# Package: randomForestSRC (via r-universe)

October 20, 2026

**Version** 3.9.3

**Date** 2026-10-20

**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/

**Config/pak/sysreqs** cmake libglpk-dev make libicu-dev libuv1-dev libxml2-dev libx11-dev

**Repository** https://kogalur.r-universe.dev

**Date/Publication** 2026-10-20 14:00:34 UTC

**RemoteUrl** https://github.com/kogalur/randomforestsrc

**RemoteRef** HEAD

**RemoteSha** 543f0c42376d9827536d7d4c6f6d56fcc7e3849a

# Contents

randomForestSRC-package . . . 3
breast . . . 6
classification.performance . . . 7
fast.saveload . . . 11
follic . . . 19
get.tree.rfsrc . . . 20
hd . . . 23
holdout.vimp.rfsrc . . . 24
housing . . . 28
imbalanced.rfsrc . . . 29
impute.learn.rfsrc . . . 38
impute.rfsrc . . . 53
max.subtree.rfsrc . . . 57
multivariate.values . . . 60
nutrigenomic . . . 63
partial.rfsrc . . . 65
pbc . . . 73
peakVO2 . . . 73
plot.competing.risk.rfsrc . . . 74
plot.quantreg.rfsrc . . . 76
plot.rfsrc . . . 79
plot.subsample.rfsrc . . . 81
plot.survival.rfsrc . . . 84
plot.variable.rfsrc . . . 92
predict.rfsrc . . . 98
print.rfsrc . . . 107
quantreg.rfsrc . . . 108
rfsrc . . . 118
rfsrc.anonymous . . . 137
rfsrc.fast . . . 140
rfsrc.news . . . 142
sidClustering.rfsrc . . . 143
subsample.rfsrc . . . 150
tune.rfsrc . . . 158
vdv . . . 162
veteran . . . 163
vimp.rfsrc . . . 163
wihs . . . 166
wine . . . 167

**Index** **169**

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

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

`classification.performance` *Classification Performance Metrics*

## Description

Evaluate predicted classes and probabilities using confusion matrices, class-specific misclassification rates, ROC area under the curve (AUC), Brier score, log loss, and binary precision-recall summaries. `get.bayes.rule()` assigns classes from predicted probabilities.

## Usage

```
get.confusion(y, class.or.prob)

get.misclass.error(y, yhat)

get.auc(y, prob)

get.brier.error(y, prob, normalized = TRUE, vector = FALSE)

get.logloss(y, prob, robust = TRUE)

get.pr.auc(truth, yhat)

get.pr.curve(truth, yhat)

get.bayes.rule(prob, class.relfrq = NULL)
```

## Arguments

| | |
|---|---|
| `y` | Observed class labels, usually a factor with levels in probability-column order. `get.auc()` also accepts nonfactor labels, with columns in `sort(unique(y))` order. `get.misclass.error()` also accepts vectors of class labels. |
| `prob` | Numeric matrix with one row per observation and one column per class, in response-level order. Name columns with the class labels. `get.auc()` uses column positions; `get.logloss()` and `get.bayes.rule()` require column names. `get.confusion()` and `get.brier.error()` supply missing names from `levels(y)`. |
| `class.or.prob` | A factor of predicted labels with the same levels as `y`, or a probability matrix as described for `prob`. |
| `yhat` | For `get.misclass.error()`, predicted class labels aligned with `y`. For precision-recall summaries, a numeric score vector or a matrix or data frame with one or two columns. A vector or single column gives positive-class scores; larger values favor that class. Scores need not lie in $[0, 1]$. See Details for two-column inputs. |

| | |
|---|---|
| truth | Binary responses coded 0/1, or a two-level factor. For 0/1 labels, 1 is positive. For other factors, the less frequent class is positive, with ties resolved by the first factor level. The positive class is selected before excluding missing scores. |
| normalized | If TRUE, scale the Brier score so that equal class probabilities give score one. If FALSE, average the squared probability errors over classes. See Details. |
| vector | If TRUE, return scaled Brier contributions for each class instead of their sum. |
| robust | For get.logloss(), exclude infinite losses before averaging. Probabilities are unchanged. |
| class.relfrq | Optional two-class relative frequencies, in probability-column order. NULL selects the largest-probability rule; supplying frequencies selects the minority-prevalence rule described below. |

## Details

**Supplying predictions:** Supply responses and predictions in matching row order. For OOB evaluation of a grow object o, use o$yvar with o$predicted.oob for probabilities or o$class.oob for predicted classes. For new data, use the prediction object's yvar with predicted or class.

To compare metrics on the same observations, retain observed labels and finite probabilities. get.logloss() and get.misclass.error() require nonmissing observed labels.

For a combined binary summary with sensitivity, specificity, F1, G-mean, and random-reference comparisons, use get.imbalanced.performance.

**Class assignments and misclassification:** With class.relfrq = NULL, get.bayes.rule() selects the class with the largest probability, breaking ties at random. Rows with all probabilities missing receive NA. With two class frequencies supplied, it assigns the minority class when its probability is at least its relative frequency, and the majority class otherwise. If frequencies tie, the class in the first probability column is treated as the minority. Supply finite probabilities for this rule.

get.confusion() tabulates observed classes in rows and predicted classes in columns. The final class.error column gives each observed class's misclassification rate, rounded to four decimal places. Missing response/prediction pairs are excluded. Probability matrices use the largest-probability rule. For RFQ or another thresholded rule, supply the predicted factor instead, such as o$class.oob.

get.misclass.error() returns one error rate per observed class, in sort(unique(y)) order. A missing predicted label makes the corresponding class rate unavailable.

**ROC area under the curve:** get.auc() computes the multiclass AUC of Hand and Till (2001). For each class pair, it averages two rank-based AUCs, one using each class's probability as the score, then averages across available pairs. For complementary binary probabilities, the two directions agree. Larger values indicate better discrimination.

Missing labels and nonfinite scores are excluded from each rank calculation. Each direction requires at least two finite scores per class. Unavailable pairs are omitted; the result is NA when none remain.

**Brier score:** The Brier score measures squared probability error (Brier, 1950). For $J$ classes, let

$$b_j = \text{mean}_i \left[ (I(y_i = j) - p_{ij})^2 \right].$$

`get.brier.error()` returns

$$\frac{J}{J-1}\sum_{j=1}^{J} b_j$$

for `normalized = TRUE`, or

$$\frac{1}{J}\sum_{j=1}^{J} b_j$$

for `normalized = FALSE`. Smaller values are better. Equal probabilities $p_{ij} = 1/J$ give normalized score one. For complementary binary probabilities, the unnormalized score is the mean squared error of either class probability; the normalized score is four times that value.

With `vector = TRUE`, each element is $b_j$ multiplied by the selected scaling constant. Missing losses are omitted within each class. The scalar sums available contributions, returning `NA` when none remain.

**Log loss:** With every response level represented, `get.logloss()` averages the negative log probability of the observed class, $-\log(p_{i,y_i})$, using the natural logarithm (see Gneiting and Raftery, 2007). Smaller values are better. Missing losses are omitted. With `robust = TRUE`, infinite losses from zero probabilities are also omitted; `robust = FALSE` retains them. Probabilities are not clipped. An unused factor level contributes one zero term to the average.

**Precision-recall summaries:** For two-column scores, named columns match the original response levels; unnamed columns follow factor-level order, or `0`, `1` order for nonfactor responses. The positive-class column is selected. Rows with a missing response or nonfinite selected score are excluded; both classes must remain.

`get.pr.auc()` returns the precision-recall area, calculated using analytic precision-recall interpolation, and a random reference area equal to the positive-class proportion among scored observations. `get.pr.curve()` returns recall, precision, and thresholds in decreasing recall order. Larger precision-recall areas are better. See Davis and Goadrich (2006) for background on precision-recall curves.

## Value

`get.confusion` A numeric matrix of class counts with an additional `class.error` column.

`get.misclass.error` An unnamed numeric vector of class-specific error rates, in `sort(unique(y))` order.

`get.auc` A scalar AUC, or `NA` when unavailable.

`get.brier.error` A scalar Brier score by default, or an unnamed length-$J$ vector of scaled class contributions in probability-column order when `vector = TRUE`.

`get.logloss` A scalar mean log loss. It can be infinite with `robust = FALSE`, or `NaN` when no losses remain for averaging.

`get.pr.auc` An unnamed numeric vector of length two: the model area followed by the random reference area. Both entries are `NA` when the calculation is unavailable.

`get.pr.curve` A numeric matrix with columns `recall`, `precision`, and `threshold`, or `NULL` when the calculation is unavailable.

`get.bayes.rule` A factor of predicted classes with levels given by the probability-column names.

## References

Brier, G.W. (1950). Verification of forecasts expressed in terms of probability. *Monthly Weather Review*, 78, 1-3. doi:10.1175/15200493(1950)078<0001:VOFEIT>2.0.CO;2

Davis, J. and Goadrich, M. (2006). The relationship between precision-recall and ROC curves. *Proceedings of the 23rd International Conference on Machine Learning*, 233-240. doi:10.1145/1143844.1143874

Gneiting, T. and Raftery, A.E. (2007). Strictly proper scoring rules, prediction, and estimation. *Journal of the American Statistical Association*, 102, 359-378. doi:10.1198/016214506000001437

Hand, D.J. and Till, R.J. (2001). A simple generalisation of the area under the ROC curve for multiple class classification problems. *Machine Learning*, 45, 171-186. doi:10.1023/A:1010920819831

## See Also

rfsrc, predict.rfsrc, imbalanced.rfsrc, get.imbalanced.performance, get.brier.survival, get.auct.survival

## Examples

```
## ------------------------------------------------------------
## A basic calculation from observed labels and probabilities
## ------------------------------------------------------------
y <- factor(c("no", "no", "no", "no", "yes", "yes"),
            levels = c("no", "yes"))
p <- c(.10, .20, .65, .35, .40, .85)
prob <- cbind(no = 1 - p, yes = p)
print(get.confusion(y, prob))
print(get.auc(y, prob))
print(get.brier.error(y, prob))


## ------------------------------------------------------------
## Class-specific errors and probability losses
## ------------------------------------------------------------
yhat <- get.bayes.rule(prob)
print(setNames(get.misclass.error(y, yhat), levels(y)))
print(get.brier.error(y, prob, normalized = FALSE))
print(setNames(get.brier.error(y, prob, vector = TRUE), colnames(prob)))
print(get.logloss(y, prob))

## Use supplied class frequencies for the binary RFQ decision rule.
class.frq <- as.numeric(prop.table(table(y)))
yhat.rfq <- get.bayes.rule(prob, class.relfrq = class.frq)
print(get.confusion(y, yhat.rfq))

## ------------------------------------------------------------
## Precision-recall from a score vector
## ------------------------------------------------------------
truth <- as.integer(y == "yes")
pr.auc <- get.pr.auc(truth, p)
print(setNames(pr.auc, c("model", "random")))
```

```
pr <- get.pr.curve(truth, p)
plot(pr[, "recall"], pr[, "precision"], type = "l",
     xlim = c(0, 1), ylim = c(0, 1),
     xlab = "Recall", ylab = "Precision")
abline(h = pr.auc[2], lty = 2)

## A single score column is also accepted.
print(get.pr.auc(truth, matrix(p, ncol = 1)))

## ------------------------------------------------------------
## Multiclass forest: OOB and test-data performance
## ------------------------------------------------------------
set.seed(17)
train <- c(1:35, 51:85, 101:135)
o <- rfsrc(Species ~ ., data = iris[train, ], ntree = 100)

## Select one common set of observed responses and finite OOB scores.
p.oob <- o$predicted.oob
keep <- !is.na(o$yvar) & rowSums(!is.finite(p.oob)) == 0
print(get.confusion(o$yvar[keep], o$class.oob[keep]))
print(get.auc(o$yvar[keep], p.oob[keep, , drop = FALSE]))

## Use the current responses and probabilities for test-data scoring.
p.test <- predict(o, newdata = iris[-train, ])
print(c(
  auc = get.auc(p.test$yvar, p.test$predicted),
  brier = get.brier.error(p.test$yvar, p.test$predicted),
  logloss = get.logloss(p.test$yvar, p.test$predicted)
))
```

---

`fast.saveload` *Fast Saving and Loading of Random Forests*

---

## Description

Save and load a minimal forest object for prediction. These functions are designed to use substantially less disk space and load large forests faster than saving the full fitted object with `saveRDS` and loading it with `readRDS`. Functions are available for a single forest or a list of forests.

## Usage

```
fast.save(o, path = NULL, testing = TRUE, units = "Mb", compact = FALSE,
          compress = 50)

fast.load(directory, path = NULL, testing = FALSE, units = "Mb")

fast.save.list(o, path = NULL, testing = FALSE, units = "Mb", compact = FALSE,
               compress = 50)

fast.load.list(directory, path = NULL, testing = FALSE, units = "Mb")
```

## Arguments

| | |
|---|---|
| o | For fast.save(), a fitted object with class beginning c("rfsrc", "grow") and a forest component. Supply the fitted object, not o$forest. For fast.save.list(), a nonempty list of these objects. |
| path | For saving, the destination directory, including its name. The default is file.path(getwd(), "forest"). An existing destination and all its contents are deleted. For loading, the parent directory containing directory; NULL uses the current working directory. |
| directory | Name of the saved directory relative to path. For fast.load.list(), this directory contains the individual forest subdirectories. |
| testing | Print component memory sizes using lsos(). Defaults to TRUE for fast.save() and FALSE for the other functions. |
| units | Units for the memory-size display, passed to format(object.size(...), units = units). The default is "Mb". |
| compact | If TRUE, omit terminal-node information from the saved forest and recompute it during prediction. Requires a non-anonymous forest containing the training predictors and responses. If FALSE (default), save the forest with its available terminal-node information. See *Compact saving* below. |
| compress | A single finite number between 0 and 100 specifying the FST compression level. The default is 50 for fast saving. Use 100 for maximum compression when smaller files are more important than saving speed. This setting is independent of compact. |

## Details

**Saving and loading a forest:** fast.save() addresses the large files and slow loading that can result from saving a full fitted forest with saveRDS(). It saves only o$forest, the minimal object used for prediction, including its tree information and any training data it contains. Predictions and performance summaries outside this component are excluded. fast.load() returns the forest for use with predict.rfsrc.

Large tabular components are saved efficiently with fst::write_fst(), and the remaining forest information with saveRDS(..., compress = FALSE). Saving requires the **fst** and **data.table** packages; loading requires **fst**. All files have a '.rda' extension despite the different formats. Keep the directory's files together and load the forest with fast.load().

**Compression:** The compress setting is passed to every fst::write_fst() call, including each forest saved by fast.save.list(). Higher compression can reduce file sizes at the cost of slower saving; the reduction depends on the saved components. The remaining forest information in 'forest.rda' continues to use saveRDS(..., compress = FALSE).

Compression changes the encoding of the saved components, not the fitted forest or the choice of components. It can be combined with compact = TRUE. Use the same fast.load() or fast.load.list() call for any compression level; no compression argument is needed when loading. Invalid compression values are rejected before the destination is deleted.

**Compact saving:** With compact = TRUE, the saved forest includes the tree structure, factor-split information, training data, seeds, and other forest information, but omits stored terminal-node information. Prediction recomputes this information from the saved forest and training data without growing new trees. The fitted object in memory is unchanged.

The saved copy has terminal.qualts = FALSE and terminal.quants = FALSE, with nativeArrayTNDS removed. Use the same fast.load() or fast.load.list() call for either saving mode. Compact saving can further reduce disk space, but prediction may take longer and peak memory use may not decrease.

Use compact = FALSE for anonymous forests. Compact saving includes the training data. Eligibility checks run before the destination is deleted; for lists, every forest is checked first.

**Shared imputer banks:** Standalone fast.save() always writes a self-contained forest. Imputers fitted with share.xvar = TRUE use the same component serializers but store their repeated training predictors once per bank. Load those complete banks with load.impute.learn. Direct fast.load() calls on shared learner subdirectories stop because their predictors depend on the bank's common table and reconstruction mappings.

**Directories: Both save functions delete an existing destination directory and all its contents.** Use a dedicated directory for each forest or list. The destination and any missing parent directories are then created.

When saving, path is the complete destination. When loading, directory is its name and path is its parent. For a destination save.path, use fast.load(basename(save.path), path = dirname(save.path)). For the default destination, use fast.load("forest").

**Lists of forests:** fast.save.list() saves forests in subdirectories 'forest1', 'forest2', and so on, inside path. List names are not saved. fast.load.list() loads every entry listed by list.files() in that directory, so it should contain only saved forest subdirectories.

The loaded list is unnamed and follows directory-listing order, which can differ from the original order: 'forest10' can precede 'forest2'. To load one forest, call fast.load() with its subdirectory name.

## Value

fast.save Saves the forest to disk and returns the memory-usage matrix produced by gc(FALSE).

fast.load The saved forest component with its original class, usually c("rfsrc", "forest", family).

fast.save.list An unnamed list of the memory-usage matrices returned by the individual fast.save() calls.

fast.load.list An unnamed list of forest objects in directory-listing order.

## Note

The sampling-size function is saved as sampfrac = o$forest$sampsize(1) and loaded as function(x) x * sampfrac. Use saveRDS() to save a custom function that is not proportional to sample size.

Compact saving checks that round(sampsize(n)) equals round(n * sampsize(1)) at the training sample size n. A mismatch stops saving before the destination is deleted. This check applies only at the training sample size.

## See Also

rfsrc, predict.rfsrc, saveRDS, readRDS

## Examples

```
## ------------------------------------------------------------
## Regression: save, load, and compare predictions
## ------------------------------------------------------------
if (requireNamespace("fst", quietly = TRUE) &&
    requireNamespace("data.table", quietly = TRUE)) {
  o <- rfsrc(mpg ~ ., data = mtcars)
  print(o)
  save.path <- tempfile("rfsrc-forest-")
  fast.save(o, path = save.path, testing = FALSE)
  oo <- fast.load(basename(save.path), path = dirname(save.path))
  p <- predict(o)
  pp <- predict(oo)
  print(summary(p$predicted - pp$predicted))
  print(summary(p$predicted.oob - pp$predicted.oob))
  unlink(save.path, recursive = TRUE)
}


## ------------------------------------------------------------
## Compact saving: the same forest, fewer saved components
## ------------------------------------------------------------
if (requireNamespace("fst", quietly = TRUE) &&
    requireNamespace("data.table", quietly = TRUE)) {
  o <- rfsrc(mpg ~ ., data = mtcars, ntree = 100)
  set.seed(19)
  reference <- predict(o, seed = -19)
  save.path <- tempfile("rfsrc-compact-")
  fast.save(o, path = save.path, compact = TRUE, testing = FALSE)
  oo <- fast.load(basename(save.path), path = dirname(save.path))
  print(oo[c("terminal.qualts", "terminal.quants")])
  print(list.files(save.path, pattern = "^nativeArrayTDNS_"))
  set.seed(19)
  restored <- predict(oo, seed = -19)
  print(all.equal(reference$predicted, restored$predicted))
  print(all.equal(reference$predicted.oob, restored$predicted.oob))
  unlink(save.path, recursive = TRUE)
}

## ------------------------------------------------------------
## Regression: a list of forests with different node sizes
## ------------------------------------------------------------
if (requireNamespace("fst", quietly = TRUE) &&
    requireNamespace("data.table", quietly = TRUE)) {
  o1 <- rfsrc(mpg ~ ., data = mtcars, nodesize = 1)
  o2 <- rfsrc(mpg ~ ., data = mtcars, nodesize = 10)
  print(o1)
  print(o2)
  models <- list(o1, o2)
  save.path <- tempfile("rfsrc-forest-list-")
  invisible(fast.save.list(models, path = save.path))
  oo <- fast.load.list(basename(save.path), path = dirname(save.path))
```

```
  print(predict(oo[[1]]))
  print(predict(oo[[2]]))
  unlink(save.path, recursive = TRUE)
}

## ------------------------------------------------------------
## RFQ for imbalanced classification
## ------------------------------------------------------------
## Use matching prediction seeds when comparing class labels.
if (requireNamespace("fst", quietly = TRUE) &&
    requireNamespace("data.table", quietly = TRUE)) {
  data(breast, package = "randomForestSRC")
  dta <- na.omit(breast)
  o <- imbalanced(status ~ ., data = dta, ntree = 100)
  print(o)
  save.path <- tempfile("rfsrc-forest-")
  fast.save(o, path = save.path, testing = FALSE)
  oo <- fast.load(basename(save.path), path = dirname(save.path))
  set.seed(19)
  p <- predict(o, seed = -19)
  set.seed(19)
  pp <- predict(oo, seed = -19)
  print(summary(p$predicted - pp$predicted))
  print(summary(p$predicted.oob - pp$predicted.oob))
  print(all.equal(as.character(p$class), as.character(pp$class)))
  print(all.equal(as.character(p$class.oob),
                  as.character(pp$class.oob)))
  unlink(save.path, recursive = TRUE)
}

## ------------------------------------------------------------
## Binary classification with rfq = TRUE
## ------------------------------------------------------------
if (requireNamespace("fst", quietly = TRUE) &&
    requireNamespace("data.table", quietly = TRUE)) {
  data(breast, package = "randomForestSRC")
  dta <- na.omit(breast)
  o <- rfsrc(status ~ ., data = dta, rfq = TRUE, ntree = 100,
             perf.type = "gmean", splitrule = "auc")
  print(o)
  save.path <- tempfile("rfsrc-forest-")
  fast.save(o, path = save.path, testing = FALSE)
  oo <- fast.load(basename(save.path), path = dirname(save.path))
  set.seed(19)
  p <- predict(o, seed = -19)
  set.seed(19)
  pp <- predict(oo, seed = -19)
  print(summary(p$predicted - pp$predicted))
  print(summary(p$predicted.oob - pp$predicted.oob))
  print(all.equal(as.character(p$class), as.character(pp$class)))
  print(all.equal(as.character(p$class.oob),
                  as.character(pp$class.oob)))
  unlink(save.path, recursive = TRUE)
```

```
}

## ------------------------------------------------------------
## Anonymous RFQ: supply the same prediction data to both forests
## ------------------------------------------------------------
if (requireNamespace("fst", quietly = TRUE) &&
    requireNamespace("data.table", quietly = TRUE)) {
  data(breast, package = "randomForestSRC")
  dta <- na.omit(breast)
  o <- rfsrc.anonymous(status ~ ., data = dta, rfq = TRUE,
                       ntree = 100, perf.type = "gmean", splitrule = "auc")
  print(o)
  save.path <- tempfile("rfsrc-forest-")
  fast.save(o, path = save.path, testing = FALSE)
  oo <- fast.load(basename(save.path), path = dirname(save.path))
  set.seed(19)
  p <- predict(o, newdata = dta, seed = -19)
  set.seed(19)
  pp <- predict(oo, newdata = dta, seed = -19)
  print(summary(p$predicted - pp$predicted))
  print(all.equal(as.character(p$class), as.character(pp$class)))
  unlink(save.path, recursive = TRUE)
}

## ------------------------------------------------------------
## Survival
## ------------------------------------------------------------
if (requireNamespace("fst", quietly = TRUE) &&
    requireNamespace("data.table", quietly = TRUE)) {
  data(pbc, package = "randomForestSRC")
  o <- rfsrc(Surv(days, status) ~ ., data = pbc, ntree = 100)
  print(o)
  save.path <- tempfile("rfsrc-forest-")
  fast.save(o, path = save.path, testing = FALSE)
  oo <- fast.load(basename(save.path), path = dirname(save.path))
  set.seed(19)
  p <- predict(o, seed = -19)
  set.seed(19)
  pp <- predict(oo, seed = -19)
  print(summary(p$predicted - pp$predicted))
  print(summary(p$predicted.oob - pp$predicted.oob))
  unlink(save.path, recursive = TRUE)
}

## ------------------------------------------------------------
## Survival with save.memory = TRUE
## ------------------------------------------------------------
if (requireNamespace("fst", quietly = TRUE) &&
    requireNamespace("data.table", quietly = TRUE)) {
  data(pbc, package = "randomForestSRC")
  o <- rfsrc(Surv(days, status) ~ ., data = pbc,
             ntree = 100, save.memory = TRUE)
  print(o)
```

```
  save.path <- tempfile("rfsrc-forest-")
  fast.save(o, path = save.path, testing = FALSE)
  oo <- fast.load(basename(save.path), path = dirname(save.path))
  set.seed(19)
  p <- predict(o, seed = -19)
  set.seed(19)
  pp <- predict(oo, seed = -19)
  print(summary(p$predicted - pp$predicted))
  print(summary(p$predicted.oob - pp$predicted.oob))
  unlink(save.path, recursive = TRUE)
}

## ------------------------------------------------------------
## Competing risks
## ------------------------------------------------------------
if (requireNamespace("fst", quietly = TRUE) &&
    requireNamespace("data.table", quietly = TRUE)) {
  data(wihs, package = "randomForestSRC")
  o <- rfsrc(Surv(time, status) ~ ., data = wihs, nsplit = 3, ntree = 100)
  print(o)
  save.path <- tempfile("rfsrc-forest-")
  fast.save(o, path = save.path, testing = FALSE)
  oo <- fast.load(basename(save.path), path = dirname(save.path))
  set.seed(19)
  p <- predict(o, seed = -19)
  set.seed(19)
  pp <- predict(oo, seed = -19)
  print(summary(p$predicted - pp$predicted))
  print(summary(p$predicted.oob - pp$predicted.oob))
  unlink(save.path, recursive = TRUE)
}

## ------------------------------------------------------------
## Multivariate regression and classification
## ------------------------------------------------------------
if (requireNamespace("fst", quietly = TRUE) &&
    requireNamespace("data.table", quietly = TRUE)) {
  data(nutrigenomic, package = "randomForestSRC")
  ydta <- data.frame(diet = nutrigenomic$diet,
                     genotype = nutrigenomic$genotype,
                     nutrigenomic$lipids)
  o <- rfsrc(get.mv.formula(colnames(ydta)),
             data = data.frame(ydta, nutrigenomic$genes),
             ntree = 100, importance = TRUE, nsplit = 10)
  print(o)
  save.path <- tempfile("rfsrc-forest-")
  fast.save(o, path = save.path, testing = FALSE)
  oo <- fast.load(basename(save.path), path = dirname(save.path))
  set.seed(19)
  p <- predict(o, seed = -19)
  set.seed(19)
  pp <- predict(oo, seed = -19)
  print(summary(get.mv.predicted(p, oob = FALSE) -
```

```
              get.mv.predicted(pp, oob = FALSE)))
  print(summary(get.mv.predicted(p) - get.mv.predicted(pp)))
  for (yn in names(p$classOutput)) {
    print(yn)
    print(all.equal(as.character(p$classOutput[[yn]]$class),
                    as.character(pp$classOutput[[yn]]$class)))
    print(all.equal(as.character(p$classOutput[[yn]]$class.oob),
                    as.character(pp$classOutput[[yn]]$class.oob)))
  }
  unlink(save.path, recursive = TRUE)
}


## Not run:

## ------------------------------------------------------------
## Classification: optional alzheimers data from varPro
## ------------------------------------------------------------
if (requireNamespace("fst", quietly = TRUE) &&
    requireNamespace("data.table", quietly = TRUE)) {
  data(alzheimers, package = "varPro")
  o <- rfsrc(Diagnosis ~ ., data = alzheimers)
  print(o)
  save.path <- tempfile("rfsrc-forest-")
  fast.save(o, path = save.path, testing = FALSE)
  oo <- fast.load(basename(save.path), path = dirname(save.path))
  set.seed(19)
  p <- predict(o, seed = -19)
  set.seed(19)
  pp <- predict(oo, seed = -19)
  print(summary(p$predicted - pp$predicted))
  print(summary(p$predicted.oob - pp$predicted.oob))
  print(all.equal(as.character(p$class), as.character(pp$class)))
  print(all.equal(as.character(p$class.oob),
                  as.character(pp$class.oob)))
  unlink(save.path, recursive = TRUE)
}


## ------------------------------------------------------------
## Optional memory-intensive anonymous survival test
## ------------------------------------------------------------
## This test repeats each PBC row 250 times and can require substantial memory.
if (requireNamespace("fst", quietly = TRUE) &&
    requireNamespace("data.table", quietly = TRUE)) {
  data(pbc, package = "randomForestSRC")
  dta <- pbc[rep(seq_len(nrow(pbc)), each = 250), ]
  o <- rfsrc.anonymous(Surv(days, status) ~ ., data = dta)
  print(o)
  save.path <- tempfile("rfsrc-forest-")
  fast.save(o, path = save.path, testing = FALSE)
  oo <- fast.load(basename(save.path), path = dirname(save.path))
  set.seed(19)
```

```
  p <- predict(o, newdata = dta, seed = -19)
  set.seed(19)
  pp <- predict(oo, newdata = dta, seed = -19)
  print(summary(p$predicted - pp$predicted))
  unlink(save.path, recursive = TRUE)
}

## End(Not run)
```

---

follic *Follicular Cell Lymphoma*

---

## Description

Competing risk data set involving follicular cell lymphoma.

