Package 'randomForestSRC'

January 16, 2025

Version 3.3.3

Date 2025-01-15

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

Author Hemant Ishwaran [aut], Udaya B. Kogalur [aut, cre]

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

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

Depends R (>= 4.3.0),

Imports parallel, data.tree, DiagrammeR

Suggests survival, pec, prodlim, mlbench, interp, caret, imbalance, cluster, fst, data.table

Description Fast OpenMP parallel computing of Breiman's random forests for univariate, multivariate, unsupervised, survival, competing risks, class imbalanced classification and quantile regression. New Mahalanobis splitting for correlated outcomes. Extreme random forests and randomized splitting. Suite of imputation methods for missing data. Fast random forests using subsampling. Confidence regions and standard errors for variable importance. New improved hold-out importance. Case-specific importance. Minimal depth variable importance. Visualize trees on your Safari or Google Chrome browser. Anonymous random forests for data privacy.

License GPL (>= 3)

URL https://www.randomforestsrc.org/ https://ishwaran.org/

NeedsCompilation yes

Repository CRAN

Date/Publication 2025-01-16 09:20:02 UTC

Contents

`randomForestSRC-package`

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

Description

Fast OpenMP parallel computing of Breiman random forests (Breiman 2001) for regression, classification, survival analysis (Ishwaran 2008), competing risks (Ishwaran 2012), multivariate (Segal and Xiao 2011), unsupervised (Mantero and Ishwaran 2020), quantile regression (Meinhausen 2006, Zhang et al. 2019, Greenwald-Khanna 2001), and class imbalanced q-classification (O'Brien and Ishwaran 2019). Different splitting rules invoked under deterministic or random splitting (Geurts et al. 2006, Ishwaran 2015) are available for all families. Variable importance (VIMP), and holdout VIMP, as well as confidence regions (Ishwaran and Lu 2019) can be calculated for single and grouped variables. Minimal depth variable selection (Ishwaran et al. 2010, 2011). Fast interface for missing data imputation using a variety of different random forest methods (Tang and Ishwaran 2017). Visualize trees on your Safari or Google Chrome browser (works for all families, see `get.tree`).

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. `rfsrc.fast`

   A fast implementation of `rfsrc` using subsampling.

3. `quantreg.rfsrc`, `quantreg`

   Univariate and multivariate quantile regression forest for training and testing. Different methods available including the Greenwald-Khanna (2001) algorithm, which is especially suitable for big data due to its high memory efficiency.

4. `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)`.

5. `sidClustering.rfsrc`, `sidClustering`

   Clustering of unsupervised data using SID (Staggered Interaction Data). Also implements the artificial two-class approach of Breiman (2003).

6. `vimp`, `subsample`, `holdout.vimp`

   Used for variable selection:

   (a) `vimp` calculates variable imporance (VIMP) from a RF-SRC grow/predict object by noising up the variable (for example by permutation). Note that grow/predict calls can always directly request VIMP.

   (b) `subsample` calculates VIMP confidence itervals via subsampling.

   (c) `holdout.vimp` measures the importance of a variable when it is removed from the model.

7. `imbalanced.rfsrc`, `imbalanced`

   q-classification and G-mean VIMP for class imbalanced data.

8. `impute.rfsrc`, `impute`

   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.

9. `partial.rfsrc`, `partial`

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

Home page, Vignettes, Discussions, Bug Reporting, Source Code, Beta Builds

1. The home page for the package, containing vignettes, manuals, links to GitHub and other useful information is found at https://www.randomforestsrc.org/index.html
2. Questions, comments, and non-bug related issues may be sent via https://github.com/kogalur/randomForestSRC/discussions/.
3. Bugs may be reported via https://github.com/kogalur/randomForestSRC/issues/. This is for bugs only. Please provide the accompanying information with any reports:
   (a) `sessionInfo()`
   (b) A minimal reproducible example consisting of the following items:
   - a minimal dataset, necessary to reproduce the error
   - the minimal runnable code necessary to reproduce the error, which can be run on the given dataset
   - the necessary information on the used packages, R version and system it is run on
   - in the case of random processes, a seed (set by `set.seed()`) for reproducibility
4. Regular stable releases of this package are available on CRAN at https://cran.r-project.org/package=randomForestSRC/
5. Interim unstable development builds with bug fixes and sometimes additional functionality are available at https://github.com/kogalur/randomForestSRC/

OpenMP Parallel Processing – Installation

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

Additional instructions for configuring OpenMP parallel processing are available at https://www.randomforestsrc.org/articles/installation.html.

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

Reproducibility

With respect to reproducibility, a model is defined by a seed, the topology of the trees in the forest, and terminal node membership of the training data. This allows the user to restore a model and, in particular, its terminal node statistics. On the other hand, VIMP and many other statistics are dependent on additional randomization, which we do not consider part of the model. These statistics are susceptible to Monte Carlo effects.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

Geurts, P., Ernst, D. and Wehenkel, L., (2006). Extremely randomized trees. *Machine learning*, 63(1):3-42.

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

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

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

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

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

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

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

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

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

Ishwaran H. and Lu M. (2019). Standard errors and confidence intervals for variable importance in random forest regression, classification, and survival. *Statistics in Medicine*, 38, 558-582.

Lu M., Sadiq S., Feaster D.J. and Ishwaran H. (2018). Estimating individual treatment effect in observational data using random forest methods. *J. Comp. Graph. Statist*, 27(1), 209-219

Mantero A. and Ishwaran H. (2021). Unsupervised random forests. *Statistical Analysis and Data Mining*, 14(2):144-167.

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

O'Brien R. and Ishwaran H. (2019). A random forests quantile classifier for class imbalanced data. *Pattern Recognition*, 90, 232-249

Segal M.R. and Xiao Y. Multivariate random forests. (2011). *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*. 1(1):80-87.

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

Zhang H., Zimmerman J., Nettleton D. and Nordman D.J. (2019). Random forest prediction intervals. *The American Statistician*. 4:1-5.

See Also

find.interaction.rfsrc,

get.tree.rfsrc,

holdout.vimp.rfsrc,

imbalanced.rfsrc, impute.rfsrc,

max.subtree.rfsrc,

partial.rfsrc, plot.competing.risk.rfsrc, plot.rfsrc, plot.survival.rfsrc, plot.variable.rfsrc, predict.rfsrc, print.rfsrc,

quantreg.rfsrc,

rfsrc, rfsrc.cart, rfsrc.fast,

sidClustering.rfsrc,

stat.split.rfsrc, subsample.rfsrc, synthetic.rfsrc,

tune.rfsrc,

var.select.rfsrc, vimp.rfsrc

---

breast *Wisconsin Prognostic Breast Cancer Data*

Description

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

Source

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

Examples

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

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

find.interaction.rfsrc

Find Interactions Between Pairs of Variables

Description

Find pairwise interactions between variables.

Usage

```
## S3 method for class 'rfsrc'
find.interaction(object, xvar.names, cause, m.target,
  importance = c("permute", "random", "anti",
                 "permute.ensemble", "random.ensemble", "anti.ensemble"),
  method = c("maxsubtree", "vimp"), sorted = TRUE, nvar, nrep = 1,
  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.
m.target	Character value for multivariate families specifying the target outcome to be used. If left unspecified, the algorithm will choose a default target.
importance	Type of variable importance (VIMP). See rfsrc for details.
method	Method of analysis: maximal subtree or VIMP. See details below.
sorted	Should variables be sorted by VIMP? Does not apply for competing risks.
nvar	Number of variables to be used.
nrep	Number of Monte Carlo replicates when 'method="vimp"'.
na.action	Action to be taken if the data contains NA values. Applies only when 'method="vimp"'.
seed	Seed for random number generator. Must be a negative integer.
do.trace	Number of seconds between updates to the user on approximate time to completion.
verbose	Set to TRUE for verbose output.
...	Further arguments passed to or from other methods.

Details

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

1. 'method="maxsubtree"'

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

2. 'method="vimp"'

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

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

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

Value

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

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

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

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

See Also

holdout.vimp.rfsrc, max.subtree.rfsrc, var.select.rfsrc, vimp.rfsrc

Examples

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

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

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

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

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

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

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

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

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

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

`follic` *Follicular Cell Lymphoma*

Description

Competing risk data set involving follicular cell lymphoma.

Format

A data frame containing:

age	age
hgb	hemoglobin (g/l)
clinstg	clinical stage: 1=stage I, 2=stage II
ch	chemotherapy
rt	radiotherapy
time	first failure time
status	censoring status: 0=censored, 1=relapse, 2=death

Source

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

References

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

Examples

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

---

get.tree.rfsrc *Extract a Single Tree from a Forest and plot it on your browser*

Description

Extracts a single tree from a forest which can then be plotted on the users browser. Works for all families. Missing data not permitted.

Usage

```
## S3 method for class 'rfsrc'
get.tree(object, tree.id, target, m.target = NULL,
   time, surv.type = c("mort", "rel.freq", "surv", "years.lost", "cif", "chf"),
   class.type = c("bayes", "rfq", "prob"),
   ensemble = FALSE, oob = TRUE, show.plots = TRUE, do.trace = FALSE)
```

Arguments

object	An object of class (rfsrc, grow).
tree.id	Integer value specifying the tree to be extracted.
target	For classification, an integer or character value specifying the class to focus on (defaults to the first class). For competing risks, an integer value between 1 and J indicating the event of interest, where J is the number of event types. The default is to use the first event type.
m.target	Character value for multivariate families specifying the target outcome to be used. If left unspecified, the algorithm will choose a default target.
time	For survival, the time at which the predicted survival value is evaluated at (depends on surv.type).
surv.type	For survival, specifies the predicted value. See details below.
class.type	For classification, specifies the predicted value. See details below.
ensemble	Use the ensemble (of all trees) for prediction, or use the requested tree for prediction (this is the default).
oob	OOB (TRUE) or in-bag (FALSE) predicted values. Only applies when ensemble=TRUE.
show.plots	Should plots be displayed?
do.trace	Number of seconds between updates to the user on approximate time to completion.

Details

Extracts a specified tree from a forest and converts the tree to a hierarchical structure suitable for use with the "data.tree" package. Plotting the object will conveniently render the tree on the users browser. Left tree splits are displayed. For continuous values, left split is displayed as an inequality with right split equal to the reversed inequality. For factors, split values are described in terms of the levels of the factor. In this case, the left daughter split is a set consisting of all levels that are assigned to the left daughter node. The right daughter split is the complement of this set.

Terminal nodes are highlighted by color and display the sample size and predicted value. By default, predicted value equals the tree predicted value and sample size are terminal node inbag sample sizes. If ensemble=TRUE, then the predicted value equals the forest ensemble value which could be useful as it allows one to visualize the ensemble predictor over a given tree and therefore for a given partition of the feature space. In this case, sample sizes are for all cases and not the tree specific inbag cases.

The predicted value displayed is as follows:

1. For regression, the mean of the response.
2. For classification, for the target class specified by 'target', either the class with most votes if class.type="bayes"; or in a two-class problem the classifier using the RFQ quantile threshold if class.type="bayes" (see imbalanced for more details); or the relative class frequency when class.type="prob".
3. For multivariate families, the predicted value of the outcome specified by 'm.target'. This being the value for regression or classification described above, depending on whether the outcome is real valued or a factor.

4. For survival, the choices are:
   - Mortality (`mort`).
   - Relative frequency of mortality (`rel.freq`).
   - Predicted survival (`surv`), where the predicted survival is for the time point specified using `time` (the default is the median follow up time).
5. For competing risks, the choices are:
   - The expected number of life years lost (`years.lost`).
   - The cumulative incidence function (`cif`).
   - The cumulative hazard function (`chf`).

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

Value

Invisibly, returns an object with hierarchical structure formatted for use with the data.tree package.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

Many thanks to @dbarg1 on GitHub for the initial prototype of this function

Examples

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

## survival - veteran data set but with factors
## note that diagtime has many levels
data(veteran, package = "randomForestSRC")
vd <- veteran
vd$celltype=factor(vd$celltype)
vd$diagtime=factor(vd$diagtime)
vd.obj <- rfsrc(Surv(time,status)~., vd, ntree = 100, nodesize = 5)
plot(get.tree(vd.obj, 3))

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

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

airq.obj <- rfsrc(Ozone ~ ., data = airquality)
plot(get.tree(airq.obj, 10))

## ------------------------------------------------------------
```

```
## two-class imbalanced data (see imbalanced function)
## ------------------------------------------------------------

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

## compare RFQ to Bayes Rule
plot(get.tree(breast.obj, 1, class.type = "rfq", ensemble = TRUE))
plot(get.tree(breast.obj, 1, class.type = "bayes", ensemble = TRUE))

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

iris.obj <- rfsrc(Species ~., data = iris, nodesize = 10)

## equivalent
plot(get.tree(iris.obj, 25))
plot(get.tree(iris.obj, 25, class.type = "bayes"))

## predicted probability displayed for terminal nodes
plot(get.tree(iris.obj, 25, class.type = "prob", target = "setosa"))
plot(get.tree(iris.obj, 25, class.type = "prob", target = "versicolor"))
plot(get.tree(iris.obj, 25, class.type = "prob", target = "virginica"))


## ------------------------------------------------------------
## multivariate regression
## ------------------------------------------------------------

mtcars.mreg <- rfsrc(Multivar(mpg, cyl) ~., data = mtcars)
plot(get.tree(mtcars.mreg, 10, m.target = "mpg"))
plot(get.tree(mtcars.mreg, 10, m.target = "cyl"))


## ------------------------------------------------------------
## 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(get.tree(mtcars.mix, 5, m.target = "cyl"))
plot(get.tree(mtcars.mix, 5, m.target = "carb"))

## ------------------------------------------------------------
## unsupervised analysis
## ------------------------------------------------------------

mtcars.unspv <- rfsrc(data = mtcars)
```

```
plot(get.tree(mtcars.unspv, 5))
```

---

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")
```

`holdout.vimp.rfsrc` *Hold out variable importance (VIMP)*

Description

Hold out VIMP is calculated from the error rate of mini ensembles of trees (blocks of trees) grown with and without a variable. Applies to all families.

Usage

```
## S3 method for class 'rfsrc'
holdout.vimp(formula, data,
  ntree = function(p, vtry){1000 * p / vtry},
  nsplit = 10,
  ntime = 50,
  sampsize = function(x){x * .632},
  samptype = "swor",
  block.size = 10,
  vtry = 1,
  ...)
```

Arguments

`formula`	A symbolic description of the model to be fit.
`data`	Data frame containing the y-outcome and x-variables.
`ntree`	Function specifying requested number of trees used for growing the forest. Inputs are dimension and number of holdout variables. The requested number of trees can also be a number.
`nsplit`	Non-negative integer value specifying number of random split points used to split a node (deterministic splitting corresponds to the value zero and is much slower).
`ntime`	Integer value used for survival to constrain ensemble calculations to a grid of `ntime` time points.
`sampsize`	Function specifying size of subsampled data. Can also be a number.
`samptype`	Type of bootstrap used.
`vtry`	Number of variables randomly selected to be held out when growing a tree. This can also be set to a list for a targeted hold out VIMP analysis. See details below for more information.
`block.size`	Specifies number of trees in a block when calculating holdout variable importance.
`...`	Further arguments to be passed to `rfsrc`.

Details

Holdout variable importance (holdout VIMP) is based on comparing error performance of two mini forests of trees (blocks of trees): the first in which a random set of `vtry` features are held out (the holdout forest), and the second in which no features are held out (the baseline forest).

To summarize, holdout VIMP measures the importance of a variable when that variable is truly removed from the tree growing process.

Specifically, if a feature is held out in a block of trees, we refer to this as the (feature, block) pair. The bootstrap for the trees in a (feature, block) pair are identical in both forests. That is, the holdout block is grown by holding out the feature, and the baseline block is grown over the same trees, with the same bootstrap, but without holding out any features. `vtry` controls how many features are held out in every tree. If set to one (default), only one variable is held out in every tree. Once a (feature, block) of trees has been grown, holdout VIMP for a given variable v is calculated as follows. Gather the block of trees where the feature was held out (from the holdout forest) and calculate OOB prediction error. Next gather the corresponding block of trees where v was not held out (from the baseline forest) and calculate OOB prediction error. Holdout VIMP for the (feature, block) pair is the difference between these two values. The final holdout VIMP estimate for a feature v is obtained by averaging holdout VIMP for (feature=v, block) over all blocks.

Accuracy of hold out VIMP depends critically on total number of trees. If total number of trees is too small, then number of times a variable is held out will be small and OOB error can suffer from high variance. Therefore, `ntree` should be set fairly high—we recommend using 1000 times the number of features. Increasing `vtry` is another way to increase number of times a variable is held out and therefore reduces the burden of growing a large number of trees. In particular, total number of trees needed decreases linearly with `vtry`. The default `ntree` equals 1000 trees for each feature divided by `vtry`. Keep in mind intrepretation of holdout VIMP is altered when `vtry` is different than one. Thus this option should be used with caution.

Accuracy also depends on the value of `block.size`. Smaller values generally produce better results but are more computationally demanding. The most computationally demanding, but most accurate, is `block.size=1`. This is similar to how `block.size` is used for usual variable importance: see the help file for `rfsrc` for details. Note the value of `block.size` should not exceed `ntree` divided by number of features, otherwise there may not be enough trees to satisify the target block size for a feature and missing values will result.

A targeted hold out VIMP analysis can be requested by setting `vtry` to a list with two entries. The first entry is a vector of integer values specifying the variables of interest. The second entry is a boolean logical flag indicating whether individual or joint VIMP should be calculated. For example, suppose variables 1, 4 and 5 are our variables of interest. To calculate holdout VIMP for these variables, and these variables only, `vtry` would be specified by

vtry = list(xvar = c(1, 4, 5), joint = FALSE)

On the other hand, if we are interested in the joint effect when we remove the three variables simultaneously, then

vtry = list(xvar = c(1, 4, 5), joint = TRUE)

The benefits of a targeted analysis is that the user may have a pre-conceived idea of which variables are interesting. Only VIMP for these variables will be calculated which greatly reduces computational time. Another benefit is that when joint VIMP is requested, this provides the user with a way to assess importance of specific groups of variables. See the iris example below for illustration.

Value

Invisibly a list with the following components (which themselves can be lists):

`importance`	Holdout VIMP.
`baseline`	Prediction error for the baseline forest.
`holdout`	Prediction error for the holdout forest.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

See Also

`vimp.rfsrc`

Examples

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

## new York air quality measurements
airq.obj <- holdout.vimp(Ozone ~ ., data = airquality, na.action = "na.impute")
print(airq.obj$importance)

## ------------------------------------------------------------
## classification analysis
## ------------------------------------------------------------

## iris data
iris.obj <- holdout.vimp(Species ~., data = iris)
print(iris.obj$importance)

## iris data using brier prediction error
iris.obj <- holdout.vimp(Species ~., data = iris, perf.type = "brier")
print(iris.obj$importance)

## ------------------------------------------------------------
## illustration of targeted holdout vimp analysis
## ------------------------------------------------------------

## iris data - only interested in variables 3 and 4
vtry <- list(xvar = c(3, 4), joint = FALSE)
print(holdout.vimp(Species ~., data = iris, vtry = vtry)$impor)
```

```
## iris data - joint importance of variables 3 and 4
vtry <- list(xvar = c(3, 4), joint = TRUE)
print(holdout.vimp(Species ~., data = iris, vtry = vtry)$impor)

## iris data - joint importance of variables 1 and 2
vtry <- list(xvar = c(1, 2), joint = TRUE)
print(holdout.vimp(Species ~., data = iris, vtry = vtry)$impor)


## ------------------------------------------------------------
## imbalanced classification (using RFQ)
## ------------------------------------------------------------

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

  ## experimental settings
  n <- 400
  q <- 20
  ir <- 6
  f <- as.formula(Class ~ .)

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

  ## VIMP for RFQ with and without blocking
  vmp1 <- imbalanced(f, d, importance = TRUE, block.size = 1)$importance[, 1]
  vmp10 <- imbalanced(f, d, importance = TRUE, block.size = 10)$importance[, 1]

  ## holdout VIMP for RFQ with and without blocking
  hvmp1 <- holdout.vimp(f, d, rfq =  TRUE,
              perf.type = "g.mean", block.size = 1)$importance[, 1]
  hvmp10 <- holdout.vimp(f, d, rfq =  TRUE,
              perf.type = "g.mean", block.size = 10)$importance[, 1]

  ## compare VIMP values
  imp <- 100 * cbind(vmp1, vmp10, hvmp1, hvmp10)
  legn <- c("vimp-1", "vimp-10","hvimp-1", "hvimp-10")
  colr <- rep(4,20+q)
  colr[1:20] <- 2
  ylim <- range(c(imp))
  nms <- 1:(20+q)
  par(mfrow=c(2,2))
  barplot(imp[,1],col=colr,las=2,main=legn[1],ylim=ylim,names.arg=nms)
  barplot(imp[,2],col=colr,las=2,main=legn[2],ylim=ylim,names.arg=nms)
  barplot(imp[,3],col=colr,las=2,main=legn[3],ylim=ylim,names.arg=nms)
  barplot(imp[,4],col=colr,las=2,main=legn[4],ylim=ylim,names.arg=nms)

}
```

```
## ------------------------------------------------------------
## multivariate regression analysis
## ------------------------------------------------------------
mtcars.mreg <- holdout.vimp(Multivar(mpg, cyl) ~., data = mtcars,
                                  vtry = 3,
                                  block.size = 1,
                                  samptype = "swr",
                                  sampsize = dim(mtcars)[1])
print(mtcars.mreg$importance)

## ------------------------------------------------------------
## mixed outcomes analysis
## ------------------------------------------------------------

mtcars.new <- mtcars
mtcars.new$cyl <- factor(mtcars.new$cyl)
mtcars.new$carb <- factor(mtcars.new$carb, ordered = TRUE)
mtcars.mix <- holdout.vimp(cbind(carb, mpg, cyl) ~., data = mtcars.new,
                                 ntree = 100,
                                 block.size = 2,
                                 vtry = 1)
print(mtcars.mix$importance)

##------------------------------------------------------------
## survival analysis
##------------------------------------------------------------

## Primary biliary cirrhosis (PBC) of the liver
data(pbc, package = "randomForestSRC")
pbc.obj <- holdout.vimp(Surv(days, status) ~ ., pbc,
                                nsplit = 10,
                                ntree = 1000,
                                na.action = "na.impute")
print(pbc.obj$importance)

##------------------------------------------------------------
## competing risks
##------------------------------------------------------------

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

data(wihs, package = "randomForestSRC")
wihs.obj <- holdout.vimp(Surv(time, status) ~ ., wihs,
                                 nsplit = 3,
                                 ntree = 100)
print(wihs.obj$importance)
```

