamples'
as.data.frame(x, row.names = NULL,
    optional = FALSE, metric = x$metric[1], ...)

modelCor(x, metric = x$metric[1], ...)

## S3 method for class 'resamples'
print(x, ...)

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.

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

modelNames an optional set of names to give to the resampling results
decreasing logical. Should the sort be increasing or decreasing?
metric  a character string for the performance measure used to sort or computing the between-model correlations
FUN     a function whose first argument is a vector and returns a scalar, to be applied to each model’s performance measure.
object  an object generated by \textit{resamples}
row.names, optional
          not currently used but included for consistency with as.data.frame

Details

The ideas and methods here are based on Hothorn et al. (2005) and Eugster et al. (2008).
The results from \textit{train} can have more than one performance metric per resample. Each metric in the input object is saved.
\textit{resamples} checks that the resampling results match; that is, the indices in the object \texttt{trainObject$control$index} are the same. Also, the argument \texttt{trainControl returnResamp} should have a value of “final” for each model.
The summary function computes summary statistics across each model/metric combination.

Value

For \textit{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 \texttt{train} method argument values for each model

For \textit{sort.resamples} a character string in the sorted order is generated. \texttt{modelCor} returns a correlation matrix.

Author(s)

Max Kuhn

References


See Also

\texttt{train}, \texttt{trainControl}, \texttt{diff.resamples}, \texttt{xyplot.resamples}, \texttt{densityplot.resamples}, \texttt{bwplot.resamples}, \texttt{splom.resamples}
resampleSummary

Examples

data(BloodBrain)
set.seed(1)

## tmp <- createDataPartition(logBBB,
##   p = 0.8,
##   times = 100)

## rpartFit <- train(bbbDescr, logBBB,
##   "rpart",
##   tuneLength = 16,
##   trControl = trainControl(
##     method = "LGOCV", index = tmp))

## ctreeFit <- train(bbbDescr, logBBB,
##   "ctree",
##   trControl = trainControl(
##     method = "LGOCV", index = tmp))

## earthFit <- train(bbbDescr, logBBB,
##   "earth",
##   tuneLength = 20,
##   trControl = trainControl(
##     method = "LGOCV", index = tmp))

## or load pre-calculated results using:
## load(url("http://caret.r-forge.r-project.org/exampleModels.RData"))

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

## resamps
## summary(resamps)

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)
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))
```

---

**rfe**  
*Backwards Feature Selection*

Description

A simple backwards selection, a.k.a. recursive feature elimination (RFE), algorithm
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'
update(object, x, y, size, ...)

Arguments

- **x**: a matrix or data frame of predictors for model training. This object must have unique column names.
- **...**: options to pass to the model fitting function (ignored in predict.rfe)
- **y**: a vector of training set outcomes (either numeric or factor)
- **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://topepo.github.io/caret/recursive-feature-elimination.html#rfe](http://topepo.github.io/caret/recursive-feature-elimination.html#rfe) has more details and examples related to this function.
- **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
- **label**: an optional character string to be printed when in verbose mode.
- **seeds**: an optional vector of integers for the size. The vector should have length of length(sizes)+1
- **object**: an object of class rfe
- **size**: a single integers corresponding to the number of features that should be retained in the updated model
Details

More details on this function can be found at `http://topepo.github.io/caret/recursive-feature-elimination.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 web page `http://topepo.github.io/caret/recursive-feature-elimination.html#rfe` 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**.

When updating a model, if the entire set of resamples were not saved using **rfeControl**(returnResamp = "final"), the existing resamples are removed with a warning.

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)`)
- **pred**: 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

```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, 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)
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 = 0.75, index = NULL, indexOut = 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)

saveDetails a logical to save the predictions and variable importances from the selection process

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

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.

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).

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://topepo.github.io/caret/recrursive-feature-elimination.html#rfe.

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.
- • ...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: `ImFuncs`, `rffuncs`, `treebagFuncs` and `nbFuncs`.

Model details about these functions, including examples, are at [http://topepo.github.io/caret/recursive-feature-elimination.html](http://topepo.github.io/caret/recursive-feature-elimination.html).

**Value**

A list
Sacramento

Author(s)
Max Kuhn

See Also
rfe, lmFuncs, rFFuncs, 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 = rFuncs,
    seeds = seeds,
    number = 50))

## End(Not run)
```

Sacramento Sacramento CA Home Prices

Description
This data frame contains house and sale price data for 932 homes in Sacramento CA. The original data were obtained from the website for the SpatialKey software. From their website: "The Sacramento real estate transactions file is a list of 985 real estate transactions in the Sacramento area reported over a five-day period, as reported by the Sacramento Bee." Google was used to fill in missing/incorrect data.

Value
Sacramento a data frame with columns 'city', 'zip', 'beds', 'baths', 'sqft', 'type', 'price', 'latitude', and 'longitude'

Source
SpatialKey website: https://support.spatialkey.com/spatialkey-sample-csv-data
Examples

data(Sacramento)

set.seed(955)
in_train <- createDataPartition(log10(Sacramento$price), p = .8, list = FALSE)

training <- Sacramento[in_train,]
testing <- Sacramento[-in_train,]

safs        Simulated annealing feature selection

Description

Supervised feature selection using simulated annealing

safs conducts a supervised binary search of the predictor space using simulated annealing (SA).
See Kirkpatrick et al (1983) for more information on this search algorithm.

This function conducts the search of the feature space repeatedly within resampling iterations. First,
the training data are split by whatever resampling method was specified in the control function. For
example, if 10-fold cross-validation is selected, the entire simulated annealing search is conducted
10 separate times. For the first fold, nine tenths of the data are used in the search while the remaining
tenth is used to estimate the external performance since these data points were not used in the search.

During the search, a measure of fitness (i.e. SA energy value) is needed to guide the search. This is
the internal measure of performance. During the search, the data that are available are the instances
selected by the top-level resampling (e.g. the nine tenths mentioned above). A common approach
is to conduct another resampling procedure. Another option is to use a holdout set of samples to
determine the internal estimate of performance (see the holdout argument of the control function).
While this is faster, it is more likely to cause overfitting of the features and should only be used
when a large amount of training data are available. Yet another idea is to use a penalized metric
(such as the AIC statistic) but this may not exist for some metrics (e.g. the area under the ROC
curve).

The internal estimates of performance will eventually overfit the subsets to the data. However, since
the external estimate is not used by the search, it is able to make better assessments of overfitting.
After resampling, this function determines the optimal number of iterations for the SA.

Finally, the entire data set is used in the last execution of the simulated annealing algorithm search
and the final model is built on the predictor subset that is associated with the optimal number of
iterations determined by resampling (although the update function can be used to manually set the
number of iterations).

This is an example of the output produced when safsControl(verbose = TRUE) is used:

Fold03  1  0.401 (11)
Fold03  2  0.401->0.410 (11+1, 91.7%) *
The text "Fold03" indicates that this search is for the third cross-validation fold. The initial subset of 11 predictors had a fitness value of 0.401. The next iteration added a single feature to the existing best subset of 11 (as indicated by "11+1") that increased the fitness value to 0.410. This new solution, which has a Jaccard similarity value of 91.7% to the current best solution, is automatically accepted. The third iteration adds another feature to the current set of 12 but does not improve the fitness. The acceptance probability for this difference is shown to be 95.6% and the "A" indicates that this new sub-optimal subset is accepted. The fourth iteration does not show an increase and is not accepted. Note that the Jaccard similarity value of 85.7% is the similarity to the current best solution (from iteration 2) and the "12+2" indicates that there are two additional features added from the current best that contains 12 predictors.

The search algorithm can be parallelized in several places:

1. each externally resampled SA can be run independently (controlled by the allowParallel option of \texttt{safsControl})
2. if inner resampling is used, these can be run in parallel (controls depend on the function used. See, for example, \texttt{trainControl})
3. any parallelization of the individual model fits. This is also specific to the modeling function.

It is probably best to pick one of these areas for parallelization and the first is likely to produces the largest decrease in run-time since it is the least likely to incur multiple re-starting of the worker processes. Keep in mind that if multiple levels of parallelization occur, this can effect the number of workers and the amount of memory required exponentially.

Usage

\texttt{safs(x, \ldots)}

## Default S3 method:
\texttt{safs(x, y, iters = 10, differences = TRUE,}
\texttt{safsControl = \texttt{safsControl}(, \ldots)}

Arguments

\texttt{x} an object where samples are in rows and features are in columns. This could be a simple matrix, data frame or other type (e.g. sparse matrix). See Details below.

\texttt{\ldots} arguments passed to the classification or regression routine specified in the function \texttt{safsControl$functions$fit}

\texttt{y} a numeric or factor vector containing the outcome for each sample.

\texttt{iters} number of search iterations

\texttt{differences} a logical: should the difference in fitness values with and without each predictor be calculated

\texttt{safsControl} a list of values that define how this function acts. See \texttt{safsControl} and URL.
Value

an object of class `safs`

Author(s)

Max Kuhn

References


Kuhn and Johnson (2013), Applied Predictive Modeling, Springer


See Also

`safsControl, predict.safs`

Examples

```r
## Not run:

set.seed(1)
train_data <- twoClassSim(100, noiseVars = 10)
test_data <- twoClassSim(10, noiseVars = 10)

## A short example
ctrl <- safsControl(functions = rfSA,
                     method = "cv",
                     number = 3)

rf_search <- safs(x = train_data[, -ncol(train_data)],
                  y = train_data$Class,
                  iters = 3,
                  safsControl = ctrl)

rf_search

## End(Not run)
```
Ancillary simulated annealing functions

Description

Built-in functions related to simulated annealing

These functions are used with the functions argument of the `safsControl` function. More information on the details of these functions are at [http://topepo.github.io/caret/feature-selection-using-simulated-annealing.html](http://topepo.github.io/caret/feature-selection-using-simulated-annealing.html).

The initial function is used to create the first predictor subset. The function `safs_initial` randomly selects 20% of the predictors. Note that, instead of a function, `safs` can also accept a vector of column numbers as the initial subset.

`safs_perturb` is an example of the operation that changes the subset configuration at the start of each new iteration. By default, it will change roughly 1% of the variables in the current subset.

The `prob` function defines the acceptance probability at each iteration, given the old and new fitness (i.e., energy values). It assumes that smaller values are better. The default probability function computed the percentage difference between the current and new fitness value and using an exponential function to compute a probability:

\[
\text{prob} = \exp\left(\frac{\text{current} - \text{new}}{\text{current} \times \text{iteration}}\right)
\]

Usage

safs_initial(vars, prob = 0.2, ...)
safs_perturb(x, vars, number = floor(vars * 0.01) + 1)
safs_prob(old, new, iteration = 1)
caretSA
treebagSA
rfSA

Arguments

- `vars` the total number of possible predictor variables
- `prob` The probability that an individual predictor is included in the initial predictor set
- `...` not currently used
- `x` the integer index vector for the current subset
- `number` the number of predictor variables to perturb
- `old, new` fitness values associated with the current and new subset
- `iteration` the number of iterations overall or the number of iterations since restart (if improve is used in `safsControl`)
Format

An object of class list of length 8.

Value

The return value depends on the function. Note that the SA code encodes the subsets as a vector of integers that are included in the subset (which is different than the encoding used for GAs).

The objects caretSA, rfSA and treebagSA are example lists that can be used with the functions argument of safsControl.

In the case of caretSA, the ... structure of safs passes through to the model fitting routine. As a consequence, the train function can easily be accessed by passing important arguments belonging to train to safs. See the examples below. By default, using caretSA will used the resampled performance estimates produced by train as the internal estimate of fitness.

For rfSA and treebagSA, the randomForest and bagging functions are used directly (i.e. train is not used). Arguments to either of these functions can also be passed to them though the safs call (see examples below). For these two functions, the internal fitness is estimated using the out-of-bag estimates naturally produced by those functions. While faster, this limits the user to accuracy or Kappa (for classification) and RMSE and R-squared (for regression).

Author(s)

Max Kuhn

References


See Also

safs, safsControl

Examples

```r
selected_vars <- safs_initial(vars = 10, prob = 0.2)
selected_vars

###
safs_perturb(selected_vars, vars = 10, number = 1)

###
safs_prob(old = .8, new = .9, iteration = 1)
safs_prob(old = .5, new = .6, iteration = 1)

grid <- expand.grid(old = c(4, 3.5),
                    new = c(4.5, 4, 3.5) + 1,
                    iter = 1:40)
```
grid <- subset(grid, old < new)

grid$prob <- apply(grid, 1, 
    function(x) 
      safs_prob(new = x["new"], 
        old = x["old"], 
        iteration = x["iter"])
)

grid$Difference <- factor(grid$new - grid$old)
grid$Group <- factor(paste("Current Value", grid$old))

ggplot(grid, aes(x = iter, y = prob, color = Difference)) + 
  geom_line() + facet_wrap(~Group) + theme_bw() + 
  ylab("Probability") + xlab("Iteration")

## Not run:
##
## Hypothetical examples
lda_sa <- safs(x = predictors, 
  y = classes, 
  safsControl = safsControl(functions = caretSA), 
  ## now pass arguments to `train`
  method = "lda", 
  metric = "Accuracy" 
  trControl = trainControl(method = "cv", classProbs = TRUE))

rf_sa <- safs(x = predictors, 
  y = classes, 
  safsControl = safsControl(functions = rfsa), 
  ## these are arguments to `randomForest`
  ntree = 1000, 
  importance = TRUE)

## End(Not run)

### sbf

**Selection By Filtering (SBF)**

**Description**

Model fitting after applying univariate filters

**Usage**

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.

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

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

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

- **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 the factors appearing as variables in the model formula.

- **object**
  - an object of class sbf

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

**Details**


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 lmSBF.

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)

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)

---

**sbfControl**

*Control Object for Selection By Filtering (SBF)*

**Description**

Controls the execution of models with simple filters for feature selection

**Usage**

```r
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, indexOut = NULL, timingSamps = 0, seeds = NA, allowParallel = TRUE, multivariate = FALSE)
```

**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)
saveDetails     a logical to save the predictions and variable importances from the selection process
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
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.
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).
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?
multivariate     a logical; should all the columns of x be exposed to the score function at once?

Details

More details on this function can be found at http://topepo.github.io/caret/feature-selection-using-univariate.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
sbfControl

- x the current set of predictor set for the held-back samples

The score function is used to return scores with names for each predictor (such as a p-value). Inputs are:

- x the predictors for the training samples. If sbfControl()$multivariate is TRUE, this will be the full predictor matrix. Otherwise it is a vector for a specific predictor.
- y the current training outcomes

When sbfControl()$multivariate is TRUE, the score function should return a named vector where length(scores) == ncol(x). Otherwise, the function's output should be a single value. Univariate examples are give by anovaScores for classification and gamScores for regression and the example below.

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://topepo.github.io/caret/ 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
set.seed(1)
RFwithGAM <- sbf(bbbDescr, logBBB,
sbfControl = sbfControl(functions = rfsBF,
  verbose = FALSE,
  seeds = sample.int(100000, 11),
```

RFwithGAM

## A simple example for multivariate scoring
rfSBF2 <- rfSBF
rfSBF2$score <- function(x, y) apply(x, 2, rfSBF$score, y = y)
set.seed(1)
RFwithGAM2 <- sbf(bbbDescr, logBBB, 
  sbfControl = sbfControl(functions = rfSBF2, 
    verbose = FALSE, 
    seeds = sample.int(100000, 11), 
    method = "cv", 
    multivariate = TRUE))
RFwithGAM2

## End(Not run)

---

**scat**

*Morphometric Data on Scat*

**Description**

Reid (2015) collected data on animal feses in coastal California. The data consist of DNA verified species designations as well as fields related to the time and place of the collection and the scat itself. The data frame `scat_orig` contains while `scat` contains data on the three main species.

