Package ‘randomForestSRC’

January 2, 2019

Version 2.8.0

Date 2019-01-02

Title Random Forests for Survival, Regression, and Classification
(RF-SRC)

Author Hemant Ishwaran <hemant.ishwaran@gmail.com>, Udaya B. Kogalur <ubk@kogalur.com>

Maintainer Udaya B. Kogalur <ubk@kogalur.com>

BugReports https://github.com/kogalur/randomForestSRC/issues/new

Depends R (>= 3.1.0),

Imports parallel

Suggests glmnet, survival, pec, prodlim, mlbench, akima, caret

Description Fast OpenMP parallel processing for Breiman's random forests for survival, competing risks, regression and classification based on Ishwaran and Kogalur's popular random survival forests (RSF) package. Handles missing data and now includes multivariate, unsupervised forests and quantile regression. New fast interface using subsampling.

License GPL (>= 3)

URL http://web.ccs.miami.edu/~hishwaran http://www.kogalur.com
https://github.com/kogalur/randomForestSRC

NeedsCompilation yes

Repository CRAN

Date/Publication 2019-01-02 16:20:08 UTC

R topics documented:

`randomForestSRC-package`

Random Forests for Survival, Regression, and Classification (RF-SRC)

Description

Fast OpenMP parallel processing unified treatment of Breiman's random forests (Breiman 2001) for a variety of data settings. Regression and classification forests are grown when the response is numeric or categorical (factor), while survival and competing risk forests (Ishwaran et al. 2008, 2012) are grown for right-censored survival data. Multivariate regression and classification responses as well as mixed outcomes (regression/classification responses) are also handled. Also includes unsupervised forests and quantile regression forests, `quantileReg`. Different splitting rules invoked under deterministic or random splitting are available for all families. Variable predictiveness can be assessed using variable importance (VIMP) measures for single, as well as grouped variables. Missing data (for x-variables and y-outcomes) can be imputed on both training and test data. The underlying code is based on Ishwaran and Kogalur's now retired **randomSurvivalForest** package (Ishwaran and Kogalur 2007), and has been significantly refactored for improved computational speed.

Package Overview

This package contains many useful functions and users should read the help file in its entirety for details. However, we briefly mention several key functions that may make it easier to navigate and understand the layout of the package.

1. `rfsrc`

   This is the main entry point to the package. It grows a random forest using user supplied training data. We refer to the resulting object as a RF-SRC grow object. Formally, the resulting object has class `(rfsrc, grow)`.

2. `rfsrcFast`

   A fast implementation of `rfsrc` using subsampling.

3. `predict.rfsrc`, `predict`

   Used for prediction. Predicted values are obtained by dropping the user supplied test data down the grow forest. The resulting object has class `(rfsrc, predict)`.

4. `max.subtree`, `var.select`

   Used for variable selection. The function `max.subtree` extracts maximal subtree information from a RF-SRC object which is used for selecting variables by making use of minimal depth variable selection. The function `var.select` provides an extensive set of variable selection options and is a wrapper to `max.subtree`.

5. `impute.rfsrc`

   Fast imputation mode for RF-SRC. Both `rfsrc` and `predict.rfsrc` are capable of imputing missing data. However, for users whose only interest is imputing data, this function provides an efficient and fast interface for doing so.

6. `partial.rfsrc`

   Used to extract the partial effects of a variable or variables on the ensembles.

Source Code, Beta Builds and Bug Reporting

1. Regular stable releases of this package are available on CRAN at `cran.r-project.org/package=randomForestSRC`

2. Interim unstable development builds with bug fixes and sometimes additional functionality are available at `github.com/kogalur/randomForestSRC`

3. Bugs may be reported via `github.com/kogalur/randomForestSRC/issues/new`. Please provide the accompanying information with any reports:

   (a) `sessionInfo()`

   (b) A minimal reproducible example consisting of the following items:

   - a minimal dataset, necessary to reproduce the error
   - the minimal runnable code necessary to reproduce the error, which can be run on the given dataset
   - the necessary information on the used packages, R version and system it is run on
   - in the case of random processes, a seed (set by `set.seed()`) for reproducibility

OpenMP Parallel Processing – Installation

This package implements OpenMP shared-memory parallel programming if the target architecture and operating system support it. This is the default mode of execution.

Additional instructions for configuring OpenMP parallel processing are available at kogalur.github.io/randomForestSRC/building.html.

An understanding of resource utilization (CPU and RAM) is necessary when running the package using OpenMP and Open MPI parallel execution. Memory usage is greater when running with OpenMP enabled. Diligence should be used not to overtax the hardware available.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Breiman L. (2001). Random forests, *Machine Learning*, 45:5-32.

Ishwaran H. and Kogalur U.B. (2007). Random survival forests for R, *Rnews*, 7(2):25-31.

Ishwaran H. (2007). Variable importance in binary regression trees and forests, *Electronic J. Statist.*, 1:519-537.

Ishwaran H., Kogalur U.B., Blackstone E.H. and Lauer M.S. (2008). Random survival forests, *Ann. App. Statist.*, 2:841-860.

Ishwaran H., Kogalur U.B., Gorodeski E.Z, Minn A.J. and Lauer M.S. (2010). High-dimensional variable selection for survival data. *J. Amer. Statist. Assoc.*, 105:205-217.

Ishwaran H., Kogalur U.B., Chen X. and Minn A.J. (2011). Random survival forests for high-dimensional data. *Stat. Anal. Data Mining*, 4:115-132

Ishwaran H., Gerds T.A., Kogalur U.B., Moore R.D., Gange S.J. and Lau B.M. (2014). Random survival forests for competing risks. *Biostatistics*, 15(4):757-773.

Ishwaran H. and Malley J.D. (2014). Synthetic learning machines. *BioData Mining*, 7:28.

Ishwaran H. (2015). The effect of splitting on random forests. *Machine Learning*, 99:75-118.

Ishwaran H. and Lu M. (2018). Standard errors and confidence intervals for variable importance in random forest regression, classification, and survival. Statistics in Medicine (in press).

Mantero A. and Ishwaran H. (2017). Unsupervised random forests.

O'Brien R. and Ishwaran H. (2017). A random forests quantile classifier for class imbalanced data.

Tang F. and Ishwaran H. (2017). Random forest missing data algorithms. *Statistical Analysis and Data Mining*, 10, 363-377.

See Also

`find.interaction`,

`impute`, `max.subtree`,

`plot.competing.risk`, `plot.rfsrc`, `plot.survival`, `plot.variable`, `predict.rfsrc`, `print.rfsrc`, `quantileReg`, `rfsrcFast`, `rfsrcSyn`,

`subsample`,

`stat.split`, `tune`, `var.select`, `vimp`

breast *Wisconsin Prognostic Breast Cancer Data*

Description

Recurrence of breast cancer from 198 breast cancer patients, all of which exhibited no evidence of distant metastases at the time of diagnosis. The first 30 features of the data describe characteristics of the cell nuclei present in the digitized image of a fine needle aspirate (FNA) of the breast mass.

Source

The data were obtained from the UCI machine learning repository, see http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Prognostic).

Examples

```
## ------------------------------------------------------------
## Standard analysis
## ------------------------------------------------------------

data(breast, package = "randomForestSRC")
breast <- na.omit(breast)
o <- rfsrc(status ~ ., data = breast, nsplit = 10)
print(o)

## ------------------------------------------------------------
## The data is imbalanced so we use balanced random forests
## with undersampling of the majority class
##
## Specifically let n0, n1 be sample sizes for majority, minority
## cases.  We sample 2 x n1 cases with majority, minority cases chosen
## with probabilities n1/n, n0/n where n=n0+n1
## ------------------------------------------------------------

y <- breast$status
o <- rfsrc(status ~ ., data = breast, nsplit = 10,
            case.wt = randomForestSRC:::make.wt(y),
            sampsize = randomForestSRC:::make.size(y))
print(o)
```

find.interaction *Find Interactions Between Pairs of Variables*

Description

Find pairwise interactions between variables.

Usage

```
## S3 method for class 'rfsrc'
find.interaction(object, xvar.names, cause, m.target,
  importance = c("permute", "random", "anti",
                 "permute.ensemble", "random.ensemble", "anti.ensemble"),
  method = c("maxsubtree", "vimp"), sorted = TRUE, nvar, nrep = 1, subset,
  na.action = c("na.omit", "na.impute"),
  seed = NULL, do.trace = FALSE, verbose = TRUE, ...)
```

Arguments

`object`	An object of class `(rfsrc, grow)` or `(rfsrc, forest)`.
`xvar.names`	Character vector of names of target x-variables. Default is to use all variables.
`cause`	For competing risk families, integer value between 1 and `J` indicating the event of interest, where `J` is the number of event types. The default is to use the first event type.
`m.target`	Character value for multivariate families specifying the target outcome to be used. If left unspecified, the algorithm will choose a default target.
`importance`	Type of variable importance (VIMP). See `rfsrc` for details.
`method`	Method of analysis: maximal subtree or VIMP. See details below.
`sorted`	Should variables be sorted by VIMP? Does not apply for competing risks.
`nvar`	Number of variables to be used.
`nrep`	Number of Monte Carlo replicates when ‘`method="vimp"`’.
`subset`	Vector indicating which rows of the x-variable matrix from the `object` are to be used. Uses all rows if not specified.
`na.action`	Action to be taken if the data contains `NA` values. Applies only when ‘`method="vimp"`’.
`seed`	Seed for random number generator. Must be a negative integer.
`do.trace`	Number of seconds between updates to the user on approximate time to completion.
`verbose`	Set to `TRUE` for verbose output.
`...`	Further arguments passed to or from other methods.

Details

Using a previously grown forest, identify pairwise interactions for all pairs of variables from a specified list. There are two distinct approaches specified by the option ‘`method`’.

1. ‘`method="maxsubtree"`’

   This invokes a maximal subtree analysis. In this case, a matrix is returned where entries [i][i] are the normalized minimal depth of variable [i] relative to the root node (normalized wrt the size of the tree) and entries [i][j] indicate the normalized minimal depth of a variable [j] wrt the maximal subtree for variable [i] (normalized wrt the size of [i]'s maximal subtree). Smaller [i][i] entries indicate predictive variables. Small [i][j] entries having small [i][i] entries are a sign of an interaction between variable i and j (note: the user should scan rows, not columns, for small entries). See Ishwaran et al. (2010, 2011) for more details.

2. ‘method="vimp"’

   This invokes a joint-VIMP approach. Two variables are paired and their paired VIMP calculated (refered to as ’Paired’ importance). The VIMP for each separate variable is also calculated. The sum of these two values is refered to as ’Additive’ importance. A large positive or negative difference between ’Paired’ and ’Additive’ indicates an association worth pursuing if the univariate VIMP for each of the paired-variables is reasonably large. See Ishwaran (2007) for more details.

Computations might be slow depending upon the size of the data and the forest. In such cases, consider setting ‘nvar’ to a smaller number. If ‘method="maxsubtree"’, consider using a smaller number of trees in the original grow call.

If ‘nrep’ is greater than 1, the analysis is repeated nrep times and results averaged over the replications (applies only when ‘method="vimp"’).

Value

Invisibly, the interaction table (a list for competing risk data) or the maximal subtree matrix.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H. (2007). Variable importance in binary regression trees and forests, *Electronic J. Statist.*, 1:519-537.

Ishwaran H., Kogalur U.B., Gorodeski E.Z, Minn A.J. and Lauer M.S. (2010). High-dimensional variable selection for survival data. *J. Amer. Statist. Assoc.*, 105:205-217.

Ishwaran H., Kogalur U.B., Chen X. and Minn A.J. (2011). Random survival forests for high-dimensional data. *Statist. Anal. Data Mining*, 4:115-132.

See Also

max.subtree, var.select, vimp

Examples

```
## ------------------------------------------------------------
## find interactions, survival setting
## ------------------------------------------------------------

data(pbc, package = "randomForestSRC")
pbc.obj <- rfsrc(Surv(days,status) ~ ., pbc, importance = TRUE)
find.interaction(pbc.obj, method = "vimp", nvar = 8)

## ------------------------------------------------------------
## find interactions, competing risks
## ------------------------------------------------------------
```

```
data(wihs, package = "randomForestSRC")
wihs.obj <- rfsrc(Surv(time, status) ~ ., wihs, nsplit = 3, ntree = 100,
                       importance = TRUE)
find.interaction(wihs.obj)
find.interaction(wihs.obj, method = "vimp")

## ------------------------------------------------------------
## find interactions, regression setting
## ------------------------------------------------------------

airq.obj <- rfsrc(Ozone ~ ., data = airquality, importance = TRUE)
find.interaction(airq.obj, method = "vimp", nrep = 3)
find.interaction(airq.obj)

## ------------------------------------------------------------
## find interactions, classification setting
## ------------------------------------------------------------

iris.obj <- rfsrc(Species ~., data = iris, importance = TRUE)
find.interaction(iris.obj, method = "vimp", nrep = 3)
find.interaction(iris.obj)

## ------------------------------------------------------------
## interactions for multivariate mixed forests
## ------------------------------------------------------------

mtcars2 <- mtcars
mtcars2$cyl <- factor(mtcars2$cyl)
mtcars2$carb <- factor(mtcars2$carb, ordered = TRUE)
mv.obj <- rfsrc(cbind(carb, mpg, cyl) ~., data = mtcars2, importance = TRUE)
find.interaction(mv.obj, method = "vimp", outcome.target = "carb")
find.interaction(mv.obj, method = "vimp", outcome.target = "mpg")
find.interaction(mv.obj, method = "vimp", outcome.target = "cyl")
```

`follic` *Follicular Cell Lymphoma*

Description

Competing risk data set involving follicular cell lymphoma.

Format

A data frame containing:

age	age
hgb	hemoglobin (g/l)
clinstg	clinical stage: 1=stage I, 2=stage II
ch	chemotherapy

rt	radiotherapy
time	first failure time
status	censoring status: 0=censored, 1=relapse, 2=death

Source

Table 1.4b, *Competing Risks: A Practical Perspective*.

References

Pintilie M., (2006) *Competing Risks: A Practical Perspective.* West Sussex: John Wiley and Sons.

Examples

```
data(follic, package = "randomForestSRC")
follic.obj <- rfsrc(Surv(time, status) ~ ., follic, nsplit = 3, ntree = 100)
```

hd *Hodgkin's Disease*

Description

Competing risk data set involving Hodgkin's disease.

Format

A data frame containing:

age	age
sex	gender
trtgiven	treatment: RT=radition, CMT=Chemotherapy and radiation
medwidsi	mediastinum involvement: N=no, S=small, L=Large
extranod	extranodal disease: Y=extranodal disease, N=nodal disease
clinstg	clinical stage: 1=stage I, 2=stage II
time	first failure time
status	censoring status: 0=censored, 1=relapse, 2=death

Source

Table 1.6b, *Competing Risks: A Practical Perspective*.

References

Pintilie M., (2006) *Competing Risks: A Practical Perspective.* West Sussex: John Wiley and Sons.

Examples

```
data(hd, package = "randomForestSRC")
```

holdoutvimp *Hold out variable importance (VIMP)*

Description

Hold out VIMP is calculated from the error rate for trees grown with and without a variable. Applies to all families.

Usage

```
## S3 method for class 'rfsrc'
holdoutvimp(formula, data,
  ntree = 1000 * ncol(data) / vtry,
  ntree.max = 2000,
  nsplit = 10,
  ntime = 50,
  mtry = NULL,
  vtry = 1,
  fast = FALSE,
  verbose = TRUE,
  ...)
```

Arguments

`formula` A symbolic description of the model to be fit.

`data` Data frame containing the y-outcome and x-variables.

`ntree` Number of trees used for growing the forest.

`ntree.max` Maximum number of trees used when calculating prediction error for determing hold out VIMP.

`nsplit` Non-negative integer value specifying number of random split points used to split a node (deterministic splitting corresponds to the value zero and is much slower).

`ntime` Integer value used for survival to constrain ensemble calculations to a grid of `ntime` time points.

`mtry` Number of variables randomly selected as candidates for splitting a node.

`vtry` Number of variables randomly selected to be held out when growning a tree.

`fast` Use fast random forests, `rfsrcFast`, in place of `rfsrc`? Improves speed but is less accurate.

`verbose` Provide verbose output?

`...` Further arguments to be passed to `rfsrc`.

Details

Prior to growing a tree, a random set of `vtry` features are held out. Tree growing proceeds as usual with the remaining features. Once the forest is grown, hold out VIMP for a given variable v is calculated as follows. Gather all trees where v was held out and calculate OOB prediction error. Next gather all trees were v was not held out and calculate OOB prediction error. Hold out VIMP for v is the difference between these two values. Thus hold out VIMP measures the importance of a variable when that variable is truly removed from tree growing.

Accuracy of hold out VIMP depends heavily on the size of the forest. If the number of trees is too small, then number of trees where v is held out will be small, and the resulting OOB error will have high variance. Thus, `ntree` should be set fairly high - we recommend using 1000 times the number of features. Increasing `vtry` is another way to increase number of hold out trees. In particular, number of trees needed should decrease linearly with `vtry`. Keep in mind however that intrepretation of holdout VIMP is altered when `vtry` is different than 1. This is likely to be more of a concern in low dimensional settings.

Uses the new `get.tree` option in `predict` to extract specific trees from a forest and the hidden option `vtry` in `rfsrc`. The latter creates a hidden array `holdout.array` of zeroes and ones indicating which variable to hold out in a tree where number of rows equals number of features and number of columns equals number of trees. The array can also be passed as a hidden option but is not checked for coherence so users should be careful when doing so.

Value

Hold out VIMP for each variable. For multivariate forests, hold out VIMP is calculated for each of the target outcomes.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Lu M. and Ishwaran H. (2018). Expert Opinion: A prediction-based alternative to p-values in regression models. *J. Thoracic and Cardiovascular Surgery*, 155(3), 1130–1136.

See Also

`vimp`

Examples

```
## ------------------------------------------------------------
## Boston housing example
## ------------------------------------------------------------

if (library("mlbench", logical.return = TRUE)) {

  data(BostonHousing)
  hv <- holdoutvimp(medv ~ ., BostonHousing)
  print(hv)

}

## ------------------------------------------------------------
## Multivariate regression analysis
## ------------------------------------------------------------

hv <- holdoutvimp(cbind(mpg, cyl) ~., mtcars)
print(hv)

## ------------------------------------------------------------
## White wine classification example
## ------------------------------------------------------------

data(wine, package = "randomForestSRC")
wine$quality <- factor(wine$quality)
hv <- holdoutvimp(quality ~ ., wine, vtry = 5)
print(100 * hv)


## ------------------------------------------------------------
## pbc survival example
## ------------------------------------------------------------

data(pbc, package = "randomForestSRC")
hv <- holdoutvimp(Surv(days, status) ~ ., pbc, splitrule = "random")
print(100 * hv)

## ------------------------------------------------------------
## WIHS competing risk example
## ------------------------------------------------------------

data(wihs, package = "randomForestSRC")
hv <- holdoutvimp(Surv(time, status) ~ ., wihs, ntree = 1000)
print(100 * hv)
```

housing *Ames Iowa Housing Data*

Description

Data from the Ames Assessor's Office used in assessing values of individual residential properties sold in Ames, Iowa from 2006 to 2010. This is a regression problem and the goal is to predict "SalePrice" which records the price of a home in thousands of dollars.

References

De Cock, D., (2011). Ames, Iowa: Alternative to the Boston housing data as an end of semester regression project. *Journal of Statistics Education*, 19(3), 1–14.

Examples

```
## load the data
data(housing, package = "randomForestSRC")

## the original data contains lots of missing data
## here's a fast but reasonably accurate way to impute the data
housing2 <- impute(data = housing, mf.q = 10, fast = TRUE)
```

---

imbalanced *Imbalanced Two Class Problems*

Description

Implements various solutions to the two-class imbalanced problem, including the newly proposed quantile-classifier approach of O'Brien and Ishwaran (2017). Also includes Breiman's balanced random forests undersampling of the majority class. Performance is assesssed using the G-mean, but misclassification error can be requested.

Usage

```
## S3 method for class 'rfsrc'
imbalanced(formula, data, ntree = 3000,
  method = c("rfq", "brf", "standard"),
  perf.type = NULL,
  fast = FALSE,
  ratio = NULL,
  optimize = FALSE,
  ngrid = 1e4,
  ...)
```

Arguments

`formula` A symbolic description of the model to be fit.

`data` Data frame containing the two-class y-outcome and x-variables.

`ntree` Number of trees.

`method` Method used for fitting the classifier. The default is `rfq` which is the random forests quantile-classifer (RFQ) approach of O'Brien and Ishwaran (2017). The method `brf` implements the balanced random forest (BRF) method of Chen et al. (2004) which undersamples the majority class so that its cardinality matches that of the minority class. The method `standard` implements a standard random forest analysis.

`perf.type` Measure used for assessing performance (and all downstream calculations based on it such as variable importance). The default for `rfq` and `brf` is to use the G-mean (Kubat et al., 1997). For standard random forests, the default is misclassification error. Users can over-ride the default performance measure by manually selecting either `g.mean` for the G-mean, `misclass` for misclassification error, or `brier` for the normalized Brier score. See the examples below.

`fast` Use fast random forests, `rfsrcFast`, in place of `rfsrc`? Improves speed but is less accurate. Only applies to RFQ.

`ratio` This is an optional parameter for expert users and included only for experimental purposes. Used to specify the ratio (between 0 and 1) for undersampling the majority class. Option is ignored for BRF.

