Package ‘randomForestSRC’

December 24, 2015

Version 2.0.5

Date 2015-12-23

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>

Depends R (>= 3.1.0), parallel

Suggests glmnet, XML, survival, pec, prodlim, Hmisc, mlbench

Description A unified treatment of Breiman's random forests for survival, regression and classification problems based on Ishwaran and Kogalur's random survival forests (RSF) package. The package runs in both serial and parallel (OpenMP) modes. Now extended to include multivariate and unsupervised forests.

License GPL (>= 3)

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

NeedsCompilation yes

Repository CRAN

Date/Publication 2015-12-24 13:46:53

R topics documented:

`randomForestSRC-package` *Random Forests for Survival, Regression and Classification (RF-SRC)*

Description

This package provides a 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 as are unsupervised forests. 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. Variable selection is implemented using minimal depth variable selection (Ishwaran et al. 2010). 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.

OpenMP Parallel Processing – Installation

This package implements OpenMP shared-memory parallel programming. However, the default installation will only execute serially. To utilize OpenMP, the target architecture and operating system must first support it.

To install the package with OpenMP parallel processing enabled, on most non-Windows systems, do the following:

1. Download the package source code randomForestSRC_X.x.x.tar.gz from CRAN (do not download the binary).
2. Open a console, navigate to the directory containing the tarball, and untar it using the command `tar -xvf randomForestSRC_X.x.x.tar.gz`

3. This will create a directory structure with the root directory of the package named `randomForestSRC`. Change into the root directory of the package using the command `cd randomForestSRC`
4. Run autoconf using the command `autoconf`
5. Change back to your working directory using the command `cd ..`
6. Run `R CMD INSTALL randomForestSRC` on the modified package. Ensure that you do not target the unmodified tarball, but instead act on the directory structure you just modified.

To install the package with OpenMP parallel processing enabled, on most Windows systems, do the following:

1. Download the Windows binary file randomForestSRC_X.x.x.zip from http://www.ccs.miami.edu/~hishwaran/rfsrc.html
2. If you are using the R GUI, start the GUI. From the menu click on
   `Packages > Install package(s) from local zip files`
   Then navigate to the directory where you downloaded the zip file and click on it.

OpenMP Parallel Processing – Setting the Number of CPUs

There are several ways to control the number of CPU cores that the package accesses during OpenMP parallel execution. First, you will need to determine the number of cores on your local machine. Do this by starting an R session and issuing the command `detectCores()`.

Then you can do the following:

At the start of every R session, you can set the number of cores accessed during OpenMP parallel execution by issuing the command `options(rf.cores = x)`, where `x` is the number of cores. If `x` is a negative number, the package will access the maximum number of cores on your machine. The options command can also be placed in the users .Rprofile file for convenience. You can, alternatively, initialize the environment variable `RF_CORES` in your shell environment.

The default value for rf.cores is -1 (-1L), if left unspecified, which uses all available cores, with a minimum of two.

R-side Parallel Processing – Setting the Number of CPUs

The package also implements R-side parallel processing by replacing the R function `lapply` with `mclapply` found in the **parallel** package. You can set the number of cores accessed by `mclapply` by issuing the command `options(mc.cores = x)`, where `x` is the number of cores. The options command can also be placed in the users .Rprofile file for convenience. You can, alternatively, initialize the environment variable `MC_CORES` in your shell environment. See the help files in **parallel** for more information.

The default value for `mclapply` on non-Windows systems is two (2L) cores. On Windows systems, the default value is one (1L) core.

Example: Setting the Number of CPUs

As an example, issuing the following options command uses all available cores for both OpenMP and R-side processing:

```
options(rf.cores=detectCores(), mc.cores=detectCores())
```

As stated above, this option command can be placed in the users .Rprofile file.

CAUTIONARY NOTE

Regarding C-side threading (accessed via OpenMP compilation) versus R-side forking (accessed via mclapply in package **parallel**).

1. Once the package has been compiled with OpenMP enabled, trees will be grown in parallel using the rf.cores option. Independently of this, we also utilize mclapply to parallelize loops in R-side pre-processing and post-processing of the forest. This is always available and independent of whether the user chooses to compile the package with the OpenMP option enabled.
2. It is important NOT to write programs that fork R processes containing OpenMP threads. That is, one should not use mclapply around the functions rfsrc, predict.rfsrc, vimp.rfsc, var.select.rfsrc, and find.interaction.rfsrc. In such a scenario, program execution is not guaranteed.
3. Note that options(rf.cores=0) disables C-side threading, and options(mc.cores=1) disables R-side forking. Therefore, setting options(rf.cores=0), is one means to wrap mclapply around the functions listed above in 2.

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. 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).
3. 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.
4. 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.