**Value**

<table>
<thead>
<tr>
<th>scat_orig</th>
<th>the entire data set in the Supplemental Materials</th>
</tr>
</thead>
<tbody>
<tr>
<td>scat</td>
<td>data on the three main species</td>
</tr>
</tbody>
</table>

**Source**

**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).

**Details**

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://topepo.github.io/caret/segmentationOriginal.RData](http://topepo.github.io/caret/segmentationOriginal.RData).

**Value**

segmentationData  
data frame of cells

**Source**


---

**SLC14_1**  
*Simulation Functions*

**Description**

This function simulates regression and classification data with truly important predictors and irrelevant predictions.
Usage

SLC14_1(n = 100, noiseVars = 0, corrVars = 0, corrType = "AR1", corrValue = 0)
SLC14_2(n = 100, noiseVars = 0, corrVars = 0, corrType = "AR1", corrValue = 0)
LPH07_1(n = 100, noiseVars = 0, corrVars = 0, corrType = "AR1", corrValue = 0, factors = FALSE, class = FALSE)
LPH07_2(n = 100, noiseVars = 0, corrVars = 0, corrType = "AR1", corrValue = 0)
twoClassSim(n = 100, intercept = -5, linearVars = 10, noiseVars = 0, corrVars = 0, corrType = "AR1", corrValue = 0, mislabel = 0, ordinal = FALSE)

Arguments

n The number of simulated data points
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.
factors Should the binary predictors be converted to factors?
class Should the simulation produce class labels instead of numbers?
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.
mislabel The proportion of data that is possibly mislabeled. Only used when ordinal = FALSE. See Details below.
ordinal Should an ordered factor be returned? See Details below.

Details

The first function (twoClassSim) generates two class data. The data are simulated in different sets. First, two multivariate normal predictors (denoted here as A and B) are created with a correlation about 0.65. They change the log-odds using main effects and an interaction:

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
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.

When \(\text{ordinal} = \text{FALSE}\), 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.

For \(\text{ordinal} = \text{TRUE}\), random normal errors are added to the linear predictor (i.e. prior to computing the probability) and cut points (0.00, 0.20, 0.75, and 1.00) are used to bin the probabilities into classes “low”, “med”, and “high” (despite the function’s name).

The remaining functions simulate regression data sets. \(\text{LPH07}_1\) and \(\text{LPH07}_2\) are from van der Laan et al. (2007). The first function uses random Bernoulli variables that have a 40% probability of being a value of 1. The true regression equation is:

\[
2,w:1*w:10 + 4*w:2*w:7 + 3*w:4*w:5 - 5*w:6*w:10 + 3*w:8*w:9 +
\]

\[
w:1*w:2*w:4 - 2*w:7*(1-w:6)*w:2*w:9 - 4*(1 - w:10)*w:1*(1-w:4)
\]

The simulated error term is a standard normal (i.e. Gaussian). The noise variables are simulated in the same manner as described above but are made binary based on whether the normal random variable is above or below 0. If \(\text{factors} = \text{TRUE}\), each of the predictors is coerced into a factor. This simulation can also be adapted for classification using the option \(\text{class} = \text{TRUE}\). In this case, the outcome is converted to be a factor by first computing the logit transformation of the equation above and using uniform random numbers to assign the observed class.

A second function (\(\text{LPH07}_2\)) uses 20 independent Gaussians with mean zero and variance 16. The functional form here is:

\[
x:1*x:2 + x:10^2 - x:3*x:17 - x:15*x:4 + x:9*x:5 + x:19 -
x:20^2 + x:9*x:8
\]

The error term is also Gaussian with mean zero and variance 16.

The function \(\text{SLC14}_1\) simulates a system from Sapp et al. (2014). All informative predictors are independent Gaussian random variables with mean zero and a variance of 9. The prediction equation is:

\[
x:1 + \sin(x:2) + \log(abs(x:3)) + x:4^2 + x:5*x:6 +
\]

\[
I(x:7*x:8*x:9 < 0) + I(x:10 > 0) + x:11*I(x:11 > 0) + sqrt(abs(x:12)) +
\]

\[
cos(x:13) + 2*x:14 + abs(x:15) + I(x:16 < -1) + x:17*I(x:17 < -1) - 2 * x:18
\]

\[- x:19*x:20\]
The random error here is also Gaussian with mean zero and a variance of 9.

SLC14_2 is also from Sapp et al. (2014). Two hundred independent Gaussian variables are generated, each having mean zero and variance 16. The functional form is

\[-1 + \log(|x_1|) + \ldots + \log(|x_{200}|)\]

and the error term is Gaussian with mean zero and a variance of 25.

For each simulation, 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

\[
\begin{pmatrix}
1 & r & r^2 & r^3 \\
0 & 1 & r & r^2 \\
0 & 0 & 1 & r \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

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)
- **Nonlinear1**, **Nonlinear2**, **Nonlinear3**: Uncorrelated random uniform predictors (J, K and L above).
- **Linear1**, **Linear2**, **Linear3**, **list()**: Optional uncorrelated standard normal predictors (C through H above)
- **Noise1**, **Noise2**, **Noise3**, **list()**: Optional uncorrelated standard normal predictions
- **Corr1**, **Corr2**, **Corr3**, **list()**: Optional correlated multivariate normal predictors (each with unit variances)

Author(s)

Max Kuhn
spatialSign

References

Examples

```r
example <- twoClassSim(100, linearVars = 1)
splom(~example[, 1:6], groups = example$Class)
```

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 $w$ is $w / \|w\|$.

Usage

```r
spatialSign(x, ...)  
## Default S3 method:
spatialSign(x, na.rm = TRUE, ...)

## S3 method for class 'matrix'
spatialSign(x, na.rm = TRUE, ...)

## S3 method for class 'data.frame'
spatialSign(x, na.rm = TRUE, ...)
```

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.
- `...` Not currently used.
- `na.rm` A logical; should missing data be removed when computing the norm of the vector?

Value

A vector, matrix or data frame with the same dim names of the original data.
summary.bagEarth

Author(s)
Max Kuhn

References

Examples
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")

summary.bagEarth  Summarize a bagged earth or FDA fit

Description
The function shows a summary of the results from a bagged earth model

Usage
## 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 the bootstrap samples.
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)
set.seed(9655)
fit <- bagEarth(trees[,3], trees[3])
summary(fit)

## End(Not run)
```

tecator Fat, Water and Protein Content of Meat Samples

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.

Details

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 -log10 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.
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,], col = plotColors[i])
}
title("Predictor Profiles for 10 Random Samples")

---

thresholder  

*Generate Data to Choose a Probability Threshold*

Description

This function uses the resampling results from a train object to generate performance statistics over a set of probability thresholds for two-class problems.
Usage

`thresholder(x, threshold, final = TRUE, statistics = "all")`

Arguments

- **x**: A `train` object where the values of `savePredictions` was either TRUE, "all", or "final" in `trainControl`. Also, the control argument `clasProbs` should have been TRUE.
- **threshold**: A numeric vector of candidate probability thresholds between [0,1]. If the class probability corresponding to the first level of the outcome is greater than the threshold, the data point is classified as that level.
- **final**: A logical: should only the final tuning parameters chosen by `train` be used when `savePredictions = 'all'`?
- **statistics**: A character vector indicating which statistics to calculate. See details below for possible choices; the default value "all" computes all of these.

Details

The argument `statistics` designates the statistics to compute for each probability threshold. One or more of the following statistics can be selected:

- Sensitivity
- Specificity
- Pos Pred Value
- Neg Pred Value
- Precision
- Recall
- F1
- Prevalence
- Detection Rate
- Detection Prevalence
- Balanced Accuracy
- Accuracy
- Kappa
- J
- Dist

For a description of these statistics (except the last two), see the documentation of `confusionMatrix`. The last two statistics are Youden’s J statistic and the distance to the best possible cutoff (i.e. perfect sensitivity and specificity).
Value

A data frame with columns for each of the tuning parameters from the model along with an additional column called `prob_threshold` for the probability threshold. There are also columns for summary statistics averaged over resamples with column names corresponding to the input argument `statistics`.

Examples

```r
## Not run:
set.seed(2444)
dat <- twoClassSim(500, intercept = -10)
table(dat$class)

ctrl <- trainControl(method = "cv",
    classProbs = TRUE,
    savePredictions = "all",
    summaryFunction = twoclassSummary)

set.seed(2863)
mod <- train(Class ~ ., data = dat,
    method = "rda",
    tuneLength = 4,
    metric = "ROC",
    trControl = ctrl)

resample_stats <- thresher(mod,
    threshold = seq(.5, 1, by = 0.05),
    final = TRUE)

ggplot(resample_stats, aes(x = prob_threshold, y = J)) +
  geom_point()

ggplot(resample_stats, aes(x = prob_threshold, y = Dist)) +
  geom_point()

ggplot(resample_stats, aes(x = prob_threshold, y = Sensitivity)) +
  geom_point() +
  geom_point(aes(y = Specificity), col = "red")

## End(Not run)
```

---

**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 %in% c("RMSE", "logLoss", "MAE"), FALSE, TRUE),
trControl = trainControl(), tuneGrid = NULL,
tuneLength = ifelse(trControl$method == "none", 1, 3))

## S3 method for class 'formula'
train(formula, ..., weights, subset,
na.action = na.fail, contrasts = NULL)

## S3 method for class 'recipe'
train(x, data, method = "rf", ...
metric = ifelse(is.factor(y_dat), "Accuracy", "RMSE"),
maximize = ifelse(metric %in% c("RMSE", "logLoss", "MAE"), FALSE, TRUE),
trControl = trainControl(), tuneGrid = NULL,
tuneLength = ifelse(trControl$method == "none", 1, 3))

Arguments

x A numeric or factor vector containing the outcome for each sample.
method A string specifying which classification or regression model to use. Possible
values are found using names(getModelInfo())). See http://topepo.github.io/caret/train-models-by-tag.html. A list of functions can also be passed for a custom model function. See http://topepo.github.io/caret/using-your-own-model-in-train.html for details.
preProcess A string vector that defines a 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. Pre-processing code is only designed to work when x is a simple matrix or data frame.
**weights**  
A numeric vector of case weights. This argument will only affect models that allow case weights.

**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. (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://topepo.github.io/caret/using-your-own-model-in-train.html. (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://topepo.github.io/caret/available-models.html. (NOTE: If given, this argument must be named.)

**tuneLength**  
An integer denoting the amount of granularity in the tuning parameter grid. By default, this argument is the number of levels for each tuning parameter that should be generated by train. If trainControl has the option search = "random", this is the maximum number of tuning parameter combinations that will be generated by the random search. (NOTE: If given, this argument must be named.)

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

**data**  
Data frame from which variables specified in formula or recipe 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 the factors appearing as variables in the model formula.

**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.

The predictors in x can be most any object as long as the underlying model fit function can deal with the object class. The function was designed to work with simple matrices and data frame inputs, so some functionality may not work (e.g. pre-processing). When using string kernels, the vector of character strings should be converted to a matrix with a single column.
More details on this function can be found at http://topepo.github.io/caret/model-training-and-tuning.html.

A variety of models are currently available and are enumerated by tag (i.e. their model characteristics) at http://topepo.github.io/caret/train-models-by-tag.html.

More details on using recipes can be found at http://topepo.github.io/caret/using-recipes-with-train.html. Note that case weights can be passed into train using a role of "case weight" for a single variable. Also, if there are non-predictor columns that should be used when determining the model's performance metrics, the role of "performance var" can be used with multiple columns and these will be made available during resampling to the summary function.

**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` A 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://topepo.github.io/caret/

Kuhn (2008), “Building Predictive Models in R Using the caret” (http://www.jstatsoft.org/article/view/v028i05/v28i05.pdf)

https://topepo.github.io/recipes/

**See Also**

models, trainControl, update.train, modelLookup, createFolds, recipe

**Examples**