## Format

A data frame containing:

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

## Source

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

## References

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

## Examples

```
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` *Random Forests for Imbalanced Two-Class Classification*

---

## Description

Fits a random forest for a two-class outcome using the random forest quantile classifier (RFQ), balanced random forest (BRF), or standard classification. RFQ changes the probability threshold used to assign class labels; BRF changes the sampling used to grow the trees. The splitting rule, performance measure, and variable importance can be specified separately. RFQ and BRF use normalized Brier performance by default for prediction error and variable importance.

Companion helpers summarize performance, select a probability threshold, or extract the class-frequency threshold from response labels.

## 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, ...)

get.imbalanced.performance(obj, prob = NULL, threshold = NULL,
                           confusion = FALSE, robust = FALSE)
get.imbalanced.optimize(obj, prob = NULL, newdata = NULL,
                        measure = c("gmean", "F1", "F1mod", "F1modgmean"),
                        ngrid = 1000, plot.it = TRUE)
get.rfq.threshold(y)

## S3 method for class 'imbalanced.performance'
print(x, digits = 4,
                                      show.confusion = TRUE, ...)
```

## Arguments

| | |
|---|---|
| formula | A model formula with one factor response having two levels and one or more predictors. |
| data | A data frame containing the response and predictors. Data-frame subclasses are converted to a plain data frame. |
| ntree | Number of trees to grow. The default is 3000. |
| method | Classification method. "rfq", the default, uses the RFQ decision rule of O'Brien and Ishwaran (2019). "brf" uses class-weighted sampling with replacement and a reduced tree sample size. "standard" uses random forest classification without enabling the RFQ rule. See Details. |
| splitrule | Splitting rule passed to the forest grow function. The default is "auc". Other choices include "gini" and "entropy"; see rfsrc for supported classification rules. This setting is separate from perf.type. |
| perf.type | Performance measure used for the forest's prediction error (err.rate), variable importance (VIMP), and downstream inference based on these quantities. With NULL, the default is "brier" (normalized Brier score) for RFQ and BRF, and "default" (misclassification error) for standard classification. Set perf.type = "gmean" to use the previous RFQ/BRF default. Choices accepted by this wrapper are "none", "default", "standard", "misclass", "brier", "gmean", and the legacy spelling "g.mean". The latter is converted to "gmean". Use "none" to omit forest performance calculations. |
| block.size | Block size passed to the selected forest grow function for performance and VIMP calculations; see rfsrc. This wrapper passes NULL explicitly when the argument is not set. Specify the same block size when comparing VIMP across methods. |
| fast | Logical. Use rfsrc.fast for the RFQ or standard method when TRUE. The BRF branch always calls rfsrc and does not use this argument. |
| ratio | Optional, experimental majority-class undersampling for the RFQ and standard methods. Supply a number between zero and one. The majority class is sampled without replacement, while the minority class is bootstrapped with replacement. Small requested fractions may be increased by the sampling helper. This is a majority-class sampling fraction, rather than a requested final minority prevalence. Ignored for BRF. See Details for OOB support. |
| obj | For the performance and optimization helpers, a grow or prediction object with a single two-class outcome and observed responses, or a two-level response factor when prob is supplied. |
| prob | Optional probabilities aligned with the responses in obj. A numeric vector or one-column matrix or data frame contains minority-class probabilities. A two-column matrix or data frame contains both class probabilities; named columns are matched to factor levels, and unnamed columns follow factor-level order. NULL uses the object's predictions as described in Details. |
| threshold | Cutoff for minority-class probabilities in get.imbalanced.performance. NULL, the default, uses the minority proportion among the observed evaluation responses. "auto" uses that prevalence for an RFQ object and 0.5 otherwise. |

| | |
|---|---|
| | "bayes" always uses 0.5. A single finite numeric value specifies the cutoff directly; values outside $[0, 1]$ are clipped to that interval. With direct response and probability inputs, "auto" uses 0.5 because no fitted classifier is supplied. |
| confusion | Return a list including the confusion matrix instead of the default named vector of performance measures? |
| robust | Add one to the diagonal counts when calculating the four classification rates and their derived summaries? See Details. |
| newdata | Optional threshold-selection data for get.imbalanced.optimize, supplied with a grow object and prob = NULL. Must contain the predictors and observed response. NULL uses predictions already in obj. Ignored when prob is supplied. |
| measure | Performance measure to maximize during threshold selection: "gmean" (the default), "F1", "F1mod", or "F1modgmean". See Details for definitions. |
| ngrid | Number of equally spaced thresholds from zero to one, including both endpoints. Must be an integer of at least two. |
| plot.it | Draw the four threshold-search diagnostic panels? |
| y | Class-label vector for get.rfq.threshold, typically a two-level factor. Missing responses are omitted when counting classes. |
| ... | Additional arguments passed to rfsrc or, when selected, rfsrc.fast. Examples include mtry, nodesize, nsplit, importance, and na.action. BRF sets its own case.wt, sampsize, and samptype. When ratio is supplied, the wrapper constructs samp and sets bootstrap = "by.user". Additional arguments to the print method are currently unused. |
| x | For the print method, the result of get.imbalanced.performance. |
| digits | Number of decimal places used in the performance display. |
| show.confusion | Print model and expected random-reference confusion tables when a confusion matrix is present in x? Obtain that matrix with confusion = TRUE in get.imbalanced.performance. This option does not recompute it. |

## Details

**Choice of classifier:** Let $\pi$ be the training proportion of the minority class and $\widehat{p}(x)$ its predicted probability. RFQ assigns an observation to the minority class when $\widehat{p}(x) \geq \pi$. The class-frequency threshold can identify minority observations whose predicted probability is below one half. The standard method instead uses class assignment based on the largest predicted probability. RFQ here is a classification rule, distinct from the continuous-response quantile estimation performed by quantreg.

BRF calls the grow function with replacement sampling, a tree sample size of twice the smaller observed class count, and case weights that give the two classes equal total sampling weight. The wrapper does not construct separate fixed-size samples from each class. Both fast and ratio are ignored in this branch.

**Performance and variable importance:** The default performance measure for RFQ and BRF is the normalized Brier score, selected by perf.type = "brier". For binary outcomes $Y_i \in \{0, 1\}$, with $Y_i = 1$ denoting the minority class, and predicted minority-class probabilities $\widehat{p}_i$, the overall score is

$$\mathrm{BS}^* = \frac{4}{n} \sum_{i=1}^{n} (Y_i - \widehat{p}_i)^2.$$

The mean is taken over the observations being scored. Smaller values are better. This is four times the binary Brier score; the normalization does not reweight the classes or change VIMP rankings.

Brier performance responds to changes in predicted probabilities even when the assigned classes stay the same. This can improve the separation of predictive variables from noise in difficult imbalanced problems. Request permutation VIMP with `importance = "permute"`; it measures the increase in the selected prediction error after a variable is permuted.

G-mean remains available with `perf.type = "gmean"`. It is the geometric mean of minority sensitivity and majority specificity, $\sqrt{\text{sensitivity specificity}}$. Larger G-mean values are better; the forest uses one minus G-mean as overall prediction error. Unlike Brier performance, it depends on the class assignments. Changing `perf.type` changes the performance measure, while `method` selects the classifier and `splitrule` selects the splitting criterion.

The selected metric is used for `err.rate` and VIMP and is inherited by prediction and VIMP calculations unless overridden. `subsample` also uses the fitted forest's metric for inference on VIMP and prediction error. The helper `get.imbalanced.performance` continues to report all its performance measures, regardless of `perf.type`. Selecting a classification threshold with `get.imbalanced.optimize` remains a separate operation; changing that threshold does not change a Brier score calculated from the same probabilities.

For comparisons, use a common `perf.type`, splitting rule, and `block.size`, unless the comparison is intended to study those settings. To compare performance metrics, `vimp` can score the same fitted forest under each metric. Compare recovery or ranking of predictive variables, rather than raw VIMP magnitudes on different scales. Increase `ntree` as needed to assess the stability of OOB performance and VIMP. The simulation example compares Brier and G-mean VIMP using known signal and noise variables.

**Sampling and missing values:** With `ratio = NULL`, the RFQ and standard branches pass missing-data handling to the selected grow function. BRF and calls using `ratio` first apply complete-case omission to the supplied data frame. This omission currently includes columns that are not selected by the formula and occurs before the grow function handles `na.action`.

With `ratio` supplied, each tree uses a bootstrap sample of the minority observations and a without-replacement sample of majority observations. At `ratio = 1`, every majority observation is in-bag for every tree, so majority-class OOB performance is unavailable. A single minority observation is also always in-bag in this sampling scheme. Use a separate evaluation sample when OOB support is absent.

**Detailed performance summaries:** `get.imbalanced.performance` calculates performance from observed responses and predicted probabilities.

**Probability inputs.** Supply a grow or prediction object, or a two-level response factor with `prob`. With `prob = NULL`, the helper uses `predicted.oob` when present and not entirely missing; otherwise it uses `predicted`. When only some OOB predictions are missing, those entries remain missing.

A numeric vector or one-column matrix or data frame supplies minority-class probabilities; no column name is required. Two-column inputs supply both class probabilities; named columns are matched to factor levels, and unnamed columns follow factor-level order. Rows must align with the responses.

**Evaluation classes.** All observed evaluation responses determine class counts, the minority class, and its proportion, before excluding unavailable predictions. The first factor level is the minority when counts tie.

**Probability checks.** Finite probabilities below zero are set to zero, and those above one are set to one. Finite two-column rows must sum to one within numerical tolerance. Scoring excludes rows with missing responses or nonfinite probabilities; both columns must be finite for two-column inputs. Unavailable scores are NA.

**Classification threshold.** An observation is assigned to the minority class when its predicted minority-class probability is at least the threshold, and to the majority class otherwise. The threshold choices are:

NULL The evaluation minority proportion for every method (default).

"auto" The evaluation minority proportion when forest$rfq is TRUE; $0.5$ otherwise, including direct response/probability inputs.

"bayes" $0.5$ for every method.

**Numeric** A single finite cutoff. Values below zero are set to zero, and values above one are set to one.

For RFQ test predictions, NULL and "auto" use evaluation prevalence, not training prevalence. The RFQ flag, rather than splitrule or perf.type, determines the "auto" choice. Changing the threshold preserves the evaluation minority definition and the fitted forest.

**Performance measures.** The summary includes sensitivity (sens), specificity (spec), precision (prec), negative predictive value (npv), G-mean, misclass, brier, brier.norm, auc, logloss, and pr.auc. For complementary binary probabilities, brier is the mean squared error of the minority probability; brier.norm is four times this value. Log loss excludes infinite losses without clipping zero probabilities.

**F-measures.** F1 is the harmonic mean of precision and sensitivity; F1mod is the harmonic mean of all four classification rates. F1gmean averages F1 and G-mean; F1modgmean averages F1mod and G-mean. A harmonic mean is zero if any component rate is zero and all are defined; it is NA if any required rate is undefined.

**Robust summaries.** robust = TRUE adds one to each diagonal confusion count when calculating the four rates and their derived summaries. Returned confusion counts, misclassification error, and probability-based scores are unchanged.

**Confusion matrix.** confusion = TRUE includes the confusion matrix: rows are observed classes and columns are predicted classes, with 0 denoting the majority and 1 the minority.

**Random-score reference:** The performance summary has a "rand" attribute containing reference metrics and expected confusion counts. This reference uses a score $U$ uniformly distributed on $[0, 1]$, independent of class, and predicts the minority class when $U$ reaches the chosen threshold. For threshold $t$, its sensitivity is $1 - t$, specificity is $t$, and ROC AUC is $1/2$. The separately returned PR-AUC reference is the proportion of positive observations among the rows scored by the PR helper. Its unnormalized Brier, normalized Brier, and log-loss references are $1/3$, $4/3$, and $1$, respectively. These differ from a predictor that always returns probability $1/2$.

Nonlinear rate summaries are computed from the reference rates (or smoothed expected counts under robust = TRUE); they are not finite-sample expectations of those nonlinear statistics. The performance print method compares the model and reference with Delta and percentage Gain, oriented so that positive values indicate improvement. The fitted-forest printer has its own probability reference; it need not use the same baseline.

**Threshold selection:** get.imbalanced.optimize searches ngrid equally spaced thresholds from zero to one, using G-mean, F1, F1mod, or F1modgmean as selected by measure. It uses the first grid point attaining the largest available value. It returns one row of performance metrics at

the selected threshold and does not change the forest or its stored class predictions. `plot.it = FALSE` suppresses its four diagnostic panels. `ngrid` must be an integer of at least two. When the chosen measure is unavailable at every grid point, the function stops with an explanatory error. A valid optimum of zero is retained. The caller's graphics settings are restored after plotting.

A grow object ordinarily supplies OOB probabilities for this search. Supplying `newdata` searches using that data's observed responses and predictions, making it a threshold-selection sample. For an independent assessment of the chosen threshold, apply it unchanged to a separate test sample. The examples select a threshold from training OOB predictions and carry it forward to the test data.

**Printing performance:** Printing a performance result displays two metric tables. The first contains classification rates and their F1 and G-mean combinations; the second contains overall performance, including G-mean, AUC, PR-AUC, misclassification, Brier scores, and log loss. The same G-mean row appears in both tables; F1-related combinations appear only in the first.

`Delta` and percentage `Gain` compare the model with its random-score reference, with positive values indicating improvement. Larger rates, G-mean, and AUC values are better; smaller errors and losses are better. `digits` controls the display precision, and `show.confusion = FALSE` omits the optional confusion tables. These settings do not change the stored performance values.

## Value

`imbalanced` returns an object from the selected forest grow function. Usual components include the class-probability matrices `predicted` and `predicted.oob`, class assignments `class` and `class.oob`, prediction error `err.rate`, and requested VIMP `importance`. Prediction error and VIMP use the selected `perf.type`; for RFQ and BRF the default is normalized Brier. The first column of `err.rate` and `importance` contains the overall error and VIMP, respectively. Availability depends on the selected grow interface and sampling options; see `rfsrc` and `rfsrc.fast`.

**Performance and threshold helpers:**

`get.imbalanced.performance` A named numeric vector of class counts, imbalance ratio, resolved numeric threshold, and performance measures. With `confusion = TRUE`, a named list also includes `confusion`, the confusion matrix with a `class.error` column. Both forms have class `"imbalanced.performance"` and a `"rand"` attribute containing the random-reference metrics and expected confusion counts.

`get.imbalanced.optimize` A one-row data frame of performance measures at the selected threshold. Extract the cutoff from the `threshold` column; the row name is its grid index.

`get.rfq.threshold` The smaller class count divided by the total observed class count, as a numeric scalar. Returns `NULL` unless the class-frequency table has exactly two entries. No forest or probability predictions are required.

The performance and optimization helpers return `NULL` when the evaluation response is not a two-level factor.

`print.imbalanced.performance` displays the summaries and returns `x` invisibly.

## 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, get.imbalanced.performance, vimp, subsample

## Examples

```
## ------------------------------------------------------------
## RFQ, BRF, and standard classification
## ------------------------------------------------------------
data(breast, package = "randomForestSRC")
dta <- na.omit(breast)
f <- status ~ .

## Use 300 trees for illustration; imbalanced() defaults to 3000.
set.seed(19)
rfq <- imbalanced(f, dta, ntree = 300)
print(rfq)
print(get.imbalanced.performance(rfq, confusion = TRUE))

## Equivalent direct RFQ call using rfsrc().
## RFQ uses equal case weights and sampling without replacement.
set.seed(19)
rfq.direct <- rfsrc(f, dta, ntree = 300,
                    rfq = TRUE, splitrule = "auc", perf.type = "brier",
                    block.size = NULL, bootstrap = "by.root",
                    samptype = "swor", sampsize = function(n) n * .632,
                    case.wt = NULL)
print(all.equal(rfq$predicted.oob, rfq.direct$predicted.oob))

## Extract the class-frequency threshold from the observed responses.
print(get.rfq.threshold(rfq$yvar))

set.seed(20)
brf <- imbalanced(f, dta, ntree = 300, method = "brf")

## Equivalent direct BRF call using rfsrc().
## Each class has equal total sampling weight; sampling is with replacement.
class.frequency <- table(dta$status)
case.wt <- min(class.frequency) /
  as.numeric(class.frequency[as.character(dta$status)])
set.seed(20)
brf.direct <- rfsrc(f, dta, ntree = 300,
                    rfq = FALSE, splitrule = "auc", perf.type = "brier",
```

```
                    block.size = NULL, bootstrap = "by.root",
                    samptype = "swr", sampsize = 2L * min(class.frequency),
                    case.wt = case.wt)
print(all.equal(brf$predicted.oob, brf.direct$predicted.oob))

standard <- imbalanced(f, dta, ntree = 300, method = "standard")

## Compare all three using the helper's default evaluation rule.
metrics <- c("sens", "spec", "gmean", "misclass", "brier.norm", "auc")
comparison <- rbind(
  RFQ = as.numeric(get.imbalanced.performance(rfq)[metrics]),
  BRF = as.numeric(get.imbalanced.performance(brf)[metrics]),
  Standard = as.numeric(get.imbalanced.performance(standard)[metrics])
)
colnames(comparison) <- metrics
print(comparison)

## Supply a cutoff explicitly to inspect another probability rule.
print(get.imbalanced.performance(standard, threshold = .5))

## Let the saved classifier setting select prevalence or 0.5.
print(get.imbalanced.performance(rfq, threshold = "auto"))
print(get.imbalanced.performance(standard, threshold = "auto"))
print(get.imbalanced.performance(brf, threshold = "auto"))

## Request the 0.5 Bayes cutoff explicitly, including for RFQ.
print(get.imbalanced.performance(rfq, threshold = "bayes"))

## Vectors, one-column objects, and named two-column matrices.
class.frequency <- table(rfq$yvar)
minority <- names(class.frequency)[which.min(class.frequency)]
p <- rfq$predicted.oob[, minority]
print(get.imbalanced.performance(rfq$yvar, p))
print(get.imbalanced.performance(rfq$yvar, data.frame(p = p)))
print(get.imbalanced.performance(
  rfq$yvar, rfq$predicted.oob[, rev(levels(rfq$yvar)), drop = FALSE]
))

## ------------------------------------------------------------
## Train/test assessment with a training-selected threshold
## ------------------------------------------------------------
set.seed(23)
trn.id <- sort(unlist(lapply(
  split(seq_len(nrow(dta)), dta$status),
  function(idx) idx[sample.int(length(idx),
    size = floor(2 * length(idx) / 3))]
)))
trn <- dta[trn.id, , drop = FALSE]
tst <- dta[-trn.id, , drop = FALSE]
print(rbind(training = table(trn$status), test = table(tst$status)))

fit <- imbalanced(f, trn, ntree = 300)
selected <- get.imbalanced.optimize(fit, measure = "gmean",
```

```
                                      ngrid = 501, plot.it = FALSE)
print(selected)
threshold.selected <- selected$threshold

pred <- predict(fit, newdata = tst)
print(get.imbalanced.performance(pred))
print(get.imbalanced.performance(pred, threshold = threshold.selected))

## Predicting labels for new rows does not require their outcomes.
class.frequency <- table(trn$status)
minority <- names(class.frequency)[which.min(class.frequency)]
majority <- setdiff(levels(trn$status), minority)
pred.x <- predict(fit, newdata = tst[, fit$xvar.names, drop = FALSE])
selected.class <- factor(
  ifelse(pred.x$predicted[, minority] >= threshold.selected,
         minority, majority),
  levels = levels(trn$status)
)
print(head(selected.class))

## ------------------------------------------------------------
## Explicit permutation VIMP and blocking
## ------------------------------------------------------------
## RFQ uses normalized Brier performance by default.
rfq.vimp <- imbalanced(f, dta, ntree = 300,
                       importance = "permute", block.size = 1)
rfq.block <- imbalanced(f, dta, ntree = 300,
                        importance = "permute", block.size = 10)
print(cbind(unblocked = rfq.vimp$importance[, 1],
            blocked = rfq.block$importance[, 1]))

## Small B is for illustration; increase it for final inference.
smp <- subsample(rfq.block, B = 25, verbose = FALSE)
print(extract.subsample(smp)$var.sel.Z)
plot.subsample(smp, cex.axis = .7, alpha = .5)

## ------------------------------------------------------------
## Strong imbalance: probability signals and permutation VIMP
## ------------------------------------------------------------
## x1 identifies a small subgroup; x2 changes risk within it.
## Both affect the outcome probability.  The other variables are noise.
## Expected minority prevalence is .04 * (.15 + .5 / 2) = .016.
## Within the subgroup, all true probabilities exceed this prevalence,
## so the population RFQ class rule uses x1 alone.  Brier performance
## can also detect the probability information supplied by x2.
local({
  ## One thread makes the blocked VIMP comparison reproducible.
  old.options <- options(rf.cores = 1)
  on.exit(options(old.options))

  set.seed(47)
  n <- 4000
  p <- 20
```

```
  x <- matrix(runif(n * p), ncol = p)
  x[, 1] <- as.numeric(x[, 1] < .04)
  colnames(x) <- paste0("x", seq_len(p))
  prob <- x[, 1] * (.15 + .5 * x[, 2])
  y <- factor(rbinom(n, 1, prob), levels = c(0, 1),
              labels = c("majority", "minority"))
  sim.data <- data.frame(y = y, x)

  ## Grow one RFQ forest, then use the same trees for both metrics.
  ## Brier is explicit here; it is also the RFQ default.
  sim.fit <- imbalanced(y ~ ., sim.data, ntree = 500, mtry = p,
                        splitrule = "auc", perf.type = "brier",
                        block.size = 50, seed = -103)
  brier <- vimp(sim.fit, importance = "permute", perf.type = "brier",
                block.size = 50, seed = -107)
  gmean <- vimp(sim.fit, importance = "permute", perf.type = "gmean",
                block.size = 50, seed = -107)

  counts <- table(y)
  cat("\nClass counts and majority/minority ratio\n")
  print(c(n.majority = unname(counts["majority"]),
          n.minority = unname(counts["minority"]),
          iratio = unname(counts["majority"] / counts["minority"])))

  ## Compare signals with noise within each metric's own scale.
  vimp.values <- cbind(Brier = brier$importance[, 1],
                       Gmean = gmean$importance[, 1])
  cat("\nPermutation VIMP: true signals and largest noise value\n")
  print(rbind(vimp.values[1:2, , drop = FALSE],
              largest.noise = apply(vimp.values[-(1:2), , drop = FALSE],
                                    2, max)), digits = 5)

  ## A smaller rank is better; the ideal ranks are 1 and 2.
  vimp.ranks <- apply(vimp.values, 2, function(z) {
    rank(-z, ties.method = "average")
  })
  cat("\nRanks of the two true signals among all 20 variables\n")
  print(vimp.ranks[1:2, , drop = FALSE])
})
```

---

`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 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.

If `supervised.formula` is supplied, the function also fits an internal supervised forest from the training data. The supervised forest is fit after training imputation and provides auxiliary predictors for test-time imputation and OOD scoring. The auxiliary predictors use supervised information learned from the training outcomes. Supervised outcomes supplied with new data are dropped and are not used at deployment time. Leave `supervised.formula` unspecified to use the learned imputer without these auxiliary predictors.