---

housing *Ames Iowa Housing Data*

Description

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

References

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

Examples

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

## the original data contains lots of missing data, so impute it
## use missForest, can be slow so grow trees with small training sizes
housing2 <- impute(data = housing, mf.q = 1, sampsize = function(x){x * .1})

## same idea ... but directly use rfsrc.fast and multivariate missForest
housing3 <- impute(data = housing, mf.q = .5, fast = TRUE)

## even faster, but potentially less acurate
housing4 <- impute(SalePrice~., housing, splitrule = "random", nimpute = 1)
```

---

imbalanced.rfsrc *Imbalanced Two Class Problems*

Description

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

Usage

```
## S3 method for class 'rfsrc'
imbalanced(formula, data, ntree = 3000,
  method = c("rfq", "brf", "standard"), splitrule = "auc",
  perf.type = NULL, block.size = NULL, fast = FALSE,
  ratio = NULL, ...)
```

Arguments

formula	A symbolic description of the model to be fit.
data	Data frame containing the two-class y-outcome and x-variables.
ntree	Number of trees.
method	Method used for fitting the classifier. The default is rfq which is the random forests quantile-classifer (RFQ) approach of O'Brien and Ishwaran (2017). The method brf implements the balanced random forest (BRF) method of Chen et al. (2004) which undersamples the majority class so that its cardinality matches that of the minority class. The method standard implements a standard random forest analysis.
splitrule	Default is AUC splitting which maximizes gmean performance. Other choices are "gini" and "entropy".
perf.type	Measure used for assessing performance (and all downstream calculations based on it such as variable importance). The default for rfq and brf is to use the G-mean (Kubat et al., 1997). For standard random forests, the default is misclassification error. Users can over-ride the default performance measure by manually selecting either gmean for the G-mean, misclass for misclassification error, or brier for the normalized Brier score. See the examples below.
block.size	Should the cumulative error rate be calculated on every tree? When NULL, it will only be calculated on the last tree. If importance is requested, VIMP is calculated in "blocks" of size equal to block.size. If not specified, uses the default value specified in rfsrc.
fast	Use fast random forests, rfsrc.fast, in place of rfsrc? Improves speed but is less accurate. Only applies to RFQ.
ratio	This is an optional parameter for expert users and included only for experimental purposes. Used to specify the ratio (between 0 and 1) for undersampling the majority class. Sampling is without replacement. Option is ignored for BRF.
...	Further arguments to be passed to the rfsrc function to specify random forest parameters.

Details

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

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

Permutation VIMP is used by default and not anti-VIMP which is the default for all other families and settings. Our experiments indicate the former performs better in imbalanced settings, especially when imbalanced ratio is high.

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

A new helper function `get.imbalanced.performance` has been added for extracting performance metrics. Metrics are self-titled and their meaning should generally be clear. Metrics that may be less familiar include: F1, the F-score or the F-measure which measures balance between the precision and the recall. F1mod, the harmonic mean of sensitivity, specificity, precision and the negative predictive value. F1gmean, the average of F1 and the G-mean. F1modgmean, the average of F1mod and the G-mean.

Value

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

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

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

O'Brien R. and Ishwaran H. (2019). A random forests quantile classifier for class imbalanced data. *Pattern Recognition*, 90, 232-249

See Also

`rfsrc`, `rfsrc.fast`

Examples

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

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

##------------------------------------------------------------------
## default RFQ call
##------------------------------------------------------------------

o.rfq <- imbalanced(f, breast)
print(o.rfq)

## equivalent to:
## rfsrc(f, breast, rfq =  TRUE, ntree = 3000,
```

```
##       perf.type = "gmean", splitrule = "auc")

##------------------------------------------------------------------
## detailed output using customized performance function
##------------------------------------------------------------------

print(get.imbalanced.performance(o.rfq))

##------------------------------------------------------------------
## RF using misclassification error with gini splitting
## ----------------------------------------------------------------

o.std <- imbalanced(f, breast, method = "stand", splitrule = "gini")

##------------------------------------------------------------------
## RF using G-mean performance with AUC splitting
## ----------------------------------------------------------------

o.std <- imbalanced(f, breast, method = "stand", perf.type = "gmean")

## equivalent to:
## rfsrc(f, breast, ntree = 3000, perf.type = "gmean", splitrule = "auc")

##------------------------------------------------------------------
## default BRF call
##------------------------------------------------------------------

o.brf <- imbalanced(f, breast, method = "brf")

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

##------------------------------------------------------------------
## BRF call with misclassification performance
##------------------------------------------------------------------

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

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

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


##------------------------------------------------------------------
##
##  illustrates how to optimize threshold on training data
```

```
##  improves Gmean for RFQ in many situations
##
##------------------------------------------------------------------

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

  ## experimental settings
  n <- 2 * 5000
  q <- 20
  ir <- 6
  f <- as.formula(Class ~ .)

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

  ## split data into train and test
  trn.pt <- sample(1:nrow(d), size = nrow(d) / 2)
  trn <- d[trn.pt, ]
  tst <- d[setdiff(1:nrow(d), trn.pt), ]

  ## run rfq on training data
  o <- imbalanced(f, trn)

  ## (1) default threshold (2) directly optimized gmean threshold
  th.1 <- get.imbalanced.performance(o)["threshold"]
  th.2 <- get.imbalanced.optimize(o)["threshold"]

  ## training performance
  cat("-------- train performance ---------\n")
  print(get.imbalanced.performance(o, thresh=th.1))
  print(get.imbalanced.performance(o, thresh=th.2))

  ## test performance
  cat("-------- test performance ---------\n")
  pred.o <- predict(o, tst)
  print(get.imbalanced.performance(pred.o, thresh=th.1))
  print(get.imbalanced.performance(pred.o, thresh=th.2))

}

##------------------------------------------------------------------
##  illustrates RFQ with and without SMOTE
##
## - simulation example using the caret R-package
## - creates imbalanced data by randomly sampling the class 1 data
## - use SMOTE from "imbalance" package to oversample the minority
##
##------------------------------------------------------------------
```

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

  ## experimental settings
  n <- 5000
  q <- 20
  ir <- 6
  f <- as.formula(Class ~ .)

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

  ## define train/test split
  trn <- sample(1:nrow(d), size = nrow(d) / 2, replace = FALSE)

  ## now make SMOTE training data
  newd.50 <- mwmote(d[trn, ], numInstances = 50, classAttr = "Class")
  newd.500 <- mwmote(d[trn, ], numInstances = 500, classAttr = "Class")

  ## fit RFQ with and without SMOTE
  o.with.50 <- imbalanced(f, rbind(d[trn, ], newd.50))
  o.with.500 <- imbalanced(f, rbind(d[trn, ], newd.500))
  o.without <- imbalanced(f, d[trn, ])

  ## compare performance on test data
  print(predict(o.with.50, d[-trn, ]))
  print(predict(o.with.500, d[-trn, ]))
  print(predict(o.without, d[-trn, ]))

}

##------------------------------------------------------------------
##
## illustrates effectiveness of blocked VIMP
##
##------------------------------------------------------------------

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

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

  ## simulate the data, create minority class data
  d <- twoClassSim(n, linearVars = 15, noiseVars = q)
  d$Class <- factor(as.numeric(d$Class) - 1)
```

```
  idx.0 <- which(d$Class == 0)
  idx.1 <- sample(which(d$Class == 1), sum(d$Class == 1) / ir , replace = FALSE)
  d <- d[c(idx.0,idx.1),, drop = FALSE]

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

  ## permutation VIMP for RFQ with and without blocking
  rfq <- imbalanced(f, d, importance = "permute", block.size = 1)
  rfqB <- imbalanced(f, d, importance = "permute", block.size = 10)

  ## compare VIMP values
  imp <- 100 * cbind(brf$importance[, 1], brfB$importance[, 1],
                     rfq$importance[, 1], rfqB$importance[, 1])
  legn <- c("BRF", "BRF-block", "RFQ", "RFQ-block")
  colr <- rep(4,20+q)
  colr[1:20] <- 2
  ylim <- range(c(imp))
  nms <- 1:(20+q)
  par(mfrow=c(2,2))
  barplot(imp[,1],col=colr,las=2,main=legn[1],ylim=ylim,names.arg=nms)
  barplot(imp[,2],col=colr,las=2,main=legn[2],ylim=ylim,names.arg=nms)
  barplot(imp[,3],col=colr,las=2,main=legn[3],ylim=ylim,names.arg=nms)
  barplot(imp[,4],col=colr,las=2,main=legn[4],ylim=ylim,names.arg=nms)

}

##------------------------------------------------------------------
##
## confidence intervals for G-mean permutation VIMP using subsampling
##
##------------------------------------------------------------------

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

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

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

  ## RFQ
  o <- imbalanced(Class ~ ., d, importance = "permute", block.size = 10)
```

```
  ## subsample RFQ
  smp.o <- subsample(o, B = 100)
  plot(smp.o, cex.axis = .7)

}
```

`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 = 100, nodesize = 1, nsplit = 10,
  nimpute = 2, fast = FALSE, blocks,
  mf.q, max.iter = 10, eps = 0.01,
  ytry = NULL, always.use = NULL, verbose = TRUE,
  ...)
```

Arguments

`formula`	A symbolic description of the model to be fit. Can be left unspecified if there are no outcomes or we don't care to distinguish between y-outcomes and x-variables in the imputation. Ignored when using multivariate missForest imputation.
`data`	Data frame containing the data to be imputed.
`ntree`	Number of trees to grow.
`nodesize`	Forest average terminal node size.
`nsplit`	Non-negative integer value used to specify random splitting.
`nimpute`	Number of iterations of the missing data algorithm. Ignored for multivariate missForest; in which case the algorithm iterates until a convergence criteria is achieved (users can however enforce a maximum number of iterations with the option `max.iter`).
`fast`	Use fast random forests, `rfsrcFast`, in place of `rfsrc`? Improves speed but is less accurate.
`blocks`	Integer value specifying the number of blocks the data should be broken up into (by rows). This can improve computational efficiency when the sample size is large but imputation efficiency decreases. By default, no action is taken if left unspecified.

mf.q	Use this to turn on missForest (which is off by default). Specifies fraction of variables (between 0 and 1) used as responses in multivariate missForest imputation. When set to 1 this corresponds to missForest, otherwise multivariate missForest is used. Can also be an integer, in which case this equals the number of multivariate responses.
max.iter	Maximum number of iterations used when implementing multivariate missForest imputation.
eps	Tolerance value used to determine convergence of multivariate missForest imputation.
ytry	Number of variables used as pseudo-responses in unsupervised forests. See details below.
always.use	Character vector of variable names to always be included as a response in multivariate missForest imputation. Does not apply for other imputation methods.
verbose	Send verbose output to terminal (only applies to multivariate missForest imputation).
...	Further arguments passed to or from other methods.

Details

1. Grow a forest and use this to impute data. All external calculations such as ensemble calculations, error rates, etc. are turned off. Use this function if your only interest is imputing the data.
2. Split statistics are calculated using non-misssing data only. If a node splits on a variable with missing data, the variable's missing data is imputed by randomly drawing values from non-missing in-bag data. The purpose of this is to make it possible to assign cases to daughter nodes based on the split.
3. If no formula is specified, unsupervised splitting is implemented using a ytry value of sqrt(p) where p equals the number of variables. More precisely, mtry variables are selected at random, and for each of these a random subset of ytry variables are selected and defined as the multivariate pseudo-responses. A multivariate composite splitting rule of dimension ytry is then applied to each of the mtry multivariate regression problems and the node split on the variable leading to the best split (Tang and Ishwaran, 2017).
4. If mf.q is specified, a multivariate version of missForest imputation (Stekhoven and Buhlmann, 2012) is applied. Specifically, a fraction mf.q of variables are used as multivariate responses and split by the remaining variables using multivariate composite splitting (Tang and Ishwaran, 2017). Missing data for responses are imputed by prediction. The process is repeated using a new set of variables for responses (mutually exclusive to the previous fit), until all variables have been imputed. This is one iteration. The entire process is repeated, and the algorithm iterated until a convergence criteria is met (specified using options max.iter and eps). Integer values for mf.q are allowed and interpreted as a request that mf.q variables be selected for the multivariate response. If mf.q=1, the algorithm reverts to the original missForest procedure. This is generally the most accurate of all the imputation procedures, but also the most computationally demanding. See examples below for strategies to increase speed.
5. Prior to imputation, the data is processed and records with all values missing are removed, as are variables having all missing values.

6. If there is no missing data, either before or after processing of the data, the algorithm returns the processed data and no imputation is performed.
7. 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 overlaid.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

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

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

See Also

`rfsrc`, `rfsrc.fast`

Examples

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

## default everything - unsupervised splitting
data(pbc, package = "randomForestSRC")
pbc1.d <- impute(data = pbc)

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

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

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

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

## ------------------------------------------------------------
```

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

data(pbc, package = "randomForestSRC")

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

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

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

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

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

## ------------------------------------------------------------
## another example of multivariate missForest imputation
## (suggested by John Sheffield)
## ------------------------------------------------------------

test_rows <- 1000

set.seed(1234)

a <- rpois(test_rows, 500)
b <- a + rnorm(test_rows, 50, 50)
c <- b + rnorm(test_rows, 50, 50)
d <- c + rnorm(test_rows, 50, 50)
e <- d + rnorm(test_rows, 50, 50)
f <- e + rnorm(test_rows, 50, 50)
g <- f + rnorm(test_rows, 50, 50)
h <- g + rnorm(test_rows, 50, 50)
i <- h + rnorm(test_rows, 50, 50)

fake_data <- data.frame(a, b, c, d, e, f, g, h, i)

fake_data_missing <- data.frame(lapply(fake_data, function(x) {
  x[runif(test_rows) <= 0.4] <- NA
  x
}))

imputed_data <- impute(
  data = fake_data_missing,
  mf.q = 0.2,
  ntree = 100,
  fast = TRUE,
  verbose = TRUE
```

```
)

par(mfrow=c(3,3))
o=lapply(1:ncol(imputed_data), function(j) {
  pt <- is.na(fake_data_missing[, j])
  x <- fake_data[pt, j]
  y <- imputed_data[pt, j]
  plot(x, y, pch = 16, cex = 0.8, xlab = "raw data",
    ylab = "imputed data", col = 2)
  points(x, y, pch = 1, cex = 0.8, col = gray(.9))
  lines(supsmu(x, y, span = .25), lty = 1, col = 4, lwd = 4)
  mtext(colnames(imputed_data)[j])
  NULL
})
```

`max.subtree.rfsrc` *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 p`x`max.order. If '`max.order=0`', a matrix of p`x`ntree 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

holdout.vimp.rfsrc, var.select.rfsrc, vimp.rfsrc

Examples

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

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

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

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

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

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

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

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

nutrigenomic *Nutrigenomic Study*

Description

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

References

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

Examples

```
## ------------------------------------------------------------
## multivariate regression forests using Mahalanobis splitting
## lipids (all real values) used as the multivariate y
## ------------------------------------------------------------

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

## parse into y and x data
ydta <- nutrigenomic$lipids
xdta <- data.frame(nutrigenomic$genes,
                   diet = nutrigenomic$diet,
                   genotype = nutrigenomic$genotype)

## multivariate mixed forest call
obj <- rfsrc(get.mv.formula(colnames(ydta)),
             data.frame(ydta, xdta),
             importance=TRUE, nsplit = 10,
             splitrule = "mahalanobis")
print(obj)

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

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

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

par(mfrow = c(1,2))
```

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


## ------------------------------------------------------------
## plot some trees
## ------------------------------------------------------------

plot(get.tree(obj, 1))
plot(get.tree(obj, 2))
plot(get.tree(obj, 3))


## ------------------------------------------------------------
##
## Compare above to (1) user specified covariance matrix
##                  (2) default composite (independent) splitting
##
## ------------------------------------------------------------

## user specified sigma matrix
obj2 <- rfsrc(get.mv.formula(colnames(ydta)),
              data.frame(ydta, xdta),
              importance = TRUE, nsplit = 10,
              splitrule = "mahalanobis",
              sigma = cov(ydta))
print(obj2)

## default independence split rule
obj3 <- rfsrc(get.mv.formula(colnames(ydta)),
              data.frame(ydta, xdta),
              importance=TRUE, nsplit = 10)
print(obj3)