```r
## 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)

## Regression Example

library(mlbench)
data(BostonHousing)

lmFit <- train(medv ~ . + rm:1stat,
               data = BostonHousing,
               method = "lm")
```

```r
library(rpart)

rpartFit <- train(medv ~ .,
data = BostonHousing,
method = "rpart",
tuneLength = 9)

### Example with a custom metric

madSummary <- function (data,
lev = NULL,
model = NULL) {
  out <- mad(data$obs - data$pred,
na.rm = TRUE)
  names(out) <- "MAD"
  out
}

robustControl <- trainControl(summaryFunction = madSummary)
marsGrid <- expand.grid(degree = 1, nprune = (1:10) * 2)

earthFit <- train(medv ~ .,
data = BostonHousing,
method = "earth",
tuneGrid = marsGrid,
metric = "MAD",
maximize = FALSE,
trControl = robustControl)

### Example with a recipe

data(cox2)

cox2 <- cox2Descr

cox2$potency <- cox2IC50

library(recipes)

cox2_recipe <- recipe(potency ~ ., data = cox2) %>%
  ## Log the outcome
  step_log(potency, base = 10) %>%
  ## Remove sparse and unbalanced predictors
  step_nzv(all_predictors()) %>%
  ## Surface area predictors are highly correlated so
  ## conduct PCA just on these.
  step_pca(contains("VSA"), prefix = "surf_area_",
threshold = .95) %>%
  ## Remove other highly correlated predictors
  step_corr(all_predictors(), ~starts_with("surf_area"),
...
trainControl

threshold = .90

## Center and scale all of the non-PCA predictors
step_center(all_predictors(), -starts_with("surf_area"))
step_scale(all_predictors(), -starts_with("surf_area"))

set.seed(888)
cox2_lm <- train(cox2_recipe, 
data = cox2,
method = "lm",
trControl = trainControl(method = "cv"))

# Parallel Processing Example via multicore package

## library(doMC)
## registerDoMC(2)

## NOTE: don't run models form RWeka when using
### multicore. The session will crash.

## The code for train() does not change:
set.seed(1)
usingMC <- train(medv ~ ., 
data = BostonHousing,
method = "glmboost")

## or use:
## library(doMPI) or
## library(doParallel) or
## library(doSMP) and so on

## End(Not run)

---

**trainControl**

*Control parameters for train*

**Description**

Control the computational nuances of the `train` function

**Usage**

```
trainControl(method = "boot", number = ifelse(grepl("cv", method), 10, 25), repeats = ifelse(grepl("[d_]cv$", method), 1, NA), p = 0.75, 
search = "grid", initialWindow = NULL, horizon = 1, 
fixedWindow = TRUE, skip = 0, verboseIter = FALSE, 
returnData = TRUE, returnResamp = "final", savePredictions = FALSE, 
```
classProbs = FALSE, summaryFunction = defaultSummary,
selectionFunction = "best", preProcOptions = list(thresh = 0.95,
ICAcomp = 3, k = 5, freqCut = 95/5, uniqueCut = 10, cutoff = 0.9),
sampling = NULL, index = NULL, indexOut = NULL,
indexFinal = NULL, timingSamps = 0, predictionBounds = rep(FALSE, 2),
seeds = NA, adaptive = list(min = 5, alpha = 0.05, method = "gls", complete = TRUE), trim = FALSE, allowParallel = TRUE)

Arguments

method
The resampling method: "boot", "boot632", "optimism_boot", "boot_all", "cv", "repeatedcv", "LOOCV", "LGOCV" (for repeated training/test splits), "none" (only fits one model to the entire training set), "oob" (only for random forest, bagged trees, bagged earth, bagged flexible discriminant analysis, or conditional tree forest models), timeslice, "adaptive_cv", "adaptive_boot" or "adaptive_LGOCV"

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

p
For leave-group out cross-validation: the training percentage

search
Either "grid" or "random", describing how the tuning parameter grid is determined. See details below.

initialWindow, horizon, fixedWindow, skip
possible arguments to createTimeSlices when method is timeslice.

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
an indicator of how much of the hold-out predictions for each resample should be saved. Values can be either "all", "final", or "none". A logical value can also be used that convert to "all" (for true) or "none" (for false). "final" saves the predictions for the optimal tuning parameters.

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. Note that if method = "oob" is used, this option is ignored and a warning is issued.

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.
a single character value describing the type of additional sampling that is conducted after resampling (usually to resolve class imbalances). Values are "none", "down", "up", "smote", or "rose". The latter two values require the \texttt{DMwR} and \texttt{ROSE} packages, respectively. This argument can also be a list to facilitate custom sampling and these details can be found on the \texttt{caret} package website for sampling (link below).

\begin{itemize}
\item \texttt{index}:
\end{itemize}

a list with elements for each resampling iteration. Each list element is a vector of integers corresponding to the rows used for training at that iteration.

\begin{itemize}
\item \texttt{indexOut}:
\end{itemize}

a list (the same length as \texttt{index}) that dictates which data are held-out for each resample (as integers). If \texttt{NULL}, then the unique set of samples not contained in \texttt{index} is used.

\begin{itemize}
\item \texttt{indexFinal}:
\end{itemize}

an optional vector of integers indicating which samples are used to fit the final model after resampling. If \texttt{NULL}, then entire data set is used.

\begin{itemize}
\item \texttt{timingSamps}:
\end{itemize}

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).

\begin{itemize}
\item \texttt{predictionBounds}:
\end{itemize}

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 \texttt{c(TRUE, FALSE)} would only constrain the lower end of predictions. If numeric, specific bounds can be used. For example, if \texttt{c(10, NA)}, values below 10 would be predicted as 10 (with no constraint in the upper side).

\begin{itemize}
\item \texttt{seeds}:
\end{itemize}

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 \texttt{NA} will stop the seed from being set within the worker processes while a value of \texttt{NULL} will set the seeds using a random set of integers. Alternatively, a list can be used. The list should have \texttt{B+1} elements where \texttt{B} is the number of resamples, unless \texttt{method} is \texttt{"boot632"} in which case \texttt{B} is the number of resamples plus 1. The first \texttt{B} elements of the list should be vectors of integers of length \texttt{M} where \texttt{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.

\begin{itemize}
\item \texttt{adaptive}:
\end{itemize}

a list used when \texttt{method} is \texttt{"adaptive_cv"}, \texttt{"adaptive_boot"} or \texttt{"adaptive_LGOCV"}. See Details below.

\begin{itemize}
\item \texttt{trim}:
\end{itemize}

a logical. If \texttt{TRUE} the final model in \texttt{object$finalModel} may have some components of the object removed so reduce the size of the saved object. The \texttt{predict} method will still work, but some other features of the model may not work. \texttt{triming} will occur only for models where this feature has been implemented.

\begin{itemize}
\item \texttt{allowParallel}:
\end{itemize}

if a parallel backend is loaded and available, should the function use it?

\section*{Details}

When setting the seeds manually, the number of models being evaluated is required. This may not be obvious as \texttt{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 \texttt{seeds} arguments is longer than actually needed, no error is thrown.
Using method = "none" and specifying more than one model in train's tuneGrid or tuneLength arguments will result in an error.

Using adaptive resampling when method is either "adaptive_cv", "adaptive_boot" or "adaptive_LGOCV", the full set of resamples is not run for each model. As resampling continues, a futility analysis is conducted and models with a low probability of being optimal are removed. These features are experimental. See Kuhn (2014) for more details. The options for this procedure are:

- min: the minimum number of resamples used before models are removed
- alpha: the confidence level of the one-sided intervals used to measure futility
- method: either generalized least squares (method = "gls") or a Bradley-Terry model (method = "BT")
- complete: if a single parameter value is found before the end of resampling, should the full set of resamples be computed for that parameter.

The option search = "grid" uses the default grid search routine. When search = "random", a random search procedure is used (Bergstra and Bengio, 2012). See http://topepo.github.io/caret/random-hyperparameter-search.html for details and an example.

The supported bootstrap methods are:

- "boot": the usual bootstrap.
- "boot632": the 0.632 bootstrap estimator (Efron, 1983).
- "optimism_boot": the optimism bootstrap estimator. (Efron and Tibshirani, 1994).
- "boot_all": all of the above (for efficiency, but "boot" will be used for calculations).

The "boot632" method should not to be confused with the 0.632+ estimator proposed later by the same author.

Note that if index or indexOut are specified, the label shown by train may not be accurate since these arguments supersede the method argument.

Value

An echo of the parameters specified

Author(s)

Max Kuhn

References


Package website for subsampling: https://topepo.github.io/caret/subsampling-for-class-imbalances.html

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 = 5)
for(i in 1:5) seeds[[i]] <- sample.int(1000, 22)

## For the last model:
seeds[[5]] <- sample.int(1000, 1)

ctrl1 <- trainControl(method = "repeatedcv",
                      repeats = 5,
                      seeds = seeds)

set.seed(1)
mod1 <- train(Species ~ ., data = iris,
             method = "knn",
             tuneLength = 12,
             trControl = ctrl1)

ctrl2 <- trainControl(method = "adaptive_cv",
                      repeats = 5,
                      verboseIter = TRUE,
                      seeds = seeds)

set.seed(1)
mod2 <- train(Species ~ ., data = iris,
             method = "knn",
             tuneLength = 12,
             trControl = ctrl2)