## 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,
  ...,
  supervised.formula = NULL,
  supervised.args = list(),
  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,
  compress = 50,
  share.xvar = FALSE)

save.impute.learn.rfsrc(object, path, wipe = TRUE, verbose = TRUE,
  compress = 50, share.xvar = NULL)

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,
  return.reconstruction = FALSE,
  verbose = TRUE,
  ...)
```

## Arguments

| | |
|---|---|
| formula | An optional symbolic model description passed to impute for the initial training-data imputation when supervised.formula is not supplied. It follows the same guidelines as in impute. This formula does not specify the full-sweep learner bank, the OOD targets, or the supervised auxiliary forest. When supervised.formula is supplied, the raw predictor block is taken from the right-hand side of supervised.formula, and formula is not used. |
| data | Training data, converted to a plain data frame before processing. Matrices, tibbles, and data.table objects can be supplied. Column names must be unique and nonempty, each column must be a vector, and numeric values must be finite or missing. Bare vectors and nested matrix or list columns are not supported. 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 training begins. If supervised.formula is supplied, data should contain both the supervised response columns and the raw predictor columns. The learned imputer is then built on the raw predictor block defined by the right-hand side of supervised.formula. |
| 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. fast is also forwarded to the saved target learners and, unless overridden by supervised.args, to the supervised forest. verbose controls progress throughout training, prediction, and scoring. The same verbose, max.predict.iter, eps, and cache.learners controls are also used by predict.impute.learn and impute.ood. Iteration and forest-size controls must be scalar integers in their permitted ranges; max.iter |

must be at least one.

`mf.q` Controls the imputation engine used by `impute`. With `mf.q = 1` and `always.use = NULL`, targets are updated one at a time using `missForest`. Other positive settings use the multivariate `missForest` generalization; a fraction below one controls the proportion of missing-data variables grouped as responses, and a value above one specifies their requested group size. A non-`NULL` `always.use` selects the multivariate branch, including an empty vector or a vector with no matching column names. If `mf.q` is omitted, training uses on-the-fly imputation. The selected method is recorded in `manifest$train.imputation`.

`eps` Finite nonnegative 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.

`supervised.formula`
Optional supervised learning formula used to augment the learned imputer for improved OOD detection in supervised settings. The left-hand side defines the supervised response and the right-hand side defines the raw predictor block to be learned by `impute.learn`. The supervised forest is fit internally after the raw predictor block has been completed. Its training out-of-bag `predicted` values and test-time `predicted` values are appended internally as auxiliary predictors. Training auxiliary values use out-of-bag predictions when available, with ordinary forest predictions and training means used as fallbacks. At deployment, auxiliary values are computed once from the initialized raw predictors and held fixed during the imputation passes. Predictors must be raw column names; nonsyntactic names can be enclosed in backticks. Interactions and transformations are not supported. These auxiliary predictors are internal and are not expected in `newdata`.

`supervised.args`
Optional named list of arguments passed to the internal supervised `rfsrc` fit. Entries for `formula`, `data`, and `forest` are controlled internally and are ignored if supplied. All other entries override the defaults used for this supervised fit, including defaults inherited from top-level arguments such as `fast` and internally supplied defaults such as `perf.type = "none"`.

`full.sweep.options`
A named `list` of options used when fitting saved target learners after initial training-data imputation. Defaults are `ntree = 100`, `nodesize = NULL`, and `nsplit = 10`, independently of the corresponding settings used for initial imputation. Recognized entries include `ntree`, `nodesize`, `nsplit`, `mtry`, `splitrule`, `bootstrap`, `sampsize`, `samptype`, `perf.type`, `rfq`, `save.memory`, `importance`, `proximity`, `terminal.qualts`, and `terminal.quants`. To request on-the-fly prediction for the saved target learners, supply `terminal.qualts = FALSE` and `terminal.quants = FALSE` in this list. These settings are passed to the fitting function selected by `anonymous`; they do not change the options for initial imputation or for the supervised auxiliary forest, which uses `supervised.args`. Unknown entries are ignored with a warning. Duplicate names produce a warning, and the last entry for each name is used.

`target.mode` Determines which raw 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 retained raw variable. If the training data are complete, `target.mode = "all"` must be used. When `supervised.formula` is supplied, the default is promoted internally to `"all"` so that every retained raw predictor can later be updated and scored. For the broadest OOD coverage, `target.mode = "all"` is recommended so every deployment-time variable can be reconstructed.

`deployment.xvars`
Controls which raw predictors are assumed to be available later when the saved imputer is used on new data. If `NULL`, all raw non-target columns are used. If a character vector, the same raw predictor set is used for all targets. If a named list, names must be target variables and each entry gives the raw predictor set for that target; targets omitted from the list fall back to all raw non-target columns. Unnamed or duplicated list entries are not allowed. Entries named for non-target variables, and predictor names not found in the training data, are ignored with a warning. When `supervised.formula` is supplied, the internal auxiliary `predicted.*` columns are added automatically on top of these raw predictor sets. In other words, `deployment.xvars` restricts the raw predictors only.

`anonymous` If `TRUE`, uses `rfsrc.anonymous` when fitting the saved target-wise full-sweep learner bank. The internal supervised forest, when requested through `supervised.formula`, is fit with `rfsrc` because it must be retained for later prediction-time auxiliary variables. Thus `anonymous` usually reduces the size of the target-wise learner bank, but not necessarily the size of the supervised forest.

`learner.prefix, learner.root`
Names used when writing saved full-sweep learners to disk. If `supervised.formula` is supplied, the internal supervised forest is saved under the same learner root. `learner.root` must be a relative directory path within the imputer directory. `learner.prefix` must be a single file-name component. Absolute paths and parent-directory components are not allowed.

`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 learner components are serialized in FST format. If `supervised.formula` is supplied, the internal supervised forest is also saved there.

`wipe` If `TRUE`, replaces an existing output directory after the new imputer has been written and verified. If `FALSE`, unrelated existing files are retained. Saving uses a temporary bundle and requires additional disk space.

`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`. If `supervised.formula` is supplied, the internal supervised forest is kept in memory under the same rule.

`keep.ximp` If `TRUE`, keeps the completed raw training predictor table 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.

`compress` A single number from 0 to 100 controlling lossless FST compression of saved learners. The default, 50, balances file size and saving speed. Use 100 for

stronger compression at the cost of slower saving. Applies to the target learners and the optional supervised forest. In `impute.learn.rfsrc`, this controls saving during fitting; in `save.impute.learn.rfsrc`, it controls the current save call and defaults to 50 independently of any earlier setting. It does not change forest fitting or prediction.

`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.

`share.xvar` Share the completed training predictor table across non-anonymous target learners instead of writing a separate '`xvar.rda`' for each learner. The fitting default is `FALSE`. Set `share.xvar = TRUE` and `anonymous = FALSE` when fitting to capture the common table and exact learner row/column mappings. The option applies to saving during fitting and to later explicit saving. For `save.impute.learn`, `NULL` uses the object's shared-storage setting; `FALSE` writes self-contained learners. An older bank fitted without shared mappings cannot be converted by setting this argument alone. See Details.

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

`path` Directory containing a saved imputer. Use a dedicated directory rather than a filesystem root, home directory, working directory, or an ancestor of these. A complete bank can be saved back to its source path. Otherwise, source and destination directories must not contain one another. An object loaded with a target subset must be saved to a different directory. Save and load operations require the **fst** package. Use `load.impute.learn` for a complete imputer, including its shared predictors when present.

`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. A load-time subset limits which learners are available for predictor completion. It can therefore differ from scoring a subset of a fully loaded bank when other predictors are missing.

`lazy` If `TRUE`, saved learners are loaded only when they are needed. If `FALSE`, all saved learners are loaded at once. If `supervised.formula` was used at fit time, the internal supervised forest follows the same lazy versus eager loading rule.

`newdata` New data to be imputed or scored, converted to a plain data frame before processing. The column-name and vector-column requirements for `data` also apply. Missing columns are added and extra columns are dropped to match the training schema. A zero-row table returns correctly typed empty output without calling forests. Retained numeric values must be finite or missing. Nonmissing values that cannot be converted to a required numeric type become missing with a warning; their row indices are recorded in `conversion.issues`. Unseen factor levels are converted to `NA` for harmonization, but they are also tracked row-wise.

In supervised mode, newdata should contain only the raw predictor columns learned by the imputer. The internal auxiliary columns are created automatically and should not be supplied by the user. If supervised response columns are supplied in newdata and are not part of the learned raw predictor block, they are treated as extra columns and dropped; they are not used to compute auxiliary predictors, imputations, or OOD scores. In impute.ood, observed raw target values are used to compute target-wise discrepancies. Missing raw target values are completed for predictor-side reconstruction but do not themselves contribute a finite target-wise OOD score. 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. Must be a nonnegative integer. Zero performs initialization without iterative target updates; OOD reconstruction is still performed.

restore.integer

If TRUE, generated imputations in integer-supported variables are rounded to the integer grid. This includes columns stored as R integers and numeric columns whose observed finite training values were all integer-valued. Observed numeric values in newdata are preserved, including fractional values, both as predictors during imputation and in the returned data. Restoration applies to initialization values and model-based updates. Numeric training columns retain numeric storage. Columns stored as R integers during training are returned as integer vectors only when every nonmissing completed value is exactly integral and within R's integer range; otherwise numeric storage preserves observed fractions and out-of-range values. Set to FALSE to retain unrounded generated numeric values. 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 $1 - \prod_j \max(1 - u_j, \varepsilon)^{\tilde{w}_j}$, where $u_j$ is the calibrated score for target $j$ and $\tilde{w}_j$ denotes weights normalized over the positively weighted, scoreable targets in the row. `"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 $-\log\{\max(1 - u_j, \varepsilon)\}$ 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 named 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`. The power must be finite and at least one; the probability floor must satisfy $0 < \varepsilon < 1$. The top-$k$ value must be finite, at least one, and no larger than `.Machine$integer.max`; it is rounded to an integer. 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 richer `info` list.

`return.reconstruction`
If `TRUE`, `impute.ood` returns the target-wise reconstructed values used during OOD scoring. The reconstructed values are returned on the raw target scale, with one column per scored target. Typically these values are computed internally and discarded.

## Details

**Training:** Training begins by converting variables that are not real-valued 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. The imputer is then fitted in two stages:

1. **Complete the training data.** Missing values are imputed using `impute`, with the same options as that function. With `mf.q = 1` and `always.use = NULL`, targets are updated one at a time. Other positive settings use the multivariate `missForest` generalization.
   If `mf.q` is omitted, on-the-fly imputation is used when `formula` is specified; otherwise default unsupervised imputation is used. Complete training data with `target.mode = "all"` skip this stage.
2. **Fit the saved learners.** A full forest sweep is fitted on the completed training data. For each target selected by `target.mode`, a forest is fitted using rows where that target was originally observed and predictors selected by `deployment.xvars`. All saved learners use the same completed training table; fitting them does not further update that table.

Training stops if every requested target learner fails. If some succeed, a partial bank is returned with one summary warning. `manifest$learners` records each target's status and error, and printing the imputer reports successful and unavailable learners.

**Choosing predictors:** By default, `deployment.xvars = NULL` uses every non-target column as a predictor. Restrict `deployment.xvars` when the training data include outcomes, future-only variables, identifiers, or other fields that could introduce leakage or will be unavailable in new data.

**Supervised auxiliary predictors:** Supplying `supervised.formula` fits an internal supervised forest. Its out-of-bag `predicted` values for training data and `predicted` values for new data provide auxiliary predictors for the saved learners. These variables can influence both imputation and OOD scoring, but are not imputation targets. Leave `supervised.formula` unspecified to use the basic unsupervised learned imputer.

The auxiliary variables are created automatically and added to each saved target's predictor set. `deployment.xvars` restricts only the raw predictors; users do not need to supply auxiliary columns in new data.

**Saving and loading:** A saved imputer consists of a manifest and a directory of learners, with an optional shared predictor table. Learner components use the same serializers as `fast.save` and `fast.load`, so saving and loading require the **fst** package. The supervised forest, when used, is saved alongside the target learners. The explicit save method can save learners from memory or load them from an attached saved path.

Set `compress = 100` for stronger FST compression when saving a deployment bundle. Supply this directly to `impute.learn` or `save.impute.learn`, rather than in `full.sweep.options`. The manifest and remaining RDS metadata use their existing compression settings. Loading requires no compression argument.

Both training-time and explicit saving write and verify the new learners before replacing an existing destination. A failed staging operation leaves the previous saved imputer unchanged. If replacement fails, restoration of the previous directory is attempted; an unrecovered backup path is reported.

**Shared training predictors:** With `share.xvar = TRUE`, the completed training table used by the final target learners is saved once in '`_shared_xvar/xvar.fst`'. In supervised mode this table also includes the auxiliary training predictors actually used in those fits. Each target stores its exact training-row indices, predictor names and column encodings. These reconstruct the learner's training predictors before prediction. Reconstruction is checked against the fitted forest before its separate '`xvar.rda`' is omitted. Trees, responses, terminal information and OOD references keep their existing representations. The single supervised forest remains self-contained.

Shared storage requires `anonymous = FALSE`. It is independent of `compress`, `terminal.qualts` and `terminal.quants`; their existing settings still apply. `keep.ximp` continues to control the returned raw completed table. An in-memory shared-capable fit separately holds the common predictor source needed for deferred saving. Saved shared banks and eagerly loaded shared objects can be saved again without reconstructing row identities from predictor values.

Load the whole imputer with `load.impute.learn`. Shared learner subdirectories are not independently portable, and direct `fast.load` calls on them stop with an explanatory message. Keep the manifest, shared table and learners together when moving or archiving the bank. A compatible shared-aware package version is required. The updated loader also accepts the existing independent-learner format. Saving a shared object with `share.xvar = FALSE` writes that independent format.

Lazy loading leaves the common table on disk until first use. Each prediction or OOD operation loads it at most once and reuses it across learner accesses, including when `cache.learners = "none"`. Reconstructed learners remain normal forest objects: caching or eagerly loading many learners can still use memory for their individual predictor tables. Shared storage primarily reduces repeated serialization and on-disk duplication. Missing shared files, checksum mismatches and invalid mappings cause errors rather than silently substituting initialization fills.

**Imputing new data:** Prediction applies the saved learners in three steps:

1. **Match the training schema.** Columns and types in `newdata` are matched to the training schema. Supervised response columns outside the learned raw predictor block are dropped and are not used for prediction or OOD scoring.
2. **Initialize missing values.** Missing raw values are filled with training means or modes. In supervised mode, the saved supervised forest then computes auxiliary predictors from this initialized table. The auxiliary predictors stay fixed throughout the subsequent passes.
3. **Update with saved learners.** Full-sweep passes update missing values in the selected targets. Each target update requires one valid prediction per requested row. Failed or unavailable predictions leave the previous imputed values unchanged and are recorded in `target.issues`. A pass with no valid model updates is reported separately from convergence.

With `target.mode = "missing.only"`, a variable that was complete in training but is missing in new data receives only an initialization value. Use `target.mode = "all"` when missing values may appear later in any raw variable. Complete training data also require this setting because there are no missing variables from which to select the saved targets.

Integer restoration applies only to generated values, including initialization values in columns without a target update. Observed numeric entries are unchanged.

**OOD training references:** With `save.ood = TRUE`, each saved learner's out-of-bag predictions are compared with observed training values to form target-wise reconstruction discrepancies:

- Continuous and integer targets use absolute reconstruction error.
- Factor targets use negative log predictive probabilities. Unavailable or invalid probabilities give missing discrepancies; zero probabilities are scored using the probability floor.

The references are stored in the manifest. Each learner entry records `n.oob.finite` and `n.oob.nonfinite`.

The training-time row reference combines calibrated target scores using a weighted mean. If `weight` is omitted, all saved OOD targets receive weight 1. Named weights are matched to targets, and omitted targets receive weight 0. These weights are also the defaults for later OOD scoring.

**OOD scoring:** `impute.ood` first completes the predictors in `newdata` using the same schema matching, initialization, and full-sweep passes as `predict.impute.learn`. It then reconstructs each requested raw target from its saved learner and compares the reconstruction with the observed value in `newdata`. A target that is missing in a row does not contribute to that row's OOD score.

Discrepancies are converted to target-wise OOD scores using the saved target-specific training references. Values strictly above a nonempty reference's maximum, including positive infinity, receive its largest stored probability. Ties, including equality at the largest quantile, follow the reference's quantile-grid convention. Missing discrepancies and empty references are unscored.

In supervised mode, the auxiliary variables are predictors for reconstruction; the supervised response is not used in new data.

**Row-level OOD scores:** `impute.ood` returns two row-level summaries:

`score` Combines calibrated target scores over the targets that are observed and scoreable in each row. The default is a bounded product rule. Alternatives are a weighted mean, weighted $L_p$, log-tail weighted $L_p$, and top-$k$ rules. These options allow greater sensitivity to sparse but severe coordinate shifts. Unless overridden, scoring uses the weights saved by `impute.learn`.

`score.percentile` Calibrates the row score against a training reference rebuilt from the saved target-wise training OOD scores. The reference uses the requested target subset, weights, and row aggregate, so percentile calibration remains available when any of these settings change.

**Unseen factor levels:** Unseen factor levels are tracked by row when matching new data to the training schema. `impute.ood` flags these rows and assigns them the maximum row-level score. If an unseen level occurs in a scored target, its target-level discrepancy is also 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 internal supervised forest when `supervised.formula` is used, optionally the completed raw training predictor table, 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. When supervised mode is active, the manifest also records the supervised family, response names, and the internally created auxiliary `predicted.*` variable names.

Shared-capable fits also contain `manifest$shared.xvar`, the common-table specification, and per-learner `xvar.ref` reconstruction mappings. `shared.xvar.data` holds the common table for in-memory fits with `keep.models = TRUE` and eagerly loaded shared objects. It is `NULL` for disk-backed lazy objects. This component is separate from `ximp`.

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

`predict.impute.learn` returns a data frame with imputed values overlaid on the raw predictor table, retaining its row names. 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, dropped supervised response columns when present, row-wise unseen-factor flags, supervised-auxiliary diagnostics when present, and any targets skipped because a learner was unavailable or a prediction failed. `pass.updated.cells` and `pass.failed.cells` count accepted updates and failed updates in each pass; `converged` and `stopping.reason` distinguish convergence from initialization-only, empty-input, iteration-limit, and failed-update stopping. `conversion.issues` records the row indices of nonmissing values that became missing during numeric conversion. In both prediction and OOD diagnostics, `n.disk.loads` counts successful target-learner load operations, including repeated loads with `cache.learners = "none"`. `disk.load.targets` lists the distinct targets loaded. Supervised forest loading is reported separately in `info$supervised` and is not included in the target-learner count.

`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"`, the probability floor bounds the mathematical aggregate by $-\log(\varepsilon)$ for $0 < \varepsilon < 1$. With the default `eps = 1e-12`, the upper bound is approximately 27.63.
- `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`.
- `target.reconstruction`: when `return.reconstruction = TRUE`, a data frame containing the saved learners' predictions for the scored targets.
- `reconstructed.data`: when `return.reconstruction = TRUE`, the harmonized raw table with scored targets replaced by their reconstructions. Other columns retain their harmonized values. Integer restoration applies to generated reconstructions, preserving observed numeric values in the other columns.
- `completed.data`: when both `return.details` and `return.reconstruction` are `TRUE`, the raw predictor table after prediction-time imputation. This is distinct from the target reconstruction table.
- `info`: a list of diagnostics including harmonization details, dropped supervised response columns when present, row-wise unseen-factor flags, learner-loading information, supervised-auxiliary diagnostics when present, 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`, `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, ]

## training the imputer
fit <- impute.learn(
  data = train,
  ntree = 25,
```

```
  mf.q = 1,
  max.iter = 5,
  full.sweep.options = list(ntree = 25, nsplit = 5)
)

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


## ------------------------------------------------------------
## Shared training predictors for a deployment bank
## ------------------------------------------------------------
## Keep these objects separate from the airquality examples below.
local({
  if (requireNamespace("fst", quietly = TRUE) &&
      requireNamespace("data.table", quietly = TRUE)) {
    set.seed(17)
    train <- data.frame(x1 = rnorm(100), x2 = runif(100),
                        group = factor(rep(c("a", "b"), 50)))
    bank.dir <- tempfile("shared-imputer-")
    fit <- impute.learn(
      data = train, target.mode = "all", anonymous = FALSE,
      share.xvar = TRUE, compress = 100, out.dir = bank.dir,
      full.sweep.options = list(ntree = 50, nsplit = 5,
                                terminal.qualts = FALSE, terminal.quants = FALSE),
      verbose = FALSE
    )
    imp <- load.impute.learn(bank.dir, verbose = FALSE)
    test <- train[1:5, ]
    test$x1[c(1, 3)] <- NA_real_
    print(predict(imp, test, verbose = FALSE))
    unlink(bank.dir, recursive = TRUE)
  }
})



## 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)
print(head(ood$score))
print(head(ood$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)
print(head(ood.lp$score.percentile))

## ------------------------------------------------------------
## supervised OOD example: regression benchmark
## the user supplies only the raw x-columns at test time
## ------------------------------------------------------------

friedman1_sim <- function(n = 150, p = 10, sigma = 1) {
  X <- matrix(runif(n * p), nrow = n)
  y <- 10 * sin(pi * X[, 1] * X[, 2]) +
       20 * (X[, 3] - 0.5)^2 +
       10 * X[, 4] + 5 * X[, 5] +
       rnorm(n, sd = sigma)
  list(X = X, y = y)
}


trn <- data.frame(friedman1_sim())
tst <- data.frame(friedman1_sim())
xvars <- setdiff(names(trn), "y")


## impute data using missForest, construct a supervised forest
## - supervised forests are used to create auxiliary variables
## - improves test time OOD in supervised problems

sup.fit <- impute.learn(
  data = trn,
  mf.q = 1,
  supervised.formula = y ~ .,
  supervised.args = list(ntree = 50, nsplit = 5),
  full.sweep.options = list(ntree = 25, nsplit = 5),
  save.ood = TRUE,
  verbose = FALSE
)

## add some missing values to the test data
xnew <- tst[, xvars, drop = FALSE]
xnew[sample(seq_len(nrow(xnew)), 5), xvars[1]] <- NA
xnew[sample(seq_len(nrow(xnew)), 5), xvars[2]] <- NA

## imputation
```

```
xnew.imp <- predict(sup.fit, xnew, max.predict.iter = 2, verbose = FALSE)
print(head(xnew.imp))

## OOD score
ood.sup <- impute.ood(sup.fit, xnew, verbose = FALSE)
print(head(ood.sup$score.percentile))

## ------------------------------------------------------------
## 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
)

## Stronger compression for a deployment bundle.
save.impute.learn(fit, bundle.dir, compress = 100, 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)
print(which(ood$info$unseen.rows))
print(summary(test.imp))
unlink(bundle.dir, recursive = TRUE)
```

---

`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, restore.integer = TRUE,
  ...)
```

## Arguments

| | |
|---|---|
| `formula` | A symbolic model description used to specify on the fly imputation. Can be omitted for unsupervised imputation in settings where the outcomes are unspecified or if distinction between outcomes and predictors is unnecessary. Ignored for missForest. |

`data` Data to be imputed. Matrices, tibbles, and `data.table` objects are converted to a plain data frame. Column names must be unique and nonempty, each column must be a vector, and numeric values must be finite or missing. Bare vectors and nested matrix or list columns are not supported.

`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. Must be a positive integer.

`eps` Minimum decrease in the between-pass change statistic required to continue missForest iterations. Must be finite and nonnegative.

`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.

`restore.integer`
If `TRUE`, generated values in variables that were integer supported in the original data are rounded to the integer grid. This includes columns stored as R integers and numeric columns whose observed finite values are all integer-valued. Only originally missing cells are rounded; observed numeric values are preserved. Numeric columns retain numeric storage. Columns stored as R integers are returned as integer vectors only when every nonmissing completed value is exactly integral and within R's integer range; otherwise numeric storage is retained. Set to `FALSE` to retain unrounded generated numeric values.

`...` Additional arguments passed to or from methods. Recognized advanced options include `full.sweep.options` (a named `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. For a table containing both observed and missing values, observations and variables with all values missing are removed before imputation. A complete table or a table whose entries are all missing is returned without imputation after container normalization.
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 response-wise missForest when `always.use = NULL`. A non-`NULL` `always.use`, including an empty or entirely unmatched vector, selects the multivariate branch. The change statistic averages factor disagreement and scaled numeric changes over originally missing entries. Iteration stops when this statistic no longer decreases by at least `eps`, or when `max.iter` is reached. The last completed pass is retained.
5. If no formula is provided and `mf.q` is omitted, unsupervised splitting is used. The default `ytry` is `sqrt(p)`, where `p` is the number of retained 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. Additional forest controls forwarded through `...` are `mtry`, `splitrule`, `bootstrap`, `sampsize`, and `samptype`. The optional final sweep has its own `full.sweep.options` list.
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.
9. If `restore.integer = TRUE`, integer support is detected from the original retained data before imputation begins. Numeric values are compared with the nearest integer using an absolute floating-point tolerance that does not increase with their magnitude. Restoration is applied after the missForest rough-fix initialization, before and after the optional final sweep, and once more before returning the completed data. The missForest iterative updates themselves are not rounded between variable or block updates.

## 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
```

`multivariate.values` *Extracting Multivariate Values*

## Description

Extract predictions, performance errors, variable importance (VIMP), and case-specific errors and VIMP from random forest fits and predictions. The functions combine response-specific results from multivariate regression, multivariate classification, and mixed-outcome forests. They also work with univariate forests. `get.mv.formula()` constructs a multivariate formula from response names.

## Usage

```
get.mv.predicted(obj, oob = TRUE)

get.mv.error(obj, standardize = FALSE, pretty = TRUE, block = FALSE)

get.mv.error.block(obj, standardize = FALSE)

get.mv.vimp(obj, standardize = FALSE, pretty = TRUE)

get.mv.cserror(obj, standardize = FALSE)

get.mv.csvimp(obj, standardize = FALSE)

get.mv.formula(ynames)
```

## Arguments

| | |
|---|---|
| `obj` | An object returned by `rfsrc()` or `predict.rfsrc()`. For multivariate and mixed-outcome objects, `obj$yvar` must contain the response columns with their original types and factor levels. |
| `oob` | If `TRUE`, use `predicted.oob` for each response, falling back to `predicted` when that response's OOB component is `NULL`. If `FALSE`, use `predicted`. |
| `standardize` | If `TRUE`, divide regression errors and VIMP by the corresponding response variance in `obj$yvar`. Classification and survival values are unchanged. See Standardization for the treatment of zero or missing variances. |
| `pretty` | If `TRUE`, return errors as a named vector and VIMP as a matrix, using only the overall `all` value for each classification response. If `FALSE`, return a list named by response, including class-specific values. Competing-risk event-specific values are included in both formats. Values are not rounded. |
| `block` | If `TRUE`, return the block-error sequence from `err.block.rate` instead of the final error from `err.rate`. This sets `pretty = FALSE`. `get.mv.error.block()` is shorthand for this choice. |
| `ynames` | Character vector of response column names for `get.mv.formula()`. These columns must be in the data used to fit the forest. |

## Details

The extraction functions use results already computed during fitting or prediction. Results are reported separately for each response. Subset the returned vector, matrix, or list to select responses or predictors.

