Package 'randomForestSRC'

May 9, 2026

Version 3.6.2

Date 2026-04-18

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, cluster, fst, data.table

SystemRequirements OpenMP

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 holdout 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 2026-04-19 05:10:02 UTC

Contents

`randomForestSRC-package`

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

Description

Fast OpenMP-parallel implementation of Breiman's random forests (Breiman, 2001) for regression, classification, survival analysis (Ishwaran, 2008), competing risks (Ishwaran, 2012), multivariate outcomes (Segal and Xiao, 2011), unsupervised learning (Mantero and Ishwaran, 2020), quantile regression (Meinshausen, 2006; Zhang et al., 2019; Greenwald and Khanna, 2001), and imbalanced q-classification (O'Brien and Ishwaran, 2019).

Supports deterministic and randomized splitting rules (Geurts et al., 2006; Ishwaran, 2015) across all families. Variable importance (VIMP), holdout VIMP, and confidence regions (Ishwaran and Lu, 2019) can be computed for single and grouped variables. Includes minimal depth variable selection (Ishwaran et al., 2010, 2011) and a fast interface for missing data imputation using multiple forest-based methods (Tang and Ishwaran, 2017).

Tree structures can be visualized in Safari or Chrome for any family; see `get.tree`.

Package Overview

This package contains many useful functions. Users are encouraged to read the help files in full for detailed guidance. Below is a brief overview of key functions to help navigate the package.

1. `rfsrc`
   The main entry point to the package. Builds a random forest using user-supplied training data. The returned object is of class `(rfsrc, grow)`.
2. `rfsrc.fast`
   A computationally efficient version of `rfsrc` using subsampling.
3. `quantreg.rfsrc`, `quantreg`
   Univariate and multivariate quantile regression forests for training and testing. Includes methods such as the Greenwald-Khanna (2001) algorithm, ideal for large data due to its memory efficiency.
4. `predict.rfsrc`, `predict`
   Predicts outcomes by dropping test data down the trained forest. Returns an object of class `(rfsrc, predict)`.
5. `sidClustering.rfsrc`, `sidClustering`
   Unsupervised clustering using SID (Staggered Interaction Data). Also includes Breiman's artificial two-class method (Breiman, 2003).
6. `vimp`, `subsample`, `holdout.vimp`
   Functions for variable selection and importance assessment:
   (a) `vimp`: Computes variable importance (VIMP) by perturbing each variable (e.g., via permutation). Can also be computed directly in `rfsrc` and `predict.rfsrc`.
   (b) `subsample`: Computes confidence intervals for VIMP using subsampling.
   (c) `holdout.vimp`: Measures the effect of removing a variable from the model.
   (d) **VarPro** (**VarPro** package): For advanced model-independent variable selection using rule-based variable priority. Supports regression, classification, survival, and unsupervised data. See `https://www.varprotools.org`.
7. `imbalanced.rfsrc`, `imbalanced`
   Implements q-classification and G-mean-based VIMP for class-imbalanced data.

8. impute.rfsrc, impute

   A fast interface for missing data imputation. While rfsrc and predict.rfsrc can handle missing data internally, this provides a dedicated, efficient solution for imputation tasks.

9. partial.rfsrc, partial

   Computes partial dependence functions to assess the marginal effect of one or more variables on the forest ensemble.

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

1. The package home page, with vignettes, manuals, GitHub links, and additional documentation, is available at: https://www.randomforestsrc.org/index.html
2. Questions, comments, and general usage discussions (non-bug-related) can be posted at: https://github.com/kogalur/randomForestSRC/discussions/
3. Bug reports should be submitted at: https://github.com/kogalur/randomForestSRC/issues/

   Please use this only for bugs, and include the following with your report:

   - Output from sessionInfo().
   - A minimal reproducible example including:
     - A minimal dataset required to reproduce the error.
     - The smallest runnable code needed to reproduce the issue.
     - Version details of R and all relevant packages.
     - A random seed (via set.seed()) if randomness is involved.
4. The latest stable release of the package is available on CRAN: https://cran.r-project.org/package=randomForestSRC/
5. Development builds (unstable) with bug fixes and new features are hosted on GitHub: https://github.com/kogalur/randomForestSRC/

OpenMP Parallel Processing – Installation

This package supports OpenMP shared-memory parallel programming on systems where the architecture and operating system permit it. OpenMP is enabled by default.

Detailed instructions for configuring OpenMP parallel processing can be found at: https://www.randomforestsrc.org/articles/installation.html

Note that running the package with OpenMP (or Open MPI) may increase memory (RAM) usage. Users are advised to understand their system's hardware limits and to monitor resource consumption to avoid overtaxing CPU and memory capacity.

Reproducibility

Model reproducibility is determined by three components: the random seed, the forest topology (i.e., the structure of trees), and terminal node membership for the training data. These elements together allow the model and its terminal node statistics to be faithfully restored.

Other outputs, such as variable importance (VIMP) and performance metrics, rely on additional internal randomization and are not considered part of the model definition. As a result, such statistics are subject to Monte Carlo variability and may differ across runs, even with the same seed.

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

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,

subsample.rfsrc,

tune.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)
```

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 specifying the tree to extract.
target	For classification: integer or character indicating the class of interest (defaults to the first class). For competing risks: integer between 1 and J (number of event types) specifying the event of interest (default is the first event type).
m.target	Character string specifying the target outcome for multivariate families. If unspecified, a default is selected.
time	For survival: time point at which the predicted value is evaluated (depends on surv.type).
surv.type	For survival: specifies the type of predicted value returned. See Details.
class.type	For classification: specifies the type of predicted value. See Details.
ensemble	Logical. If TRUE, prediction is based on the ensemble of all trees. If FALSE (default), prediction is based on the specified tree.
oob	Logical. Use OOB predicted values (TRUE) or in-bag values (FALSE). Only applies when ensemble=TRUE.
show.plots	Logical. Should plots be displayed?
do.trace	Number of seconds between progress updates.

Details

Extracts a specified tree from a forest and converts it into a hierarchical structure compatible with the **data.tree** package. Plotting the resulting object renders an interactive tree visualization in the user's web browser.

Left-hand splits are shown. For continuous variables, the left split is displayed as an inequality (e.g., x < value); the right split is the reverse. For factor variables, the left daughter node is defined by a set of levels assigned to it; the right daughter is its complement.

Terminal nodes are highlighted with color and display both sample size and predicted value. By default, the predicted value corresponds to the prediction from the selected tree, and the sample size refers to the in-bag cases reaching the terminal node. If ensemble = TRUE, the predicted value equals the forest ensemble prediction, allowing visualization of the full forest predictor over the selected tree's partition. In this case, sample sizes refer to all observations (not just in-bag cases).

Predicted values displayed in terminal nodes are defined as follows:

1. For regression: the mean of the response.
2. For classification: depends on the class.type argument and target class:
   - If class.type = "bayes", the predicted class with the most votes, or the RFQ classifier threshold in two-class problems.
   - If class.type = "prob", the class probability for the target class.
3. For multivariate families: the predicted value for the outcome specified by m.target, using the logic above depending on whether the outcome is continuous or categorical.
4. For survival:
   - mort: estimated mortality (Ishwaran et al., 2008).

- `rel.freq`: relative frequency of mortality.
- `surv`: predicted survival probability at the specified time (`time`).

5. For competing risks:
   - `years.lost`: expected number of life years lost.
   - `cif`: cumulative incidence function.
   - `chf`: cause-specific cumulative hazard function.

   For `cif` and `chf`, predictions are evaluated at the time point given by `time`, and all metrics are specific to the event type indicated by `target`.

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`	Specifies the number of trees used to grow the forest. Can be a function of data dimension and number of holdout variables, or a fixed numeric value.
`nsplit`	Non-negative integer specifying the number of random split points used to split a node. A value of zero corresponds to deterministic splitting, which is significantly slower.
`ntime`	Integer value used for survival settings to constrain ensemble calculations to a grid of `ntime` time points.
`sampsize`	Specifies the size of the subsampled data. Can be either a function or a numeric value.
`samptype`	Type of bootstrap used when subsampling.
`vtry`	Number of variables randomly selected to be held out when growing a tree. Can also be a list for targeted holdout variable importance analysis. See `details` for more information.
`block.size`	Specifies the number of trees in a block when calculating holdout variable importance.
`...`	Further arguments passed to `rfsrc`.

Details

Holdout variable importance (holdout VIMP) measures the importance of a variable by comparing prediction error between two forests (blocks of trees): one in which selected variables are held out during tree growing (the *holdout forest*) and one in which no variables are held out (the *baseline forest*).

For each variable-block combination, the bootstrap samples used to grow the trees are the same in both forests. The difference in out-of-bag (OOB) prediction error between the holdout and baseline forests gives the holdout VIMP for that variable-block pair. The final holdout VIMP for a variable is the average of these differences over all blocks in which the variable was held out.

The option `vtry` controls how many variables are held out per tree. The default is one, meaning a single variable is held out per tree. Larger values of `vtry` increase the number of times each variable

is held out, reducing the required total number of trees. However, interpretation of holdout VIMP changes when `vtry` exceeds one, and this option should be used cautiously.

High accuracy requires a sufficiently large number of trees. As a general guideline, we recommend using `ntree = 1000 * p / vtry`, where `p` is the number of features. Accuracy also depends on `block.size`, which determines how many trees comprise a block. Smaller values yield better accuracy but are computationally more demanding. The most accurate setting is `block.size = 1`. Ensure that `block.size` does not exceed `ntree / p`, otherwise insufficient trees may be available for certain variables.

Targeted holdout VIMP analysis can be requested by specifying `vtry` as a list with two components: a vector of variable indices (`xvar`) and a logical flag `joint` indicating whether to compute joint VIMP. For example, to compute holdout VIMP only for variables 1, 4, and 5 individually:

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

To compute the joint effect of removing these three variables together:

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

Targeted analysis is useful when the user has prior knowledge of variables of interest and can significantly reduce computation. Joint VIMP quantifies the combined 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`	A data frame containing the two-class y-outcome and x-variables.
`ntree`	Number of trees to grow.
`method`	Method used to fit the classifier. The default is `"rfq"`, which implements the random forest quantile classifier (RFQ) of O'Brien and Ishwaran (2017). The option `"brf"` applies the balanced random forest (BRF) approach of Chen et al. (2004), which undersamples the majority class to match the minority class size. The option `"standard"` performs a standard random forest analysis.
`splitrule`	Splitting rule used to grow trees. The default is `"auc"`, which optimizes G-mean performance. Other supported options are `"gini"` and `"entropy"`.
`perf.type`	Performance metric used for evaluating the classifier and computing downstream quantities such as VIMP. Defaults depend on the method: `"gmean"` for RFQ and BRF; `"misclass"` (misclassification error) for standard random forests. Users may override this by specifying `"gmean"`, `"misclass"`, or `"brier"` (normalized Brier score). See examples for usage.
`block.size`	Controls how the cumulative error rate is computed. If `NULL`, it is calculated only once for the final tree. If set to an integer, cumulative error and VIMP are computed in blocks of that size. If unspecified, uses the default in `rfsrc`.
`fast`	Logical. If `TRUE`, uses the fast random forest implementation via `rfsrc.fast` instead of `rfsrc`. Improves speed at the cost of accuracy. Applies only to RFQ.
`ratio`	Optional and experimental. Specifies the proportion (between 0 and 1) of majority class cases to sample during RFQ training. Sampling is without replacement. Ignored for BRF.
`...`	Additional arguments passed to `rfsrc` to control random forest behavior.

Details

Imbalanced data, also known as the minority class problem, refers to two-class classification settings where the majority class significantly outnumbers the minority class. This function supports two approaches to address class imbalance:

- The random forests quantile classifier (RFQ) proposed by O'Brien and Ishwaran (2017).
- The balanced random forest (BRF) undersampling method of Chen et al. (2004).

By default, the performance metric used is the G-mean (Kubat et al., 1997), which balances sensitivity and specificity.

Handling of missing values: Missing data are not supported for BRF or when the `ratio` option is specified. In these cases, records with missing values are removed prior to analysis.

Variable importance: Permutation-based VIMP is used by default in this setting, in contrast to anti-VIMP which is the default for other families. Empirical results suggest that permutation VIMP is more reliable in highly imbalanced settings.

Tree count recommendation: We recommend using a relatively large value for `ntree` in imbalanced problems to ensure stable performance estimation, especially for G-mean. As a general guideline, use at least five times the usual number of trees.

Performance metrics: A helper function, `get.imbalanced.performance`, is provided for extracting classification performance summaries. The metric names are self-explanatory in most cases. Some key metrics include:

- `F1`: The harmonic mean of precision and recall.
- `F1mod`: The harmonic mean of sensitivity, specificity, precision, and negative predictive value.
- `F1gmean`: The average of F1 and G-mean.
- `F1modgmean`: The average of F1mod and 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 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.learn.rfsrc` *Learn a predictive imputer for test-time imputation and OOD scoring*

Description

Learns a predictive imputer from training data for later use on new data.

If the training data contain missing values, the function first imputes them using `impute`. It then fits one saved full-sweep learner per selected target on the completed training data and reuses those learners later to update missing values in new data without refitting on the test set.

The same saved learner bank can also be used to score new data for out-of-distribution (OOD) behavior. Note that OOD scores are available even when new data have missing values. Each selected target is reconstructed from its saved conditional learner and compared with the observed

value. Target-wise discrepancies are calibrated against a training reference calculated from out-of-bag predictions computed during the training.

If the training data are complete and `target.mode = "all"`, the initial training-data imputation step is skipped and the full-sweep learners are fit directly from the complete training data.

Usage