## End(Not run)

```

---

<table>
<thead>
<tr>
<th><strong>train_model_list</strong></th>
<th>A List of Available Models in train</th>
</tr>
</thead>
</table>

---
Description

These models are included in the package via wrappers for train. Custom models can also be created. See the URL below.

**AdaBoost Classification Trees** (method = 'adaboost')

For classification using package **fastAdaboost** with tuning parameters:

- Number of Trees (nIter, numeric)
- Method (method, character)

**AdaBoost.M1**( method = 'AdaBoost.M1')

For classification using packages **adabag** and **plyr** with tuning parameters:

- Number of Trees (mfinal, numeric)
- Max Tree Depth (maxdepth, numeric)
- Coefficient Type (coeflearn, character)

**Adaptive Mixture Discriminant Analysis** (method = 'amdai')

For classification using package **adaptDA** with tuning parameters:

- Model Type (model, character)

**Adaptive-Network-Based Fuzzy Inference System** (method = 'ANFIS')

For regression using package **frbs** with tuning parameters:

- Number of Fuzzy Terms (num.labels, numeric)
- Max. Iterations (max.iter, numeric)

**Adjacent Categories Probability Model for Ordinal Data** (method = 'vglmAdjCat')

For classification using package **VGAM** with tuning parameters:

- Parallel Curves (parallel, logical)
- Link Function (link, character)

**Bagged AdaBoost** (method = 'AdaBag')

For classification using packages **adabag** and **plyr** with tuning parameters:

- Number of Trees (mfinal, numeric)
- Max Tree Depth (maxdepth, numeric)

**Bagged CART** (method = 'treebag')

For classification and regression using packages **ipred, plyr** and **e1071** with no tuning parameters.

**Bagged FDA using gCV Pruning** (method = 'bagFDAGCV')

For classification using package **earth** with tuning parameters:

- Product Degree (degree, numeric)
Note: Unlike other packages used by train, the earth package is fully loaded when this model is used.

**Bagged Flexible Discriminant Analysis** *(method = 'bagFDA')*

For classification using packages earth and mda with tuning parameters:

- Product Degree (degree, numeric)
- Number of Terms (nprune, numeric)

Note: Unlike other packages used by train, the earth package is fully loaded when this model is used.

**Bagged Logic Regression** *(method = 'logicBag')*

For classification and regression using package logicFS with tuning parameters:

- Maximum Number of Leaves (nleaves, numeric)
- Number of Trees (ntrees, numeric)

Note: Unlike other packages used by train, the logicFS package is fully loaded when this model is used.

**Bagged MARS** *(method = 'bagEarth')*

For classification and regression using package earth with tuning parameters:

- Number of Terms (nprune, numeric)
- Product Degree (degree, numeric)

Note: Unlike other packages used by train, the earth package is fully loaded when this model is used.

**Bagged MARS using gCV Pruning** *(method = 'bagEarthGCV')*

For classification and regression using package earth with tuning parameters:

- Product Degree (degree, numeric)

Note: Unlike other packages used by train, the earth package is fully loaded when this model is used.

**Bagged Model** *(method = 'bag')*

For classification and regression using package caret with tuning parameters:

- Number of Randomly Selected Predictors (vars, numeric)

**Bayesian Additive Regression Trees** *(method = 'bartMachine')*

For classification and regression using package bartMachine with tuning parameters:

- Number of Trees (num_trees, numeric)
- Prior Boundary (k, numeric)
- Base Terminal Node Hyperparameter (alpha, numeric)
- Power Terminal Node Hyperparameter (beta, numeric)
- Degrees of Freedom (nu, numeric)
Bayesian Generalized Linear Model (method = 'bayesglm')
For classification and regression using package arm with no tuning parameters.

Bayesian Regularized Neural Networks (method = 'brnn')
For regression using package brnn with tuning parameters:
  • Number of Neurons (neurons, numeric)

Bayesian Ridge Regression (method = 'bridge')
For regression using package monomvn with no tuning parameters.

Bayesian Ridge Regression (Model Averaged) (method = 'blassoAveraged')
For regression using package monomvn with no tuning parameters.
Note: This model makes predictions by averaging the predictions based on the posterior estimates of the regression coefficients. While it is possible that some of these posterior estimates are zero for non-informative predictors, the final predicted value may be a function of many (or even all) predictors.

Binary Discriminant Analysis (method = 'binda')
For classification using package binda with tuning parameters:
  • Shrinkage Intensity (lambda.freqs, numeric)

Boosted Classification Trees (method = 'ada')
For classification using packages ada and plyr with tuning parameters:
  • Number of Trees (iter, numeric)
  • Max Tree Depth (maxdepth, numeric)
  • Learning Rate (nu, numeric)

Boosted Generalized Additive Model (method = 'gamboost')
For classification and regression using packages mboost, plyr and import with tuning parameters:
  • Number of Boosting Iterations (mstop, numeric)
  • AIC Prune? (prune, character)
Note: The prune option for this model enables the number of iterations to be determined by the optimal AIC value across all iterations. See the examples in ?mboost::mstop. If pruning is not used, the ensemble makes predictions using the exact value of the mstop tuning parameter value.

Boosted Generalized Linear Model (method = 'glmboost')
For classification and regression using packages plyr and mboost with tuning parameters:
  • Number of Boosting Iterations (mstop, numeric)
  • AIC Prune? (prune, character)
Note: The prune option for this model enables the number of iterations to be determined by the optimal AIC value across all iterations. See the examples in ?mboost::mstop. If pruning is not used, the ensemble makes predictions using the exact value of the mstop tuning parameter value.

Boosted Linear Model (method = 'BstLm')
For classification and regression using packages bst and plyr with tuning parameters:
• Number of Boosting Iterations (mstop, numeric)
• Shrinkage (nu, numeric)

**Boosted Logistic Regression** (method = 'LogitBoost')
For classification using package caTools with tuning parameters:
• Number of Boosting Iterations (nIter, numeric)

**Boosted Smoothing Spline** (method = 'bstSm')
For classification and regression using packages bst and plyr with tuning parameters:
• Number of Boosting Iterations (mstop, numeric)
• Shrinkage (nu, numeric)

**Boosted Tree** (method = 'blackboost')
For classification and regression using packages party, mboost and plyr with tuning parameters:
• Number of Trees (mstop, numeric)
• Max Tree Depth (maxdepth, numeric)

**Boosted Tree** (method = 'bstTree')
For classification and regression using packages bst and plyr with tuning parameters:
• Number of Boosting Iterations (mstop, numeric)
• Max Tree Depth (maxdepth, numeric)
• Shrinkage (nu, numeric)

**C4.5-like Trees** (method = 'J48')
For classification using package RWeka with tuning parameters:
• Confidence Threshold (C, numeric)
• Minimum Instances Per Leaf (M, numeric)

**C5.0** (method = 'C5.0')
For classification using packages C50 and plyr with tuning parameters:
• Number of Boosting Iterations (trials, numeric)
• Model Type (model, character)
• Winnow (winnow, logical)

**CART** (method = 'rpart')
For classification and regression using package rpart with tuning parameters:
• Complexity Parameter (cp, numeric)
**CART** (method = 'rpart1SE')
For classification and regression using package **rpart** with no tuning parameters.

Note: This CART model replicates the same process used by the rpart function where the model complexity is determined using the one-standard error method. This procedure is replicated inside of the resampling done by **train** so that an external resampling estimate can be obtained.

**CART** (method = 'rpart2')
For classification and regression using package **rpart** with tuning parameters:

- Max Tree Depth (maxdepth, numeric)

**CART or Ordinal Responses** (method = 'rpartScore')
For classification using packages **rpartScore** and **plyr** with tuning parameters:

- Complexity Parameter (cp, numeric)
- Split Function (split, character)
- Pruning Measure (prune, character)

**CHi-squared Automated Interaction Detection** (method = 'chaid')
For classification using package **CHAID** with tuning parameters:

- Merging Threshold (alpha2, numeric)
- Splitting former Merged Threshold (alpha3, numeric)
- Splitting former Merged Threshold (alpha4, numeric)

**Conditional Inference Random Forest** (method = 'cforest')
For classification and regression using package **party** with tuning parameters:

- Number of Randomly Selected Predictors (mtry, numeric)

**Conditional Inference Tree** (method = 'ctree')
For classification and regression using package **party** with tuning parameters:

- 1 - P-Value Threshold (mincriterion, numeric)

**Conditional Inference Tree** (method = 'ctree2')
For classification and regression using package **party** with tuning parameters:

- Max Tree Depth (maxdepth, numeric)
- 1 - P-Value Threshold (mincriterion, numeric)

**Continuation Ratio Model for Ordinal Data** (method = 'vglmContRatio')
For classification using package **VGAM** with tuning parameters:

- Parallel Curves (parallel, logical)
- Link Function (link, character)

**Cost-Sensitive C5.0** (method = 'C5.0Cost')
For classification using packages **C50** and **plyr** with tuning parameters:
- Number of Boosting Iterations (trials, numeric)
- Model Type (model, character)
- Winnow (winnow, logical)
- Cost (cost, numeric)

**Cost-Sensitive CART** (method = 'rpartCost')
For classification using packages rpart and plyr with tuning parameters:
- Complexity Parameter (cp, numeric)
- Cost (Cost, numeric)

**Cubist** (method = 'cubist')
For regression using package Cubist with tuning parameters:
- Number of Committees (committees, numeric)
- Number of Instances (neighbors, numeric)

**Cumulative Probability Model for Ordinal Data** (method = 'vglmCumulative')
For classification using package VGAM with tuning parameters:
- Parallel Curves (parallel, logical)
- Link Function (link, character)

**DeepBoost** (method = 'deepboost')
For classification using package deepboost with tuning parameters:
- Number of Boosting Iterations (num_iter, numeric)
- Tree Depth (tree_depth, numeric)
- L1 Regularization (beta, numeric)
- Tree Depth Regularization (lambda, numeric)
- Loss (loss_type, character)

**Diagonal Discriminant Analysis** (method = 'dda')
For classification using package sparsediscrim with tuning parameters:
- Model (model, character)
- Shrinkage Type (shrinkage, character)

**Distance Weighted Discrimination with Polynomial Kernel** (method = 'dwdPoly')
For classification using package kerndwd with tuning parameters:
- Regularization Parameter (lambda, numeric)
- q (qval, numeric)
- Polynomial Degree (degree, numeric)
- Scale (scale, numeric)
Distance Weighted Discrimination with Radial Basis Function Kernel (method = 'dwdRadial')
For classification using packages kernlab and kerndwd with tuning parameters:

- Regularization Parameter (lambda, numeric)
- q (qval, numeric)
- Sigma (sigma, numeric)

Dynamic Evolving Neural-Fuzzy Inference System (method = 'DENFIS')
For regression using package frbs with tuning parameters:

- Threshold (dthr, numeric)
- Max. Iterations (max.iter, numeric)

Elasticnet (method = 'enet')
For regression using package elasticnet with tuning parameters:

- Fraction of Full Solution (fraction, numeric)
- Weight Decay (lambda, numeric)

Ensembles of Generalized Linear Models (method = 'randomGLM')
For classification and regression using package randomGLM with tuning parameters:

- Interaction Order (maxInteractionOrder, numeric)

Note: Unlike other packages used by train, the randomGLM package is fully loaded when this model is used.

eXtreme Gradient Boosting (method = 'xgbDART')
For classification and regression using packages xgboost and plyr with tuning parameters:

- Number of Boosting Iterations (nrounds, numeric)
- Max Tree Depth (max_depth, numeric)
- Shrinkage (eta, numeric)
- Minimum Loss Reduction (gamma, numeric)
- Subsample Percentage (subsample, numeric)
- Subsample Ratio of Columns (colsample_bytree, numeric)
- Fraction of Trees Dropped (rate_drop, numeric)
- Prob. of Skipping Drop-out (skip_drop, numeric)
- Minimum Sum of Instance Weight (min_child_weight, numeric)

eXtreme Gradient Boosting (method = 'xgbLinear')
For classification and regression using package xgboost with tuning parameters:

- Number of Boosting Iterations (nrounds, numeric)
- L2 Regularization (lambda, numeric)
- L1 Regularization (alpha, numeric)
- Learning Rate (eta, numeric)
eXtreme Gradient Boosting (method = 'xgbTree')
For classification and regression using packages xgboost and plyr with tuning parameters:
  • Number of Boosting Iterations (nrounds, numeric)
  • Max Tree Depth (max_depth, numeric)
  • Shrinkage (eta, numeric)
  • Minimum Loss Reduction (gamma, numeric)
  • Subsample Ratio of Columns (colsample_bytree, numeric)
  • Minimum Sum of Instance Weight (min_child_weight, numeric)
  • Subsample Percentage (subsample, numeric)

Extreme Learning Machine (method = 'elm')
For classification and regression using package elmNN with tuning parameters:
  • Number of Hidden Units (nhid, numeric)
  • Activation Function (actfun, character)

Factor-Based Linear Discriminant Analysis (method = 'RFlda')
For classification using package HiDimDA with tuning parameters:
  • Number of Factors (q, numeric)

Flexible Discriminant Analysis (method = 'fda')
For classification using packages earth and mda with tuning parameters:
  • Product Degree (degree, numeric)
  • Number of Terms (nprune, numeric)

Note: Unlike other packages used by train, the earth package is fully loaded when this model is used.

Fuzzy Inference Rules by Descent Method (method = 'FIR.DM')
For regression using package frbs with tuning parameters:
  • Number of Fuzzy Terms (num.labels, numeric)
  • Max. Iterations (max.iter, numeric)

Fuzzy Rules Using Chi’s Method (method = 'FRBCS.CHI')
For classification using package frbs with tuning parameters:
  • Number of Fuzzy Terms (num.labels, numeric)
  • Membership Function (type.mf, character)

Fuzzy Rules Using Genetic Cooperative-Competitive Learning and Pittsburgh (method = 'FH.GBML')
For classification using package frbs with tuning parameters:
  • Max. Number of Rules (max.num.rule, numeric)
  • Population Size (popu.size, numeric)
• Max. Generations (max_gen, numeric)

**Fuzzy Rules Using the Structural Learning Algorithm on Vague Environment** *(method = 'SLAVE')*

For classification using package `frbs` with tuning parameters:

• Number of Fuzzy Terms (num_labels, numeric)
• Max. Iterations (max_iter, numeric)
• Max. Generations (max_gen, numeric)

**Fuzzy Rules via MOGUL** *(method = 'GFS.FR.MOGUL')*

For regression using package `frbs` with tuning parameters:

• Max. Generations (max_gen, numeric)
• Max. Iterations (max_iter, numeric)
• Max. Tuning Iterations (max_tune, numeric)

**Fuzzy Rules via Thrift** *(method = 'GFS.THRIFT')*

For regression using package `frbs` with tuning parameters:

• Population Size (popu_size, numeric)
• Number of Fuzzy Labels (num_labels, numeric)
• Max. Generations (max_gen, numeric)

**Fuzzy Rules with Weight Factor** *(method = 'FRBCS.W')*

For classification using package `frbs` with tuning parameters:

• Number of Fuzzy Terms (num_labels, numeric)
• Membership Function (type_mf, character)

**Gaussian Process** *(method = 'gaussprLinear')*

For classification and regression using package `kernlab` with no tuning parameters.

**Gaussian Process with Polynomial Kernel** *(method = 'gaussprPoly')*

For classification and regression using package `kernlab` with tuning parameters:

• Polynomial Degree (degree, numeric)
• Scale (scale, numeric)

**Gaussian Process with Radial Basis Function Kernel** *(method = 'gaussprRadial')*

For classification and regression using package `kernlab` with tuning parameters:

• Sigma (sigma, numeric)

**Generalized Additive Model using LOESS** *(method = 'gamLoess')*

For classification and regression using package `gam` with tuning parameters:

• Span (span, numeric)
• Degree (degree, numeric)
Note: Which terms enter the model in a nonlinear manner is determined by the number of unique values for the predictor. For example, if a predictor only has four unique values, most basis expansion method will fail because there are not enough granularity in the data. By default, a predictor must have at least 10 unique values to be used in a nonlinear basis expansion. Unlike other packages used by train, the gam package is fully loaded when this model is used.

**Generalized Additive Model using Splines** (method = 'bam')

For classification and regression using package mgcv with tuning parameters:

- Feature Selection (select, logical)
- Method (method, character)

Note: Which terms enter the model in a nonlinear manner is determined by the number of unique values for the predictor. For example, if a predictor only has four unique values, most basis expansion method will fail because there are not enough granularity in the data. By default, a predictor must have at least 10 unique values to be used in a nonlinear basis expansion. Unlike other packages used by train, the mgcv package is fully loaded when this model is used.

**Generalized Additive Model using Splines** (method = 'gam')

For classification and regression using package mgcv with tuning parameters:

- Feature Selection (select, logical)
- Method (method, character)

Note: Which terms enter the model in a nonlinear manner is determined by the number of unique values for the predictor. For example, if a predictor only has four unique values, most basis expansion method will fail because there are not enough granularity in the data. By default, a predictor must have at least 10 unique values to be used in a nonlinear basis expansion. Unlike other packages used by train, the mgcv package is fully loaded when this model is used.

**Generalized Additive Model using Splines** (method = 'gamSpline')

For classification and regression using package gam with tuning parameters:

- Degrees of Freedom (df, numeric)

Note: Which terms enter the model in a nonlinear manner is determined by the number of unique values for the predictor. For example, if a predictor only has four unique values, most basis expansion method will fail because there are not enough granularity in the data. By default, a predictor must have at least 10 unique values to be used in a nonlinear basis expansion. Unlike other packages used by train, the gam package is fully loaded when this model is used.

**Generalized Linear Model** (method = 'glm')

For classification and regression with no tuning parameters.

**Generalized Linear Model with Stepwise Feature Selection** (method = 'glmStepAIC')

For classification and regression using package MASS with no tuning parameters.

**Generalized Partial Least Squares** (method = 'gpls')

For classification using package gpls with tuning parameters:

- Number of Components (K, prov, numeric)
Genetic Lateral Tuning and Rule Selection of Linguistic Fuzzy Systems (method = 'GFS.LT.RS')

For regression using package frbs with tuning parameters:

- Population Size (popu.size, numeric)
- Number of Fuzzy Labels (num.labels, numeric)
- Max. Generations (max.gen, numeric)

glmnet (method = 'glmnet_h2o')

For classification and regression using package h2o with tuning parameters:

- Mixing Percentage (alpha, numeric)
- Regularization Parameter (lambda, numeric)