`optimize` Calculate the G-mean under various threshold values? Returns out-of-bag G-mean values for each tested threshold value. See examples below for illustration.

`ngrid` Number of threshold values attempted when `optimize` is requested

`...` Further arguments to be passed to the `rfsrc` function to specify random forest parameters.

Details

Imbalanced data, or the so-called imbalanced minority class problem, refers to classification settings involving two-classes where the ratio of the majority class to the minority class is much larger than one. Two solutions to the two-class imbalanced problem are provided here, including the newly proposed random forests quantile-classifier (RFQ) of O'Brien and Ishwaran (2017), and the balanced random forests (BRF) undersampling approach of Chen et al. (2004). The default performance metric is the G-mean (Kubat et al., 1997).

Currently, missing values cannot be handled for BRF or when the `ratio` option is used; in these cases, missing data is removed prior to the analysis.

We recommend setting `ntree` to a relatively large value when dealing with imbalanced data to ensure convergence of the performance value – this is especially true for the G-mean. Consider using 5 times the usual number of trees.

Value

A two-class random forest fit under the requested method and performance value.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Chen, C., Liaw, A. and Breiman, L. (2004). Using random forest to learn imbalanced data. University of California, Berkeley, Technical Report 110.

Kubat, M., Holte, R. and Matwin, S. (1997). Learning when negative examples abound. *Machine learning*, ECML-97: 146-153.

O'Brien R. and Ishwaran H. (2017). A random forests quantile classifier for class imbalanced data.

See Also

`rfsrc`, `rfsrcFast`

Examples

```
## ------------------------------------------------------------
## use the breast data for illustration
## ------------------------------------------------------------

data(breast, package = "randomForestSRC")
breast <- na.omit(breast)
f <- as.formula(status ~ .)

##----------------------------------------------------------------
## example 1: default RFQ call
##----------------------------------------------------------------
o.rfq <- imbalanced(f, breast)
print(o.rfq)

## equivalent to:
## rfsrc(f, breast, rfq =  TRUE, perf.type = "g.mean")

##----------------------------------------------------------------
## example 2: RFQ call with fast rfsrc
##----------------------------------------------------------------
o.rfq <- imbalanced(f, breast, fast = TRUE)
print(o.rfq)

## equivalent to:
## rfsrcFast(f, breast, rfq =  TRUE, perf.type = "g.mean")

##-----------------------------------------------------------------
## example 3: standard RF (uses misclassification)
## ------------------------------------------------------------
o.std <- imbalanced(f, breast, method = "stand")

##-----------------------------------------------------------------
## example 4: standard RF using G-mean performance
## ------------------------------------------------------------
o.std <- imbalanced(f, breast, method = "stand", perf.type = "g.mean")

## equivalent to:
## rfsrc(f, breast, perf.type = "g.mean")

##----------------------------------------------------------------
## example 5: default BRF call
##----------------------------------------------------------------
o.brf <- imbalanced(f, breast, method = "brf")

## equivalent to:
## imbalanced(f, breast, method = "brf", perf.type = "g.mean")

##----------------------------------------------------------------
## example 6: BRF call with misclassification performance
##----------------------------------------------------------------
```

```
o.brf <- imbalanced(f, breast, method = "brf", perf.type = "misclass")

##----------------------------------------------------------------
## example 7: RFQ with optimized threshold
##----------------------------------------------------------------
o.rfq.opt <- imbalanced(f, breast, optimize = TRUE)
plot(o.rfq.opt$gmean, type = "l")

##----------------------------------------------------------------
## example 8: train/test example
##----------------------------------------------------------------

trn <- sample(1:nrow(breast), size = nrow(breast) / 2)
o.trn <- imbalanced(f, breast[trn,], importance = TRUE)
o.tst <- predict(o.trn, breast[-trn,], importance = TRUE)
print(o.trn)
print(o.tst)
print(100 * cbind(o.trn$impo[, 1], o.tst$impo[, 1]))

##----------------------------------------------------------------
## example 9: simulation example using the caret R-package
## simulate classification data with strong and noisy predictors
## create imbalanced data by randomly sampling the class 1 data
##
## illustrates the effectiveness of blocked VIMP
## note that rfsrc uses blocked VIMP by default
##
##----------------------------------------------------------------

if (library("caret", logical.return = TRUE)) {

  ## experimental settings
  n <- 1000
  q <- 20
  ir <- 6
  f <- as.formula(Class ~ .)

  ## simulate the data, create minority class data
  d <- twoClassSim(n, linearVars = 15, noiseVars = q)
  d$Class <- factor(as.numeric(d$Class) - 1)
  idx.0 <- which(d$Class == 0)
  idx.1 <- sample(which(d$Class == 1), sum(d$Class == 1) / ir , replace = FALSE)
  d <- d[c(idx.0,idx.1),, drop = FALSE]

  ## VIMP for BRF with and without blocking
  ## blocked VIMP is a hybrid of Breiman-Cutler/Ishwaran-Kogalur VIMP
  brf <- imbalanced(f, d, method = "brf", importance = TRUE, block.size = 1)
  brfB <- imbalanced(f, d, method = "brf", importance = TRUE, block.size = 10)

  ## VIMP for RFQ with and without blocking
  rfq <- imbalanced(f, d, importance = TRUE, block.size = 1)
  rfqB <- imbalanced(f, d, importance = TRUE, block.size = 10)
  ## compare VIMP values
  imp <- 100 * cbind(brf$importance[, 1], brfB$importance[, 1],
                     rfq$importance[, 1], rfqB$importance[, 1])
  legn <- c("BRF", "BRF-block", "RFQ", "RFQ-block")
  colr <- rep(4,20+q)
  colr[1:20] <- 2
  ylim <- range(c(imp))
  nms <- 1:(20+q)
  par(mfrow=c(2,2))
  barplot(imp[,1],col=colr,las=2,main=legn[1],ylim=ylim,names.arg=nms)
  barplot(imp[,2],col=colr,las=2,main=legn[2],ylim=ylim,names.arg=nms)
  barplot(imp[,3],col=colr,las=2,main=legn[3],ylim=ylim,names.arg=nms)
  barplot(imp[,4],col=colr,las=2,main=legn[4],ylim=ylim,names.arg=nms)

}
```

impute *Impute Only Mode*

Description

Fast imputation mode. A random forest is grown and used to impute missing data. No ensemble estimates or error rates are calculated.

Usage

```
## S3 method for class 'rfsrc'
impute(formula, data,
  ntree = 500, nodesize = 1, nsplit = 10,
  nimpute = 2, fast = FALSE, blocks,
  mf.q, max.iter = 10, eps = 0.01,
  ytry = NULL, always.use = NULL, verbose = TRUE,
  ...)
```

Arguments

formula	A symbolic description of the model to be fit. Can be left unspecified if there are no outcomes or we don't care to distinguish between y-outcomes and x-variables in the imputation. Ignored when using multivariate missForest imputation.
data	Data frame containing the data to be imputed.
ntree	Number of trees to grow.
nodesize	Forest average terminal node size.
nsplit	Non-negative integer value used to specify random splitting.

`nimpute` Number of iterations of the missing data algorithm. Ignored for multivariate missForest; in which case the algorithm iterates until a convergence criteria is achieved (users can however enforce a maximum number of iterations with the option `max.iter`).

`fast` Use fast random forests, `rfsrcFast`, in place of `rfsrc`? Improves speed but is less accurate.

`blocks` Integer value specifying the number of blocks the data should be broken up into (by rows). This can improve computational efficiency when the sample size is large but imputation efficiency decreases. By default, no action is taken if left unspecified.

`mf.q` Use this to turn on missForest (which is off by default). Specifies fraction of variables (between 0 and 1) used as responses in multivariate missForest imputation. Can also be an integer, in which case this equals the number of multivariate responses.

`max.iter` Maximum number of iterations used when implementing multivariate missForest imputation.

`eps` Tolerance value used to determine convergence of multivariate missForest imputation.

`ytry` Number of variables used as pseudo-responses in unsupervised forests. See details below.

`always.use` Character vector of variable names to always be included as a response in multivariate missForest imputation. Does not apply for other imputation methods.

`verbose` Send verbose output to terminal (only applies to multivariate missForest imputation).

`...` Further arguments passed to or from other methods.

Details

1. Grow a forest and use this to impute data. All external calculations such as ensemble calculations, error rates, etc. are turned off. Use this function if your only interest is imputing the data.
2. Split statistics are calculated using non-misssing data only. If a node splits on a variable with missing data, the variable's missing data is imputed by randomly drawing values from non-missing in-bag data. The purpose of this is to make it possible to assign cases to daughter nodes based on the split.
3. If no formula is specified, unsupervised splitting is implemented using a `ytry` value of sqrt(`p`) where `p` equals the number of variables. More precisely, `mtry` variables are selected at random, and for each of these a random subset of `ytry` variables are selected and defined as the multivariate pseudo-responses. A multivariate composite splitting rule of dimension `ytry` is then applied to each of the `mtry` multivariate regression problems and the node split on the variable leading to the best split (Tang and Ishwaran, 2017).
4. If `mf.q` is specified, a multivariate version of missForest imputation (Stekhoven and Buhlmann, 2012) is applied. A fraction `mf.q` of variables are used as multivariate responses and split by the remaining variables using multivariate composite splitting (Tang and Ishwaran, 2017). Missing data for responses are imputed by prediction. The process is repeated using a new

set of variables for responses (mutually exclusive to the previous fit), until all variables have been imputed. This is one iteration. The entire process is repeated, and the algorithm iterated until a convergence criteria is met (specified using options `max.iter` and `eps`). Integer values for `mf.q` are allowed and interpreted as a request that `mf.q` variables be selected for the multivariate response. This is generally the most accurate of all the imputation procedures, but also the most computationally demanding. However see examples below for strategies to increase speed.

5. Prior to imputation, the data is processed and records with all values missing are removed, as are variables having all missing values.
6. If there is no missing data, either before or after processing of the data, the algorithm returns the processed data and no imputation is performed.
7. The default choice `nimpute=2` is chosen for coherence with the default missing data algorithm implemented in grow mode. Thus, if the user imputes data with `nimpute=2` and runs a grow forest using this imputed data, then performance values such as VIMP and error rates will coincide with those obtained by running a grow forest on the original non-imputed data using `na.action = "na.impute"`. Ignored for multivariate missForest.
8. All options are the same as `rfsrc` and the user should consult the `rfsrc` help file for details.

Value

Invisibly, the data frame containing the orginal data with imputed data overlayed.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H., Kogalur U.B., Blackstone E.H. and Lauer M.S. (2008). Random survival forests, *Ann. App. Statist.*, 2:841-860.

Stekhoven D.J. and Buhlmann P. (2012). MissForest–non-parametric missing value imputation for mixed-type data. *Bioinformatics*, 28(1):112-118.

Tang F. and Ishwaran H. (2017). Random forest missing data algorithms. *Statistical Analysis and Data Mining*, 10, 363-377.

See Also

`rfsrc` `rfsrcFast`

Examples

```
## ------------------------------------------------------------
## example of survival imputation
## ------------------------------------------------------------

## default everything - unsupervised splitting
data(pbc, package = "randomForestSRC")
```

```
pbc1.d <- impute(data = pbc)

## imputation using outcome splitting
f <- as.formula(Surv(days, status) ~ .)
pbc2.d <- impute(f, data = pbc, nsplit = 3)

## random splitting can be reasonably good
pbc3.d <- impute(f, data = pbc, splitrule = "random", nimpute = 5)

## ------------------------------------------------------------
## example of regression imputation
## ------------------------------------------------------------

air1.d <- impute(data = airquality, nimpute = 5)
air2.d <- impute(Ozone ~ ., data = airquality, nimpute = 5)
air3.d <- impute(Ozone ~ ., data = airquality, fast = TRUE)

## ------------------------------------------------------------
## multivariate missForest imputation
## ------------------------------------------------------------

data(pbc, package = "randomForestSRC")

## missForest algorithm - uses 1 variable at a time for the response
pbc.d <- impute(data = pbc, mf.q = 1)

## multivariate missForest - use 10 percent of variables as responses
## i.e. multivariate missForest
pbc.d <- impute(data = pbc, mf.q = .01)

## missForest but faster by using random splitting
pbc.d <- impute(data = pbc, mf.q = 1, splitrule = "random")

## missForest but faster by increasing nodesize
pbc.d <- impute(data = pbc, mf.q = 1, nodesize = 20, splitrule = "random")

## missForest but faster by using rfsrcFast
pbc.d <- impute(data = pbc, mf.q = 1, fast = TRUE)
```

`max.subtree` *Acquire Maximal Subtree Information*

Description

Extract maximal subtree information from a RF-SRC object. Used for variable selection and identifying interactions between variables.

Usage

```
## S3 method for class 'rfsrc'
max.subtree(object,
  max.order = 2, sub.order = FALSE, conservative = FALSE, ...)
```

Arguments

`object` An object of class `(rfsrc, grow)` or `(rfsrc, forest)`.

`max.order` Non-negative integer specifying the target number of order depths. Default is to return the first and second order depths. Used to identify predictive variables. Setting '`max.order=0`' returns the first order depth for each variable by tree. A side effect is that '`conservative`' is automatically set to `FALSE`.

`sub.order` Set this value to `TRUE` to return the minimal depth of each variable relative to another variable. Used to identify interrelationship between variables. See details below.

`conservative` If `TRUE`, the threshold value for selecting variables is calculated using a conservative marginal approximation to the minimal depth distribution (the method used in Ishwaran et al. 2010). Otherwise, the minimal depth distribution is the tree-averaged distribution. The latter method tends to give larger threshold values and discovers more variables, especially in high-dimensions.

`...` Further arguments passed to or from other methods.

Details

The maximal subtree for a variable *x* is the largest subtree whose root node splits on *x*. Thus, all parent nodes of *x*'s maximal subtree have nodes that split on variables other than *x*. The largest maximal subtree possible is the root node. In general, however, there can be more than one maximal subtree for a variable. A maximal subtree may also not exist if there are no splits on the variable. See Ishwaran et al. (2010, 2011) for details.

The minimal depth of a maximal subtree (the first order depth) measures predictiveness of a variable *x*. It equals the shortest distance (the depth) from the root node to the parent node of the maximal subtree (zero is the smallest value possible). The smaller the minimal depth, the more impact *x* has on prediction. The mean of the minimal depth distribution is used as the threshold value for deciding whether a variable's minimal depth value is small enough for the variable to be classified as strong.

The second order depth is the distance from the root node to the second closest maximal subtree of *x*. To specify the target order depth, use the `max.order` option (e.g., setting '`max.order=2`' returns the first and second order depths). Setting '`max.order=0`' returns the first order depth for each variable for each tree.

Set '`sub.order=TRUE`' to obtain the minimal depth of a variable relative to another variable. This returns a `pxp` matrix, where `p` is the number of variables, and entries [i][j] are the normalized relative minimal depth of a variable [j] within the maximal subtree for variable [i], where normalization adjusts for the size of [i]'s maximal subtree. Entry [i][i] is the normalized minimal depth of i relative to the root node. The matrix should be read by looking across rows (not down columns) and identifies interrelationship between variables. Small [i][j] entries indicate interactions. See `find.interaction` for related details.

For competing risk data, maximal subtree analyses are unconditional (i.e., they are non-event specific).

Value

Invisibly, a list with the following components:

`order` Order depths for a given variable up to `max.order` averaged over a tree and the forest. Matrix of dimension `pxmax.order`. If '`max.order=0`', a matrix of `pxntree` is returned containing the first order depth for each variable by tree.

`count` Averaged number of maximal subtrees, normalized by the size of a tree, for each variable.

`nodes.at.depth` Number of non-terminal nodes by depth for each tree.

`sub.order` Average minimal depth of a variable relative to another variable. Can be `NULL`.

`threshold` Threshold value (the mean minimal depth) used to select variables.

`threshold.1se` Mean minimal depth plus one standard error.

`topvars` Character vector of names of the final selected variables.

`topvars.1se` Character vector of names of the final selected variables using the 1se threshold rule.

`percentile` Minimal depth percentile for each variable.

`density` Estimated minimal depth density.

`second.order.threshold`
Threshold for second order depth.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H., Kogalur U.B., Gorodeski E.Z, Minn A.J. and Lauer M.S. (2010). High-dimensional variable selection for survival data. *J. Amer. Statist. Assoc.*, 105:205-217.

Ishwaran H., Kogalur U.B., Chen X. and Minn A.J. (2011). Random survival forests for high-dimensional data. *Statist. Anal. Data Mining*, 4:115-132.

See Also

`find.interaction`, `var.select`, `vimp`

Examples

```
## ------------------------------------------------------------
## survival analysis
## first and second order depths for all variables
## ------------------------------------------------------------
```

```
data(veteran, package = "randomForestSRC")
v.obj <- rfsrc(Surv(time, status) ~ . , data = veteran)
v.max <- max.subtree(v.obj)

# first and second order depths
print(round(v.max$order, 3))

# the minimal depth is the first order depth
print(round(v.max$order[, 1], 3))

# strong variables have minimal depth less than or equal
# to the following threshold
print(v.max$threshold)

# this corresponds to the set of variables
print(v.max$topvars)

## ------------------------------------------------------------
## regression analysis
## try different levels of conservativeness
## ------------------------------------------------------------

mtcars.obj <- rfsrc(mpg ~ ., data = mtcars)
max.subtree(mtcars.obj)$topvars
max.subtree(mtcars.obj, conservative = TRUE)$topvars
```

nutrigenomic *Nutrigenomic Study*

Description

Study the effects of five diet treatments on 21 liver lipids and 120 hepatic gene expression in wild-type and PPAR-alpha deficient mice. We use a multivariate mixed random forest analysis by regressing gene expression, diet and genotype (the x-variables) on lipid expressions (the multivariate y-responses).

References

Martin P.G. et al. (2007). Novel aspects of PPAR-alpha-mediated regulation of lipid and xenobiotic metabolism revealed through a nutrigenomic study. *Hepatology*, 45(3), 767–777.

Examples

```
## ------------------------------------------------------------
## multivariate mixed forests
## lipids used as the multivariate y-responses
## ------------------------------------------------------------
## load the data
data(nutrigenomic, package = "randomForestSRC")

## multivariate mixed forest call
mv.obj <- rfsrc(get.mv.formula(colnames(nutrigenomic$lipids)),
            data.frame(do.call(cbind, nutrigenomic)),
            importance=TRUE, nsplit = 10)

## ------------------------------------------------------------
## plot the standarized performance and VIMP values
## ------------------------------------------------------------

## acquire the error rate for each of the 21-coordinates
## standardize to allow for comparison across coordinates
serr <- get.mv.error(mv.obj, standardize = TRUE)

## acquire standardized VIMP
svimp <- get.mv.vimp(mv.obj, standardize = TRUE)

par(mfrow = c(1,2))
plot(serr, xlab = "Lipids", ylab = "Standardized Performance")
matplot(svimp, xlab = "Genes/Diet/Genotype", ylab = "Standardized VIMP")
```

partial *Acquire Partial Effect of a Variable*

Description

Acquire the partial effect of a variable on the ensembles.

Usage

```
partial.rfsrc(object, oob = TRUE, m.target = NULL,
  partial.type = NULL, partial.xvar = NULL, partial.values = NULL,
  partial.xvar2 = NULL, partial.values2 = NULL,
  partial.time = NULL, get.tree = NULL, seed = NULL, do.trace = FALSE, ...)
```

Arguments

object	An object of class (rfsrc, grow).
oob	By default out-of-bag values are returned, but inbag values can be requested by setting this option to FALSE.
m.target	Character value for multivariate families specifying the target outcome to be used. If left unspecified, the algorithm will choose a default target.
partial.type	Character value of the type of predicted value. See details below.
partial.xvar	Character value specifying the single primary partial x-variable to be used.

`partial.values` Vector of values that the primary partialy x-variable will assume.

`partial.xvar2` Vector of character values specifying the second order x-variables to be used.

`partial.values2`
Vector of values that the second order x-variables will assume. Each second order x-variable can only assume a single value. This the length of `partial.xvar2` and `partial.values2` will be the same. In addition, the user must do the appropriate conversion for factors, and represent a value as a numeric element.

`partial.time` For survival families, the time at which the predicted survival value is evaluated at (depends on `partial.type`).

`get.tree` Vector of integer(s) identifying trees over which the partial values are calculated over. By default, uses all trees in the forest.

`seed` Negative integer specifying seed for the random number generator.

`do.trace` Number of seconds between updates to the user on approximate time to completion.

`...` Further arguments passed to or from other methods.

Details

Out-of-bag (OOB) values are returned by default.

For factors, the partial value should be encoded as a positive integer reflecting the level number of the factor. The actual label of the factor should not be used.

A list of length equal to the number of outcomes (length is one for univariate families) with entries depending on the underlying family:

1. For regression, the predicted response is returned of dim `[n] x [length(partial.values)]`.
2. For classification, the predicted probabilities are returned of dim `[n] x [1 + yvar.nlevels[.]] x [length(partial`
3. For survival, the choices are:
   - Relative frequency of mortality (`rel.freq`) or mortality (`mort`) is of dim `[n] x [length(partial.values`
   - The cumulative hazard function (`chf`) is of dim `[n] x [length(partial.time)] x [length(partial.values)]`.
   - The survival function (`surv`) is of dim `[n] x [length(partial.time)] x [length(partial.values)]`
4. For competing risks, the choices are:
   - The expected number of life years lost (`years.lost`) is of dim `[n] x [length(event.info$event.type)] x [length(partial.values)]`.
   - The cumulative incidence function (`cif`) is of dim `[n] x [length(partial.time)] x [length(event.info$event.type)] x [length(partial.values)]`.
   - The cumulative hazard function (`chf`) is of dim `[n] x [length(partial.time)] x [length(event.info$event` `x [length(partial.values)]`.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H., Kogalur U.B. (2007). Random survival forests for R, *Rnews*, 7(2):25-31.