## compare vimp
imp <- data.frame(mahalanobis  = rowMeans(get.mv.vimp(obj,  standardize = TRUE)),
                  mahalanobis2 = rowMeans(get.mv.vimp(obj2, standardize = TRUE)),
                  default      = rowMeans(get.mv.vimp(obj3, standardize = TRUE)))

print(head(100 * imp[order(imp$mahalanobis, decreasing = TRUE), ], 15))
```

`partial.rfsrc` *Acquire Partial Effect of a Variable*

Description

Direct, fast inferface for partial effect of a variable. Works for all families.

Usage

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

Arguments

object	An object of class (rfsrc, grow).
oob	By default out-of-bag values are returned, but inbag values can be requested by setting this option to FALSE.
partial.type	Character vector specifying type of predicted value requested. See details below.
partial.xvar	Character value specifying the single primary partial x-variable to be used.
partial.values	Vector of values that the primary partialy x-variable will assume.
partial.xvar2	Vector of character values specifying the second order x-variables to be used.
partial.values2	Vector of values that the second order x-variables will assume. Each second order x-variable can only assume a single value. This the length of partial.xvar2 and partial.values2 will be the same. In addition, the user must do the appropriate conversion for factors, and represent a value as a numeric element.
partial.time	For survival families, the time at which the predicted survival value is evaluated at (depends on partial.type).
get.tree	Vector of integer(s) identifying trees over which the partial values are calculated over. By default, uses all trees in the forest.
seed	Negative integer specifying seed for the random number generator.
do.trace	Number of seconds between updates to the user on approximate time to completion.
...	Further arguments passed to or from other methods.

Details

Used for direct, efficient call to obtain partial plot effects. This function is intended primarily for experts.

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

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

The utility function get.partial.plot.data is supplied for processing returned raw partial effects in a format more convenient for plotting. Options are specified as in plot.variable. See examples for illustration.

Raw partial plot effects data is returned either as an array or a list of length equal to the number of outcomes (length is one for univariate families) with entries depending on the underlying family:

1. For regression, partial plot data is returned as a list in regrOutput with dim [n] x [length(partial.values)].

2. For classification, partial plot data is returned as a list in `classOutput` of dim `[n]` x `[1 + yvar.nlevels[.]]` x `[length(partial.values)]`.
3. For mixed multivariate regression, values are returned in list format both in `regrOutput` and `classOutput`
4. For survival, values are returned as either a matrix or array in `survOutput`. Depending on partial type specified this can be:
   - For partial type `surv` returns the survival function of dim `[n]` x `[length(partial.time)]` x `[length(partial.values)]`.
   - For partial type `mort` returns mortality of dim `[n]` x `[length(partial.values)]`.
   - For partial type `chf` returns the cumulative hazard function of dim `[n]` x `[length(partial.time)]` x `[length(partial.values)]`.
5. For competing risks, values are returned as either a matrix or array in `survOutput`. Depending on the options specified this can be:
   - For partial type `years.lost` returns the expected number of life years lost of dim `[n]` x `[length(event.info$event.type)]` x `[length(partial.values)]`.
   - For partial type `cif` returns the cumulative incidence function of dim `[n]` x `[length(partial.time)]` x `[length(event.info$event.type)]` x `[length(partial.values)]`.
   - For partial type `chf` returns the cumulative hazard function of dim `[n]` x `[length(partial.time)]` x `[length(event.info$event.type)]` x `[length(partial.values)]`.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

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

See Also

`plot.variable.rfsrc`

Examples

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

airq.obj <- rfsrc(Ozone ~ ., data = airquality)

## partial effect for wind
partial.obj <- partial(airq.obj,
                  partial.xvar = "Wind",
```

```
                  partial.values = airq.obj$xvar$Wind)
pdta <- get.partial.plot.data(partial.obj)

## plot partial values
plot(pdta$x, pdta$yhat, type = "b", pch = 16,
      xlab = "wind", ylab = "partial effect of wind")


## example where we display all the partial effects
## instead of averaging - use the granule=TRUE option
pdta <- get.partial.plot.data(partial.obj, granule = TRUE)
boxplot(pdta$yhat ~ pdta$x, xlab = "Wind", ylab = "partial effect")

## ------------------------------------------------------------
##
## regression: partial effects for two variables simultaneously
##
## ------------------------------------------------------------

airq.obj <- rfsrc(Ozone ~ ., data = airquality)

## specify wind and temperature values of interest
wind <- sort(unique(airq.obj$xvar$Wind))
temp <- sort(unique(airq.obj$xvar$Temp))

## partial effect for wind, for a given temp
pdta <- do.call(rbind, lapply(temp, function(x2) {
  o <- partial(airq.obj,
         partial.xvar = "Wind", partial.xvar2 = "Temp",
         partial.values = wind, partial.values2 = x2)
  cbind(wind, x2, get.partial.plot.data(o)$yhat)
}))
pdta <- data.frame(pdta)
colnames(pdta) <- c("wind", "temp", "effectSize")

## coplot of partial effect of wind and temp
coplot(effectSize ~ wind|temp, pdta, pch = 16, overlap = 0)


## ------------------------------------------------------------
##
## regression: partial effects for three variables simultaneously
## (can be slow, so modify accordingly)
##
## ------------------------------------------------------------

n <- 1000
x <- matrix(rnorm(n * 3), ncol = 3)
y <- x[, 1] + x[, 1] * x[, 2] + x[, 1] * x[, 2] * x[, 3]
o <- rfsrc(y ~ ., data = data.frame(y = y, x))

## define target x values
x1 <- seq(-3, 3, length = 40)
```

```
x2 <- x3 <- seq(-3, 3, length = 10)

## extract second order partial effects
pdta <- do.call(rbind,
          lapply(x3, function(x3v) {
            cat("outer loop x3 = ", x3v, "\n")
            do.call(rbind,lapply(x2, function(x2v) {
              o <- partial(o,
                      partial.xvar = "X1",
                      partial.values = x1,
                      partial.xvar2 = c("X2", "X3"),
                      partial.values2 = c(x2v, x3v))
              cbind(x1, x2v, x3v, get.partial.plot.data(o)$yhat)
            }))
          }))
pdta <- data.frame(pdta)
colnames(pdta) <- c("x1", "x2", "x3", "effectSize")

## coplot of partial effects
coplot(effectSize ~ x1|x2*x3, pdta, pch = 16, overlap = 0)


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

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

## partial effect for sepal length
partial.obj <- partial(iris.obj,
                  partial.xvar = "Sepal.Length",
                  partial.values = iris.obj$xvar$Sepal.Length)

## extract partial effects for each species outcome
pdta1 <- get.partial.plot.data(partial.obj, target = "setosa")
pdta2 <- get.partial.plot.data(partial.obj, target = "versicolor")
pdta3 <- get.partial.plot.data(partial.obj, target = "virginica")

## plot the results
par(mfrow=c(1,1))
plot(pdta1$x, pdta1$yhat, type="b", pch = 16,
     xlab = "sepal length", ylab = "adjusted probability",
     ylim = range(pdta1$yhat,pdta2$yhat,pdta3$yhat))
points(pdta2$x, pdta2$yhat, col = 2, type = "b", pch = 16)
points(pdta3$x, pdta3$yhat, col = 4, type = "b", pch = 16)
legend("topleft", legend=levels(iris.obj$yvar), fill = c(1, 2, 4))

## ------------------------------------------------------------
##
## survival
##
```

```
## ------------------------------------------------------------

data(veteran, package = "randomForestSRC")
v.obj <- rfsrc(Surv(time,status)~., veteran, nsplit = 10, ntree = 100)

## partial effect of age on mortality
partial.obj <- partial(v.obj,
  partial.type = "mort",
  partial.xvar = "age",
  partial.values = v.obj$xvar$age,
  partial.time = v.obj$time.interest)
pdta <- get.partial.plot.data(partial.obj)

plot(lowess(pdta$x, pdta$yhat, f = 1/3),
   type = "l", xlab = "age", ylab = "adjusted mortality")

## example where x is discrete - partial effect of age on mortality
## we use the granule=TRUE option
partial.obj <- partial(v.obj,
       partial.type = "mort",
       partial.xvar = "trt",
       partial.values = v.obj$xvar$trt,
       partial.time = v.obj$time.interest)
pdta <- get.partial.plot.data(partial.obj, granule = TRUE)
boxplot(pdta$yhat ~ pdta$x, xlab = "treatment", ylab = "partial effect")


## partial effects of karnofsky score on survival
karno <- quantile(v.obj$xvar$karno)
partial.obj <- partial(v.obj,
  partial.type = "surv",
  partial.xvar = "karno",
  partial.values = karno,
  partial.time = v.obj$time.interest)
pdta <- get.partial.plot.data(partial.obj)

matplot(pdta$partial.time, t(pdta$yhat), type = "l", lty = 1,
     xlab = "time", ylab = "karnofsky adjusted survival")
legend("topright", legend = paste0("karnofsky = ", karno), fill = 1:5)


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

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

## partial effect of age on years lost
partial.obj <- partial(follic.obj,
  partial.type = "years.lost",
```

```
  partial.xvar = "age",
  partial.values = follic.obj$xvar$age,
  partial.time = follic.obj$time.interest)
pdta1 <- get.partial.plot.data(partial.obj, target = 1)
pdta2 <- get.partial.plot.data(partial.obj, target = 2)

par(mfrow=c(2,2))
plot(lowess(pdta1$x, pdta1$yhat),
   type = "l", xlab = "age", ylab = "adjusted years lost relapse")
plot(lowess(pdta2$x, pdta2$yhat),
   type = "l", xlab = "age", ylab = "adjusted years lost death")

## partial effect of age on cif
partial.obj <- partial(follic.obj,
  partial.type = "cif",
  partial.xvar = "age",
  partial.values = quantile(follic.obj$xvar$age),
  partial.time = follic.obj$time.interest)
pdta1 <- get.partial.plot.data(partial.obj, target = 1)
pdta2 <- get.partial.plot.data(partial.obj, target = 2)

matplot(pdta1$partial.time, t(pdta1$yhat), type = "l", lty = 1,
     xlab = "time", ylab = "age adjusted cif for relapse")
matplot(pdta2$partial.time, t(pdta2$yhat), type = "l", lty = 1,
     xlab = "time", ylab = "age adjusted cif for death")


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

mtcars2 <- mtcars
mtcars2$carb <- factor(mtcars2$carb)
mtcars2$cyl <- factor(mtcars2$cyl)
mtcars.mix <- rfsrc(Multivar(carb, mpg, cyl) ~ ., data = mtcars2)

## partial effect of displacement for each the three-outcomes
partial.obj <- partial(mtcars.mix,
                  partial.xvar = "disp",
                  partial.values = mtcars.mix$xvar$disp)
pdta1 <- get.partial.plot.data(partial.obj, m.target = "carb")
pdta2 <- get.partial.plot.data(partial.obj, m.target = "mpg")
pdta3 <- get.partial.plot.data(partial.obj, m.target = "cyl")

par(mfrow=c(2,2))
plot(lowess(pdta1$x, pdta1$yhat), type = "l", xlab="displacement", ylab="carb")
plot(lowess(pdta2$x, pdta2$yhat), type = "l", xlab="displacement", ylab="mpg")
plot(lowess(pdta3$x, pdta3$yhat), type = "l", xlab="displacement", ylab="cyl")
```

pbc *Primary Biliary Cirrhosis (PBC) Data*

Description

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

Source

Flemming and Harrington, 1991, Appendix D.1.

References

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

Examples

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

peakVO2 *Systolic Heart Failure Data*

Description

The data involve 2231 patients with systolic heart failure who underwent cardiopulmonary stress testing at the Cleveland Clinic. The primary end point was all-cause death. In total, 39 variables were measured for each patient, including baseline clinical values and exercise stress test results. A key variable of interest is peak VO2 (mL/kg per min), the peak respiratory exchange ratio. More details regarding the data can be found in Hsich et al. (2011).

References

Hsich E., Gorodeski E.Z.,Blackstone E.H., Ishwaran H. and Lauer M.S. (2011). Identifying important risk factors for survival in systolic heart failure patients using random survival forests. Circulation: Cardio. Qual. Outcomes, 4(1), 39-45.

Examples

```
## load the data
data(peakVO2, package = "randomForestSRC")

## random survival forest analysis
o <- rfsrc(Surv(ttodead, died)~., peakVO2)
print(o)

## partial effect of peak V02 on mortality
partial.o <- partial(o,
       partial.type = "mort",
       partial.xvar = "peak.vo2",
       partial.values = o$xvar$peak.vo2,
       partial.time = o$time.interest)
pdta.m <- get.partial.plot.data(partial.o)


## partial effect of peak V02 on survival
pvo2 <- quantile(o$xvar$peak.vo2)
partial.o <- partial(o,
       partial.type = "surv",
       partial.xvar = "peak.vo2",
       partial.values = pvo2,
       partial.time = o$time.interest)
pdta.s <- get.partial.plot.data(partial.o)


## compare the two plots
par(mfrow=c(1,2))

plot(lowess(pdta.m$x, pdta.m$yhat, f = 2/3),
     type = "l", xlab = "peak VO2", ylab = "adjusted mortality")
rug(o$xvar$peak.vo2)

matplot(pdta.s$partial.time, t(pdta.s$yhat), type = "l", lty = 1,
          xlab = "years", ylab = "peak VO2 adjusted survival")
legend("bottomleft", legend = paste0("peak VO2 = ", pvo2),
       bty = "n", cex = .75, fill = 1:5)
```

`plot.competing.risk.rfsrc` *Plots for Competing Risks*

Description

Plot useful summary curves from a random survival forest 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

Given a random survival forest object from a competing risk analysis (Ishwaran et al. 2014), plots from top to bottom, left to right: (1) cause-specific cumulative hazard function (CSCHF) for each event, (2) cumulative incidence function (CIF) for each event, and (3) continuous probability curves (CPC) for each event (Pepe and Mori, 1993).

Does not apply to right-censored data. Whenever possible, out-of-bag (OOB) values are displayed.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

Pepe, M.S. and Mori, M., (1993). Kaplan-Meier, marginal or conditional probability curves in summarizing competing risks failure time data? *Statistics in Medicine*, 12(8):737-751.

See Also

follic, hd, rfsrc, wihs

Examples

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

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

## ------------------------------------------------------------
## Hodgkin's Disease
## ------------------------------------------------------------

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

```
  hd.obj <- rfsrc(Surv(time, status) ~ ., hd, nsplit = 3, ntree = 100)
  print(hd.obj)
  plot.competing.risk(hd.obj)

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

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

`plot.quantreg.rfsrc` *Plot Quantiles from Quantile Regression Forests*

Description

Plots quantiles obtained from a quantile regression forest. Additionally insets the continuous rank probability score (crps), a useful diagnostic of accuracy.

Usage

```
## S3 method for class 'rfsrc'
plot.quantreg(x, prbL = .25, prbU = .75,
   m.target = NULL, crps = TRUE, subset = NULL, xlab = NULL, ylab = NULL, ...)
```

Arguments

`x`	A quantile regression object obtained from calling `quantreg`.
`prbL`	Lower quantile (preferably < .5).
`prbU`	Upper quantile (preferably > .5).
`m.target`	Character value for multivariate families specifying the target outcome to be used. If left unspecified, the algorithm will choose a default target.
`crps`	Calculate crps and inset it?
`subset`	Restricts plotted values to a subset of the data. Default is to use the entire data.
`xlab`	Horizontal axis label.
`ylab`	Vertical axis label.
`...`	Further arguments passed to or from other methods.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

See Also

quantreg.rfsrc

---

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

Description

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

Usage

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

Arguments

x	An object of class (rfsrc, grow), (rfsrc, synthetic), or (rfsrc, predict).
m.target	Character value for multivariate families specifying the target outcome to be used. If left unspecified, the algorithm will choose a default target.
plots.one.page	Should plots be placed on one page?
sorted	Should variables be sorted by importance values?
verbose	Should VIMP be printed?
...	Further arguments passed to or from other methods.

Details

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

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

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

Examples

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

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

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

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

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

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

---

plot.subsample.rfsrc *Plot Subsampled VIMP Confidence Intervals*

Description

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

Usage

```
## S3 method for class 'rfsrc'
plot.subsample(x, alpha = .01, xvar.names,
 standardize = TRUE, normal = TRUE, jknife = FALSE, target, m.target = NULL,
 pmax = 75, main = "", sorted = TRUE, show.plots = TRUE, ...)
```

Arguments

x	An object obtained from calling subample.
alpha	Desired level of significance.
xvar.names	Names of the x-variables to be used. If not specified all variables used.
standardize	Standardize VIMP? For regression families, VIMP is standardized by dividing by the variance. For all other families, VIMP is unaltered.
normal	Use parametric normal confidence regions or nonparametric regions? Generally, parametric regions perform better.
jknife	Use the delete-d jackknife variance estimator?
target	For classification families, an integer or character value specifying the class VIMP will be conditioned on (default is to use unconditional VIMP). For competing risk families, an integer value between 1 and J indicating the event VIMP is requested, where J is the number of event types. The default is to use the first event.
m.target	Character value for multivariate families specifying the target outcome to be used. If left unspecified, the algorithm will choose a default target.
pmax	Trims the data to this number of variables (sorted by VIMP).
main	Title used for plot.
sorted	Should variables be sorted by importance values?
show.plots	Should plots be displayed? Allows users to produce their own custom plots.
...	Further arguments that can be passed to bxp.

Details

Most of the options used by the R function bxp will work here and can be used for customization of plots. Currently the following parameters will work:

"xaxt", "yaxt", "las", "cex.axis", "col.axis", "cex.main", "col.main", "sub", "cex.sub", "col.sub", "ylab", "cex.lab", "col.lab"

Value

Invisibly, returns the boxplot data that is plotted.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

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

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

See Also

subsample.rfsrc

Examples

```
o <- rfsrc(Ozone ~ ., airquality)
oo <- subsample(o)
plot.subsample(oo)
plot.subsample(oo, xvar.names = o$xvar.names[1:3])
plot.subsample(oo, jknife = FALSE)
plot.subsample(oo, alpha = .01)
plot(oo,cex.axis=.5)
```

---

plot.survival.rfsrc *Plot of Survival Estimates*

Description

Plot various survival estimates.

Usage

```
## S3 method for class 'rfsrc'
plot.survival(x, show.plots = TRUE, subset,
  collapse = FALSE, cens.model = c("km", "rfsrc"), ...)
```

Arguments

x	An object of class (rfsrc, grow) or (rfsrc, predict).
show.plots	Should plots be displayed?
subset	Vector indicating which cases from x we want estimates for. All cases used if not specified.
collapse	Collapse the survival function?
cens.model	Using the training data, specifies method for estimating the censoring distribution used in the inverse probability of censoring weights (IPCW) for calculating the Brier score: km: Uses the Kaplan-Meier estimator. rfscr: Uses a censoring random survival forest estimator.
...	Further arguments passed to or from other methods.

Details

Produces the following 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 overall (non-stratified) Brier score.
3. Continuous rank probability score (CRPS) equal to the integrated Brier score divided by time.
4. Plot of mortality of each individual versus observed time. Points in blue correspond to events, black points are censored observations. Not given for prediction settings lacking survival response information.

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

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

Mortality (Ishwaran et al., 2008) represents estimated risk for an individual calibrated to the scale of 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).

The utility function `get.brier.survival` can be used to extract the Brier score among other useful quantities.

Value

Invisibly, the conditional and unconditional Brier scores, and the integrated Brier score.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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.

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

See Also

`plot.competing.risk.rfsrc`, `predict.rfsrc`, `rfsrc`

Examples

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

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

## use subset to focus on specific individuals
plot.survival(pbc.obj, subset = 3)
plot.survival(pbc.obj, subset = c(3, 10))
plot.survival(pbc.obj, subset = c(3, 10), collapse = TRUE)

## get.brier.survival function does many nice things!
plot(get.brier.survival(pbc.obj, cens.model="km")$brier.score,type="s", col=2)
lines(get.brier.survival(pbc.obj, cens.model="rfsrc")$brier.score, type="s", col=4)
legend("bottomright", legend=c("cens.model = km", "cens.model = rfsrc"), fill=c(2,4))
```

---

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

Description

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

Usage

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

Arguments

x	An object of class (rfsrc, grow), (rfsrc, synthetic), or (rfsrc, plot.variable).
xvar.names	Names of the x-variables to be used. If not supplied all variables will be used.
target	For classification, an integer or character value specifying the class to focus on (defaults to the first class). For competing risks, an integer value between 1 and J indicating the event of interest, where J is the number of event types. The default is to use the first event type.

m.target	Character value for multivariate families specifying the target outcome to be used. If left unspecified, the algorithm will choose a default target.
time	For survival, the time at which the predicted survival value is evaluated at (depends on surv.type).
surv.type	For survival, specifies the predicted value. See details below.
class.type	For classification, specifies the predicted value. See details below.
partial	Should partial plots be used?
oob	OOB (TRUE) or in-bag (FALSE) predicted values.
show.plots	Should plots be displayed?
plots.per.page	Integer value controlling page layout.
granule	Integer value controlling whether a plot for a specific variable should be treated as a factor and therefore given as a boxplot. Larger values coerce boxplots.
sorted	Should variables be sorted by importance values.
nvar	Number of variables to be plotted. Default is all.
npts	Maximum number of points used when generating partial plots for continuous variables.
smooth.lines	Use lowess to smooth partial plots.
subset	Vector indicating which rows of the x-variable matrix x$xvar to use. All rows are used if not specified. Do not define subset based on the original data (which could have been processed due to missing values or for other reasons in the previous forest call) but define subset based on the rows of x$xvar.
...	Further arguments passed to or from other methods.