glmnet (method = 'glmnet')

For classification and regression using packages glmnet and Matrix with tuning parameters:

- Mixing Percentage (alpha, numeric)
- Regularization Parameter (lambda, numeric)

Gradient Boosting Machines (method = 'gbm_h2o')

For classification and regression using package h2o with tuning parameters:

- Number of Boosting Iterations (ntrees, numeric)
- Max Tree Depth (max_depth, numeric)
- Min. Terminal Node Size (min_rows, numeric)
- Shrinkage (learn_rate, numeric)
- Number of Randomly Selected Predictors (col_sample_rate, numeric)

Greedy Prototype Selection (method = 'protoclass')

For classification using packages proxy and proto class with tuning parameters:

- Ball Size (eps, numeric)
- Distance Order (Minkowski, numeric)

Heteroscedastic Discriminant Analysis (method = 'hda')

For classification using package hda with tuning parameters:

- Gamma (gamma, numeric)
- Lambda (lambda, numeric)
- Dimension of the Discriminative Subspace (newdim, numeric)

High Dimensional Discriminant Analysis (method = 'hdda')

For classification using package HDclassif with tuning parameters:

- Threshold (threshold, character)
- Model Type (model, numeric)
High-Dimensional Regularized Discriminant Analysis (method = 'hdrda')
For classification using package sparsediscrim with tuning parameters:
  • Gamma (gamma, numeric)
  • Lambda (lambda, numeric)
  • Shrinkage Type (shrinkage_type, character)

Hybrid Neural Fuzzy Inference System (method = 'HYFIS')
For regression using package frbs with tuning parameters:
  • Number of Fuzzy Terms (num.labels, numeric)
  • Max. Iterations (max.iter, numeric)

Independent Component Regression (method = 'icr')
For regression using package fastICA with tuning parameters:
  • Number of Components (n.comp, numeric)

k-Nearest Neighbors (method = 'knn')
For classification and regression using package kknn with tuning parameters:
  • Max. Number of Neighbors (kmax, numeric)
  • Distance (distance, numeric)
  • Kernel (kernel, character)

k-Nearest Neighbors (method = 'knn')
For classification and regression with tuning parameters:
  • Number of Neighbors (k, numeric)

L2 Regularized Linear Support Vector Machines with Class Weights (method = 'svmLinearWeights2')
For classification using package LiblineaR with tuning parameters:
  • Cost (cost, numeric)
  • Loss Function (Loss, character)
  • Class Weight (weight, numeric)

L2 Regularized Support Vector Machine (dual) with Linear Kernel (method = 'svmLinear3')
For classification and regression using package LiblineaR with tuning parameters:
  • Cost (cost, numeric)
  • Loss Function (Loss, character)

Learning Vector Quantization (method = 'lvq')
For classification using package class with tuning parameters:
  • Codebook Size (size, numeric)
  • Number of Prototypes (k, numeric)
Least Angle Regression (method = 'lars')
For regression using package lars with tuning parameters:
• Fraction (fraction, numeric)

Least Angle Regression (method = 'lars2')
For regression using package lars with tuning parameters:
• Number of Steps (step, numeric)

Least Squares Support Vector Machine (method = 'lssvmLinear')
For classification using package kernlab with tuning parameters:
• Regularization Parameter (tau, numeric)

Least Squares Support Vector Machine with Polynomial Kernel (method = 'lssvmPoly')
For classification using package kernlab with tuning parameters:
• Polynomial Degree (degree, numeric)
• Scale (scale, numeric)
• Regularization Parameter (tau, numeric)

Least Squares Support Vector Machine with Radial Basis Function Kernel (method = 'lssvmRadial')
For classification using package kernlab with tuning parameters:
• Sigma (sigma, numeric)
• Regularization Parameter (tau, numeric)

Linear Discriminant Analysis (method = 'lda')
For classification using package MASS with no tuning parameters.

Linear Discriminant Analysis (method = 'lda2')
For classification using package MASS with tuning parameters:
• Number of Discriminant Functions (dimen, numeric)

Linear Discriminant Analysis with Stepwise Feature Selection (method = 'stepLDA')
For classification using packages klaR and MASS with tuning parameters:
• Maximum Number of Variables (maxvar, numeric)
• Search Direction (direction, character)

Linear Distance Weighted Discrimination (method = 'dwdLinear')
For classification using package kerndwd with tuning parameters:
• Regularization Parameter (lambda, numeric)
• q (qval, numeric)

Linear Regression (method = 'lm')
For regression with tuning parameters:
• intercept (intercept, logical)

**Linear Regression with Backwards Selection** (method = 'leapBackward')
For regression using package **leaps** with tuning parameters:
  • Maximum Number of Predictors (nvmax, numeric)

**Linear Regression with Forward Selection** (method = 'leapForward')
For regression using package **leaps** with tuning parameters:
  • Maximum Number of Predictors (nvmax, numeric)

**Linear Regression with Stepwise Selection** (method = 'leapSeq')
For regression using package **leaps** with tuning parameters:
  • Maximum Number of Predictors (nvmax, numeric)

**Linear Regression with Stepwise Selection** (method = 'lmStepAIC')
For regression using package **MASS** with no tuning parameters.

**Linear Support Vector Machines with Class Weights** (method = 'svmLinearWeights')
For classification using package **e1071** with tuning parameters:
  • Cost (cost, numeric)
  • Class Weight (weight, numeric)

**Localized Linear Discriminant Analysis** (method = 'loclda')
For classification using package **klaR** with tuning parameters:
  • Number of Nearest Neighbors (k, numeric)

**Logic Regression** (method = 'logreg')
For classification and regression using package **LogicReg** with tuning parameters:
  • Maximum Number of Leaves (treesize, numeric)
  • Number of Trees (ntrees, numeric)

**Logistic Model Trees** (method = 'LMT')
For classification using package **RWeka** with tuning parameters:
  • Number of Iteratons (iter, numeric)

**Maximum Uncertainty Linear Discriminant Analysis** (method = 'Mlda')
For classification using package **HiDimDA** with no tuning parameters.

**Mixture Discriminant Analysis** (method = 'mda')
For classification using package **mda** with tuning parameters:
  • Number of Subclasses Per Class (subclasses, numeric)

**Model Averaged Naive Bayes Classifier** (method = 'manb')
For classification using package **bnclassify** with tuning parameters:
• Smoothing Parameter (smooth, numeric)
• Prior Probability (prior, numeric)

Model Averaged Neural Network (method = 'avNNet')
For classification and regression using package nnet with tuning parameters:
• Number of Hidden Units (size, numeric)
• Weight Decay (decay, numeric)
• Bagging (bag, logical)

Model Rules (method = 'M5Rules')
For regression using package RWeka with tuning parameters:
• Pruned (pruned, character)
• Smoothed (smoothed, character)

Model Tree (method = 'M5')
For regression using package RWeka with tuning parameters:
• Pruned (pruned, character)
• Smoothed (smoothed, character)
• Rules (rules, character)

Monotone Multi-Layer Perceptron Neural Network (method = 'monmlp')
For classification and regression using package monmlp with tuning parameters:
• Number of Hidden Units (hidden1, numeric)
• Number of Models (n.ensemble, numeric)

Multi-Layer Perceptron (method = 'mlp')
For classification and regression using package RSNNS with tuning parameters:
• Number of Hidden Units (size, numeric)

Multi-Layer Perceptron (method = 'mlpWeightDecay')
For classification and regression using package RSNNS with tuning parameters:
• Number of Hidden Units (size, numeric)
• Weight Decay (decay, numeric)

Multi-Layer Perceptron, multiple layers (method = 'mlpWeightDecayML')
For classification and regression using package RSNNS with tuning parameters:
• Number of Hidden Units layer1 (layer1, numeric)
• Number of Hidden Units layer2 (layer2, numeric)
• Number of Hidden Units layer3 (layer3, numeric)
• Weight Decay (decay, numeric)
Multi-Layer Perceptron, with multiple layers (method = 'mlpM')
For classification and regression using package RSNNS with tuning parameters:

- Number of Hidden Units layer1 (layer1, numeric)
- Number of Hidden Units layer2 (layer2, numeric)
- Number of Hidden Units layer3 (layer3, numeric)

Multi-Step Adaptive MCP-Net (method = 'msaenet')
For classification and regression using package msaenet with tuning parameters:

- Alpha (alphas, numeric)
- Number of Adaptive Estimation Steps (nsteps, numeric)
- Adaptive Weight Scaling Factor (scale, numeric)

Multilayer Perceptron Network by Stochastic Gradient Descent (method = 'mlpSGD')
For classification and regression using packages FCNN4R and plyr with tuning parameters:

- Number of Hidden Units (size, numeric)
- L2 Regularization (l2reg, numeric)
- RMSE Gradient Scaling (lambda, numeric)
- Learning Rate (learn_rate, numeric)
- Momentum (momentum, numeric)
- Learning Rate Decay (gamma, numeric)
- Batch Size (minibatchsz, numeric)
- Number of Models (repeats, numeric)

Multilayer Perceptron Network with Dropout (method = 'mlpKerasDropout')
For classification and regression using package keras with tuning parameters:

- Number of Hidden Units (size, numeric)
- Dropout Rate (dropout, numeric)
- Batch Size (batch_size, numeric)
- Learning Rate (lr, numeric)
- Rho (rho, numeric)
- Learning Rate Decay (decay, numeric)
- Activation Function (activation, character)

Note: After train completes, the keras model object is serialized so that it can be used between R session. When predicting, the code will temporarily unsearalize the object. To make the predictions more efficient, the user might want to use keras::unsearalize_model(object$finalModel$object) in the current R session so that that operation is only done once. Also, this model cannot be run in parallel due to the nature of how tensorflow does the computations. Unlike other packages used by train, the dplyr package is fully loaded when this model is used.

Multilayer Perceptron Network with Dropout (method = 'mlpKerasDropoutCost')
For classification using package keras with tuning parameters:
train_model_list

- Number of Hidden Units (size, numeric)
- Dropout Rate (dropout, numeric)
- Batch Size (batch_size, numeric)
- Learning Rate (lr, numeric)
- Rho (rho, numeric)
- Learning Rate Decay (decay, numeric)
- Cost (cost, numeric)
- Activation Function (activation, character)

Note: After train completes, the keras model object is serialized so that it can be used between R session. When predicting, the code will temporarily unsearalize the object. To make the predictions more efficient, the user might want to use keras::unsearalize_model(object$finalModel$object) in the current R session so that that operation is only done once. Also, this model cannot be run in parallel due to the nature of how tensorflow does the computations. Finally, the cost parameter weights the first class in the outcome vector. Unlike other packages used by train, the dplyr package is fully loaded when this model is used.

**Multilayer Perceptron Network with Weight Decay** (method = 'mlpKerasDecay')

For classification and regression using package keras with tuning parameters:

- Number of Hidden Units (size, numeric)
- L2 Regularization (lambda, numeric)
- Batch Size (batch_size, numeric)
- Learning Rate (lr, numeric)
- Rho (rho, numeric)
- Learning Rate Decay (decay, numeric)
- Activation Function (activation, character)

Note: After train completes, the keras model object is serialized so that it can be used between R session. When predicting, the code will temporarily unsearalize the object. To make the predictions more efficient, the user might want to use keras::unsearalize_model(object$finalModel$object) in the current R session so that that operation is only done once. Also, this model cannot be run in parallel due to the nature of how tensorflow does the computations. Unlike other packages used by train, the dplyr package is fully loaded when this model is used.

**Multilayer Perceptron Network with Weight Decay** (method = 'mlpKerasDecayCost')

For classification using package keras with tuning parameters:

- Number of Hidden Units (size, numeric)
- L2 Regularization (lambda, numeric)
- Batch Size (batch_size, numeric)
- Learning Rate (lr, numeric)
- Rho (rho, numeric)
- Learning Rate Decay (decay, numeric)
- Cost (cost, numeric)
• Activation Function (activation, character)

Note: After train completes, the keras model object is serialized so that it can be used between R
session. When predicting, the code will temporarily unsearalize the object. To make the predictions
more efficient, the user might want to use keras::unsearalize_model(object$finalModel$object)
in the current R session so that that operation is only done once. Also, this model cannot be run
in parallel due to the nature of how tensorflow does the computations. Finally, the cost parameter
weights the first class in the outcome vector. Unlike other packages used by train, the dplyr
package is fully loaded when this model is used.

**Multivariate Adaptive Regression Spline** (method = 'earth')

For classification and regression using package earth with tuning parameters:

• Number of Terms (nprune, numeric)
• Product Degree (degree, numeric)

Note: Unlike other packages used by train, the earth package is fully loaded when this model is
used.

**Multivariate Adaptive Regression Spline**s (method = 'gcvEarth')

For classification and regression using package earth with tuning parameters:

• Product Degree (degree, numeric)

Note: Unlike other packages used by train, the earth package is fully loaded when this model is
used.

**Naive Bayes** (method = 'naive_bayes')

For classification using package naivebayes with tuning parameters:

• Laplace Correction (laplace, numeric)
• Distribution Type (usekernel, logical)
• Bandwidth Adjustment (adjust, numeric)

**Naive Bayes** (method = 'nb')

For classification using package klaR with tuning parameters:

• Laplace Correction (fl, numeric)
• Distribution Type (usekernel, logical)
• Bandwidth Adjustment (adjust, numeric)

**Naive Bayes Classifier** (method = 'nbDiscrete')

For classification using package bnclassify with tuning parameters:

• Smoothing Parameter (smooth, numeric)

**Naive Bayes Classifier with Attribute Weighting** (method = 'awnb')

For classification using package bnclassify with tuning parameters:

• Smoothing Parameter (smooth, numeric)
Nearest Shrunken Centroids (method = 'pam')
For classification using package pamr with tuning parameters:
  • Shrinkage Threshold (threshold, numeric)

Negative Binomial Generalized Linear Model (method = 'glm.nb')
For regression using package MASS with tuning parameters:
  • Link Function (link, character)

Neural Network (method = 'mxnet')
For classification and regression using package mxnet with tuning parameters:
  • Number of Hidden Units in Layer 1 (layer1, numeric)
  • Number of Hidden Units in Layer 2 (layer2, numeric)
  • Number of Hidden Units in Layer 3 (layer3, numeric)
  • Learning Rate (learning_rate, numeric)
  • Momentum (momentum, numeric)
  • Dropout Rate (dropout, numeric)
  • Activation Function (activation, character)

Note: The mxnet package is not yet on CRAN. See http://mxnet.io for installation instructions.

Neural Network (method = 'mxnetAdam')
For classification and regression using package mxnet with tuning parameters:
  • Number of Hidden Units in Layer 1 (layer1, numeric)
  • Number of Hidden Units in Layer 2 (layer2, numeric)
  • Number of Hidden Units in Layer 3 (layer3, numeric)
  • Dropout Rate (dropout, numeric)
  • beta1 (beta1, numeric)
  • beta2 (beta2, numeric)
  • Learning Rate (learning_rate, numeric)
  • Activation Function (activation, character)

Note: The mxnet package is not yet on CRAN. See http://mxnet.io for installation instructions.
Users are strongly advised to define num.round themselves.

Neural Network (method = 'neuralnet')
For regression using package neuralnet with tuning parameters:
  • Number of Hidden Units in Layer 1 (layer1, numeric)
  • Number of Hidden Units in Layer 2 (layer2, numeric)
  • Number of Hidden Units in Layer 3 (layer3, numeric)

Neural Network (method = 'nnet')
For classification and regression using package nnet with tuning parameters:
• Number of Hidden Units (size, numeric)
• Weight Decay (decay, numeric)

**Neural Networks with Feature Extraction** (method = 'pcaNNet')
For classification and regression using package *nnet* with tuning parameters:

• Number of Hidden Units (size, numeric)
• Weight Decay (decay, numeric)

**Non-Convex Penalized Quantile Regression** (method = 'rqnc')
For regression using package *rqPen* with tuning parameters:

• L1 Penalty (lambda, numeric)
• Penalty Type (penalty, character)

**Non-Informative Model** (method = 'null')
For classification and regression with no tuning parameters.
Note: Since this model always predicts the same value, R-squared values will always be estimated to be NA.

**Non-Negative Least Squares** (method = 'nnls')
For regression using package *nnls* with no tuning parameters.

**Oblique Random Forest** (method = 'ORFlog')
For classification using package *obliqueRF* with tuning parameters:

• Number of Randomly Selected Predictors (mtry, numeric)
Note: Unlike other packages used by train, the *obliqueRF* package is fully loaded when this model is used.

**Oblique Random Forest** (method = 'ORFpls')
For classification using package *obliqueRF* with tuning parameters:

• Number of Randomly Selected Predictors (mtry, numeric)
Note: Unlike other packages used by train, the *obliqueRF* package is fully loaded when this model is used.

**Oblique Random Forest** (method = 'ORFridge')
For classification using package *obliqueRF* with tuning parameters:

• Number of Randomly Selected Predictors (mtry, numeric)
Note: Unlike other packages used by train, the *obliqueRF* package is fully loaded when this model is used.

**Oblique Random Forest** (method = 'ORFsvm')
For classification using package *obliqueRF* with tuning parameters:

• Number of Randomly Selected Predictors (mtry, numeric)
Note: Unlike other packages used by \texttt{train}, the \texttt{obliqueRF} package is fully loaded when this model is used.

\textbf{Optimal Weighted Nearest Neighbor Classifier} (method = 'ownn')

For classification using package \texttt{snn} with tuning parameters:

- Number of Neighbors (K, numeric)

\textbf{Ordered Logistic or Probit Regression} (method = 'polr')

For classification using package \texttt{MASS} with tuning parameters:

- parameter (method, character)

\textbf{Parallel Random Forest} (method = 'parRF')

For classification and regression using packages \texttt{e1071, randomForest, foreach} and \texttt{import} with tuning parameters:

- Number of Randomly Selected Predictors (mtry, numeric)

\textbf{partDSA} (method = 'partDSA')

For classification and regression using package \texttt{partDSA} with tuning parameters:

- Number of Terminal Partitions (cut.off.growth, numeric)
- Minimum Percent Difference (MPD, numeric)

\textbf{Partial Least Squares} (method = 'kernelpls')

For classification and regression using package \texttt{pls} with tuning parameters:

- Number of Components (ncomp, numeric)

\textbf{Partial Least Squares} (method = 'pls')

For classification and regression using package \texttt{pls} with tuning parameters:

- Number of Components (ncomp, numeric)

\textbf{Partial Least Squares} (method = 'simpls')

For classification and regression using package \texttt{pls} with tuning parameters:

- Number of Components (ncomp, numeric)

\textbf{Partial Least Squares} (method = 'widekernelpls')