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

See Also

`find.interaction`, `impute.rfsrc`, `max.subtree`, `plot.competing.risk`, `plot.rfsrc`, `plot.survival`, `plot.variable`, `predict.rfsrc`, `print.rfsrc`, `rf2rfz`, `rfsrcSyn`, `stat.split` `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

```
## Not run: 
## ------------------------------------------------------------
## classification analysis
## ------------------------------------------------------------

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



## End(Not run)
```

`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, outcome.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", "na.random"),
  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.

`outcome.target` Character value multivariate families specifying the target outcome to be used. The default is to use the first coordinate.

`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` Parameter ignored.

`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

```
## Not run:
## ------------------------------------------------------------
## find interactions, survival setting
## ------------------------------------------------------------

data(pbc, package = "randomForestSRC")
pbc.obj <- rfsrc(Surv(days,status) ~ ., pbc, nsplit = 10)
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)
find.interaction(wihs.obj)
find.interaction(wihs.obj, method = "vimp")

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

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

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

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

## ------------------------------------------------------------
## interactions for 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)
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")

## End(Not run)
```

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

```
## Not run:
data(follic, package = "randomForestSRC")
follic.obj <- rfsrc(Surv(time, status) ~ ., follic, nsplit = 3, ntree = 100)

## End(Not run)
```

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")
```

impute.rfsrc *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, mtry = NULL,
  xvar.wt = NULL, nodesize = 1, splitrule = NULL, nsplit = 1,
  na.action = c("na.impute", "na.random"), nimpute = 2, mf.q, blocks,
  always.use = NULL, max.iter = 10, eps = 0.01, verbose = TRUE,
  do.trace = FALSE, ...)
```

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.

`data` Data frame containing the data to be imputed.

`ntree` Number of trees to grow.

`mtry` Number of variables randomly sampled at each split.

`nodesize` Minimum terminal node size.

`splitrule` Splitting rule used to grow trees.

`nsplit` Non-negative integer value used to specify random splitting.

`na.action` Missing value action. See details below.

`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`).

`mf.q` Fraction of variables (between 0 and 1) used as responses in multivariate missForest imputation. By default, multivariate missForest imputation is not performed if left unspecifed. Can be an integer, in which case this equals the number of multivariate responses.

`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.

`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.

`xvar.wt` Weights for selecting variables for splitting on.

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

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

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

`do.trace` Parameter ignored.

`...` 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. Regardless of what algorithm is selected, prior to splitting a node, if there is missing data for a variable, the 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 in the event the node is split on a variable with missing data. Imputed data is however not used to calculate the split-statistic, which uses non-missing data only. Users can select ‘`na.action="na.random"`’ for a cruder, but faster imputation. Unlike the default action, data is not imputed as the tree is grown, instead following the split to a node, data points with missing values on the variable used to split the node are randomly assigned to daughter nodes.
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.
4. If `mf.q` is specified, then a multivariate version of missForest imputation (Stekhoven and Buhlmann, 2012) is applied. A fraction `mf.q` of the variables are used as multivariate responses and split on the remaining variables using a multivariate composite splitting rule. Missing data for responses are imputed by prediction. This is repeated with a new set of variables used as responses (mutually exclusive to the previous), until all variables have been imputed. The entire process is repeated, and the algorithm is iterated until a convergence criteria is met (specified using options `max.iter` and `eps`). Using an integer value for `mf.q` is allowed, in which case a total of `mf.q` variables are used as multivariate responses.

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. (2015). Random forest missing data algorithms.

See Also

`rfsrc`

Examples

```
## Not run:
## ------------------------------------------------------------
## example of survival imputation
## ------------------------------------------------------------

#imputation using outcome splitting
data(pbc, package = "randomForestSRC")
pbc.d <- impute.rfsrc(Surv(days, status) ~ ., data = pbc, nsplit = 3)

#when no formula is given we default to unsupervised splitting
pbc2.d <- impute.rfsrc(data = pbc, nodesize = 1, nsplit = 10, nimpute = 5)

#random splitting can be reasonably good
pbc3.d <- impute.rfsrc(Surv(days, status) ~ ., data = pbc,
          splitrule = "random", nodesize = 1, nimpute = 5)
```

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

air.d <- impute.rfsrc(Ozone ~ ., data = airquality, nimpute = 5)
air2.d <- impute.rfsrc(data = airquality, nimpute = 5, nodesize = 1)
air3.d <- impute.rfsrc(Ozone ~ ., data = airquality, nimpute = 5,
           splitrule = "random", nodesize = 1)

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

data(pbc, package = "randomForestSRC")

## use 10 percent of variables as responses
## i.e. multivariate missForest
pbc.d <- impute.rfsrc(data = pbc, mf.q = .01, nodesize = 1)

## use 1 variable as the response
## i.e. original missForest algorithm
pbc.d <- impute.rfsrc(data = pbc, mf.q = 1, nodesize = 1)

## End(Not run)
```

---

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

```
## Not run:
## ------------------------------------------------------------
## 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

## End(Not run)
```

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

```
## Not run:
## ------------------------------------------------------------
## multivariate mixed forests
## lipids used as the multivariate y-responses
## ------------------------------------------------------------