```
impute.learn.rfsrc(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,
  ...,
  full.sweep.options = list(ntree = 100, nsplit = 10),
  target.mode = c("missing.only", "all"),
  deployment.xvars = NULL,
  anonymous = TRUE,
  learner.prefix = "impute.learner.",
  learner.root = "learners",
  out.dir = NULL,
  wipe = TRUE,
  keep.models = is.null(out.dir),
  keep.ximp = FALSE,
  save.on.fit = !is.null(out.dir),
  save.ood = TRUE,
  weight = NULL)

save.impute.learn.rfsrc(object, path, wipe = TRUE, verbose = TRUE)

load.impute.learn.rfsrc(path, targets = NULL, lazy = TRUE, verbose = TRUE)

## S3 method for class 'impute.learn.rfsrc'
predict(object, newdata,
  max.predict.iter = 3L,
  eps = 1e-3,
  targets = NULL,
  restore.integer = TRUE,
  cache.learners = c("session", "none", "all"),
  verbose = TRUE,
  ...)

impute.ood.rfsrc(object, newdata,
  targets = NULL,
  max.predict.iter = 3L,
  eps = 1e-3,
  cache.learners = c("all", "session", "none"),
  weight = NULL,
  aggregate = c("bounded.product",  "weighted.mean",
```