For classification and regression using package \texttt{pls} with tuning parameters:

- Number of Components (ncomp, numeric)

\textbf{Partial Least Squares Generalized Linear Models} (method = 'plsRglm')

For classification and regression using package \texttt{plsRglm} with tuning parameters:

- Number of PLS Components (nt, numeric)
- p-Value threshold (alpha.pvals.expli, numeric)
Note: Unlike other packages used by train, the plsRglm package is fully loaded when this model is used.

**Patient Rule Induction Method** (method = 'PRIM')
For classification using package supervisedPRIM with tuning parameters:
- peeling quantile (peel.alpha, numeric)
- pasting quantile (paste.alpha, numeric)
- minimum mass (mass.min, numeric)

**Penalized Discriminant Analysis** (method = 'pda')
For classification using package mda with tuning parameters:
- Shrinkage Penalty Coefficient (lambda, numeric)

**Penalized Discriminant Analysis** (method = 'pda2')
For classification using package mda with tuning parameters:
- Degrees of Freedom (df, numeric)

**Penalized Linear Discriminant Analysis** (method = 'PenalizedLDA')
For classification using packages penalizedLDA and plyr with tuning parameters:
- L1 Penalty (lambda, numeric)
- Number of Discriminant Functions (K, numeric)

**Penalized Linear Regression** (method = 'penalized')
For regression using package penalized with tuning parameters:
- L1 Penalty (lambda1, numeric)
- L2 Penalty (lambda2, numeric)

**Penalized Logistic Regression** (method = 'plr')
For classification using package stepPlr with tuning parameters:
- L2 Penalty (lambda, numeric)
- Complexity Parameter (cp, character)

**Penalized Multinomial Regression** (method = 'multinom')
For classification using package nnet with tuning parameters:
- Weight Decay (decay, numeric)

**Penalized Ordinal Regression** (method = 'ordinalNet')
For classification using packages ordinalNet and plyr with tuning parameters:
- Mixing Percentage (alpha, numeric)
- Selection Criterion (criteria, character)
- Link Function (link, character)
Note: Requires ordinalNet package version >= 2.0

**Polynomial Kernel Regularized Least Squares** (method = 'krlsPoly')
For regression using package KRLS with tuning parameters:
  - Regularization Parameter (lambda, numeric)
  - Polynomial Degree (degree, numeric)

**Principal Component Analysis** (method = 'pcr')
For regression using package pls with tuning parameters:
  - Number of Components (ncomp, numeric)

**Projection Pursuit Regression** (method = 'ppr')
For regression with tuning parameters:
  - Number of Terms (nterms, numeric)

**Quadratic Discriminant Analysis** (method = 'qda')
For classification using package MASS with no tuning parameters.

**Quadratic Discriminant Analysis with Stepwise Feature Selection** (method = 'stepQDA')
For classification using packages klaR and MASS with tuning parameters:
  - Maximum Number of Variables (maxvar, numeric)
  - Search Direction (direction, character)

**Quantile Random Forest** (method = 'qrf')
For regression using package quantregForest with tuning parameters:
  - Number of Randomly Selected Predictors (mtry, numeric)

**Quantile Regression Neural Network** (method = 'qrnn')
For regression using package qrnn with tuning parameters:
  - Number of Hidden Units (n.hidden, numeric)
  - Weight Decay (penalty, numeric)
  - Bagged Models? (bag, logical)

**Quantile Regression with LASSO penalty** (method = 'rqlasso')
For regression using package rqPen with tuning parameters:
  - L1 Penalty (lambda, numeric)

**Radial Basis Function Kernel Regularized Least Squares** (method = 'krlsRadial')
For regression using packages KRLS and kernlab with tuning parameters:
  - Regularization Parameter (lambda, numeric)
  - Sigma (sigma, numeric)
**Radial Basis Function Network** (method = 'rbf')
For classification and regression using package **RSNNS** with tuning parameters:
- Number of Hidden Units (size, numeric)

**Radial Basis Function Network** (method = 'rbfdDA')
For classification and regression using package **RSNNS** with tuning parameters:
- Activation Limit for Conflicting Classes (negativeThreshold, numeric)

**Random Ferns** (method = 'rFerns')
For classification using package **rFerns** with tuning parameters:
- Fern Depth (depth, numeric)

**Random Forest** (method = 'ranger')
For classification and regression using packages **e1071**, **ranger** and **dplyr** with tuning parameters:
- Number of Randomly Selected Predictors (mtry, numeric)
- Splitting Rule (splitrule, character)
- Minimal Node Size (min.node.size, numeric)

**Random Forest** (method = 'Rborist')
For classification and regression using package **Rborist** with tuning parameters:
- Number of Randomly Selected Predictors (predFixed, numeric)
- Minimal Node Size (minNode, numeric)

**Random Forest** (method = 'rf')
For classification and regression using package **randomForest** with tuning parameters:
- Number of Randomly Selected Predictors (mtry, numeric)

**Random Forest by Randomization** (method = 'extraTrees')
For classification and regression using package **extraTrees** with tuning parameters:
- Number of Randomly Selected Predictors (mtry, numeric)
- Number of Random Cuts (numRandomCuts, numeric)

**Random Forest Rule-Based Model** (method = 'rfRules')
For classification and regression using packages **randomForest**, **inTrees** and **plyr** with tuning parameters:
- Number of Randomly Selected Predictors (mtry, numeric)
- Maximum Rule Depth (maxdepth, numeric)

**Regularized Discriminant Analysis** (method = 'rda')
For classification using package **klaR** with tuning parameters:
- Gamma (gamma, numeric)
• Lambda (lambda, numeric)

**Regularized Linear Discriminant Analysis** *(method = 'rlda')*

For classification using package `sparsediscrim` with tuning parameters:

• Regularization Method (estimator, character)

**Regularized Logistic Regression** *(method = 'regLogistic')*

For classification using package `LiblineaR` with tuning parameters:

• Cost (cost, numeric)
• Loss Function (loss, character)
• Tolerance (epsilon, numeric)

**Regularized Random Forest** *(method = 'RRF')*

For classification and regression using packages `randomForest` and `RRF` with tuning parameters:

• Number of Randomly Selected Predictors (mtry, numeric)
• Regularization Value (coefReg, numeric)
• Importance Coefficient (coefImp, numeric)

**Regularized Random Forest** *(method = 'RRFglobal')*

For classification and regression using package `RRF` with tuning parameters:

• Number of Randomly Selected Predictors (mtry, numeric)
• Regularization Value (coefReg, numeric)

**Relaxed Lasso** *(method = 'relaxo')*

For regression using packages `relaxo` and `plyr` with tuning parameters:

• Penalty Parameter (lambda, numeric)
• Relaxation Parameter (phi, numeric)

**Relevance Vector Machines with Linear Kernel** *(method = 'rvmLinear')*

For regression using package `kernlab` with no tuning parameters.

**Relevance Vector Machines with Polynomial Kernel** *(method = 'rvmPoly')*

For regression using package `kernlab` with tuning parameters:

• Scale (scale, numeric)
• Polynomial Degree (degree, numeric)

**Relevance Vector Machines with Radial Basis Function Kernel** *(method = 'rvmRadial')*

For regression using package `kernlab` with tuning parameters:

• Sigma (sigma, numeric)

**Ridge Regression** *(method = 'ridge')*

For regression using package `elasticnet` with tuning parameters:
- Weight Decay (lambda, numeric)

**Ridge Regression with Variable Selection** (method = 'foba')
For regression using package `foba` with tuning parameters:
- Number of Variables Retained (k, numeric)
- L2 Penalty (lambda, numeric)

**Robust Linear Discriminant Analysis** (method = 'Linda')
For classification using package `rrcov` with no tuning parameters.

**Robust Linear Model** (method = 'rlm')
For regression using package `MASS` with tuning parameters:
- intercept (intercept, logical)
- psi (psi, character)

**Robust Mixture Discriminant Analysis** (method = 'rmda')
For classification using package `robustDA` with tuning parameters:
- Number of Subclasses Per Class (k, numeric)
- Model (model, character)

**Robust Quadratic Discriminant Analysis** (method = 'QdaCov')
For classification using package `rrcov` with no tuning parameters.

**Robust Regularized Linear Discriminant Analysis** (method = 'rrlda')
For classification using package `rrlda` with tuning parameters:
- Penalty Parameter (lambda, numeric)
- Robustness Parameter (hp, numeric)
- Penalty Type (penalty, character)

Note: Unlike other packages used by `train`, the `rrlda` package is fully loaded when this model is used.

**Robust SIMCA** (method = 'RSimca')
For classification using package `rrcovHD` with no tuning parameters.
Note: Unlike other packages used by `train`, the `rrcovHD` package is fully loaded when this model is used.

**ROC-Based Classifier** (method = 'rocc')
For classification using package `rocc` with tuning parameters:
- Number of Variables Retained (xgenes, numeric)

**Rotation Forest** (method = 'rotationForest')
For classification using package `rotationForest` with tuning parameters:
- Number of Variable Subsets (k, numeric)
• Ensemble Size (L, numeric)

**Rotation Forest** (method = 'rotationForestCp')

For classification using packages *rpart*, *plyr* and *rotationForest* with tuning parameters:

- Number of Variable Subsets (K, numeric)
- Ensemble Size (L, numeric)
- Complexity Parameter (cp, numeric)

**Rule-Based Classifier** (method = 'JRip')

For classification using package *RWeka* with tuning parameters:

- Number of Optimizations (numOpt, numeric)
- Number of Folds (numFolds, numeric)
- Min Weights (minWeights, numeric)

**Rule-Based Classifier** (method = 'PART')

For classification using package *RWeka* with tuning parameters:

- Confidence Threshold (threshold, numeric)
- Pruning (pruned, character)

**Self-Organizing Maps** (method = 'xyf')

For classification and regression using package *kohonen* with tuning parameters:

- Rows (xdim, numeric)
- Columns (ydim, numeric)
- Layer Weight (user.weights, numeric)
- Topology (topo, character)

Note: As of version 3.0.0 of the *kohonen* package, the argument user.weights replaces the old alpha parameter. user.weights is usually a vector of relative weights such as c(1, 3) but is parameterized here as a proportion such as c(1-.75, .75) where the .75 is the value of the tuning parameter passed to train and indicates that the outcome layer has 3 times the weight as the predictor layer.

**Semi-Naive Structure Learner Wrapper** (method = 'nbSearch')

For classification using package *bnclassify* with tuning parameters:

- Number of Folds (k, numeric)
- Minimum Absolute Improvement (epsilon, numeric)
- Smoothing Parameter (smooth, numeric)
- Final Smoothing Parameter (final_smooth, numeric)
- Search Direction (direction, character)

**Shrinkage Discriminant Analysis** (method = 'sda')

For classification using package *sda* with tuning parameters:
- Diagonalize (diagonal, logical)
- shrinkage (lambda, numeric)

SIMCA (method = 'Csimca')
For classification using packages rrcov and rrcovHD with no tuning parameters.

Simplified TSK Fuzzy Rules (method = 'FS.HGD')
For regression using package frbs with tuning parameters:
  - Number of Fuzzy Terms (num.labels, numeric)
  - Max. Iterations (max.iter, numeric)

Single C5.0 Ruleset (method = 'C5.0Rules')
For classification using package C50 with no tuning parameters.

Single C5.0 Tree (method = 'C5.0Tree')
For classification using package C50 with no tuning parameters.

Single Rule Classification (method = '0neR')
For classification using package RWeka with no tuning parameters.

Sparse Distance Weighted Discrimination (method = 'sdwd')
For classification using package sdwd with tuning parameters:
  - L1 Penalty (lambda, numeric)
  - L2 Penalty (lambda2, numeric)

Sparse Linear Discriminant Analysis (method = 'sparseLDA')
For classification using package sparseLDA with tuning parameters:
  - Number of Predictors (NumVars, numeric)
  - Lambda (lambda, numeric)

Sparse Mixture Discriminant Analysis (method = 'smda')
For classification using package sparseLDA with tuning parameters:
  - Number of Predictors (NumVars, numeric)
  - Lambda (lambda, numeric)
  - Number of Subclasses (r, numeric)

Sparse Partial Least Squares (method = 'spls')
For classification and regression using package spls with tuning parameters:
  - Number of Components (k, numeric)
  - Threshold (eta, numeric)
  - Kappa (kappa, numeric)

Spike and Slab Regression (method = 'spikeslab')
For regression using packages spikeslab and plyr with tuning parameters:
• Variables Retained (\texttt{vars}, numeric)

Note: Unlike other packages used by \texttt{train}, the \texttt{spikeslab} package is fully loaded when this model is used.

**Stabilized Linear Discriminant Analysis** (method = \texttt{`slda’})
For classification using package \texttt{ipred} with no tuning parameters.

**Stabilized Nearest Neighbor Classifier** (method = \texttt{`snn’})
For classification using package \texttt{snn} with tuning parameters:
  • Stabilization Parameter (\texttt{lambda}, numeric)

**Stacked AutoEncoder Deep Neural Network** (method = \texttt{`dnn’})
For classification and regression using package \texttt{deepnet} with tuning parameters:
  • Hidden Layer 1 (\texttt{layer1}, numeric)
  • Hidden Layer 2 (\texttt{layer2}, numeric)
  • Hidden Layer 3 (\texttt{layer3}, numeric)
  • Hidden Dropouts (\texttt{hidden\_dropout}, numeric)
  • Visible Dropout (\texttt{visible\_dropout}, numeric)

**Stochastic Gradient Boosting** (method = \texttt{`gbm’})
For classification and regression using packages \texttt{gbm} and \texttt{plyr} with tuning parameters:
  • Number of Boosting Iterations (\texttt{n.trees}, numeric)
  • Max Tree Depth (\texttt{interaction\_depth}, numeric)
  • Shrinkage (\texttt{shrinkage}, numeric)
  • Min. Terminal Node Size (\texttt{n.minobsinnode}, numeric)

**Subtractive Clustering and Fuzzy c-Means Rules** (method = \texttt{`SBC’})
For regression using package \texttt{frbs} with tuning parameters:
  • Radius (\texttt{r.a}, numeric)
  • Upper Threshold (\texttt{eps.high}, numeric)
  • Lower Threshold (\texttt{eps.low}, numeric)

**Supervised Principal Component Analysis** (method = \texttt{`superpc’})
For regression using package \texttt{superpc} with tuning parameters:
  • Threshold (\texttt{threshold}, numeric)
  • Number of Components (\texttt{n.components}, numeric)

**Support Vector Machines with Boundrange String Kernel** (method = \texttt{`svmBoundrangeString’})
For classification and regression using package \texttt{kernlab} with tuning parameters:
  • length (\texttt{length}, numeric)
  • Cost (\texttt{C}, numeric)
Support Vector Machines with Class Weights (method = 'svmRadialWeights')
For classification using package kernlab with tuning parameters:
  • Sigma (sigma, numeric)
  • Cost (C, numeric)
  • Weight (Weight, numeric)

Support Vector Machines with Exponential String Kernel (method = 'svmExpoString')
For classification and regression using package kernlab with tuning parameters:
  • lambda (lambda, numeric)
  • Cost (C, numeric)

Support Vector Machines with Linear Kernel (method = 'svmLinear')
For classification and regression using package kernlab with tuning parameters:
  • Cost (C, numeric)

Support Vector Machines with Linear Kernel (method = 'svmLinear2')
For classification and regression using package e1071 with tuning parameters:
  • Cost (cost, numeric)

Support Vector Machines with Polynomial Kernel (method = 'svmPoly')
For classification and regression using package kernlab with tuning parameters:
  • Polynomial Degree (degree, numeric)
  • Scale (scale, numeric)
  • Cost (C, numeric)

Support Vector Machines with Radial Basis Function Kernel (method = 'svmRadial')
For classification and regression using package kernlab with tuning parameters:
  • Sigma (sigma, numeric)
  • Cost (C, numeric)

Support Vector Machines with Radial Basis Function Kernel (method = 'svmRadialCost')
For classification and regression using package kernlab with tuning parameters:
  • Cost (C, numeric)

Support Vector Machines with Radial Basis Function Kernel (method = 'svmRadialSigma')
For classification and regression using package kernlab with tuning parameters:
  • Sigma (sigma, numeric)
  • Cost (C, numeric)
Note: This SVM model tunes over the cost parameter and the RBF kernel parameter sigma. In the latter case, using `tunelength` will, at most, evaluate six values of the kernel parameter. This enables a broad search over the cost parameter and a relatively narrow search over sigma.

**Support Vector Machines with Spectrum String Kernel** (method = 'svmSpectrumString')

For classification and regression using package `kernlab` with tuning parameters:

- length (length, numeric)
- Cost (C, numeric)

**The Bayesian lasso** (method = 'blasso')

For regression using package `monomvn` with tuning parameters:

- Sparsity Threshold (sparsity, numeric)

Note: This model creates predictions using the mean of the posterior distributions but sets some parameters specifically to zero based on the tuning parameter sparsity. For example, when sparsity = .5, only coefficients where at least half the posterior estimates are nonzero are used.

**The lasso** (method = 'lasso')

For regression using package `elasticnet` with tuning parameters:

- Fraction of Full Solution (fraction, numeric)

**Tree Augmented Naive Bayes Classifier** (method = 'tan')

For classification using package `bnclassify` with tuning parameters:

- Score Function (score, character)
- Smoothing Parameter (smooth, numeric)

**Tree Augmented Naive Bayes Classifier Structure Learner Wrapper** (method = 'tanSearch')

For classification using package `bnclassify` with tuning parameters:

- Number of Folds (k, numeric)
- Minimum Absolute Improvement (epsilon, numeric)
- Smoothing Parameter (smooth, numeric)
- Final Smoothing Parameter (final_smooth, numeric)
- Super-Parent (sp, logical)

**Tree Augmented Naive Bayes Classifier with Attribute Weighting** (method = 'awtan')

For classification using package `bnclassify` with tuning parameters:

- Score Function (score, character)
- Smoothing Parameter (smooth, numeric)

**Tree Models from Genetic Algorithms** (method = 'evtree')

For classification and regression using package `evtree` with tuning parameters:

- Complexity Parameter (alpha, numeric)
**Tree-Based Ensembles** (method = 'nodeHarvest')
For classification and regression using package **nodeHarvest** with tuning parameters:

- Maximum Interaction Depth (maxinter, numeric)
- Prediction Mode (mode, character)

**Variational Bayesian Multinomial Probit Regression** (method = 'vbmpRadial')
For classification using package **vbmp** with tuning parameters:

- Theta Estimated (estimateTheta, character)

**Wang and Mendel Fuzzy Rules** (method = 'WM')
For regression using package **frbs** with tuning parameters:

- Number of Fuzzy Terms (num.labels, numeric)
- Membership Function (type.mf, character)

**Weighted Subspace Random Forest** (method = 'wsrf')
For classification using package **wsrf** with tuning parameters:

- Number of Randomly Selected Predictors (mtry, numeric)

**References**


---

**update.safs**

*Update or Re-fit a SA or GA Model*

**Description**

update allows a user to over-ride the search iteration selection process.

Based on the results of plotting a **gafs** or **safs** object, these functions can be used to supersede the number of iterations determined analytically from the resamples.

Any values of ... originally passed to **gafs** or **safs** are automatically passed on to the updated model (i.e. they do not need to be supplied again to update.

**Usage**