Ishwaran H., Kogalur U.B., Blackstone E.H. and Lauer M.S. (2008). Random survival forests, *Ann. App. Statist.*, 2:841-860.

See Also

plot.variable.rfsrc

Examples

```
## ------------------------------------------------------------
## survival/competing risk
## ------------------------------------------------------------

## survival
data(veteran, package = "randomForestSRC")
v.obj <- rfsrc(Surv(time,status)~., veteran, nsplit = 10, ntree = 100)
partial.obj <- partial(v.obj,
  partial.type = "rel.freq",
  partial.xvar = "age",
  partial.values = v.obj$xvar[, "age"],
  partial.time = v.obj$time.interest)

## competing risks
data(follic, package = "randomForestSRC")
follic.obj <- rfsrc(Surv(time, status) ~ ., follic, nsplit = 3, ntree = 100)
partial.obj <- partial(follic.obj,
  partial.type = "cif",
  partial.xvar = "age",
  partial.values = follic.obj$xvar[, "age"],
  partial.time = follic.obj$time.interest)

## regression
airq.obj <- rfsrc(Ozone ~ ., data = airquality)
partial.obj <- partial(airq.obj,
  partial.xvar = "Wind",
  partial.values = airq.obj$xvar[, "Wind"],
  oob = FALSE)

## classification
iris.obj <- rfsrc(Species ~., data = iris)
partial.obj <- partial(iris.obj,
  partial.xvar = "Sepal.Length",
  partial.values = iris.obj$xvar[, "Sepal.Length"])

## multivariate mixed outcomes
mtcars2 <- mtcars
mtcars2$carb <- factor(mtcars2$carb)
mtcars2$cyl <- factor(mtcars2$cyl)
```

```
mtcars.mix <- rfsrc(Multivar(carb, mpg, cyl) ~ ., data = mtcars2)
partial.obj <- partial(mtcars.mix,
  partial.xvar = "disp",
  partial.values = mtcars.mix$xvar[, "disp"])

## second order variable specification
mtcars.obj <- rfsrc(mpg ~., data = mtcars)
partial.obj <- partial(mtcars.obj,
  partial.xvar = "cyl",
  partial.values = c(4, 8),
  partial.xvar2 = c("gear", "disp", "carb"),
  partial.values2 = c(4, 200, 3))
```

pbc *Primary Biliary Cirrhosis (PBC) Data*

Description

Data from the Mayo Clinic trial in primary biliary cirrhosis (PBC) of the liver conducted between 1974 and 1984. A total of 424 PBC patients, referred to Mayo Clinic during that ten-year interval, met eligibility criteria for the randomized placebo controlled trial of the drug D-penicillamine. The first 312 cases in the data set participated in the randomized trial and contain largely complete data.

Source

Flemming and Harrington, 1991, Appendix D.1.

References

Flemming T.R and Harrington D.P., (1991) *Counting Processes and Survival Analysis.* New York: Wiley.

Examples

```
data(pbc, package = "randomForestSRC")
pbc.obj <- rfsrc(Surv(days, status) ~ ., pbc, nsplit = 3)
```

plot.competing.risk *Plots for Competing Risks*

Description

Plot the ensemble cumulative incidence function (CIF) and cause-specific cumulative hazard function (CSCHF) from a competing risk analysis.

Usage

```
## S3 method for class 'rfsrc'
plot.competing.risk(x, plots.one.page = FALSE, ...)
```

Arguments

x	An object of class (rfsrc, grow) or (rfsrc, predict).
plots.one.page	Should plots be placed on one page?
...	Further arguments passed to or from other methods.

Details

Ensemble ensemble CSCHF and CIF functions for each event type. Does not apply to right-censored data. Whenever possible, out-of-bag (OOB) values are displayed.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H., Gerds T.A., Kogalur U.B., Moore R.D., Gange S.J. and Lau B.M. (2014). Random survival forests for competing risks. *Biostatistics*, 15(4):757-773.

See Also

follic, hd, rfsrc, wihs

Examples

```
## ------------------------------------------------------------
## follicular cell lymphoma
## ------------------------------------------------------------

  data(follic, package = "randomForestSRC")
  follic.obj <- rfsrc(Surv(time, status) ~ ., follic, nsplit = 3, ntree = 100)
  plot.competing.risk(follic.obj)
```

```
## ------------------------------------------------------------
## competing risk analysis of pbc data from the survival package
## events are transplant (1) and death (2)
## ------------------------------------------------------------

if (library("survival", logical.return = TRUE)) {
   data(pbc, package = "survival")
   pbc$id <- NULL
   plot.competing.risk(rfsrc(Surv(time, status) ~ ., pbc))
}
```

plot.rfsrc *Plot Error Rate and Variable Importance from a RF-SRC analysis*

Description

Plot out-of-bag (OOB) error rates and variable importance (VIMP) from a RF-SRC analysis. This is the default plot method for the package.

Usage

```
## S3 method for class 'rfsrc'
plot(x, m.target = NULL,
  plots.one.page = TRUE, sorted = TRUE, verbose = TRUE,  ...)
```

Arguments

`x` An object of class `(rfsrc, grow)`, `(rfsrc, synthetic)`, or `(rfsrc, predict)`.

`m.target` Character value for multivariate families specifying the target outcome to be used. If left unspecified, the algorithm will choose a default target.

`plots.one.page` Should plots be placed on one page?

`sorted` Should variables be sorted by importance values?

`verbose` Should VIMP be printed?

`...` Further arguments passed to or from other methods.

Details

Plot cumulative OOB error rates as a function of number of trees and variable importance (VIMP) if available. Note that the default settings are now such that the error rate is no longer calculated on every tree and VIMP is only calculated if requested. To get OOB error rates for ever tree, use the option `block.size = 1` when growing or restoring the forest. Likewise, to view VIMP, use the option `importance` when growing or restoring the forest.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Breiman L. (2001). Random forests, *Machine Learning*, 45:5-32.

Ishwaran H. and Kogalur U.B. (2007). Random survival forests for R, *Rnews*, 7(2):25-31.

See Also

predict.rfsrc, rfsrc

Examples

```
## ------------------------------------------------------------
## classification example
## ------------------------------------------------------------

iris.obj <- rfsrc(Species ~ ., data = iris,
     block.size = 1, importance = TRUE)
plot(iris.obj)

## ------------------------------------------------------------
## competing risk example
## ------------------------------------------------------------

## use the pbc data from the survival package
## events are transplant (1) and death (2)
if (library("survival", logical.return = TRUE)) {
  data(pbc, package = "survival")
  pbc$id <- NULL
  plot(rfsrc(Surv(time, status) ~ ., pbc, block.size = 1))
}

## ------------------------------------------------------------
## multivariate mixed forests
## ------------------------------------------------------------

mtcars.new <- mtcars
mtcars.new$cyl <- factor(mtcars.new$cyl)
mtcars.new$carb <- factor(mtcars.new$carb, ordered = TRUE)
mv.obj <- rfsrc(cbind(carb, mpg, cyl) ~., data = mtcars.new, block.size = 1)
plot(mv.obj, m.target = "carb")
plot(mv.obj, m.target = "mpg")
plot(mv.obj, m.target = "cyl")
```

plot.subsample *Plot Subsampled VIMP Confidence Intervals*

Description

Plots VIMP (variable importance) confidence regions obtained from subsampling a forest.

Usage

```
## S3 method for class 'rfsrc'
plot.subsample(x, alpha = .05,
 standardize = TRUE, normal = TRUE, jknife = TRUE,
 target, m.target = NULL, pmax = 75, main = "", cex = 1, ...)
```

Arguments

`x`	An object obtained from calling `subample`.
`alpha`	Desired level of significance.
`standardize`	Standardize VIMP? For regression families, VIMP is standardized by dividing by the variance and then multipled by 100. For all other families, VIMP is scaled by 100.
`normal`	Use parametric normal confidence regions or nonparametric regions? Generally, parametric regions perform better.
`jknife`	Use the delete-d jackknife variance estimator?
`target`	For classification families, an integer or character value specifying the class VIMP will be conditioned on (default is to use unconditional VIMP). For competing risk families, an integer value between 1 and `J` indicating the event VIMP is requested, where `J` is the number of event types. The default is to use the first event.
`m.target`	Character value for multivariate families specifying the target outcome to be used. If left unspecified, the algorithm will choose a default target.
`pmax`	Trims the data to this number of variables (sorted by VIMP).
`main`	Title used for plot.
`cex`	Character expansion used for variable names.
`...`	Further arguments that can be passed to `bxp`.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H. and Lu M. (2017). Standard errors and confidence intervals for variable importance in random forest regression, classification, and survival.

Politis, D.N. and Romano, J.P. (1994). Large sample confidence regions based on subsamples under minimal assumptions. *The Annals of Statistics*, 22(4):2031-2050.

Shao, J. and Wu, C.J. (1989). A general theory for jackknife variance estimation. *The Annals of Statistics*, 17(3):1176-1197.

See Also

subsample

Examples

```
o <- rfsrc(Ozone ~ ., airquality)
oo <- subsample(o)
plot.subsample(oo)
plot.subsample(oo, jknife = FALSE)
plot.subsample(oo, alpha = .01)
plot(oo)
```

---

plot.survival *Plot of Survival Estimates*

Description

Plot various survival estimates.

Usage

```
## S3 method for class 'rfsrc'
plot.survival(x, plots.one.page = TRUE,
  show.plots = TRUE, subset, collapse = FALSE,
  haz.model = c("spline", "ggamma", "nonpar", "none"),
  k = 25, span = "cv", cens.model = c("km", "rfsrc"), ...)
```

Arguments

`x` An object of class (`rfsrc, grow`) or (`rfsrc, predict`).

`plots.one.page` Should plots be placed on one page?

`show.plots` Should plots be displayed?

`subset` Vector indicating which individuals we want estimates for. All individuals are used if not specified.

`collapse` Collapse the survival and cumulative hazard function across the individuals specified by ‘`subset`’? Only applies when ‘`subset`’ is specified.

`haz.model` Method for estimating the hazard. See details below. Applies only when ‘`subset`’ is specified.

`k` The number of natural cubic spline knots used for estimating the hazard function. Applies only when ‘`subset`’ is specified.

`span` The fraction of the observations in the span of Friedman’s super-smoother used for estimating the hazard function. Applies only when ‘`subset`’ is specified.

`cens.model` Method for estimating the censoring distribution used in the inverse probability of censoring weights (IPCW) for the Brier score:

`km`: Uses the Kaplan-Meier estimator.

`rfscr`: Uses random survival forests.

`...` Further arguments passed to or from other methods.

Details

If '`subset`' is not specified, generates the following three plots (going from top to bottom, left to right):

1. Forest estimated survival function for each individual (thick red line is overall ensemble survival, thick green line is Nelson-Aalen estimator).
2. Brier score (0=perfect, 1=poor, and 0.25=guessing) stratified by ensemble mortality. Based on the IPCW method described in Gerds et al. (2006). Stratification is into 4 groups corresponding to the 0-25, 25-50, 50-75 and 75-100 percentile values of mortality. Red line is the overall (non-stratified) Brier score.
3. Plot of mortality of each individual versus observed time. Points in blue correspond to events, black points are censored observations.

When '`subset`' is specified, then for each individual in '`subset`', the following three plots are generated:

1. Forest estimated survival function.
2. Forest estimated cumulative hazard function (CHF) (displayed using black lines). Blue lines are the CHF from the estimated hazard function. See the next item.
3. A smoothed hazard function derived from the forest estimated CHF (or survival function). The default method, '`haz.model="spline"`', models the log CHF using natural cubic splines as described in Royston and Parmar (2002). The lasso is used for model selection, implemented using the `glmnet` package (this package must be installed for this option to work). If '`haz.model="ggamma"`', a three-parameter generalized gamma distribution (using the parameterization described in Cox et al, 2007) is fit to the smoothed forest survival function, where smoothing is imposed using Friedman's supersmoother (implemented by `supsmu`). If '`haz.model="nonpar"`', Friedman's supersmoother is applied to the forest estimated hazard function (obtained by taking the crude derivative of the smoothed forest CHF). Finally, setting '`haz.model="none"`' suppresses hazard estimation and no hazard estimate is provided.

   At this time, please note that all hazard estimates are considered experimental and users should interpret the results with caution.

Note that when the object `x` is of class `(rfsrc, predict)` not all plots will be produced. In particular, Brier scores are not calculated.

Only applies to survival families. In particular, fails for competing risk analyses. Use `plot.competing.risk` in such cases.

Whenever possible, out-of-bag (OOB) values are used.

Value

Invisibly, the conditional and unconditional Brier scores, and the integrated Brier score (if they are available).

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Cox C., Chu, H., Schneider, M. F. and Munoz, A. (2007). Parametric survival analysis and taxonomy of hazard functions for the generalized gamma distribution. Statistics in Medicine 26:4252-4374.

Gerds T.A and Schumacher M. (2006). Consistent estimation of the expected Brier score in general survival models with right-censored event times, *Biometrical J.*, 6:1029-1040.

Graf E., Schmoor C., Sauerbrei W. and Schumacher M. (1999). Assessment and comparison of prognostic classification schemes for survival data, *Statist. in Medicine*, 18:2529-2545.

Ishwaran H. and Kogalur U.B. (2007). Random survival forests for R, *Rnews*, 7(2):25-31.

Royston P. and Parmar M.K.B. (2002). Flexible parametric proportional-hazards and proportional-odds models for censored survival data, with application to prognostic modelling and estimation of treatment effects, *Statist. in Medicine*, 21::2175-2197.

See Also

plot.competing.risk, predict.rfsrc, rfsrc

Examples

```
## veteran data
data(veteran, package = "randomForestSRC")
plot.survival(rfsrc(Surv(time, status)~ ., veteran), cens.model = "rfsrc")

## pbc data
data(pbc, package = "randomForestSRC")
pbc.obj <- rfsrc(Surv(days, status) ~ ., pbc)

# default spline approach
plot.survival(pbc.obj, subset = 3)
plot.survival(pbc.obj, subset = 3, k = 100)

# three-parameter generalized gamma is approximately the same
# but notice that its CHF estimate (blue line) is not as accurate
plot.survival(pbc.obj, subset = 3, haz.model = "ggamma")

# nonparametric method is too wiggly or undersmooths
plot.survival(pbc.obj, subset = 3, haz.model = "nonpar", span = 0.1)
plot.survival(pbc.obj, subset = 3, haz.model = "nonpar", span = 0.8)
```

`plot.variable` *Plot Marginal Effect of Variables*

Description

Plot the marginal effect of an x-variable on the class probability (classification), response (regression), mortality (survival), or the expected years lost (competing risk). Users can select between marginal (unadjusted, but fast) and partial plots (adjusted, but slower).

Usage

```
## S3 method for class 'rfsrc'
plot.variable(x, xvar.names, target,
  m.target = NULL, time, surv.type = c("mort", "rel.freq",
  "surv", "years.lost", "cif", "chf"), class.type =
  c("prob", "bayes"), partial = FALSE, oob = TRUE,
  show.plots = TRUE, plots.per.page = 4, granule = 5, sorted = TRUE,
  nvar, npts = 25, smooth.lines = FALSE, subset, ...)
```

Arguments

`x` An object of class (`rfsrc, grow`), (`rfsrc, synthetic`), or (`rfsrc, plot.variable`). See the examples below for illustration of the latter.

`xvar.names` Names of the x-variables to be used.

`target` For classification, an integer or character value specifying the class to focus on (defaults to the first class). For competing risks, an integer value between 1 and `J` indicating the event of interest, where `J` is the number of event types. The default is to use the first event type.

`m.target` Character value for multivariate families specifying the target outcome to be used. If left unspecified, the algorithm will choose a default target.

`time` For survival, the time at which the predicted survival value is evaluated at (depends on `surv.type`).

`surv.type` For survival, specifies the predicted value. See details below.

`class.type` For classification, specifies the predicted value. See details below.

`partial` Should partial plots be used?

`oob` OOB (TRUE) or in-bag (FALSE) predicted values.

`show.plots` Should plots be displayed?

`plots.per.page` Integer value controlling page layout.

`granule` Integer value controlling whether a plot for a specific variable should be treated as a factor and therefore given as a boxplot. Larger values coerce boxplots.

`sorted` Should variables be sorted by importance values.

`nvar` Number of variables to be plotted. Default is all.

`npts` Maximum number of points used when generating partial plots for continuous variables.

`smooth.lines` Use lowess to smooth partial plots.

`subset` Vector indicating which rows of the x-variable matrix `x$xvar` to use. All rows are used if not specified. Do not define subset based on the original data (which could have been processed due to missing values or for other reasons in the previous forest call) but define subset based on the rows of `x$xvar`.

`...` Further arguments passed to or from other methods.

Details

The vertical axis displays the ensemble predicted value, while x-variables are plotted on the horizontal axis.

1. For regression, the predicted response is used.
2. For classification, it is the predicted class probability specified by '`target`', or the class of maximum probability depending on '`class.type`'.
3. For multivariate families, it is the predicted value of the outcome specified by '`m.target`' and if that is a classification outcome, by '`target`'.
4. For survival, the choices are:
   - Mortality (`mort`).
   - Relative frequency of mortality (`rel.freq`).
   - Predicted survival (`surv`), where the predicted survival is for the time point specified using `time` (the default is the median follow up time).
5. For competing risks, the choices are:
   - The expected number of life years lost (`years.lost`).
   - The cumulative incidence function (`cif`).
   - The cumulative hazard function (`chf`).

   In all three cases, the predicted value is for the event type specified by '`target`'. For `cif` and `chf` the quantity is evaluated at the time point specified by `time`.

For partial plots use '`partial=TRUE`'. Their interpretation are different than marginal plots. The y-value for a variable X, evaluated at $X = x$, is

$$\tilde{f}(x) = \frac{1}{n}\sum_{i=1}^{n} \hat{f}(x, x_{i,o}),$$

where $x_{i,o}$ represents the value for all other variables other than X for individual i and $\hat{f}$ is the predicted value. Generating partial plots can be very slow. Choosing a small value for `npts` can speed up computational times as this restricts the number of distinct x values used in computing $\tilde{f}$.

For continuous variables, red points are used to indicate partial values and dashed red lines indicate a smoothed error bar of +/- two standard errors. Black dashed line are the partial values. Set '`smooth.lines=TRUE`' for lowess smoothed lines. For discrete variables, partial values are indicated using boxplots with whiskers extending out approximately two standard errors from the mean. Standard errors are meant only to be a guide and should be interpreted with caution.

Partial plots can be slow. Setting 'npts' to a smaller number can help.

For greater customization and flexibility in partial plot calls, consider using the function partial.rfsrc which provides a direct interface for calculating partial plot data.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Friedman J.H. (2001). Greedy function approximation: a gradient boosting machine, *Ann. of Statist.*, 5:1189-1232.

Ishwaran H., Kogalur U.B. (2007). Random survival forests for R, *Rnews*, 7(2):25-31.

Ishwaran H., Kogalur U.B., Blackstone E.H. and Lauer M.S. (2008). Random survival forests, *Ann. App. Statist.*, 2:841-860.

Ishwaran H., Gerds T.A., Kogalur U.B., Moore R.D., Gange S.J. and Lau B.M. (2014). Random survival forests for competing risks. *Biostatistics*, 15(4):757-773.

See Also

rfsrc, rfsrcSyn, partial.rfsrc, predict.rfsrc

Examples

```
## ------------------------------------------------------------
## survival/competing risk
## ------------------------------------------------------------

## survival
data(veteran, package = "randomForestSRC")
v.obj <- rfsrc(Surv(time,status)~., veteran, ntree = 100)
plot.variable(v.obj, plots.per.page = 3)
plot.variable(v.obj, plots.per.page = 2, xvar.names = c("trt", "karno", "age"))
plot.variable(v.obj, surv.type = "surv", nvar = 1, time = 200)
plot.variable(v.obj, surv.type = "surv", partial = TRUE, smooth.lines = TRUE)
plot.variable(v.obj, surv.type = "rel.freq", partial = TRUE, nvar = 2)

## example of plot.variable calling a pre-processed plot.variable object
p.v <- plot.variable(v.obj, surv.type = "surv", partial = TRUE, smooth.lines = TRUE)
plot.variable(p.v)
p.v$plots.per.page <- 1
p.v$smooth.lines <- FALSE
plot.variable(p.v)

## competing risks
data(follic, package = "randomForestSRC")
follic.obj <- rfsrc(Surv(time, status) ~ ., follic, nsplit = 3, ntree = 100)
plot.variable(follic.obj, target = 2)
```

```
## ------------------------------------------------------------
## regression
## ------------------------------------------------------------

## airquality
airq.obj <- rfsrc(Ozone ~ ., data = airquality)
plot.variable(airq.obj, partial = TRUE, smooth.lines = TRUE)
plot.variable(airq.obj, partial = TRUE, subset = airq.obj$xvar$Solar.R < 200)

## motor trend cars
mtcars.obj <- rfsrc(mpg ~ ., data = mtcars)
plot.variable(mtcars.obj, partial = TRUE, smooth.lines = TRUE)

## ------------------------------------------------------------
## classification
## ------------------------------------------------------------

## iris
iris.obj <- rfsrc(Species ~., data = iris)
plot.variable(iris.obj, partial = TRUE)

## motor trend cars: predict number of carburetors
mtcars2 <- mtcars
mtcars2$carb <- factor(mtcars2$carb,
   labels = paste("carb", sort(unique(mtcars$carb))))
mtcars2.obj <- rfsrc(carb ~ ., data = mtcars2)
plot.variable(mtcars2.obj, partial = TRUE)

## ------------------------------------------------------------
## multivariate regression
## ------------------------------------------------------------
mtcars.mreg <- rfsrc(Multivar(mpg, cyl) ~., data = mtcars)
plot.variable(mtcars.mreg, m.target = "mpg", partial = TRUE, nvar = 1)
plot.variable(mtcars.mreg, m.target = "cyl", partial = TRUE, nvar = 1)


## ------------------------------------------------------------
## multivariate mixed outcomes
## ------------------------------------------------------------
mtcars2 <- mtcars
mtcars2$carb <- factor(mtcars2$carb)
mtcars2$cyl <- factor(mtcars2$cyl)
mtcars.mix <- rfsrc(Multivar(carb, mpg, cyl) ~ ., data = mtcars2)
plot.variable(mtcars.mix, m.target = "cyl", target = "4", partial = TRUE, nvar = 1)
plot.variable(mtcars.mix, m.target = "cyl", target = 2, partial = TRUE, nvar = 1)
```

predict.rfsrc *Prediction for Random Forests for Survival, Regression, and Classification*

Description

Obtain predicted values using a forest. Also returns performance values if the test data contains y-outcomes.