Details

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

1. For regression, the predicted response is used.
2. For classification, it is the predicted class probability specified by 'target', or the class of maximum probability depending on 'class.type' is set to "prob" or "bayes".
3. For multivariate families, it is the predicted value of the outcome specified by 'm.target' and if that is a classification outcome, by 'target'.
4. For survival, the choices are:
   - Mortality (mort). Mortality (Ishwaran et al., 2008) represents estimated risk for an individual calibrated to the scale of 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).
   - Relative frequency of mortality (rel.freq).
   - Predicted survival (surv), where the predicted survival is for the time point specified using time (the default is the median follow up time).
5. For competing risks, the choices are:

- The expected number of life years lost (`years.lost`).
- The cumulative incidence function (`cif`).
- The cumulative hazard function (`chf`).

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

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

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

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

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

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

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

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

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

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

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

See Also

`rfsrc`, `synthetic.rfsrc`, `partial.rfsrc`, `predict.rfsrc`

Examples

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

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

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

## example using a pre-processed plot.variable to define custom plots
p.v <- plot.variable(v.obj, surv.type = "surv", partial = TRUE, show.plots = FALSE)
plotthis <- p.v$plotthis
plot(plotthis[["age"]], xlab = "age", ylab = "partial effect", type = "b")
boxplot(yhat ~ x, plotthis[["trt"]], xlab = "treatment", ylab = "partial effect")


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

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

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

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

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

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

## motor trend cars: predict number of carburetors
mtcars2 <- mtcars
mtcars2$carb <- factor(mtcars2$carb,
```

```
   labels = paste("carb", sort(unique(mtcars$carb))))
mtcars2.obj <- rfsrc(carb ~ ., data = mtcars2)
plot.variable(mtcars2.obj, partial = TRUE)

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


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

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

Description

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

Usage

```
## S3 method for class 'rfsrc'
predict(object,
  newdata,
  m.target = NULL,
  importance = c(FALSE, TRUE, "none", "anti", "permute", "random"),
  get.tree = NULL,
  block.size = if (any(is.element(as.character(importance),
                    c("none", "FALSE")))) NULL else 10,
  na.action = c("na.omit", "na.impute", "na.random"),
  outcome = c("train", "test"),
  perf.type = NULL,
  proximity = FALSE,
  forest.wt = FALSE,
```

```
  ptn.count = 0,
  distance = FALSE,
  var.used = c(FALSE, "all.trees", "by.tree"),
  split.depth = c(FALSE, "all.trees", "by.tree"),
  case.depth = FALSE,
  seed = NULL,
  do.trace = FALSE, membership = FALSE, statistics = FALSE,

  ...)
```

Arguments

object	An object of class (rfsrc, grow) or (rfsrc, forest).
newdata	Test data. If missing, the original grow (training) data is used.
m.target	Character vector for multivariate families specifying the target outcomes to be used. The default uses all coordinates.
importance	Method used for variable importance (VIMP). Also see vimp for more flexibility, including joint vimp calculations. See holdoutvimp for an alternate importance measure.
get.tree	Vector of integer(s) identifying trees over which the ensembles are calculated over. By default, uses all trees in the forest. As an example, the user can extract the ensemble, the VIMP , or proximity from a single tree (or several trees). Note that block.size will be over-ridden so that it is no larger than the requested number of trees. See example below illustrating how to extract VIMP for each tree.
block.size	Should the error rate be calculated on every tree? When NULL, it will only be calculated on the last tree. To view the error rate on every nth tree, set the value to an integer between 1 and ntree. If importance is requested, VIMP is calculated in "blocks" of size equal to block.size, thus resulting in a compromise between ensemble and permutation VIMP.
na.action	Missing value action. The default na.omit removes the entire record if any entry is NA. Selecting 'na.random' uses fast random imputation, while 'na.impute' uses the imputation method described in rfsrc.
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.
perf.type	Optional character value for requesting metric used for predicted value, variable importance (VIMP) and error rate. If not specified, values returned are calculated by the default action used for the family. Currently applicable only to classification and multivariate classification; allowed values are perf.type="misclass" (default), perf.type="brier" and perf.type="gmean".
proximity	Should proximity between test observations be calculated? Possible choices are "inbag", "oob", "all", TRUE, or FALSE — but some options may not be valid

	and will depend on the context of the predict call. The safest choice is TRUE if proximity is desired.
distance	Should distance between test observations be calculated? Possible choices are "inbag", "oob", "all", TRUE, or FALSE — but some options may not be valid and will depend on the context of the predict call. The safest choice is TRUE if distance is desired.
forest.wt	Should the forest weight matrix for test observations be calculated? Choices are the same as proximity.
ptn.count	The number of terminal nodes that each tree in the grow forest should be pruned back to. The terminal node membership for the pruned forest is returned but no other action is taken. The default is ptn.count=0 which does no pruning.
var.used	Record the number of times a variable is split?
split.depth	Return minimal depth for each variable for each case?
case.depth	Return a matrix recording the depth at which a case first splits in a tree. Default is FALSE.
seed	Negative integer specifying seed for the random number generator.
do.trace	Number of seconds between updates to the user on approximate time to completion.
membership	Should terminal node membership and inbag information be returned?
statistics	Should split statistics be returned? Values can be parsed using stat.split.
...	Further arguments passed to or from other methods.

Details

Predicted values are obtained by "dropping" test data down the trained forest (forest calculated using training data). Performance values are returned if test data contains y-outcome values. Single as well as joint VIMP are also returned if requested.

If no test data is provided, the original training data is used, and the code reverts to restore mode allowing the user to restore the original trained forest. This feature allows extracting outputs from the forest 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). Terminal nodes from the trained forest are recalculated using 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 predicted values are obtained from test data. Error rates and VIMP are calculated by bootstrapping the test data and using out-of-bagging to ensure unbiased estimates.

csv=TRUE returns case specific VIMP; cse=TRUE returns case specific error rates. Applies to all families except survival. These options can also be applied while training.

Value

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

call	The original grow call to rfsrc.
family	The family used in the analysis.

n	Sample size of test data (depends upon NA values).
ntree	Number of trees in the grow forest.
yvar	Test set y-outcomes or original grow y-outcomes if none.
yvar.names	A character vector of the y-outcome names.
xvar	Data frame of test set x-variables.
xvar.names	A character vector of the x-variable names.
leaf.count	Number of terminal nodes for each tree in the grow forest. Vector of length ntree.
proximity	Symmetric proximity matrix of the test data.
forest	The grow forest.
membership	Matrix recording terminal node membership for the test data where each column contains the node number that a case falls in for that tree.
inbag	Matrix recording inbag membership for the test data where each column contains the number of times that a case appears in the bootstrap sample for that tree.
var.used	Count of the number of times a variable was used in growing the forest.
imputed.indv	Vector of indices of records in test data with missing values.
imputed.data	Data frame comprising imputed test data. The first columns are the y-outcomes followed by the x-variables.
split.depth	Matrix (i,j) or array (i,j,k) recording the minimal depth for variable j for case i, either averaged over the forest, or by tree k.
node.stats	Split statistics returned when statistics=TRUE which can be parsed using stat.split.
err.rate	Cumulative OOB error rate for the test data if y-outcomes are present.
importance	Test set variable importance (VIMP). Can be NULL.
predicted	Test set predicted value.
predicted.oob	OOB predicted value (NULL unless 'outcome="test"').
quantile	Quantile value at probabilities requested.
quantile.oob	OOB quantile value at probabilities requested (NULL unless 'outcome="test"').
++++++++	for classification settings, additionally ++++++++
class	In-bag predicted class labels.
class.oob	OOB predicted class labels (NULL unless 'outcome="test"').
++++++++	for multivariate settings, additionally ++++++++
regrOutput	List containing performance values for test multivariate regression responses (applies only in multivariate settings).

clasOutput	List containing performance values for test multivariate categorical (factor) responses (applies only in multivariate settings).
++++++++	for survival settings, additionally ++++++++
chf	Cumulative hazard function (CHF).
chf.oob	OOB CHF (NULL unless 'outcome="test"').
survival	Survival function.
survival.oob	OOB survival function (NULL unless 'outcome="test"').
time.interest	Ordered unique death times.
ndead	Number of deaths.
++++++++	for competing risks, additionally ++++++++
chf	Cause-specific cumulative hazard function (CSCHF) for each event.
chf.oob	OOB CSCHF for each event (NULL unless 'outcome="test"').
cif	Cumulative incidence function (CIF) for each event.
cif.oob	OOB CIF (NULL unless 'outcome="test"').

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 which can be extracted using functions get.mv.error, get.mv.predicted and get.mv.vimp.

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

holdout.vimp.rfsrc, plot.competing.risk.rfsrc, plot.rfsrc, plot.survival.rfsrc, plot.variable.rfsrc, rfsrc, rfsrc.fast, stat.split.rfsrc, synthetic.rfsrc, vimp.rfsrc

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, ])
veteran.pred <- predict(veteran.grow, veteran[-train, ])
print(veteran.grow)
print(veteran.pred)


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

## first train the forest
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))

## ------------------------------------------------------------
## prediction when test data has missing values
## ------------------------------------------------------------

data(pbc, package = "randomForestSRC")
trn <- pbc[1:312,]
tst <- pbc[-(1:312),]
o <- rfsrc(Surv(days, status) ~ ., trn)

## default imputation method used by rfsrc
print(predict(o, tst, na.action = "na.impute"))

## random imputation
```

```
print(predict(o, tst, na.action = "na.random"))

## ------------------------------------------------------------
## requesting different performance for classification
## ------------------------------------------------------------

## default performance is misclassification
o <- rfsrc(Species~., iris)
print(o)

## get (normalized) brier performance
print(predict(o, perf.type = "brier"))

## ------------------------------------------------------------
## vimp for each tree: illustrates get.tree
## ------------------------------------------------------------

## regression analysis but no VIMP
o <- rfsrc(mpg~., mtcars)

## now extract VIMP for each tree using get.tree
vimp.tree <- do.call(rbind, lapply(1:o$ntree, function(b) {
     predict(o, get.tree = b, importance = TRUE)$importance
}))

## boxplot of tree VIMP
boxplot(vimp.tree, outline = FALSE, col = "cyan")
abline(h = 0, lty = 2, col = "red")

## summary information of tree VIMP
print(summary(vimp.tree))

## extract tree-averaged VIMP using importance=TRUE
## remember to set block.size to 1
print(predict(o, importance = TRUE, block.size = 1)$importance)

## use direct call to vimp() for tree-averaged VIMP
print(vimp(o, block.size = 1)$importance)

## ------------------------------------------------------------
## vimp for just a few trees
## illustrates how to get vimp if you have a large data set
## ------------------------------------------------------------

## survival analysis but no VIMP
data(pbc, package = "randomForestSRC")
o <- rfsrc(Surv(days, status) ~ ., pbc, ntree = 2000)

## get vimp for a small number of trees
print(predict(o, get.tree=1:250, importance = TRUE)$importance)


## ------------------------------------------------------------
```

```
## case-specific vimp
## returns VIMP for each case
## ------------------------------------------------------------

o <- rfsrc(mpg~., mtcars)
op <- predict(o, importance = TRUE, csv = TRUE)
csvimp <- get.mv.csvimp(op, standardize=TRUE)
print(csvimp)

## ------------------------------------------------------------
## case-specific error rate
## returns tree-averaged error rate for each case
## ------------------------------------------------------------

o <- rfsrc(mpg~., mtcars)
op <- predict(o, importance = TRUE, cse = TRUE)
cserror <- get.mv.cserror(op, standardize=TRUE)
print(cserror)


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

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


## ------------------------------------------------------------
## 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")
veteran2 <- data.frame(lapply(veteran, factor))
veteran2$time <- veteran$time
veteran2$status <- veteran$status

## split the data into unbalanced train/test data (25/75)
## the train/test data have the same levels, but different labels
train <- sample(1:nrow(veteran2), round(nrow(veteran2) * .25))
summary(veteran2[train,])
summary(veteran2[-train,])

## train the forest and use this to predict on test data
o.grow <- rfsrc(Surv(time, status) ~ ., veteran2[train, ])
o.pred <- predict(o.grow, veteran2[-train , ])
```

```
print(o.grow)
print(o.pred)

## even harder ... factor level not previously encountered in training
veteran3 <- veteran2[1:3, ]
veteran3$celltype <- factor(c("newlevel", "1", "3"))
o2.pred <- predict(o.grow, veteran3)
print(o2.pred)
## the unusual level is treated like a missing value but is not removed
print(o2.pred$xvar)

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

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

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

## training step
set.seed(542899)
data(pbc, package = "randomForestSRC")
```

```
train <- sample(1:nrow(pbc), round(nrow(pbc) * 0.50))
pbc.out <- rfsrc(Surv(days, status) ~ .,  data=pbc[train, ])

## standard prediction 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$time.interest
matplot(Time, t(pbc.train$survival[1:3,]), ylab = "Survival", col = 1, type = "l")
matlines(Time, t(pbc.test$survival[1:3,]), col = 2)

## ------------------------------------------------------------
## ... just for _fun_ ...
## survival analysis using mixed multivariate outcome analysis
## compare the predicted value to RSF
## ------------------------------------------------------------

## train survival forest using pbc data
data(pbc, package = "randomForestSRC")
rsf.obj <- rfsrc(Surv(days, status) ~ ., pbc)
yvar <- rsf.obj$yvar

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

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

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

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

Description

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

Usage

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

Arguments

x	An object of class (rfsrc, grow), (rfsrc, synthetic), or (rfsrc, predict).
outcome.target	Character value for multivariate families specifying the target outcome to be used. The default is to use the first coordinate from the continuous outcomes (otherwise if none, the first coordinate from the categorical outcomes).
...	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.

Examples

```
options(rf.cores=2, mc.cores=2)
iris.obj <- rfsrc(Species ~., data = iris, ntree=10)
print(iris.obj)
```

---

quantreg.rfsrc *Quantile Regression Forests*

Description

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

Usage

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

Arguments

formula	A symbolic description of the model to be fit. Must be specified unless object is given.
data	Data frame containing the y-outcome and x-variables in the model. Must be specified unless object is given.
object	(Optional) A previously grown quantile regression forest.
newdata	(Optional) Test data frame used for prediction. Note that prediction on test data must always be done with the quantreg function and not the predict function. See example below.
method	Method used to calculate quantiles. Three methods are provided: (1) A variation of the method used in Meinshausen (2006) based on forest weight (method = "forest"); (2) The Greenwald-Khanna algorithm, suited for big data, and specified by any one of the following: "gk", "GK", "G-K", "g-k"; (3) The default method, method = "local", which uses the local adjusted cdf approach of Zhang et al. (2019). This does not rely on forest weights and is reasonably fast. See below for further discussion.
splitrule	The default action is local adaptive quantile regression splitting, but this can be over-ridden by the user. Not applicable to multivariate forests. See details below.
prob	Target quantile probabilities when training. If left unspecified, uses percentiles (1 through 99) for method = "forest", and for Greenwald-Khanna selects equally spaced percentiles optimized for accuracy (see below).
prob.epsilon	Greenwald-Khanna allowable error for quantile probabilities when training.
oob	Return OOB (out-of-bag) quantiles? If false, in-bag values are returned.
fast	Use fast random forests, rfsrc.fast, in place of rfsrc? Improves speed but may be less accurate.
maxn	Maximum number of unique y training values used when calculating the conditional density.
...	Further arguments to be passed to the rfsrc function used for fitting the quantile regression forest.

Details

The most common method for calculating RF quantiles uses the method described in Meinshausen (2006) using forest weights. The forest weights method employed here (specified using method="forest"), however differs in that quantiles are estimated using a weighted local cumulative distribution function estimator. For this reason, results may differ from Meinshausen (2006). Moreover, results may also differ as the default splitting rule uses local adaptive quantile regression splitting instead of CART regression mean squared splitting which was used by Meinshausen (2006). Note that local adaptive quantile regression splitting is not available for multivariate forests which reverts to the default multivariate composite splitting rule. In multivariate regression, users however do have the option to over-ride this using Mahalanobis splitting by setting splitrule="mahalanobis"

A second method for estimating quantiles uses the Greenwald-Khanna (2001) algorithm (invoked by method="gk", "GK", "G-K" or "g-k"). While this will not be as accurate as forest weights, the

high memory efficiency of Greenwald-Khanna makes it feasible to implement in big data settings unlike forest weights.

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

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

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

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

Finally, the default method, `method="local"`, implements the locally adjusted cdf estimator of Zhang et al. (2019). This does not use forest weights and is reasonably fast and can be used for large data. However, this relies on the assumption of homogeneity of the error distribution, i.e. that errors are iid and therefore have equal variance. While this is reasonably robust to departures of homogeneity, there are instances where this may perform poorly; see Zhang et al. (2019) for details. If hetereogeneity is suspected we recommend `method="forest"`.

Value

Returns the object `quantreg` containing quantiles for each of the requested probabilities (which can be conveniently extracted using `get.quantile`). Also contains the conditional density (and conditional cdf) for each case in the training data (or test data if provided) evaluated at each of the unique grow y-values. The conditional density can be used to calculate conditional moments, such as the mean and standard deviation. Use `get.quantile.stat` as a way to conveniently obtain these quantities.

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

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

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

Zhang H., Zimmerman J., Nettleton D. and Nordman D.J. (2019). Random forest prediction intervals. *The American Statistician*. 4:1-5.

See Also

rfsrc

Examples

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

## standard call
o <- quantreg(mpg ~ ., mtcars)

## extract conditional quantiles
print(get.quantile(o))
print(get.quantile(o, c(.25, .50, .75)))

## extract conditional mean and standard deviation
print(get.quantile.stat(o))

## standardized continuous rank probabiliy score (crps) performance
plot(get.quantile.crps(o), type = "l")


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

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

## extract test set quantiles and conditional statistics
print(get.quantile(o.tst))
print(get.quantile.stat(o.tst))


## ------------------------------------------------------------
## quantile regression for Boston Housing using forest method
## ------------------------------------------------------------

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

  ## quantile regression with mse splitting
  data(BostonHousing)
  o <- quantreg(medv ~ ., BostonHousing, method = "forest", nodesize = 1)

  ## standardized continuous rank probabiliy score (crps)
  plot(get.quantile.crps(o), type = "l")
```

```
  ## quantile regression plot
  plot.quantreg(o, .05, .95)
  plot.quantreg(o, .25, .75)

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

  ## (B) values expected under normality
  quant.stat <- get.quantile.stat(o)
  c.mean <- quant.stat$mean
  c.std <- quant.stat$std
  q.25.est <- c.mean + qnorm(.25) * c.std
  q.75.est <- c.mean + qnorm(.75) * c.std

  ## compare (A) and (B)
  print(head(data.frame(quant.dat[, -2],  q.25.est, q.75.est)))


}

## ------------------------------------------------------------
## multivariate mixed outcomes example
## quantiles are only returned for the continous outcomes
## ------------------------------------------------------------

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

plot.quantreg(o, m.target = "mpg")
plot.quantreg(o, m.target = "disp")

## ------------------------------------------------------------
## multivariate regression example using Mahalanobis splitting
## ------------------------------------------------------------

dta <- mtcars
o <- quantreg(cbind(mpg, disp) ~., data = dta, splitrule = "mahal")

plot.quantreg(o, m.target = "mpg")
plot.quantreg(o, m.target = "disp")

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

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

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

```
## extract "probabilities" = density values
qo.dens <- o$quantreg$density
yunq <- o$quantreg$yunq
colnames(qo.dens) <- yunq

## convert y to a factor
yvar <- factor(cut(o$yvar, c(-1, yunq), labels = yunq))

## confusion matrix
qo.confusion <- get.confusion(yvar, qo.dens)
print(qo.confusion)

## normalized Brier score
cat("Brier:", 100 * get.brier.error(yvar, qo.dens), "\n")


## ------------------------------------------------------------
## example of large data using Greenwald-Khanna algorithm
## ------------------------------------------------------------

## load the data and do quick and dirty imputation
data(housing, package = "randomForestSRC")
housing <- impute(SalePrice ~ ., housing,
         ntree = 50, nimpute = 1, splitrule = "random")

## Greenwald-Khanna algorithm
## request a small number of quantiles
o <- quantreg(SalePrice ~ ., housing, method = "gk",
        prob = (1:20) / 20, prob.epsilon = 1 / 20, ntree = 250)
plot.quantreg(o)

## ------------------------------------------------------------
## using mse splitting with local cdf method for large data
## ------------------------------------------------------------

## load the data and do quick and dirty imputation
data(housing, package = "randomForestSRC")
housing <- impute(SalePrice ~ ., housing,
         ntree = 50, nimpute = 1, splitrule = "random")

## use mse splitting and reduce number of trees
o <- quantreg(SalePrice ~ ., housing, splitrule = "mse", ntree = 250)
plot.quantreg(o)
```

---

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

Description

Fast OpenMP parallel computing of random forests (Breiman 2001) for regression, classification, survival analysis (Ishwaran et al. 2008), competing risks (Ishwaran et al. 2012), multivariate (Segal and Xiao 2011), unsupervised (Mantero and Ishwaran 2020), quantile regression (Meinhausen 2006, Zhang et al. 2019, Greenwald-Khanna 2001), and class imbalanced q-classification (O'Brien and Ishwaran 2019). Different splitting rules invoked under deterministic or random splitting (Geurts et al. 2006, Ishwaran 2015) are available for all families. Different types of variable importance (VIMP), holdout VIMP, as well as confidence regions (Ishwaran and Lu 2019) can be calculated for single and grouped variables. Minimal depth variable selection (Ishwaran et al. 2010, 2011). Fast interface for missing data imputation using a variety of different random forest methods (Tang and Ishwaran 2017).