```
  "weighted.lp", "weighted.lp.log", "top.k"),
aggregate.args = list(),
return.details = FALSE,
verbose = TRUE,
...)
```

Arguments

formula	A symbolic model description. Can be omitted. The same interpretation as in impute is used for the initial training-data imputation stage. The saved full-sweep learner bank is controlled by deployment.xvars, not by formula.
data	Training data. Variables that are not real-valued are coerced to factors before fitting when possible; otherwise fitting stops with an error. Rows and columns that are entirely missing are dropped before the training begins.
ntree, nodesize, nsplit, nimpute, fast, blocks, max.iter, ytry, always.use, verbose	Arguments passed to impute for the initial training-data imputation. The argument full.sweep is controlled internally and should not be supplied. The same verbose, max.predict.iter, eps, and cache.learners controls are also used by predict.impute.learn and impute.ood.
mf.q	Controls the imputation engine used by impute. If mf.q = 1, training uses standard missForest. If mf.q > 1, training uses the multivariate missForest generalization. If mf.q is omitted, the training imputation follows the default behavior of impute.
eps	Convergence threshold. In impute.learn this controls the initial training-data imputation. In predict.impute.learn and impute.ood it controls early stopping for the prediction-time sweep.
...	For impute.learn, additional arguments passed to impute. For predict.impute.learn and impute.ood, additional arguments are currently ignored.
full.sweep.options	A list of options used when fitting the full sweep after the training data have been imputed. Recognized entries include ntree, nodesize, nsplit, mtry, splitrule, bootstrap, sampsize, samptype, perf.type, rfq, save.memory, importance, and proximity. Unknown entries are ignored with a warning.
target.mode	Determines which variables receive a saved full-sweep learner. The default "missing.only" saves learners only for variables that were missing in the training data. The option "all" saves a learner for every variable. If the training data are complete, target.mode = "all" must be used. For the broadest OOD coverage, target.mode = "all" is recommended so every deployment-time variable can be reconstructed.
deployment.xvars	Controls which predictors are assumed to be available later when the saved imputer is used on new data. If NULL, all columns except the target are used. If a character vector, the same predictor set is used for all targets. If a named list, each target can have its own predictor set. By default, all non-target columns are eligible predictors, so users should exclude outcomes, future information, identifiers, or any variables that will not be available at deployment time.

anonymous — If TRUE, uses rfsrc.anonymous when fitting the full sweep. This usually reduces the size of the saved object.

learner.prefix, learner.root — Names used when writing saved full-sweep learners to disk.

out.dir — Optional output directory. If supplied and save.on.fit = TRUE, the manifest and the saved full-sweep learners are written to this directory during fitting. This requires the **fst** package because learners are serialized with fast.save.

wipe — If TRUE, removes an existing output directory before writing a new one.

keep.models — If TRUE, keeps the fitted full-sweep learners in memory in the returned object. At least one storage mode must be enabled: either keep.models = TRUE or out.dir with save.on.fit = TRUE.

keep.ximp — If TRUE, keeps the completed training data in the returned object. This is not required for later prediction.

save.on.fit — If TRUE and out.dir is supplied, writes the imputer to disk during fitting.

save.ood — If TRUE, computes and stores an OOD reference. The reference is built during training from target-wise out-of-bag reconstruction discrepancies, their target-wise calibrated training scores, and a default row-level weighted mean using the saved OOD target weights. If no weight is supplied at fit time, equal target weights are used. The saved target-wise training-score matrix allows impute.ood to rebuild a calibrated row-level percentile for arbitrary target subsets, test-time weight overrides, and alternate row aggregates besides the weighted mean.

object — An object returned by impute.learn or load.impute.learn.

path — Directory containing a saved imputer. Save and load operations require the **fst** package because learners are read and written with fast.save and fast.load.

targets — Optional subset of target variables to load, update, or score. Unknown names are ignored with a warning. For impute.ood, row-level percentile calibration is rebuilt for the requested target subset from the saved target-wise training OOD scores whenever those scores are available in the manifest.

lazy — If TRUE, saved learners are loaded only when they are needed. If FALSE, all saved learners are loaded at once.

newdata — New data to be imputed or scored. Missing columns are added and extra columns are dropped to match the training schema. Unseen factor levels are converted to NA for harmonization, but they are also tracked row-wise. In impute.ood, any row containing an unseen factor level is flagged and its row-level OOD score is set to the maximum value.

max.predict.iter — Maximum number of full-sweep passes applied to newdata before the saved learner bank is used for OOD reconstruction or returned prediction-time imputations.

restore.integer — If TRUE, integer columns in the returned data are rounded and restored as integers. Factor columns are always conformed back to the training schema. The package operates on real-valued and factor variables; inputs that are not real-valued are coerced to factors during preprocessing when possible, otherwise an error is raised.

`cache.learners` How saved learners are reused during prediction or OOD scoring. For `predict.impute.learn`, the default `"session"` loads each needed learner once per call. The option `"none"` reloads a learner every time it is needed. The option `"all"` loads all requested learners before work starts. For `impute.ood`, `"all"` is the default because the saved learner bank is typically reused once for predictor-side completion and again for target reconstruction.

`weight` Optional nonnegative target weights used for row-level OOD aggregation. In `impute.learn`, these weights define the default row-level OOD weighting scheme stored in the manifest. In `impute.ood`, they define the active row-level weighting scheme for the current scoring call. If supplied as a named vector, entries are matched to targets by name; omitted targets are set to zero, and extra names are ignored. If omitted in `impute.ood`, the saved training-time OOD weights are used automatically. Because the fit stores target-wise training OOD scores, `score.percentile` can be recalibrated automatically for test-time weight overrides rather than being limited to the original training-time weights.

`aggregate` Row-level aggregation metric used by `impute.ood` to combine calibrated target-wise OOD scores. The default `"bounded.product"` applies a weighted product of the form $1 - \prod_j (1 - u_j + \varepsilon)^{\tilde{w}_j}$ where u_j are row level calibrated scores for target feature j. `"weighted.mean"` is the weighted average. `"weighted.lp"` applies a weighted Minkowski L_p aggregation to the calibrated target scores. `"weighted.lp.log"` first applies the tail-stretching transform $-\log(1-u_j+\varepsilon)$ to each calibrated target score u_j and then applies the weighted L_p aggregation. `"top.k"` averages only the k largest scoreable target scores among the positively weighted targets. When the fit stores target-wise training OOD scores, `score.percentile` is rebuilt for the requested aggregate as well as the requested targets and weights.

`aggregate.args` Optional list of tuning arguments for `aggregate`. Recognized entries are `p` for `"weighted.lp"` and `"weighted.lp.log"`, `k` (or `top.k`) for `"top.k"`, and `eps` for `"weighted.lp.log"` and `"bounded.product"`. The default values are `p = 2`, `k = 1`, and `eps = 1e-12`. Unknown entries are ignored with a warning.

`return.details` If `TRUE`, `impute.ood` returns the per-target discrepancy and calibrated target-score matrices and a row-by-variable unseen-level mask in the diagnostics.

Details

A predictive imputer is calculated in two stages.

The training data are first normalized to a data frame. Variables that are not real-valued are coerced to factors when possible; otherwise fitting stops with an error. Rows and columns that are entirely missing are removed before the training schema is stored.

If the resulting training data contain missing values, the first stage uses `impute` to complete the training data. The imputation engine is chosen in exactly the same way as for `impute` itself. In particular, `mf.q = 1` gives standard `missForest`, `mf.q > 1` gives the multivariate `missForest` generalization, and if `mf.q` is omitted the default `impute` behavior is used. If the training data are already complete and `target.mode = "all"`, this initial imputation step is skipped.

In the second stage, a full sweep is fit on the completed training data. For each target selected by `target.mode`, rows where that target was observed are used to fit a forest with that target on the left-hand side and the selected deployment predictors on the right-hand side. The saved learner bank

therefore depends on `deployment.xvars`. The `formula` argument affects the initial training-data imputation step, but it does not define the saved predictor bank for the later test-time sweep.

By default, `deployment.xvars = NULL` allows every non-target column to be used as a predictor. This is convenient, but it can also introduce leakage if the training data include outcomes, future-only variables, identifiers, or any fields that will not be available when the learned imputer is applied to new data. Restrict `deployment.xvars` when that is a concern.

When the imputer is saved to disk, each full-sweep learner is written separately using `fast.save`. Loading uses `fast.load`. In practice this gives a small manifest plus a directory of saved learners. The **fst** package is therefore required for save and load operations. The explicit save method can write learners either from memory or by reloading them from an attached saved path.

Prediction starts by matching `newdata` to the training schema, filling missing values with training means or modes, and then applying one or more full-sweep passes. Only the targets selected by `target.mode` are updated by saved learners.

If `target.mode = "missing.only"`, a variable that was complete in training but missing in new data is initialized from the training fit but does not receive a model-based update. Use `target.mode = "all"` if missing values may appear later in any variable. Complete training data also require `target.mode = "all"`, because otherwise there are no missing variables from which to determine the saved targets.

If `save.ood = TRUE`, the fit also stores an OOD reference in the manifest. For each saved target, the out-of-bag prediction from the fitted learner is compared with the observed training value to form a target-wise reconstruction discrepancy. Continuous and integer targets use absolute reconstruction error. Factor targets prefer the negative log predictive probability assigned to the observed class. When class probabilities are unavailable, unordered factors fall back to a 0/1 mismatch score and ordered factors fall back to a scaled rank distance.

The row-level OOD calibration stored at fit time is built by aggregating the target-wise training scores with a weighted mean using `weight`. If `weight` is omitted at fit time, all saved OOD targets receive weight 1. If a named vector is supplied, entries are matched by target name, omitted saved OOD targets receive weight 0, and the resulting weighting scheme is carried forward in the manifest for later deployment-time scoring.

`impute.ood` first completes the predictor side of `newdata` using the same harmonization, initialization, and iterative sweep logic used by `predict.impute.learn`. It then reconstructs each requested target directly from its saved learner and compares the reconstruction with the observed value. Raw target discrepancies are converted to target-wise OOD scores using the saved target-specific training references, which places continuous and factor targets on a common scale.

The row-level OOD score combines those calibrated target-wise scores over the targets that are both observed and scoreable for that row. By default, `impute.ood` uses a bounded product rule, but the row aggregate can be changed to weighted mean, a weighted L_p rule, a log-tail weighted L_p rule, or a top-k rule. This makes it possible to explore row scores that are more sensitive to sparse but severe coordinate shifts. By default, `impute.ood` reuses the same OOD weights saved during `impute.learn`, so a pipeline can fix its weighting scheme once upstream and carry it forward automatically.

A second component, `score.percentile`, is obtained by rebuilding the row-level training reference from the saved target-wise training OOD scores using the requested target subset, the active weight vector, and the active row aggregate. This means percentile calibration remains available when the user leaves the saved weights in place, overrides them at test time, scores only a subset of the saved OOD targets, or experiments with alternate row aggregates.

Unseen factor levels are tracked row-wise during harmonization. Because such values are immediate anomalies relative to the training schema, impute.ood flags those rows and assigns them the maximum row-level score. If the unseen level occurs in a scored target itself, the corresponding target-level discrepancy is also treated as maximal.

Value

impute.learn returns an object of class c("impute.learn.rfsrc", "impute.learn"). The object contains a manifest, optionally the fitted full-sweep learners, optionally the completed training data, and optionally a path to the saved imputer on disk. If save.ood = TRUE, the manifest also contains an ood component storing compact target-wise OOD references, the saved row-by-target training OOD score matrix used for later percentile recalibration, and the default OOD aggregation weights.

load.impute.learn returns an object of the same class.

predict.impute.learn returns a data frame with imputed values overlaid. An attribute named "impute.learn.info" contains prediction-time diagnostics such as the number of sweep passes, pass-difference history, caching mode, disk-load counts, schema harmonization details, row-wise unseen-factor flags, and any targets skipped because a learner was unavailable or a prediction failed.

impute.ood returns an object of class c("impute.ood.rfsrc", "impute.ood"). It is a list with the following components:

- score: the row-level aggregate of calibrated target-wise OOD scores under the requested aggregate and weight. Larger values indicate greater out-of-distribution behavior. For aggregate = "weighted.lp.log", this raw score is on a positive unbounded scale.
- score.percentile: the percentile of score relative to a row-level training reference rebuilt from the saved target-wise training OOD scores for the requested targets, weights, and row aggregate. For legacy fitted objects that do not contain those saved training scores, the original saved row-level reference is used when possible; otherwise NA.
- targets.used: the number of weighted targets that contributed to each row-level score.
- target.score: optional matrix of target-wise calibrated OOD scores, returned when return.details = TRUE.
- target.delta: optional matrix of raw target-wise reconstruction discrepancies, returned when return.details = TRUE.
- info: a list of diagnostics including harmonization details, row-wise unseen-factor flags, learner-loading information, the active row aggregate and its arguments, whether the saved row-level calibration was used, and any target-specific issues.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

impute.rfsrc, rfsrc, and predict.rfsrc.

Examples

```
## ------------------------------------------------------------
## small data example: uses missForest for impute engine
## ------------------------------------------------------------

set.seed(101)
aq <- airquality[, c("Ozone", "Solar.R", "Wind", "Temp", "Month")]
aq$Month <- factor(aq$Month)

id <- sample(1:nrow(aq), 100)
train <- aq[id, ]
test <- aq[-id, ]

fit <- impute.learn(
  data = train,
  ntree = 25,
  mf.q = 1,
  max.iter = 5,
  full.sweep.options = list(ntree = 25, nsplit = 5)
)

test.imp <- predict(fit, test, max.predict.iter = 2, verbose = FALSE)
head(test.imp)

## OOD scoring is most informative when every deployment-time
## variable can be reconstructed, so target.mode = "all" is recommended.
## Optional named OOD weights can also be supplied here. Any omitted
## targets receive weight 0, and the saved weights are reused
## automatically later by impute.ood().
ood.fit <- impute.learn(
  data = train,
  ntree = 25,
  mf.q = 1,
  max.iter = 5,
  target.mode = "all",
  save.ood = TRUE,
  full.sweep.options = list(ntree = 25, nsplit = 5),
  verbose = FALSE
)

ood <- impute.ood(ood.fit, test, return.details = TRUE, verbose = FALSE)
head(ood$score)
head(ood$score.percentile)

## try a more spike-sensitive row aggregate
ood.lp <- impute.ood(ood.fit, test,
                     aggregate = "weighted.lp",
                     aggregate.args = list(p = 4),
```

```
                      verbose = FALSE)
head(ood.lp$score.percentile)


## Not run:
## ------------------------------------------------------------
## Save the learned imputer to disk and load it later.
## This explicit save example writes learners kept in memory.
## Uses missForest for the impute engine.
## ------------------------------------------------------------

bundle.dir <- file.path(tempdir(), "aq.imputer")

fit <- impute.learn(
  data = train,
  ntree = 25,
  mf.q = 1,
  max.iter = 5,
  full.sweep.options = list(ntree = 25, nsplit = 5),
  keep.models = TRUE,
  verbose = FALSE
)

save.impute.learn(fit, bundle.dir, verbose = FALSE)
imp <- load.impute.learn(bundle.dir, lazy = TRUE, verbose = FALSE)
test.imp <- predict(imp, test, max.predict.iter = 2, verbose = FALSE)

unlink(bundle.dir, recursive = TRUE)



## ------------------------------------------------------------
## Challenging example with factors, uses save/reload
## ------------------------------------------------------------

## load pbc, convert everything to factors
data(pbc, package = "randomForestSRC")
dta <- data.frame(lapply(pbc, factor))
dta$days <- pbc$days
dta$status <- dta$status

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

## even harder ... factor level not previously encountered in training
levels(test$stage) <- c(levels(test$stage), "fake")
test$stage[sample(seq_len(nrow(test)), 10)] <- "fake"

## train forest
fit <- suppressWarnings(
```

```
  impute.learn(Surv(days, status) ~ ., train,
               target.mode = "all",
               save.ood = TRUE,
               keep.models = TRUE)
)

## save/reload
bundle.dir <- file.path(tempdir(), "pbc.imputer")
save.impute.learn(fit, bundle.dir, verbose = FALSE)
imp <- load.impute.learn(bundle.dir, lazy = TRUE, verbose = FALSE)
test.imp <- predict(imp, test, max.predict.iter = 2, verbose = FALSE)
ood <- impute.ood(imp, test, return.details = TRUE, verbose = FALSE)
which(ood$info$unseen.rows)
print(summary(test.imp))
unlink(bundle.dir, recursive = TRUE)