Usage

```
## S3 method for class 'rfsrc'
predict(object,
  newdata,
  m.target = NULL,
  importance = c(FALSE, TRUE, "none", "permute", "random", "anti"),
  get.tree = NULL, block.size = NULL,
  ensemble = NULL,
  na.action = c("na.omit", "na.impute"),
  outcome = c("train", "test"),
  proximity = FALSE,
  forest.wt = FALSE,
  ptn.count = 0,

  distance = FALSE,
  var.used = c(FALSE, "all.trees", "by.tree"),
  split.depth = c(FALSE, "all.trees", "by.tree"), seed = NULL,
  do.trace = FALSE, membership = FALSE, statistics = FALSE,
  ...)
```

Arguments

`object` An object of class `(rfsrc, grow)` or `(rfsrc, forest)`.

`newdata` Test data. If missing, the original grow (training) data is used.

`ensemble` Optional parameter for specifying the type of ensemble. Can be `oob`, `inbag` or `all`, although not all choices will be applicable depending on the setting (e.g. when predicting on newdata there is no notion of out-of-bag).

`m.target` Character vector for multivariate families specifying the target outcomes to be used. The default is to use all coordinates.

`importance` Method for computing variable importance (VIMP) when test data contains y-outcomes values. Also see `vimp` for more flexibility, including joint vimp calculations.

`get.tree` Vector of integer(s) identifying trees over which the ensembles are calculated over. By default, uses all trees in the forest. As an example, the user can extract the ensemble, the variable importance, or proximity from a single tree (or several trees). Note that `block.size` will be over-ridden so that it is no larger than the requested number of trees.

`block.size` Should the error rate be calculated on every tree? When `NULL`, it will only be calculated on the last tree. To view the error rate on every nth tree, set the value to an integer between `1` and `ntree`. If importance is requested, VIMP is calculated in "blocks" of size equal to `block.size`, thus resulting in a useful compromise between ensemble and permutation VIMP.

`na.action` Missing value action. The default `na.omit` removes the entire record if even one of its entries is `NA`. Selecting '`na.impute`' imputes the test data.

`outcome` Determines whether the y-outcomes from the training data or the test data are used to calculate the predicted value. The default and natural choice is `train` which uses the original training data. Option is ignored when `newdata` is missing as the training data is used for the test data in such settings. The option is also ignored whenever the test data is devoid of y-outcomes. See the details and examples below for more information.

`proximity` Should proximity between test observations be calculated? Possible choices are `"inbag"`, `"oob"`, `"all"`, `TRUE`, or `FALSE` — but some options may not be valid and will depend on the context of the predict call. The safest choice is `TRUE` if proximity is desired.

`distance` Should distance between test observations be calculated? Possible choices are `"inbag"`, `"oob"`, `"all"`, `TRUE`, or `FALSE` — but some options may not be valid and will depend on the context of the predict call. The safest choice is `TRUE` if distance is desired.

`forest.wt` Should the forest weight matrix for test observations be calculated? Choices are the same as proximity.

`ptn.count` The number of terminal nodes that each tree in the grow forest should be pruned back to. The terminal node membership for the pruned forest is returned but no other action is taken. The default is `ptn.count=0` which does no pruning.

`var.used` Record the number of times a variable is split?

`split.depth` Return minimal depth for each variable for each case?

`seed` Negative integer specifying seed for the random number generator.

`do.trace` Number of seconds between updates to the user on approximate time to completion.

`membership` Should terminal node membership and inbag information be returned?

`statistics` Should split statistics be returned? Values can be parsed using `stat.split`.

`...` Further arguments passed to or from other methods.

Details

Predicted values are obtained by dropping test data down the grow forest (the forest grown using the training data). The overall error rate and VIMP are also returned if the test data contains y-outcome values. Single as well as joint VIMP measures can be requested. Note that calculating VIMP can be computationally expensive (especially when the dimension is high), thus if VIMP is not needed, computational times can be significantly improved by setting '`importance="none"`' which turns VIMP off.

Setting '`na.action="na.impute"`' imputes missing test data (x-variables and/or y-outcomes). Imputation uses the grow-forest and only training data is used to impute test data to avoid biasing error rates and VIMP (Ishwaran et al. 2008). See the `rfsrc` help file for details.

If no test data is provided, then the original training data is used and the code reverts to restore mode allowing the user to restore the original grow forest. This is useful, because it gives the user the ability to extract outputs from the forest that were not asked for in the original grow call.

If '`outcome="test"`', the predictor is calculated by using y-outcomes from the test data (outcome information must be present). In this case, the terminal nodes from the grow-forest are recalculated using the y-outcomes from the test set. This yields a modified predictor in which the topology of the forest is based solely on the training data, but where the predicted value is based on the test data. Error rates and VIMP are calculated by bootstrapping the test data and using out-of-bagging to ensure unbiased estimates. See the examples for illustration.

Value

An object of class `(rfsrc, predict)`, which is a list with the following components:

`call` The original grow call to `rfsrc`.

`family` The family used in the analysis.

`n` Sample size of test data (depends upon `NA` values).

`ntree` Number of trees in the grow forest.

`yvar` Test set y-outcomes or original grow y-outcomes if none.

`yvar.names` A character vector of the y-outcome names.

`xvar` Data frame of test set x-variables.

`xvar.names` A character vector of the x-variable names.

`leaf.count` Number of terminal nodes for each tree in the grow forest. Vector of length `ntree`.

`proximity` Symmetric proximity matrix of the test data.

`forest` The grow forest.

`membership` Matrix recording terminal node membership for the test data where each column contains the node number that a case falls in for that tree.

`inbag` Matrix recording inbag membership for the test data where each column contains the number of times that a case appears in the bootstrap sample for that tree.

`var.used` Count of the number of times a variable was used in growing the forest.

`imputed.indv` Vector of indices of records in test data with missing values.

`imputed.data` Data frame comprising imputed test data. The first columns are the y-outcomes followed by the x-variables.

`split.depth` Matrix [i][j] or array [i][j][k] recording the minimal depth for variable [j] for case [i], either averaged over the forest, or by tree [k].

`node.stats` Split statistics returned when `statistics=TRUE` which can be parsed using `stat.split`.

`err.rate` Cumulative OOB error rate for the test data if y-outcomes are present.

`importance` Test set variable importance (VIMP). Can be `NULL`.

`predicted` Test set predicted value.

`predicted.oob` OOB predicted value (`NULL` unless '`outcome="test"`').

`quantile` Quantile value at probabilities requested.

`quantile.oob` OOB quantile value at probabilities requested (`NULL` unless ‘`outcome="test"`’).

`++++++++` for classification settings, additionally ++++++++

`class` In-bag predicted class labels.

`class.oob` OOB predicted class labels (`NULL` unless ‘`outcome="test"`’).

`++++++++` for multivariate settings, additionally ++++++++

`regrOutput` List containing performance values for test multivariate regression responses (applies only in multivariate settings).

`clasOutput` List containing performance values for test multivariate categorical (factor) responses (applies only in multivariate settings).

`++++++++` for survival settings, additionally ++++++++

`chf` Cumulative hazard function (CHF).

`chf.oob` OOB CHF (`NULL` unless ‘`outcome="test"`’).

`survival` Survival function.

`survival.oob` OOB survival function (`NULL` unless ‘`outcome="test"`’).

`time.interest` Ordered unique death times.

`ndead` Number of deaths.

`++++++++` for competing risks, additionally ++++++++

`chf` Cause-specific cumulative hazard function (CSCHF) for each event.

`chf.oob` OOB CSCHF for each event (`NULL` unless ‘`outcome="test"`’).

`cif` Cumulative incidence function (CIF) for each event.

`cif.oob` OOB CIF for each event (`NULL` unless ‘`outcome="test"`’).

`time.interest` Ordered unique event times.

`ndead` Number of events.

Note

The dimensions and values of returned objects depend heavily on the underlying family and whether y-outcomes are present in the test data. In particular, items related to performance will be `NULL` when y-outcomes are not present. For multivariate families, predicted values, VIMP, error rate, and performance values are stored in the lists `regrOutput` and `clasOutput`.

For more detailed information regarding returned values (such as `predicted`) see the `rfsrc` help file.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Breiman L. (2001). Random forests, *Machine Learning*, 45:5-32.

Ishwaran H., Kogalur U.B., Blackstone E.H. and Lauer M.S. (2008). Random survival forests, *Ann. App. Statist.*, 2:841-860.

Ishwaran H. and Kogalur U.B. (2007). Random survival forests for R, *Rnews*, 7(2):25-31.

See Also

`plot.competing.risk`, `plot.rfsrc`, `plot.survival`, `plot.variable`, `rfsrc`, `stat.split`, `vimp`

Examples

```
## ------------------------------------------------------------
## typical train/testing scenario
## ------------------------------------------------------------

data(veteran, package = "randomForestSRC")
train <- sample(1:nrow(veteran), round(nrow(veteran) * 0.80))
veteran.grow <- rfsrc(Surv(time, status) ~ ., veteran[train, ], ntree = 100)
veteran.pred <- predict(veteran.grow, veteran[-train , ])
print(veteran.grow)
print(veteran.pred)


## ------------------------------------------------------------
## predicted probability and predicted class labels are returned
## in the predict object for classification analyses
## ------------------------------------------------------------

data(breast, package = "randomForestSRC")
breast.obj <- rfsrc(status ~ ., data = breast[(1:100), ])
breast.pred <- predict(breast.obj, breast[-(1:100), ])
print(head(breast.pred$predicted))
print(breast.pred$class)

## ------------------------------------------------------------
## example illustrating restore mode
## if predict is called without specifying the test data
## the original training data is used and the forest is restored
## ------------------------------------------------------------

# first we make the grow call
airq.obj <- rfsrc(Ozone ~ ., data = airquality)

# now we restore it and compare it to the original call
# they are identical
predict(airq.obj)
print(airq.obj)

# we can retrieve various outputs that were not asked for in
# in the original call
```

```
#here we extract the proximity matrix
prox <- predict(airq.obj, proximity = TRUE)$proximity
print(prox[1:10,1:10])

#here we extract the number of times a variable was used to grow
#the grow forest
var.used <- predict(airq.obj, var.used = "by.tree")$var.used
print(head(var.used))

## ------------------------------------------------------------
## unique feature of randomForestSRC
## cross-validation can be used when factor labels differ over
## training and test data
## ------------------------------------------------------------

# first we convert all x-variables to factors
data(veteran, package = "randomForestSRC")
veteran.factor <- data.frame(lapply(veteran, factor))
veteran.factor$time <- veteran$time
veteran.factor$status <- veteran$status

# split the data into unbalanced train/test data (5/95)
# the train/test data have the same levels, but different labels
train <- sample(1:nrow(veteran), round(nrow(veteran) * .05))
summary(veteran.factor[train,])
summary(veteran.factor[-train,])

# grow the forest on the training data and predict on the test data
veteran.f.grow <- rfsrc(Surv(time, status) ~ ., veteran.factor[train, ])
veteran.f.pred <- predict(veteran.f.grow, veteran.factor[-train , ])
print(veteran.f.grow)
print(veteran.f.pred)

## ------------------------------------------------------------
## example illustrating the flexibility of outcome = "test"
## illustrates restoration of forest via outcome = "test"
## ------------------------------------------------------------

# first we make the grow call
data(pbc, package = "randomForestSRC")
pbc.grow <- rfsrc(Surv(days, status) ~ ., pbc)

# now use predict with outcome = TEST
pbc.pred <- predict(pbc.grow, pbc, outcome = "test")

# notice that error rates are the same!!
print(pbc.grow)
print(pbc.pred)

# note this is equivalent to restoring the forest
pbc.pred2 <- predict(pbc.grow)
print(pbc.grow)
print(pbc.pred)
print(pbc.pred2)

# similar example, but with na.action = "na.impute"
airq.obj <- rfsrc(Ozone ~ ., data = airquality, na.action = "na.impute")
print(airq.obj)
print(predict(airq.obj))
# ... also equivalent to outcome="test" but na.action = "na.impute" required
print(predict(airq.obj, airquality, outcome = "test", na.action = "na.impute"))

# classification example
iris.obj <- rfsrc(Species ~., data = iris)
print(iris.obj)
print(predict.rfsrc(iris.obj, iris, outcome = "test"))

## ------------------------------------------------------------
## another example illustrating outcome = "test"
## unique way to check reproducibility of the forest
## ------------------------------------------------------------

# primary call
set.seed(542899)
data(pbc, package = "randomForestSRC")
train <- sample(1:nrow(pbc), round(nrow(pbc) * 0.50))
pbc.out <- rfsrc(Surv(days, status) ~ .,  data=pbc[train, ])

# standard predict call
pbc.train <- predict(pbc.out, pbc[-train, ], outcome = "train")
#non-standard predict call: overlays the test data on the grow forest
pbc.test <- predict(pbc.out, pbc[-train, ], outcome = "test")

# check forest reproducibility by comparing "test" predicted survival
# curves to "train" predicted survival curves for the first 3 individuals
Time <- pbc.out$time.interest
matplot(Time, t(exp(-pbc.train$chf)[1:3,]), ylab = "Survival", col = 1, type = "l")
matlines(Time, t(exp(-pbc.test$chf)[1:3,]), col = 2)

## ------------------------------------------------------------
## survival analysis using mixed multivariate outcome analysis
## compare the predicted value to RSF
## ------------------------------------------------------------

# fit the pbc data using RSF
data(pbc, package = "randomForestSRC")
rsf.obj <- rfsrc(Surv(days, status) ~ ., pbc)
yvar <- rsf.obj$yvar

# fit a mixed outcome forest using days and status as y-variables
pbc.mod <- pbc
pbc.mod$status <- factor(pbc.mod$status)
mix.obj <- rfsrc(Multivar(days, status) ~., pbc.mod)

# compare oob predicted values
```

```
rsf.pred <- rsf.obj$predicted.oob
mix.pred <- mix.obj$regrOutput$days$predicted.oob
plot(rsf.pred, mix.pred)

# compare C-index error rate
rsf.err <- randomForestSRC:::cindex(yvar$days, yvar$status, rsf.pred)
mix.err <- 1 - randomForestSRC:::cindex(yvar$days, yvar$status, mix.pred)
cat("RSF                :", rsf.err, "\n")
cat("multivariate forest:", mix.err, "\n")
```

print.rfsrc *Print Summary Output of a RF-SRC Analysis*

Description

Print summary output from a RF-SRC analysis. This is the default print method for the package.

Usage

```
## S3 method for class 'rfsrc'
print(x, outcome.target = NULL, ...)
```

Arguments

`x`	An object of class (`rfsrc, grow`), (`rfsrc, synthetic`), or (`rfsrc, predict`).
`outcome.target`	Character value for multivariate families specifying the target outcome to be used. The default is to use the first coordinate.
`...`	Further arguments passed to or from other methods.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H. and Kogalur U.B. (2007). Random survival forests for R, *Rnews*, 7/2:25-31.

See Also

`rfsrc`, `rfsrcSyn`, `predict.rfsrc`

Examples

```
  iris.obj <- rfsrc(Species ~., data = iris, ntree=100)
  print(iris.obj)
```

`quantileReg` *Quantile Regression Forests*

Description

Grows a univariate or multivariate quantile regression forest and returns its conditional quantile and density values. Can be used for both training and testing purposes.

Usage

```
## S3 method for class 'rfsrc'
quantileReg(formula, data, object, newdata,
  method = "forest", prob = NULL, prob.epsilon = NULL,
  oob = TRUE, fast = FALSE, maxn = 1e3, ...)
```

Arguments

`formula` A symbolic description of the model to be fit. Must be specified unless `object` is given.

`data` Data frame containing the y-outcome and x-variables in the model. Must be specified unless `object` is given.

`object` (Optional) A previously grown quantile regression forest.

`method` Method used to calculate quantiles. Forest weighted averaging is used by default. While this works well for standard data, consider using the Greenwald-Khanna algorithm for big data. The latter is specified by any one of the following: "gk", "GK", "G-K", "g-k".

`prob` Target quantile probabilities when training. If left unspecified, uses percentiles (1 through 99) for `method = "forest"`, and for Greenwald-Khanna selects equally spaced percentiles optimized for accuracy (see below).

`prob.epsilon` Greenwald-Khanna allowable error for quantile probabilities when training.

`newdata` Test data (optional) over which conditional quantiles are evaluated over.

`oob` Return OOB (out-of-bag) quantiles? If false, in-bag values are returned.

`fast` Use fast random forests, `rfsrcFast`, in place of `rfsrc`? Improves speed but may be less accurate.

`maxn` Maximum number of unique y training values used when calculating the conditional density.

`...` Further arguments to be passed to the `rfsrc` function used for fitting the quantile regression forest.

Details

Grows a univariate or multivariate quantile regression forest using quantile regression splitting using the new splitrule `quantile.regr` based on the quantile loss function (often called the "check function").

The default method for calculating quantiles is `method="forest"` which uses forest weights as in Meinshausen (2006). However, because quantile regression splitting is used, and not mean-squared error splitting (as used by Meinshuasen), results may differ substantially from Meinshausen. We believe quantile regression splitting will provide superior performance.

While calculating quantiles using forest weights works well for standard data, a second approach, the Greenwald-Khanna (2001) algorithm, will be more appropriate for big data due to its high memory efficiency.

The Greenwald-Khanna algorithm is implemented roughly as follows. To form a distribution of values for each case, from which we sample to determine quantiles, we create a chain of values for the case as we grow the forest. Every time a case lands in a terminal node, we insert all of its co-inhabitants to its chain of values.

The best case scenario is when tree node size is 1 because each case gets only one insert into its chain for that tree. The worst case scenario is when node size is so large that trees stump. This is because each case receives insertions for the entire in-bag population.

What the user needs to know is that Greenwald-Khanna can become slow in counter-intutive settings such as when node size is large. The easy fix is to change the epsilon quantile approximation that is requested. You will see a significant speed-up just by doubling `prob.epsilon`. This is because the chains stay a lot smaller as epsilon increases, which is exactly what you want when node sizes are large. Both time and space requirements for the algorithm are affected by epsilon.

The best results for Greenwald-Khanna come from setting the number of quantiles equal to 2 times the sample size and epsilon to 1 over 2 times the sample size which is the default values used if left unspecified. This will be slow, especially for big data, and less stringent choices should be used if computational speed is of concern.

Value

Returns quantiles for each of the requested probabilities. Also returns the conditional density (and conditional cdf) for unique y-values in the training data (or test data if provided). The conditional density can be used to calculate conditional moments, such as the mean and standard deviation. For convenience, the mean is returned as the object `yhat`.

For multivariate forests, the returned object will be a list of length equal to the number of target outcomes.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Greenwald M. and Khanna S. (2001). Space-efficient online computation of quantile summaries. *Proceedings of ACM SIGMOD*, 30(2):58–66.

Meinshausen N. (2006) Quantile regression forests, *Journal of Machine Learning Research*, 7:983–999.