Highlighted new items:

1. For computational speed, the default VIMP is no longer "permute" (Breiman-Cutler permutation importance) and has been switched to "anti" (`importance="anti"`, `importance=TRUE`; see below for details). Be aware in some situations, such as highly imbalanced classification, that permutation VIMP may perform better. Permutation VIMP is obtained using `importance="permute"`.
2. `save.memory` can be used for big data to save memory; especially useful for survival and competing risks.
3. Mahalanobis splitting for multivariate regression with correlated y-outcomes (`splitrule="mahalanobis"`). Now allows for a user specified covariance matrix.
4. Visualize trees on your Safari or Google Chrome browser (works for all families). See `get.tree`.

This is the main entry point to the **randomForestSRC** package. For more information about this package and OpenMP parallel processing, use the command `package?randomForestSRC`.

Usage

```
rfsrc(formula, data, ntree = 500,
  mtry = NULL, ytry = NULL,
  nodesize = NULL, nodedepth = NULL,
  splitrule = NULL, nsplit = NULL,
  importance = c(FALSE, TRUE, "none", "anti", "permute", "random"),
  block.size = if (any(is.element(as.character(importance),
                    c("none", "FALSE")))) NULL else 10,
  bootstrap = c("by.root", "none", "by.user"),
  samptype = c("swor", "swr"), samp = NULL, membership = FALSE,
  sampsize = if (samptype == "swor") function(x){x * .632} else function(x){x},
  na.action = c("na.omit", "na.impute"), nimpute = 1,
  ntime = 150, cause,
  perf.type = NULL,
  proximity = FALSE, distance = FALSE, forest.wt = FALSE,
  xvar.wt = NULL, yvar.wt = NULL, split.wt = NULL, case.wt  = NULL,
  case.depth = FALSE,
  forest = TRUE,
  save.memory = FALSE,
```

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

## convenient interface for growing a CART tree
rfsrc.cart(formula, data, ntree = 1, mtry = ncol(data), bootstrap = "none", ...)
```

Arguments

formula	Object of class 'formula' describing the model to fit. Interaction terms are not supported. If missing, unsupervised splitting is implemented.
data	Data frame containing the y-outcome and x-variables.
ntree	Number of trees.
mtry	Number of variables to possibly split at each node. Default is number of variables divided by 3 for regression. For all other families (including unsupervised settings), the square root of number of variables. Values are rounded up.
ytry	The number of randomly selected pseudo-outcomes for unsupervised families (see details below). Default is ytry=1.
nodesize	Minumum size of terminal node. The defaults are: survival (15), competing risk (15), regression (5), classification (1), mixed outcomes (3), unsupervised (3). It is recommended to experiment with different nodesize values.
nodedepth	Maximum depth to which a tree should be grown. Parameter is ignored by default.
splitrule	Splitting rule (see below).
nsplit	Non-negative integer specifying number of random splits for splitting a variable. When zero, all split values are used (deterministic splitting), which can be slower. By default 10 is used.
importance	Method for computing variable importance (VIMP); see below. Default action is importance="none" but VIMP can be recovered later using vimp or predict.
block.size	Determines how cumulative error rate is calculated. When NULL, it is calculated once for the entire forest, resulting in a flat cumulative error rate plot. To view the cumulative error rate for every nth tree, set block.size to an integer between 1 and ntree. If importance is requested, VIMP is calculated in blocks of size block.size, balancing between ensemble and tree VIMP. The default is 10 trees.
bootstrap	Bootstrap protocol. Default is by.root, which bootstraps the data by sampling with or without replacement (default is without replacement; see samptype below). If none, the data is not bootstrapped (OOB ensembles or prediction error cannot be returned). If by.user, the bootstrap specified by samp is used.
samptype	Type of bootstrap used when by.root is in effect. Choices are swor (sampling without replacement; the default) and swr (sampling with replacement).

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

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

sampsize — Specifies bootstrap size when by.root is in effect. For sampling without replacement, it is the requested size of the sample, defaulting to .632 times the sample size. For sampling with replacement, it is the sample size. Can also be specified as a number.

na.action — Action taken if the data contains NAs. Possible values are na.omit or na.impute. The default, na.omit, removes the entire record if any entry is NA. Selecting na.impute imputes the data (see below for details). Also see the function impute for fast imputation.

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

ntime — Integer value for survival to constrain ensemble calculations to an ntime grid of time points over the observed event times. If a vector of values with length greater than one is supplied, it is assumed these are the time points to be used (adjusted to match closest observed event times). Setting ntime to zero (or NULL) uses all observed event times.

cause — Integer value between 1 and J indicating the event of interest for splitting a node for competing risks, where J is the number of event types. If not specified, the default is a composite splitting rule averaging over all event types. Can also be a vector of non-negative weights of length J specifying weights for each event (a vector of ones reverts to the default composite split statistic). Estimates for all event types are returned regardless of how cause is specified.

perf.type — Optional character value specifying the metric used for predicted value, variable importance (VIMP), and error rate. Defaults to the family metric if not specified. perf.type="none" turns off performance entirely, useful for turning off C-error rate calculations for large survival data (which can be expensive). Values allowed for univariate/multivariate classification are: perf.type="misclass" (default), perf.type="brier", and perf.type="gmean".

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

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

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

xvar.wt — Vector of non-negative weights (does not have to sum to 1) representing the probability of selecting a variable for splitting. Default is uniform weights.

yvar.wt	Vector of non-negative weights (does not have to sum to 1) representing the probability of selecting a response variable in multivariate regression. Used when the number of responses are high-dimensional. See the example below.
split.wt	Vector of non-negative weights used for multiplying the split statistic for a variable. A large value encourages the node to split on a specific variable. Default is uniform weights.
case.wt	Vector of non-negative weights (does not have to sum to 1) for sampling cases. Observations with larger weights will be selected with higher probability in the bootstrap (or subsampled) samples. It is generally better to use real weights rather than integers. See the breast data example below illustrating its use for class imbalanced data.
case.depth	Return a matrix recording the depth at which a case first splits in a tree. Default is FALSE.
forest	Save key forest values? Used for prediction on new data and required by many of the package functions. Turn this off if you are only interested in training a forest.
save.memory	Save memory? Default is to store terminal node quantities used for prediction on test data. This yields rapid prediction but can be memory intensive for big data, especially competing risks and survival models. Turn this flag off in those cases.
var.used	Return statistics on number of times a variable split? Default is FALSE. Possible values are all.trees which returns total number of splits of each variable, and by.tree which returns a matrix of number a splits for each variable for each tree.
split.depth	Records the minimal depth for each variable. Default is FALSE. Possible values are all.trees which returns a matrix of the average minimal depth for a variable (columns) for a specific case (rows), and by.tree which returns a three-dimensional array recording minimal depth for a specific case (first dimension) for a variable (second dimension) for a specific tree (third dimension).
seed	Negative integer specifying seed for the random number generator.
do.trace	Number of seconds between updates to the user on approximate time to completion.
statistics	Should split statistics be returned? Values can be parsed using stat.split.
...	Further arguments passed to or from other methods.

Details

1. *Types of forests*

   There is no need to set the type of forest as the package automagically determines the underlying random forest requested from the type of outcome and the formula supplied. There are several possible scenarios:

   (a) Regression forests for continuous outcomes.

   (b) Classification forests for factor outcomes.

   (c) Multivariate forests for continuous and/or factor outcomes and for mixed (both type) of outcomes.

(d) Unsupervised forests when there is no outcome.
(e) Survival forests for right-censored survival.
(f) Competing risk survival forests for competing risk.

2. *Splitting*
   (a) Splitting rules are specified by the option `splitrule`.
   (b) For all families, pure random splitting can be invoked by setting `splitrule="random"`.
   (c) For all families, computational speed can be increased using randomized splitting invoked by the option `nsplit`. See Improving Computational Speed.
3. *Available splitting rules*
   - Regression analysis:
     (a) `splitrule="mse"` (default split rule): weighted mean-squared error splitting (Breiman et al. 1984, Chapter 8.4).
     (b) `splitrule="quantile.regr"`: quantile regression splitting via the "check-loss" function. Requires specifying the target quantiles. See `quantreg.rfsrc` for further details.
     (c) `la.quantile.regr`: local adaptive quantile regression splitting. See `quantreg.rfsrc`.
   - Classification analysis:
     (a) `splitrule="gini"` (default splitrule): Gini index splitting (Breiman et al. 1984, Chapter 4.3).
     (b) `splitrule="auc"`: AUC (area under the ROC curve) splitting for both two-class and multiclass setttings. AUC splitting is appropriate for imbalanced data. See `imbalanced` for more information.
     (c) `splitrule="entropy"`: entropy splitting (Breiman et al. 1984, Chapter 2.5, 4.3).
   - Survival analysis:
     (a) `splitrule="logrank"` (default splitrule): log-rank splitting (Segal, 1988; Leblanc and Crowley, 1993).
     (b) `splitrule="bs.gradient"`: gradient-based (global non-quantile) brier score splitting. The time horizon used for the Brier score is set to the 90th percentile of the observed event times. This can be over-ridden by the option `prob`, which must be a value between 0 and 1 (set to .90 by default).
     (c) `splitrule="logrankscore"`: log-rank score splitting (Hothorn and Lausen, 2003).
   - Competing risk analysis (for details see Ishwaran et al., 2014):
     (a) `splitrule="logrankCR"` (default splitrule): modified weighted log-rank splitting rule modeled after Gray's test (Gray, 1988). Use this to find *all* variables that are informative and when the goal is long term prediction.
     (b) `splitrule="logrank"`: 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. Use this to find variables that affect a *specific* cause of interest and when the goal is a targeted analysis of a specific cause. However in order for this to be effective, remember to set the `cause` option to the targeted cause of interest. See examples below.
   - Multivariate analysis:
     (a) Default is the multivariate normalized composite split rule using mean-squared error and Gini index (Tang and Ishwaran, 2017).

(b) `splitrule="mahalanobis"`: Mahalanobis splitting that adjusts for correlation (also allows for a user specified covariance matrix, see example below). Only works for multivariate regression (all outcomes must be real).

- Unsupervised analysis: In settings where there is no outcome, unsupervised splitting that uses pseudo-outcomes is applied using the default multivariate splitting rule (see below for details) Also see `sidClustering` for a more sophisticated method for unsupervised analysis (Mantero and Ishwaran, 2020).
- 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.

4. *Improving computational speed*

   See the function `rfsrc.fast` for a fast implementation of `rfsrc`. Key methods for increasing speed are as follows:

   - *Nodesize*
     Increasing `nodesize` has the greatest effect in speeding calculations. In some big data settings this can also lead to better prediction performance.
   - *Save memory*
     Use option `save.memory="TRUE"` for big data competing risk and survival models. By default the package stores terminal node quantities to be used in prediction for test data but this can be memory intensive for big data.
   - *Block size*
     Make sure `block.size="NULL"` (or set to number of trees) so that the cumulative error is calculated only once.
   - *Turn off performace*
     The C-error rate calculation can be very expensive for big survival data. Set `perf.type="none"` to turn this off and all other performance calculations (then consider using the function `get.brier.survival` as a fast way to get survival performance).
     `perf.type="none"` applies to all other families as well.
   - *Randomized splitting rules*
     Set `nsplit` to a small non-zero integer value. Then a maximum of `nsplit` split points are chosen randomly for each of the candidate splitting variables when splitting a tree node, thus significantly reducing computational costs.
     For more details about randomized splitting see Loh and Shih (1997), Dietterich (2000), and Lin and Jeon (2006). Geurts et al. (2006) introduced extremely randomized trees using the extra-trees algorithm. This algorithm corresponds to `nsplit=1`. In our experience however this may be too low for general use (Ishwaran, 2015).

For completely randomized (pure random) splitting use `splitrule="random"`. In pure splitting, nodes are split by randomly selecting a variable and randomly selecting its split point (Cutler and Zhao, 2001).

- *Subsampling*
  Reduce the size of the bootstrap using `sampsize` and `samptype`. See `rfsrc.fast` for a fast forest implementation using subsampling.
- *Unique time points*
  Setting `ntime` to a reasonably small value such as 50 constrains survival ensemble calculations to a restricted grid of time points and significantly improves computational times.
- *Large number of variables*
  Try filtering variables ahead of time. Make sure not to request VIMP (variable importance can always be recovered later using `vimp` or `predict`). Also if variable selection is desired, but is too slow, consider using `max.subtree` which calculates minimal depth, a measure of the depth that a variable splits, and yields fast variable selection (Ishwaran, 2010).

5. *Prediction Error*

   Prediction error is calculated using OOB data. The metric used is mean-squared-error for regression, and misclassification error for classification. A normalized Brier score (relative to a coin-toss) and the AUC (area under the ROC curve) is also provided upon printing a classification forest. Performance for Brier score can be specified using `perf.type="brier"`. G-mean performance is also available, see the function `imbalanced` for more details.

   For survival, prediction error is measured by the C-error rate defined as 1-C, where C is Harrell's (Harrell et al., 1982) concordance index. The C-error is between 0 and 1, and measures error in ranking (classifying) 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 `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)*

   VIMP is calculated using OOB data in several ways. `importance="permute"` yields permutation VIMP (Breiman-Cutler importance) by permuting OOB cases. `importance="random"` uses random left/right assignments whenever a split is encountered for the target variable. The default `importance="anti"` (equivalent to `importance=TRUE`) assigns cases to the anti (opposite) split.

   VIMP depends upon `block.size`, an integer value between 1 and `ntree`, specifying number of trees in a block used for VIMP. When `block.size=1`, VIMP is calculated for each tree. When `block.size="ntree"`, VIMP is calculated for the entire forest by comparing the perturbed OOB forest ensemble (using all trees) to the unperturbed OOB forest ensemble (using all trees). This yields ensemble VIMP, which does not measure the tree average effect of a variable, but rather its overall forest effect.

   A useful compromise between tree VIMP and ensemble VIMP can be obtained by setting `block.size` to a value between 1 and `ntree`. Smaller values generally gives better accuracy, however computational times will be higher because VIMP is calculated over more blocks. However, see `imbalanced` for imbalanced classification data where larger `block.size` often works better (O'Brien and Ishwaran, 2019).

See vimp for a user interface for extracting VIMP and subsampling for calculating confidence intervals for VIMP.

Also see holdout.vimp for holdout VIMP, which calculates importance by holding out variables. This is more conservative, but with good false discovery properties.

For classification, VIMP is returned as a matrix with J+1 columns where J is the number of classes. The first column "all" is the unconditional VIMP, while the remaining columns are conditional VIMP calculated using only OOB cases with the class label.

7. *Multivariate Forests*

   Multivariate forests can be specified in two ways:

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

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

   By default, a multivariate normalized composite splitting rule is used to split nodes (for multivariate regression, users have the option to use Mahalanobis splitting).

   The nature of the outcomes informs the code as to what type of multivariate forest is grown; i.e. whether it is real-valued, categorical, or a combination of both (mixed). Performance measures (when requested) are returned for all outcomes.

   Helper functions get.mv.formula, get.mv.predicted, get.mv.error can be used for defining the multivariate forest formula and extracting predicted values (all outcomes) and VIMP (all variables, all outcomes; assuming importance was requested in the call). The latter two functions also work for univariate (regular) forests. Both functions return standardized values (dividing by the variance for regression, or multiplying by 100, otherwise) using option standardize="TRUE".

8. *Unsupervised Forests and sidClustering*

   See sidClustering sidClustering for a more sophisticated method for unsupervised analysis.

   Otherwise a more direct (but naive) way to proceed is to use the unsupervised splitting rule. The following are equivalent ways to grow an unsupervised forest via unsupervised splitting:

   rfsrc(data = my.data)

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

   In unsupervised mode, features take turns acting as target y-outcomes and x-variables for splitting. Specifically, mtry x-variables are randomly selected for splitting the node. Then for each mtry feature, ytry variables are selected from the remaining features to act as the target pseduo-outcomes. Splitting uses the multivariate normalized composite splitting rule.

   The default value of ytry is 1 but can be increased. As illustration, the following equivalent unsupervised calls set mtry=10 and ytry=5:

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

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

   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 outcome 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).

(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*

`na.action="na.impute"` imputes missing data (both x and y-variables) using the missing data algorithm of Ishwaran et al. (2008). But also see the `impute` for an alternate way to do fast and accurate imputation.

The missing data algorithm can be iterated by setting `nimpute` to a positive integer greater than 1. When 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. A side effect of iteration is that missing values in the returned objects `xvar`, `yvar` are replaced by imputed values. In other words the incoming data is overlaid with the missing data. Also, performance measures such as error rates and VIMP become optimistically biased.

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.

11. *Allowable data types and factors*

Data types must be real valued, integer, factor or logical – however all except factors are coerced and treated as if real valued. For ordered x-variable factors, splits are similar to real valued variables. For unordered factors, 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 number of splits attempted is not greater than number of cases in a node or the value of `nsplit`.

For coherence, an immutable map is applied to each factor that ensures factor levels in the training data are consistent with the factor levels in any subsequent test data. This map is applied to each factor before and after the native C library is executed. Because of this, if all x-variables all factors, then computational time will be long in high dimensional problems. Consider converting factors to real if this is the case.

Value

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

`call`	The original call to `rfsrc`.
`family`	The family used in the analysis.
`n`	Sample size of the data (depends upon `NA`'s, see `na.action`).
`ntree`	Number of trees grown.
`mtry`	Number of variables randomly selected for splitting at each node.

nodesize	Minimum size of terminal nodes.
nodedepth	Maximum depth allowed for a tree.
splitrule	Splitting rule used.
nsplit	Number of randomly selected split points.
yvar	y-outcome values.
yvar.names	A character vector of the y-outcome names.
xvar	Data frame of x-variables.
xvar.names	A character vector of the x-variable names.
xvar.wt	Vector of non-negative weights specifying the probability used to select a variable for splitting a node.
split.wt	Vector of non-negative weights specifying multiplier by which the split statistic for a covariate is adjusted.
cause.wt	Vector of weights used for the composite competing risk splitting rule.
leaf.count	Number of terminal nodes for each tree in the forest. Vector of length ntree. A value of zero indicates a rejected tree (can occur when imputing missing data). Values of one indicate tree stumps.
proximity	Proximity matrix recording the frequency of pairs of data points occur within the same terminal node.
forest	If forest=TRUE, the forest object is returned. This object is used for prediction with new test data sets and is required for other R-wrappers.
forest.wt	Forest weight matrix.
membership	Matrix recording terminal node membership where each column records node mebership for a case for a tree (rows).
splitrule	Splitting rule used.
inbag	Matrix recording inbag membership where each column contains the number of times that a case appears in the bootstrap sample for a tree (rows).
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-outcomes, 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.
err.block.rate	When importance=TRUE, vector of the cumulative error rate for each ensemble block comprised of block.size trees. So with block.size = 10, entries are the cumulative error rate for the first 10 trees, the first 20 trees, 30 trees, and so on. As another exmple, if block.size = 1, entries are the error rate for each tree.
importance	Variable importance (VIMP) for each x-variable.

`predicted`	In-bag predicted value.
`predicted.oob`	OOB predicted value.
`++++++++`	for classification settings, additionally ++++++++
`class`	In-bag predicted class labels.
`class.oob`	OOB predicted class labels.
`++++++++`	for multivariate settings, additionally ++++++++
`regrOutput`	List containing performance values for multivariate regression outcomes (applies only in multivariate settings).
`clasOutput`	List containing performance values for multivariate categorical (factor) outcomes (applies only in multivariate settings).
`++++++++`	for survival settings, additionally ++++++++
`survival`	In-bag survival function.
`survival.oob`	OOB survival function.
`chf`	In-bag cumulative hazard function (CHF).
`chf.oob`	OOB CHF.
`time.interest`	Ordered unique death times.
`ndead`	Number of deaths.
`++++++++`	for competing risks, additionally ++++++++
`chf`	In-bag cause-specific cumulative hazard function (CSCHF) for each event.
`chf.oob`	OOB CSCHF.
`cif`	In-bag cumulative incidence function (CIF) for each event.
`cif.oob`	OOB CIF.

Note

Values returned depend heavily on the family. In particular, predicted values from the forest (`predicted` and `predicted.oob`) are as follows:

1. For regression, a vector of predicted y-outcomes.
2. For classification, a matrix with columns containing the estimated class probability for each class. Performance values and VIMP for classification are reported as a matrix with J+1 columns where J is the number of classes. The first column "all" is the unconditional value for performance (VIMP), while the remaining columns are performance (VIMP) conditioned on cases corresponding to that class label.

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

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

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.

Dietterich, T. G. (2000). An experimental comparison of three methods for constructing ensembles of decision trees: bagging, boosting, and randomization. *Machine Learning*, 40, 139-157.

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

Geurts, P., Ernst, D. and Wehenkel, L., (2006). Extremely randomized trees. *Machine learning*, 63(1):3-42.

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

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

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

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

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

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

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

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

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

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

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

Lin, Y. and Jeon, Y. (2006). Random forests and adaptive nearest neighbors. *J. Amer. Statist. Assoc.*, 101(474), 578-590.

Lu M., Sadiq S., Feaster D.J. and Ishwaran H. (2018). Estimating individual treatment effect in observational data using random forest methods. *J. Comp. Graph. Statist*, 27(1), 209-219

Ishwaran H. and Lu M. (2019). Standard errors and confidence intervals for variable importance in random forest regression, classification, and survival. *Statistics in Medicine*, 38, 558-582.

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

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

Mantero A. and Ishwaran H. (2021). Unsupervised random forests. *Statistical Analysis and Data Mining*, 14(2):144-167.

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

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

O'Brien R. and Ishwaran H. (2019). A random forests quantile classifier for class imbalanced data. *Pattern Recognition*, 90, 232-249

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

Segal M.R. and Xiao Y. Multivariate random forests. (2011). *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*. 1(1):80-87.

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

Zhang H., Zimmerman J., Nettleton D. and Nordman D.J. (2019). Random forest prediction intervals. *The American Statistician*. 4:1-5.

See Also

`find.interaction.rfsrc`,

`get.tree.rfsrc`,

`holdout.vimp.rfsrc`,

`imbalanced.rfsrc`, `impute.rfsrc`,

`max.subtree.rfsrc`,

partial.rfsrc, plot.competing.risk.rfsrc, plot.rfsrc, plot.survival.rfsrc, plot.variable.rfsrc, predict.rfsrc, print.rfsrc,

quantreg.rfsrc,

rfsrc, rfsrc.anonymous, rfsrc.cart, rfsrc.fast,

sidClustering.rfsrc,

stat.split.rfsrc, subsample.rfsrc, synthetic.rfsrc,

tune.rfsrc,

var.select.rfsrc, vimp.rfsrc

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, block.size = 1)

## plot tree number 3
plot(get.tree(v.obj, 3))

## print results of trained forest
print(v.obj)

## plot results of trained forest
plot(v.obj)

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

## plot survival curves for first 10 individuals
## using function "plot.survival"
plot.survival(v.obj, subset = 1:10)

## obtain Brier score using KM and RSF censoring distribution estimators
bs.km <- get.brier.survival(v.obj, cens.model = "km")$brier.score
bs.rsf <- get.brier.survival(v.obj, cens.model = "rfsrc")$brier.score

## plot the brier score
plot(bs.km, type = "s", col = 2)
lines(bs.rsf, type ="s", col = 4)
legend("topright", legend = c("cens.model = km", "cens.model = rfsrc"), fill = c(2,4))

## plot CRPS (continuous rank probability score) as function of time
## here's how to calculate the CRPS for every time point
trapz <- randomForestSRC:::trapz
time <- v.obj$time.interest
```

```
crps.km <- sapply(1:length(time), function(j) {
  trapz(time[1:j], bs.km[1:j, 2] / diff(range(time[1:j])))
})
crps.rsf <- sapply(1:length(time), function(j) {
  trapz(time[1:j], bs.rsf[1:j, 2] / diff(range(time[1:j])))
})
plot(time, crps.km, ylab = "CRPS", type = "s", col = 2)
lines(time, crps.rsf, type ="s", col = 4)
legend("bottomright", legend=c("cens.model = km", "cens.model = rfsrc"), fill=c(2,4))


## fast nodesize optimization for veteran data
## optimal nodesize in survival is larger than other families
## see the function "tune" for more examples
tune.nodesize(Surv(time,status) ~ ., veteran)


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


## save.memory example for survival
## growing many deep trees creates memory issue without this option!
data(pbc, package = "randomForestSRC")
print(rfsrc(Surv(days, status) ~ ., pbc, splitrule = "random",
            ntree = 25000, nodesize = 1, save.memory = TRUE))



##------------------------------------------------------------
## trees can be plotted for any family
## see get.tree for details and more examples
##------------------------------------------------------------

## survival where factors have many levels
data(veteran, package = "randomForestSRC")
vd <- veteran
vd$celltype=factor(vd$celltype)
vd$diagtime=factor(vd$diagtime)
vd.obj <- rfsrc(Surv(time,status)~., vd, ntree = 100, nodesize = 5)
plot(get.tree(vd.obj, 3))

## classification
iris.obj <- rfsrc(Species ~., data = iris)
plot(get.tree(iris.obj, 25, class.type = "bayes"))
plot(get.tree(iris.obj, 25, target = "setosa"))
plot(get.tree(iris.obj, 25, target = "versicolor"))
plot(get.tree(iris.obj, 25, target = "virginica"))

## ------------------------------------------------------------
## simple example of VIMP using iris classification
```

```
## ------------------------------------------------------------

## directly from trained forest
print(rfsrc(Species~.,iris,importance=TRUE)$importance)

## VIMP (and performance) use misclassification error by default
## but brier prediction error can be requested
print(rfsrc(Species~.,iris,importance=TRUE,perf.type="brier")$importance)

## example using vimp function (see vimp help file for details)
iris.obj <- rfsrc(Species ~., data = iris)
print(vimp(iris.obj)$importance)
print(vimp(iris.obj,perf.type="brier")$importance)

## example using hold out vimp (see holdout.vimp help file for details)
print(holdout.vimp(Species~.,iris)$importance)
print(holdout.vimp(Species~.,iris,perf.type="brier")$importance)

## ------------------------------------------------------------
## confidence interval for vimp using subsampling
## compare with holdout vimp
## ------------------------------------------------------------

## new York air quality measurements
o <- rfsrc(Ozone ~ ., data = airquality)
so <- subsample(o)
plot(so)

## compare with holdout vimp
print(holdout.vimp(Ozone ~ ., data = airquality)$importance)


##------------------------------------------------------------
## example of imputation in survival analysis
##------------------------------------------------------------

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

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

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

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

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

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

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

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

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

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

##------------------------------------------------------------
## competing risks
##------------------------------------------------------------
```