**Predictions:** `get.mv.predicted()` returns a matrix with observations in rows and predictions in columns:

- Regression: one column per response.
- Classification: one probability column per class, named `response.class`.
- Right-censored survival: the mortality prediction.
- Competing risks: one prediction column per event, named `response.event`.

Responses follow `obj$yvar.names`; class and event columns follow their order in the object. Time-indexed survival, cumulative hazard, and cumulative incidence arrays are not included.

By default, OOB predictions are used for each response when its `predicted.oob` component is present. Missing values within that component remain missing, even if every value is missing. The function uses `predicted` only when the entire OOB component is `NULL`. Set `oob = FALSE` to use full-ensemble or new-data predictions.

**Performance errors and variable importance:** `get.mv.error()` returns the final performance error for each response, using the error measure selected during fitting or prediction. With `pretty = FALSE`, classification results include `all` and the class-specific errors. Survival results include all error columns, including event-specific errors for competing risks.

`get.mv.error.block()` returns the full block-error sequence computed during fitting or prediction. It is equivalent to `get.mv.error(obj, standardize = standardize, block = TRUE)`; the block size is determined when those errors are computed.

`get.mv.vimp()` extracts variable importance. Request importance when fitting the forest or use `vimp` before calling this function. With `pretty = TRUE`, the result is a matrix with predictors in rows and response results in columns. With `pretty = FALSE`, each response has its own matrix, including classification `all` and class-specific columns and competing-risk event columns.

**Case-specific values:** `get.mv.cserror()` calculates case-specific error as `cse.num / cse.den`; `get.mv.csvimp()` calculates case-specific VIMP as `csv.num / csv.den`. These numerator and denominator components must be saved in the fitted or prediction object. Case-specific error is calculated from these components, not by applying a loss to `predicted.oob` and the observed response.

With one response, values are returned directly; with multiple responses, they are returned in a list named by response. Case-specific VIMP has observations in rows and VIMP variables in columns, with variable names taken from `importance` when available. Both functions return `NULL` for right-censored survival and competing risks.

**Standardization:** For a regression response $Y$, `standardize = TRUE` divides each error or VIMP value by `var(Y, na.rm = TRUE)`. The variance uses the response values in `obj$yvar`: training responses for a fitted object and evaluation responses for a test-prediction object. Classification and survival values are unchanged. Standardization is off by default and does not apply to predictions.

`get.mv.error()`, `get.mv.error.block()`, and `get.mv.vimp()` divide by the variance directly, so a zero or missing variance can produce nonfinite values. The case-specific functions use a divisor of one when the variance is zero or `NA`.

**Missing results:** A NULL component means that the corresponding result is absent from obj. An NA entry is a missing value within an existing result.

- get.mv.error() returns NULL when no response has an error result. Otherwise, a response without an error is represented by NA in vector output or NULL in list output. The same list behavior applies to get.mv.error.block().
- get.mv.vimp(), get.mv.cserror(), and get.mv.csvimp() return NULL if the first response lacks the corresponding result, even when later responses have results. If the first response has a result, missing later results are NULL in list output.

**Constructing a multivariate formula:** get.mv.formula(ynames) returns Multivar(y1, y2, ...) ~ .. Responses may be continuous, factors, or a mixture; their types are determined from the data when the forest is fitted. The dot specifies the remaining data columns as predictors.

## Value

get.mv.predicted A numeric matrix with one row per observation and columns for response predictions, class probabilities, or event-specific predictions. A single-column result is also a matrix.

get.mv.error A named numeric vector by default, or a list named by response when pretty = FALSE. Entries contain the final error or error row. With block = TRUE, the result is a list of block-error sequences. Returns NULL when no response has an error result.

get.mv.error.block A list of block-error vectors or matrices named by response, or NULL when no response has block errors.

get.mv.vimp A numeric matrix by default, or a list of matrices named by response when pretty = FALSE. Returns NULL if the first response has no importance result.

get.mv.cserror Case-specific error values for one response, with their original vector or array dimensions, or a list named by response for multiple responses. Returns NULL for survival families or if the first response has no case-specific error result.

get.mv.csvimp A case-by-variable matrix for one response, or a list of these matrices named by response for multiple responses. Returns NULL for survival families or if the first response has no case-specific VIMP result.

get.mv.formula An R formula with the supplied response names on the left of ~ and a dot on the right.

## See Also

rfsrc, predict.rfsrc, vimp, subsample, classification.performance

## Examples

```
## ------------------------------------------------------------
## A basic multivariate analysis
## ------------------------------------------------------------
o <- rfsrc(cbind(Ozone, Temp) ~ ., data = na.omit(airquality))
print(head(get.mv.predicted(o)))
print(get.mv.error(o))
```

```
## ------------------------------------------------------------
## Select a response from the stored results
## ------------------------------------------------------------
pred.oob <- get.mv.predicted(o)
print(head(pred.oob[, "Temp", drop = FALSE]))
print(get.mv.error(o)["Temp"])
print(get.mv.error(o, standardize = TRUE))
print(head(get.mv.predicted(o, oob = FALSE)))

## ------------------------------------------------------------
## Formula construction, VIMP, and block errors
## ------------------------------------------------------------
f <- get.mv.formula(c("Ozone", "Temp"))
print(f)

o.vimp <- rfsrc(f, data = na.omit(airquality), ntree = 100,
                importance = "permute", block.size = 10)
print(get.mv.vimp(o.vimp))
print(get.mv.vimp(o.vimp, standardize = TRUE))
print(get.mv.vimp(o.vimp, pretty = FALSE)[["Temp"]])
print(head(get.mv.error.block(o.vimp)[["Temp"]]))

## Optional case-specific values are NULL when not saved in the object.
print(get.mv.cserror(o.vimp))
print(get.mv.csvimp(o.vimp))

## ------------------------------------------------------------
## Mixed outcomes: include class-specific entries
## ------------------------------------------------------------
f.mix <- get.mv.formula(c("Sepal.Length", "Species"))
mix <- rfsrc(f.mix, data = iris, ntree = 100,
             importance = "permute", block.size = 10)
print(colnames(get.mv.predicted(mix)))
print(get.mv.error(mix))
print(get.mv.error(mix, pretty = FALSE)[["Species"]])
print(get.mv.vimp(mix, pretty = FALSE)[["Species"]])

## ------------------------------------------------------------
## Extract predictions and errors for held-out observations
## ------------------------------------------------------------
dta <- na.omit(airquality)
set.seed(17)
train <- sample(seq_len(nrow(dta)), floor(.7 * nrow(dta)))
fit <- rfsrc(f, data = dta[train, ], ntree = 100)
p.test <- predict(fit, newdata = dta[-train, ])
print(head(get.mv.predicted(p.test, oob = FALSE)))
print(get.mv.error(p.test))
```

---

`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. The examples fit a multivariate regression forest with lipid expression profiles as the responses and gene expression, diet, and genotype as predictors.

## Details

The examples use `get.mv.formula` to construct the response formula, and `get.mv.error` and `get.mv.vimp` to extract response-specific performance errors and variable importance (VIMP). For these regression responses, `standardize = TRUE` divides each error or importance value by the variance of the corresponding response. See Extracting Multivariate Values (`multivariate.values`) for the helper interfaces and returned formats.

## 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.

## See Also

`rfsrc`, `predict.rfsrc`, Extracting Multivariate Values (`multivariate.values`)

## 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 *Compute Partial Dependence Values*

---

## Description

Compute predictions for partial dependence analysis using a fitted forest. This advanced interface is the workhorse called by `plot.variable(..., partial = TRUE)`, which provides automatic grid selection and plotting. Use `partial()` directly for custom predictor grids, additional fixed predictors, several survival evaluation times, or observation-level predictions.

`partial()` returns predictions for each training observation. `get.partial.plot.data()` extracts their means or individual values for custom plots. Supported outcomes include regression, classification, multivariate and mixed outcomes, right-censored survival, and competing risks. See `plot.variable.rfsrc` for the simpler plotting interface.

## 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, ...)

get.partial.plot.data(o, target, m.target = NULL, granule = FALSE)
```

## Arguments

| | |
|---|---|
| `object` | An object of class `(rfsrc, grow)` with a saved forest, or an object of class `(rfsrc, forest)`. The forest must contain its training data. Anonymous forests are not supported. |
| `oob` | Logical. If `TRUE` (default), predict each observation using trees that excluded it during training. If `FALSE`, use the non-OOB ensemble. |
| `partial.type` | Prediction summary for survival outcomes. For right-censored survival, choose `"mort"`, `"rel.freq"`, `"surv"`, or `"chf"`. For competing risks, choose `"years.lost"`, `"cif"`, or `"chf"`. Specify one type for these families; see Details for their meanings. Ignored for regression and classification. |
| `partial.xvar` | Name of the predictor to vary. |
| `partial.values` | Nonempty vector of finite numeric values for `partial.xvar`. Values are sorted and duplicates removed before calculation. For a factor, supply integer codes from its training level order; see Details. |
| `partial.xvar2` | Optional character vector naming additional predictors to hold fixed. Names must be distinct and different from `partial.xvar`. |
| `partial.values2` | Numeric vector giving one finite fixed value per predictor in `partial.xvar2`, in the same order. Factors use their training level codes. Each value applies to all observations throughout the primary predictor grid. |
| `partial.time` | Nonempty vector of finite evaluation times for `"surv"`, `"cif"`, or `"chf"`. Times must be at least the first value in the forest's `time.interest`. See Details for matching requests to the time grid. Ignored for `"mort"`, `"rel.freq"`, and `"years.lost"`. |
| `get.tree` | Optional vector of tree indices. By default all trees are considered, with OOB eligibility applied when `oob = TRUE`. |

| | |
|---|---|
| seed | Negative integer specifying the random-number seed. |
| do.trace | Number of seconds between progress updates; FALSE suppresses updates. |
| ... | Additional internal prediction options. Supply graphical arguments to the subsequent plotting call. |
| o | An object returned by partial(). |
| target | For classification, a class label or its numeric position in the selected response's levels. For competing risks, an event code from o$event.info$event.type. The first class or event is used when omitted. Ignored for regression and right-censored survival. |
| m.target | Response name to extract from a multivariate or mixed partial object. The default is the first response in o$yvar.names. |
| granule | Logical used by get.partial.plot.data(). If FALSE (default), return the mean prediction at each grid value. If TRUE, return individual predictions with a grid value for each observation. This differs from the numeric granule argument of plot.variable(). |

## Details

**Partial predictions and averaging:** At each value in partial.values, partial() sets partial.xvar to that value for every training observation and predicts with the fitted forest. The other predictors stay at each observation's training values, except for any additional predictors fixed by partial.xvar2. No trees are refitted.

For a predictor $X_j$, with no additional predictors fixed, write the prediction for observation $i$ at value $z$ as $P_i(z) = \widehat{f}(z, X_{i,-j})$. Averaging these predictions gives the partial dependence at $z$: the model's mean prediction when $X_j$ is set to $z$, averaged over the training values of the other predictors.

partial() returns the individual $P_i(z)$ values; get.partial.plot.data() computes their means. With oob = TRUE, each $P_i(z)$ uses the trees for which observation $i$ was OOB. This changes the contributing trees, while the training observations still provide the values over which the mean is calculated.

**Choosing predictor values:** Choose a grid for the range of interest, such as equally spaced values or selected training quantiles. Each grid point requires a prediction for every observation, so fewer points reduce computation and output size.

For factors, use the integer codes corresponding to the training levels. For example, levels c("low", "middle", "high") have codes 1, 2, and 3. Use match(labels, levels(training.factor)) to obtain codes for selected labels. The extraction helper returns numeric codes, which can be labeled in the plotting call.

**Fixing additional predictors:** Use partial.xvar2 and partial.values2 to hold additional predictors at specified values for every observation. For example, vary wind over a grid while fixing temperature at one value. The resulting curve averages predictions over the remaining predictors.

To compare curves at several temperatures, repeat the call with a different fixed temperature each time. This also provides the values for a two-dimensional display: one predictor varies along the grid, and the other changes between calls.

**Selecting a response:** `get.partial.plot.data()` extracts predicted responses for regression and the selected class probability for classification. For multivariate and mixed outcomes, `partial()` computes all responses in one call. Use `m.target` in the extraction helper to choose a response, and `target` to choose a class when that response is a factor. Several responses or classes can be extracted from the same partial object without recalculating predictions.

For competing risks, `target` selects an event code from `o$event.info$event.type`. This differs from `plot.variable()`, where `target` selects the event's position in that vector.

**Survival summaries and evaluation times:** For right-censored survival, `"mort"` returns mortality, `"surv"` returns survival probabilities, and `"chf"` returns cumulative hazards. In `partial()` and its extraction helper, `"rel.freq"` returns the same unnormalized mortality as `"mort"`.

For competing risks, `"years.lost"` returns expected years lost, `"cif"` returns cumulative incidence, and `"chf"` returns cause-specific cumulative hazard. The extraction helper selects the event of interest.

Supply `partial.time` for `"surv"`, `"cif"`, or `"chf"`. Predictions use the forest's `time.interest` grid. Each request uses the last grid time less than or equal to it; a request after the final grid time uses that final time. Using values from `time.interest` gives exact matches. The returned `partial.time` contains the requested times, which can differ from the grid times used for prediction.

Mortality and years lost are scalar summaries and do not use `partial.time`.

**Extracting results for custom plots:** Pass the partial object to `get.partial.plot.data()` to obtain predictor values `x` and corresponding predictions `yhat`. The `granule` argument controls the summary:

`granule = FALSE` Return the mean at each grid value. Only finite predictions contribute to the mean; if none are finite, the mean is `NA`.

`granule = TRUE` Return the individual predictions, grouped by grid value and then in training-observation order. Each grid value is repeated once per observation. Nonfinite predictions become `NA`.

For time-dependent survival summaries, `yhat` has one column per requested time. The individual values are on the original prediction scale. Unlike the boxplot values from `plot.variable(..., partial = TRUE)`, they are not shrunk toward their mean.

For automatic plots and numerical summaries, use `plot.variable(..., partial = TRUE)`. Its `partial.summary` component contains means, standard deviations, row-averaging standard errors, and finite prediction counts. See `plot.variable.rfsrc` for the interpretation of its standard errors and display bands.

## Value

`partial()` returns an object of class `(rfsrc, partial, family)`. It includes `call`, `family`, `partial.values` (the grid used), `partial.time` (the requested times), `yvar.names`, and `event.info`, together with the prediction components below. Let $n$ be the number of training observations, $K$ the number of predictor grid values, $T$ the number of requested times, and $J$ the number of events.

`regrOutput` Named list with one $n \times K$ matrix per regression response. Rows correspond to observations and columns to predictor grid values.

`classOutput` Named list with one $n \times (1 + C) \times K$ array per classification response, where $C$ is its number of classes. The second dimension is named `"all"` followed by the class labels; the class-labeled columns contain class probabilities.

`survOutput` For right-censored mortality (`"mort"` or `"rel.freq"`), an $n \times K$ matrix. For survival or cumulative hazard, an $n \times T \times K$ array. For competing-risk years lost, an $n \times J \times K$ array. For competing-risk cumulative incidence or cumulative hazard, an $n \times T \times J \times K$ array.

`get.partial.plot.data()` returns a list with:

`x` Predictor grid values. With `granule = TRUE`, each value is repeated $n$ times to match the individual predictions.

`yhat` Mean predictions when `granule = FALSE`, or individual predictions when `granule = TRUE`. Scalar summaries give a vector of length $K$ or $nK$, respectively. Time-dependent summaries give a matrix with $K$ or $nK$ rows and $T$ columns.

`partial.time` The evaluation times supplied to `partial()`.

## 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: explicit grid and raw observation-level output.
##
## ------------------------------------------------------------
set.seed(19)
dta <- na.omit(airquality)
fit <- rfsrc(Ozone ~ ., data = dta, ntree = 100)
wind <- unique(as.numeric(quantile(fit$xvar$Wind, seq(0, 1, length.out = 11))))
p <- partial(fit, partial.xvar = "Wind", partial.values = wind)
pd <- get.partial.plot.data(p)
print(head(data.frame(Wind = pd$x, partial = pd$yhat)))
plot(pd$x, pd$yhat, type = "b", xlab = "Wind", ylab = "Partial prediction")
raw <- get.partial.plot.data(p, granule = TRUE)
boxplot(raw$yhat ~ raw$x, xlab = "Wind", ylab = "Individual partial predictions")

## ------------------------------------------------------------
##
## Vary wind while fixing temperature at three values.
##
## ------------------------------------------------------------
```

```
temperature <- as.numeric(quantile(fit$xvar$Temp, c(.25, .5, .75)))
slices <- lapply(temperature, function(tt) {
  pp <- partial(fit, partial.xvar = "Wind", partial.values = wind,
                partial.xvar2 = "Temp", partial.values2 = tt)
  get.partial.plot.data(pp)$yhat
})
matplot(wind, do.call(cbind, slices), type = "l", lty = 1,
        xlab = "Wind", ylab = "Partial prediction")
legend("topright", legend = paste("Temp =", temperature),
       col = seq_along(temperature), lty = 1)

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

n <- 200
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: extract a named class from a univariate response.
##
## ------------------------------------------------------------
iris.fit <- rfsrc(Species ~ ., data = iris, ntree = 100)
```

```
sp <- partial(iris.fit, partial.xvar = "Sepal.Length",
              partial.values = seq(4.5, 7.5, length.out = 11))
sp.data <- get.partial.plot.data(sp, target = "versicolor")
plot(sp.data$x, sp.data$yhat, type = "b", ylim = c(0, 1),
     xlab = "Sepal length", ylab = "Partial probability of versicolor")

## ------------------------------------------------------------
##
## Multivariate regression: one partial call, two extracted responses.
##
## ------------------------------------------------------------
mv <- rfsrc(cbind(Ozone, Temp) ~ ., data = dta, ntree = 100)
mp <- partial(mv, partial.xvar = "Wind", partial.values = wind)
print(names(mp$regrOutput))
mp.ozone <- get.partial.plot.data(mp, m.target = "Ozone")
mp.temp <- get.partial.plot.data(mp, m.target = "Temp")
print(head(data.frame(Wind = mp.temp$x, Temp = mp.temp$yhat)))

## ------------------------------------------------------------
##
## Mixed outcomes: extract a named class from the selected response.
##
## ------------------------------------------------------------
car.mix <- mtcars
car.mix$am <- factor(car.mix$am)
mix <- rfsrc(Multivar(mpg, am) ~ ., data = car.mix, ntree = 100)
mix.p <- partial(mix, partial.xvar = "wt", partial.values = c(2, 3, 4))
mix.am <- get.partial.plot.data(mix.p, m.target = "am", target = "1")
print(data.frame(wt = mix.am$x, probability = mix.am$yhat))

## ------------------------------------------------------------
##
## Factors: keep the training level codes when choosing labels.
##
## ------------------------------------------------------------
car.data <- mtcars
car.data$cyl <- factor(car.data$cyl)
car.fit <- rfsrc(mpg ~ ., data = car.data, ntree = 100)
lev <- levels(car.fit$xvar$cyl)
code <- match(lev, levels(car.fit$xvar$cyl))
cp <- partial(car.fit, partial.xvar = "cyl", partial.values = code)
cd <- get.partial.plot.data(cp, granule = TRUE)
boxplot(cd$yhat ~ factor(cd$x, levels = code, labels = lev),
        xlab = "Cylinders", ylab = "Individual partial predictions")

## ------------------------------------------------------------
##
## Right-censored survival: several stored evaluation times.
##
## ------------------------------------------------------------
data(veteran, package = "randomForestSRC")
vfit <- rfsrc(Surv(time, status) ~ ., data = veteran, ntree = 100)
grid <- vfit$time.interest
```

```
when <- grid[unique(round(seq(1, length(grid), length.out = 3)))]
karno <- unique(as.numeric(quantile(vfit$xvar$karno)))
vp <- partial(vfit, partial.type = "surv", partial.xvar = "karno",
              partial.values = karno, partial.time = when)
vd <- get.partial.plot.data(vp)
print(vd$partial.time)
matplot(vd$x, vd$yhat, type = "l", lty = 1,
        xlab = "Karnofsky score", ylab = "Partial survival probability")
legend("bottomright", legend = paste("Time =", vd$partial.time),
       col = seq_along(vd$partial.time), lty = 1)

## ------------------------------------------------------------
##
## Survival curves at selected quantiles of Karnofsky score.
## Reuse vfit from the preceding right-censored survival example.
##
## ------------------------------------------------------------
karno.probs <- c(.10, .50, .90)
karno.values <- quantile(vfit$xvar$karno, probs = karno.probs,
                         na.rm = TRUE)
print(karno.values)

## Set Karnofsky score to each requested value for every reference row.
## Keep the other predictors at their observed values and use OOB trees.
surv.partial <- partial(vfit,
                         oob = TRUE,
                         partial.type = "surv",
                         partial.xvar = "karno",
                         partial.values = as.numeric(karno.values),
                         partial.time = vfit$time.interest)
surv.data <- get.partial.plot.data(surv.partial)

## yhat has one row per distinct Karnofsky value and one column per time.
## Transpose it so that each plotted column is a survival curve over time.
## Each curve averages over the same reference rows.
curve.id <- seq_along(surv.data$x)
matplot(surv.data$partial.time, t(surv.data$yhat),
        type = "s", col = curve.id, lty = curve.id, lwd = 2,
        ylim = c(0, 1), xlab = "Time", ylab = "Survival probability",
        main = "Partial survival curves by Karnofsky score")

## Use the returned grid for labels: partial() removes tied quantile values.
legend("topright",
       legend = paste("Karnofsky =", format(surv.data$x, trim = TRUE)),
       col = curve.id, lty = curve.id, lwd = 2, bty = "n")

## ------------------------------------------------------------
##
## Competing risks: select an event code from the saved event map.
##
## ------------------------------------------------------------
data(follic, package = "randomForestSRC")
crfit <- rfsrc(Surv(time, status) ~ ., data = follic, ntree = 100, nsplit = 3)
```

```
age <- unique(as.numeric(quantile(crfit$xvar$age)))
crp <- partial(crfit, partial.type = "years.lost",
               partial.xvar = "age", partial.values = age)
event <- crp$event.info$event.type[1]
crd <- get.partial.plot.data(crp, target = event)
plot(crd$x, crd$yhat, type = "b", xlab = "Age", ylab = "Partial years lost")
```

---

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 Conditional Quantiles and CRPS Diagnostics*

---

## Description

Plots observed responses against their predicted conditional quantiles. An optional inset compares the forest's continuous ranked probability score (CRPS) with a reference that uses the training-response distribution without predictors. Numerical pinball losses can also be displayed for selected quantile levels.

## Usage

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

## Arguments

| | |
|---|---|
| `x` | A training or prediction object returned by `quantreg`, containing observed responses for the observations to be plotted. For new-data diagnostics, include response columns in the `quantreg()` prediction call. |
| `prbL`, `prbU` | Lower and upper quantile levels, each a single probability in $(0, 1)$, with `prbL` < `prbU`. The middle level is 0.5 when it lies strictly between them; otherwise it is their midpoint. Each level is matched to the nearest stored probability by `get.quantile()`. Include the desired levels in `prob` when calling `quantreg()` for exact matching. |

`m.target` Name of one continuous response to display. The default is the first response with quantile results. For multivariate or mixed outcomes, this selects results from `x` without calculating new predictions.

`crps` If `TRUE`, display the standardized, finite-grid CRPS curve from `get.quantile.crps()` in an inset.

`subset` Observations to plot and score. Supply positive integer row indices or a logical vector with one nonmissing entry per row of the quantile output. `NULL` uses all rows. Indices refer to the returned results after training or prediction preprocessing; repeated indices include an observation more than once.

`xlab, ylab` Main-panel axis labels. Defaults are the selected response name and `"Target Quantiles"`.

`...` Named graphical arguments for the main panel, passed to `plot.default`. Use `xlim` and `ylim` for axis limits, `main` for the title, and `cex.axis`, `las`, and related options for the axes. The arguments `pch`, `col`, and `cex` style the middle-quantile points. The function sets the plotted coordinates and initial panel `type`. These settings do not style the CRPS inset.

`inset.args` Named list of graphical arguments for the CRPS inset, passed to `plot.default`. Use `xlim` and `ylim` for inset limits, and `col`, `lty`, and `lwd` for the forest curve. Axis options such as `cex.axis`, `las`, and `tck` also apply. `axes = FALSE` hides both inset axes; `yaxt = "n"` hides its right-hand axis. The default `list()` uses the inset's own settings. Ignored when `crps = FALSE`. Supply the full argument name.

`quantreg.tau` Quantile levels for numerical pinball-loss labels, as a vector of finite probabilities strictly between zero and one. For each level, the plot displays the mean loss for the selected response and subset in the lower-right corner of the main panel. The default `NULL` omits these labels, even when the session option `quantreg.tau` is set. The plotted quantiles and CRPS inset are unchanged. Supply the full argument name.

`crps.null` If `TRUE`, include the null reference in the CRPS inset as a dashed black curve, with a legend identifying `Forest` and `Null`. Set to `FALSE` for the forest curve alone. Ignored when `crps = FALSE`. Supply the full argument name.

## Details

The plot compares observed responses with the predicted response distributions. The main panel shows selected quantiles for individual observations; the CRPS inset summarizes distributional prediction error across observations. Both displays use the response selected by `m.target` and the observations selected by `subset`.

**Quantile panel:** Each observation is placed at its observed response on the horizontal axis. A point marks its predicted middle quantile, and a vertical segment with endpoint marks spans its lower and upper quantiles. The dashed diagonal marks equality between observed and predicted values. The default levels request the median and an interval from the 25th to the 75th percentile.

Horizontal jitter separates overlapping observations. All scores use the original response values, not these display positions. Observations with an unavailable response or requested quantile are omitted from the main panel with a warning.

**Pinball-loss labels:** Supply `quantreg.tau = c(.1, .5, .9)` to display mean pinball losses for the 10th, 50th, and 90th percentile predictions as text in the lower-right corner of the main panel. Smaller losses indicate better quantile predictions. These levels can differ from the quantiles drawn in the panel.

The values are calculated by `get.pinball.error(x, tau = quantreg.tau, subset = subset, m.target = m.target)` for the selected response. Each loss averages over the selected observations with finite responses and quantile predictions at that level. Plotting omits these labels unless levels are supplied in the call; the session reporting option alone does not add them.

**CRPS inset:** The inset evaluates the predicted CDF across response thresholds. At each threshold, it averages the squared difference between the predicted CDF and the indicator that the observed response is at or below that threshold. It integrates these errors by the trapezoidal rule from the first reporting-grid value to each successive value, then divides by the integration width.