See Also

rfsrc

Examples

```
## ------------------------------------------------------------
## regression example
## ------------------------------------------------------------

## standard call
o <- quantileReg(mpg ~ ., mtcars)
qo <- o$quantileReg

## calculate the conditional mean, compare to OOB predicted value
## note that the conditional mean is returned as "yhat"
c.mean <- qo$density %*% qo$yunq
print(data.frame(c.mean = c.mean, yhat = qo$yhat, pred.oob = o$predicted.oob))

## calculate conditional standard deviation
c.std <- sqrt(qo$density %*% qo$yunq^2 - c.mean ^ 2)
quant <- qo$quantile
colnames(quant) <- paste("q", 100 * qo$prob, sep = "")
print(data.frame(quant, c.std))


## ------------------------------------------------------------
## train/test regression example
## ------------------------------------------------------------

## train (grow) call followed by test call
trn <- quantileReg(mpg ~ ., mtcars[1:20,])
test <- quantileReg(object = trn, newdata = mtcars[-(1:20),-1])

## calculate test set conditional mean and standard deviation
qo <- test$quantileReg
c.mean <- qo$density %*% qo$yunq
c.std <- sqrt(qo$density %*% qo$yunq^2 - c.mean ^ 2)
quant <- qo$quant
colnames(quant) <- paste("q", 100 * qo$prob, sep = "")
print(data.frame(quant, c.mean, c.std))


## ------------------------------------------------------------
## multivariate mixed outcomes example
## ------------------------------------------------------------

dta <- mtcars
dta$cyl <- factor(dta$cyl)
dta$carb <- factor(dta$carb, ordered = TRUE)
o <- quantileReg(cbind(carb, mpg, cyl, disp) ~., data = dta)
```

```
print(head(o$quantileReg$mpg$quant))
print(head(o$quantileReg$disp$quant))


## ------------------------------------------------------------
## quantile regression plot for Boston Housing data
## ------------------------------------------------------------

if (library("mlbench", logical.return = TRUE)) {

  ## apply quantile regression to Boston Housing data
  data(BostonHousing)
  o <- quantileReg(medv ~ ., BostonHousing, nodesize = 1)
  y <- o$yvar
  qo <- o$quantileReg

  ## pull desired quantiles - nice little wrapper for doing this
  get.quantile <- function(q, target.prob) {
    target.prob <- sort(unique(target.prob))
    q.dta <- do.call(cbind, lapply(target.prob, function(pr) {
      q$quant[, which.min(abs(pr - q$prob))]
    }))
    colnames(q.dta) <-  paste("q.", 100 * target.prob, sep = "")
    q.dta
   }

  ## extract 25,50,75 quantiles
  quant.dat <- get.quantile(qo, c(.25, .50, .75))

  ## quantile regression plot
  plot(range(y), range(quant.dat), xlab = "y",
       ylab = ".25-.75 Quantiles", type = "n")
  jitter.y <- jitter(y, 10)
  points(jitter.y, quant.dat[, 2], pch = 15, col = 4, cex = 0.75)
  segments(jitter.y, quant.dat[, 2], jitter.y, quant.dat[, 1], col = "grey")
  segments(jitter.y, quant.dat[, 2], jitter.y, quant.dat[, 3], col = "grey")
  points(jitter.y, quant.dat[, 1], pch = "-", cex = 1)
  points(jitter.y, quant.dat[, 3], pch = "-", cex = 1)
  abline(0, 1, lty = 2, col = 2)

  ## compare 25-75 percentiles to values expected under normality
  c.mean <- qo$density %*% qo$yunq
  c.std <- sqrt(qo$density %*% qo$yunq^2 - c.mean ^ 2)
  q.25.est <- c.mean + qnorm(.25) * c.std
  q.75.est <- c.mean + qnorm(.75) * c.std
  print(head(data.frame(quant.dat[, -2],  q.25.est, q.75.est)))

  ## compare performance of quantile regression estimator to
  ## standard random forest estimator of averaged tree mean
  cat("quantile regression yhat error:", mean((o$yvar-qo$yhat)^2), "\n")
  cat("RF averaged tree mean error:", mean((o$yvar-o$predicted.oob)^2), "\n")
}

## ------------------------------------------------------------
## example of quantile regression for ordinal data
## ------------------------------------------------------------

 ## use the wine data for illustration
 data(wine, package = "randomForestSRC")

 ## run quantile regression
 o <- quantileReg(quality ~ ., wine, ntree = 100)

 ## extract "probabilities" = density values
 qo <- o$quantileReg
 yunq <- qo$yunq
 yvar <- factor(cut(o$yvar, c(-1, yunq), labels = yunq))
 qo.dens <- qo$density
 colnames(qo.dens) <- yunq
 qo.class <- randomForestSRC:::bayes.rule(qo.dens)
 qo.confusion <- table(yvar, qo.class)
 qo.err <- 1 - diag(qo.confusion) / rowSums(qo.confusion)
 qo.confusion <- cbind(qo.confusion, qo.err)
 print(qo.confusion)
 cat("Normalized Brier:", 100 * randomForestSRC:::brier(yvar, qo.dens), "\n")
```

rfsrc *Random Forests for Survival, Regression, and Classification (RF-SRC)*

Description

Fast OpenMP parallel processing unified treatment of Breiman's random forests (Breiman 2001) for a variety of data settings. Applies when the y-response is numeric or categorical, yielding Breiman regression and classification forests, while random survival forests (Ishwaran et al. 2008, 2012) are grown for right-censored survival and competing risk data. Multivariate regression and classification responses as well as mixed regression/classification responses are also handled. Also includes unsupervised forests and quantile regression forests, `quantileReg`. Different splitting rules invoked under deterministic or random splitting are available for all families. Variable predictiveness can be assessed using variable importance (VIMP) measures for single, as well as grouped variables. Missing data can be imputed on both training and test data; see `impute`. The forest object, informally referred to as an RF-SRC object, contains many useful values which can be directly extracted by the user and/or parsed using additional functions (see the examples below).

This is the main entry point to the **randomForestSRC** package. Also see `rfsrcFast` for a fast implementation of `rfsrc`.

For more information about this package and OpenMP, use the command `package?randomForestSRC`.

Usage

```
rfsrc(formula, data, ntree = 1000,
  mtry = NULL, ytry = NULL,
  nodesize = NULL, nodedepth = NULL,
  splitrule = NULL, nsplit = 10,
  importance = c(FALSE, TRUE, "none", "permute", "random", "anti"),
  block.size = if (importance == "none" || as.character(importance) == "FALSE") NULL
    else 10,
  ensemble = c("all", "oob", "inbag"),
  bootstrap = c("by.root", "by.node", "none", "by.user"),
  samptype = c("swr", "swor"), sampsize = NULL, samp = NULL, membership = FALSE,
  na.action = c("na.omit", "na.impute"), nimpute = 1,
  ntime, cause,
  proximity = FALSE, distance = FALSE, forest.wt = FALSE,
  xvar.wt = NULL, yvar.wt = NULL, split.wt = NULL, case.wt  = NULL,
  forest = TRUE,
  var.used = c(FALSE, "all.trees", "by.tree"),
  split.depth = c(FALSE, "all.trees", "by.tree"),
  seed = NULL,
  do.trace = FALSE,
  statistics = FALSE,
  ...)
```

Arguments

`formula` A symbolic description of the model to be fit. If missing, unsupervised splitting is implemented.

`data` Data frame containing the y-outcome and x-variables.

`ntree` Number of trees in the forest.

`mtry` Number of variables randomly selected as candidates for splitting a node. The default is p/3 for regression, where p equals the number of variables. For all other families (including unsupervised settings), the default is sqrt(p). Values are always rounded up.

`ytry` For unsupervised forests, sets the number of randomly selected pseudo-responses (see below for more details). The default is `ytry`=1, which selects one pseudo-response.

`nodesize` Forest average number of unique cases (data points) in a terminal node. The defaults are: survival (15), competing risk (15), regression (5), classification (1), mixed outcomes (3), unsupervised (3). It is recommended to experiment with different `nodesize` values.

`nodedepth` Maximum depth to which a tree should be grown. The default behaviour is that this parameter is ignored.

`splitrule` Splitting rule used to grow trees. See below for details.

`nsplit` Non-negative integer value. When zero or NULL, deterministic splitting for an x-variable is in effect. When non-zero, a maximum of nsplit split points are randomly chosen among the possible split points for the x-variable. This significantly increases speed.

`importance` Method for computing variable importance (VIMP). Because VIMP is computationally expensive, the default action is `importance="none"` (VIMP can always be recovered later using the functions `vimp` or `predict`). Setting `importance=TRUE` implements permutation VIMP. See below for more details.

`block.size` Should the cumulative error rate be calculated on every tree? When `NULL`, it will only be calculated on the last tree and the plot of the cumulative error rate will result in a flat line. To view the cumulative error rate on every nth tree, set the value to an integer between `1` and `ntree`. As an intended side effect, if importance is requested, VIMP is calculated in "blocks" of size equal to `block.size`, thus resulting in a useful compromise between ensemble and permutation VIMP. The default action is to use 10 trees. See VIMP below for more details.

`ensemble` Specifies the type of ensemble. By default both out-of-bag (OOB) and inbag ensembles are returned. Always use OOB values for interfence on the training data.

`bootstrap` Bootstrap protocol. The default is `by.root` which bootstraps the data by sampling with replacement at the root node before growing the tree (for sampling without replacement, see the option `samptype`). If `by.node` is choosen, the data is bootstrapped at each node during the grow process. If `none` is chosen, the data is not bootstrapped at all. If `by.user` is choosen, the bootstrap specified by `samp` is used. It is not possible to return OOB ensembles or prediction error if `by.node` or `none` are in effect.

`samptype` Type of bootstrap when `by.root` is in effect. Choices are `swr` (sampling with replacement, the default action) and `swor` (sampling without replacement).

`sampsize` Requested size of bootstrap when `by.root` is in effect (if missing the default action is the usual bootstrap).

`samp` Bootstrap specification when `by.user` is in effect. This is a array of dim `n x ntree` specifying how many times each record appears inbag in the bootstrap for each tree.

`membership` Should terminal node membership and inbag information be returned?

`na.action` Action taken if the data contains `NA`'s. Possible values are `na.omit` or `na.impute`. The default `na.omit` removes the entire record if even one of its entries is `NA` (for x-variables this applies only to those specifically listed in 'formula'). Selecting `na.impute` imputes the data. See below for more details regarding missing data imputation.

`nimpute` Number of iterations of the missing data algorithm. Performance measures such as out-of-bag (OOB) error rates tend to become optimistic if `nimpute` is greater than 1.

`ntime` Integer value used for survival to constrain ensemble calculations to a grid of `ntime` time points. Alternatively if a vector of values of length greater than one is supplied, it is assumed these are the time points to be used to constrain the calculations (note that the constrained time points used will be the observed event times closest to the user supplied time points). If no value is specified, the default action is to use all observed event times.

`cause` Integer value between 1 and `J` indicating the event of interest for competing risks, where `J` is the number of event types (this option applies only to competing risks and is ignored otherwise). While growing a tree, the splitting of a node is restricted to the event type specified by `cause`. If not specified, the default is to use a composite splitting rule which is an average over the entire set of event types (a democratic approach). Users can also pass a vector of non-negative weights of length `J` if they wish to use a customized composite split statistic (for example, passing a vector of ones reverts to the default composite split statistic). In all instances when `cause` is set incorrectly, splitting reverts to the default. Finally, note that regardless of how `cause` is specified, the returned forest object always provides estimates for all event types.

`proximity` Proximity of cases as measured by the frequency of sharing the same terminal node. This is an `nxn` matrix, which can be large. Choices are `inbag`, `oob`, `all`, `TRUE`, or `FALSE`. Setting `proximity = TRUE` is equivalent to `proximity = "inbag"`.

`distance` Distance between cases as measured by the ratio of the sum of the count of edges from each case to their immediate common ancestor node to the sum of the count of edges from each case to the root node. If the cases are co-terminal for a tree, this measure is zero and reduces to 1 - the proximity measure for these cases in a tree. This is an `nxn` matrix, which can be large. Choices are `inbag`, `oob`, `all`, `TRUE`, or `FALSE`. Setting `distance = TRUE` is equivalent to `distance = "inbag"`.

`forest.wt` Should the forest weight matrix be calculated? Creates an `nxn` matrix which can be used for prediction and constructing customized estimators. Choices are similar to proximity: `inbag`, `oob`, `all`, `TRUE`, or `FALSE`. The default is `TRUE` which is equivalent to `inbag`.

`xvar.wt` Vector of non-negative weights where entry `k`, after normalizing, is the probability of selecting variable `k` as a candidate for splitting a node. Default is to use uniform weights. Vector must be of dimension `p`, where `p` equals the number of variables, otherwise the default is invoked. It is generally better to use real weights rather than integers. With larger sizes of `p`, the slightly different sampling algorithms used in the two scenarios can result in dramatically different execution times.

`yvar.wt` NOT YET IMPLEMENTED: Vector of non-negative weights where entry `k`, after normalizing, is the probability of selecting response `k` as a candidate for inclusion in the split statistic in unsupervised settings. Default is to use uniform weights. Vector must be of the same length as the number of respones in the data set.

`split.wt` Vector of non-negative weights where entry `k`, after normalizing, is the multiplier by which the split statistic for a variable is adjusted. A large value encourages the node to split on the variable. Default is to use uniform weights. Vector must be of dimension `p`, where `p` equals the number of variables, otherwise the default is invoked.

`case.wt` Vector of non-negative weights where entry `k`, after normalizing, is the probability of selecting case `k` as a candidate for the bootstrap. Default is to use uniform weights. Vector must be of dimension `n`, where `n` equals the number of cases in the processed data set (missing values may be removed, thus altering the original sample size). It is generally better to use real weights rather than integers. With larger sizes of `n`, the slightly different sampling algorithms used in the two scenarios can result in dramatically different execution times. See the example below for the breast data set for an illustration of its use for class imbalanced data.

`forest` Should the forest object be returned? Used for prediction on new data and required by many of the functions used to parse the RF-SRC object. It is recommended not to change the default setting.

`var.used` Return variables used for splitting? Default is `FALSE`. Possible values are `all.trees` which returns a vector where each element records the number of times a split occurred on a variable, and `by.tree` which is a matrix recording the number of times a split occurred on a variable in a specific tree.

`split.depth` Records the minimal depth for each variable. Default is `FALSE`. Possible values are `all.trees` which returns a matrix recording the minimal depth for a variable (columns) for a specific case (rows) averaged over the forest, and `by.tree` which returns a three-dimensional array recording minimal depth for a specific case (first dimension) for a variable (second dimension) for a specific tree (third dimension).

`seed` Negative integer specifying seed for the random number generator.

`do.trace` Number of seconds between updates to the user on approximate time to completion.

`statistics` Should split statistics be returned? Values can be parsed using `stat.split`.

`...` Further arguments passed to or from other methods.

Details

1. *Families*

   Do *not* set this value as the package automagically determines the underlying random forest family from the type of response and the formula supplied. There are eight possible scenarios:

   (a) Regression forests for continuous responses.
   (b) Multivariate regression forests for multivariate continuous responses.
   (c) Classification forests for factor responses.
   (d) Multivariate classification forests for multivariate factor responses.
   (e) Multivariate mixed forests for mixed continuous and factor responses.
   (f) Unsupervised forests when there is no response.
   (g) Survival forest for right-censored survival settings.
   (h) Competing risk survival forests for competing risk scenarios.

   See below for how to code the response in the two different survival scenarios and for specifying a multivariate forest formula.

2. *Splitrules*

   Splitrules are set according to the option `splitrule` as follows:

   - Regression analysis:
     (a) The default rule is weighted mean-squared error splitting `mse` (Breiman et al. 1984, Chapter 8.4).
     (b) Unweighted and heavy weighted mean-squared error splitting rules can be invoked using splitrules `mse.unwt` and `mse.hvwt`. Generally `mse` works best, but see Ishwaran (2015) for details.

(c) Quantile regression splitting `quantile.regr` using the "check-loss" function. Requires specifying the target quantiles. See `quantileReg` for further details.

- Multivariate regression analysis: For multivariate regression responses, a composite normalized mean-squared error splitting rule is used.
- Classification analysis:
  (a) The default rule is Gini index splitting `gini` (Breiman et al. 1984, Chapter 4.3).
  (b) Unweighted and heavy weighted Gini index splitting rules can be invoked using splitrules `gini.unwt` and `gini.hvwt`. Generally `gini` works best, but see Ishwaran (2015) for details.
- Multivariate classification analysis: For multivariate classification responses, a composite normalized Gini index splitting rule is used.
- Mixed outcomes analysis: When both regression and classification responses are detected, a multivariate normalized composite split rule of mean-squared error and Gini index splitting is invoked. See Tang and Ishwaran (2017) for details.
- Unsupervised analysis: In settings where there is no outcome, unsupervised splitting is invoked. In this case, the mixed outcome splitting rule (above) is applied. See Mantero and Ishwaran (2017) for details.
- Survival analysis:
  (a) The default rule is `logrank` which implements log-rank splitting (Segal, 1988; Leblanc and Crowley, 1993).
  (b) `logrankscore` implements log-rank score splitting (Hothorn and Lausen, 2003).
- Competing risk analysis:
  (a) The default rule is `logrankCR` which implements a modified weighted log-rank splitting rule modeled after Gray's test (Gray, 1988).
  (b) `logrank` implements weighted log-rank splitting where each event type is treated as the event of interest and all other events are treated as censored. The split rule is the weighted value of each of log-rank statistics, standardized by the variance. For more details see Ishwaran et al. (2014).
- Custom splitting: All families except unsupervised are available for user defined custom splitting. Some basic C-programming skills are required. The harness for defining these rules is in `splitCustom.c`. In this file we give examples of how to code rules for regression, classification, survival, and competing risk. Each family can support up to sixteen custom split rules. Specifying `splitrule="custom"` or `splitrule="custom1"` will trigger the first split rule for the family defined by the training data set. Multivariate families will need a custom split rule for both regression and classification. In the examples, we demonstrate how the user is presented with the node specific membership. The task is then to define a split statistic based on that membership. Take note of the instructions in `splitCustom.c` on how to *register* the custom split rules. It is suggested that the existing custom split rules be kept in place for reference and that the user proceed to develop `splitrule="custom2"` and so on. The package must be recompiled and installed for the custom split rules to become available.
- Random splitting. For all families, pure random splitting can be invoked by setting `splitrule="random"`. See below for more details regarding randomized splitting rules.

3. *Allowable data types*

Data types must be real valued, integer, factor or logical – however all except factors are coerced and treated as if real valued. For ordered x-variable factors, splits are similar to real valued variables. If the x-variable factor is unordered, a split will move a subset of the levels in the parent node to the left daughter, and the complementary subset to the right daughter. All possible complementary pairs are considered and apply to factors with an unlimited number of levels. However, there is an optimization check to ensure that the number of splits attempted is not greater than the number of cases in a node (this internal check will override the `nsplit` value in random splitting mode if `nsplit` is large enough; see below for information about `nsplit`).

4. *Improving computational speed*

   See the function `rfsrcFast` for a fast implementation of `rfsrc`. In general, the key methods for increasing speed are as follows:

   - *Randomized splitting rules*
     Trees tend to favor splits on continuous variables and factors with large numbers of levels (Loh and Shih, 1997). To mitigate this bias and improve speed, randomized splitting can be invoked using the option `nsplit`. If `nsplit` is set to a non-zero positive integer, then a maximum of `nsplit` split points are chosen randomly for each of the `mtry` variables within a node and only these points are used to determine the best split. Pure random splitting can be invoked by setting `splitrule="random"`. In this case, a variable is randomly selected and the node is split using a random split point (Cutler and Zhao, 2001; Lin and Jeon, 2006). Note when pure random splitting is in effect, `nsplit` is set to one.
   - *Subsampling*
     Subsampling can be used to reduce the size of the in-sample data used to grow a tree and therefore can greatly reduce computational load. Subsampling is implemented using options `sampsize` and `samptype`.
   - *Unique time points*
     For large survival problems, users should consider setting `ntime` to a reasonably small value (such as 50 or 100). This constrains ensemble calculations such as survival functions to a restricted grid of time points of length no more than `ntime` and considerably reduces computational times.
   - *Large number of variables*
     Use the default setting of `importance="none"` which turns off variable importance (VIMP) calculations and considerably reduces computational times when there are a large number of variables (see below for more details about variable importance). Variable importance calculations can always be recovered later using functions `vimp` or `predict`. Also consider using the function `max.subtree` which calculates minimal depth, a measure of the depth that a variable splits, and yields fast variable selection (Ishwaran, 2010).
   - *Factors*
     For coherence, an immutable map is applied to each factor that ensures that factor levels in the training data set are consistent with the factor levels in any subsequent test data set. This map is applied to each factor before and after the native C library is executed. Because of this, if x-variables are all factors, then computational times may become very long in high dimensional problems. Consider converting factors to real if this is the case.

5. *Prediction Error*

   Prediction error is calculated using OOB data. Performance is measured in terms of mean-squared-error for regression, and misclassification error for classification. A normalized Brier score (relative to a coin-toss) is also provided upon printing a classification forest.

   For survival, prediction error is measured by 1-C, where C is Harrell's (Harrell et al., 1982) concordance index. Prediction error is between 0 and 1, and measures how well the predictor correctly ranks (classifies) two random individuals in terms of survival. A value of 0.5 is no better than random guessing. A value of 0 is perfect.

When bootstrapping is `by.node` or `none`, a coherent OOB subset is not available to assess prediction error. Thus, all outputs dependent on this are suppressed. In such cases, prediction error is only available via classical cross-validation (the user will need to use the `predict.rfsrc` function).

6. *Variable Importance (VIMP)*

   To calculate VIMP, use the option `importance`. Classical permutation VIMP is implemented when `permute` or `TRUE` is selected. In this case, OOB cases for a variable *x* are randomly permuted and dropped down a tree. VIMP is calculated by comparing OOB prediction performance for the permuted predictor to the original predictor.

   The exact calculation for VIMP depends upon `block.size` (an integer value between `1` and `ntree`) specifying the number of trees in a block used to determine VIMP. When the value is `1`, VIMP is calculated by tree (blocks of size 1). Specifically, the difference between prediction error under the perturbed predictor and the original predictor is calculated for each tree and averaged over the forest. This yields the original Breiman-Cutler VIMP (Breiman 2001).