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

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


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

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

  ## modified Gray's weighted log-rank splitting
  ## (equivalent to cause=c(1,1) and splitrule="logrankCR")
  pbc.cr <- rfsrc(Surv(time, status) ~ ., pbc)

  ## log-rank cause-1 specific splitting and targeted VIMP for cause 1
  pbc.log1 <- rfsrc(Surv(time, status) ~ ., pbc,
              splitrule = "logrankCR", cause = c(1,0), importance = TRUE)

  ## log-rank cause-2 specific splitting and targeted VIMP for cause 2
  pbc.log2 <- rfsrc(Surv(time, status) ~ ., pbc,
              splitrule = "logrankCR", cause = c(0,1), importance = TRUE)

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

}

## ------------------------------------------------------------
```

```
## 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)

## ------------------------------------------------------------
## regression with custom bootstrap
## ------------------------------------------------------------

ntree <- 25
n <- nrow(mtcars)
s.size <- n / 2
swr <- TRUE
samp <- randomForestSRC:::make.sample(ntree, n, s.size, swr)
o <- rfsrc(mpg ~ ., mtcars, bootstrap = "by.user", samp = samp)

## ------------------------------------------------------------
## classification analysis
## ------------------------------------------------------------

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

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

## ------------------------------------------------------------
## big data set, reduce number of variables using simple method
## ------------------------------------------------------------

## use Iowa housing data set
data(housing, package = "randomForestSRC")

## original data contains lots of missing data, use fast imputation
## however see impute for other methods
housing2 <- impute(data = housing, fast = TRUE)

## run shallow trees to find variables that split any tree
xvar.used <- rfsrc(SalePrice ~., housing2, ntree = 250, nodedepth = 4,
                   var.used="all.trees", mtry = Inf, nsplit = 100)$var.used

## now fit forest using filtered variables
xvar.keep  <- names(xvar.used)[xvar.used >= 1]
o <- rfsrc(SalePrice~., housing2[, c("SalePrice", xvar.keep)])
```

```
print(o)

## ------------------------------------------------------------
## imbalanced classification data
## see the "imbalanced" function for further details
##
## (a) use balanced random forests with undersampling of the majority class
## Specifically let n0, n1 be sample sizes for majority, minority
## cases.  We sample 2 x n1 cases with majority, minority cases chosen
## with probabilities n1/n, n0/n where n=n0+n1
##
## (b) balanced random forests using "imbalanced"
##
## (c) q-classifier (RFQ) using "imbalanced"
##
## ------------------------------------------------------------

## Wisconsin breast cancer example
data(breast, package = "randomForestSRC")
breast <- na.omit(breast)

## balanced random forests - brute force
y <- breast$status
obdirect <- rfsrc(status ~ ., data = breast, nsplit = 10,
            case.wt = randomForestSRC:::make.wt(y),
            sampsize = randomForestSRC:::make.size(y))
print(obdirect)
print(get.imbalanced.performance(obdirect))

## balanced random forests - using "imbalanced"
ob <- imbalanced(status ~ ., data = breast, method = "brf")
print(ob)
print(get.imbalanced.performance(ob))

## q-classifier (RFQ) - using "imbalanced"
oq <- imbalanced(status ~ ., data = breast)
print(oq)
print(get.imbalanced.performance(oq))

## q-classifier (RFQ) - with auc splitting
oqauc <- imbalanced(status ~ ., data = breast, splitrule = "auc")
print(oqauc)
print(get.imbalanced.performance(oqauc))

## ------------------------------------------------------------
## unsupervised analysis
## ------------------------------------------------------------

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

```
## illustration of sidClustering for the mtcars data
## see sidClustering for more details
mtcars.sid <- sidClustering(mtcars, k = 1:10)
print(split(mtcars, mtcars.sid$cl[, 3]))
print(split(mtcars, mtcars.sid$cl[, 10]))


## ------------------------------------------------------------
## bivariate regression using Mahalanobis splitting
## also illustrates user specified covariance matrix
## ------------------------------------------------------------

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

  ## load boston housing data, specify the bivariate regression
  data(BostonHousing)
  f <- formula("Multivar(lstat, nox) ~.")

  ## Mahalanobis splitting
  bh.mreg <- rfsrc(f, BostonHousing, importance = TRUE, splitrule = "mahal")

  ## performance error and vimp
  vmp <- get.mv.vimp(bh.mreg)
  pred <- get.mv.predicted(bh.mreg)

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

  ## same analysis, but with user specified covariance matrix
  sigma <- cov(BostonHousing[, c("lstat","nox")])
  bh.mreg2 <- rfsrc(f, BostonHousing, splitrule = "mahal", sigma = sigma)

}

## ------------------------------------------------------------
## multivariate mixed forests (nutrigenomic study)
## study effects of diet, lipids and gene expression for mice
## diet, genotype and lipids used as the multivariate y
## genes used for the x features
## ------------------------------------------------------------

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

## assemble the multivariate y data
ydta <- data.frame(diet = nutrigenomic$diet,
                   genotype = nutrigenomic$genotype,
                   nutrigenomic$lipids)

## multivariate mixed forest call
## uses "get.mv.formula" for conveniently setting formula
mv.obj <- rfsrc(get.mv.formula(colnames(ydta)),
```

```
               data.frame(ydta, nutrigenomic$genes),
importance=TRUE, nsplit = 10)

## print results for diet and genotype y values
print(mv.obj, outcome.target = "diet")
print(mv.obj, outcome.target = "genotype")

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

## plot standardized VIMP for diet, genotype and lipid for each gene
boxplot(t(svimp), col = "bisque", cex.axis = .7, las = 2,
        outline = FALSE,
        ylab = "standardized VIMP",
        main = "diet/genotype/lipid VIMP for each gene")


## ------------------------------------------------------------
## illustrates yvar.wt which sets the probability of selecting
## the response variables in multivariate regression
## ------------------------------------------------------------

## use mtcars: add fake responses
mult.mtcars  <- cbind(mtcars, mtcars$mpg, mtcars$mpg)
names(mult.mtcars) = c(names(mtcars), "mpg2", "mpg3")

## noise up the fake responses
mult.mtcars$mpg2 <- sample(mtcars$mpg)
mult.mtcars$mpg3 <- sample(mtcars$mpg)

formula = as.formula(Multivar(mpg, mpg2, mpg3)  ~ .)

## select 2 of the 3 responses randomly at each split with an associated weight vector.
## choose the noisy y responses which should degrade performance
yvar.wt = c(0.000001, 0.5, 0.5)
ytry = 2

mult.grow <- rfsrc(formula = formula, data = mult.mtcars, ytry = ytry, yvar.wt = yvar.wt)

print(mult.grow)
print(get.mv.error(mult.grow))

## Also, compare the following two results, as they should be similar:
yvar.wt = c(1.0, 00000.1, 00000.1)
ytry = 1

result1 = rfsrc(formula = formula, data = mult.mtcars, ytry = ytry, yvar.wt = yvar.wt)
result2 = rfsrc(mpg ~ ., mtcars)

print(get.mv.error(result1))
print(get.mv.error(result2))

## ------------------------------------------------------------
```

```
## custom splitting using the pre-coded examples
## ------------------------------------------------------------

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

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

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

`rfsrc.anonymous` *Anonymous Random Forests*

Description

Anonymous random forests applies random forests but is carefully modified so as not to save the original training data. This allows users to share their forest with other researchers but without having to share their original data.

Usage

```
rfsrc.anonymous(formula, data, forest = TRUE, ...)
```

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.
`forest`	Should the forest object be returned? Used for prediction on new data and required by many of the package functions.
`...`	Further arguments as in `rfsrc`. See the `rfsrc` help file for details.

Details

Calls `rfsrc` and returns an object with the training data removed so that users can share their forest while maintaining privacy of their data.

In order to predict on test data, it is however necessary for certain minimal information to be saved from the training data. This includes the names of the original variables, and if factor variables are present, the levels of the factors. The mean value and maximal class value for real and factor variables in the training data are also stored for the purposes of imputation on test data (see below). The topology of grow trees is also saved, which includes among other things, the split values used for splitting tree nodes.

For the most privacy, we recommend that variable names be made non-identifiable and that data be coerced to real values. If factors are required, the user should consider using non-identifiable factor levels. However, in all cases, it is the users responsibility to de-identify their data and to check that data privacy holds. We provide NO GUARANTEES of this.

Missing data is especially delicate with anonymous forests. Training data cannot be imputed and the option `na.action="na.impute"` simply reverts to `na.action="na.omit"`. Therefore if you have training data with missing values consider using pre-imputing the data using `impute`. It is however possible to impute on test data. The option `na.action="na.impute"` in the prediction call triggers a rough and fast imputation method where the value of missing test data are replaced by the mean (or maximal class) value from the training data. A second option `na.action="na.random"` uses a fast random imputation method.

In general, it is important to keep in mind that while anonymous forests tries to play nice with other functions in the package, it only works with calls that do not specifically require training data.

Value

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

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

See Also

`rfsrc`

Examples

```
## ------------------------------------------------------------
## regression
## ------------------------------------------------------------
print(rfsrc.anonymous(mpg ~ ., mtcars))

## ------------------------------------------------------------
## plot anonymous regression tree (using get.tree)
## TBD CURRENTLY NOT IMPLEMENTED
## ------------------------------------------------------------
## plot(get.tree(rfsrc.anonymous(mpg ~ ., mtcars), 10))

## ------------------------------------------------------------
## classification
## ------------------------------------------------------------
print(rfsrc.anonymous(Species ~ ., iris))

## ------------------------------------------------------------
## survival
## ------------------------------------------------------------
data(veteran, package = "randomForestSRC")
print(rfsrc.anonymous(Surv(time, status) ~ ., data = veteran))
```

```
## ------------------------------------------------------------
## competing risks
## ------------------------------------------------------------
data(wihs, package = "randomForestSRC")
print(rfsrc.anonymous(Surv(time, status) ~ ., wihs, ntree = 100))

## ------------------------------------------------------------
## unsupervised forests
## ------------------------------------------------------------
print(rfsrc.anonymous(data = iris))

## ------------------------------------------------------------
## multivariate regression
## ------------------------------------------------------------
print(rfsrc.anonymous(Multivar(mpg, cyl) ~., data = mtcars))

## ------------------------------------------------------------
## prediction on test data with missing values using pbc data
## cases 1 to 312 have no missing values
## cases 313 to 418 having missing values
## ------------------------------------------------------------
data(pbc, package = "randomForestSRC")
pbc.obj <- rfsrc.anonymous(Surv(days, status) ~ ., pbc)
print(pbc.obj)

## mean value imputation
print(predict(pbc.obj, pbc[-(1:312),], na.action = "na.impute"))

## random imputation
print(predict(pbc.obj, pbc[-(1:312),], na.action = "na.random"))

## ------------------------------------------------------------
## train/test setting but tricky because 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 train/test data (25/75)
# the train/test data have the same levels, but different labels
train <- sample(1:nrow(veteran), round(nrow(veteran) * .5))
summary(veteran.factor[train, ])
summary(veteran.factor[-train, ])

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

`rfsrc.fast` *Fast Random Forests*

Description

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

Usage

```
rfsrc.fast(formula, data,
  ntree = 500,
  nsplit = 10,
  bootstrap = "by.root",
  sampsize = function(x){min(x * .632, max(150, x ^ (3/4)))},
  samptype = "swor",
  samp = NULL,
  ntime = 50,
  forest = FALSE,
  save.memory = TRUE,
  ...)
```

Arguments

`formula`	Model to be fit. If missing, unsupervised splitting is implemented.
`data`	Data frame containing the y-outcome and x-variables.
`ntree`	Number of trees.
`nsplit`	Non-negative integer value specifying number of random split points used to split a node (deterministic splitting corresponds to the value zero and can be slower).
`bootstrap`	Bootstrap protocol used in growing a tree.
`sampsize`	Function specifying size of subsampled data. Can also be a number.
`samptype`	Type of bootstrap used.
`samp`	Bootstrap specification when `"by.user"` is used.
`ntime`	Integer value used for survival to constrain ensemble calculations to a grid of `ntime` time points.
`forest`	Save key forest values? Turn this on if you want prediction on test data.
`save.memory`	Save memory? Setting this to `FALSE` stores terminal node quantities used for prediction on test data. This yields rapid prediction but can be memory intensive for big data, especially competing risks and survival models.
`...`	Further arguments to be passed to `rfsrc`.

Details

Calls rfsrc by choosing options (like subsampling) to encourage computational speeds. This will provide a good approximation but will not be as good as default settings of rfsrc.

Value

An object of class (rfsrc, grow).