## End(Not run)
```

`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. Optionally, a final sweep can be performed to re-fit each variable that had original missingness on the final covariates and overwrite only its originally-missing entries.

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,
  full.sweep = FALSE,
  ...)
```

Arguments

`formula`	A symbolic model description. Can be omitted if outcomes are unspecified or if distinction between outcomes and predictors is unnecessary. Ignored for missForest.
`data`	A data frame containing variables to be imputed.
`ntree`	Number of trees grown for each imputation.
`nodesize`	Minimum terminal node size in each tree.

nsplit	Non-negative integer for specifying random splitting.
nimpute	Number of iterations for the missing data algorithm. Ignored for multivariate missForest, which iterates to convergence unless capped by max.iter.
fast	If TRUE, uses rfsrc.fast instead of rfsrc. Increases speed but may reduce accuracy.
blocks	Number of row-wise blocks to divide the data into. May improve speed for large data, but can reduce imputation accuracy. No action if unspecified.
mf.q	Enables missForest. Either a fraction (between 0 and 1) of variables treated as responses, or an integer indicating number of response variables. mf.q = 1 corresponds to standard missForest.
max.iter	Maximum number of iterations for multivariate missForest.
eps	Convergence threshold for multivariate missForest (change in imputed values).
ytry	Number of variables used as pseudo-responses in unsupervised forests. See Details.
always.use	Character vector of variables always included as responses in multivariate missForest. Ignored by other methods.
verbose	If TRUE, prints progress during multivariate missForest imputation.
full.sweep	If TRUE, performs a final sweep after the main imputation (both standard and missForest). For each variable that had any *original* missingness, a forest is fit on rows where the variable was observed using the *final* imputed covariates; predictions are then written back only to the originally missing cells. This can improve self-consistency across variables at the cost of extra computation.
...	Additional arguments passed to or from methods. Recognized advanced options include full.sweep.options (a list) controlling the final sweep hyperparameters: ntree (default 500), nodesize (default NULL), nsplit (default 10); and the standard rfsrc controls mtry, splitrule, bootstrap, sampsize, samptype that apply to the sweep.

Details

1. Before imputation, observations and variables with all values missing are removed.
2. A forest is grown and used solely for imputation. No ensemble statistics (e.g., error rates) are computed. Use this function when imputation is the only goal.
3. For standard imputation (not missForest), splits are based only on non-missing data. If a split variable has missing values, they are temporarily imputed by randomly drawing from in-bag, non-missing values to allow node assignment.
4. If mf.q is specified, multivariate missForest imputation is applied (Stekhoven and B\"uhlmann, 2012). A fraction (or integer count) of variables are selected as multivariate responses, predicted using the remaining variables with multivariate composite splitting. Each round imputes a disjoint set of variables, and the full cycle is repeated until convergence, controlled by max.iter and eps. Setting mf.q = 1 reverts to standard missForest. This method is typically the most accurate, but also the most computationally intensive.
5. If no formula is provided, unsupervised splitting is used. The default ytry is sqrt(p), where p is the number of variables. For each of mtry candidate variables, a random subset of ytry variables is selected as pseudo-responses. A multivariate composite splitting rule is applied, and the split is made on the variable yielding the best result (Tang and Ishwaran, 2017).

6. If no missing values remain after preprocessing, the function returns the processed data without further action.
7. All standard `rfsrc` options apply; see examples below for illustration.
8. *Optional final sweep*: if `full.sweep = TRUE`, a post-imputation sweep is performed for every variable with original missingness. Each such variable is re-fit on its observed rows using the final imputed covariates, and predictions overwrite only the originally missing entries. Defaults for the sweep are `ntree = 500`, `nodesize = NULL`, `nsplit = 10`, and can be customized via `full.sweep.options` passed through `....` This applies to both standard and missForest modes.

Value

Invisibly, the data frame containing the original data with imputed data overlaid.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

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

`impute.learn.rfsrc`, `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)

## optional final sweep (standard imputation)
pbc3.fs <- impute(f, data = pbc, splitrule = "random", nimpute = 5,
                  full.sweep = TRUE)
```

```
## ------------------------------------------------------------
## 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)

## final sweep with custom options (e.g., larger forest)
air3.fs <- impute(Ozone ~ ., data = airquality, nimpute = 5,
                  full.sweep = TRUE,
                  full.sweep.options = list(ntree = 1000, nodesize = 5, nsplit = 0,
                                            mtry = 3, splitrule = "random"))

## ------------------------------------------------------------
## 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
pbc.mv <- impute(data = pbc, mf.q = .10)

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

## missForest + final sweep
pbc.fast.fs <- impute(data = pbc, mf.q = 1, splitrule = "random",
                      full.sweep = TRUE)
```

`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 maximum interaction order for which minimal depth is calculated. Defaults to 2. Set max.order=0 to return first-order depths only. When max.order=0, conservative is automatically set to FALSE.
sub.order	Logical. If TRUE, returns the minimal depth of each variable conditional on every other variable. Useful for investigating variable interdependence. See Details.
conservative	Logical. If TRUE, uses a conservative threshold for selecting variables based on the marginal minimal depth distribution (Ishwaran et al., 2010). If FALSE, uses the tree-averaged distribution, which is less conservative and typically identifies more variables in high-dimensional settings.
...	Additional arguments passed to or from other methods.

Details

The maximal subtree for a variable *x* is the largest subtree in which the root node splits on *x*. The largest possible maximal subtree is the full tree (root node), though multiple maximal subtrees may exist for a variable. A variable may also have no maximal subtree if it is never used for splitting. See Ishwaran et al. (2010, 2011) for further discussion.

The minimal depth of a maximal subtree-called the *first-order depth*-quantifies the predictive strength of a variable. It is defined as the distance from the root node to the parent of the closest maximal subtree for *x*. Smaller values indicate stronger predictive impact. A variable is flagged as strong if its minimal depth is below the mean of the minimal depth distribution.

The *second-order depth* is the distance from the root to the second-closest maximal subtree of *x*. To request depths beyond first order, use the max.order option (e.g., max.order = 2 returns both first and second-order depths). Set max.order = 0 to retrieve first-order depths for each variable in each tree.

Set sub.order = TRUE to obtain the relative minimal depth of each variable *j* within the maximal subtree of another variable *i*. This returns a p x p matrix (with p the number of variables) whose entry (i,j) is the normalized relative depth of *j* in *i*'s subtree. Entry (i,i) gives the depth of *i* relative to the root. Read the matrix across rows to assess inter-variable relationships: small (i,j) entries suggest interactions between variables *i* and *j*.

For competing risks, all analyses are unconditional (non-event specific).

Value

Invisibly returns a list with the following components:

order	Matrix of order depths for each variable up to max.order, averaged over trees. The matrix has p rows and max.order columns, where p is the number of variables. If max.order = 0, returns a matrix of dimension p x ntree containing first-order depths for each variable by tree.
count	Average number of maximal subtrees per variable, normalized by tree size.
nodes.at.depth	List of vectors recording the number of non-terminal nodes at each depth level for each tree.

sub.order	Matrix of average minimal depths of each variable relative to others (i.e., conditional minimal depth matrix). NULL if sub.order = FALSE.
threshold	Threshold value for selecting strong variables based on the mean of the minimal depth distribution.
threshold.1se	Conservative threshold equal to the mean minimal depth plus one standard error.
topvars	Character vector of selected variable names using the threshold criterion.
topvars.1se	Character vector of selected variable names using the threshold.1se criterion.
percentile	Percentile value of minimal depth for each variable.
density	Estimated density of the minimal depth distribution.
second.order.threshold	Threshold used for selecting strong second-order depth variables.

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, 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

Investigates the effects of five dietary treatments on 21 liver lipids and 120 hepatic gene expressions in wild-type and PPAR-alpha deficient mice. A multivariate mixed random forest analysis is performed by regressing gene expression, diet, and genotype (x-variables) on lipid expression profiles (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 returned by a call to quantreg.
prbL	Lower quantile level, typically less than 0.5.
prbU	Upper quantile level, typically greater than 0.5.
m.target	Character string specifying the target outcome for multivariate families. If not provided, a default target is selected automatically.
crps	Logical. If TRUE, calculates the continuous ranked probability score (CRPS) and adds it to the plot.
subset	Optional vector specifying a subset of the data to be plotted. Defaults to plotting all data points.
xlab	Label for the x-axis.
ylab	Label for the y-axis.
...	Additional 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), 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	Character vector of x-variable names to include. If not specified, all variables are used.
target	For classification, an integer or character specifying the class of interest (default is the first class). For competing risks, an integer between 1 and J indicating the event of interest, where J is the number of event types. Default is the first event type.
m.target	Character value for multivariate families specifying the target outcome. If unspecified, a default is automatically chosen.
time	(Survival only) Time point at which the predicted survival value is evaluated, depending on surv.type.
surv.type	(Survival only) Type of predicted survival value to compute. See plot.variable details.
class.type	(Classification only) Type of predicted classification value to use. See plot.variable details.
partial	Logical. If TRUE, partial dependence plots are generated.
oob	Logical. If TRUE, out-of-bag predictions are used; otherwise, in-bag predictions are used.
show.plots	Logical. If TRUE, plots are displayed on the screen.
plots.per.page	Integer controlling the number of plots displayed per page.
granule	Integer controlling the coercion of continuous variables to factors (used to generate boxplots). Larger values increase coercion.
sorted	Logical. If TRUE, variables are sorted by variable importance.
nvar	Number of variables to plot. Defaults to all available variables.
npts	Maximum number of points used when generating partial plots for continuous variables.
smooth.lines	Logical. If TRUE, applies lowess smoothing to partial plots.
subset	Vector indicating which rows of x$xvar to use. Defaults to all rows. Important: do not define subset based on the original dataset (which may have been altered due to missing data or other processing); define it relative to x$xvar.
...	Additional arguments passed to or from other methods.

Details

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

1. For regression, the predicted response is plotted.
2. For classification, the plotted value is the predicted class probability for the class specified by `target`, or the most probable class (Bayes rule) depending on whether `class.type` is set to `"prob"` or `"bayes"`.
3. For multivariate families, the prediction corresponds to the outcome specified by `m.target`. If this is a classification outcome, `target` may also be used to indicate the class of interest.
4. For survival, the vertical axis shows the predicted value determined by `surv.type`, with the following options:
   - `mort`: Mortality (Ishwaran et al., 2008), interpreted as the expected number of events for an individual with the same covariates.
   - `rel.freq`: Relative frequency of mortality.
   - `surv`: Predicted survival probability at a specified time point (default is the median follow-up time), controlled via `time`.
5. For competing risks, the vertical axis shows one of the following quantities, depending on `surv.type`:
   - `years.lost`: Expected number of life-years lost.
   - `cif`: Cumulative incidence function for the specified event.
   - `chf`: Cause-specific cumulative hazard function.

   In all competing risks settings, the event of interest is specified using `target`, and `cif` and `chf` are evaluated at the time point given by `time`.

To generate partial dependence plots, set `partial = TRUE`. These differ from marginal plots in that they isolate the effect of a single variable X on the predicted value by averaging over all other covariates:

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

where $x_{i,o}$ denotes the observed values of all covariates other than X for individual i, and $\hat{f}$ is the prediction function. Generating partial plots can be computationally expensive; use a smaller value for `npts` to reduce the number of grid points evaluated for x.

Plot display conventions:

- For continuous variables: red points indicate partial values; dashed red lines represent an error band of two standard errors. Black dashed lines show the raw partial values. Use `smooth.lines = TRUE` to overlay a `lowess` smoothed line.
- For discrete (factor) variables: boxplots are used, with whiskers extending approximately two standard errors from the mean.
- Standard errors are provided only as rough indicators and should be interpreted cautiously.

Partial plots can be slow to compute. Setting `npts` to a small value can improve performance.

For additional flexibility and speed, consider using `partial.rfsrc`, which directly computes partial plot data and allows for greater customization.

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, 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,
  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,
  marginal.xvar = NULL,  ...)
```