## load the data
data(nutrigenomic, package = "randomForestSRC")

## nice wrapper for making multivariate formula
mvrfsrc.f <- function(ynames, dat) {
  as.formula(paste("Multivar(", paste(ynames, collapse = ","),paste(") ~ ."), sep = ""))
}

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


## ------------------------------------------------------------
## nice wrappers to extract standardized performance values
## works for all forests - including multivariate forests
## ------------------------------------------------------------
## pull the standardized error from a forest object
get.error <- function(obj) {
  100 * c(sapply(obj$yvar.names, function(nn) {
    o.coerce <- randomForestSRC:::coerce.multivariate(obj, nn)
    if (o.coerce$family == "class") {
      tail(o.coerce$err.rate[, 1], 1)
    }
    else {
      tail(o.coerce$err.rate, 1) / var(o.coerce$yvar, na.rm = TRUE)
    }
  }))
}

## pull the standardized VIMP from a forest object
get.vimp <- function(obj) {
  vimp <- 100 * do.call(cbind, lapply(obj$yvar.names, function(nn) {
    o.coerce <- randomForestSRC:::coerce.multivariate(obj, nn)
    if (o.coerce$family == "class") {
      o.coerce$importance[, 1]
    }
    else {
      o.coerce$importance / var(o.coerce$yvar, na.rm = TRUE)
    }
  }))
  colnames(vimp) <- obj$yvar.names
  vimp
}

## ------------------------------------------------------------
## 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.error(mv.obj)

## acquire standardized VIMP
svimp <- get.vimp(mv.obj)

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



## End(Not run)
```

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

```
## Not run:
data(pbc, package = "randomForestSRC")
pbc.obj <- rfsrc(Surv(days, status) ~ ., pbc, nsplit = 3)

## End(Not run)
```

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.

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

```
## Not run:
## ------------------------------------------------------------
## 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, nsplit = 10))
}

## End(Not run)
```

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, outcome.target = NULL,
  plots.one.page = TRUE, sorted = TRUE, verbose = TRUE,  ...)
```

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.

`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. Plot variable importance (VIMP) if 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.

See Also

`predict.rfsrc`, `rfsrc`

Examples

```
## Not run:
## --------------------------------------------------------------
## classification example
## --------------------------------------------------------------

iris.obj <- rfsrc(Species ~ ., data = iris)
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, nsplit = 10))
}
```

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

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

```
## Not run:
## 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, nsplit = 10)

# 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)


## End(Not run)
```

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) from a RF-SRC analysis. Users can select between marginal (unadjusted, but fast) and partial plots (adjusted, but slow).

Usage

```
## S3 method for class 'rfsrc'
plot.variable(x, xvar.names, which.class, outcome.target,
  time, surv.type = c("mort", "rel.freq", "surv", "years.lost", "cif", "chf"),
  partial = FALSE, 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`), (`rfsrc, predict`), or (`rfsrc, plot.variable`). See the examples below for illustration of the latter.
`xvar.names`	Names of the x-variables to be used.
`which.class`	For classification families, an integer or character value specifying the class to focus on (defaults to the first class). For competing risk families, 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.
`outcome.target`	Character value for multivariate families specifying the target outcome to be used. The default is to use the first coordinate.
`time`	For survival families, the time at which the predicted survival value is evaluated at (depends on `surv.type`).
`surv.type`	For survival families, specifies the predicted value. See details below.
`partial`	Should partial plots be used?
`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.
`...`	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 '`which.class`'.
3. For multivariate families, it is the predicted value of the outcome specified by '`outcome.target`' and if that is a classification outcome, by '`which.class`'.
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 '`which.class`'. 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.

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. To appear in *Biostatistics*.

See Also

rfsrc, rfsrcSyn, predict.rfsrc

Examples

```
## Not run:
## ------------------------------------------------------------
## survival/competing risk
## ------------------------------------------------------------

## survival
data(veteran, package = ”randomForestSRC”)
v.obj <- rfsrc(Surv(time,status)~., veteran, nsplit = 10, 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, which.class = 2)

## ------------------------------------------------------------
## regression
## ------------------------------------------------------------

## airquality
airq.obj <- rfsrc(Ozone ~ ., data = airquality)
plot.variable(airq.obj, partial = TRUE, smooth.lines = TRUE)

## 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, outcome.target = "mpg", partial = TRUE, nvar = 1)
plot.variable(mtcars.mreg, outcome.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, outcome.target = "cyl", which.class = 1, partial = TRUE, nvar = 1)
plot.variable(mtcars.mix, outcome.target = "cyl", which.class = 2, partial = TRUE, nvar = 1)