```r
## S3 method for class 'safs'
update(object, iter, x, y, ...)
```
Arguments

object       An object produced by `gafs` or `safs`
iter         a single numeric integer
x, y         the original training data used in the call to `gafs` or `safs`
...          not currently used

Value

an object of class `gafs` or `safs`

Author(s)

Max Kuhn

See Also

gafs, safs

Examples

```r
## Not run:
set.seed(1)
train_data <- twoClassSim(100, noiseVars = 10)
test_data  <- twoClassSim(10, noiseVars = 10)

## A short example
ctrl <- safsControl(functions = rfSA,
                     method = "cv",
                     number = 3)

rf_search <- safs(x = train_data[, -ncol(train_data)],
y = train_data$Class,
iter = 3,
safsControl = ctrl)

rf_search2 <- update(rf_search,
iter = 1,
x = train_data[, -ncol(train_data)],
y = train_data$Class)

rf_search2

## End(Not run)
```
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

```r
## 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
varImp(object, ...)
```

## S3 method for class 'bagEarth'
```r
varImp(object, ...)
```

## S3 method for class 'bagFDA'
```r
varImp(object, ...)
```

## S3 method for class 'C5.0'
```r
varImp(object, ...)
```

## S3 method for class 'cubist'
```r
varImp(object, weights = c(0.5, 0.5), ...)
```

## S3 method for class 'dsa'
```r
varImp(object, cuts = NULL, ...)
```

## S3 method for class 'glm'
```r
varImp(object, ...)
```
## S3 method for class 'glmnet'
varImp(object, lambda = NULL, ...)

## S3 method for class 'JRip'
varImp(object, ...)

## S3 method for class 'multinom'
varImp(object, ...)

## S3 method for class 'nnet'
varImp(object, ...)

## S3 method for class 'avNNet'
varImp(object, ...)

## S3 method for class 'PART'
varImp(object, ...)

## S3 method for class 'RRF'
varImp(object, ...)

## 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 = NULL, ...)

## S3 method for class 'classbagg'
varImp(object, ...)

## 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 'earth'
varImp(object, value = "gcv", ...)
## S3 method for class 'RandomForest'
varImp(object, ...)

## S3 method for class 'plsda'
varImp(object, ...)

## S3 method for class 'fda'
varImp(object, value = "gcv", ...)

## S3 method for class 'gam'
varImp(object, ...)

## S3 method for class 'Gam'
varImp(object, ...)

## S3 method for class 'train'
varImp(object, useModel = TRUE, nonpara = TRUE,
    scale = TRUE, ...)

### Arguments

- **object**: an object corresponding to a fitted model
- **...**: parameters to pass to the specific varImp methods
- **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).
- **cuts**: the number of rule sets to use in the model (for partDSA only)
- **lambda**: a single value of the penalty parameter
- **surrogates**: should surrogate splits contribute to the importance calculation?
- **competes**: should competing splits contribute to the importance calculation?
- **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)
- **estimate**: which estimate of performance should be used? See mvrVal
- **value**: the statistic that will be used to calculate importance: either gcv, nsubsets, or rss
- **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?
Details

For models that do not have corresponding \texttt{varImp} methods, see \texttt{filterVarImp}.

Otherwise:

\textbf{Linear Models}: the absolute value of the t–statistic for each model parameter is used.

\texttt{glmboost} and \texttt{glmnet}: the absolute value of the coefficients corresponding the the tuned model are used.

\textbf{Random Forest}: \texttt{varImp.RANDOMforest} and \texttt{varImp.RandomForest} are wrappers around the importance functions from the \texttt{randomForest} and \texttt{party} packages, respectively.

\textbf{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.

\textbf{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 \texttt{maxcompete} argument in \texttt{rpart.control}. This method does not currently provide class--specific measures of importance when the response is a factor.

\textbf{Bagged Trees}: The same methodology as a single tree is applied to all bootstrapped trees and the total importance is returned

\textbf{Boosted Trees}: \texttt{varImp.gbm} is a wrapper around the function from that package (see the \texttt{gbm} package vignette)

\textbf{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(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(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(object,value =" nsubsets")}, which counts the number of subsets where the variable is used (in the final, pruned model).

\textbf{Nearest shrunkent 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{pamrtrained} is given to \texttt{varImp}.

\textbf{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 `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 `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**


Description

Variable importance scores for `safs` and `gafs` objects.

Usage