Arguments

`object`	An object of class (`rfsrc`, `grow`) or (`rfsrc`, `forest`).
`newdata`	Test data. If omitted, the original training data is used.
`importance`	Method for computing variable importance (VIMP). See `vimp` for additional options including joint importance. See `holdout.vimp` for an alternative importance measure.
`get.tree`	Vector of integers specifying which trees to use for ensemble calculations. Defaults to all trees. Useful for extracting ensembles, VIMP, or proximity from specific trees. If specified, `block.size` is overridden to match the number of trees. See examples for per-tree VIMP extraction.
`block.size`	Controls the granularity of error rate and VIMP calculation. If `NULL`, error is reported only for the final tree. Set to an integer `k` to compute error every `k` trees. For VIMP, calculations are done in blocks of size `block.size`, balancing between tree-level and forest-level assessments.

na.action	Action to take when missing values are present. Options are "na.omit" (default), "na.random" for fast random imputation, or "na.impute" to use the imputation method in rfsrc.
outcome	Specifies whether predicted values should be based on the outcomes from the training data ("train", default) or test data. Ignored if newdata is missing or if test outcomes are unavailable.
perf.type	Optional metric for prediction, VIMP, and error. Currently used for classification and multivariate classification. Choices: "misclass" (default), "brier", and "gmean".
proximity	Whether to compute the proximity matrix for test observations. Options include "inbag", "oob", "all", TRUE, or FALSE. Not all options are valid in all contexts; TRUE is the safest choice.
distance	Whether to compute the distance matrix. Options are the same as for proximity.
forest.wt	Whether to compute the forest weight matrix. Options are the same as for proximity.
ptn.count	If nonzero, each tree is pruned to have this many terminal nodes. Only the terminal node membership is returned; no prediction is made. Default is ptn.count = 0.
var.used	If TRUE, records how many times each variable was used for splitting.
split.depth	If TRUE, returns minimal depth of each variable per case.
case.depth	If TRUE, returns a matrix of the depth at which each case first splits in each tree.
seed	Negative integer used to set the random seed.
do.trace	Number of seconds between progress updates during execution.
membership	If TRUE, returns terminal node membership and in-bag information.
marginal.xvar	Vector of variable names to marginalize over when calculating weights or proximity. If a variable is marginalized, its split does not partition the data; all cases are passed to both daughters. When all splits involve marginalized variables, terminal nodes contain the full dataset. When no marginalized variables are used, membership is unchanged. Default is NULL (no marginalization).
...	Additional arguments passed to or from other methods.

Details

Predicted values are obtained by "dropping" the test data down the trained forest-i.e., the forest grown using the training data. If the test data includes y-outcome values, performance metrics are also returned. Variable importance (VIMP), including joint VIMP, is returned if requested.

If no test data is supplied, the function uses the original training data and enters "restore" mode. This allows users to extract outputs from the trained forest that were not requested during the original grow call.

If outcome = "test", predictions are computed using y-outcomes from the test data (which must include outcome values). Terminal node statistics are recalculated using these outcomes, while the tree topology remains fixed from training. Error rates and VIMP are then computed by bootstrapping the test set and applying out-of-bagging to maintain unbiased estimates.

Set csv = TRUE to return case-specific VIMP, and cse = TRUE to return case-specific error rates. These apply to all families except survival. Both options can also be used at training time.

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 the test data (after handling missing values).

ntree Number of trees in the trained forest.

yvar Y-outcome values from the test data or original grow data (if newdata is missing).

yvar.names Character vector of response variable names.

xvar Data frame of test set predictor variables.

xvar.names Character vector of predictor variable names.

leaf.count Vector of length ntree giving the number of terminal nodes per tree.

proximity Proximity matrix computed on the test data.

forest The trained forest object.

forest.wt Forest weight matrix for test cases.

ptn.membership Matrix of pruned terminal node membership. Only returned if ptn.count > 0.

membership Matrix of terminal node membership for test cases. Each column corresponds to one tree.

inbag Matrix indicating how many times each case appears in the bootstrap sample for each tree.

var.used Number of times each variable was used in splitting.

imputed.indv Indices of test observations with missing values.

imputed.data Imputed version of the test data. Columns are ordered with responses first, followed by predictors.

split.depth Matrix or array recording minimal depth of each variable for each case, optionally by tree.

err.rate Cumulative out-of-bag (OOB) error rate, if y-outcomes are present.

importance Variable importance (VIMP) for the test data. May be NULL.

predicted Predicted values for the test data.

predicted.oob OOB predicted values. May be NULL depending on context.

quantile Estimated quantile values at the requested probabilities (quantile regression only).

quantile.oob OOB quantile values. May be NULL.

class (Classification only) Predicted class labels.

class.oob (Classification only) OOB predicted class labels.

regrOutput (Multivariate only) List of performance measures for multivariate regression outcomes.

clasOutput (Multivariate only) List of performance measures for multivariate categorical outcomes.

chf (Survival or competing risks) Cumulative hazard function (CHF).

chf.oob (Survival or competing risks) OOB CHF. May be NULL.

survival (Survival only) Survival function estimates.

survival.oob (Survival only) OOB survival function. May be NULL.

time.interest (Survival or competing risks) Sorted unique event times.

ndead (Survival or competing risks) Number of deaths observed.

cif (Competing risks only) Cumulative incidence function (CIF) for each event type.

cif.oob (Competing risks only) OOB CIF. May be NULL.

chf (Competing risks only) Cause-specific cumulative hazard function (CSCHF).

chf.oob (Competing risks only) OOB CSCHF. May be NULL.

Note

The dimensions and contents of returned objects depend on the forest family and whether y-outcomes are available in the test data. In particular, performance-related components (e.g., error rate, VIMP) will be NULL if y-outcomes are missing.

For multivariate families, predicted values, VIMP, error rates, and performance metrics are stored in the lists regrOutput and clasOutput. These can be accessed using the helper functions get.mv.predicted, get.mv.vimp, and get.mv.error.

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, 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)
```

```
## ------------------------------------------------------------
## multivariate forest example
## ------------------------------------------------------------

## train the forest
trn <- 1:20
o <- rfsrc(cbind(mpg, disp)~.,mtcars[trn,])

## print training results for each outcome
print(o, outcome.target="mpg")
print(o, outcome.target="disp")

## print test results for each outcome
p <- predict(o, mtcars[-trn,])
print(p, outcome.target="mpg")
print(p, outcome.target="disp")
```

`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`), 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

```
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

Methods

1. 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"`
2. 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.
3. 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"`.

Quantile-error metrics

1. When trees are grown with the quantile splitting rule (`splitrule = "quantile"`), `print.rfsrc` reports error metrics aligned with the quantile objective: (i) the standardized continuous ranked probability score (CRPS) computed from the estimated conditional CDF, and (ii) the pinball loss at selected quantile levels τ.
2. These metrics supplement the usual regression error rate printed by `print.rfsrc`. The line labeled "Requested performance error" remains the OOB mean squared error (MSE) by default (and is the quantity used to print `R squared`), regardless of the split rule. These quantile-metrics are printed in addition to MSE and do not change model fitting.
3. Pinball levels can be controlled via `options(rfsrc.pinball.taus = c(0.1, 0.5, 0.9))`. The CRPS curves used by `plot.quantreg` can be obtained with `get.quantile.crps()`.

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")

## ------------------------------------------------------------
## quantile-metrics provided in print.rfsrc output
## ------------------------------------------------------------
op <- options(rfsrc.pinball.taus = c(0.1, 0.5, 0.9))

o <- quantreg(mpg ~ ., data = mtcars)

print(o)

options(op)  ## restore options

## ------------------------------------------------------------
## 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 canonical call
## ------------------------------------------------------------

if (library("mlbench", logical.return = TRUE)) {
  data(BostonHousing)
  o <- quantreg(medv ~ ., BostonHousing, nodesize = 1)
  print(o)
  plot.quantreg(o)
}


## ------------------------------------------------------------
## quantile regression for Boston Housing using forest method
## also demonstrates quantile statistics
## ------------------------------------------------------------

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 implementation of random forests (Breiman, 2001) for regression, classification, survival analysis (Ishwaran et al., 2008), competing risks (Ishwaran et al., 2012), multivariate outcomes (Segal and Xiao, 2011), unsupervised learning (Mantero and Ishwaran, 2020), quantile regression (Meinshausen, 2006; Zhang et al., 2019; Greenwald and Khanna, 2001), and imbalanced q-classification (O'Brien and Ishwaran, 2019).

The package supports both deterministic and randomized splitting rules (Geurts et al., 2006; Ishwaran, 2015) across all families. Multiple types of variable importance (VIMP) are available, including holdout VIMP and confidence regions (Ishwaran and Lu, 2019), for both individual and grouped variables. Variable selection can be performed using minimal depth (Ishwaran et al., 2010, 2011). Fast interfaces for missing data imputation are provided using several forest-based algorithms (Tang and Ishwaran, 2017).