The horizontal axis is the response threshold; the right-hand vertical axis is standardized CRPS. Lower values indicate less error over the corresponding integration interval. This is the finite-grid curve returned by `get.quantile.crps()`, so the grid's range and resolution affect the scores. The first value is unavailable because its integration width is zero. An entirely unavailable curve is omitted with a warning.

**Null reference:** The null curve provides a comparison with predictions based on the training-response distribution alone. It uses the same empirical CDF for every evaluated observation:

$$F_0(t) = \frac{1}{n_0} \sum_{j=1}^{n_0} I(Y_j^{\text{train}} \leq t),$$

where $n_0$ is the number of finite training responses for the selected outcome. Every training observation contributes, so repeated response values contribute according to their frequencies. Test responses are used only to evaluate the reference, not to construct it.

Forest and null curves use the same response grid, subset, integration rule, and standardization. At each threshold, both exclude observations with an unavailable response or forest CDF prediction. The null distribution itself always uses all finite training responses, regardless of `subset`. For training plots, it includes the evaluated responses and is an in-sample benchmark rather than an OOB or leave-one-out estimate.

Training responses come from `x$forest$yvar`, or `x$yvar` for a training object without responses saved in its forest. If they are unavailable, the null curve is omitted with a warning.

**Customizing the display:** Use `...` for the main panel and `inset.args` for the inset. For example, `inset.args = list(ylim = c(0, 0.3), lwd = 1.5)` changes the inset's vertical limits and forest-curve line width. Default inset limits cover both curves.

Axis limits change only the displayed region; they do not change the observations scored or the CRPS integration limits. After the inset is drawn, graphics settings return to the main panel so that further drawing applies there. Successive calls can be used in a multi-panel layout.

## Value

Used for its graphical side effect. Invisibly returns `NULL`.

## Author(s)

Hemant Ishwaran and Udaya B. Kogalur

## See Also

quantreg, get.quantile, get.quantile.crps, get.pinball.error, plot.default

## Examples

```
## Univariate quantiles, with forest and null CRPS curves.
library(randomForestSRC)
set.seed(19)
dta <- na.omit(airquality)
prob <- c(.25, .50, .75)
q <- quantreg(Temp ~ ., data = dta, prob = prob, ntree = 100)
plot.quantreg(q)

## Add pinball losses for this plot only; the interval and inset are unchanged.
plot.quantreg(q, quantreg.tau = c(.2, .5, .8))
print(get.pinball.error(q, tau = c(.2, .5, .8)))

## Customize the main panel directly, and the inset separately.
plot.quantreg(q, main = "Temperature",
              xlim = c(50, 100), ylim = c(45, 110), pch = 19,
              inset.args = list(xlim = c(60, 95), ylim = c(0, .35),
                                lwd = 1.5))

## Keep the inset, but suppress its null reference.
plot.quantreg(q, crps.null = FALSE)

## Multivariate forest: select each response for plotting only.
mv <- quantreg(cbind(Ozone, Temp) ~ ., data = dta,
               splitrule = "mahalanobis", prob = prob, ntree = 100)
print(names(get.quantile(mv, pretty = FALSE)))
op <- par(mfrow = c(1, 2))
plot.quantreg(mv, m.target = "Ozone")
plot.quantreg(mv, m.target = "Temp", main = "Temperature",
              xlim = c(50, 100), ylim = c(45, 110),
              inset.args = list(ylim = c(0, .35)))
par(op)
```

---

# 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 Subsampling Confidence Intervals for Variable Importance*

---

## Description

Plot confidence intervals for variable importance (VIMP) from a `subsample` analysis. Compare predictors and assess whether their importance is positive under the chosen interval method. Joint VIMP and prediction error can also be displayed when included in the resampling analysis.

## 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 returned by `subsample.rfsrc`, including double-bootstrap output. |
| `alpha` | Significance level, strictly between zero and one. Intervals have nominal confidence level $1 - \alpha$. The default `.01` gives 99 percent intervals. Use `alpha = .05` to match the default for extraction and printing. |
| `xvar.names` | Names of rows to display: predictor names, or `"joint"` and `"err"` when those statistics were requested. If omitted, all rows are considered, subject to `pmax`. |
| `standardize` | For regression, divide estimates by the selected response's variance in the full training data. The same divisor applies to all replicates, VIMP, and prediction error. Other families are unchanged. Set `FALSE` for the original scale. |
| `normal` | Use normal-approximation intervals when `TRUE` and nonparametric intervals when `FALSE`. For double-bootstrap objects, these are bootstrap normal and percentile intervals. |
| `jknife` | Use the delete-$d$ jackknife standard error for normal intervals. The default uses the subsampling standard error. Ignored when `normal = FALSE` or for double-bootstrap objects. |

| | |
|---|---|
| target | For classification, use 0 (the default) for the overall statistic, or a class label or position 1, 2, and so on for a class-specific statistic. For competing risks, use an event position from 1 to J, where J is the number of event types; the first event is the default. Ignored for regression and right-censored survival. |
| m.target | Name of one response in a multivariate or mixed-outcome forest. The default selects the first regression response with results, or the first classification response with results if none is available for regression. Use target to select a class or event within that response. |
| pmax | Maximum number of rows to display. Supply a positive integer or Inf for all rows. When this limit is exceeded, the rows with the largest reference estimates are displayed in decreasing order, even if sorted = FALSE. |
| main | Main plot title. |
| sorted | Order rows by decreasing reference estimate. For VIMP, this places the most important predictors first. See pmax. |
| show.plots | Draw the plot. Set FALSE to return its numerical summaries without drawing. |
| ... | Named graphical arguments, such as xlim, ylim, xlab, ylab, boxfill, border, whisklty, and whisklwd. A named pars list is also accepted; direct arguments take precedence. See Details. |

## Details

After fitting smp <- subsample(obj), use plot.subsample(smp) to compare VIMP estimates and their uncertainty. Each row represents a predictor or a requested joint-VIMP or prediction-error statistic. Changing the confidence level, interval method, or selected results uses the saved replicates without fitting more forests.

**Interpreting the display:** The default plot is horizontal, with VIMP or prediction error on the horizontal axis. Whiskers mark the confidence limits; the box spans the lower and upper quartiles of the interval summary. For subsampling, the center line marks the full-data estimate for normal intervals and the interval median for nonparametric intervals. For double-bootstrap output, it marks the bootstrap mean.

Boxes are red when the lower confidence limit exceeds zero and blue otherwise. For VIMP, red identifies positive importance under the chosen interval method. An err row summarizes prediction error, so its color is not a variable-selection result. Intervals are calculated separately, without a multiple-testing adjustment. These boxes describe uncertainty in the estimates, not variation in the observed predictor values.

**Choosing the interval method:** Subsampling provides three displays:

**Normal subsampling (default).** Use normal = TRUE, jknife = FALSE. The standard error measures variation about the subsample mean.

**Normal jackknife.** Use normal = TRUE, jknife = TRUE. The standard error uses deviations from the full-data estimate, including its difference from the subsample mean.

**Nonparametric subsampling.** Use normal = FALSE. Limits use empirical quantiles of centered and scaled subsample estimates.

The jackknife standard error need not be larger than the subsampling standard error. See subsample.rfsrc for the formulas and finite-replicate scaling.

For double-bootstrap objects, `normal = TRUE` uses normal intervals centered at the bootstrap mean; `normal = FALSE` uses percentile intervals. `jknife` has no effect.

Use the same `alpha` when comparing plots with printed or extracted results. Plotting defaults to `.01`; printing and extraction default to `.05`.

**Selecting predictors and outcomes:** Use `xvar.names` to choose rows, `m.target` to choose a response, and `target` to choose its class or event statistic. For example, `m.target = "Ozone"` selects results for the response named `Ozone`, not a predictor to display.

Competing-risk indexing differs from extraction: the first event uses `target = 1` here but `target = 0` in `extract.subsample()`.

**Customizing the plot:** `xlim` and `ylim` control the horizontal and vertical axes as displayed. Use `horizontal = FALSE` to place VIMP or prediction error on the vertical axis.

Set `boxfill` (or `col`) for box colors, `border` for borders, and `whisklty` and `whisklwd` for the confidence-limit lines. Outliers are hidden by default (`outline = FALSE`).

`cex.axis`, `col.axis`, and `las` control axis labels. Use `xaxt = "n"` or `yaxt = "n"` to suppress one axis, or `axes = FALSE` for both. A single `ylab` string sets the vertical-axis title. For custom row labels, supply `names` with one label per selected row before sorting and limiting the display to `pmax` rows. `at` sets positions for the final displayed intervals.

**Obtaining numerical results:** `plot.subsample(smp, show.plots = FALSE)` returns the selected interval matrix and labels without drawing. `extract.subsample(smp, raw = TRUE)` returns all interval methods and the replicate estimates. Both use the requested `standardize` setting.

## Value

Invisibly returns a list with the following plotting components:

| | |
|---|---|
| `stats` | Five-row interval matrix, with one column per displayed statistic. Rows contain the lower confidence limit, lower quartile, center, upper quartile, and upper confidence limit. |
| `names` | Labels corresponding to the columns of `stats`. |

Other components support the boxplot drawing and are not additional confidence intervals. The same list is returned with `show.plots = FALSE`.

## 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

subsample.rfsrc, extract.subsample, bxp

## Examples

```
## Small settings are for illustration; increase B for final inference.
set.seed(19)
dta <- na.omit(airquality)
o <- rfsrc(Ozone ~ ., data = dta, ntree = 100,
           importance = "permute", block.size = 1)
smp <- subsample(o, B = 25, verbose = FALSE)

## Three interval displays, all at the same confidence level.
plot.subsample(smp, alpha = .05, main = "Subsampling normal intervals")
plot.subsample(smp, alpha = .05, jknife = TRUE,
                main = "Jackknife normal intervals")
plot.subsample(smp, alpha = .05, normal = FALSE,
                main = "Nonparametric intervals")

## Restrict the display and customize the main axes and whiskers.
plot.subsample(smp, alpha = .05,
                xvar.names = c("Solar.R", "Wind", "Temp"),
                xlim = c(-.1, .8), las = 2, cex.axis = .75,
                whisklty = 1, whisklwd = 1.5)
plot.data <- plot.subsample(smp, alpha = .05, show.plots = FALSE)
print(plot.data)

## Multivariate regression: one subsample bank, two response displays.
mv <- rfsrc(cbind(Ozone, Temp) ~ ., data = dta, ntree = 100,
            importance = "permute", block.size = 1)
mv.smp <- subsample(mv, B = 25, verbose = FALSE)
plot.subsample(mv.smp, m.target = "Ozone", alpha = .05,
                main = "Ozone")
plot.subsample(mv.smp, m.target = "Temp", alpha = .05,
                main = "Temperature")
print(extract.subsample(mv.smp, m.target = "Temp", alpha = .05)$var.sel.Z)
```

---

# plot.survival.rfsrc *Survival Prediction Performance and Diagnostic Plots*

---

## Description

Display survival predictions and assess their accuracy using Brier scores, integrated Brier scores, and time-dependent AUC. Compare the forest Brier score with a predictor-free Kaplan–Meier reference, or obtain numerical scores for further analysis.

The separate get.cindex helper calculates concordance error from observed outcomes and risk predictions for right-censored survival or competing risks.

## Usage

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

get.brier.survival(o, subset = NULL,
  cens.model = c("km", "rfsrc"), papply = lapply,
  times = NULL, conf.int = FALSE, keep.matrix = TRUE)

get.auct.survival(o, subset = NULL,
  cens.model = c("km", "rfsrc"), papply = lapply,
  times = NULL, conf.int = FALSE)

plotBrierAUC(x, subset = NULL,
  cens.model = c("km", "rfsrc"), papply = lapply,
  times = NULL, conf.int = TRUE,
  plots = c("brier", "auct"), show.plots = TRUE,
  brier.null = TRUE, ...)

get.cindex(time, censoring, predicted, weight, fast, do.trace = FALSE)
```

## Arguments

| | |
|---|---|
| x, o | A fitted or prediction object of class (rfsrc, grow) or (rfsrc, predict). The functions get.brier.survival, get.auct.survival, and plotBrierAUC also accept a saved forest of class (rfsrc, forest). |
| show.plots | If TRUE, draw the requested plots. Use FALSE to return the results without drawing. |
| subset | Observations to include, specified by positive row indices or a logical vector with one entry per predicted observation. By default, all observations are included. |
| collapse | For plot.survival, show the mean survival curve without the individual curves? The marginal Nelson–Aalen curve, when available, is also shown. |
| cens.model | Censoring model for inverse-probability-of-censoring weighting (IPCW), estimated from the full training data:<br>"km": A Kaplan–Meier estimate common to all observations. This is the default.<br>"rfsrc": A random survival forest estimate that allows the censoring distribution to depend on the predictors. When censoring is present, the object must contain both training and evaluation predictors. See **Note**. |
| papply | Function used in place of lapply for repeated calculations. |
| times | Optional numeric vector of times at which to calculate Brier score or AUC. Survival predictions are evaluated as right-continuous step functions on this grid. By default, use the forest's time grid. |

`conf.int` Pointwise confidence intervals: `FALSE` omits intervals, `TRUE` requests 95 percent intervals, and a number strictly between zero and one specifies another confidence level. Defaults to `TRUE` for `plotBrierAUC` and `FALSE` for the numerical helpers.

`keep.matrix` For `get.brier.survival`, include the subject-by-time matrix of IPCW Brier contributions in the result? Use `FALSE` when only the summary scores are needed.

`plots` For `plotBrierAUC`, select `"brier"`, `"auct"`, or both. The name `"auc"` is an alias for `"auct"`.

`brier.null` For `plotBrierAUC`, compare the forest Brier score with a predictor-free Kaplan–Meier estimate from the full training data? The default is `TRUE`. The reference is also returned with `show.plots=FALSE`. Ignored when only AUC is requested.

`...` Further graphical arguments. For `plotBrierAUC`, `xlim`, `ylim`, titles, and axis options apply to each requested panel. The arguments `col`, `lwd`, `lty`, and `type` control the forest curve; `band.col` and `band.border` control its confidence band. Use `null.col`, `null.lwd`, and `null.lty` for the Brier reference (defaults: black, width 2, dashed). The Forest/Null legend is placed at `legend.pos="topleft"` by default; set `legend.pos` to another position or `FALSE` to omit it.

`time` For `get.cindex`, observed follow-up times, in the same order as `censoring` and the prediction rows.

`censoring` For `get.cindex`, numeric status codes: zero for censoring and one for an event in right-censored survival. For competing risks, use event codes `1`, ..., `J`.

`predicted` For `get.cindex`, risk predictions, with larger values representing higher risk. For right-censored survival, supply a vector or one-column matrix or data frame. For competing risks, supply a matrix or data frame with column `j` for event code `j`, or a list of event-specific prediction vectors. The first `J` columns or list entries are used.

`weight` For `get.cindex`, optional precomputed concordance weights, one per observation. Omit for unweighted concordance. This helper does not estimate censoring weights.

`fast` For `get.cindex`, optional concordance-algorithm selector. Usually left unspecified so that the underlying library chooses the algorithm. Ignored for weighted competing-risk concordance.

`do.trace` For `get.cindex`, display diagnostic output from the underlying concordance calculation?

## Details

These functions assess survival predictions over follow-up. Brier score measures agreement between predicted survival probabilities and observed outcomes. Time-dependent AUC measures how well the predictions distinguish individuals with an event by a given time from those still event-free.

Use `plot.survival` for a four-panel diagnostic display, `plotBrierAUC` for performance curves with confidence bands, or `get.brier.survival` and `get.auct.survival` for numerical results and custom plots.

These plots and scores support right-censored survival. For competing-risk plots, use `plot.competing.risk.rfsrc`. The separate `get.cindex` helper supports both settings.

**Diagnostic panels:** `plot.survival` draws two rows of panels, from left to right:

1. **Survival curves.** Individual forest predictions, their mean across the selected individuals (thick red line), and the marginal Nelson–Aalen survival estimate from their outcomes (thick green line).
2. **Brier score.** Prediction error over time for four groups defined by the 0–25, 25–50, 50–75, and 75–100 percentile ranges of predicted mortality. The red curve is the overall Brier score.
3. **Integrated Brier score.** Cumulative average Brier error from the first plotting time to each subsequent time, labeled CRPS. The groups and red overall curve match the Brier panel.
4. **Mortality versus observed time.** Predicted mortality plotted against observed follow-up time, with blue points for events and black points for censored observations. Mortality (Ishwaran et al., 2008) expresses risk on the scale of the number of events. For example, a value of 100 means that an average of 100 events would be expected if all individuals had the same covariate values.

**Training and test evaluation:** For a grow object, the functions use out-of-bag (OOB) survival and mortality predictions when both are available; otherwise, they use full-ensemble predictions. For a prediction object, they use its test predictions and corresponding test outcomes.

The censoring model always uses the full training data, including when scoring a test set. Training outcomes must therefore be available. The argument `subset` selects the observations to evaluate, not the data used to estimate censoring.

If no evaluation outcomes are available, `plot.survival` displays only predicted survival curves and their mean. `get.brier.survival`, `get.auct.survival`, and `plotBrierAUC` require outcomes for the observations being evaluated.

**Brier score and integrated error:** The Brier score uses squared prediction errors adjusted for censoring by IPCW (Graf et al., 1999; Gerds and Schumacher, 2006). Smaller scores indicate better predictions.

Let $T_i$ be the observed follow-up time, $\Delta_i$ the event status (zero for censoring), $\widehat{S}_i(t)$ the predicted survival probability, and $\widehat{G}(t)$ the estimated censoring survival probability. The contribution from individual $i$ is

$$L_i(t) = \frac{I(T_i \le t, \Delta_i > 0)}{\widehat{G}(T_i-)} \widehat{S}_i(t)^2 + \frac{I(T_i > t)}{\widehat{G}(t)} \{1 - \widehat{S}_i(t)\}^2,$$

For `cens.model="rfsrc"`, $\widehat{G}$ also depends on the individual's predictors. The Brier score averages the finite contributions at each time. Individuals censored at or before $t$ contribute zero and are included in this average.

Integrated Brier scores use trapezoidal integration. The CRPS panel in `plot.survival` divides the integral through each time by the elapsed time since the first grid point. Its first value is `NA` because no time has elapsed.

The scalar summaries from `get.brier.survival` are `crps`, the integral over the evaluation grid, and `crps.std`, that integral divided by the largest evaluation time. This differs from the elapsed-time normalization in the CRPS panel.

**Time-dependent AUC:** The cumulative/dynamic AUC compares individuals with an observed event by time $t$ (cases) with those followed beyond $t$ (controls). Individuals censored at or before $t$ are excluded from this comparison. Comparisons use the risk score $1 - \widehat{S}_i(t)$ and IPCW weights from the training-data censoring model.

A larger AUC indicates better discrimination; `plotBrierAUC` shows a reference line at 0.5. The effective sample sizes and largest normalized weights in the output help identify times when a few observations dominate the comparison.

**Null reference for the Brier score:** By default, `plotBrierAUC` compares the forest with a Kaplan–Meier estimate from the full training sample. This reference assigns the same survival curve to every individual, without using predictors. A forest Brier curve below the dashed reference indicates lower estimated prediction error.

The reference survival curve is

$$\widehat{S}_0(t) = \prod_{u \le t} \left\{ 1 - \frac{d_{\text{grow}}(u)}{Y_{\text{grow}}(u)} \right\},$$

where $d_{\text{grow}}(u)$ and $Y_{\text{grow}}(u)$ are the event count and risk-set size at training event time $u$. This estimates event survival; `cens.model` separately determines the censoring distribution.

The reference substitutes $\widehat{S}_0(t)$ for $\widehat{S}_i(t)$ in the Brier loss. Both curves use the same times, censoring weights, and observations with finite forest losses, including zero contributions from individuals already censored. Confidence bands apply only to the forest; no test of the difference between curves is performed.

The reference uses training outcomes only, regardless of `subset`. For training evaluation, it is an in-sample benchmark even when the forest predictions are OOB. Set `brier.null=FALSE` to omit it.

**Confidence intervals:** Confidence intervals use pointwise normal approximations, treating the survival predictions and censoring estimates as fixed. They describe evaluation uncertainty, excluding uncertainty from fitting either model. The shaded regions in `plotBrierAUC` show these pointwise intervals, not simultaneous confidence bands.

**Brier score:** The standard error is the sample standard deviation of the finite individual IPCW losses divided by the square root of their count. This is equivalent to deleting one loss at a time while keeping the predictions and weights fixed.

**AUC:** A stratified delete-one jackknife removes one case or control at a time and renormalizes the remaining weights within that group. The case and control variance components are added. With equal weights, this gives the DeLong variance (DeLong et al., 1988).

For an independent test sample, the intervals describe conditional test performance. For a grow object, shared OOB fits and censoring estimates create dependence between observations. Interpret these as fixed-fit working intervals, not repeated-training confidence intervals.

**Concordance error:** `get.cindex` uses supplied outcomes and risk predictions directly. Supply `predicted.oob` for OOB evaluation, or `predicted` with the corresponding new-data outcomes for test evaluation. Concordance error is distinct from the time-specific AUC returned by `get.auct.survival`.

A status code greater than one selects competing-risk concordance. Here, `J` is the largest usable status code, and prediction column `j` must correspond to event code `j`; column names do not determine this match. Without weights, each event-specific calculation includes only censored

observations and observations with that event. With weights, the event-specific weighted concordance calculation is used.

Missing times, statuses, predictions, and supplied weights are excluded from the applicable calculation.

## Value

`plot.survival`: With evaluation outcomes, invisibly returns a data frame with `time`, mortality-stratified Brier curves `bs.q25`, `bs.q50`, `bs.q75`, `bs.q100`, and the overall `bs.all`. The corresponding integrated curves are `crps.q25`, `crps.q50`, `crps.q75`, `crps.q100`, and `crps.all`.

Without outcomes, invisibly returns a list with `time`, `survival` (a time-by-individual matrix), `survival.mean`, and `subset`.

`get.brier.survival`: Returns a list containing:

`brier.score` Data frame with `time` and `brier.score`. With intervals, also includes `std.err`, `lower`, `upper`, and `n.eval`.

`brier.matx` Subject-by-time IPCW losses; `NULL` when `keep.matrix=FALSE`.

`crps`, `crps.std` Integrated Brier scores defined in **Details**.

`n.eval` Number of finite Brier contributions at each time, including zeros from censored observations.

Supporting components include the time grid, censoring distribution, training and evaluation event information, survival predictions, mortality, and selected subset.

`get.auct.survival`: Returns a list with `auct.score` and the same supporting components as the Brier helper. The `auct.score` data frame contains `time`, `auct`, counts `n.case` and `n.control`, effective sample sizes `n.case.eff` and `n.control.eff`, and largest normalized weights `max.case.weight` and `max.control.weight`. With intervals, also includes `std.err`, `lower`, and `upper`.

`plotBrierAUC`: Invisibly returns the requested `brier` and/or `auct` result lists, including with `show.plots=FALSE`. The Brier result has `brier.matx=NULL`.

With `brier.null=TRUE`, `brier$null` contains reference scores `brier.score` (columns `time` and `brier.score`), `crps`, `crps.std`, and contribution counts `n.eval`. It also contains the reference `survival` vector, `n.train` (finite training outcomes), `method="kaplan-meier"`, and `source="grow"`. Integration and standardization match the forest scores. Individual reference losses and reference confidence intervals are not returned.

`get.cindex`: Returns a numeric concordance error for right-censored survival or a vector in event-code order `1`, ..., `J` for competing risks. In unweighted competing-risk calculations, an event with fewer than two usable observations has error `NA`.

## Note

When censoring is present, `cens.model="rfsrc"` requires training and evaluation predictors. For a reduced `rfsrc.fast` object fitted with `forest=FALSE`, use `cens.model="km"` or refit with `forest=TRUE`.

A score and its interval are `NA` when a required censoring probability is zero or nonfinite. AUC also requires at least one case and one control; its standard error requires at least two of each and positive remaining weight after each deletion. The Brier standard error requires at least two finite

losses. The null Brier score is NA if the forest score is unavailable or a required reference loss is nonfinite.

The Kaplan–Meier reference uses all finite training outcomes, counts repeated observations at ties, and processes events before censoring at a tie. Its final estimate applies beyond the last observed time, provided the required censoring probabilities remain positive and finite.

With collapse=FALSE and more than 500 selected individuals, plot.survival draws 500 randomly selected individual curves. The mean curve and scores still use the full selected group.

## 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 Journal*, 48:1029–1040.

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

DeLong E.R., DeLong D.M. and Clarke-Pearson D.L. (1988). Comparing the areas under two or more correlated receiver operating characteristic curves: a nonparametric approach, *Biometrics*, 44:837–845.

Efron B. and Tibshirani R.J. (1993). *An Introduction to the Bootstrap*. Chapman and Hall, New York.

Heagerty P.J. and Zheng Y. (2005). Survival model predictive accuracy and ROC curves, *Biometrics*, 61:92–105.

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

Ishwaran H., Kogalur U.B., Blackstone E.H. and Lauer M.S. (2008). Random survival forests, *Annals of Applied Statistics*, 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)

## ------------------------------------------------------------
## Concordance error from the stored OOB risk predictions
## ------------------------------------------------------------
print(get.cindex(pbc.obj$yvar[, 1], pbc.obj$yvar[, 2],
```

```
                pbc.obj$predicted.oob))

## standard survival diagnostics
plot.survival(pbc.obj)
plot.survival(pbc.obj, subset = c(3, 10), collapse = TRUE)

## Brier and AUCT helpers with pointwise intervals
brier.obj <- get.brier.survival(pbc.obj, conf.int = TRUE)
print(head(brier.obj$brier.score))
auct.obj <- get.auct.survival(pbc.obj, conf.int = TRUE)
print(head(auct.obj$auct.score))

## compare two grow-data censoring models
brier.km <- get.brier.survival(pbc.obj, cens.model = "km")
brier.rf <- get.brier.survival(pbc.obj, cens.model = "rfsrc")
plot(brier.km$brier.score$time,
     brier.km$brier.score$brier.score, type = "s", col = 2,
     xlab = "Time", ylab = "Brier Score")
lines(brier.rf$brier.score$time,
      brier.rf$brier.score$brier.score, type = "s", col = 4)
legend("bottomright",
       legend = c("cens.model = km", "cens.model = rfsrc"),
       col = c(2, 4), lty = 1)

## Brier and AUCT curves; Brier includes the grow-data KM null reference
perf <- plotBrierAUC(pbc.obj)
print(head(data.frame(
  time = perf$brier$time,
  forest = perf$brier$brier.score$brier.score,
  null = perf$brier$null$brier.score$brier.score
)))
plotBrierAUC(pbc.obj, plots = "auct", conf.int = 0.90)
plotBrierAUC(pbc.obj, plots = "brier", conf.int = 0.90,
             ylim = c(0, .4), null.lty = 3, legend.pos = "topleft")

## Show only the forest Brier curve and its confidence band
plotBrierAUC(pbc.obj, plots = "brier", brier.null = FALSE)

## Independent test evaluation: the null still uses training outcomes
set.seed(19)
pbc.complete <- na.omit(pbc)
trn <- sample(seq_len(nrow(pbc.complete)),
              size = floor(0.7 * nrow(pbc.complete)))
grow <- rfsrc(Surv(days, status) ~ ., pbc.complete[trn, ], ntree = 100)
test <- predict(grow, newdata = pbc.complete[-trn, ])
test.perf <- plotBrierAUC(test, plots = "brier")
print(c(forest = test.perf$brier$crps.std,
        null = test.perf$brier$null$crps.std))

## Obtain both curves without opening a graphics device
perf <- plotBrierAUC(test, plots = "brier", show.plots = FALSE)
print(head(perf$brier$null$brier.score))
```

`plot.variable.rfsrc` *Plot Marginal and Partial Dependence of Predictors*

## Description

Display ensemble predictions against predictors, either at their observed values (marginal plots) or after fixing one predictor at a sequence of values and averaging over the other predictors (partial plots). Plot data can be returned without drawing and reused for custom graphics.