## End(Not run)
```

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, outcome.target = NULL,
  importance = c("permute", "random", "anti",
               "permute.ensemble", "random.ensemble", "anti.ensemble", "none"),
 na.action = c("na.omit", "na.impute", "na.random"), outcome = c("train", "test"),
  proximity = FALSE,



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

`outcome.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). See `rfsrc` for details. Only applies when the test data contains y-outcome values.

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

`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` Parameter ignored.

`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). The ‘`na.action="na.random"`’ implements a cruder, but faster missing data algorithm. 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"`’).

++++++++ 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 regression responses (if such values are present).

`clasOutput` List containing performance values for test categorical (factor) responses (if such values are present).

++++++++ 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 detailed definitions of 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)

## Not run:
## ------------------------------------------------------------
## 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), ], nsplit = 10)
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, nsplit = 10)

# 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, ],
        nsplit = 10)

# 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 reproducibilility 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, nsplit = 10)
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, nsplit = 10)

# 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
if (library("Hmisc", logical.return = TRUE))
{
  rsf.err <- rcorr.cens(rsf.pred, Surv(yvar[, 1], yvar[, 2]))[1]
  mix.err <- 1 - rcorr.cens(mix.pred, Surv(yvar[, 1], yvar[, 2]))[1]
  cat("RSF                :", rsf.err, "\n")
  cat("multivariate forest:", mix.err, "\n")
}

## End(Not run)
```

`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)
```

rf2rfz *Save RF-SRC in .rfz Compressed Format*

Description

rf2rfz saves a RF-SRC object as a .rfz compressed file that is readable by the **randomForestSRC** Java plugin that is capable of visualizing the trees in the forest.

Usage

```
rf2rfz(object, forestName = NULL, ...)
```

Arguments

object	An object of class (rfsrc, grow) or (rfsrc, forest). Requires 'forest=TRUE' in the original rfsrc call.
forestName	The desired prefix name for forest as a string.
...	Further arguments passed to or from other methods.

Details

An .rfz compressed file is actually a .zip file consisting of three files. The first is an ASCII file of type .txt containing the $nativeArray component of the forest. The second is an ASCII file of type .factor.txt containing the $nativefactorArray component of the forest. The third is an ASCII file of type .xml containing the PMML DataDictionary component.

PMML or the Predictive Model Markup Language is an XML based language which provides a way for applications to define statistical and data mining models and to share models between PMML compliant applications. More information about PMML and the Data Mining Group can be found at http:

The function rf2rfz is used to import the geometry of the forest to the RF-SRC Java plugin that is capable of visualizing the trees in the forest.

The geometry of the forest is saved as a file called forestName.rfz in the users working directory. This file can then be read by the **randomForestSRC** Java plugin.

Contact the authors on downloading the Java plugin.

Value

None.

Note

Contact the authors on downloading the Java plugin.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

http:

See Also

rfsrc

Examples

```
## Not run:
# Example 1:  Growing a forest, saving it as a \emph{.rfz} file ready
# for import into the Java plugin.

library("XML")

data(veteran, package = "randomForestSRC")
v.obj <- rfsrc(Surv(time, status)~., data = veteran)
rf2rfz(v.obj$forest, forestName = "veteran")

## End(Not run)
```

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

Description

A random forest (Breiman, 2001) is grown using user supplied training data. Applies when the response (outcome) is numeric, categorical (factor), or right-censored (including competing risk), and yields regression, classification, and survival forests, respectively. The resulting forest, informally referred to as a 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.

Note that the package now handles multivariate regression and classification responses as well as mixed outcomes (regression/classification responses). In such cases, a multivariate forest is grown, informally referred to as an mRF-SRC object. Unsupervised forests are also available.

The package implements OpenMP shared-memory parallel programming. However, the default installation will only execute serially. Users should consult the randomForestSRC-package help file for details on installing the OpenMP version of the package. The help file is readily accessible via the command `package?randomForestSRC`.

Usage

```
rfsrc(formula, data, ntree = 1000,
  bootstrap = c("by.root", "by.node", "none"),
  mtry = NULL,
  nodesize = NULL,
  nodedepth = NULL,
  splitrule = NULL,
  nsplit = 0,
  split.null = FALSE,
  importance = c("permute", "random", "anti", "permute.ensemble",
               "random.ensemble", "anti.ensemble", "none"),
  na.action = c("na.omit", "na.impute", "na.random"),
  nimpute = 1,
  ntime,
  cause,
  proximity = FALSE,