Highlighted updates:

1. For survival and competing risk analysis, concordance-based performance is now computed using Uno inverse-probability-of-censoring weighting (Uno et al. 2001). This affects all survival performance values derived from the concordance index, including out-of-bag and test error rates and variable importance (VIMP). To revert to unweighted Harrell concordance, set `use.uno = FALSE` when fitting a survival forest. Concordance calculations now use an efficient $O(n \log n)$ algorithm based on a binary indexed tree (Fenwick 1994, Therneau 2024), replacing the naive $O(n^2)$ pairwise computation for large n.
2. For variable selection, we recommend using **VarPro**, an R package for model-independent variable selection using rule-based variable priority. It supports regression, classification, survival analysis, and includes a new mode for unsupervised learning. See https://www.varprotools.org for more information.
3. For computational speed, the default VIMP method has changed from "permute" (Breiman-Cutler permutation) to "anti" (`importance = "anti"` or `importance = TRUE`). While faster, this may be less accurate in settings such as highly imbalanced classification. To revert to permutation VIMP, use `importance = "permute"`.

This is the main entry point to the **randomForestSRC** package. For more information on OpenMP support and the package as a whole, see `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,
  use.uno = 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,
  ...)

## convenient interface for growing a CART tree
rfsrc.cart(formula, data, ntree = 1, mtry = ncol(data),
   bootstrap = "none", nsplit = 0, ...)
```

Arguments

formula	A formula describing the model to fit. Interaction terms are not supported. If missing, unsupervised splitting is used.
data	Data frame containing the response and predictor variables.
ntree	Number of trees to grow.
mtry	Number of candidate variables randomly selected at each split. Defaults: regression uses p/3, others use sqrt(p); rounded up.
ytry	Number of pseudo-response variables randomly selected for unsupervised splitting. Default is 1.
nodesize	Minimum terminal node size. Defaults: survival/competing risks (15), regression (5), classification (1), mixed/unsupervised (3).
nodedepth	Maximum tree depth. Ignored by default.
splitrule	Splitting rule. See Details.
nsplit	Number of random split points per variable. 0 uses all values (deterministic). Default is 10.
importance	Variable importance (VIMP) method. Choices: FALSE, TRUE, "none", "anti", "permute", "random". Default is "none". VIMP can be computed later using vimp or predict.
block.size	Controls frequency of cumulative error/VIMP updates. Default is 10 if importance is requested; otherwise NULL. See Details.
bootstrap	Bootstrap method. Options: "by.root" (default), "by.user", or "none" (no OOB error possible).

samptype	Sampling type for by.root bootstrap. Options: "swor" (without replacement, default), "swr" (with replacement).
samp	Bootstrap weights (only for bootstrap="by.user"). A matrix of size n x ntree giving in-bag counts per tree.
membership	Return inbag and terminal node membership?
sampsize	Bootstrap sample size (used when bootstrap="by.root"). Defaults: 0.632 x n for swor; n for swr. Can also be numeric.
na.action	Missing data handling. "na.omit" (default) removes rows with any NA; "na.impute" performs fast internal imputation. See also impute.
nimpute	Number of iterations for internal imputation. If >1, OOB error rates may be optimistic.
ntime	For survival models: number or grid of time points used in ensemble estimation. If NULL or 0, uses all event times.
cause	For competing risks: event of interest (1 to J), or a vector of weights over all J events. Defaults to an average over all events.
perf.type	Optional performance metric for prediction, VIMP, and error. Defaults to the family-specific metric. "none" disables performance. See Details.
proximity	Compute proximity matrix? Options: "inbag", "oob", "all", TRUE (inbag), or FALSE.
distance	Compute pairwise distances between cases? Similar options as proximity. See Details.
forest.wt	Return forest weight matrix? Same options as proximity. Default is TRUE (inbag).
xvar.wt	Optional weights on x-variables for sampling at splits. Does not need to sum to 1. Defaults to uniform.
yvar.wt	Weights on response variables (for multivariate regression). Used when y is high-dimensional.
split.wt	Weights applied to each variable's split statistic. Higher weight increases likelihood of splitting.
case.wt	Sampling weights for cases in the bootstrap. Higher values increase selection probability. See class imbalance example.
case.depth	Return matrix recording depth of first split for each case? Default is FALSE.
forest	Save forest object for future prediction? Set FALSE if prediction is not needed.
use.uno	Logical. If TRUE (default for survival), concordance-based performance is computed using inverse-probability-of-censoring weights as proposed by Uno et al. (2011). This affects all survival performance values derived from the C-index, including error rates (C-error) and variable importance (VIMP). Set use.uno = FALSE to use the unweighted Harrell concordance. Applies only to survival families.
save.memory	Reduce memory usage by avoiding storage of prediction quantities. Recommended for large survival or competing risk forests.
var.used	Return variable usage statistics? Options: FALSE, "all.trees", "by.tree".

split.depth	Return minimal depth of splits for each variable? Options: FALSE, "all.trees", "by.tree". See Details.
seed	Integer seed for reproducibility (negative values only).
do.trace	Print progress updates every do.trace seconds.
...	Additional arguments passed to or from other methods.

Details

1. *Types of forests*

   The type of forest is automatically inferred from the outcome and formula. Supported forest types include:

   - Regression forests for continuous outcomes.
   - Classification forests for factor outcomes.
   - Multivariate forests for continuous, categorical, or mixed outcomes.
   - Unsupervised forests when no outcome is specified.
   - Survival forests for right-censored time-to-event data.
   - Competing risk forests for multi-event survival settings.

2. *Splitting*

   (a) Splitting rules are set using the splitrule option.
   (b) Random splitting is invoked via splitrule = "random".
   (c) Use nsplit to enable randomized splitting and improve speed; see *Improving computational speed*.

3. *Available splitting rules*

   - **Regression**
     (a) "mse" (default): weighted mean squared error (Breiman et al., 1984).
     (b) "quantile.regr": quantile regression via check-loss; see quantreg.rfsrc.
     (c) "la.quantile.regr": local adaptive quantile regression.
   - **Classification**
     (a) "gini" (default): Gini index.
     (b) "auc": AUC-based splitting; appropriate for imbalanced data.
     (c) "entropy": entropy-based splitting.
   - **Survival**
     (a) "logrank" (default): log-rank splitting.
     (b) "bs.gradient": Brier score gradient splitting. Uses 90th percentile of observed times by default, or set prob.
     (c) "logrankscore": log-rank score splitting.
   - **Competing risks** (see Ishwaran et al., 2014)
     (a) "logrankCR" (default): Gray's test-based weighted log-rank splitting.
     (b) "logrank": cause-specific weighted log-rank; use cause to target specific events.
   - **Multivariate**
     (a) Default: normalized composite splitting (Tang and Ishwaran, 2017).

(b) `"mahalanobis"`: Mahalanobis splitting with optional covariance matrix; for multivariate regression.

- **Unsupervised** Splitting uses pseudo-outcomes and the composite rule. See `sidClustering` for advanced unsupervised analysis.
- **Custom splitting** Custom rules can be defined using `splitCustom.c`. Up to 16 rules per family are allowed. Use `"custom"`, `"custom1"`, etc. Compilation required.

4. *Improving computational speed*

   See `rfsrc.fast`. Strategies include:

   - Increase `nodesize`.
   - Set `save.memory = TRUE` for large survival or competing risk models.
   - Set `block.size = NULL` to avoid repeated cumulative error computation.
   - Use `perf.type = "none"` to disable VIMP and C-index calculations.
   - Set `nsplit` to a small integer (e.g., 1-10).
   - Reduce bootstrap size with `sampsize`, `samptype`.
   - Set `ntime` to a coarse grid (e.g., 50) for survival models.
   - Pre-filter variables; use `max.subtree` for fast variable selection.

5. *Prediction Error*

   Error is computed using OOB data:

   - Regression: mean squared error.
   - Classification: misclassification rate, Brier score, AUC.
   - Survival: C-error = 1 - Harrell's concordance index.

   If `bootstrap = "none"`, OOB error is unavailable. Use `predict.rfsrc` for cross-validation error instead.

6. *Variable Importance (VIMP)*

   VIMP methods:

   - `"permute"`: permutation VIMP (Breiman-Cutler).
   - `"random"`: randomized left/right assignment.
   - `"anti"` (default): anti-split assignment.

   The `block.size` option controls granularity. For confidence intervals, see `subsampling`. Also see `holdout.vimp` for a more conservative variant.

7. *Multivariate Forests*

   Use:

   ```
   rfsrc(Multivar(y1, ..., yd) ~ ., data)
   ```

   or

   ```
   rfsrc(cbind(y1, ..., yd) ~ ., data)
   ```

   Use `get.mv.formula`, `get.mv.predicted`, `get.mv.error` for multivariate extraction.

8. *Unsupervised Forests*

   Use:

   ```
   rfsrc(data = X)
   ```

   or