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

See Also

rfsrc

Examples

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

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

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

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

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

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

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

## ------------------------------------------------------------
## grow fast random survival forests without C-calculation
## use brier score to assess model performance
## compare pure random splitting to logrank splitting
```

```
## ------------------------------------------------------------

data(peakVO2, package = "randomForestSRC")
f <- as.formula(Surv(ttodead, died)~.)
o1 <- rfsrc.fast(f, peakVO2, perf.type = "none")
o2 <- rfsrc.fast(f, peakVO2, perf.type = "none", splitrule = "random")
bs1 <- get.brier.survival(o1, cens.model = "km")
bs2 <- get.brier.survival(o2, cens.model = "km")
plot(bs2$brier.score, type = "s", col = 2)
lines(bs1$brier.score, type = "s", col = 4)
legend("bottomright", legend = c("random", "logrank"), fill = c(2,4))

## ------------------------------------------------------------
## competing risks
## ------------------------------------------------------------

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

## ------------------------------------------------------------
## class imbalanced data using gmean performance
## ------------------------------------------------------------

data(breast, package = "randomForestSRC")
breast <- na.omit(breast)
f <- as.formula(status ~ .)
o <- rfsrc.fast(f, breast, perf.type = "gmean")
print(o)

## ------------------------------------------------------------
## class imbalanced data using random forests quantile-classifer (RFQ)
## fast=TRUE => rfsrc.fast
## see imbalanced function for further details
## ------------------------------------------------------------

data(breast, package = "randomForestSRC")
breast <- na.omit(breast)
f <- as.formula(status ~ .)
o <- imbalanced(f, breast, fast = TRUE)
print(o)
```

`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

---

sidClustering.rfsrc *sidClustering using SID (Staggered Interaction Data) for Unsupervised Clustering*

Description

Clustering of unsupervised data using SID (Mantero and Ishwaran, 2020). Also implements the artificial two-class approach of Breiman (2003).

Usage

```
## S3 method for class 'rfsrc'
sidClustering(data,
  method = "sid",
  k = NULL,
  reduce = TRUE,
  ntree = 500,
  ntree.reduce = function(p, vtry){100 * p / vtry},
  fast = FALSE,
  x.no.sid = NULL,
  use.sid.for.x = TRUE,
  x.only = NULL, y.only = NULL,
  dist.sharpen = TRUE, ...)
```

Arguments

data	Data frame containing the unsupervised data.
method	The method used for unsupervised clustering. Default is "sid" which implements sidClustering using SID (Staggered Interaction Data; see Mantero and Ishwaran, 2020). A second approach transforms the unsupervised learning problem into a two-class supervised problem (Breiman, 2003) using artificial data created using mode 1 or mode 2 of Shi-Horvath (2006). This approach is specified

by any one of the following: "sh", "SH", "sh1", "SH1" for mode 1, or "sh2", "SH2" for mode 2. Finally, a third approach is a plain vanilla method where the data are used both as features and response with splitting implemented using the multivariate splitting rule. This is faster than sidClustering but potentially less accurate. This method is specified using "unsupv".

`k` Requested number of clusters. Can be a number or a vector. If a fixed number, returns a vector recording clustering of data. If a vector, returns a matrix of clusters with each column recording the clustering of the data for the specified number of clusters.

`reduce` Apply dimension reduction? Uses holdout vimp which is computationally intensive and conservative but has good false discovery properties. Only applies to `method="sid"`.

`ntree` Number of trees used by sidClustering in the main analysis.

`ntree.reduce` Number of trees used by holdout vimp in the reduction step. See `holdout.vimp` for details.

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

`x.no.sid` Features not to be "sid-ified": meaning that these features are to be included in the final design matrix without SID processing. Can be either a data frame (should not overlap with `data`), or a character vector containing the names of features from the original data that the user wishes to protect from sidification. Applies only to `method="sid"`.

`use.sid.for.x` If FALSE, reverses features and outcomes in the SID analysis. Thus, staggered interactions are used for the outcomes rather than staggered features. This is much slower and is generally much less effective. This option is only retained for legacy reasons. Applies only to `method="sid"`.

`x.only` Use only these variables for the features. Applies only to `method="unsupv"`.

`y.only` Use only these variables for the multivariate outcomes. Applies only to `method="unsupv"`.

`dist.sharpen` By default, distance sharpening is requested, which applies Euclidean distance to the random forest distance matrix to sharpen it. Because of this, the returned distance matrix will not have values between 0 and 1 (as for random forests distance) when this option is in effect. Distance sharpening is a useful, but slow step. Set this option to `FALSE` to improve computational times, however clustering performance will not be as good. Applies only when `method="sid"` or `method="unsupv"`.

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

Details

Given an unsupervised data set, random forests is used to calculate the distance between all pairs of data points. The distance matrix is used for clustering the unsupervised data where the default is to use hierarchcial clustering. Users can apply other clustering procedures to the distance matrix. See the examples below.

The default method, `method="sid"`, implements sidClustering. The sidClustering algorithm begins by first creating an enhanced SID (Staggered Interaction Data) feature space by sidification of the

original variables. Sidification results in: (a) SID main features which are the original features that have been shifted in order to make them strictly positive and staggered so all of their ranges are mutually exclusive; and (b) SID interaction features which are the multiplicative interactions formed between every pair of SID main features. Multivariate random forests are then trained to predict the main SID features using the interaction SID features as predictors. The basic premise is if features are informative for clusters, then they will vary over the space in a systematic manner, and because each SID interaction feature is uniquely determined by the original feature values used to form the interaction, cuts along the SID interaction feature will be able to find the regions where the informative features vary by cluster, thereby not only reducing impurity, but also separating the clusters which are dependent on those features. See Mantero and Ishwaran (2020) for details.

Because SID uses all pairwise interactions, the dimension of the feature space is proportional to the square of the number of original features (or even larger if factors are present). Thus it is helpful to reduce the feature space. The reduction step (applied by default) utilizes holdout VIMP to accomplish this. It is recommended this step be skipped only when the dimension is reasonably small. For very large data sets this step may be slow.

A second approach (Breiman, 2003; Shi-Horvath, 2006) transforms the unsupervised learning problem into a two class supervised problem. The first class consists of the original observations, while the second class is artificially created. The idea is that in detecting the first class out of the second, the model will generate the random forest proximity between observations of which those for the original class can be extracted and used for clustering. Note in this approach the distance matrix is defined to equal one minus the proximity. This is unlike the distance matrix from SID which is not proximity based. Artificial data is created using "mode 1" or "mode 2" of Shi-Horvath (2006). Mode 1 randomly draws from each set of observed features. Mode 2 draws a uniform value from the minimum and maximum values of a feature.

Mantero and Ishwaran (2020) studied both methods and found SID worked well in all settings, whereas Breiman/Shi-Horvath was sensitive to cluster structure. Performance was poor when clusters were hidden in lower dimensional subspaces; for example when interactions were present or in mixed variable settings (factors/continuous variables). See the V-shaped cluster example below. Generally Shi-Horvath mode 1 outperforms mode 2.

Finally, a third method where the data is used for both the features and outcome is implemented using `method="unsupv"`. Tree nodes are split using the multivariate splitting rule. This is much faster than sidClustering but potentially less accurate.

There is an internal function `sid.perf.metric` for evaluating performance of the procedures using a normalized measure score. Smaller values indicate better performance. See Mantero and Ishwaran (2020) for details.

Value

A list with the following components:

`clustering`	Vector or matrix containing indices mapping data points to their clusters.
`rf`	Random forest object (either a multivariate forest or RF-C object).
`dist`	Distance matrix.
`sid`	The "sid-ified" data. Conveniently broken up into separate values for outcomes and features used by the multivariate forest.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Breiman, L. (2003). *Manual on setting up, using and understanding random forest, V4.0*. University of California Berkeley, Statistics Department, Berkeley.

Mantero A. and Ishwaran H. (2021). Unsupervised random forests. *Statistical Analysis and Data Mining*, 14(2):144-167.

Shi, T. and Horvath, S. (2006). Unsupervised learning with random forest predictors. *Journal of Computational and Graphical Statistics*, 15(1):118-138.

See Also

rfsrc, rfsrc.fast

Examples

```
## ------------------------------------------------------------
## mtcars example
## ------------------------------------------------------------

## default SID method
o1 <- sidClustering(mtcars)
print(split(mtcars, o1$cl[, 10]))

## using artifical class approach
o1.sh <- sidClustering(mtcars, method = "sh")
print(split(mtcars, o1.sh$cl[, 10]))


## ------------------------------------------------------------
## glass data set
## ------------------------------------------------------------

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

  ## this is a supervised problem, so we first strip the class label
  data(Glass)
  glass <- Glass
  y <- Glass$Type
  glass$Type <- NULL

  ## default SID call
  o2 <- sidClustering(glass, k = 6)
  print(table(y, o2$cl))
  print(sid.perf.metric(y, o2$cl))

  ## compare with Shi-Horvath mode 1
  o2.sh <- sidClustering(glass, method = "sh1", k = 6)
  print(table(y, o2.sh$cl))
```

```
  print(sid.perf.metric(y, o2.sh$cl))

  ## plain-vanilla unsupervised analysis
  o2.un <- sidClustering(glass, method = "unsupv", k = 6)
  print(table(y, o2.un$cl))
  print(sid.perf.metric(y, o2.un$cl))

}

## ------------------------------------------------------------
## vowel data set
## ------------------------------------------------------------

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

  ## strip the class label
  data(Vowel)
  vowel <- Vowel
  y <- Vowel$Class
  vowel$Class <- NULL

  ## SID
  o3 <- sidClustering(vowel, k = 11)
  print(table(y, o3$cl))
  print(sid.perf.metric(y, o3$cl))

  ## compare to Shi-Horvath which performs poorly in
  ## mixed variable settings
  o3.sh <- sidClustering(vowel, method = "sh1", k = 11)
  print(table(y, o3.sh$cl))
  print(sid.perf.metric(y, o3.sh$cl))

  ## Shi-Horvath improves with PAM clustering
  ## but still not as good as SID
  o3.sh.pam <- pam(o3.sh$dist, k = 11)$clustering
  print(table(y, o3.sh.pam))
  print(sid.perf.metric(y, o3.sh.pam))

  ## plain-vanilla unsupervised analysis
  o3.un <- sidClustering(vowel, method = "unsupv", k = 11)
  print(table(y, o3.un$cl))
  print(sid.perf.metric(y, o3.un$cl))

}

## ------------------------------------------------------------
##  two-d V-shaped cluster (y=x, y=-x) sitting in 12-dimensions
##  illustrates superiority of SID to Breiman/Shi-Horvath
## ------------------------------------------------------------

p <- 10
m <- 250
```

```
n <- 2 * m
std <- .2

x <- runif(n, 0, 1)
noise <- matrix(runif(n * p, 0, 1), n)
y <- rep(NA, n)
y[1:m] <- x[1:m] + rnorm(m, sd = std)
y[(m+1):n] <- -x[(m+1):n] + rnorm(m, sd = std)
vclus <- data.frame(clus = c(rep(1, m), rep(2,m)), x = x, y = y, noise)

## SID
o4 <- sidClustering(vclus[, -1], k = 2)
print(table(vclus[, 1], o4$cl))
print(sid.perf.metric(vclus[, 1], o4$cl))

## Shi-Horvath
o4.sh <- sidClustering(vclus[, -1], method = "sh1", k = 2)
print(table(vclus[, 1], o4.sh$cl))
print(sid.perf.metric(vclus[, 1], o4.sh$cl))

## plain-vanilla unsupervised analysis
o4.un <- sidClustering(vclus[, -1], method = "unsupv", k = 2)
print(table(vclus[, 1], o4.un$cl))
print(sid.perf.metric(vclus[, 1], o4.un$cl))


## ------------------------------------------------------------
##  two-d V-shaped cluster using fast random forests
## ------------------------------------------------------------

o5 <- sidClustering(vclus[, -1], k = 2, fast = TRUE)
print(table(vclus[, 1], o5$cl))
print(sid.perf.metric(vclus[, 1], o5$cl))
```

`stat.split.rfsrc` *Acquire Split Statistic Information*

Description

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

1. *treeID* Tree identifier.
2. *nodeID* Node identifier.
3. *parmID* Variable indentifier.

4. *contPT* Value node was split in the case of a continuous variable.
5. *mwcpSZ* Size of the multi-word complementary pair in the case of a factor split.
6. *dpthID* Zero (0) based depth of split.
7. *spltTY* Split type for parent node:

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

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

Usage

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

Arguments

`object`	An object of class `(rfsrc, grow)`, `(rfsrc, synthetic)` or `(rfsrc, predict)`
`...`	Further arguments passed to or from other methods.

Value

Invisibly, a list with the following components:

`...`	...

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

Examples

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

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

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

---

subsample.rfsrc *Subsample Forests for VIMP Confidence Intervals*

Description

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

Usage

```
## S3 method for class 'rfsrc'
subsample(obj,
  B = 100,
  block.size = 1,
  importance,
  subratio = NULL,
  stratify = TRUE,
```

```
  performance = FALSE,
  performance.only = FALSE,
  joint = FALSE,
  xvar.names = NULL,
  bootstrap = FALSE,
  verbose = TRUE)
```

Arguments

obj	A forest grow object.
B	Number of subsamples (or number of bootstraps).
block.size	Specifies number of trees in a block when calculating VIMP. This is over-ridden if VIMP is present in the original grow call in which case the grow value is used.
importance	Optional: specifies the type of importance to be used, selected from one of "anti", "permute", "random". If not specified reverts to default importance used by the package. Also, this is over-ridden if the original grow object contains importance, in which case importance used in the original grow call is used.
subratio	Ratio of subsample size to original sample size. The default is approximately equal to the inverse square root of the sample size.
stratify	Use stratified subsampling? See details below.
performance	Generalization error? User can also request standard error and confidence regions for generalization error.
performance.only	Only calculate standard error and confidence region for the generalization error (no VIMP).
joint	Joint VIMP for all variables? Users can also request joint VIMP for specific variables using xvar.names.
xvar.names	Specifies variables for calculating joint VIMP. By default all variables are used.
bootstrap	Use double bootstrap approach in place of subsampling? Much slower, but potentially more accurate.
verbose	Provide verbose output?

Details

Using a previously trained forest, subsamples the data and constructs subsampled forests to estimate standard errors and confidence intervals for VIMP (Ishwaran and Lu, 2019). If bootstrapping is requested, a double bootstrap is applied in place of subsampling. The option performance="TRUE" constructs standard errors and confidence regions for the error rate (OOB performance) of the trained forest. Options joint and xvar.names can be used to obtain joint VIMP for all or some variables.

If the trained forest does not have VIMP values, the algorithm first needs to calculate VIMP. Therefore, if the user plans to make repeated calls to subsample, it is advisable to include VIMP in the original grow call. Also, by calling VIMP in the original call, the type of importance used and other related parameters are set by values used in the original call which can eliminate confusion about what parameters are being used in the subsampled forests. For example, the default importance used is not permutation importance. Thus, it is generally advised to call VIMP in the original call.

Subsampled forests are calculated using the same tuning parameters as the original forest. While a sophisticated algorithm is utilized to acquire as many of these parameters as possible, keep in mind there are some conditions where this will fail: for example there are certain settings where the user has specified non-standard sampling in the grow forest.

Delete-d jackknife estimators of the variance (Shao and Wu, 1989) are returned alongside subsampled variance estimators (Politis and Romano, 1994). While these methods are closely related, the jackknife estimator generally gives *larger* standard errors, which is a form of bias correction, and which occurs primarily for the signal variables.

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

The function `extract.subsample` can be used to extract information from the subsampled object. Returned values for VIMP are "standardized" (this means for regression families, VIMP is standardized by dividing by the variance of Y; for all other families, VIMP is unaltered). Use `standardize="FALSE"` if you want unstandardized VIMP. Setting the option `raw="TRUE"` returns a more complete set of information that is used by the function plot.subsample.rfsrc for plotting confidence intervals. Keep in mind some of this information will be subsampled VIMP that is "raw" in the sense it equals VIMP from a forest constructed with a much smaller sample size. This option is for experts only.

When printing or plotting results, the default is to standardize VIMP which can be turned off using the option `standardize`. Also these wrappers preset the "alpha" value used for confidence intervals; users can change this using option `alpha`.

Value

A list with the following key components:

`rf`	Original forest grow object.
`vmp`	Variable importance values for grow forest.
`vmpS`	Variable importance subsampled values.
`subratio`	Subratio used.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H. and Lu M. (2019). Standard errors and confidence intervals for variable importance in random forest regression, classification, and survival. *Statistics in Medicine*, 38, 558-582.

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

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

See Also

holdout.vimp.rfsrc plot.subsample.rfsrc, rfsrc, vimp.rfsrc

Examples

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

## training the forest
reg.o <- rfsrc(Ozone ~ ., airquality)

## default subsample call
reg.smp.o <- subsample(reg.o)

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

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

## joint vimp and confidence region for generalization error
reg.smp.o2 <- subsample(reg.o, performance = TRUE,
           joint = TRUE, xvar.names = c("Day", "Month"))
plot.subsample(reg.smp.o2)

## now try the double bootstrap (slower)
reg.dbs.o <- subsample(reg.o, B = 25, bootstrap = TRUE)
print(reg.dbs.o)
plot.subsample(reg.dbs.o)

## standard error and confidence region for generalization error only
gerror <- subsample(reg.o, performance.only = TRUE)
plot.subsample(gerror)

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

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

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

  ## same as above, but with VIMP defined using normalized Brier score
```

```
  cls.o2 <- rfsrc(Class ~ ., d, perf.type = "brier")
  cls.smp.o2 <- subsample(cls.o2, B = 100)
  plot.subsample(cls.smp.o2, cex.axis = .7)
}

## ------------------------------------------------------------
## class-imbalanced data using RFQ classifier with G-mean VIMP
## ------------------------------------------------------------

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

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

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

  ## RFQ classifier
  oq <- imbalanced(Class ~ ., d, importance = TRUE, block.size = 10)

  ## subsample the RFQ-classifier
  smp.oq <- subsample(oq, B = 100)
  plot.subsample(smp.oq, cex.axis = .7)

}

## ------------------------------------------------------------
## survival
## ------------------------------------------------------------

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

## ------------------------------------------------------------
## competing risks
## target event is death (event = 2)
## ------------------------------------------------------------

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

```
}

## ------------------------------------------------------------
## multivariate
## ------------------------------------------------------------

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

## ------------------------------------------------------------
## largish data example - use rfsrc.fast for fast forests
## ------------------------------------------------------------

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

  ## use a subsampled forest with Brier score performance
  ## remember to set forest=TRUE for rfsrc.fast
  o <- rfsrc.fast(Class ~ ., d, ntree = 100,
           forest = TRUE, perf.type = "brier")
  so <- subsample(o, B = 100)
  plot.subsample(so, cex.axis = .7)
}
```

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

Arguments

formula	Model to be fit. Must be specified unless object is given.
data	Data frame containing the y-outcome and x-variables in the model. Must be specified unless object is given.
object	An object of class (rfsrc, synthetic). Not required when formula and data are supplied.
newdata	Test data used for prediction (optional).
ntree	Number of trees.
mtry	mtry value for over-arching synthetic forest.
nodesize	Nodesize value for over-arching synthetic forest.
nsplit	nsplit-randomized splitting for significantly increased speed.
mtrySeq	Sequence of mtry values used for fitting the collection of RF machines. If NULL, default is number of variables divided by 3, rounded up.
nodesizeSeq	Sequence of nodesize values used for the fitting the collection of RF machines.
min.node	Minimum forest averaged number of nodes a RF machine must exceed in order to be used as a synthetic feature.
fast	Use fast random forests, rfsrc.fast, in place of rfsrc? Improves speed but may be less accurate.
use.org.features	In addition to synthetic features, should the original features be used when fitting synthetic forests?
na.action	Missing value action. The default na.omit removes the entire record if even one of its entries is NA. The action na.impute pre-imputes the data using fast imputation via impute.rfsrc.
oob	Preserve "out-of-bagness" so that error rates and VIMP are honest? Default is yes (‘oob=TRUE’).
verbose	Set to TRUE for verbose output.
...	Further arguments to be passed to the rfsrc function used for fitting the synthetic forest.

Details

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

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

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

Value

A list with the following components:

`rfMachines`	RF machines used to construct the synthetic features.
`rfSyn`	The (grow) synthetic RF built over training data.
`rfSynPred`	The predict synthetic RF built over test data (if available).
`synthetic`	List containing the synthetic features.
`opt.machine`	Optimal machine: RF machine with smallest OOB error rate.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

See Also

`rfsrc`, `rfsrc.fast`

Examples

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

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

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

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

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

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


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

}

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

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

## regression forests
regF <- rfsrc(y ~ ., dat[-test, ], )
pred.regF <- predict(regF, dat[test, ])

## synthetic forests using fast rfsrc
synF1 <- synthetic(y ~ ., dat[-test, ], newdata = dat[test, ])
synF2 <- synthetic(y ~ ., 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)
trn <- sample(1:nrow(mtcars.new), nrow(mtcars.new)/2)
mvSyn <- synthetic(cbind(carb, mpg, cyl) ~., mtcars.new[trn,])
mvSyn.pred <- synthetic(object = mvSyn, newdata = mtcars.new[-trn,])
```

`tune.rfsrc` *Tune Random Forest for the optimal mtry and nodesize parameters*

Description

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

Usage

```
## S3 method for class 'rfsrc'
tune(formula, data,
  mtryStart = ncol(data) / 2,
  nodesizeTry = c(1:9, seq(10, 100, by = 5)), ntreeTry = 100,
  sampsize = function(x){min(x * .632, max(150, x ^ (3/4)))},
  nsplit = 1, stepFactor = 1.25, improve = 1e-3, strikeout = 3, maxIter = 25,
  trace = FALSE, doBest = FALSE, ...)

## S3 method for class 'rfsrc'
tune.nodesize(formula, data,
  nodesizeTry = c(1:9, seq(10, 150, by = 5)), ntreeTry = 100,
  sampsize = function(x){min(x * .632, max(150, x ^ (4/5)))},
  nsplit = 1, trace = TRUE, ...)
```

Arguments

`formula`	A symbolic description of the model to be fit.
`data`	Data frame containing the y-outcome and x-variables.
`mtryStart`	Starting value of mtry.
`nodesizeTry`	Values of nodesize optimized over.
`ntreeTry`	Number of trees used for the tuning step.
`sampsize`	Function specifying requested size of subsampled data. Can also be passed in as a number.
`nsplit`	Number of random splits used for splitting.
`stepFactor`	At each iteration, mtry is inflated (or deflated) by this value.

improve	The (relative) improvement in out-of-sample error must be by this much for the search to continue.
strikeout	The search is discontinued when the relative improvement in OOB error is negative. However strikeout allows for some tolerance in this. If a negative improvement is noted a total of strikeout times, the search is stopped. Increase this value only if you want an exhaustive search.
maxIter	The maximum number of iterations allowed for each mtry bisection search.
trace	Print the progress of the search?
doBest	Return a forest fit with the optimal mtry and nodesize parameters?
...	Further options to be passed to rfsrc.fast.