## 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 a saved `(rfsrc, plot.variable)` object. New calculations require the training predictors and a usable saved forest. Anonymous and unsupervised forests are not supported. |
| `xvar.names` | Character vector naming predictors to include. The default is all available predictors. When replotting, select among predictors already present in the saved plotting object. |
| `target` | For classification, a class label or its position in the response levels; the first class is used by default. For competing risks, the event index, with 1 selecting the first event. |
| `m.target` | Single response name for a multivariate or mixed fit. The default selects the first available regression response, or the first available classification response when there is no regression response. For a factor response, `target` selects its class. |
| `time` | Single evaluation time for time-dependent survival summaries. The default is the median of the forest's stored `time.interest` grid. Supply a finite scalar no earlier than the first stored time. Between stored times, the last time not exceeding the request is used; requests beyond the grid use its last point. Ignored for mortality, relative mortality, and years lost. |
| `surv.type` | Survival summary to display. For right-censored data, select `"mort"`, `"rel.freq"`, `"surv"`, or `"chf"`. For competing risks, select `"years.lost"`, `"cif"`, or `"chf"`. Defaults are mortality and years lost, respectively. See Details. |

`class.type` For a factor response, `"prob"` displays the probability of the selected class; `"bayes"` selects the class-assignment display. In partial mode, `"bayes"` retains the numeric training class-code display. Use `"prob"` for interpretable partial class probabilities; averaging unordered class codes does not estimate a class probability.

`partial` Logical. `FALSE` produces marginal plots; `TRUE` computes partial dependence summaries.

`oob` Logical. Request OOB predictions by default; set to `FALSE` for the non-OOB ensemble. OOB predictions use trees that excluded each observation during training.

`show.plots` Logical. Set to `FALSE` to compute and return plotting data without opening or changing a graphics device.

`plots.per.page` Positive integer controlling the panel layout. The current layout uses up to this many columns and up to this many rows. Thus a value of 4 can place more than four panels on a page.

`granule` Numeric threshold for displaying a numeric predictor as a boxplot. Factors always use boxplots. For marginal plots the comparison uses the number of distinct observed values; for partial plots it uses the number of evaluation-grid values.

`sorted` Logical. Order predictors by decreasing available variable importance before applying `nvar`. Unavailable importance values follow finite ones. Ties retain predictor order. When no importance is stored, retain the predictor order.

`nvar` Maximum number of predictors selected for a new calculation. The default is all predictors selected by `xvar.names`.

`npts` Maximum number of evaluation points for a numeric predictor in partial mode. Points are selected from sorted distinct values, approximately evenly in their index order. Factors are not thinned.

`smooth.lines` Logical. Apply `lowess` to continuous partial curves and, when available, their display bands. The stored partial means are unchanged. Continuous marginal plots use `lowess` regardless of this option.

`subset` Numeric row indices or a logical selection relative to the fitted object's `xvar`. The default uses all stored training rows. Use `x$xvar`, not row numbers from an original data set that may have lost rows during fitting. In partial mode the selection determines both the displayed predictor grid and the requested averaging rows. Logical selections must have one entry per stored row. Missing and out-of-range numeric indices are omitted; fractional indices are rejected.

`...` Named graphical arguments for the displayed panels. Use `main`, `xlab`, `ylab`, `xlim`, and `ylim` for titles, labels, and limits. `col`, `lty`, `lwd`, `pch`, and `cex` customize continuous displays. `type` controls the fitted series; the partial mean defaults to points joined by lines. Partial variability bands are drawn separately. Boxplots accept `notch`, `boxfill`, `whisklwd`, `outline`, `horizontal`, `names`, and related `bxp` settings. Limits always refer to the displayed axes, including horizontal boxes. A named `pars` list is also accepted; individually supplied arguments take precedence. Graphical settings do not change the predictions, means, or standard errors.

## Details

**Marginal and partial plots:** Marginal plots relate the ensemble's predictions to the observed value of a predictor. They preserve the observed combinations of predictors. Continuous predictors receive a smoothed curve; factors and numeric predictors with few distinct values receive grouped boxplots.

Partial plots fix the selected predictor at each grid value for every observation, retaining the other predictor coordinates, and summarize those predictions. For a grid value $z$, write the observation-level partial prediction as $P_i(z) = \widehat{f}(z, X_{i,-j})$. With complete predictions the curve is $n^{-1} \sum_i P_i(z)$; missing predictions are omitted from the mean. When OOB output is requested, $P_i(z)$ uses the OOB trees for row $i$.

Partial mode reuses the fitted trees but performs additional prediction work at each grid value. Restrict `xvar.names` or `nvar` and use a moderate `npts` to control the computation. For explicit grids or several fixed predictors, use `partial.rfsrc`.

**Response selection:** Regression displays the predicted response. Classification displays the selected class probability by default. The `m.target` argument chooses the response for a multivariate or mixed forest; `target` then chooses a class when that response is a factor.

Right-censored survival displays mortality, relative-frequency mortality, survival probability, or cumulative hazard. Competing risks displays expected years lost, cumulative incidence, or cause-specific cumulative hazard for the selected event. Survival probability, cumulative incidence, and cumulative hazard depend on `time`; mortality and years lost are scalar summaries and do not use the requested time.

The existing marginal display multiplies right-censored survival and cumulative hazard values by 100; partial displays retain their raw scales. Relative mortality also retains its existing mode-specific scaling: marginal mortality is divided by the maximum of the selected row count and the available full-vector mortality values; partial mortality is divided by the available row count at each grid value.

**Subsets and missing values:** The plotting calculation starts from the stored training predictors and overlays saved imputed predictor values when available. It does not grow a replacement forest. Define subsets relative to the stored row order. In partial mode, grid construction uses nonmissing values of the selected predictor within that subset; the same subset is passed to the partial-prediction extraction step.

A subset affects the population being summarized, not which observations trained the forest. Likewise, OOB selection changes the contributing trees for an observation, not the definition of the plotting subset.

**Partial means and standard errors:** A single calculation is used for regression predictions, class probabilities, and scalar survival or competing-risk predictions. At grid value $z$, let $I_z$ contain the selected rows with finite partial predictions, and let $n_z$ be their number. The mean and empirical row-averaging standard error are

$$\bar{P}(z) = \frac{1}{n_z} \sum_{i \in I_z} P_i(z), \qquad \mathrm{se}(z) = \left\{ \frac{\sum_{i \in I_z} [P_i(z) - \bar{P}(z)]^2}{n_z(n_z - 1)} \right\}^{1/2}.$$

Thus `se` is the sample standard deviation divided by $\sqrt{n_z}$. Classification uses the observed variation among predicted probabilities, rather than a Bernoulli variance calculated from their mean.

Both numerator and denominator use the same available rows. With no finite predictions the mean is NA; with fewer than two, the standard deviation and standard error are NA. Identical finite predictions have zero standard error when at least two are present.

This standard error describes reference-row averaging with a fixed prediction function under independent reference sampling. For the training/OOB plots here, it is a descriptive row-averaging scale: it does not account for fitting the forest or for dependence between overlapping OOB fits. The continuous display uses pointwise bands $\bar{P}(z) \pm 2\mathrm{se}(z)$, not refit-based confidence intervals or simultaneous confidence bands. The numeric class-code display uses this calculation on its code scale, not a probability scale.

**Discrete partial displays:** Discrete displays retain boxplots of the transformed values $\bar{P}(z) + [P_i(z) - \bar{P}(z)]/\sqrt{n_z}$. Their sample standard deviation is exactly se when $n_z > 1$, so continuous and discrete displays use the same row-averaging scale. The boxes and whiskers remain ordinary boxplot summaries of these transformed values; their endpoints are not the mean plus or minus two standard errors. Notches likewise concern these display values, not uncertainty from refitting the forest. A single available value remains at its mean, with an unavailable standard error.

The returned plotthis tables retain these display values. partial.summary gives the means, standard deviations, standard errors, and available counts directly for every predictor, including factors. For unscaled observation-level predictions, use partial() and get.partial.plot.data(..., granule = TRUE).

**Returning and reusing plot data:** Assign the result of plot.variable(..., show.plots = FALSE) to retain the calculations. For partial mode, plotthis contains one data frame per predictor with columns x and yhat. Continuous curves normally have one row per grid value; discrete displays retain the shrunken values for each selected observation.

Call plot.variable(saved) to redraw without recalculating predictions. The saved object determines the marginal or partial mode, outcome, subset, grid, and display settings. Its plots.per.page and smooth.lines components can be changed before replotting. To change the prediction target, subset, or grid, create a new plotting object from the grow object.

## Value

Invisibly returns an object of class (rfsrc, plot.variable, family). Common components include family, partial, event.info, target, ylabel, n, xvar.names, nvar, plots.per.page, granule, and smooth.lines.

| | |
|---|---|
| pData | For partial plots, a named list of predictor-specific components: xvar.names, yhat, yhat.se, n.x, x.uniq, and the nonmissing predictor values x. |
| plotthis | For partial plots, named data frames with columns x and yhat, suitable for custom displays. These are display data, as described in Details. |
| partial.summary | For partial plots, named data frames with one row per evaluated grid value and columns x, mean, sd, se, and n.used. Counts refer to available predictions, not an estimated independence-adjusted effective sample size. |
| yhat, xvar | For marginal plots, the selected predictions and predictor data. |

## Author(s)

Hemant Ishwaran and Udaya B. Kogalur

## References

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

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

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

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

## See Also

rfsrc, partial.rfsrc, predict.rfsrc

## Examples

```
## ------------------------------------------------------------
##
## Regression: marginal versus partial display.
##
## ------------------------------------------------------------
set.seed(19)
dta <- na.omit(airquality)
fit <- rfsrc(Ozone ~ ., data = dta, ntree = 100, importance = TRUE)
plot.variable(fit, xvar.names = c("Wind", "Temp"), sorted = FALSE,
              plots.per.page = 2)
pv <- plot.variable(fit, xvar.names = c("Wind", "Temp"), sorted = FALSE,
                    partial = TRUE, npts = 15, plots.per.page = 2,
                    show.plots = FALSE)
print(head(pv$plotthis$Wind))
print(pv$partial.summary$Wind)
plot.variable(pv, main = "Partial dependence", lwd = 2)

## ------------------------------------------------------------
##
## Inspect the mean and its row-averaging standard error.
##
## ------------------------------------------------------------
ss <- pv$partial.summary$Wind
plot(ss$x, ss$mean, type = "b", xlab = "Wind", ylab = "Partial prediction")
lines(ss$x, ss$mean - 2 * ss$se, lty = 3)
lines(ss$x, ss$mean + 2 * ss$se, lty = 3)

## ------------------------------------------------------------
##
## Customize a stored continuous curve without another forest calculation.
##
## ------------------------------------------------------------
pv$plots.per.page <- 1
pv$smooth.lines <- TRUE
plot.variable(pv, xvar.names = "Temp", main = "Temperature")
```

```
## ------------------------------------------------------------
##
## Subset indices refer to the fitted object's stored rows.
##
## ------------------------------------------------------------
use <- fit$xvar$Solar.R < 200
ps <- plot.variable(fit, xvar.names = "Wind", partial = TRUE,
                    subset = use, npts = 15, show.plots = FALSE)
print(head(ps$plotthis$Wind))

## ------------------------------------------------------------
##
## Multivariate regression: choose a response for the plot.
##
## ------------------------------------------------------------
mv <- rfsrc(cbind(Ozone, Temp) ~ ., data = dta, ntree = 100)
mv.temp <- plot.variable(mv, m.target = "Temp", xvar.names = "Wind",
                         partial = TRUE, npts = 15, show.plots = FALSE)
print(mv.temp$partial.summary$Wind)
plot.variable(mv.temp)

## ------------------------------------------------------------
##
## Classification: choose a class label.
##
## ------------------------------------------------------------
iris.fit <- rfsrc(Species ~ ., data = iris, ntree = 100)
plot.variable(iris.fit, target = "versicolor", xvar.names = "Petal.Length",
              partial = TRUE, npts = 15)

## ------------------------------------------------------------
##
## A factor predictor receives a boxplot.
##
## ------------------------------------------------------------
car.data <- mtcars
car.data$cyl <- factor(car.data$cyl)
car.fit <- rfsrc(mpg ~ ., data = car.data, ntree = 100)
pc <- plot.variable(car.fit, xvar.names = "cyl", partial = TRUE,
                    show.plots = FALSE)
boxplot(yhat ~ x, data = pc$plotthis$cyl,
        xlab = "Cylinders", ylab = "Partial display values")

## ------------------------------------------------------------
##
## Mixed outcomes: m.target selects the response, target selects its class.
##
## ------------------------------------------------------------
car.data$am <- factor(car.data$am)
mix <- rfsrc(Multivar(mpg, am) ~ ., data = car.data, ntree = 100)
plot.variable(mix, m.target = "am", target = "1", xvar.names = "wt",
              partial = TRUE, npts = 15)
```

```
## ------------------------------------------------------------
##
## Right-censored survival at a stored evaluation time.
##
## ------------------------------------------------------------
data(veteran, package = "randomForestSRC")
vfit <- rfsrc(Surv(time, status) ~ ., data = veteran, ntree = 100)
when <- vfit$time.interest[ceiling(length(vfit$time.interest) / 2)]
plot.variable(vfit, xvar.names = "age", surv.type = "surv", time = when,
              partial = TRUE, npts = 15)

## ------------------------------------------------------------
##
## Competing risks: cumulative incidence for the second event.
##
## ------------------------------------------------------------
data(follic, package = "randomForestSRC")
crfit <- rfsrc(Surv(time, status) ~ ., data = follic, ntree = 100, nsplit = 3)
when <- crfit$time.interest[ceiling(length(crfit$time.interest) / 2)]
plot.variable(crfit, xvar.names = "age", target = 2, surv.type = "cif",
              time = when, partial = TRUE, npts = 15)
```

---

`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. The "test" setting requires test outcomes. |
| 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 a proximity matrix. Options include "inbag", "oob", "all", "hybrid", TRUE, or FALSE. With newdata, "hybrid" compares test observations (rows) with original training observations (columns). Hybrid mode is valid only when newdata is supplied with the default outcome = "train". |
| distance | Whether to compute the distance matrix. Options are the same as for proximity. Hybrid mode uses the same rectangular test-by-training layout. |
| forest.wt | Whether to compute the forest weight matrix. Options are FALSE (default), TRUE, "inbag", "oob", or "all". |
| 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 | Record how many times each variable was used for splitting. Options are FALSE (default), "all.trees", or "by.tree". |

| | |
|---|---|
| split.depth | Currently disabled in this prediction interface and internally set to FALSE. To obtain split depths for the training data, request split.depth = "all.trees" or split.depth = "by.tree" in rfsrc. |
| 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.

When newdata is supplied, non-hybrid proximity and distance matrices compare test observations with one another. Setting either option to "hybrid" instead compares each retained test observation with each retained original training observation, producing an ntest by ntrain matrix without constructing a square matrix over the combined data. Hybrid mode should not be combined with marginal.xvar.

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. Use get.mv.csvimp and get.mv.cserror to extract these stored values; see Extracting Multivariate Values (multivariate.values).

For prediction from a quantile forest, use quantreg.rfsrc with object and newdata. Its helpers include get.quantile for quantile extraction and get.pinball.error for mean pinball losses.

## Value

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

**call** The prediction call.

**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. In hybrid mode, rows correspond to retained test observations and columns correspond to retained original training observations.

**distance** Distance matrix. Hybrid mode uses the same test-by-training layout as hybrid proximity.

**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** `NULL`; split-depth calculations are currently disabled in this prediction interface.

**err.rate** Prediction error, if y-outcomes are present. For ordinary prediction with `newdata`, this is test error; restoring the training data uses OOB error.

**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) Response-named list of predictions and performance measures for continuous outcomes.

**classOutput** (Multivariate only) Response-named list of predictions and performance measures for categorical outcomes.

**chf** (Survival or competing risks) Cumulative hazard function (CHF); cause-specific CHF for competing risks.

**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`.

## 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 classOutput. These can be accessed using the helper functions get.mv.predicted, get.mv.vimp, and get.mv.error. See Extracting Multivariate Values (multivariate.values) for OOB selection, standardization, block errors, and return formats.

Additional classification scores are documented under Classification Performance Metrics (classification.performance); get.imbalanced.performance provides binary-class summaries with selectable thresholds. For right-censored survival diagnostics, see plotBrierAUC.

## 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

Classification Performance Metrics (classification.performance), Extracting Multivariate Values (multivariate.values), Fast Saving and Loading Objects (fast.saveload), holdout.vimp, imbalanced.rfsrc, impute.learn.rfsrc, plot.competing.risk.rfsrc, plot.rfsrc, plot.quantreg.rfsrc, plot.survival.rfsrc, plotBrierAUC, plot.variable.rfsrc, print.rfsrc, quantreg.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)

## ------------------------------------------------------------
## hybrid proximity and distance
## - rows are test cases; columns are training cases
## ------------------------------------------------------------
```

```
data(mtcars)
hybrid.train <- mtcars[1:24, ]
hybrid.test <- mtcars[25:32, ]
hybrid.obj <- rfsrc(mpg ~ ., data = hybrid.train, ntree = 50)

hybrid.proximity <- predict(hybrid.obj, hybrid.test,
                            proximity = "hybrid")$proximity
hybrid.distance <- predict(hybrid.obj, hybrid.test,
                           distance = "hybrid")$distance

## Both matrices have 8 test rows and 24 training columns.
print(dim(hybrid.proximity))
print(dim(hybrid.distance))


## ------------------------------------------------------------
## 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, ...,
                           quantreg.tau = 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. |
| quantreg.tau | Optional numeric vector of finite levels strictly between zero and one for the printed pinball losses of a quantile regression object. Supply this argument by name. When NULL, uses the session option quantreg.tau, then the former rfsrc.pinball.taus option for compatibility, then c(0.1, 0.5, 0.9). Explicit levels apply to this print call only and do not change the fitted object or session options. See quantreg for details. |

### 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

Estimates conditional quantiles of continuous responses using random forests. Returns predicted percentiles, cumulative distribution functions (CDFs), and distribution summaries for training or new observations. Supports univariate, multivariate, and mixed-outcome forests, with quantile estimates for each continuous response.

### Usage

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

extract.quantile(o)
get.quantile(o, target.prob = NULL, pretty = TRUE)
get.quantile.stat(o, pretty = TRUE)
```