```r
## S3 method for class 'gafs'
varImp(object, metric = object$control$metric["external"],
       maximize = object$control$maximize["external"], ...)
```

Arguments

- `object` an `safs` or `gafs` object
- `metric` a metric to compute importance (see Details below)
- `maximize` are larger values of the metric better?
- `...` not currently uses

Details

A crude measure of importance is computed for the two search procedures. At the end of a search process, the difference in the fitness values is computed for models with and without each feature (based on the search history). If a predictor has at least two subsets that include and did not include the predictor, a t-statistic is computed (otherwise a value of `NA` is assigned to the predictor).

This computation is done separately for each resample and the t-statistics are averaged (NA values are ignored) and this average is reported as the importance. If the fitness value should be minimized, the negative value of the t-statistic is used in the average.

As such, the importance score reflects the standardized increase in fitness that occurs when the predictor is included in the subset. Values near zero (or negative) indicate that the predictor may not be important to the model.

Value

a data frame where the rownames are the predictor names and the column is the average t-statistic

Author(s)

Max Kuhn

See Also

`safs, gafs`
**var_seq**

Sequences of Variables for Tuning

**Description**

This function generates a sequence of mtry values for random forests.

**Usage**

```r
default(var_seq(p, classification = FALSE, len = 3)
```

**Arguments**

- `p` The number of predictors
- `classification` Is the outcome a factor (classification = TRUE or numeric?)
- `len` The number of mtry values to generate.

**Details**

If the number of predictors is less than 500, a simple sequence of values of length len is generated between 2 and p. For larger numbers of predictors, the sequence is created using log2 steps.

If len = 1, the defaults from the randomForest package are used.

**Value**

a numeric vector

**Author(s)**

Max Kuhn

**Examples**

```r
var_seq(p = 100, len = 10)
var_seq(p = 600, len = 10)
```
Description

Lattice and ggplot 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[1], units = "min", ...)

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

## S3 method for class 'resamples'
splom(x, data = NULL, variables = "models",
models = x$models, metric = NULL, panelRange = NULL, ...)

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

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

## S3 method for class 'resamples'
dotplot(x, data = NULL, models = x$models,
metric = x$metric, conf.level = 0.95, ...)

## S3 method for class 'resamples'
ggplot(data = NULL, mapping = NULL,
environment = NULL, models = data$models, metric = data$metric[1],
conf.level = 0.95, ...)
```

Arguments

- **x**: an object generated by `resamples`
- **data**: Only used for the `ggplot` method; an object generated by `resamples`
- **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`.

- **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"`.
- **units**: either "sec", "min" or "hour"; which what is either "tTime", "mTime" or "pTime", how should the timings be scaled?
- **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.
- **conf.level**: the confidence level for intervals about the mean (obtained using `t.test`)
- **mapping, environment**: Not used.

**Details**

The ideas and methods here are based on Hothorn et al. (2005) and Eugster et al. (2008).

- `dotplot` and `ggplot` 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://topepo.github.io/caret/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**
  bag, 7
  BloodBrain, 14
  cars, 20
  cox2, 27
  dhfr, 34
  GermanCredit, 60
  mdrr, 79
  oil, 89
  pickSizeBest, 96
  pottery, 106
  Sacramento, 133
  safs_initial, 137
  scat, 145
  segmentationData, 146
  tecator, 152

*Topic **graphs**
  panel.needle, 93

*Topic **hplot**
  calibration, 16
  densityplot.rfe, 32
  dotPlot, 37
  dotplot.diff.resamples, 38
  featurePlot, 46
  ggpplot.rfe, 61
  ggpplot.train, 62
  histogram.train, 64
  lift, 74
  panel.lift2, 91
  plot.gafs, 98
  plot.varImp.train, 99
  plotClassProbs, 100
  plotObsVsPred, 102
  prcomp.resamples, 107
  resampleHist, 122
  xypplot.resamples, 207

*Topic **manip**
  classDist, 20
  extractPrediction, 43
  findCorrelation, 48
  findLinearCombos, 50
  negPredValue, 83
  oneSE, 89
  recall, 119
  spatialSign, 150
  summary.bagEarth, 151

*Topic **models**
  bag, 7
  caretSBF, 19
  diff.resamples, 34
  dummyVars, 40
  filterVarImp, 47
  format.bagEarth, 51
  gafs.default, 52
  learning_curve_dat, 72
  nullModel, 87
  pickSizeBest, 96
  plsda, 104
  predictors, 113
  resamples, 123
  rfe, 126
  safs, 134
  sbf, 139
  SLCP1_1, 146
  train, 155
  train_model_list, 165
  update.safs, 197
  update.train, 199
  var_seq, 206
  varImp, 200

*Topic **multivariate**
  icr.formula, 66
  knn3, 68
  knnreg, 70
  predict.gafs, 110
  predict.knn3, 111
  predict.knnreg, 112

*Topic **neural**
createTimeSlices, 162
cREATE_TIMESLICES (CREATE_DATAPARTITION), 27
cree, 113
ctreeBag (bag), 7
defaultSummary, 30, 162
densityPlot, 33, 38, 65, 101, 122, 208
dENSITY_PLOT, 36
densityPlot.diff.resamples, 36
dENSITY_PLOT.DIFF.RESAMPLES
   (dotplot.diff.resamples), 38
densityPlot.resamples, 124
dENSITY_PLOT.RESAMPLES
   (xyplot.resamples), 207
densityPlot.rfe, 32
densityPlot.train, 122
densityPlot.train (histogram.train), 64
dhfr, 34
diff.resamples, 34, 38, 124
dist, 78
dotPlot, 37
dotplot, 37, 38, 93, 100, 102, 208
dotplot.diff.resamples, 35, 36, 38
dotplot.resamples (xyplot.resamples), 207
downSample, 39
dummyVars, 40
earth, 11, 52, 109, 113
dendpoints (tecator), 152
devimp, 203
dexpotrans, 117
dexpotrans (BoxCoxTrans), 14
eXTRACT_PREDICTION, 43, 102
extractProb, 100, 101
extractProb (extractPrediction), 43
formula, 42
gafs, 53, 55, 57–59, 98, 99, 110, 111, 197, 198, 205
gafs (gafs.default), 52
gafs.default, 52
gafs_initial, 58
gafs_lrselection (gafs_initial), 58
gafs_nlrselection (gafs_initial), 58
gafs_radmutation (gafs_initial), 58
gafs_rwsselection (gafs_initial), 58
gafs_spCrossover (gafs_initial), 58
gafs_tourselection (gafs_initial), 58
gafs_uCrossover (gafs_initial), 58
gafsControl, 53, 54, 55, 58, 59
gamFuncs (pickSizeBest), 96
gamScores, 144
gamScores (caretSBF), 19
genetic, 49
GermanCredit, 60
getModelInfo, 157
getModelInfo (modelLookup), 80
getSamplingInfo, 60
getTrainPerf (defaultSummary), 30
ggplot, 62, 64, 99
GGPLOT.CALIBRATION (CALIBRATION), 16
GGPLOT.LIFT (LIFT), 74
GGPLOT.RESAMPLES (XYPLOT.RESAMPLES), 207
GGPLOT.RFE, 61
GGPLOT.TRAIN, 62
GGPLOT.VAR_IMP.TRAIN
   (plot.varImp.train), 99
grepl, 80, 81
GROUP_KFOLD (CREATE_DATAPARTITION), 27
hclust, 107, 108
HISTOGRAM, 33, 65, 101, 122, 208
HISTOGRAM.RFE (DENSITY_PLOT.RFE), 32
HISTOGRAM.TRAIN, 64, 122
icr (icr.formula), 66
icr.formula, 66
INDEX2VEC, 68
install.packages, 80, 81
ipredBAGG, 113
ipredknn, 70, 71
KNN, 70, 71, 113
knn3, 68, 112
INDEX

knnTrain (knn3), 68
knnStrain, 68
knnTrain (knnreg), 70

lattice.options, 17, 75
ldaBag (bag), 7
ldaFuncs (pickSizeBest), 96
ldaSBF, 144
ldaSBF (caretSBF), 19
leaps, 49
learning_curve_dat, 72
levelplot, 38, 63, 64
levelplot.diff.resamples, 36
levelplot.diff.resamples
  (dotplot.diff.resamples), 38
lift, 74, 91, 92
lm, 47, 67
lmFuncs, 132, 133
lmFuncs (pickSizeBest), 96
lmSBF, 141, 144
lmSBF (caretSBF), 19
loess, 47
logBBB (BloodBrain), 14
LPH07.1 (SLC14_1), 146
LPH07.2 (SLC14_1), 146
lrFuncs (pickSizeBest), 96

MAE (defaultSummary), 30
mahalanobis, 22
maxDissim, 76
mcnemar.test, 24
mdrr, 79
mdrrClass (mdrr), 79
mdrrDescr (mdrr), 79
minDiss (maxDissim), 76
mmLogLoss (defaultSummary), 30
model.matrix, 41, 42
modelCor (resamples), 123
modelLookup, 80, 159
models, 159
models (train_model_list), 165
multiClassSummary (defaultSummary), 30
mvrVal, 202

NaiveBayes, 105
nbBag (bag), 7
nbFuncs, 132, 133
nbFuncs (pickSizeBest), 96
nbSBF, 144

nbSBF (caretSBF), 19
nearZeroVar, 81, 114, 115
negPredValue, 81, 114, 115
nnet, 6, 7, 94, 95, 113
nnetBag (bag), 7
nullModel, 87
nzv (nearZeroVar), 81

oil, 89
oilType (oil), 89
oneSE, 89
optim, 16

p.adjust, 35
pamr.train, 113
panel.calibration, 18
panel.calibration (calibration), 16
panel.dotplot, 93
panel.lift (panel.lift2), 91
panel.lift2, 75, 91
panel.needle, 93, 100
panel.xyplot, 92
parallelplot.resamples
  (xyplot.resamples), 207
pcaANNet, 94
pickSizeBest, 96, 132, 133
pickSizeTolerance, 132, 133
pickSizeTolerance (pickSizeBest), 96
pickVars (pickSizeBest), 96
plot.gafs, 98
plot.pcomp.resamples
  (pcomp.resamples), 107
plot.rfe (ggplot.rfe), 61
plot.safs (plot.gafs), 98
plot.train (ggplot.train), 62
plot.varImp.train, 99
plotClassProbs, 44, 45, 100
plotObsVsPred, 44, 45, 102
plsBag (bag), 7
 plsda, 104
 plsr, 104, 105
 posPredValue, 24
 posPredValue (negPredValue), 83
 postResample, 126
 postResample (defaultSummary), 30
 pottery, 106
 potteryClass (pottery), 106
 pcomp, 21, 107, 117
 pcomp.resamples, 107
INDEX

sumDiss (maxDissim), 76
summary.bag (bag), 7
summary.bagEarth, 151
summary.bagFDA (summary.bagEarth), 151
summary.cubist, 204
summary.diff.resamples (diff.resamples), 34
summary.Gam, 19, 20
summary.resamples (resamples), 123
superpc.train, 113
svmBag (bag), 7
t.test, 35, 36, 208
table, 22
tecator, 152
terms.formula, 42
thresholder, 153
tolerance (oneSE), 89
train, 9, 25, 26, 30, 31, 35, 44, 45, 58, 62–65, 72,
73, 80, 81, 90, 91, 113, 115, 118,
122–124, 138, 153, 154, 155, 157,
161, 162, 164, 166, 197, 199
train_model_list, 165
trainControl, 26, 30–32, 44, 45, 54–56, 65,
89–91, 122, 124, 135, 154, 156–159,
161, 199, 208
treebagFuncs, 132, 133
treebagFuncs (pickSizeBest), 96
treebagGA, 54, 58
treebagGA (gafs_initial), 58
treebagSA, 58
treebagSA (safs_initial), 137
treebagSBF, 144
treebagSBF (caretSBF), 19
trellis.par.set, 18, 75, 76, 92
trim.matrix, 50
twoClassSim (SLC14_1), 146
twoClassSummary (defaultSummary), 30
update.gafs (update.safs), 197
update.rfe (rfe), 126
update.safs, 197
update.train, 159, 199
update.trellis, 75
upsample (downsample), 39

var_seq, 206
varImp, 37, 200
varImp.gafs, 205

varImp.safs (varImp.gafs), 205

xyplot, 17, 18, 33, 38, 61–65, 74–76, 92, 98,
99, 102, 107, 108, 208
xyplot.calibration (calibration), 16
xyplot.lift (lift), 74
xyplot.resamples, 124, 207
xyplot.rfe (densityplot.rfe), 32
xyplot.train, 122
xyplot.train (histogram.train), 64

varimp (varImp.gafs), 205