Details

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

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

All calculations (including the final optimized forest) are based on the fast forest interface rfsrc.fast which utilizes subsampling. However, while this yields a fast optimization strategy, such a solution can only be considered approximate. Users may wish to tweak various options to improve accuracy. Increasing the default sampsize will definitely help. Increasing ntreeTry (which is set to 100 for speed) may also help. It is also useful to look at contour plots of the out-of-sample error as a function of mtry and nodesize (see example below) to identify regions of the parameter space where error rate is small.

tune.nodesize returns the optimal nodesize where optimization is over nodesize only.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

See Also

rfsrc.fast

Examples

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

## load the data
data(wine, package = "randomForestSRC")
wine$quality <- factor(wine$quality)

## set the sample size manually
o <- tune(quality ~ ., wine, sampsize = 100)
```

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

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

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

  ## plot the surface
  plot.tune(o)

}

## ------------------------------------------------------------
## tuning for class imbalanced data problem
## - see imbalanced function for details
## - use rfq and perf.type = "gmean"
## ------------------------------------------------------------

data(breast, package = "randomForestSRC")
breast <- na.omit(breast)
o <- tune(status ~ ., data = breast, rfq = TRUE, perf.type = "gmean")
print(o)


## ------------------------------------------------------------
## tune nodesize for competing risk - wihs data
## ------------------------------------------------------------

data(wihs, package = "randomForestSRC")
plot(tune.nodesize(Surv(time, status) ~ ., wihs, trace = TRUE)$err)
```

`var.select.rfsrc` *Variable Selection*

Description

Variable selection using minimal depth.

Usage

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

Arguments

`formula`	A symbolic description of the model to be fit. Must be specified unless `object` is given.
`data`	Data frame containing the y-outcome and x-variables in the model. Must be specified unless `object` is given.
`object`	An object of class (`rfsrc`, `grow`). Not required when `formula` and `data` are supplied.
`cause`	Integer value between 1 and `J` indicating the event of interest for competing risks, where `J` is the number of event types (this option applies only to competing risk families). The default is to use the first event type.
`m.target`	Character value for multivariate families specifying the target outcome to be used. If left unspecified, the algorithm will choose a default target.
`method`	Variable selection method:
	`md`**:** minimal depth (default).

vh: variable hunting.

vh.vimp: variable hunting with VIMP (variable importance).

conservative Level of conservativeness of the thresholding rule used in minimal depth selection:

high: Use the most conservative threshold.

medium: Use the default less conservative tree-averaged threshold.

low: Use the more liberal one standard error rule.

ntree Number of trees to grow.

mvars Number of randomly selected variables used in the variable hunting algorithm (ignored when 'method="md"').

mtry The mtry value used.

nodesize Forest average terminal node size.

splitrule Splitting rule used.

nsplit If non-zero, the specified tree splitting rule is randomized which significantly increases speed.

xvar.wt Vector of non-negative weights specifying the probability of selecting a variable for splitting a node. Must be of dimension equal to the number of variables. Default (NULL) invokes uniform weighting or a data-adaptive method depending on prefit$action.

refit Should a forest be refit using the selected variables?

fast Speeds up the cross-validation used for variable hunting for a faster analysis. See miscellanea below.

na.action Action to be taken if the data contains NA values.

always.use Character vector of variable names to always be included in the model selection procedure and in the final selected model.

nrep Number of Monte Carlo iterations of the variable hunting algorithm.

K Integer value specifying the K-fold size used in the variable hunting algorithm.

nstep Integer value controlling the step size used in the forward selection process of the variable hunting algorithm. Increasing this will encourage more variables to be selected.

prefit List containing parameters used in preliminary forest analysis for determining weight selection of variables. Users can set all or some of the following parameters:

action: Determines how (or if) the preliminary forest is fit. See details below.

ntree: Number of trees used in the preliminary analysis.

mtry: mtry used in the preliminary analysis.

nodesize: nodesize used in the preliminary analysis.

nsplit: nsplit value used in the preliminary analysis.

verbose Set to TRUE for verbose output.

block.size VIMP is calculated in "blocks" of trees of this size.

seed Negative integer specifying seed for the random number generator.

... Further arguments passed to forest grow call.

Details

This function implements random forest variable selection using tree minimal depth methodology (Ishwaran et al., 2010). The option 'method' allows for two different approaches:

1. 'method="md"'

   Invokes minimal depth variable selection. Variables are selected using minimal depth variable selection. Uses all data and all variables simultaneously. This is basically a front-end to the max.subtree wrapper. Users should consult the max.subtree help file for details.

   Set 'mtry' to larger values in high-dimensional problems.

2. 'method="vh"' or 'method="vh.vimp"'

   Invokes variable hunting. Variable hunting is used for problems where the number of variables is substantially larger than the sample size (e.g., p/n is greater than 10). It is always prefered to use 'method="md"', but to find more variables, or when computations are high, variable hunting may be preferred.

   When 'method="vh"': Using training data from a stratified K-fold subsampling (stratification based on the y-outcomes), a forest is fit using mvars randomly selected variables (variables are chosen with probability proportional to weights determined using an initial forest fit; see below for more details). The mvars variables are ordered by increasing minimal depth and added sequentially (starting from an initial model determined using minimal depth selection) until joint VIMP no longer increases (signifying the final model). A forest is refit to the final model and applied to test data to estimate prediction error. The process is repeated nrep times. Final selected variables are the top P ranked variables, where P is the average model size (rounded up to the nearest integer) and variables are ranked by frequency of occurrence.

   The same algorithm is used when 'method="vh.vimp"', but variables are ordered using VIMP. This is faster, but not as accurate.

Miscellanea

1. When variable hunting is used, a preliminary forest is run and its VIMP is used to define the probability of selecting a variable for splitting a node. Thus, instead of randomly selecting mvars at random, variables are selected with probability proportional to their VIMP (the probability is zero if VIMP is negative). A preliminary forest is run once prior to the analysis if prefit$action=TRUE, otherwise it is run prior to each iteration (this latter scenario can be slow). When 'method="md"', a preliminary forest is fit only if prefit$action=TRUE. Then instead of randomly selecting mtry variables at random, mtry variables are selected with probability proportional to their VIMP. In all cases, the entire option is overridden if xvar.wt is non-null.
2. If object is supplied and 'method="md"', the grow forest from object is parsed for minimal depth information. While this avoids fitting another forest, thus saving computational time, certain options no longer apply. In particular, the value of cause plays no role in the final selected variables as minimal depth is extracted from the grow forest, which has already been grown under a preselected cause specification. Users wishing to specify cause should instead use the formula and data interface. Also, if the user requests a prefitted forest via prefit$action=TRUE, then object is not used and a refitted forest is used in its place for variable selection. Thus, the effort spent to construct the original grow forest is not used in this case.

3. If 'fast=TRUE', and variable hunting is used, the training data is chosen to be of size n/K, where n=sample size (i.e., the size of the training data is swapped with the test data). This speeds up the algorithm. Increasing K also helps.
4. Can be used for competing risk data. When 'method="vh.vimp"', variable selection based on VIMP is confined to an event specific cause specified by cause. However, this can be unreliable as not all y-outcomes can be guaranteed when subsampling (this is true even when stratifed subsampling is used as done here).

Value

Invisibly, a list with the following components:

err.rate	Prediction error for the forest (a vector of length nrep if variable hunting is used).
modelsize	Number of variables selected.
topvars	Character vector of names of the final selected variables.
varselect	Useful output summarizing the final selected variables.
rfsrc.refit.obj	Refitted forest using the final set of selected variables (requires 'refit=TRUE').
md.obj	Minimal depth object. NULL unless 'method="md"'.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

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

See Also

find.interaction.rfsrc, holdout.vimp.rfsrc, max.subtree.rfsrc, vimp.rfsrc

Examples

```
## ------------------------------------------------------------
## Minimal depth variable selection
## survival analysis
## use larger node size which is better for minimal depth
## ------------------------------------------------------------

data(pbc, package = "randomForestSRC")
pbc.obj <- rfsrc(Surv(days, status) ~ ., pbc, nodesize = 20, importance = TRUE)

# default call corresponds to minimal depth selection
vs.pbc <- var.select(object = pbc.obj)
```

```
topvars <- vs.pbc$topvars

# the above is equivalent to
max.subtree(pbc.obj)$topvars

# different levels of conservativeness
var.select(object = pbc.obj, conservative = "low")
var.select(object = pbc.obj, conservative = "medium")
var.select(object = pbc.obj, conservative = "high")

## ------------------------------------------------------------
## Minimal depth variable selection
## competing risk analysis
## use larger node size which is better for minimal depth
## ------------------------------------------------------------

## competing risk data set involving AIDS in women
data(wihs, package = "randomForestSRC")
vs.wihs <- var.select(Surv(time, status) ~ ., wihs, nsplit = 3,
                      nodesize = 20, ntree = 100, importance = TRUE)

## competing risk analysis of pbc data from survival package
## implement cause-specific variable selection
if (library("survival", logical.return = TRUE)) {
  data(pbc, package = "survival")
  pbc$id <- NULL
  var.select(Surv(time, status) ~ ., pbc, cause = 1)
  var.select(Surv(time, status) ~ ., pbc, cause = 2)
}

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

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

## ------------------------------------------------------------
## Variable hunting high-dimensional example
## van de Vijver microarray breast cancer survival data
## nrep is small for illustration; typical values are nrep = 100
## ------------------------------------------------------------

data(vdv, package = "randomForestSRC")
vh.breast <- var.select(Surv(Time, Censoring) ~ ., vdv,
      method = "vh", nrep = 10, nstep = 5)

# plot top 10 variables
plot.variable(vh.breast$rfsrc.refit.obj,
  xvar.names = vh.breast$topvars[1:10])
plot.variable(vh.breast$rfsrc.refit.obj,
  xvar.names = vh.breast$topvars[1:10], partial = TRUE)
```

```
## similar analysis, but using weights from univarate cox p-values
if (library("survival", logical.return = TRUE))
{
  cox.weights <- function(rfsrc.f, rfsrc.data) {
    event.names <- all.vars(rfsrc.f)[1:2]
    p <- ncol(rfsrc.data) - 2
    event.pt <- match(event.names, names(rfsrc.data))
    xvar.pt <- setdiff(1:ncol(rfsrc.data), event.pt)
    sapply(1:p, function(j) {
      cox.out <- coxph(rfsrc.f, rfsrc.data[, c(event.pt, xvar.pt[j])])
      pvalue <- summary(cox.out)$coef[5]
      if (is.na(pvalue)) 1.0 else 1/(pvalue + 1e-100)
    })
  }
  data(vdv, package = "randomForestSRC")
  rfsrc.f <- as.formula(Surv(Time, Censoring) ~ .)
  cox.wts <- cox.weights(rfsrc.f, vdv)
  vh.breast.cox <- var.select(rfsrc.f, vdv, method = "vh", nstep = 5,
    nrep = 10, xvar.wt = cox.wts)
}
```

---

vdv *van de Vijver Microarray Breast Cancer*

Description

Gene expression profiling for predicting clinical outcome of breast cancer (van't Veer et al., 2002). Microarray breast cancer data set of 4707 expression values on 78 patients with survival information.

References

van't Veer L.J. et al. (2002). Gene expression profiling predicts clinical outcome of breast cancer. *Nature*, **12**, 530–536.

Examples

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

veteran *Veteran's Administration Lung Cancer Trial*

Description

Randomized trial of two treatment regimens for lung cancer. This is a standard survival analysis data set.

Source

Kalbfleisch and Prentice, *The Statistical Analysis of Failure Time Data.*

References

Kalbfleisch J. and Prentice R, (1980) *The Statistical Analysis of Failure Time Data.* New York: Wiley.

Examples

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

vimp.rfsrc *VIMP for Single or Grouped Variables*

Description

Calculate variable importance (VIMP) for a single variable or group of variables for training or test data.

Usage

```
## S3 method for class 'rfsrc'
vimp(object, xvar.names, m.target = NULL,
  importance = c("anti", "permute", "random"), block.size = 10,
  joint = FALSE, seed = NULL, do.trace = FALSE, ...)
```

Arguments

`object`	An object of class (`rfsrc`, `grow`) or (`rfsrc`, `forest`). Requires '`forest=TRUE`' in the original `rfsrc` call.
`xvar.names`	Names of the x-variables to be used. If not specified all variables are used.
`m.target`	Character value for multivariate families specifying the target outcome to be used. If left unspecified, the algorithm will choose a default target.
`importance`	Type of VIMP.
`block.size`	Specifies number of trees in a block when calculating VIMP.

joint	Individual or joint VIMP?
seed	Negative integer specifying seed for the random number generator.
do.trace	Number of seconds between updates to the user on approximate time to completion.
...	Further arguments passed to or from other methods.

Details

Using a previously trained forest, calculate the VIMP for variables xvar.names. By default, VIMP is calculated for the original data, but the user can specify a new test data for the VIMP calculation using newdata. See rfsrc for more details about how VIMP is calculated.

'joint=TRUE' returns joint VIMP, defined as importance for a group of variables when the group is perturbed simultaneously.

csv=TRUE return case specific VIMP. Applies to all families except survival families. See example below.

Value

An object of class (rfsrc, predict) containing importance values.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

See Also

holdout.vimp.rfsrc, rfsrc

Examples

```
## ------------------------------------------------------------
## classification example
## showcase different vimp
## ------------------------------------------------------------

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

## anti vimp (default)
print(vimp(iris.obj)$importance)

## anti vimp using brier prediction error
print(vimp(iris.obj, perf.type = "brier")$importance)

## permutation vimp
```

```
print(vimp(iris.obj, importance = "permute")$importance)

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

## joint anti vimp
print(vimp(iris.obj, joint = TRUE)$importance)

## paired anti 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)

## ------------------------------------------------------------
## survival example
## anti versus permute VIMP with different block sizes
## ------------------------------------------------------------

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

print(vimp(pbc.obj)$importance)
print(vimp(pbc.obj, block.size=1)$importance)
print(vimp(pbc.obj, importance="permute")$importance)
print(vimp(pbc.obj, importance="permute", block.size=1)$importance)

## ------------------------------------------------------------
## imbalanced classification example
## see the imbalanced function for more details
## ------------------------------------------------------------

data(breast, package = "randomForestSRC")
breast <- na.omit(breast)
f <- as.formula(status ~ .)
o <- rfsrc(f, breast, ntree = 2000)

## permutation vimp
print(100 * vimp(o, importance = "permute")$importance)

## anti vimp using gmean performance
print(100 * vimp(o, perf.type = "gmean")$importance[, 1])

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

airq.obj <- rfsrc(Ozone ~ ., airquality)
print(vimp(airq.obj))

## ------------------------------------------------------------
## regression example where vimp is calculated on test data
## ------------------------------------------------------------

set.seed(100080)
```

```
train <- sample(1:nrow(airquality), size = 80)
airq.obj <- rfsrc(Ozone~., airquality[train, ])

## training data vimp
print(airq.obj$importance)
print(vimp(airq.obj)$importance)

## test data vimp
print(vimp(airq.obj, newdata = airquality[-train, ])$importance)

## ------------------------------------------------------------
## case-specific vimp
## returns VIMP for each case
## ------------------------------------------------------------

o <- rfsrc(mpg~., mtcars)
v <- vimp(o, csv = TRUE)
csvimp <- get.mv.csvimp(v, standardize=TRUE)
print(csvimp)

## ------------------------------------------------------------
## case-specific joint vimp
## returns joint VIMP for each case
## ------------------------------------------------------------

o <- rfsrc(mpg~., mtcars)
v <- vimp(o, joint = TRUE, csv = TRUE)
csvimp <- get.mv.csvimp(v, standardize=TRUE)
print(csvimp)

## ------------------------------------------------------------
## case-specific joint vimp for multivariate regression
## returns joint VIMP for each case, for each outcome
## ------------------------------------------------------------

o <- rfsrc(Multivar(mpg, cyl) ~., data = mtcars)
v <- vimp(o, joint = TRUE, csv = TRUE)
csvimp <- get.mv.csvimp(v, standardize=TRUE)
print(csvimp)
```

`wihs` *Women's Interagency HIV Study (WIHS)*

Description

Competing risk data set involving AIDS in women.

Format

A data frame containing:

time	time to event
status	censoring status: 0=censoring, 1=HAART initiation, 2=AIDS/Death before HAART
ageatfda	age in years at time of FDA approval of first protease inhibitor
idu	history of IDU: 0=no history, 1=history
black	race: 0=not African-American; 1=African-American
cd4nadir	CD4 count (per 100 cells/ul)

Source

Study included 1164 women enrolled in WIHS, who were alive, infected with HIV, and free of clinical AIDS on December, 1995, when the first protease inhibitor (saquinavir mesylate) was approved by the Federal Drug Administration. Women were followed until the first of the following occurred: treatment initiation, AIDS diagnosis, death, or administrative censoring (September, 2006). Variables included history of injection drug use at WIHS enrollment, whether an individual was African American, age, and CD4 nadir prior to baseline.

References

Bacon M.C, von Wyl V., Alden C., et al. (2005). The Women's Interagency HIV Study: an observational cohort brings clinical sciences to the bench, *Clin Diagn Lab Immunol*, 12(9):1013-1019.

Examples

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

---

wine *White Wine Quality Data*

Description

The inputs include objective tests (e.g. PH values) and the output is based on sensory data (median of at least 3 evaluations made by wine experts) of white wine. Each expert graded the wine quality between 0 (very bad) and 10 (very excellent).

References

Cortez, P., Cerdeira, A., Almeida, F., Matos T. and Reis, J. (2009). Modeling wine preferences by data mining from physicochemical properties. In *Decision Support Systems*, Elsevier, 47(4):547-553.

Examples

```
## load wine and convert to a multiclass problem
data(wine, package = "randomForestSRC")
wine$quality <- factor(wine$quality)
```

Index