  xvar.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,
  membership = TRUE,
  statistics = FALSE,
  fast.restore = FALSE,
  coerce.factor = NULL,
  ...)
```

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.

`bootstrap` Bootstrap protocol. The default is `by.root` which bootstraps the data by sampling with replacement at the root node before growing the tree. 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. See the details below on prediction error when the default choice is not in effect.

`mtry` Number of variables randomly selected as candidates for each node split. The default is sqrt(p), except for regression families where p/3 is used, where p equals the number of variables. Values are rounded up.

`nodesize` Minimum number of unique cases (data points) in a terminal node. The defaults are: survival (3), competing risk (6), regression (5), classification (1), mixed outcomes (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, 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 can significantly increase speed over deterministic splitting. In the case of pure random splitting, a value of one is used as the default, since deterministic splitting is not a compatible concept in that scenario. However, values greater than one are accepted, as with the other split rules.

`split.null` Set this value to TRUE when testing the null hypothesis. In particular, this assumes there is no relationship between y and x.

`importance` Method for computing variable importance (VIMP). Calculating VIMP can be computationally expensive when the number of variables is high, thus if VIMP is not needed consider setting `importance="none"`. See the details below for more about VIMP.

`na.action` Action taken if the data contains `NA`'s. Possible values are `na.omit`, `na.impute` or `na.random`. 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` or `na.random` 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 families to constrain ensemble calculations to a grid of time values of no more than `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. Use this option when the sample size is large to improve computational efficiency.

`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".

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.

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 and by.tree.

split.depth
Records the minimal depth for each variable. Default is FALSE. Possible values are all.trees and by.tree. Used for variable selection.

seed
Negative integer specifying seed for the random number generator.

do.trace
Parameter ignored.

membership
Should terminal node membership and inbag information be returned?

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

fast.restore
Should forest restoration be fast? Be aware that this option uses significantly more RAM and may not be suitable for all platforms depending on the size of the data set being analyzed.

coerce.factor
Names of variables (can either be x-variables or y-outcomes) to be analyzed as factors. The variables must be coded as sequential positive integers 1,2,... This option is useful in high-dimensional problems with a large number of factors.

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

Details

1. *Families*

   The package automagically determines the underlying random forest family to be fit from the following eight choices:

   regr, regr+, class, class+, mix+, unsupv, surv, and surv-CR.

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

   See below for how to code the response in the two different survival scenarios and for responses for multivariate forests.

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.
- 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.
- Unsupervised analysis: In settings where there is no outcome, unsupervised splitting is invoked. In this case, the mixed outcome splitting rule (above) is applied.
- 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.

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 factors, splits are similar to real valued variables. If the 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*
   - *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 considerably improve computational speed, a randomize version of a splitting rule can be invoked using '`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. The splitting rule is applied to the random split points and the node is split on that variable and random split point yielding the best value (as measured by the splitting rule). Pure random splitting can be invoked by setting '`splitrule="random"`'. For each node, a variable is randomly selected and the node is split using a random split point (Cutler and Zhao, 2001; Lin and Jeon, 2006).
     The value of '`nsplit`' has a significant impact on the time taken to grow a forest. This is because non-random splitting (i.e. the default option '`nsplit=0`'), considers all possible split points for each of the '`mtry`' variables, and iterating over each split point can be CPU intensive, especially in large sample size settings.
     In general, regardless of computational speed, it is good practice to use the '`nsplit`' when the data contains a mix of continuous and discrete variables. Using a reasonably small value mitigates bias and may not compromise accuracy.
   - *Large sample size*
     For increased efficiency for survival families, users should consider setting '`ntime`' to a relatively small value when the sample size (number of rows of the data) is large. This constrains ensemble calculations such as survival functions to a restricted grid of time points of length no more than '`ntime`' which can considerably reduce computational times.
   - *Large number of variables*
     For increased efficiency when the number of variables are large, set `importance="none"` which turns off variable importance (VIMP) calculations (see below for more details about variable importance) and can considerably reduce computational times. Note that variable importance calculations can always be recovered later from the grow forest using the function `vimp` and/or `predict`.
   - *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.
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)*

   The option '`importance`' allows several ways to calculate VIMP. The default `permute` returns Breiman-Cutler permutation VIMP as described in Breiman (2001). For each tree, the prediction error on the out-of-bag (OOB) data is recorded. Then for a given variable *x*, OOB cases are randomly permuted in *x* and the prediction error is recorded. The VIMP for *x* is defined as the difference between the perturbed and unperturbed error rate, averaged over all trees. If `random` is used, then *x* is not permuted, but rather an OOB case is assigned a daughter node randomly whenever a split on *x* is encountered in the in-bag tree. If `anti` is used, then *x* is assigned to the opposite node whenever a split on *x* is encountered in the in-bag tree.

   When '`importance`' is selected from one of the '`ensemble`' choices, then VIMP is calculated by comparing the error rate for the perturbed OOB forest ensemble to the unperturbed OOB forest ensemble, where the perturbed ensemble is obtained by either permuting *x*, or by random daughter node assignment, or by anti-splitting on *x*. Thus, unlike the Breiman-Cutler measure, here VIMP does not measure the tree average effect of *x*, but rather its overall forest effect. See Ishwaran et al. (2008) for further details.

   Finally, 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 can take one of two forms:

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

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

   The nature of the outcomes will inform the code as to what type of multivariate forest to grow; i.e. whether it is real-valued, categorical, or a combination of both (mixed). Note that performance measures such as VIMP and error rates are returned for all outcomes. For faster speed, VIMP can be turned off and the `predict` function used later to extract these values.