   When `block.size` is set to `ntree`, VIMP is calculated by comparing the error rate for the perturbed OOB forest ensemble (using all trees) to the unperturbed OOB forest ensemble (using all trees). Thus, unlike Breiman-Cutler VIMP, ensemble VIMP does not measure the tree average effect of *x*, but rather its overall forest effect. This is called Ishwaran-Kogalur VIMP (Ishwaran et al. 2008).

   A useful compromise between Breiman-Cutler (BC) and Ishwaran-Kogalur (IK) VIMP can be obtained by setting `block.size` to a value between `1` and `ntree`. Smaller values are closer to BC and larger values closer to IK. Smaller generally gives better accuracy, however computational times will be higher because VIMP is calculated over more blocks.

   The option `importance` permits different ways to perturb a variable. If `random` is specified, then instead of permuting *x*, OOB case are assigned a daughter node randomly whenever a split on *x* is encountered. If `anti` is specified, *x* is assigned to the opposite node whenever a split on *x* is encountered.

   Note that the option `none` turns VIMP off entirely.

   Note that the function `vimp` provides a friendly user interface for extracting VIMP.

7. *Multivariate Forests*

   Multivariate forests are specified by using the multivariate formula interface. Such a call takes one of two forms:

   rfsrc(Multivar(y1, y2, ..., yd) ~ . , my.data, ...)

   rfsrc(cbind(y1, y2, ..., yd) ~ . , my.data, ...)

   A multivariate normalized composite splitting rule is used to split nodes. The nature of the outcomes will inform the code as to the type of multivariate forest to be grown; i.e. whether it is real-valued, categorical, or a combination of both (mixed). Note that performance measures (when requested) are returned for all outcomes.

8. *Unsupervised Forests*

   In the case where no y-outcomes are present, unsupervised forests can be invoked by one of two means:

   rfsrc(data = my.data)

   rfsrc(Unsupervised() ~ ., data = my.data)

To split a tree node, a random subset of `ytry` variables are selected from the available features, and these variables function as "pseudo-responses" to be split on. Thus, in unsupervised mode, the features take turns acting as both target y-outcomes and x-variables for splitting.

More precisely, as in supervised forests, `mtry` x-variables are randomly selected from the set of `p` features for splitting the node. Then on each `mtry` loop, `ytry` variables are selected from the `p`-1 remaining features to act as the target pseduo-responses to be split on (there are `p`-1 possibilities because we exclude the currently selected x-variable for the current `mtry` loop — also, only pseudo-responses that pass purity checks are used). The split-statistic for splitting the node on the pseudo-responses using the x-variable is calculated. The best split over the `mtry` pairs is used to split the node.

The default value of `ytry` is 1 but can be increased by the `ytry` option. A value larger than 1 initiates multivariate splitting. As illustration, consider the call:

rfsrc(data = my.data, ytry = 5, mtry = 10)

This is equivalent to the call:

rfsrc(Unsupervised(5) ~ ., my.data, mtry = 10)

In the above, a node will be split by selecting `mtry=10` x-variables, and for each of these a random subset of 5 features will be selected as the multivariate pseudo-responses. The split-statistic is a multivariate normalized composite splitting rule which is applied to each of the 10 multivariate regression problems. The node is split on the variable leading to the best split.

Note that all performance values (error rates, VIMP, prediction) are turned off in unsupervised mode.

9. *Survival, Competing Risks*

   (a) Survival settings require a time and censoring variable which should be identifed in the formula as the response using the standard `Surv` formula specification. A typical formula call looks like:
   Surv(my.time, my.status) ~ .
   where `my.time` and `my.status` are the variables names for the event time and status variable in the users data set.

   (b) For survival forests (Ishwaran et al. 2008), the censoring variable must be coded as a non-negative integer with 0 reserved for censoring and (usually) 1=death (event). The event time must be non-negative.

   (c) For competing risk forests (Ishwaran et al., 2013), the implementation is similar to survival, but with the following caveats:

   - Censoring must be coded as a non-negative integer, where 0 indicates right-censoring, and non-zero values indicate different event types. While 0,1,2,..,J is standard, and recommended, events can be coded non-sequentially, although 0 must always be used for censoring.
   - Setting the splitting rule to `logrankscore` will result in a survival analysis in which all events are treated as if they are the same type (indeed, they will coerced as such).
   - Generally, competing risks requires a larger `nodesize` than survival settings.

10. *Missing data imputation*

    Setting `na.action="na.impute"` imputes missing data (both x and y-variables) using a modification of the missing data algorithm of Ishwaran et al. (2008). See also Tang and Ishwaran (2017). Split statistics are calculated using non-misssing data only. If a node splits on a variable with missing data, the variable's missing data is imputed by randomly drawing values from non-missing in-bag data. The purpose of this is to make it possible to assign cases to daughter nodes based on the split. Following a node split, imputed data are reset to missing and the process is repeated until terminal nodes are reached. Missing data in terminal nodes are imputed using in-bag non-missing terminal node data. For integer valued variables and censoring indicators, imputation uses a maximal class rule, whereas continuous variables and survival time use a mean rule.

The missing data algorithm can be iterated by setting `nimpute` to a positive integer greater than 1. Using only a few iterations are needed to improve accuracy. When the algorithm is iterated, at the completion of each iteration, missing data is imputed using OOB non-missing terminal node data which is then used as input to grow a new forest. Note that when the algorithm is iterated, a side effect is that missing values in the returned objects `xvar`, `yvar` are replaced by imputed values. Further, imputed objects such as `imputed.data` are set to `NULL`. Also, keep in mind that if the algorithm is iterated, performance measures such as error rates and VIMP become optimistically biased.

Finally, records in which all outcome and x-variable information are missing are removed from the forest analysis. Variables having all missing values are also removed.

See the function `impute` for a fast impute interface.

Value

An object of class `(rfsrc, grow)` with the following components:

`call` The original call to `rfsrc`.

`family` The family used in the analysis.

`n` Sample size of the data (depends upon `NA`'s, see `na.action`).

`ntree` Number of trees grown.

`mtry` Number of variables randomly selected for splitting at each node.

`nodesize` Minimum size of terminal nodes.

`nodedepth` Maximum depth allowed for a tree.

`splitrule` Splitting rule used.

`nsplit` Number of randomly selected split points.

`yvar` y-outcome values.

`yvar.names` A character vector of the y-outcome names.

`xvar` Data frame of x-variables.

`xvar.names` A character vector of the x-variable names.

`xvar.wt` Vector of non-negative weights specifying the probability used to select a variable for splitting a node.

`split.wt` Vector of non-negative weights where entry `k`, after normalizing, is the multiplier by which the split statistic for a covariate is adjusted.

`cause.wt` Vector of weights used for the composite competing risk splitting rule.

`leaf.count` Number of terminal nodes for each tree in the forest. Vector of length `ntree`. A value of zero indicates a rejected tree (can occur when imputing missing data). Values of one indicate tree stumps.

`proximity` Proximity matrix recording the frequency of pairs of data points occur within the same terminal node.

`forest` If `forest=TRUE`, the forest object is returned. This object is used for prediction with new test data sets and is required for other R-wrappers.

`forest.wt` Forest weight matrix.

`membership` Matrix recording terminal node membership where each column contains the node number that a case falls in for that tree.

`splitrule` Splitting rule used.

`inbag` Matrix recording inbag membership where each column contains the number of times that a case appears in the bootstrap sample for that tree.

`var.used` Count of the number of times a variable is used in growing the forest.

`imputed.indv` Vector of indices for cases with missing values.

`imputed.data` Data frame of the imputed data. The first column(s) are reserved for the y-responses, after which the x-variables are listed.

`split.depth` Matrix [i][j] or array [i][j][k] recording the minimal depth for variable [j] for case [i], either averaged over the forest, or by tree [k].

`node.stats` Split statistics returned when `statistics=TRUE` which can be parsed using `stat.split`.

`err.rate` Tree cumulative OOB error rate.

`importance` Variable importance (VIMP) for each x-variable.

`predicted` In-bag predicted value.

`predicted.oob` OOB predicted value.

`++++++++` for classification settings, additionally ++++++++

`class` In-bag predicted class labels.

`class.oob` OOB predicted class labels.

`++++++++` for multivariate settings, additionally ++++++++

`regrOutput` List containing performance values for multivariate regression responses (applies only in multivariate settings).

`clasOutput` List containing performance values for multivariate categorical (factor) responses (applies only in multivariate settings).

`++++++++` for survival settings, additionally ++++++++

`survival` In-bag survival function.

`survival.oob` OOB survival function.

`chf` In-bag cumulative hazard function (CHF).

`chf.oob` OOB CHF.

`time.interest` Ordered unique death times.

`ndead` Number of deaths.

`++++++++` for competing risks, additionally ++++++++

`chf` In-bag cause-specific cumulative hazard function (CSCHF) for each event.

`chf.oob` OOB CSCHF.

`cif` In-bag cumulative incidence function (CIF) for each event.

`cif.oob` OOB CIF.

`time.interest` Ordered unique event times.

`ndead` Number of events.

Note

Values returned depend heavily on the family. In particular, `predicted` and `predicted.oob` are the following values calculated using in-bag and OOB data:

1. For regression, a vector of predicted y-responses.
2. For classification, a matrix with columns containing the estimated class probability for each class. Performance values and VIMP for classification are reported as a matrix with J+1 columns where J is the number of classes. The first column "all" is the unconditional value for performance or VIMP, while the remaining columns are performance and VIMP conditioned on cases corresponding to that class label.
3. For survival, a vector of mortality values (Ishwaran et al., 2008) representing estimated risk for each individual calibrated to the scale of the number of events (as a specific example, if *i* has a mortality value of 100, then if all individuals had the same x-values as *i*, we would expect an average of 100 events). Also returned are matrices containing the CHF and survival function. Each row corresponds to an individual's ensemble CHF or survival function evaluated at each time point in `time.interest`.
4. For competing risks, a matrix with one column for each event recording the expected number of life years lost due to the event specific cause up to the maximum follow up (Ishwaran et al., 2013). Also returned are the cause-specific cumulative hazard function (CSCHF) and the cumulative incidence function (CIF) for each event type. These are encoded as a three-dimensional array, with the third dimension used for the event type, each time point in `time.interest` making up the second dimension (columns), and the case (individual) being the first dimension (rows).
5. For multivariate families, predicted values (and other performance values such as VIMP and error rates) are stored in the lists `regrOutput` and `clasOutput` which can be parsed using the functions `get.mv.error`, `get.mv.predicted` and `get.mv.vimp`.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Breiman L., Friedman J.H., Olshen R.A. and Stone C.J. *Classification and Regression Trees*, Belmont, California, 1984.

Breiman L. (2001). Random forests, *Machine Learning*, 45:5-32.

Cutler A. and Zhao G. (2001). Pert-Perfect random tree ensembles. *Comp. Sci. Statist.*, 33: 490-497.

Gray R.J. (1988). A class of k-sample tests for comparing the cumulative incidence of a competing risk, *Ann. Statist.*, 16: 1141-1154.

Harrell et al. F.E. (1982). Evaluating the yield of medical tests, *J. Amer. Med. Assoc.*, 247:2543-2546.

Hothorn T. and Lausen B. (2003). On the exact distribution of maximally selected rank statistics, *Comp. Statist. Data Anal.*, 43:121-137.

Ishwaran H. (2007). Variable importance in binary regression trees and forests, *Electronic J. Statist.*, 1:519-537.

Ishwaran H. and Kogalur U.B. (2007). Random survival forests for R, *Rnews*, 7(2):25-31.

Ishwaran H., Kogalur U.B., Blackstone E.H. and Lauer M.S. (2008). Random survival forests, *Ann. App. Statist.*, 2:841-860.

Ishwaran H., Kogalur U.B., Gorodeski E.Z, Minn A.J. and Lauer M.S. (2010). High-dimensional variable selection for survival data. *J. Amer. Statist. Assoc.*, 105:205-217.

Ishwaran H., Kogalur U.B., Chen X. and Minn A.J. (2011). Random survival forests for high-dimensional data. *Stat. Anal. Data Mining*, 4:115-132

Ishwaran H., Gerds T.A., Kogalur U.B., Moore R.D., Gange S.J. and Lau B.M. (2014). Random survival forests for competing risks. *Biostatistics*, 15(4):757-773.

Ishwaran H. and Malley J.D. (2014). Synthetic learning machines. *BioData Mining*, 7:28.

Ishwaran H. (2015). The effect of splitting on random forests. *Machine Learning*, 99:75-118.

Ishwaran H. and Lu M. (2018). Standard errors and confidence intervals for variable importance in random forest regression, classification, and survival. Statistics in Medicine (in press).

Lin Y. and Jeon Y. (2006). Random forests and adaptive nearest neighbors, *J. Amer. Statist. Assoc.*, 101:578-590.

LeBlanc M. and Crowley J. (1993). Survival trees by goodness of split, *J. Amer. Statist. Assoc.*, 88:457-467.

Loh W.-Y and Shih Y.-S (1997). Split selection methods for classification trees, *Statist. Sinica*, 7:815-840.

Mantero A. and Ishwaran H. (2017). Unsupervised random forests.

Mogensen, U.B, Ishwaran H. and Gerds T.A. (2012). Evaluating random forests for survival analysis using prediction error curves, *J. Statist. Software*, 50(11): 1-23.

O'Brien R. and Ishwaran H. (2017). A random forests quantile classifier for class imbalanced data.

Segal M.R. (1988). Regression trees for censored data, *Biometrics*, 44:35-47.

Tang F. and Ishwaran H. (2017). Random forest missing data algorithms. *Statistical Analysis and Data Mining*, 10, 363-377.

See Also

find.interaction,

impute, max.subtree,

plot.competing.risk, plot.rfsrc, plot.survival, plot.variable, predict.rfsrc, print.rfsrc, quantileReg, rfsrcFast, rfsrcSyn,

subsample,

stat.split, tune, var.select, vimp

Examples

```
##------------------------------------------------------------
## Survival analysis
##------------------------------------------------------------

## veteran data
## randomized trial of two treatment regimens for lung cancer
data(veteran, package = "randomForestSRC")
v.obj <- rfsrc(Surv(time, status) ~ ., data = veteran,
                   ntree = 100, block.size = 1)

## print and plot the grow object
print(v.obj)
plot(v.obj)

## plot survival curves for first 10 individuals -- direct way
matplot(v.obj$time.interest, 100 * t(v.obj$survival.oob[1:10, ]),
    xlab = "Time", ylab = "Survival", type = "l", lty = 1)

## plot survival curves for first 10 individuals -- use wrapper
plot.survival(v.obj, subset = 1:10)


## Primary biliary cirrhosis (PBC) of the liver
data(pbc, package = "randomForestSRC")
pbc.obj <- rfsrc(Surv(days, status) ~ ., pbc)
print(pbc.obj)


##------------------------------------------------------------
## Example of imputation in survival analysis
##------------------------------------------------------------

data(pbc, package = "randomForestSRC")
pbc.obj2 <- rfsrc(Surv(days, status) ~ ., pbc,
           nsplit = 10, na.action = "na.impute")


## same as above but we iterate the missing data algorithm
pbc.obj3 <- rfsrc(Surv(days, status) ~ ., pbc,
         na.action = "na.impute", nimpute = 3)
```

```
## fast way to impute the data (no inference is done)
## see impute for more details
pbc.imp <- impute(Surv(days, status) ~ ., pbc, splitrule = "random")

##--------------------------------------------------------------
## Compare RF-SRC to Cox regression
## Illustrates C-index and Brier score measures of performance
## assumes "pec" and "survival" libraries are loaded
##--------------------------------------------------------------

if (library("survival", logical.return = TRUE)
    & library("pec", logical.return = TRUE)
    & library("prodlim", logical.return = TRUE))

{
  ##prediction function required for pec
  predictSurvProb.rfsrc <- function(object, newdata, times, ...){
    ptemp <- predict(object,newdata=newdata,...)$survival
    pos <- sindex(jump.times = object$time.interest, eval.times = times)
    p <- cbind(1,ptemp)[, pos + 1]
    if (NROW(p) != NROW(newdata) || NCOL(p) != length(times))
      stop("Prediction failed")
    p
  }

  ## data, formula specifications
  data(pbc, package = "randomForestSRC")
  pbc.na <- na.omit(pbc)  ##remove NA's
  surv.f <- as.formula(Surv(days, status) ~ .)
  pec.f <- as.formula(Hist(days,status) ~ 1)

  ## run cox/rfsrc models
  ## for illustration we use a small number of trees
  cox.obj <- coxph(surv.f, data = pbc.na, x = TRUE)
  rfsrc.obj <- rfsrc(surv.f, pbc.na, ntree = 150)

  ## compute bootstrap cross-validation estimate of expected Brier score
  ## see Mogensen, Ishwaran and Gerds (2012) Journal of Statistical Software
  set.seed(17743)
  prederror.pbc <- pec(list(cox.obj,rfsrc.obj), data = pbc.na, formula = pec.f,
                       splitMethod = "bootcv", B = 50)
  print(prederror.pbc)
  plot(prederror.pbc)

  ## compute out-of-bag C-index for cox regression and compare to rfsrc
  rfsrc.obj <- rfsrc(surv.f, pbc.na)
  cat("out-of-bag Cox Analysis ...", "\n")
  cox.err <- sapply(1:100, function(b) {
    if (b%%10 == 0) cat("cox bootstrap:", b, "\n")
    train <- sample(1:nrow(pbc.na), nrow(pbc.na), replace = TRUE)
    cox.obj <- tryCatch({coxph(surv.f, pbc.na[train, ])}, error=function(ex){NULL})
    if (!is.null(cox.obj)) {
      randomForestSRC:::cindex(pbc.na$days[-train],
                                        pbc.na$status[-train],
                                        predict(cox.obj, pbc.na[-train, ]))
    } else NA
  })
  cat("\n\tOOB error rates\n\n")
  cat("\tRSF            : ", rfsrc.obj$err.rate[rfsrc.obj$ntree], "\n")
  cat("\tCox regression : ", mean(cox.err, na.rm = TRUE), "\n")
}

##------------------------------------------------------------
## Competing risks
##------------------------------------------------------------

## WIHS analysis
## cumulative incidence function (CIF) for HAART and AIDS stratified by IDU

data(wihs, package = "randomForestSRC")
wihs.obj <- rfsrc(Surv(time, status) ~ ., wihs, nsplit = 3, ntree = 100)
plot.competing.risk(wihs.obj)
cif <- wihs.obj$cif.oob
Time <- wihs.obj$time.interest
idu <- wihs$idu
cif.haart <- cbind(apply(cif[,,1][idu == 0,], 2, mean),
                   apply(cif[,,1][idu == 1,], 2, mean))
cif.aids  <- cbind(apply(cif[,,2][idu == 0,], 2, mean),
                   apply(cif[,,2][idu == 1,], 2, mean))
matplot(Time, cbind(cif.haart, cif.aids), type = "l",
        lty = c(1,2,1,2), col = c(4, 4, 2, 2), lwd = 3,
        ylab = "Cumulative Incidence")
legend("topleft",
       legend = c("HAART (Non-IDU)", "HAART (IDU)", "AIDS (Non-IDU)", "AIDS (IDU)"),
       lty = c(1,2,1,2), col = c(4, 4, 2, 2), lwd = 3, cex = 1.5)


## illustrates the various splitting rules
## illustrates event specific and non-event specific variable selection
if (library("survival", logical.return = TRUE)) {

  ## use the pbc data from the survival package
  ## events are transplant (1) and death (2)
  data(pbc, package = "survival")
  pbc$id <- NULL

  ## modified Gray's weighted log-rank splitting
  pbc.cr <- rfsrc(Surv(time, status) ~ ., pbc)

  ## log-rank event-one specific splitting
  pbc.log1 <- rfsrc(Surv(time, status) ~ ., pbc,
              splitrule = "logrank", cause = c(1,0), importance = TRUE)

  ## log-rank event-two specific splitting
  pbc.log2 <- rfsrc(Surv(time, status) ~ ., pbc,
              splitrule = "logrank", cause = c(0,1), importance = TRUE)

  ## extract VIMP from the log-rank forests: event-specific
  ## extract minimal depth from the Gray log-rank forest: non-event specific
  var.perf <- data.frame(md = max.subtree(pbc.cr)$order[, 1],
                         vimp1 = 100 * pbc.log1$importance[ ,1],
                         vimp2 = 100 * pbc.log2$importance[ ,2])
  print(var.perf[order(var.perf$md), ])

}



## ------------------------------------------------------------
## Regression analysis
## ------------------------------------------------------------

## New York air quality measurements
airq.obj <- rfsrc(Ozone ~ ., data = airquality, na.action = "na.impute")

# partial plot of variables (see plot.variable for more details)
plot.variable(airq.obj, partial = TRUE, smooth.lines = TRUE)

## motor trend cars
mtcars.obj <- rfsrc(mpg ~ ., data = mtcars)


## ------------------------------------------------------------
## Classification analysis
## ------------------------------------------------------------

## Edgar Anderson's iris data
iris.obj <- rfsrc(Species ~., data = iris)

## Wisconsin prognostic breast cancer data
data(breast, package = "randomForestSRC")
breast.obj <- rfsrc(status ~ ., data = breast, block.size=1)
plot(breast.obj)

## ------------------------------------------------------------
## Classification analysis with class imbalanced data
## ------------------------------------------------------------

data(breast, package = "randomForestSRC")
breast <- na.omit(breast)
o <- rfsrc(status ~ ., data = breast)
print(o)

## The data is imbalanced so we use balanced random forests
## with undersampling of the majority class
##
## Specifically let n0, n1 be sample sizes for majority, minority
## cases.  We sample 2 x n1 cases with majority, minority cases chosen
## with probabilities n1/n, n0/n where n=n0+n1

y <- breast$status
o <- rfsrc(status ~ ., data = breast,
           case.wt = randomForestSRC:::make.wt(y),
           sampsize = randomForestSRC:::make.size(y))
print(o)


## ------------------------------------------------------------
## Unsupervised analysis
## ------------------------------------------------------------

# two equivalent ways to implement unsupervised forests
mtcars.unspv <- rfsrc(Unsupervised() ~., data = mtcars)
mtcars2.unspv <- rfsrc(data = mtcars)

## ------------------------------------------------------------
## Multivariate regression analysis
## ------------------------------------------------------------

mtcars.mreg <- rfsrc(Multivar(mpg, cyl) ~., data = mtcars,
           block.size=1, importance = TRUE)

## extract error rates, vimp, and OOB predicted values for all targets
err <- get.mv.error(mtcars.mreg)
vmp <- get.mv.vimp(mtcars.mreg)
pred <- get.mv.predicted(mtcars.mreg)

## standardized error and vimp
err.std <- get.mv.error(mtcars.mreg, standardize = TRUE)
vmp.std <- get.mv.vimp(mtcars.mreg, standardize = TRUE)


## ------------------------------------------------------------
## Mixed outcomes analysis
## ------------------------------------------------------------

mtcars.new <- mtcars
mtcars.new$cyl <- factor(mtcars.new$cyl)
mtcars.new$carb <- factor(mtcars.new$carb, ordered = TRUE)
mtcars.mix <- rfsrc(cbind(carb, mpg, cyl) ~., data = mtcars.new, block.size=1)
print(mtcars.mix, outcome.target = "mpg")
print(mtcars.mix, outcome.target = "cyl")
plot(mtcars.mix, outcome.target = "mpg")
plot(mtcars.mix, outcome.target = "cyl")


## ------------------------------------------------------------
## Custom splitting using the pre-coded examples
## ------------------------------------------------------------

## motor trend cars
```

```
mtcars.obj <- rfsrc(mpg ~ ., data = mtcars, splitrule = "custom")

## iris analysis
iris.obj <- rfsrc(Species ~., data = iris, splitrule = "custom1")

## WIHS analysis
wihs.obj <- rfsrc(Surv(time, status) ~ ., wihs, nsplit = 3,
                  ntree = 100, splitrule = "custom1")
```

rfsrc.news *Show the NEWS file*

Description

Show the NEWS file of the **randomForestSRC** package.