```
rfsrc(Unsupervised(ytry) ~ ., data = X)
```

Random subsets of `ytry` pseudo-responses are used for each `mtry` variable. No performance metrics are computed.

9. *Survival, Competing Risks*
   - Survival: use `Surv(time, status) ~ .`. Status must be 0 (censored) or 1 (event).
   - Competing risks: status = 0 (censored), 1-J (event types). Use `cause` to target specific events.
   - Larger `nodesize` is typically needed for competing risks.
10. *Missing data imputation*
    Use `na.action = "na.impute"`. Iteration with `nimpute > 1` replaces missing values using OOB predictions. Observations or variables with all missing values are removed.
11. *Allowable data types and factors*
    Variables must be numeric, integer, factor, or logical. Non-factors are coerced to numeric. For unordered factors, all complementary subsets are considered for splits.
    Factor levels are mapped to ensure consistency across training/test data. Consider converting factors to numeric for high-dimensional settings.

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 after applying `na.action`.

ntree Number of trees grown.

mtry Number of variables randomly selected at each node.

nodesize Minimum terminal node size.

nodedepth Maximum depth allowed for each tree.

splitrule Splitting rule used.

nsplit Number of random split points.

yvar Response values.

yvar.names Character vector of response variable names.

xvar Data frame of predictor variables.

xvar.names Character vector of predictor variable names.

xvar.wt Non-negative weights specifying the selection probability of each variable.

split.wt Non-negative weights adjusting each variable's split statistic.

cause.wt Weights for composite competing risk splitting.

leaf.count Number of terminal nodes per tree. A value of 0 indicates a rejected tree (may occur with missing data); a value of 1 indicates a stump.

proximity Proximity matrix indicating how often case pairs fall in the same terminal node.

forest Forest object, returned if `forest=TRUE`. Required for prediction and most wrappers.

forest.wt Forest weight matrix.

membership Terminal node membership matrix (rows: trees; columns: cases).

inbag Inbag count matrix (rows: trees; columns: cases).

var.used Number of times each variable is used to split a node.

imputed.indv Indices of individuals with missing values.

imputed.data Imputed dataset with responses followed by predictors.

split.depth Matrix or array recording minimal split depth of variables by case and tree.

err.rate Cumulative OOB error rate.

err.block.rate Cumulative error per ensemble block (size defined by `block.size`). If `block.size` = 1, error per tree.

importance Variable importance (VIMP) for each predictor.

predicted In-bag predicted values.

predicted.oob Out-of-bag (OOB) predicted values.

class (Classification) In-bag predicted class labels.

class.oob (Classification) OOB predicted class labels.

regrOutput (Multivariate) List of performance results for continuous outcomes.

clasOutput (Multivariate) List of performance results for categorical outcomes.

survival (Survival) In-bag survival functions.

survival.oob (Survival) OOB survival functions.

chf (Survival or competing risks) In-bag cumulative hazard function.

chf.oob (Survival or competing risks) OOB cumulative hazard function.

time.interest (Survival or competing risks) Unique sorted event times.

ndead (Survival or competing risks) Total number of observed events.

cif (Competing risks) In-bag cumulative incidence function by cause.

cif.oob (Competing risks) OOB cumulative incidence function by cause.

Note

Values returned by the forest depend on the family:

- **Regression**: `predicted` and `predicted.oob` are vectors of predicted values.
- **Classification**: `predicted` and `predicted.oob` are matrices of class probabilities. VIMP and performance metrics are returned as a matrix with `J+1` columns (J = number of classes). The first column ("all") gives unconditional results; remaining columns give class-conditional results.
- **Survival**: `predicted` contains mortality estimates (Ishwaran et al., 2008). These are calibrated to the number of expected events under identical covariate profiles. Also returned are matrices of the survival function and CHF for each individual over `time.interest`.
- **Competing risks**: `predicted` contains expected life years lost by cause (Ishwaran et al., 2013). Also returned are three-dimensional arrays for CIF and CSCHF indexed by case, time, and event type.
- **Multivariate**: Predictions, VIMP, and error rates are returned in `regrOutput` and `clasOutput`. Use `get.mv.predicted`, `get.mv.vimp`, and `get.mv.error` to extract results.

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.

Fenwick, P.M. (1994). A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336.

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.

Therneau, T.M. (2024). Concordance. Vignette in the **survival** R package.

Uno, H., Cai, T., Pencina, M.J., D'Agostino, Ralph B. and Wei, L-J. (2011). On the C-statistics for evaluating overall adequacy of risk prediction procedures with censored survival data. *Statistics in Medicine*, 30:1105-1117.

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

See Also

`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`,

`subsample.rfsrc`,

`tune.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
## use uno weights throughout
rfsrc.obj <- rfsrc(surv.f, pbc.na)
uno.wt <- rfsrc.obj$forest$uno.weights$weight
cat("out-of-bag Cox Analysis ...", "\n")
cox.err <- sapply(1:100, function(b) {
  if (b%%10 == 0) cat("cox bootstrap:", b, "\n")
  train <- sample(1:nrow(pbc.na), nrow(pbc.na), replace = TRUE)
  cox.obj <- tryCatch({coxph(surv.f, pbc.na[train, ])}, error=function(ex){NULL})
  if (!is.null(cox.obj)) {
    get.cindex(pbc.na$days[-train],
               pbc.na$status[-train],
               predict(cox.obj, pbc.na[-train, ]),
               weight=uno.wt[-train])
  } else NA
})
cat("\n\tOOB error rates (with Uno-IPCW):\n")
cat("\tRSF            : ", get.mv.error(rfsrc.obj), "\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 is carefully modified to ensure that the original training data is not retained. This enables users to share the trained forest with others without disclosing the underlying data.

Usage

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

Arguments

`formula`	A symbolic description of the model to be fit. If missing, unsupervised splitting is performed.
`data`	A data frame containing the y-outcome and x-variables.
`forest`	Logical. Should the forest object be returned? Required for prediction on new data and by many other package functions.
`...`	Additional arguments passed to `rfsrc`. See the `rfsrc` help file for full details.

Details

This function calls `rfsrc` and returns a forest object with the original training data removed. This enables users to share their forest while preserving the privacy of their data.

To enable prediction on new (test) data, certain minimal information from the training data must still be retained. This includes:

- Names of the original variables.
- For factor variables, the levels of each factor.
- Summary statistics used for imputation: the mean for continuous variables and the most frequent class for factors.
- Tree topology, including split points used to grow the trees.

For maximal privacy, users are strongly encouraged to replace variable names with non-identifiable labels and convert all variables to continuous format when possible. If factor variables are used, their levels should also be anonymized. However, the user is solely responsible for de-identifying the data and verifying that privacy is maintained. **We provide NO GUARANTEES regarding data confidentiality.**

Missing data handling: Anonymous forests do not support imputation of training data. The option `na.action = "na.impute"` is automatically downgraded to `"na.omit"`. If training data contain missing values, we recommend pre-imputing them using `impute`.

Test data, however, *can* be imputed at prediction time:

- `na.action = "na.impute"` performs a fast imputation by replacing missing values with the training mean (for numeric variables) or most frequent class (for factors).
- `na.action = "na.random"` uses a fast random draw from training distributions for imputation.

Although anonymous forests are compatible with many package functions, they are only guaranteed to work with functions that do not explicitly require access to the original 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, 2021). 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	A data frame containing the unsupervised data.
method	Clustering method. Default is "sid", which implements SID clustering using Staggered Interaction Data (Mantero and Ishwaran, 2021). An alternative approach reformulates the problem as a two-class supervised learning task using artificial data, per Breiman (2003) and Shi-Horvath (2006). Mode 1 is specified via "sh", "SH", "sh1", or "SH1"; Mode 2 via "sh2" or "SH2". A third method, "unsupv", uses a plain unsupervised forest where the data act as both features and responses, split using the multivariate rule. This is faster than SID but may be less accurate.
k	Requested number of clusters. Can be a single integer or a vector. If a scalar, returns a vector assigning each observation to a cluster. If a vector, returns a matrix with one column per requested value of k, each containing a clustering assignment.
reduce	Logical. If TRUE, applies a variable reduction step via holdout VIMP. This is conservative and computationally intensive but has strong false discovery control. Applies only when method = "sid".
ntree	Number of trees used in the main SID clustering analysis.
ntree.reduce	Number of trees used in the holdout VIMP step during variable reduction. See holdout.vimp for details.
fast	Logical. If TRUE, uses the fast implementation rfsrc.fast instead of rfsrc. Improves speed at the cost of accuracy.
x.no.sid	Variables to exclude from SID transformation. Can be either a separate data frame (not overlapping with data) or a character vector of variable names from data. These variables will be included in the final design matrix without SID processing. Applies only when method = "sid".
use.sid.for.x	Logical. If FALSE, reverses the roles of features and responses in the SID process. Staggered interactions are applied to the outcome rather than to features. This option is slower and generally less effective. Included for legacy compatibility. Applies only when method = "sid".
x.only	Character vector specifying which variables to use as features. Applies only when method = "unsupv".
y.only	Character vector specifying which variables to use as multivariate responses. Applies only when method = "unsupv".
dist.sharpen	Logical. If TRUE (default), applies Euclidean distance to the forest distance matrix to improve clustering ("distance sharpening"). The resulting distance matrix will not be bounded between 0 and 1. Turning this off speeds up computation but may reduce clustering quality. Applies only when method = "sid" or "unsupv".
...	Additional arguments passed to rfsrc to control forest construction.

Details

Given an unsupervised dataset, random forests is used to compute a distance matrix measuring dissimilarity between all pairs of observations. By default, hierarchical clustering is applied to this

distance matrix, although users may apply any other clustering algorithm. See the examples below for alternative workflows.

The default method, `method = "sid"`, implements SID clustering (sidClustering). The algorithm begins by enhancing the original feature space using Staggered Interaction Data (SID). This transformation creates:

- SID main features: shifted and staggered versions of the original features that are made strictly positive and mutually non-overlapping in range;
- SID interaction features: pairwise multiplicative interactions formed between all SID main features.

A multivariate random forest is trained to predict SID main features using the SID interaction features as predictors. The rationale is that if a feature is informative for distinguishing clusters, it will exhibit systematic variation across the data space. Because each interaction feature is uniquely defined by the features it is formed from, node splits on interaction terms are able to capture and separate such variation, thus effectively identifying the clusters. See Mantero and Ishwaran (2021) for further details.

Since SID includes all pairwise interactions, the dimensionality of the feature space grows quadratically with the number of original variables (or worse when factor variables are present). As such, the reduction step using holdout variable importance (VIMP) is strongly recommended (enabled by default). This step can be disabled using `reduce = FALSE`, but only when the original feature space is of manageable size.

A second approach, proposed by Breiman (2003) and refined by Shi and Horvath (2006), transforms the unsupervised task into a two-class supervised classification problem. The first class consists of the original data, while the second class is generated artificially. The goal is to separate real data from synthetic data. A proximity matrix is constructed from this supervised model, and the proximity values for the original class are extracted and converted into a distance matrix (distance = 1 - proximity) for clustering.

Artificial data can be generated using two modes:

- `mode 1` (default): draws random values from the empirical distribution of each feature;
- `mode 2`: draws uniformly between the observed minimum and maximum of each feature.

This method is invoked by setting `method = "sh"`, `"sh1"`, or `"sh2"`. Mantero and Ishwaran (2021) found that while this approach works in certain settings, it can fail when clusters exist in lower-dimensional subspaces (e.g., when defined by interactions or involving both factors and continuous variables). Among the two modes, mode 1 is generally more robust.

The third method, `method = "unsupv"`, trains a multivariate forest using the data both as predictors and as responses. The multivariate splitting rule is applied at each node. This method is fast and simple but may be less accurate compared to SID clustering.

The package includes a helper function `sid.perf.metric` for evaluating clustering performance using a normalized score; smaller values indicate better performance. See Mantero and Ishwaran (2021) for theoretical background and empirical benchmarking.

Value

A list with the following components:

clustering	A vector or matrix assigning each observation to a cluster. If multiple values of k were specified, this is a matrix with one column per clustering solution.
rf	The trained random forest object used in the clustering procedure. This is typically a multivariate forest (for method = "sid" or "unsupv") or a classification forest (for Breiman-style methods).
dist	The distance matrix computed from the forest. Used for clustering. For method = "sid", this is based on the forest dissimilarity; for Breiman/SH methods, this is one minus the proximity matrix.
sid	The SID-transformed data used in the clustering (applies only to method = "sid"). Provided as a list with separate components for the staggered features and their interactions, corresponding to outcomes and predictors in 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))
```

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 of class (rfsrc, grow).
B	Number of subsamples (or bootstrap iterations, if bootstrap = TRUE).
block.size	Number of trees in each block used when calculating VIMP. If VIMP is already included in the original grow object, that setting is used instead.
importance	Type of variable importance (VIMP) to compute. Choices are "anti", "permute", or "random". If not specified, the default importance setting from the original grow call is used (if available).
subratio	Subsample size as a proportion of the original sample size. The default is approximately the inverse square root of the sample size.
stratify	Logical. If TRUE, uses stratified subsampling to preserve class balance. See Details for more information.
performance	Logical. If TRUE, calculates generalization error along with standard error and confidence intervals.
performance.only	Logical. If TRUE, only generalization error and its uncertainty are returned; VIMP is not computed.
joint	Logical. If TRUE, joint VIMP is computed for all variables. To calculate joint VIMP for a subset of variables, use xvar.names.

xvar.names	Character vector specifying variables to be used for joint VIMP. If omitted, all variables are included.
bootstrap	Logical. If TRUE, uses the double bootstrap instead of subsampling. This is typically slower but may provide more accurate uncertainty estimates.
verbose	Logical. If TRUE, prints progress updates during computation.