```
get.quantile.crps(o, pretty = TRUE, subset = NULL, standardize = TRUE)
get.pinball.error(o, tau = NULL, subset = NULL, m.target = NULL,
                  pretty = TRUE)
```

## Arguments

| | |
|---|---|
| formula | Model formula. Required when growing a new forest and ignored when object is supplied. At least one response must be continuous. |
| data | Training data containing the responses and predictors. Required when growing a new forest. Converted to a plain data frame. |
| object | The original fitted object returned by quantreg, including its saved forest, training responses, and residuals. Use this object for subsequent prediction calls. |
| newdata | Data for new observations, containing the same predictor variables as the training data. Converted to a plain data frame. Response columns are optional and are used for performance evaluation. Use quantreg(object = fit, newdata = testdata) to obtain quantile output. Without newdata, or with newdata = NULL, results are calculated for the training observations. |
| method | How to estimate the response distribution:<br>"exchangeable" Add quantiles of the equally weighted training-residual distribution to each forest mean prediction (default).<br>"forest" Use forest weights to estimate a predictor-specific residual distribution, then add its quantiles to the forest mean prediction.<br>"gk" Compute quantiles with the Greenwald–Khanna algorithm. Also accepts "GK", "G-K", and "g-k".<br>The former name "local" is an alias for "exchangeable". The method applies to the current call; specify it again in later calls to use a nondefault method. |
| splitrule | Splitting rule used to grow the forest. The default for univariate regression is "la.quantile.regr", local adaptive quantile regression splitting. Multivariate forests use their default rule when either "la.quantile.regr" or "quantile.regr" is specified. Other applicable rules include "mse" for regression and "mahalanobis" for multivariate regression. |
| prob | Quantile levels to calculate, as a nonempty numeric vector of finite probabilities strictly between zero and one. Can be supplied during fitting or prediction; levels are sorted, including any duplicates. With NULL, the residual methods calculate percentiles 1 through 99 during fitting, and an object call uses the probabilities saved during fitting. GK training defaults also depend on prob.epsilon; supply prob to control the number of quantiles. |
| prob.epsilon | Greenwald–Khanna probability approximation tolerance, a finite number in $(0, 1)$. Larger values allow coarser summaries. A prediction-time NULL uses the saved training tolerance. When both prob and prob.epsilon are NULL during GK fitting, the default requests $2n$ quantiles, where $n$ is the processed training sample size. |
| oob | Use out-of-bag (OOB) predictions for training observations? During fitting, TRUE also selects OOB training residuals for the residual methods; FALSE selects residuals from full-ensemble predictions. Later calls reuse that residual |

selection. New observations always use full-ensemble predictions. See *Training residuals and OOB predictions* in Details for the distinction between the saved residuals and current predictions.

`fast` If `TRUE`, grow a new forest with `rfsrc.fast` instead of `rfsrc`. Ignored when `object` is supplied.

`maxn` Maximum number of response values in the CDF reporting grid, as a positive integer or `Inf`. Controls the resolution of `cdf`, `density`, and grid-integrated CRPS. For a fixed forest and prediction call, it does not change the quantiles or the residual-method means and standard deviations.

`o` A training or prediction object returned by `quantreg`. CRPS and pinball scoring require observed responses in `o$yvar`.

`target.prob` Quantile levels to extract with `get.quantile`. `NULL` returns all stored levels. Otherwise, supply finite probabilities in $[0, 1]$; each is matched to the nearest stored level.

`pretty` If `TRUE`, simplify a helper result for one continuous response to a matrix, data frame, or numeric vector, as described in Value. If `FALSE`, return a response-named list.

`subset` Rows to evaluate for CRPS or pinball loss. Supply positive integer indices or a logical vector with one nonmissing entry per row of the returned object. `NULL` uses all rows. The selection must be nonempty; repeated indices include a row more than once.

`standardize` If `TRUE`, divide each cumulative CRPS integral by its integration width.

`tau` Quantile levels at which to calculate pinball loss, as finite probabilities strictly between zero and one. Scores follow the supplied order, including repeated levels. `NULL` uses the session option `quantreg.tau`, with the defaults described under *Printed performance*. Supplied levels apply to this call only.

`m.target` For `get.pinball.error`, the name of one continuous response to evaluate. `NULL` evaluates all available continuous responses. Selects results from `o` without a prediction call.

`...` Additional options for `rfsrc` when growing a forest, or for its prediction method when `object` is supplied. `quantreg` sets the quantile method and any required forest-weight options.

## Details

Quantile regression describes the distribution of a response given its predictors. Whereas a regression forest gives a mean prediction, `quantreg()` estimates lower, middle, and upper quantiles. These estimates describe variation in response values and can be used to form prediction intervals. For example, `prob = c(.05, .50, .95)` requests the estimated 5th percentile, median, and 95th percentile for each observation.

The default method combines each forest mean prediction with a common distribution of training residuals. The `"forest"` method allows this residual distribution to vary with the predictors, and `"gk"` computes quantiles with the Greenwald–Khanna algorithm. Each method returns quantiles and a CDF for every evaluated observation.

**Fitting and prediction:** Supply `formula` and `data` to fit a forest. Save the resulting training object, for example as `fit`, and use `quantreg(object = fit, newdata = testdata)` to predict new observations. Use `quantreg(object = fit, prob = c(.1, .5, .9))` to calculate different quantiles for the training observations without growing new trees. Subsequent calls require the original training object, rather than a prediction object.

New data can contain predictors alone. Include response columns to calculate performance scores or use `plot.quantreg()`. These responses are used for evaluation; the saved training data continue to define the residual distributions and CDF reporting grid. Training and new-data prediction can also be requested in a single call by supplying `formula`, `data`, and `newdata`.

**Training residuals and OOB predictions:** With the default `oob = TRUE`, the residual methods use

$$R_j = Y_j - \widehat{m}_j^{\mathrm{OOB}},$$

where $\widehat{m}_j^{\mathrm{OOB}}$ averages predictions from trees whose training samples exclude observation $j$. With `oob = FALSE` during fitting, residuals instead use full-ensemble training predictions. This choice is saved during fitting and used in all subsequent calls.

For training observations, `oob` also selects the current mean predictions, forest weights, or GK quantiles. Changing `oob` in a later call changes these predictions, but not the saved residual selection. New observations use full-ensemble predictions together with the previously selected residuals. Thus a fit made with the default settings uses OOB training residuals even when predicting new observations. When fitting and new-data prediction are combined in one call, `oob` first selects the training residuals before the new observations are predicted.

If the requested OOB output component is absent, full-ensemble output is used instead. Missing entries within an existing OOB component stay missing; nonfinite residuals are excluded from the residual distribution. The common exchangeable distribution includes each training observation's finite OOB residual, including when calculating that observation's quantiles. GK quantiles are computed directly by the forest and do not use residuals.

**Exchangeable residual method:** The default `method = "exchangeable"` assigns equal weight to all finite training residuals. Their empirical CDF is

$$\widehat{F}_R(u) = \frac{1}{N_R} \sum_{j \in \mathcal{I}_R} I\{R_j \le u\},$$

where $\mathcal{I}_R$ indexes the $N_R$ finite residuals. Let $\widehat{Q}_R(\tau)$ be its quantile at probability $\tau$, and $\widehat{m}(x)$ the current forest mean prediction. The predicted response quantile and CDF are

$$\widehat{Q}(\tau \mid x) = \widehat{m}(x) + \widehat{Q}_R(\tau), \qquad \widehat{F}(s \mid x) = \widehat{F}_R\{s - \widehat{m}(x)\}.$$

Every observation therefore has the same residual-distribution shape and spread, shifted by its own forest prediction. Exchangeability is the working assumption behind pooling residuals in this way; OOB describes how the default residuals are calculated. This is the locally adjusted residual-CDF construction associated with Zhang et al. (2019).

**Forest-weighted residual method:** With `method = "forest"`, each observation has a weighted distribution of the saved training residuals:

$$\widehat{F}(s \mid x) = \sum_j w_j(x) I\{R_j \le s - \widehat{m}(x)\}.$$

The forest weights $w_j(x)$ sum to one over the usable residuals. Because the weights depend on $x$, both the shape and spread of the residual distribution can vary with the predictors. The weights are applied to residuals, rather than directly to training responses as in Meinshausen (2006).

Training observations use OOB weights when `oob = TRUE` and full-ensemble weights otherwise. New observations use full-ensemble weights. The weight matrix has one row per evaluated observation and one column per training observation, which can require substantial memory for large data sets.

**Residual quantiles and distribution summaries:** Both residual methods calculate a quantile by ordering the residuals and selecting the first whose cumulative weight reaches the requested probability. They then add the current forest prediction. This is the lower generalized inverse; equal weights give the type-1 sample quantile convention. Predicted quantiles can extend beyond the observed training-response range.

The mean and standard deviation describe the same predicted distribution. With $\bar{R}(x) = \sum_j w_j(x) R_j$, they are

$$\widehat{\mu}(x) = \widehat{m}(x) + \bar{R}(x), \qquad \widehat{\sigma}(x) = \left\{ \sum_j w_j(x)[R_j - \bar{R}(x)]^2 \right\}^{1/2}.$$

The exchangeable method uses equal weights. Residuals are used without recentering, so the distribution mean can differ from the forest mean prediction. These calculations use the full usable residual distribution, independently of the CDF reporting grid.

Nonfinite residuals are excluded together with their weights, and the remaining forest weights are normalized for each observation. Quantiles and summaries are `NA` when there are no usable residuals, the current prediction is nonfinite, or the weights for finite residuals contain nonfinite values or have no positive sum. Negative weights and incompatible weight-matrix dimensions cause an error.

**Greenwald–Khanna method:** The `"gk"` method computes quantiles with the native Greenwald–Khanna algorithm, without requesting an explicit forest-weight matrix. The returned quantiles come directly from this algorithm. A step CDF is constructed from their values and probability levels: at each response-grid value, it uses the largest requested probability whose quantile is no greater than that value. It is zero below all requested quantiles and reaches one at the largest finite training response.

For this method, `prob` controls both the requested quantiles and the resolution of the reconstructed CDF. The argument `maxn` controls only the grid at which that CDF is evaluated.

**CDF reporting grid and storage:** The reporting grid, `yunq`, contains sorted unique finite training responses, separately for each continuous response. When there are more than `maxn` values, the function selects approximately equally spaced positions in this ordered set. Use `maxn = Inf` to include all values.

For the residual methods, `cdf` evaluates the predicted CDF at these grid values. Its last value may be less than one when the shifted residual distribution extends above the grid. Quantiles, means, and standard deviations still use the entire usable residual distribution, including values outside the grid.

The component `density` contains interval probabilities, not density values divided by interval widths. Its first column is the CDF at the first grid value; subsequent columns are successive CDF differences. The row sum equals the final reported CDF value, without adjustment for probability above the grid.

Quantile storage grows with the number of evaluated observations times `length(prob)`. CDF and interval-probability storage grow with that observation count times the grid size. Reducing `maxn` reduces these latter matrices, but not the forest-weight matrix or the number of GK quantiles.

**Extracting results:**

`extract.quantile(o)` Returns all quantile components as a named list, with one entry per continuous response.

`get.quantile(o)` Returns the stored quantiles. Use `target.prob` to select particular levels. The helper matches each requested level to the nearest stored probability; column labels show the requested levels. To calculate additional levels, call `quantreg()` on the original training object with those levels in `prob`.

`get.quantile.stat(o)` Returns `mean`, `median`, and `std` for each observation. The residual methods use the distribution summaries above. GK means and standard deviations are calculated from the reporting-grid CDF increments and depend on that grid and its represented probability mass. The median uses the stored level nearest 0.5; include 0.5 in `prob` to obtain that exact level.

Quantile matrices have observations in rows and probability levels in columns, including when there is only one row or level.

**CRPS: evaluating the predicted distribution:** `get.quantile.crps(o)` assesses the predicted CDF using the observed responses. It returns a finite-grid continuous ranked probability score (CRPS) curve, with columns `y` and `crps`. Smaller scores indicate less prediction error.

At each reporting-grid value $s_k$, it averages $[I\{Y_i \leq s_k\} - \widehat{F}(s_k \mid X_i)]^2$ over the selected observations with finite response and CDF values. It then integrates these errors by the trapezoidal rule from the first grid value to $s_k$. With `standardize = TRUE`, each integral is divided by its integration width.

The last entry summarizes the entire reporting grid. Integration covers only the range of this grid. The first standardized entry is `NA` because its integration width is zero; with `standardize = FALSE`, the first entry is zero. Grid resolution and range therefore affect this score.

**Pinball loss: evaluating individual quantiles:** `get.pinball.error(o)` evaluates predictions at specified quantile levels. For response $Y_i$ and predicted quantile $\widehat{Q}_i(\tau)$, the loss is

$$\rho_\tau\{Y_i - \widehat{Q}_i(\tau)\}, \qquad \rho_\tau(u) = u\{\tau - I(u < 0)\}.$$

The helper returns the mean loss at each requested level; smaller values indicate better quantile predictions. Each mean uses the selected observations with finite responses and quantiles at that level. If none are available, the result is `NA`.

For `"exchangeable"` and `"forest"`, the helper uses exactly matching stored quantiles when available and otherwise calculates them from the saved residual distribution. Thus `tau` need not be included in `prob`, and scoring is independent of the reporting grid. GK scoring uses interpolation of the reporting CDF. Scoring uses the object's current predictions and saved residual selection, without refitting or a prediction call.

**Printed performance:** For a univariate quantile object with observed responses and performance output, `print.rfsrc` reports raw and standardized grid-integrated CRPS and mean pinball losses. For a multivariate object, use `print(object, outcome.target = "response.name")` to display quantile scores for one continuous response. Without `outcome.target`, the multivariate display

gives mean and response-specific regression or classification errors. Selecting a response for printing or extraction does not recalculate predictions. The OOB label for quantile scores follows that response's quantile predictions; the regression error is labeled according to the predictions used for that error.

Set levels for one print call with `print(object, quantreg.tau = c(.2, .5, .8))`, or set defaults for the session with `options(quantreg.tau = c(.2, .5, .8))`. Supplied levels override the session option. When neither is set, the former option `rfsrc.pinball.taus` is used if available, followed by the default `c(.1, .5, .9)`. These settings also determine the default `tau` for `get.pinball.error()`. They control reporting only and do not modify the fitted object. Invalid levels cause an error.

To display numerical pinball losses on a plot, supply levels in the plotting call, for example, `plot.quantreg(object, quantreg.tau = c(.2, .5, .8))`. The losses appear as text in the lower-right corner of the main panel. The plotted quantile interval and CRPS inset are unchanged. Without this argument, plotting omits the pinball labels even when the session option is set.

## Value

`quantreg()` returns an `rfsrc` training (grow) or prediction object with the additional class `quantreg`. For a univariate regression response, `o$quantreg` contains:

| | |
|---|---|
| `quantiles` | Matrix with evaluated observations in rows and requested probability levels in columns. |
| `prob` | Sorted probability levels corresponding to `quantiles`. |
| `cdf` | CDF matrix with evaluated observations in rows and `yunq` grid values in columns. |
| `density` | CDF increments, with the same dimensions as `cdf`. |
| `yunq` | Ordered training-response grid used to report the CDF. |
| `mean`, `std` | Predicted distribution mean and standard deviation vectors for the residual methods. `NULL` for GK; use `get.quantile.stat()` for its grid-based summaries. |
| `method` | Method used in this call: `"exchangeable"`, `"forest"`, or `"gk"`. The former name `"local"` is reported as `"exchangeable"`. |
| `oob` | Whether the current mean predictions or native GK quantiles use OOB output. |
| `prediction` | Current forest mean predictions used to shift the residual distribution. `NULL` for GK. |
| `residual.source` | Source of the saved training residuals: `"oob"` or `"ensemble"` for the residual methods. `NULL` for GK. A new-data result can have `residual.source = "oob"` and `oob = FALSE`: the training residuals are OOB, but the new predictions use the full ensemble. |

For multivariate and mixed-outcome forests, `o$quantreg` is a named list of these components, one per returned continuous response. Classification responses have no quantile component.

**Training data and residuals:** The usual `o$xvar` and `o$yvar` describe the observations being predicted, with `yvar` available when responses are supplied for evaluation. The saved forest's `xvar` and `yvar` describe the training data. The CDF grid and null CRPS reference use training responses; performance scores use the responses for the observations being evaluated.

For a univariate training object, `o$residual` = `o$yvar` - `o$predicted` and `o$residual.oob` = `o$yvar` - `o$predicted.oob`, when the respective predictions are available. For multivariate and mixed outcomes, these vectors are in `o$regrOutput[[response.name]]`. An unavailable prediction component gives a `NULL` residual component; missing entries within OOB predictions stay missing in the corresponding residuals.

The saved forest also contains both training-residual components for future prediction calls. They are vectors for univariate regression or matrices with training observations in rows and named continuous responses in columns. An unavailable response has an `NA` column in a partly available matrix; a wholly unavailable residual component is `NULL`. `o$forest$quantreg$residual.source` identifies the original selection for each response. Prediction objects include these training residuals under `forest`, separately from their current predictions.

**Helper functions:** `extract.quantile()` always returns a response-named list of quantile components. The other helpers return the following for one continuous response when `pretty` = `TRUE`, and a response-named list otherwise:

`get.quantile` An observation-by-probability matrix, with column names such as `q.50`.

`get.quantile.stat` A data frame with columns `mean`, `median`, and `std`, one row per observation.

`get.quantile.crps` A data frame with columns `y` and `crps`, one row per reporting-grid value.

`get.pinball.error` A numeric vector of mean losses, with names such as `"tau=0.2"`, one entry per requested level.

## 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*.

## See Also

`rfsrc`, `predict.rfsrc`, `plot.quantreg`

## Examples

```
## ------------------------------------------------------------
## A basic analysis using default settings
## ------------------------------------------------------------
o <- quantreg(Temp ~ ., data = na.omit(airquality))
plot.quantreg(o)


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

```
## Exchangeable OOB residual quantiles for wine alcohol
## ------------------------------------------------------------
data(wine, package = "randomForestSRC")
set.seed(17)
prob <- c(.05, .25, .50, .75, .95)
o <- quantreg(alcohol ~ ., data = wine, method = "exchangeable",
              oob = TRUE, prob = prob, ntree = 100)
print(head(get.quantile(o)))
print(head(get.quantile(o, c(.25, .50, .75))))
print(head(get.quantile.stat(o)))

## The common residual quantiles are added to each OOB mean prediction.
r.oob <- as.numeric(o$residual.oob)
r.oob <- r.oob[is.finite(r.oob)]
q.res <- as.numeric(quantile(r.oob, probs = prob, type = 1))
q.oob <- outer(as.numeric(o$predicted.oob), q.res, "+")
print(all.equal(unname(get.quantile(o)), unname(q.oob)))
print(o$quantreg$residual.source)

## Inspect the stored levels and reporting-grid probability mass.
print(o$quantreg$prob)
print(summary(rowSums(o$quantreg$density)))
crps <- get.quantile.crps(o)
print(crps)
plot(crps, type = "l")

## Pinball losses at nondefault levels: no refitting is needed.
print(o, quantreg.tau = c(.2, .5, .8))
print(get.pinball.error(o, tau = c(.2, .5, .8)))
plot.quantreg(o, quantreg.tau = c(.2, .5, .8))

## Optionally use the same reporting levels throughout the session.
op <- options(quantreg.tau = c(.2, .5, .8))
print(getOption("quantreg.tau"))
print(o)
options(op)

## ------------------------------------------------------------
## Forest-weighted residuals and predictor-only deployment
## ------------------------------------------------------------
set.seed(23)
train <- sample.int(nrow(wine), floor(.7 * nrow(wine)))
o <- quantreg(alcohol ~ ., data = wine[train, ],
              method = "forest", prob = prob, ntree = 100)
o.test <- quantreg(object = o, newdata = wine[-train, ],
                   method = "forest")
print(head(get.quantile(o.test)))
print(tail(get.quantile.crps(o.test, standardize = FALSE), 1))

x.test <- wine[-train, setdiff(names(wine), "alcohol"), drop = FALSE]
o.predict <- quantreg(object = o, newdata = x.test, method = "forest")
print(head(get.quantile(o.predict)))
```

```
## Exchangeable prediction uses the same saved OOB residual bank.
o.exchangeable <- quantreg(object = o, newdata = x.test,
                           method = "exchangeable")
print(head(get.quantile(o.exchangeable)))
print(identical(o.exchangeable$forest$residual.oob, o$residual.oob))
print(o.exchangeable$quantreg$residual.source)  # "oob": training residuals
print(o.exchangeable$quantreg$oob)              # FALSE: new-row predictions
print(c(evaluated.rows = o.exchangeable$n,
        training.rows = length(o.exchangeable$forest$residual.oob)))

## Calculate a single level using the original grow object.
o.median <- quantreg(object = o, newdata = x.test,
                     method = "forest", prob = .5)
print(head(get.quantile(o.median)))

## ------------------------------------------------------------
## Grid resolution does not determine residual quantiles or moments
## ------------------------------------------------------------
coarse <- quantreg(object = o, newdata = x.test, method = "exchangeable",
                   prob = prob, maxn = 2, seed = 37)
fine <- quantreg(object = o, newdata = x.test, method = "exchangeable",
                 prob = prob, maxn = Inf, seed = 37)
print(all.equal(get.quantile(coarse), get.quantile(fine)))
print(all.equal(get.quantile.stat(coarse), get.quantile.stat(fine)))

## ------------------------------------------------------------
## Multivariate and mixed outcomes
## ------------------------------------------------------------
dta <- na.omit(airquality)
mv <- quantreg(cbind(Ozone, Temp) ~ ., data = dta,
               splitrule = "mahalanobis", prob = prob, ntree = 100)
q.mv <- get.quantile(mv, pretty = FALSE)
print(names(q.mv))
print(head(q.mv$Ozone))
print(head(q.mv$Temp))
print(head(mv$regrOutput$Temp$residual.oob))
print(identical(mv$regrOutput$Temp$residual.oob,
                as.numeric(mv$forest$residual.oob[, "Temp"])))

## Restore all responses; select Temp only for extraction and printing.
mv.restore <- quantreg(object = mv, prob = .5)
q.restore <- get.quantile(mv.restore, pretty = FALSE)
print(names(q.restore))
print(head(q.restore$Temp))
print(get.mv.error(mv.restore))
print(mv.restore, outcome.target = "Temp")

dta$Month <- factor(dta$Month)
mixed <- quantreg(cbind(Ozone, Temp, Month) ~ ., data = dta,
                  prob = prob, ntree = 100)
print(names(extract.quantile(mixed)))

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

```
## Native GK quantiles and alternative regression splitting
## ------------------------------------------------------------
gk <- quantreg(alcohol ~ ., data = wine, method = "gk",
               prob = c(.05, .25, .50, .75, .95),
               prob.epsilon = .01, maxn = 100, ntree = 100)
print(head(get.quantile(gk)))

mse <- quantreg(alcohol ~ ., data = wine, method = "exchangeable",
                splitrule = "mse", prob = c(.05, .50, .95), ntree = 100)
print(head(get.quantile(mse)))

## ------------------------------------------------------------
## Larger data set; iowa housing
## ------------------------------------------------------------

data(housing,  package = "randomForestSRC")

## the original data contains lots of missing data; use fast imputation
iowa <- housing
iowa$PID <- NULL
iowa$SalePrice <- log(iowa$SalePrice)
iowa <- impute(SalePrice ~. , iowa, splitrule = "random", nimpute = 1)

## use fewer trees and shallow trees for speed
o <- quantreg(SalePrice ~., iowa, ntree = 50, nodesize = 20)
plot.quantreg(o, prbL=.05, prbU=.95)
```

---

`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. 2011). 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.rfsrc. |
| 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 FALSE. |
| 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 for quantile estimation methods and extraction helpers.
  (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, or G-mean-based error, as selected by `perf.type`.
- Survival: C-error = 1 - concordance, using Uno weighting when `use.uno = TRUE` and unweighted Harrell concordance otherwise.

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

See Classification Performance Metrics (`classification.performance`) for additional classification scores and `get.imbalanced.performance` for binary-class performance summaries and threshold choices. For right-censored survival, `plotBrierAUC` displays Brier and AUC curves, including a Kaplan–Meier reference for Brier score; `get.cindex` calculates concordance-based error.

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 and the `extract.subsample` and `extract.bootsample` helpers, see `subsample.rfsrc`; use `plot.subsample.rfsrc` to display the results. 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` to construct a multivariate formula. For `get.mv.predicted`, `get.mv.error`, `get.mv.vimp`, block errors, and case-specific extraction helpers, see Extracting Multivariate Values (`multivariate.values`).

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. See `impute.rfsrc` for standalone imputation and `impute.learn.rfsrc` for learning an imputer to apply to new data.

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: cases; columns: trees).

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

**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) Response-named list of predictions and performance results for continuous outcomes.

**classOutput** (Multivariate) Response-named list of predictions and 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 `classOutput`. Use `get.mv.predicted`, `get.mv.vimp`, and `get.mv.error` to extract results. Their OOB selection, standardization, and return formats are described in Extracting Multivariate Values (`multivariate.values`).

## 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

Classification Performance Metrics (`classification.performance`), Extracting Multivariate Values (`multivariate.values`), Fast Saving and Loading Objects (`fast.saveload`),

`get.tree.rfsrc`,

`holdout.vimp`,

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

`max.subtree.rfsrc`,

`partial.rfsrc`, `plot.competing.risk.rfsrc`, `plot.rfsrc`, `plot.quantreg.rfsrc`, `plot.subsample.rfsrc`, `plot.survival.rfsrc`, `plotBrierAUC`, `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)

## Brier and AUCT helpers with pointwise intervals
## see plot.survival for more details
brier.obj <- get.brier.survival(v.obj, conf.int = TRUE)
head(brier.obj$brier.score)
auct.obj <- get.auct.survival(v.obj, conf.int = TRUE)
head(auct.obj$auct.score)

## Brier and AUCT curves with confidence bands
## see plot.survival for more details
plotBrierAUC(v.obj, xlim = c(0, 600))
plotBrierAUC(v.obj, plots = "brier", xlim = c(0, 600))
plotBrierAUC(v.obj, plots = "auct", xlim = c(0, 600))

## 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$time.interest
bscore <- brier.obj$brier.score
crps <- sapply(1:length(time), function(j) {
  trapz(time[1:j], bscore[1:j, 2] / diff(range(time[1:j])))
})
plot(time, crps, ylab = "CRPS", type = "s", lwd = 2)


## 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, ...)

sid.perf.metric(truth, cluster, mode = c("entropy", "gini"))
```

## 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". |
| truth | Known class labels used to evaluate a clustering. Supply a complete vector with unused factor levels removed. |
| cluster | Cluster labels for the same observations, in the same order as truth. Supply one complete vector of the same length, with unused factor levels removed. If several clustering solutions were requested, select one column of clustering. |

| | |
|---|---|
| mode | Measure used by sid.perf.metric: cluster-size-weighted entropy (the default) or Gini impurity. Smaller values indicate purer clusters. |
| ... | 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.

**Evaluating a clustering:** `sid.perf.metric` measures how well a clustering separates known classes. It evaluates the mixture of true classes within each cluster: a cluster containing only one true class has zero impurity, while a cluster containing several classes has positive impurity. The true labels are supplied for evaluation after clustering. The helper can evaluate SID or any other clustering method.

Supply `truth` and `cluster` for the same observations in the same order. Cluster labels are arbitrary identifiers; they need not match the names or numerical values in `truth`. The number of clusters may differ from the number of true classes. For a matrix of clustering solutions, evaluate one column at a time. Use complete label vectors with unused factor levels removed.

Let $K$ be the number of observed true classes and $J$ the number of nonempty clusters. In the cross-tabulation `table(truth, cluster)`, let $n_{kj}$ count observations from true class $k$ assigned to cluster $j$. Write $n_j = \sum_{k=1}^{K} n_{kj}$ for the size of cluster $j$, $n = \sum_{j=1}^{J} n_j$ for the total sample size, and $p_{kj} = n_{kj}/n_j$ for the within-cluster class proportions.

For `mode = "entropy"` (the default), the impurity of cluster $j$ is

$$H_j = -\sum_{k=1}^{K} p_{kj} \log_2(p_{kj}),$$

with zero-probability terms contributing zero. Entropy measures the remaining uncertainty about the true class within that cluster, in bits. For `mode = "gini"`, the impurity is

$$G_j = 1 - \sum_{k=1}^{K} p_{kj}^2.$$

This is the probability that two independent draws from the cluster's empirical class distribution have different labels.

The returned `result` is the cluster-size-weighted average,

$$H = \sum_{j=1}^{J} \frac{n_j}{n} H_j \quad \text{or} \quad G = \sum_{j=1}^{J} \frac{n_j}{n} G_j.$$

Weighting by $n_j/n$ makes a cluster's contribution proportional to the number of observations it represents, rather than giving a small cluster the same weight as a large one. The weighted entropy is the empirical conditional entropy of the true class given cluster membership. Smaller values indicate purer clusters in either mode.

**Normalizing and interpreting the clustering score:** The maximum impurity of a cluster with $K$ possible true classes occurs when those classes are equally represented. The maxima are $\log_2(K)$ for entropy and $1 - 1/K$ for Gini. Accordingly, `normalized_measure` is

$$H_{\text{norm}} = \frac{H}{\log_2(K)} \quad \text{or} \quad G_{\text{norm}} = \frac{G}{1 - 1/K}.$$

For $K > 1$, the normalized score lies between zero and one. Zero means every cluster contains only one true class; one means every cluster contains the same proportion $1/K$ of each true class. The score is a fraction of the uniform-class maximum impurity, rather than a misclassification rate. With only one true class, the raw score is zero and the normalized score is undefined.

For example, suppose there are ten observations from two true classes, A and B. One cluster contains four A observations and is pure; a second contains three A and three B observations and

is evenly mixed. Their weights are $0.4$ and $0.6$. The entropy impurities are zero and one, giving `result = 0.6` and `normalized_measure = 0.6`. The Gini impurities are zero and $0.5$, giving `result = 0.3` and `normalized_measure = 0.6` after division by $0.5$.

The normalization uses the uniform-class maximum, not the impurity of the observed marginal class proportions or an expected score under random clustering. Consequently, when the true classes are imbalanced, a clustering that reproduces the overall class proportions in every cluster can provide no class separation and still have a normalized score below one.

These measures assess within-cluster purity. Dividing one true class into several pure clusters still gives zero impurity, and assigning each observation to its own cluster also gives zero. Thus a low score does not by itself select an appropriate number of clusters or require a one-to-one correspondence with the true classes. For method comparisons, use the same observations and true labels, and compare solutions with the same number of clusters when isolating differences in class separation. See Mantero and Ishwaran (2021) for theoretical background and empirical benchmarking.

## Value

`sidClustering` returns 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. |

**Clustering performance:** `sid.perf.metric` returns a list with three components:

**result** The cluster-size-weighted entropy or Gini impurity.

**measure** The selected mode, `"entropy"` or `"gini"`.

**normalized_measure** The weighted impurity divided by its maximum for equally represented true classes, as described in Details.

## 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))
  print(sid.perf.metric(y, o2$clustering, mode = "gini"))

  ## 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` *Subsampling Inference for Variable Importance and Prediction Error*

---

## Description

Estimate standard errors and confidence intervals for variable importance (VIMP) and out-of-bag prediction error by fitting forests to repeated subsamples of the training data. Extract, print, or plot the results at different confidence levels from the same subsample estimates. An optional double-bootstrap procedure is also available.