Usage

```
rfsrc.news(...)
```

Arguments

...	Further arguments passed to or from other methods.

Value

None.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

rfsrcFast *Fast Random Forests*

Description

Fast approximate random forests using subsampling with forest options set to encourage computational speed. Applies to all families.

Usage

```
## S3 method for class 'rfsrc'
rfsrcFast(formula, data,
  ntree = 500,
  nsplit = 10,
  bootstrap = "by.root",
  ensemble = "oob",
  sampsize = function(x){min(x * .632, max(150, sqrt(x)))},
  samptype = "swor",
  samp = NULL,
  ntime = 50,
  forest = FALSE,
  ...)
```

Arguments

`formula`	A symbolic description of the model to be fit. If missing, unsupervised splitting is implemented.
`data`	Data frame containing the y-outcome and x-variables.
`ntree`	Number of trees.
`nsplit`	Non-negative integer value specifying number of random split points used to split a node (deterministic splitting corresponds to the value zero and is much slower).
`bootstrap`	Bootstrap protocol used in growing a tree.
`ensemble`	Specifies the type of ensemble. We request only out-of-sample which corresponds to "oob".
`sampsize`	Function specifying requested size of subsampled data relative to the original data. The requested sample size can also be passed in as a number.
`samptype`	Type of bootstrap used.
`samp`	Bootstrap specification when `"by.user"` is used.
`ntime`	Integer value used for survival to constrain ensemble calculations to a grid of `ntime` time points.
`forest`	Should the forest object be returned?
`...`	Further arguments to be passed to `rfsrc`.

Details

Calls `rfsrc` under various options (including subsampling) to encourage computational speeds. This will provide a good approximation but will not be as good as default settings of `rfsrc`.

Value

An object of class `(rfsrc, grow)`.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

See Also

rfsrc

Examples

```
## ------------------------------------------------------------
## Iowa housing regression example
## ------------------------------------------------------------

## load the Iowa housing data
data(housing, package = "randomForestSRC")

## do quick and *dirty* imputation
housing2 <- impute(SalePrice ~ ., housing,
         ntree = 50, nimpute = 1, splitrule = "random")

## grow a fast forest
o1 <- rfsrcFast(SalePrice ~ ., housing2)
o2 <- rfsrcFast(SalePrice ~ ., housing2, nodesize = 1)
print(o1)
print(o2)

## grow a fast bivariate forest
o3 <- rfsrcFast(cbind(SalePrice,Overall.Qual) ~ ., housing2)
print(o3)

## ------------------------------------------------------------
## White wine classification example
## ------------------------------------------------------------

data(wine, package = "randomForestSRC")
wine$quality <- factor(wine$quality)
o <- rfsrcFast(quality ~ ., wine)
print(o)


## ------------------------------------------------------------
## pbc survival example
## ------------------------------------------------------------

data(pbc, package = "randomForestSRC")
o <- rfsrcFast(Surv(days, status) ~ ., pbc)
print(o)

## ------------------------------------------------------------
## WIHS competing risk example
## ------------------------------------------------------------
```

```
data(wihs, package = "randomForestSRC")
o <- rfsrcFast(Surv(time, status) ~ ., wihs)
print(o)
```

rfsrcSyn *Synthetic Random Forests*

Description

Grows a synthetic random forest (RF) using RF machines as synthetic features. Applies only to regression and classification settings.

Usage

```
## S3 method for class 'rfsrc'
rfsrcSyn(formula, data, object, newdata,
  ntree = 1000, mtry = NULL, nodesize = 5, nsplit = 10,
  mtrySeq = NULL, nodesizeSeq = c(1:10,20,30,50,100),
  min.node = 3,
  fast = TRUE,
  use.org.features = TRUE,
  na.action = c("na.omit", "na.impute"),
  oob = TRUE,
  verbose = TRUE,
  ...)
```

Arguments

formula	A symbolic description of the model to be fit. Must be specified unless object is given.
data	Data frame containing the y-outcome and x-variables in the model. Must be specified unless object is given.
object	An object of class (rfsrc, synthetic). Not required when formula and data are supplied.
newdata	Test data used for prediction (optional).
ntree	Number of trees.
mtry	mtry value for over-arching synthetic forest.
nodesize	Nodesize value for over-arching synthetic forest.
nsplit	nsplit-randomized splitting for significantly increased speed.
mtrySeq	Sequence of mtry values used for fitting the collection of RF machines. If NULL, set to the default value p/3.
nodesizeSeq	Sequence of nodesize values used for the fitting the collection of RF machines.

`min.node` Minimum forest averaged number of nodes a RF machine must exceed in order to be used as a synthetic feature.

`fast` Use fast random forests, `rfsrcFast`, in place of `rfsrc`? Improves speed but may be less accurate.

`use.org.features` In addition to synthetic features, should the original features be used when fitting synthetic forests?

`na.action` Missing value action. The default `na.omit` removes the entire record if even one of its entries is `NA`. The action `na.impute` pre-imputes the data using fast imputation via `impute.rfsrc`.

`oob` Preserve "out-of-bagness" so that error rates and VIMP are honest? Default is yes (‘`oob=TRUE`’).

`verbose` Set to `TRUE` for verbose output.

`...` Further arguments to be passed to the `rfsrc` function used for fitting the synthetic forest.

Details

A collection of random forests are fit using different nodesize values. The predicted values from these machines are then used as synthetic features (called RF machines) to fit a synthetic random forest (the original features are also used in constructing the synthetic forest). Currently only implemented for regression and classification settings (univariate and multivariate).

Synthetic features are calculated using out-of-bag (OOB) data to avoid over-using training data. However, to guarantee that performance values such as error rates and VIMP are honest, bootstrap draws are fixed across all trees used in the construction of the synthetic forest and its synthetic features. The option ‘`oob=TRUE`’ ensures that this happens. Change this option at your own peril.

If values for `mtrySeq` are given, RF machines are constructed for each combination of nodesize and mtry values specified by `nodesizeSeq` `mtrySeq`.

Value

A list with the following components:

`rfMachines` RF machines used to construct the synthetic features.

`rfSyn` The (grow) synthetic RF built over training data.

`rfSynPred` The predict synthetic RF built over test data (if available).

`synthetic` List containing the synthetic features.

`opt.machine` Optimal machine: RF machine with smallest OOB error rate.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H. and Malley J.D. (2014). Synthetic learning machines. *BioData Mining*, 7:28.

See Also

rfsrc, rfsrcFast

Examples

```
## ------------------------------------------------------------
## compare synthetic forests to regular forest (classification)
## ------------------------------------------------------------

## rfsrc and rfsrcSyn calls
if (library("mlbench", logical.return = TRUE)) {

  ## simulate the data
  ring <- data.frame(mlbench.ringnorm(250, 20))

  ## classification forests
  ringRF <- rfsrc(classes ~., ring)

  ## synthetic forests
  ## 1 = nodesize varied
  ## 2 = nodesize/mtry varied
  ringSyn1 <- rfsrcSyn(classes ~., ring)
  ringSyn2 <- rfsrcSyn(classes ~., ring, mtrySeq = c(1, 10, 20))

  ## test-set performance
  ring.test <- data.frame(mlbench.ringnorm(500, 20))
  pred.ringRF <- predict(ringRF, newdata = ring.test)
  pred.ringSyn1 <- rfsrcSyn(object = ringSyn1, newdata = ring.test)$rfSynPred
  pred.ringSyn2 <- rfsrcSyn(object = ringSyn2, newdata = ring.test)$rfSynPred


  print(pred.ringRF)
  print(pred.ringSyn1)
  print(pred.ringSyn2)

}

## ------------------------------------------------------------
## compare synthetic forest to regular forest (regression)
## ------------------------------------------------------------

## simulate the data
n <- 250
ntest <- 1000
N <- n + ntest
d <- 50
std <- 0.1
x <- matrix(runif(N * d, -1, 1), ncol = d)
y <- 1 * (x[,1] + x[,4]^3 + x[,9] + sin(x[,12]*x[,18]) + rnorm(n, sd = std)>.38)
dat <- data.frame(x = x, y = y)
test <- (n+1):N
```

```
## regression forests
regF <- rfsrc(y ~ ., dat[-test, ], )
pred.regF <- predict(regF, dat[test, ], importance = "none")

## synthetic forests using fast rfsrc
synF1 <- rfsrcSyn(y ~ ., dat[-test, ], fast = TRUE, newdata = dat[test, ])
synF2 <- rfsrcSyn(y ~ ., dat[-test, ], fast = TRUE,
  newdata = dat[test, ], mtrySeq = c(1, 10, 20, 30, 40, 50))

## standardized MSE performance
mse <- c(tail(pred.regF$err.rate, 1),
         tail(synF1$rfSynPred$err.rate, 1),
         tail(synF2$rfSynPred$err.rate, 1)) / var(y[-test])
names(mse) <- c("forest", "synthetic1", "synthetic2")
print(mse)

## ------------------------------------------------------------
## multivariate synthetic forests
## ------------------------------------------------------------

mtcars.new <- mtcars
mtcars.new$cyl <- factor(mtcars.new$cyl)
mtcars.new$carb <- factor(mtcars.new$carb, ordered = TRUE)
trn <- sample(1:nrow(mtcars.new), nrow(mtcars.new)/2)
mvSyn <- rfsrcSyn(cbind(carb, mpg, cyl) ~., mtcars.new[trn,])
mvSyn.pred <- rfsrcSyn(object = mvSyn, newdata = mtcars.new[-trn,])
```

stat.split *Acquire Split Statistic Information*

Description

Extract split statistic information from the forest. The function returns a list of length ntree, in which each element corresponds to a tree. The element [[b]] is itself a vector of length xvar.names identified by its x-variable name. Each element [[b]]$xvar contains the complete list of splits on xvar with associated identifying information. The information is as follows:

1. *treeID* Tree identifier.
2. *nodeID* Node identifier.
3. *parmID* Variable indentifier.
4. *contPT* Value node was split in the case of a continuous variable.
5. *mwcpSZ* Size of the multi-word complementary pair in the case of a factor split.
6. *dpthID* Zero (0) based depth of split.

7. *spltTY* Split type for parent node:

bit 1	bit 0	meaning
——	——	——-
0	0	0 = both daughters have valid splits
0	1	1 = only the right daughter is terminal
1	0	2 = only the left daughter is terminal
1	1	3 = both daughters are terminal

8. *spltEC* End cut statistic for real valued variables between [0,0.5] that is small when the split is towards the edge and large when the split is towards the middle. Subtracting this value from 0.5 yields the end cut statistic studied in Ishwaran (2014) and is a way to identify ECP behavior (end cut preference behavior).
9. *spltST* Split statistic:
   (a) For objects of class (rfsrc, grow), this is the split statistic that resulted in the variable being choosen for the split.
   (b) For an object of class (rfsrc, pred) this is the variance of the response within the node for the test data. This value is relevant only for real valued responses. In classification and survival, it is not relevant.

Usage

```
## S3 method for class 'rfsrc'
stat.split(object, ...)
```

Arguments

`object` An object of class `(rfsrc, grow)`, `(rfsrc, synthetic)` or `(rfsrc,   predict)`

`...` Further arguments passed to or from other methods.

Value

Invisibly, a list with the following components:

`...` ...

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H. (2015). The effect of splitting on random forests. *Machine Learning*, 99:75-118.

Examples

```
## run a forest, then make a call to stat.split
grow.obj <- rfsrc(mpg ~., data = mtcars, membership=TRUE, statistics=TRUE)
stat.obj <- stat.split(grow.obj)

## nice wrapper to extract split-statistic for desired variable
## for continuous variables plots ECP data
get.split <- function(splitObj, xvar, inches = 0.1, ...) {
  which.var <- which(names(splitObj[[1]]) == xvar)
  ntree <- length(splitObj)
  stat <- data.frame(do.call(rbind, sapply(1:ntree, function(b) {
    splitObj[[b]][which.var]})))
  dpth <- stat$dpthID
  ecp <- 1/2 - stat$spltEC
  sp <- stat$contPT
  if (!all(is.na(sp))) {
    fgC <- function(x) {
      as.numeric(as.character(cut(x, breaks = c(-1, 0.2, 0.35, 0.5),
      labels = c(1, 4, 2))))
    }
    symbols(jitter(sp), jitter(dpth), ecp, inches = inches, bg = fgC(ecp),
      xlab = xvar, ylab = "node depth", ...)
    legend("topleft", legend = c("low ecp", "med ecp", "high ecp"),
      fill = c(1, 4, 2))
   }
  invisible(stat)
}

## use get.split to investigate ECP behavior of variables
get.split(stat.obj, "disp")
```

---

subsample   *Subsample Forests for VIMP Confidence Intervals*

Description

Use subsampling to calculate confidence intervals and standard errors for VIMP (variable importance). Applies to all families.

Usage

```
## S3 method for class 'rfsrc'
subsample(obj,
  B = 100,
  block.size = 1,
  subratio = NULL,
  stratify = TRUE,
  joint = FALSE,
  bootstrap = FALSE,
  verbose = TRUE)
```

Arguments

`obj` A forest grow object.

`B` Number of subsamples (or number of bootstraps).

`block.size` Specifies number of trees in a block when calculating VIMP. This is over-ridden if VIMP is present in the original grow call in which case the grow value is used.

`subratio` Ratio of subsample size to original sample size. The default is the inverse square root of the sample size.

`stratify` Use stratified subsampling? See details below.

`joint` Include the VIMP for all variables jointly perturbed? This is useful reference problems where one might be suspicious that many (or all) variables are noise.

`bootstrap` Use double bootstrap approach in place of subsampling? Much slower, but potentially more accurate.

`verbose` Provide verbose output?

Details

Given a forest object, subsamples the forest to obtain standard errors and confidence intervals for VIMP (Ishwaran and Lu, 2018). If bootstrapping is requested, then the double bootstrap is applied in place of subsampling.

If VIMP is not present in the original forest object, the algorithm will first need to calculate VIMP. Therefore, if the user plans to make repeated calls to `subsample`, it is advisable to include VIMP in the original grow call. Note that the subsampled forest inherits the same tuning parameters as the original forests. While a sophisticated algorithm is utilized to acquire as many of the original forest parameters as possible to be applied to the subsampled forest, there are some conditions where this will fail: for example there are certain settings where the user has specified non-standard sampling in the grow forest.

Delete-d jackknife estimators (Shao and Wu, 1989) are returned along with subsampling estimators (Politis and Romano, 1994). While these two methods are closely related, standard errors for delete-d estimators are generally larger than the subsampled estimates, which is a form of bias correction, which occurs primarily for variables with true signal. Confidence interval coverage is generally better under delete-d estimators. Note that undercoverage for strong variables and overcoverage for noise variables exhibited by both estimators may be beneficial if the goal is variable selection (Ishwaran and Lu, 2018).

By default, stratified subsampling is used for classification, survival, and competing risk families. For classification, stratification is on the class label, while for survival and competing risk, stratification is on the event type and censoring. Users are discouraged from over-riding this option, especially in small sample settings, as this could lead to error due to subsampled data not having full representation of class labels in classification settings, and in survival settings, subsampled data may be devoid of deaths and/or have reduced number of competing risks. Finally, note that stratified sampling is not available for multivariate families in which case users should especially exercise caution when selecting subsampling rates.

Note that subsampling and bootstrapping do not take into account missing data imputation that may have been performed on the forest grow object. In such cases there is no guarantee that standard errors and confidence intervals will be accurate.

The function `extract.subsample` is useful for studying the subsampled object. This function has been exported for the convenience of users to experiment with.

When printing and or plotting results, the default setting is to standardize VIMP, where for regression families, VIMP is standardized by dividing by the variance and multiplying by 100. For all other families, VIMP is scaled by 100. This can be turned off using the option `standardize` in those wrappers.

Value

A list with the following key components:

`rf`	Original forest grow object.
`vmp`	Variable importance subsampled values.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H. and Lu M. (2018). Standard errors and confidence intervals for variable importance in random forest regression, classification, and survival. Statistics in Medicine (in press).

Politis, D.N. and Romano, J.P. (1994). Large sample confidence regions based on subsamples under minimal assumptions. *The Annals of Statistics*, 22(4):2031-2050.

Shao, J. and Wu, C.J. (1989). A general theory for jackknife variance estimation. *The Annals of Statistics*, 17(3):1176-1197.

See Also

`plot.subsample`, `rfsrc`, `vimp`

Examples

```
## ------------------------------------------------------------
## regression example
## ------------------------------------------------------------

## grow the forest - request VIMP
reg.o <- rfsrc(mpg ~ ., mtcars)

## very small sample size so need largish subratio
reg.smp.o <- subsample(reg.o, B = 100, subratio = .5)

## plot confidence regions
plot.subsample(reg.smp.o)
```

```
## summary of results
print(reg.smp.o)

## now try the double bootstrap (slow!!)
reg.dbs.o <- subsample(reg.o, B = 100, bootstrap = TRUE)
print(reg.dbs.o)
plot.subsample(reg.dbs.o)

## ------------------------------------------------------------
## classification example
## ------------------------------------------------------------

## 3 non-linear, 15 linear, and 5 noise variables
if (library(”caret”, logical.return = TRUE)) {
  d <- twoClassSim(1000, linearVars = 15, noiseVars = 5)

  ## VIMP based on (default) misclassification error
  cls.o <- rfsrc(Class ~ ., d)
  cls.smp.o <- subsample(cls.o, B = 100)
  plot.subsample(cls.smp.o, cex = .7)

  ## same as above, but with VIMP defined using normalized Brier score
  cls.o2 <- rfsrc(Class ~ ., d, perf.type = ”brier”)
  cls.smp.o2 <- subsample(cls.o2, B = 100)
  plot.subsample(cls.smp.o2, cex = .7)
}

## ------------------------------------------------------------
## survival example
## ------------------------------------------------------------

data(pbc, package = ”randomForestSRC”)
srv.o <- rfsrc(Surv(days, status) ~ ., pbc)
srv.smp.o <- subsample(srv.o, B = 100)
plot.subsample(srv.smp.o)

## ------------------------------------------------------------
## competing risk example
## target event is death (event = 2)
## ------------------------------------------------------------

if (library(”survival”, logical.return = TRUE)) {
  data(pbc, package = ”survival”)
  pbc$id <- NULL
  cr.o <- rfsrc(Surv(time, status) ~ ., pbc, splitrule = ”logrank”, cause = 2)
  cr.smp.o <- subsample(cr.o, B = 100)
  plot.subsample(cr.smp.o, target = 2)
}

## ------------------------------------------------------------
## multivariate family
## ------------------------------------------------------------
```

```
if (library("mlbench", logical.return = TRUE)) {
  ## simulate the data
  data(BostonHousing)
  bh <- BostonHousing
  bh$rm <- factor(round(bh$rm))
  o <- rfsrc(cbind(medv, rm) ~ ., bh)
  so <- subsample(o)
  plot(so)
  plot(so, m.target = "rm")
}

## ------------------------------------------------------------
## largish data example - use rfsrcFast for fast forests
## ------------------------------------------------------------

if (library("caret", logical.return = TRUE)) {
  ## largish data set
  d <- twoClassSim(1000, linearVars = 15, noiseVars = 5)

  ## use a subsampled forest with Brier score performance
  o <- rfsrcFast(Class ~ ., d, ntree = 100, perf.type = "brier")
  so <- subsample(o, B = 100)
  plot.subsample(so, cex = .7)
}
```

tune — *Tune Random Forest for the optimal mtry and nodesize parameters*

Description

Finds the optimal mtry and nodesize tuning parameter for a random forest using out-of-bag (OOB) error. Applies to all families.