Details

This function applies subsampling (or optional double bootstrapping) to a previously trained forest to estimate standard errors and construct confidence intervals for variable importance (VIMP), as described in Ishwaran and Lu (2019). It also supports inference for the out-of-bag (OOB) prediction error via the performance = TRUE option. Joint VIMP for selected or all variables can be obtained using joint and xvar.names.

If the original forest does not include VIMP, it will be computed prior to subsampling. For repeated calls to subsample, it is recommended that VIMP be requested in the original rfsrc call. This not only avoids redundant computation, but also ensures consistency of the importance type (e.g., anti, permute, or random) and related parameters, which may otherwise be unclear. Note that permutation importance is *not* the default for most families.

Subsampled forests are constructed using the same tuning parameters as the original forest. While most settings are automatically recovered, certain advanced configurations (e.g., custom sampling schemes) may not be fully supported.

Both subsampled variance estimates (Politis and Romano, 1994) and delete-\(d\) jackknife variance estimates (Shao and Wu, 1989) are returned. The jackknife estimator tends to produce larger standard errors, offering a conservative bias correction, particularly for signal variables.

By default, stratified subsampling is used for classification, survival, and competing risk families:

- For classification, strata correspond to class labels.
- For survival and competing risks, strata include event type and censoring.

Stratification helps ensure representation of key outcome types and is especially important for small sample sizes. Overriding this behavior is discouraged. Note that stratification is *not* available for multivariate families, and caution should be exercised when subsampling in that context.

The function extract.subsample can be used to retrieve detailed information from the subsample object. By default, returned VIMP values are standardized: for regression families, VIMP is divided by the variance of the response; for other families, no transformation is applied. To obtain raw (unstandardized) values, set standardize = FALSE. For expert users, the option raw = TRUE returns detailed internal output, including VIMP from each individual subsampled forest (constructed on a smaller sample size), which is used internally by plot.subsample.rfsrc to generate confidence intervals.

Printed and plotted outputs also standardize VIMP by default. This behavior can be disabled via standardize. The alpha option controls the confidence level and is preset in wrapper functions but can be adjusted by the user.

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)
```

```
}
```

`tune.rfsrc` *Tune Random Forest for optimal* `mtry` *and* `nodesize`

Description

Finds the optimal `mtry` and `nodesize` for a random forest using out-of-bag (OOB) error. Two search strategies are supported: a grid-based search and a golden-section search with noise control. Works for all response families supported by `rfsrc.fast`.

Usage

```
## S3 method for class 'rfsrc'
tune(formula, data,
  mtry.start = ncol(data) / 2,
  nodesize.try = c(1:9, seq(10, 100, by = 5)), ntree.try = 100,
  sampsize = function(x) { min(x * .632, max(150, x^(3/4))) },
  nsplit = 1, step.factor = 1.25, improve = 1e-3, strikeout = 3, max.iter = 25,
  method = c("grid", "golden"),
  final.window = 5, reps.initial = 2, reps.final = 3,
  trace = FALSE, do.best = TRUE, seed = NULL, ...)

## S3 method for class 'rfsrc'
tune.nodesize(formula, data,
  nodesize.try = c(1:9, seq(10, 150, by = 5)), ntree.try = 100,
  sampsize = function(x) { min(x * .632, max(150, x^(4/5))) },
  nsplit = 1, method = c("grid", "golden"),
  final.window = 5, reps.initial = 2, reps.final = 3, max.iter = 50,
  trace = TRUE, seed = NULL, ...)
```

Arguments

`formula`	A model formula.
`data`	A data frame with response and predictors.
`mtry.start`	Initial `mtry` for `tune`.
`nodesize.try`	Candidate `nodesize` values. Only values $\leq$ `floor(sampsize(n)/2)` are used.
`ntree.try`	Number of trees grown at each tuning evaluation.
`sampsize`	Function or numeric giving the per-tree subsample size. During tuning a single numeric size `ssize` is computed and passed to `rfsrc.fast`. If a vector is supplied (e.g., class specific), its total is used for `ssize`.
`nsplit`	Number of random split points to consider at each node.

step.factor	Multiplicative step-out factor over mtry for grid search in tune.
improve	Minimum relative improvement required to continue a search step in tune.
strikeout	Maximum number of consecutive non-improving steps allowed in tune.
max.iter	Maximum number of iterations for the step-out search in tune or the coordinate loop when method = "golden".
method	Search strategy: "grid" (default) or "golden".
final.window	For golden search, the terminal bracket width for the one-dimensional line search.
reps.initial	Replicates averaged at interior evaluations during golden iterations.
reps.final	Replicates averaged for each candidate during the final local sweep in golden search.
trace	If TRUE, prints progress.
do.best	If TRUE, tune fits and returns a forest at the optimal pair.
seed	Optional integer for reproducible tuning. The holdout split (when used) and all tuning fits become deterministic for a given seed.
...	Additional arguments passed to rfsrc.fast. Arguments that control tuning itself (perf.type, forest, save.memory, ntree, mtry, nodesize, sampsize, nsplit) are managed internally.

Details

Error estimate. If 2 * ssize < n, a disjoint holdout of size ssize is used for evaluation; otherwise OOB error is used.

Subsample used during tuning. Both functions derive a single integer ssize from sampsize and pass it to rfsrc.fast for all tuning fits. This improves stability and comparability across candidates. When do.best = TRUE in tune, the final forest is fit with the user-supplied sampsize exactly as provided.

Grid search. tune performs a step-out search over mtry for each nodesize in nodesize.try, using step.factor, improve, strikeout, and max.iter. tune.nodesize evaluates the supplied nodesize.try grid directly.

Golden search. Uses a guarded golden-section line search with noise control. For each one-dimensional search (over nodesize or mtry), the routine probes a small left-anchor grid 1:9, iterates golden shrinkage until the bracket width is at most final.window, then runs a short local sweep with reps.final replicates. In tune the searches over nodesize and mtry alternate in a simple coordinate loop, with improve and strikeout as stopping controls.

Value

For tune:

- results: matrix with columns nodesize, mtry, err.
- optimal: named numeric vector c(nodesize = ..., mtry = ...).
- rf: fitted forest at the optimum if do.best = TRUE.

For tune.nodesize:

- nsize.opt: optimal nodesize.
- err: data frame with columns nodesize and err.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

See Also

rfsrc.fast

Examples

```
## ------------------------------------------------------------
## White wine classification example
## ------------------------------------------------------------
data(wine, package = "randomForestSRC")
wine$quality <- factor(wine$quality)

## Fixed seed makes tuning reproducible
set.seed(1)

## Full tuner over nodesize and mtry (grid)
o1 <- tune(quality ~ ., wine, sampsize = 100, method = "grid")
print(o1$optimal)

## Golden search alternative
o2 <- tune(quality ~ ., wine, sampsize = 100, method = "golden",
           reps.initial = 2, reps.final = 3, seed = 1)
print(o2$optimal)

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

  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.tune(o1)
```

```
  plot.tune(o2)
}

## ------------------------------------------------------------
## nodesize only: grid vs golden
## ------------------------------------------------------------
o3 <- tune.nodesize(quality ~ ., wine, sampsize = 100, method = "grid",
                    trace = TRUE, seed = 1)
o4 <- tune.nodesize(quality ~ ., wine, sampsize = 100, method = "golden",
                    reps.initial = 2, reps.final = 3, trace = TRUE, seed = 1)
plot(o3$err, type = "s", xlab = "nodesize", ylab = "error")

## ------------------------------------------------------------
## Tuning for class imbalance (rfq with geometric mean performance)
## ------------------------------------------------------------
data(breast, package = "randomForestSRC")
breast <- na.omit(breast)
o5 <- tune(status ~ ., data = breast, rfq = TRUE, perf.type = "gmean",
           method = "golden", seed = 1)
print(o5$optimal)

## ------------------------------------------------------------
## Competing risks example (nodesize only)
## ------------------------------------------------------------
data(wihs, package = "randomForestSRC")
plot(tune.nodesize(Surv(time, status) ~ ., wihs, trace = TRUE)$err, type = "s")
```

---

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,
  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). The original rfsrc call must have been made with forest = TRUE.
xvar.names	Character vector of x-variable names to be evaluated. If not specified, all variables are used.
importance	Type of variable importance (VIMP) to compute.
block.size	Integer specifying the number of trees per block used for VIMP calculation. Balances between ensemble-level and tree-level estimates.

joint	Logical indicating whether to compute joint VIMP for the specified variables.
seed	Negative integer used to set the random number generator seed.
do.trace	Number of seconds between printed progress updates.
...	Additional arguments passed to or from other methods.

Details

Using a previously trained forest, this function calculates variable importance (VIMP) for the specified variables in xvar.names. By default, VIMP is computed using the original training data, but the user may supply a new test set via the newdata argument. See rfsrc for further details on how VIMP is computed.

If joint = TRUE, joint VIMP is returned. This is defined as the importance of a group of variables when the entire group is perturbed simultaneously.

Setting csv = TRUE returns case-specific VIMP, which provides VIMP estimates at the individual observation level. This applies to all families except survival. See examples 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