8. *Unsupervised Splitting*

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

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

   rfsrc(data = my.data)

   In this case, a random subset of `mtry` pairs of variables are selected at each tree node. For each pair, one is chosen at random to be the response (called the pseudo-response) and the remaining variable is treated as the candidate variable to be split on. The best split (measured in terms of the pseudo-response) over the `mtry` pairs is used to split the node.

   Multivariate unsupervised splitting can also be implemented. A typical call looks like:

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

In the above, `mtry=10` variables are selected at random, and for each of these a random subset of 5 variables are selected and defined as the multivariate pseudo-responses. In essence this creates a collection of 10 multivariate regression problems (the number of regressions, 10, is defined by `mtry` and the number of pseudo-responses in each regression, 5, is determined by the unsupervised formula; informally we call this value "ytry"). A multivariate normalized composite splitting rule is then applied to each of the 10 multivariate regression problems and the node 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). Prior to splitting a node, missing data for a variable is imputed by randomly drawing values from non-missing in-bag data. The purpose of this imputed data is to make it possible to assign cases to daughter nodes in the event the node is split on a variable with missing data. Imputed data is however not used to calculate the split-statistic which uses non-missing data only. 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 OOB 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.

    Choosing the option '`na.action="na.random"`' implements a cruder version of the missing data algorithm, but which should be computationally faster. Unlike the default method, data is not imputed as the tree is grown, instead following the split to a node, data points with missing values on the variable used to split the node are randomly assigned to daughter nodes. When terminal nodes are reached, missing data are imputed as before by using out-of-bag (OOB) non-missing terminal node data.

Both missing data algorithms 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.

Regardless of what method is used, 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.rfsrc` for a fast impute interface.

11. *Miscellanea*
    (a) Setting '`var.used="all.trees"`' returns a vector of size `p` where each element is a count of the number of times a split occurred on a variable. If '`var.used="by.tree"`', a matrix of size `ntreexp` is returned. Each element [i][j] is the count of the number of times a split occurred on variable [j] in tree [i].
    (b) Setting '`split.depth="all.trees"`' returns a matrix of size `nxp` where entry [i][j] is the minimal depth for variable [j] for case [i] averaged over the forest. That is, for case [i], the entry [i][j] records the first time case [i] splits on variable [j] averaged over the forest. If '`split.depth="by.tree"`', a three-dimensional array is returned where the third dimension [k] records the tree and the first two coordinates [i][j] record the case and the variable. Thus entry [i][j][k] is the minimal depth for case [i] for variable [j] for tree [k].

Value

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

`call`	The original call to `rfsrc`.
`formula`	The formula used in the call.
`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.

`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.

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

`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 regression responses (if they are present).

`clasOutput` List containing performance values for categorical (factor) responses (if they are present).

`++++++++` 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

1. Values returned depend heavily on the family. In particular, `predicted` and `predicted.oob` are the following values calculated using in-bag and OOB data:
   (a) For regression, a vector of predicted y-responses.
   (b) For classification, a matrix with columns containing the estimated class probability for each class.
   (c) 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 included in the grow object 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`.
   (d) 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). The grow object also contains 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).
   (e) For multivariate families, predicted values (and other performance values such as VIMP and error rates) are stored in the lists `regrOutput` and `clasOutput`.
2. Different R-wrappers are provided to aid in parsing the grow object.

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

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.

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.

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

See Also

`find.interaction`, `impute.rfsrc`, `max.subtree`, `plot.competing.risk`, `plot.rfsrc`, `plot.survival`, `plot.variable`, `predict.rfsrc`, `print.rfsrc`, `rf2rfz`, `rfsrcSyn`, `stat.split`, `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)

# 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[1:10, ]),
    xlab = "Time", ylab = "Survival", type = "l", lty = 1)

# plot survival curves for first 10 individuals
# indirect way: using plot.survival (also generates hazard plots)
plot.survival(v.obj, subset = 1:10, haz.model = "ggamma")

## Not run:
## Primary biliary cirrhosis (PBC) of the liver

data(pbc, package = "randomForestSRC")
pbc.obj <- rfsrc(Surv(days, status) ~ ., pbc, nsplit = 10)
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")


# here's a nice wrapper to combine original data + imputed data
combine.impute <- function(object) {
 impData <- cbind(object$yvar, object$xvar)
 if (!is.null(object$imputed.indv)) {
   impData[object$imputed.indv, ] <- object$imputed.data
 }
 impData
}

# combine original data + imputed data
pbc.imp.data <- combine.impute(pbc.obj2)

# same as above but we iterate the missing data algorithm
pbc.obj3 <- rfsrc(Surv(days, status) ~ ., pbc, nsplit=10,
         na.action = "na.impute", nimpute = 3)
pbc.iterate.imp.data <- combine.impute(pbc.obj3)

# fast way to impute the data (no inference is done)
# see impute.rfsc for more details
pbc.fast.imp.data <- impute.rfsrc(data = pbc, nsplit = 10, nimpute = 5)

##------------------------------------------------------------
## 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)
    & library("Hmisc", 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)
  rfsrc.obj <- rfsrc(surv.f, pbc.na, nsplit = 10, 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, nsplit = 10)
  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.list(cox.obj)) {
      rcorr.cens(predict(cox.obj, pbc.na[-train, ]),
                 Surv(pbc.na$days[-train],
                 pbc.na$status[-train]))[1]
    } 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$cif
Time <- wihs.obj$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, nsplit = 10)

  ## log-rank event-one specific splitting
  pbc.log1 <- rfsrc(Surv(time, status) ~ ., pbc, nsplit = 10,
              splitrule = "logrank", cause = c(1,0))

  ## log-rank event-two specific splitting
  pbc.log2 <- rfsrc(Surv(time, status) ~ ., pbc, nsplit = 10,
              splitrule = "logrank", cause = c(0,1))
  ## 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)

# minimal depth variable selection via max.subtree
md.obj <- max.subtree(mtcars.obj)
cat("top variables:\n")
print(md.obj$topvars)

# equivalent way to select variables
# see var.select for more details
vs.obj <- var.select(object = mtcars.obj)


## ------------------------------------------------------------
## 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, nsplit = 10)
plot(breast.obj)

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

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