Usage

```
## S3 method for class 'rfsrc'
tune(formula, data,
  mtryStart = ncol(data) / 2,
  nodesizeTry = c(1:9, seq(10, 100, by = 5)), ntreeTry = 50,
  stepFactor = 1.25, improve = 1e-3, strikeout = 3, maxIter = 25,
  trace = FALSE, doBest = TRUE, ...)
```

Arguments

`formula`	A symbolic description of the model to be fit.

`data`	Data frame containing the y-outcome and x-variables.
`mtryStart`	Starting value of mtry.
`nodesizeTry`	Values of nodesize optimized over.
`ntreeTry`	Number of trees used for the tuning step.
`stepFactor`	At each iteration, mtry is inflated (or deflated) by this value.
`improve`	The (relative) improvement in OOB error must be by this much for the search to continue.
`strikeout`	The search is discontinued when the relative improvement in OOB error is negative. However `strikeout` allows for some tolerance in this. If a negative improvement is noted a total of `strikeout` times, the search is stopped. Increase this value only if you want an exhaustive search.
`maxIter`	The maximum number of iterations allowed for each mtry bisection search.
`trace`	Print the progress of the search?
`doBest`	Return a forest fit with the optimal mtry and nodesize parameters?
`...`	Further options to be passed to `rfsrcFast`.

Details

Returns a matrix whose first and second columns contain the nodesize and mtry values searched and whose third column is the corresponding OOB error. Uses standardized OOB error and in the case of multivariate forests it is the averaged standardized OOB error over the outcomes and for competing risks it is the averaged standardized OOB error over the event types.

If `doBest=TRUE`, also returns a forest object fit using the optimal `mtry` and `nodesize` values.

All calculations (including the final optimized forest) are based on the fast forest interface `rfsrcFast`. Using `rfsrcFast` allows the optimization strategy to be implemented quickly, however the solution can only be considered approximate. Users may wish to tweak various options to improve stability. For example, increasing `ntreeTry` (which is set to 50 for speed) may help. It is also useful to look at contour plots of the OOB error as a function of `mtry` and `nodesize` (see example below) to identify regions of the parameter space where error rate is small.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

See Also

`rfsrcFast`

Examples

```
## ------------------------------------------------------------
## White wine classification example
## ------------------------------------------------------------

## load the data
```

```
data(wine, package = "randomForestSRC")
wine$quality <- factor(wine$quality)

## default tuning call
o <- tune(quality ~ ., wine)

## here is the optimized forest
print(o$rf)

## visualize the nodesize/mtry OOB surface
if (library("akima", logical.return = TRUE)) {

  ## nice little wrapper for plotting results
  plot.tune <- function(o, linear = TRUE) {
    x <- o$results[,1]
    y <- o$results[,2]
    z <- o$results[,3]
    so <- interp(x=x, y=y, z=z, linear = linear)
    idx <- which.min(z)
    x0 <- x[idx]
    y0 <- y[idx]
    filled.contour(x = so$x,
                   y = so$y,
                   z = so$z,
                   xlim = range(so$x, finite = TRUE) + c(-2, 2),
                   ylim = range(so$y, finite = TRUE) + c(-2, 2),
                   color.palette =
                     colorRampPalette(c("yellow", "red")),
                   xlab = "nodesize",
                   ylab = "mtry",
                   main = "OOB error for nodesize and mtry",
                   key.title = title(main = "OOB error", cex.main = 1),
                   plot.axes = {axis(1);axis(2);points(x0,y0,pch="x",cex=1,font=2);
                                points(x,y,pch=16,cex=.25)})
  }

  ## plot the surface
  plot.tune(o)

}
```

var.select *Variable Selection*

Description

Variable selection using minimal depth.

Usage

```
## S3 method for class 'rfsrc'
var.select(formula,
  data,
  object,
  cause,
  m.target,
  method = c("md", "vh", "vh.vimp"),
  conservative = c("medium", "low", "high"),
  ntree = (if (method == "md") 1000 else 500),
  mvars = (if (method != "md") ceiling(ncol(data)/5) else NULL),
  mtry = (if (method == "md") ceiling(ncol(data)/3) else NULL),
  nodesize = 2, splitrule = NULL, nsplit = 10, xvar.wt = NULL,
  refit = (method != "md"), fast = FALSE,
  na.action = c("na.omit", "na.impute"),
  always.use = NULL, nrep = 50, K = 5, nstep = 1,
  prefit =  list(action = (method != "md"), ntree = 100,
  mtry = 500, nodesize = 3, nsplit = 1),
  verbose = TRUE, ...)
```

Arguments

`formula` A symbolic description of the model to be fit. Must be specified unless `object` is given.

`data` Data frame containing the y-outcome and x-variables in the model. Must be specified unless `object` is given.

`object` An object of class `(rfsrc, grow)`. Not required when `formula` and `data` are supplied.

`cause` Integer value between 1 and `J` indicating the event of interest for competing risks, where `J` is the number of event types (this option applies only to competing risk families). The default is to use the first event type.

`m.target` Character value for multivariate families specifying the target outcome to be used. If left unspecified, the algorithm will choose a default target.

`method` Variable selection method:

`md`**:** minimal depth (default).

`vh`**:** variable hunting.

`vh.vimp`**:** variable hunting with VIMP (variable importance).

`conservative` Level of conservativeness of the thresholding rule used in minimal depth selection:

`high`**:** Use the most conservative threshold.

`medium`**:** Use the default less conservative tree-averaged threshold.

`low`**:** Use the more liberal one standard error rule.

`ntree` Number of trees to grow.

`mvars` Number of randomly selected variables used in the variable hunting algorithm (ignored when ‘`method="md"`’).

`mtry` The mtry value used.

`nodesize` Forest average terminal node size.

`splitrule` Splitting rule used.

`nsplit` If non-zero, the specified tree splitting rule is randomized which significantly increases speed.

`xvar.wt` Vector of non-negative weights specifying the probability of selecting a variable for splitting a node. Must be of dimension equal to the number of variables. Default (`NULL`) invokes uniform weighting or a data-adaptive method depending on `prefit$action`.

`refit` Should a forest be refit using the selected variables?

`fast` Speeds up the cross-validation used for variable hunting for a faster analysis. See miscellanea below.

`na.action` Action to be taken if the data contains `NA` values.

`always.use` Character vector of variable names to always be included in the model selection procedure and in the final selected model.

`nrep` Number of Monte Carlo iterations of the variable hunting algorithm.

`K` Integer value specifying the `K`-fold size used in the variable hunting algorithm.

`nstep` Integer value controlling the step size used in the forward selection process of the variable hunting algorithm. Increasing this will encourage more variables to be selected.

`prefit` List containing parameters used in preliminary forest analysis for determining weight selection of variables. Users can set all or some of the following parameters:

- `action`**:** Determines how (or if) the preliminary forest is fit. See details below.
- `ntree`**:** Number of trees used in the preliminary analysis.
- `mtry`**:** mtry used in the preliminary analysis.
- `nodesize`**:** nodesize used in the preliminary analysis.
- `nsplit`**:** nsplit value used in the preliminary analysis.

`verbose` Set to `TRUE` for verbose output.

`...` Further arguments passed to forest grow call.

Details

This function implements random forest variable selection using tree minimal depth methodology (Ishwaran et al., 2010). The option ‘`method`’ allows for two different approaches:

1. ‘`method="md"`’

   Invokes minimal depth variable selection. Variables are selected using minimal depth variable selection. Uses all data and all variables simultaneously. This is basically a front-end to the `max.subtree` wrapper. Users should consult the `max.subtree` help file for details.

   Set ‘`mtry`’ to larger values in high-dimensional problems.

2. ‘method="vh"’ or ‘method="vh.vimp"’

   Invokes variable hunting. Variable hunting is used for problems where the number of variables is substantially larger than the sample size (e.g., p/n is greater than 10). It is always prefered to use ‘method="md"’, but to find more variables, or when computations are high, variable hunting may be preferred.

   When ‘method="vh"’: Using training data from a stratified K-fold subsampling (stratification based on the y-outcomes), a forest is fit using mvars randomly selected variables (variables are chosen with probability proportional to weights determined using an initial forest fit; see below for more details). The mvars variables are ordered by increasing minimal depth and added sequentially (starting from an initial model determined using minimal depth selection) until joint VIMP no longer increases (signifying the final model). A forest is refit to the final model and applied to test data to estimate prediction error. The process is repeated nrep times. Final selected variables are the top P ranked variables, where P is the average model size (rounded up to the nearest integer) and variables are ranked by frequency of occurrence.

   The same algorithm is used when ‘method="vh.vimp"’, but variables are ordered using VIMP. This is faster, but not as accurate.

Miscellanea

1. When variable hunting is used, a preliminary forest is run and its VIMP is used to define the probability of selecting a variable for splitting a node. Thus, instead of randomly selecting mvars at random, variables are selected with probability proportional to their VIMP (the probability is zero if VIMP is negative). A preliminary forest is run once prior to the analysis if prefit$action=TRUE, otherwise it is run prior to each iteration (this latter scenario can be slow). When ‘method="md"’, a preliminary forest is fit only if prefit$action=TRUE. Then instead of randomly selecting mtry variables at random, mtry variables are selected with probability proportional to their VIMP. In all cases, the entire option is overridden if xvar.wt is non-null.
2. If object is supplied and ‘method="md"’, the grow forest from object is parsed for minimal depth information. While this avoids fitting another forest, thus saving computational time, certain options no longer apply. In particular, the value of cause plays no role in the final selected variables as minimal depth is extracted from the grow forest, which has already been grown under a preselected cause specification. Users wishing to specify cause should instead use the formula and data interface. Also, if the user requests a prefitted forest via prefit$action=TRUE, then object is not used and a refitted forest is used in its place for variable selection. Thus, the effort spent to construct the original grow forest is not used in this case.
3. If ‘fast=TRUE’, and variable hunting is used, the training data is chosen to be of size n/K, where n=sample size (i.e., the size of the training data is swapped with the test data). This speeds up the algorithm. Increasing K also helps.
4. Can be used for competing risk data. When ‘method="vh.vimp"’, variable selection based on VIMP is confined to an event specific cause specified by cause. However, this can be unreliable as not all y-outcomes can be guaranteed when subsampling (this is true even when stratifed subsampling is used as done here).

Value

Invisibly, a list with the following components:

err.rate Prediction error for the forest (a vector of length nrep if variable hunting is used).

modelsize Number of variables selected.

topvars Character vector of names of the final selected variables.

varselect Useful output summarizing the final selected variables.

rfsrc.refit.obj
Refitted forest using the final set of selected variables (requires 'refit=TRUE').

md.obj Minimal depth object. NULL unless 'method="md"'.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H., Kogalur U.B., Gorodeski E.Z, Minn A.J. and Lauer M.S. (2010). High-dimensional variable selection for survival data. *J. Amer. Statist. Assoc.*, 105:205-217.

Ishwaran H., Kogalur U.B., Chen X. and Minn A.J. (2011). Random survival forests for high-dimensional data. *Statist. Anal. Data Mining*, 4:115-132.

See Also

find.interaction, max.subtree, vimp

Examples

```
## ------------------------------------------------------------
## Minimal depth variable selection
## survival analysis
## use larger node size which is better for minimal depth
## ------------------------------------------------------------

data(pbc, package = "randomForestSRC")
pbc.obj <- rfsrc(Surv(days, status) ~ ., pbc, nodesize = 20, importance = TRUE)

# default call corresponds to minimal depth selection
vs.pbc <- var.select(object = pbc.obj)
topvars <- vs.pbc$topvars

# the above is equivalent to
max.subtree(pbc.obj)$topvars

# different levels of conservativeness
var.select(object = pbc.obj, conservative = "low")
var.select(object = pbc.obj, conservative = "medium")
var.select(object = pbc.obj, conservative = "high")

## ------------------------------------------------------------
## Minimal depth variable selection
## competing risk analysis
## use larger node size which is better for minimal depth
## ------------------------------------------------------------

## competing risk data set involving AIDS in women
data(wihs, package = "randomForestSRC")
vs.wihs <- var.select(Surv(time, status) ~ ., wihs, nsplit = 3,
                      nodesize = 20, ntree = 100, importance = TRUE)

## competing risk analysis of pbc data from survival package
## implement cause-specific variable selection
if (library("survival", logical.return = TRUE)) {
  data(pbc, package = "survival")
  pbc$id <- NULL
  var.select(Surv(time, status) ~ ., pbc, cause = 1)
  var.select(Surv(time, status) ~ ., pbc, cause = 2)
}

## ------------------------------------------------------------
## Minimal depth variable selection
## classification analysis
## ------------------------------------------------------------

vs.iris <- var.select(Species ~ ., iris)

## ------------------------------------------------------------
## Variable hunting high-dimensional example
## van de Vijver microarray breast cancer survival data
## nrep is small for illustration; typical values are nrep = 100
## ------------------------------------------------------------

data(vdv, package = "randomForestSRC")
vh.breast <- var.select(Surv(Time, Censoring) ~ ., vdv,
      method = "vh", nrep = 10, nstep = 5)

# plot top 10 variables
plot.variable(vh.breast$rfsrc.refit.obj,
  xvar.names = vh.breast$topvars[1:10])
plot.variable(vh.breast$rfsrc.refit.obj,
  xvar.names = vh.breast$topvars[1:10], partial = TRUE)

## similar analysis, but using weights from univarate cox p-values
if (library("survival", logical.return = TRUE))
{
  cox.weights <- function(rfsrc.f, rfsrc.data) {
    event.names <- all.vars(rfsrc.f)[1:2]
    p <- ncol(rfsrc.data) - 2
    event.pt <- match(event.names, names(rfsrc.data))
    xvar.pt <- setdiff(1:ncol(rfsrc.data), event.pt)
    sapply(1:p, function(j) {
      cox.out <- coxph(rfsrc.f, rfsrc.data[, c(event.pt, xvar.pt[j])])
      pvalue <- summary(cox.out)$coef[5]
      if (is.na(pvalue)) 1.0 else 1/(pvalue + 1e-100)
    })
  }
  data(vdv, package = "randomForestSRC")
  rfsrc.f <- as.formula(Surv(Time, Censoring) ~ .)
  cox.wts <- cox.weights(rfsrc.f, vdv)
  vh.breast.cox <- var.select(rfsrc.f, vdv, method = "vh", nstep = 5,
    nrep = 10, xvar.wt = cox.wts)
}
```

vdv *van de Vijver Microarray Breast Cancer*

Description

Gene expression profiling for predicting clinical outcome of breast cancer (van't Veer et al., 2002). Microarray breast cancer data set of 4707 expression values on 78 patients with survival information.

References

van't Veer L.J. et al. (2002). Gene expression profiling predicts clinical outcome of breast cancer. *Nature*, **12**, 530–536.

Examples

```
data(vdv, package = "randomForestSRC")
```

veteran *Veteran's Administration Lung Cancer Trial*

Description

Randomized trial of two treatment regimens for lung cancer. This is a standard survival analysis data set.

Source

Kalbfleisch and Prentice, *The Statistical Analysis of Failure Time Data.*

References

Kalbfleisch J. and Prentice R, (1980) *The Statistical Analysis of Failure Time Data.* New York: Wiley.

Examples

```
data(veteran, package = "randomForestSRC")
```

vimp *VIMP for Single or Grouped Variables*

Description

Calculate variable importance (VIMP) for a single variable or group of variables for training or test data.

Usage

```
## S3 method for class 'rfsrc'
vimp(object, xvar.names, m.target = NULL,
  importance = c("permute", "random", "anti"), block.size = 1,
  joint = FALSE, subset, seed = NULL, do.trace = FALSE, ...)
```

Arguments

object	An object of class (rfsrc, grow) or (rfsrc, forest). Requires 'forest=TRUE' in the original rfsrc call.
xvar.names	Names of the x-variables to be used. If not specified all variables are used.
m.target	Character value for multivariate families specifying the target outcome to be used. If left unspecified, the algorithm will choose a default target.
importance	Type of VIMP.
block.size	Specifies number of trees in a block when calculating VIMP.
joint	Individual or joint VIMP?
subset	Vector indicating which rows of the grow data to restrict VIMP calculations to; i.e. this option yields VIMP which is restricted to a specific subset of the data. Note that the vector should correspond to the rows of object$xvar and not the original data passed in the grow call. All rows used if not specified.
seed	Negative integer specifying seed for the random number generator.
do.trace	Number of seconds between updates to the user on approximate time to completion.
...	Further arguments passed to or from other methods.

Details

Using a previously grown forest, calculate the VIMP for variables xvar.names. By default, VIMP is calculated for the original data, but the user can specify a new test data for the VIMP calculation using newdata. See rfsrc for more details about how VIMP is calculated.

Joint VIMP is requested using 'joint' and equals importance for a group of variables when the group is perturbed simultaneously.

Value

An object of class (rfsrc, predict) containing importance values.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H. (2007). Variable importance in binary regression trees and forests, *Electronic J. Statist.*, 1:519-537.

See Also

rfsrc

Examples

```
## ------------------------------------------------------------
## classification example
## showcase different vimp
## ------------------------------------------------------------

iris.obj <- rfsrc(Species ~ ., data = iris)

# Permutation vimp
print(vimp(iris.obj)$importance)

# Random daughter vimp
print(vimp(iris.obj, importance = "random")$importance)

# Joint permutation vimp
print(vimp(iris.obj, joint = TRUE)$importance)

# Paired vimp
print(vimp(iris.obj, c("Petal.Length", "Petal.Width"), joint = TRUE)$importance)
print(vimp(iris.obj, c("Sepal.Length", "Petal.Width"), joint = TRUE)$importance)


## ------------------------------------------------------------
## regression example
## ------------------------------------------------------------

airq.obj <- rfsrc(Ozone ~ ., airquality)
print(vimp(airq.obj))


## ------------------------------------------------------------
## regression example where vimp is calculated on test data
## ------------------------------------------------------------

set.seed(100080)
train <- sample(1:nrow(airquality), size = 80)
airq.obj <- rfsrc(Ozone~., airquality[train, ])
```

```
#training data vimp
print(airq.obj$importance)
print(vimp(airq.obj)$importance)

#test data vimp
print(vimp(airq.obj, newdata = airquality[-train, ])$importance)

## ------------------------------------------------------------
## survival example
## study how vimp depends on tree imputation
## makes use of the subset option
## ------------------------------------------------------------

data(pbc, package = "randomForestSRC")

# determine which records have missing values
which.na <- apply(pbc, 1, function(x){any(is.na(x))})

# impute the data using na.action = "na.impute"
pbc.obj <- rfsrc(Surv(days,status) ~ ., pbc, nsplit = 3,
        na.action = "na.impute", nimpute = 1)

# compare vimp based on records with no missing values
# to those that have missing values
# note the option na.action="na.impute" in the vimp() call
vimp.not.na <- vimp(pbc.obj, subset = !which.na, na.action = "na.impute")$importance
vimp.na <- vimp(pbc.obj, subset = which.na, na.action = "na.impute")$importance
print(data.frame(vimp.not.na, vimp.na))
```

wihs *Women's Interagency HIV Study (WIHS)*

Description

Competing risk data set involving AIDS in women.

Format

A data frame containing:

time	time to event
status	censoring status: 0=censoring, 1=HAART initiation, 2=AIDS/Death before HAART
ageatfda	age in years at time of FDA approval of first protease inhibitor
idu	history of IDU: 0=no history, 1=history
black	race: 0=not African-American; 1=African-American
cd4nadir	CD4 count (per 100 cells/ul)

Source

Study included 1164 women enrolled in WIHS, who were alive, infected with HIV, and free of clinical AIDS on December, 1995, when the first protease inhibitor (saquinavir mesylate) was approved by the Federal Drug Administration. Women were followed until the first of the following occurred: treatment initiation, AIDS diagnosis, death, or administrative censoring (September, 2006). Variables included history of injection drug use at WIHS enrollment, whether an individual was African American, age, and CD4 nadir prior to baseline.

References

Bacon M.C, von Wyl V., Alden C., et al. (2005). The Women's Interagency HIV Study: an observational cohort brings clinical sciences to the bench, *Clin Diagn Lab Immunol*, 12(9):1013-1019.

Examples

```
data(wihs, package = "randomForestSRC")
wihs.obj <- rfsrc(Surv(time, status) ~ ., wihs, nsplit = 3, ntree = 100)
```

---

wine *White Wine Quality Data*

Description

The inputs include objective tests (e.g. PH values) and the output is based on sensory data (median of at least 3 evaluations made by wine experts) of white wine. Each expert graded the wine quality between 0 (very bad) and 10 (very excellent).

References

Cortez, P., Cerdeira, A., Almeida, F., Matos T. and Reis, J. (2009). Modeling wine preferences by data mining from physicochemical properties. In *Decision Support Systems*, Elsevier, 47(4):547-553.

Examples

```
data(wine, package = "randomForestSRC")
```

Index