```
#minimal depth variable selection applies!
var.select(mtcars2.unspv)

## ------------------------------------------------------------
## Multivariate regression analysis
## ------------------------------------------------------------
mtcars.mreg <- rfsrc(Multivar(mpg, cyl) ~., data = mtcars)
print(mtcars.mreg, outcome.target = "mpg")
print(mtcars.mreg, outcome.target = "cyl")
plot(mtcars.mreg, outcome.target = "mpg")
plot(mtcars.mreg, outcome.target = "cyl")


## ------------------------------------------------------------
## 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)
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")
## Edgar Anderson's iris data
iris.obj <- rfsrc(Species ~., data = iris, splitrule="custom1")
## WIHS analysis
wihs.obj <- rfsrc(Surv(time, status) ~ ., wihs, nsplit = 3,
                  ntree = 100, splitrule="custom1")



## End(Not run)
```

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

`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,
  mtrySeq = NULL,
  nodesize = 5,
  nodesizeSeq = c(1:10,20,30,50,100),
  nsplit = 0,
  min.node = 3,
  use.org.features = TRUE,
  na.action = c("na.omit", "na.impute"),
  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 synthetic forest.

`mtrySeq` Sequence of mtry values used for fitting the collection of RF machines. If `NULL`, set to the default value p/3.

`nodesize` Nodesize value for the synthetic forest.

`nodesizeSeq` Sequence of nodesize values used for the fitting the collection of RF machines.

`nsplit` If non-zero, nsplit-randomized splitting is used which can significantly increase speed.

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

`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`.

`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 when fitting the synthetic forest). Currently only implemented for regression and classification settings (univariate and multivariate).

Note that synthetic features are constructed using out-of-bag (OOB) data in order to avoid double dipping into training data. Nevertheless, the internal OOB error rate for the synthetic forest will be biased and thus cross-validation must be used for determining performance.

If `mtrySeq` is set, RF machines are constructed for each combination of nodesize and mtry values specified by `nodesizeSeq` `mtrySeq`. However, a sequence of values for `mtrySeq` generally does not work as well as using a fixed value. Generally, performance gains are observed when one of the two sequences is fixed: `mtrySeq` is fixed and `nodesizeSeq` is varied, or `nodesizeSeq` is fixed and `mtrySeq` is varied. However, see the examples below where this is not the case.

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, impute.rfsrc

Examples

```
## Not run:
## -------------------------------------------------------------
## 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 ~., data = ring)

  ## synthetic forests:
  ## 1 = nodesize varied
  ## 2 = nodesize/mtry varied
  ringSyn1 <- rfsrcSyn(classes ~., data = ring)
  ringSyn2 <- rfsrcSyn(classes ~., data = 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 ~ ., data = dat[-test, ], )
pred.regF <- predict(regF, dat[test, ], importance = "none")

## synthetic forests
## we pass both the training and testing data
## but this can be split into separate commands as in the
## previous classification example
synF1 <- rfsrcSyn(y ~ ., data = dat[-test, ],
  newdata = dat[test, ])
synF2 <- rfsrcSyn(y ~ ., data = dat[-test, ],
  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)
mvSyn <- rfsrcSyn(cbind(carb, mpg, cyl) ~., data = mtcars.new)

## End(Not run)
```

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

```
## Not run:
## run a forest, then make a call to stat.split
grow.obj <- rfsrc(mpg ~., data = mtcars, 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”)

## End(Not run)
```

---

var.select *Variable Selection*

Description

Variable selection using minimal depth.

Usage

```
## S3 method for class 'rfsrc'
var.select(formula, data, object, cause, outcome.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", "na.random"),
  always.use = NULL, nrep = 50, K = 5, nstep = 1,
  prefit =  list(action = (method != "md"), ntree = 100,
  mtry = 500, nodesize = 3, nsplit = 1),
  do.trace = 0, 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.

`outcome.target` Integer value for multivariate families specifying the target outcome to be used when VIMP is utilized (see the option '`method`' below for more details). A vector of coordinate values is allowed which induces a composite average. The default is to use a composite average of all coordinates.

`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` Minimum number of unique cases in a terminal node.

`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.

`do.trace` Parameter ignored.

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

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

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

```
## Not run:
## ------------------------------------------------------------
## Minimal depth variable selection
## survival analysis
## ------------------------------------------------------------

data(pbc, package = "randomForestSRC")
pbc.obj <- rfsrc(Surv(days, status) ~ ., pbc, nsplit = 10)

# 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
## ------------------------------------------------------------

## competing risk data set involving AIDS in women
data(wihs, package = "randomForestSRC")
vs.wihs <- var.select(Surv(time, status) ~ ., wihs, nsplit = 3, ntree = 100)
```

```
## 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, nsplit = 10, cause = 1)
  var.select(Surv(time, status) ~ ., pbc, nsplit = 10, cause = 2)
}

## ------------------------------------------------------------
## Minimal depth variable selection
## classification analysis
## ------------------------------------------------------------

vs.iris <- var.select(Species ~ ., iris)

## ------------------------------------------------------------
## Minimal depth variable selection
## Regression analysis
## ------------------------------------------------------------

#Variable hunting (overkill for low dimensions)
vh.air <- var.select(Ozone ~., airquality, method = "vh", nrep = 10, mvars = 5)

#better analysis
vs.air <- var.select(Ozone ~., airquality)

## ------------------------------------------------------------
## Minimal depth high-dimensional example
## van de Vijver microarray breast cancer survival data
## predefined weights for *selecting* a gene for node splitting
## determined from a preliminary forest analysis
## ------------------------------------------------------------

data(vdv, package = "randomForestSRC")
md.breast <- var.select(Surv(Time, Censoring) ~ ., vdv,
  prefit = list(action = TRUE))

## same analysis, but with customization for the preliminary forest fit
## note the large mtry and small nodesize values used
md.breast.custom <- var.select(Surv(Time, Censoring) ~ ., vdv,
  prefit = list(action = TRUE, mtry = 500, nodesize = 1))

## ------------------------------------------------------------
## Minimal depth high-dimensional example
## van de Vijver microarray breast cancer survival data
## predefined weights for genes for *splitting* tree nodes
## weights defined in terms of cox p-values
## ------------------------------------------------------------

if (library("survival", logical.return = TRUE)
    & library("Hmisc", 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)
    unlist(mclapply(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)
  breast.obj <- rfsrc(rfsrc.f, vdv, nsplit = 10, xvar.wt = cox.wts)
  md.breast.splitwt <- var.select(object = breast.obj)
}


## ------------------------------------------------------------
## 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)
  & library("Hmisc", 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)
}


## End(Not run)
```

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, outcome.target=NULL,
  importance = c("permute", "random", "anti",
                 "permute.ensemble", "random.ensemble", "anti.ensemble", "none"),
  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.
`outcome.target`	Character vector for multivariate families specifying the target outcomes to be used. The default is to use all coordinates.
`importance`	Type of 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`	Parameter ignored.
`...`	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`. Depending upon the option `importance`, VIMP is calculated either by random daughter assignment or by random permutation of the variable(s). The default is Breiman-Cutler permutation VIMP. See `rfsrc` for more details.

Joint VIMP is requested using '`joint`'. The joint VIMP is the importance for a group of variables when the group is perturbed simultaneously.

Value

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

`err.rate` OOB error rate for the ensemble restricted to the subsetted data.

`importance` Variable importance (VIMP).

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

```
## Not run:
## ------------------------------------------------------------
## classification example
## showcase different vimp
## ------------------------------------------------------------

iris.obj <- rfsrc(Species ~ ., data = iris)

# Breiman-Cutler permutation vimp
print(vimp(iris.obj)$importance)

# Breiman-Cutler random daughter vimp
print(vimp(iris.obj, importance = "random")$importance)

# Breiman-Cutler joint permutation vimp
print(vimp(iris.obj, joint = TRUE)$importance)

# Breiman-Cuter 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
## compare Breiman-Cutler vimp to ensemble based vimp
## ------------------------------------------------------------

airq.obj <- rfsrc(Ozone ~ ., airquality)
vimp.all <- cbind(
     ensemble = vimp(airq.obj, importance = "permute.ensemble")$importance,
     breimanCutler = vimp(airq.obj, importance = "permute")$importance)
print(vimp.all)


## ------------------------------------------------------------
## regression example
## calculate VIMP 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))

## End(Not run)
```

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

```
## Not run:
data(wihs, package = "randomForestSRC")
wihs.obj <- rfsrc(Surv(time, status) ~ ., wihs, nsplit = 3, ntree = 100)

## End(Not run)
```

Index