## 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)
```

```
extract.subsample(obj, alpha = .05, target = 0, m.target = NULL,
                  standardize = TRUE, raw = FALSE)

extract.bootsample(obj, alpha = .05, target = 0, m.target = NULL,
                   standardize = TRUE, raw = FALSE)

## S3 method for class 'subsample'
print(x, alpha = .05, standardize = TRUE, ...)

## S3 method for class 'bootsample'
print(x, alpha = .05, standardize = TRUE, ...)
```

## Arguments

| | |
|---|---|
| obj | For subsample, a forest grow object of class (rfsrc, grow) with saved forest information and training data. Grow the forest with forest = TRUE and bootstrap = "by.root". Anonymous forests and nonuniform case weights are not supported. For extract.subsample, an object returned by subsample, including double-bootstrap output. For extract.bootsample, an object returned by subsample(..., bootstrap = TRUE). |
| B | Number of subsamples, or double-bootstrap replicates when bootstrap = TRUE. Each replicate requires a new forest fit. Supply a positive integer; at least two replicates are needed to calculate a sample standard deviation. |
| block.size | Number of trees per block for calculating VIMP when it must first be computed for obj. If obj already contains VIMP, its block size is used instead. |
| importance | VIMP perturbation method, such as "anti", "permute", or "random". For subsampling, existing VIMP in obj determines the method. If VIMP is absent, it is computed using this argument, or importance = TRUE when omitted. Ignored when performance.only = TRUE. |
| subratio | Fraction of training observations to draw for each subsample. Supply a finite number greater than zero and at most one; for inference, choose a fraction below one that gives enough observations to fit each forest. The default is $\exp(-1)$ for $n \leq 1000$ and $\exp(-1)\sqrt{1000/n}$ for larger $n$. This controls the data set for each forest, not its per-tree sampsize. Ignored when bootstrap = TRUE. |
| stratify | Sample separately within response classes for univariate classification, or within event-status groups, including censoring, for survival and competing risks. Regression, multivariate, and mixed-outcome forests use unstratified subsampling. Ignored when bootstrap = TRUE. |
| performance | Include uncertainty estimates for OOB prediction error as well as VIMP. The error measure is determined by the original forest's perf.type. |
| performance.only | Compute uncertainty for OOB prediction error only. Sets performance = TRUE and turns off individual and joint VIMP. |
| joint | Include joint VIMP for the predictor group specified by xvar.names, in addition to individual VIMP. Joint VIMP uses the default anti perturbation of vimp, independently of the method used for individual VIMP. |

| | |
|---|---|
| `xvar.names` | Predictors to perturb together for joint VIMP. The default `NULL` uses all predictors. Individual VIMP is still computed for all predictors. To select results for display, use `xvar.names` in `plot.subsample`. |
| `bootstrap` | Use the double-bootstrap procedure instead of subsampling without replacement. See Details. |
| `verbose` | Print progress information during resampling. |
| `alpha` | Significance level for extraction or printing, strictly between zero and one. Intervals have nominal confidence level $1 - \alpha$; the default `.05` gives 95 percent intervals. |
| `target` | Statistic to extract within the selected response. Supply a nonnegative integer: `0` for regression or right-censored survival; `0` for overall classification or `1`, `2`, and so on for its classes; and `0`, `1`, and so on for competing-risk events. Classes and events follow the order in the forest. For competing risks, the first event uses `target = 0` here but `target = 1` in `plot.subsample`. |
| `m.target` | Name of one response to extract from a multivariate or mixed-outcome forest. The default selects the first regression response with results, or the first classification response with results if none is available for regression. Not needed for univariate forests. |
| `standardize` | For regression, divide estimates by the response variance in the full training data. This applies to VIMP, prediction error, and their uncertainty summaries. Other families are unchanged. Set `FALSE` for the original scale. |
| `raw` | Include replicate estimates and interval matrices in the extraction result. This controls the amount of output, not its scale; use `standardize = FALSE` for unscaled values. |
| `x` | An object returned by `subsample`. Use `print.subsample` for subsampling and `print.bootsample` for double-bootstrap output. |
| `...` | Additional arguments to the print methods; currently unused. |

## Details

Use `subsample()` to add standard errors and confidence intervals to a forest's VIMP estimates. The original forest provides the full-data estimate for each predictor. Forests grown on repeated subsamples show how these estimates vary when the training observations change. The same approach can estimate uncertainty in OOB prediction error.

The analysis has two stages: `subsample()` fits the replicate forests, then `extract.subsample()`, `print()`, or `plot.subsample()` uses their saved estimates to construct intervals. Once resampling is complete, you can change the confidence level, interval method, or response without fitting more forests.

**Choosing the statistics:** By default, the analysis computes uncertainty for individual VIMP. Specify the desired VIMP method in the original grow call, for example `importance = "permute"`. When `obj` already contains VIMP, subsampling uses that method and block size, even if different values are supplied to `subsample()`. Otherwise, VIMP is first calculated on the saved trees using the requested `importance` and `block.size`.

Two additional statistics can be included:

**Joint VIMP.** `joint = TRUE` adds a row named `joint`, measuring the effect of perturbing the predictors in `xvar.names` together. Individual VIMP is also included.

**Prediction error.** `performance = TRUE` adds a row named `err` for OOB prediction error. Use `performance.only = TRUE` to compute only this row.

**How subsampling works:** Each replicate draws training observations without replacement and grows a new forest on that subset. `B` controls the number of replicates; `subratio` controls their nominal sample size. Increase `B` for more stable interval endpoints, especially at high confidence levels.

Subsampling the data and sampling observations for individual trees are separate operations. `subratio` determines the data set available to each new forest. The original forest's `sampsize` function is then applied to that data set's size to determine its per-tree sample size. The forest settings used for these fits are listed in the Note.

With `stratify = TRUE`, univariate classification samples within classes, and survival and competing risks sample within event-status groups. Integer allocation can leave small groups unrepresented; missing group labels are excluded. Other families sample without stratification.

For survival and competing risks, a subsample must contain at least one observed event after missing-value omission. Otherwise the procedure stops and asks for a larger `subratio`; it does not draw a replacement subsample.

**Extracting and displaying results:** After `smp <- subsample(obj)`, `extract.subsample(smp, alpha = .05)` returns all three subsampling interval summaries for one response: nonparametric, normal subsampling, and normal jackknife. For multivariate or mixed outcomes, use `m.target` to choose the response. For classification or competing risks, `target` selects its overall, class-specific, or event-specific statistic as described under Arguments.

`print(smp, alpha = .05)` displays intervals for every response and statistic column, rounded to three decimal places. `plot.subsample(smp, alpha = .05)` draws the normal subsampling intervals; its `normal` and `jknife` arguments select the other methods. Use extraction for numerical results at full precision, and `raw = TRUE` for the replicate estimates and interval matrices listed under Value.

Extraction and printing default to 95 percent intervals (`alpha = .05`); plotting defaults to 99 percent (`alpha = .01`). Supply the same `alpha` when comparing them. Intervals are calculated separately for each statistic, without a multiple-testing adjustment.

**Standardization:** Extraction, printing, and plotting use `standardize = TRUE` by default. For regression, the full-data and replicate estimates are divided by the response variance in the full training data. This same divisor is used for every replicate and for both VIMP and prediction error. Confidence limits and standard errors are then on the corresponding standardized scale.

Use `standardize = FALSE` for the original scale. Classification and survival statistics are unchanged by this option. Regression standardization requires a finite, positive response variance.

**Subsampling intervals:** For one statistic, let $\widehat{\theta}$ be its full-data estimate and $\theta_b^*$ its estimate in subsample $b$. Let $n$ be the training sample size, $r$ the value of `subratio`, and $m = nr$ the nominal subsample size. Write $\bar{\theta}_*$ and $s_*$ for the mean and sample standard deviation of the available subsample estimates.

**Normal subsampling intervals.** The subsampling standard error measures dispersion about the subsample mean:

$$\mathrm{SE}_{\mathrm{sub}} = \sqrt{r}\, s_*.$$

It is returned as `se.Z`. The interval is $\widehat{\theta} \pm z_{1-\alpha/2}\mathrm{SE}_{\mathrm{sub}}$ and is reported in `var.sel.Z`.

**Normal jackknife intervals.** A subsample of size $m$ also deletes $d = n - m$ observations. The delete-$d$ jackknife calculation measures dispersion about the full-data estimate:

$$\mathrm{SE}_{\mathrm{jk}} = \left\{ r \, \mathrm{mean}_b(\theta_b^* - \widehat{\theta})^2 \right\}^{1/2}.$$

It is returned as `se.jk.Z`. The interval is $\widehat{\theta} \pm z_{1-\alpha/2}\mathrm{SE}_{\mathrm{jk}}$ and is reported in `var.jk.sel.Z`.

**Nonparametric subsampling intervals.** These use the empirical distribution of the centered roots $Z_b^* = \sqrt{m}(\theta_b^* - \widehat{\theta})$. If $q_p$ is their empirical quantile, computed with the default `quantile` interpolation, the endpoints are

$$\left[ \widehat{\theta} - q_{1-\alpha/2}/\sqrt{n}, \widehat{\theta} - q_{\alpha/2}/\sqrt{n} \right].$$

The interval is reported in `var.sel`. Centering, scaling, and reversing the root quantiles converts subsample variation to an interval for the full-data statistic; this is not a percentile interval of the unscaled VIMP replicates.

The jackknife calculation includes both subsample variation and the difference between the subsample mean and the full-data estimate:

$$\mathrm{mean}_b(\theta_b^* - \widehat{\theta})^2 = \mathrm{mean}_b(\theta_b^* - \bar{\theta}_*)^2 + (\bar{\theta}_* - \widehat{\theta})^2.$$

The final term is the nonnegative bias adjustment discussed by Ishwaran and Lu (2019), equations (10) and (11). The Note explains the scaling and finite-replicate denominators used here.

**Interpreting the summary tables:** Each table gives interval endpoints and a `signif` column indicating whether the lower endpoint exceeds zero. For VIMP, this identifies positive importance under the selected interval method. An `err` row instead describes uncertainty in prediction error; its positive lower endpoint is not a variable-selection result. The normal tables report the full-data estimate in `mean`; the nonparametric table reports an interval median.

Normal-table p-values are one-sided upper-tail tests of zero VIMP. A positive lower endpoint of a two-sided $1 - \alpha$ normal interval corresponds to a one-sided level of $\alpha/2$. The nonparametric p-value is the proportion of available roots satisfying $Z_b^* > \sqrt{n}\,\widehat{\theta}$. Ties are excluded, and no Monte Carlo adjustment is added. Because interval endpoints use interpolated quantiles, the empirical p-value and interval can give different decisions at finite-sample boundaries.

**Double-bootstrap inference:** Set `bootstrap = TRUE` to draw $n$ observations with replacement for each outer replicate, then generate the per-tree samples from that bootstrap data set. With `samptype = "swor"`, trees sample without replacement from its rows; an original observation can still appear more than once because the bootstrap data set can contain duplicates.

For these objects, `extract.subsample()` calls `extract.bootsample()`. It returns percentile intervals and normal intervals centered at the bootstrap mean, using the bootstrap standard deviation. Printing displays both methods. The percentile table has no p-value column; the centered-root p-value above applies only to subsampling without replacement.

## Value

`subsample()` returns the reference forest and replicate estimates. Use extraction, printing, or plotting to obtain intervals. Its class includes `subsample` for subsampling or `bootsample` for double bootstrapping, together with the original forest classes.

`rf` Reference forest with full-data statistics. If VIMP was initially absent, it has been calculated on the saved trees.

`vmp` List of full-data statistic matrices, one per outcome. Rows correspond to predictors and any requested `joint` or `err` statistic. Classification and competing risks can have several columns. Unavailable appended statistics are `NA` where their outcome and column can be matched. These values are not standardized.

`vmpS` For subsampling, a length-`B` list of replicate results, each organized by outcome. The tables use the outcomes, row names, and column names in `vmp`; unavailable entries are `NA`.

`vmpB` Double-bootstrap replicate estimates, in place of `vmpS`.

`subratio` Nominal subsample fraction, or `NULL` for double-bootstrap objects.

`performance.only` Whether only prediction error was requested.

**Extraction helpers:** Each helper returns a list for one response and statistic column. Summary tables have one row per predictor or requested joint/error statistic. All extracted values use the requested `standardize` setting.

`extract.subsample()` For subsampling, returns `se.Z` and `se.jk.Z`, the subsampling and jackknife standard errors; `var.sel`, the nonparametric table; `var.sel.Z`, the normal subsampling table; and `var.jk.sel.Z`, the normal jackknife table. For a double-bootstrap object, it returns the `extract.bootsample()` result.

`extract.bootsample()` Returns `se`, the bootstrap standard errors; `var.sel`, the percentile table; and `var.sel.Z`, the normal table. Both tables contain `lower`, `mean`, `upper`, and `signif`; only `var.sel.Z` includes `pvalue`.

With `raw = TRUE`, the helpers also return:

`vmp` Full-data estimates for subsampling, or bootstrap means for double bootstrapping.

`vmpS` Matrix with statistics in rows and replicates in columns, for either procedure.

`ci`, `ci.Z`, `ci.jk.Z` Five-row interval matrices for the nonparametric, normal subsampling, and normal jackknife methods. Rows give the lower endpoint, lower quartile, center, upper quartile, and upper endpoint. Double-bootstrap output includes only `ci` and `ci.Z`, both with the bootstrap mean in the middle row.

`boxplot.dta` Centered and rescaled replicate values used for subsampling plots. Not returned for double bootstrapping.

The print methods display interval summaries. Use the extraction helpers for numerical results. `print.subsample.rfsrc` and `print.bootsample.rfsrc` have the same arguments as `print.subsample` and `print.bootsample`, respectively.

## Note

**Forest settings and missing data:** Subsample forests use the original formula, number of trees, `mtry`, node-size and depth settings, splitting rule, `nsplit`, predictor and split weights, sampling-size function, sampling type, and performance settings. Survival forests also use the saved time grid and choice of Uno weighting. Specialized grow options outside these settings are not reproduced.

Each new fit omits rows with missing values rather than repeating the original imputation. Missing-value omission and integer allocation within strata can reduce the actual subsample size. Interval scaling uses the nominal size $m = nr$, where $r$ is `subratio`; actual sample counts are not saved in the result.

**Variance scaling:** `se.jk.Z` uses the factor $m/n$; equation (11) of Ishwaran and Lu (2019) uses $m/(n-m)$. These factors are asymptotically equivalent when $m/n \to 0$. With $K$ available estimates, `se.Z` uses the sample-variance denominator $K-1$, whereas `se.jk.Z` averages squared deviations with denominator $K$. Consequently, the returned jackknife standard error can be smaller than the subsampling standard error for a finite set of replicates.

## 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.

Geyer, C.J. (2013). 5601 Notes: The Subsampling Bootstrap. July 11, 2013. Section 5, equation (12).

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

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

## See Also

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

## Examples

```
## Small settings keep these examples manageable. Increase B for more
## stable interval endpoints, especially at high confidence levels.
set.seed(19)
dta <- na.omit(airquality)

## ------------------------------------------------------------
## Regression: grow once, then reuse the saved replicate estimates
## ------------------------------------------------------------
reg.o <- rfsrc(Ozone ~ ., data = dta, ntree = 100,
               importance = "permute", block.size = 1)
reg.smp <- subsample(reg.o, B = 25, verbose = FALSE)

ans <- extract.subsample(reg.smp, alpha = .05)
print(ans$var.sel.Z)
print(ans$var.jk.sel.Z)
print(reg.smp, alpha = .05)
plot.subsample(reg.smp, alpha = .05)
plot.subsample(reg.smp, alpha = .05, normal = FALSE)
```

```
## Detailed output and unstandardized VIMP are separate requests.
raw <- extract.subsample(reg.smp, alpha = .05,
                         standardize = FALSE, raw = TRUE)
print(raw$ci.Z)
print(dim(raw$vmpS))

## ------------------------------------------------------------
## Joint VIMP and prediction-error uncertainty
## ------------------------------------------------------------
with.error <- subsample(reg.o, B = 25, performance = TRUE,
                         joint = TRUE, xvar.names = c("Day", "Month"),
                         verbose = FALSE)
ans <- extract.subsample(with.error, alpha = .05)
print(ans$var.sel.Z[c("joint", "err"), , drop = FALSE])

error.only <- subsample(reg.o, B = 25, performance.only = TRUE,
                         verbose = FALSE)
print(extract.subsample(error.only, alpha = .05)$var.sel.Z)

## ------------------------------------------------------------
## Multivariate regression: select an outcome after subsampling
## ------------------------------------------------------------
mv <- rfsrc(cbind(Ozone, Temp) ~ ., data = dta, ntree = 100,
            importance = "permute", block.size = 1)
mv.smp <- subsample(mv, B = 25, verbose = FALSE)
print(extract.subsample(mv.smp, m.target = "Temp", alpha = .05)$var.sel.Z)
plot.subsample(mv.smp, m.target = "Ozone", alpha = .05)
plot.subsample(mv.smp, m.target = "Temp", alpha = .05,
               jknife = TRUE)

## ------------------------------------------------------------
## Classification with a different performance measure
## ------------------------------------------------------------
if (requireNamespace("caret", quietly = TRUE)) {
  cls.data <- caret::twoClassSim(500, linearVars = 5, noiseVars = 5)
  cls <- rfsrc(Class ~ ., data = cls.data, ntree = 100,
               importance = "permute", block.size = 1,
               perf.type = "brier")
  cls.smp <- subsample(cls, B = 25, verbose = FALSE)
  print(extract.subsample(cls.smp, alpha = .05)$var.sel.Z)
  plot.subsample(cls.smp, alpha = .05, cex.axis = .7)
}

## ------------------------------------------------------------
## Survival
## ------------------------------------------------------------
data(pbc, package = "randomForestSRC")
pbc.complete <- na.omit(pbc)
srv <- rfsrc(Surv(days, status) ~ ., data = pbc.complete,
              ntree = 100, importance = "permute", block.size = 1)
srv.smp <- subsample(srv, B = 25, verbose = FALSE)
plot.subsample(srv.smp, alpha = .05)
```

```
## ------------------------------------------------------------
## Optional double bootstrap (separate resampling procedure)
## ------------------------------------------------------------
reg.boot <- subsample(reg.o, B = 25, bootstrap = TRUE, verbose = FALSE)
print(reg.boot, alpha = .05)
print(extract.bootsample(reg.boot, alpha = .05)$var.sel.Z)
```

---

`tune.rfsrc` *Tune Random Forest Parameters*

---

## Description

Select `mtry` and `nodesize` for a supervised random forest using holdout or out-of-bag (OOB) prediction error. `tune()` searches over both parameters; `tune.nodesize()` searches over `nodesize` only. Both functions offer grid and golden-section search and fit the tuning forests with `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 specifying the response and predictors. |
| `data` | A data frame containing the response and predictors. |
| `mtry.start` | Starting value of `mtry` for `tune()`. |
| `nodesize.try` | Candidate terminal-node sizes for grid search, subject to the limits in **Note**. Not used for golden search. |
| `ntree.try` | Number of trees in each tuning forest, excluding the final forest returned by `tune()`. |

| | |
|---|---|
| sampsize | Per-tree sample size, supplied as a function of the sample size or as a number. For a numeric vector, such as class-specific sizes, the total is used during tuning. |
| nsplit | Number of random split points considered at each node. |
| step.factor | Factor for increasing or decreasing mtry during grid search in tune(). Must be greater than one. |
| improve | Relative-error reduction threshold for tune(). Must be nonnegative; 1e-3 corresponds to a 0.1 percent decrease. |
| strikeout | Number of additional steps allowed after error increases in grid search, counted across each search direction. For golden search in tune(), the number of consecutive alternating-search iterations without sufficient improvement before stopping. |
| max.iter | Maximum iterations per search direction (grid) or per one-dimensional search (golden). For tune() with golden search, also limits the number of alternating searches. |
| method | Search method: "grid" (default) or "golden". See **Details**. |
| final.window | Target interval width for golden search. Also sets the distance searched on either side of the best value from the initial small-value comparisons. |
| reps.initial | Number of replicate forests per setting during golden-section search; their errors are averaged. |
| reps.final | Number of replicate forests per setting in the final integer comparisons for golden search; their errors are averaged. |
| trace | If TRUE, print tuning progress and errors. |
| do.best | If TRUE, tune() also fits a forest on all available data using the selected settings. |
| seed | Optional integer seed for reproducible tuning, including the holdout sample when used. |
| ... | Additional options for rfsrc.fast, such as splitrule and perf.type. In tune(), ntree, mtry, nodesize, forest, and save.memory are controlled by the tuner; values supplied here are ignored. |

## Details

Tuning compares forest settings to find those with the lowest prediction error. The parameter mtry controls how many predictors are considered at each split, and nodesize controls terminal-node size. A forest is grown for each candidate setting and its prediction error is calculated. The selected setting has the smallest error in the returned results table.

### Choosing a search method:

method = "grid": For each eligible nodesize.try value, tune() starts at mtry.start and searches toward smaller and larger mtry values. step.factor sets the spacing; improve, strikeout, and max.iter control when to stop. The grid specifies nodesize values, while mtry values are chosen during the search. tune.nodesize() evaluates the nodesize.try grid directly, subject to the sample-size limit in **Note**.

`method = "golden"`: Comparisons within an interval progressively narrow the search toward values with smaller error. `tune()` alternates between searching over `nodesize` with `mtry` fixed and searching over `mtry` with `nodesize` fixed. `tune.nodesize()` searches over `nodesize` only.

Each one-dimensional search also checks `1:9` within its limits. It finishes with individual integer comparisons near the best small value or within the remaining interval. Averaging replicate forests, controlled by `reps.initial` and `reps.final`, reduces the effect of random variation in forest fits. `nodesize.try` and `step.factor` are not used.

**Comparing prediction errors:**

Both functions use one per-tree sample size, `ssize`, throughout tuning. Let `n` be the sample size after preprocessing. When `2 * ssize < n`, a random holdout of `ssize` observations is set aside. Forests are grown on the remaining data, and all candidates are evaluated on this same holdout. Otherwise, forests use all available data and are evaluated by OOB error.

Supply `perf.type` to choose the tuning error measure. `tune()` averages the available errors from `get.mv.error` with `standardize = TRUE`: regression errors are divided by the response variance, and errors for multivariate responses are averaged.

**Fitting the final forest:**

With `do.best = TRUE`, `tune()` fits a new forest on all available data, using the selected `mtry` and `nodesize`, the original `sampsize` argument, and the specified `nsplit`. The forest is returned as `rf`.

This final fit uses the default tree count and performance measure of `rfsrc.fast`. To choose these yourself, set `do.best = FALSE` and fit a forest separately with the returned `optimal` values. `tune.nodesize()` returns its selected node size and error table without a final forest fit.

## Value

`tune()` returns a list with components:

**results** A numeric matrix with columns `nodesize`, `mtry`, and `err` (prediction error). For golden search, this table contains the final integer comparisons; repeated parameter pairs are combined by averaging their errors.

**optimal** A named numeric vector giving the `nodesize` and `mtry` with the smallest error in `results`.

**rf** The final fitted forest when `do.best = TRUE`; otherwise `NULL`.

`tune.nodesize()` returns a list with components:

**nsize.opt** The selected `nodesize`.

**err** A data frame with columns `nodesize` and `err`, giving the candidate values and their prediction errors.

## Note

The tuning sample size `ssize` is obtained by evaluating `sampsize(n)`, using a numeric scalar directly, or summing a numeric vector. In `tune()`, it is rounded and bounded between 2 and `n`.

For `tune()`, `mtry` ranges from 1 to the number of predictors, and the upper `nodesize` limit is `max(10, floor(ssize / 2))`. Grid values above this limit are excluded. For `tune.nodesize()`, the upper grid limit is half the per-tree sample size, rounded down.

## Author(s)

Hemant Ishwaran and Udaya B. Kogalur

## See Also

rfsrc.fast, rfsrc, get.mv.error

## 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 = "Tuning 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


∗ **anonymous**
rfsrc.anonymous, 137
∗ **classif**
classification.performance, 7
imbalanced.rfsrc, 29
∗ **clustering**
sidClustering.rfsrc, 143
∗ **confidence interval**
subsample.rfsrc, 150
∗ **datasets**
breast, 6
follic, 19
hd, 23
housing, 28
nutrigenomic, 63
pbc, 73
peakVO2, 73
vdv, 162
veteran, 163
wihs, 166
wine, 167
∗ **documentation**
rfsrc.news, 142
∗ **fast**
rfsrc.fast, 140
∗ **file**
fast.saveload, 11
∗ **forest**
predict.rfsrc, 98
rfsrc, 118
rfsrc.anonymous, 137
rfsrc.fast, 140
tune.rfsrc, 158
∗ **missing data**
impute.learn.rfsrc, 38
impute.rfsrc, 53
∗ **models**
classification.performance, 7
multivariate.values, 60
∗ **package**
randomForestSRC-package, 3
∗ **partial**
partial.rfsrc, 65
∗ **plot**
get.tree.rfsrc, 20
plot.competing.risk.rfsrc, 74
plot.quantreg.rfsrc, 76
plot.rfsrc, 79
plot.subsample.rfsrc, 81
plot.survival.rfsrc, 84
plot.variable.rfsrc, 92
∗ **predict**
predict.rfsrc, 98
vimp.rfsrc, 163
∗ **print**
print.rfsrc, 107
∗ **quantile regression forests**
quantreg.rfsrc, 108
∗ **subsampling**
subsample.rfsrc, 150
∗ **survival**
plot.survival.rfsrc, 84
∗ **tree**
imbalanced.rfsrc, 29
∗ **tune**
tune.rfsrc, 158
∗ **unsupervised**
sidClustering.rfsrc, 143
∗ **variable selection**
max.subtree.rfsrc, 57
vimp.rfsrc, 163
∗ **vimp**
holdout.vimp.rfsrc, 24
subsample.rfsrc, 150

breast, 6
bxp, *84*

classification.performance, 7, *62*, *102*,

*123*, *127*

extract.bootsample, *123*
extract.bootsample *(*subsample.rfsrc*)*, 150
extract.quantile *(*quantreg.rfsrc*)*, 108
extract.subsample, *84*, *123*
extract.subsample *(*subsample.rfsrc*)*, 150

fast.load *(*fast.saveload*)*, 11
fast.save *(*fast.saveload*)*, 11
fast.saveload, 11, *102*, *127*
follic, 19, *75*

get.auc *(*classification.performance*)*, 7
get.auct.survival, *10*
get.auct.survival *(*plot.survival.rfsrc*)*, 84
get.bayes.rule *(*classification.performance*)*, 7
get.brier.error *(*classification.performance*)*, 7
get.brier.survival, *10*
get.brier.survival *(*plot.survival.rfsrc*)*, 84
get.cindex, *123*
get.cindex *(*plot.survival.rfsrc*)*, 84
get.confusion *(*classification.performance*)*, 7
get.imbalanced.optimize *(*imbalanced.rfsrc*)*, 29
get.imbalanced.performance, *8*, *10*, *32*, *35*, *102*, *123*
get.imbalanced.performance *(*imbalanced.rfsrc*)*, 29
get.logloss *(*classification.performance*)*, 7
get.misclass.error *(*classification.performance*)*, 7
get.mv.cserror, *100*
get.mv.cserror *(*multivariate.values*)*, 60
get.mv.csvimp, *100*
get.mv.csvimp *(*multivariate.values*)*, 60
get.mv.error, *64*, *102*, *123*, *125*, *160*, *161*
get.mv.error *(*multivariate.values*)*, 60
get.mv.formula, *64*, *123*
get.mv.formula *(*multivariate.values*)*, 60
get.mv.predicted, *102*, *123*, *125*
get.mv.predicted *(*multivariate.values*)*, 60
get.mv.vimp, *64*, *102*, *123*, *125*
get.mv.vimp *(*multivariate.values*)*, 60
get.partial.plot.data *(*partial.rfsrc*)*, 65
get.pinball.error, *79*, *100*
get.pinball.error *(*quantreg.rfsrc*)*, 108
get.pr.auc *(*classification.performance*)*, 7
get.pr.curve *(*classification.performance*)*, 7
get.quantile, *79*, *100*
get.quantile *(*quantreg.rfsrc*)*, 108
get.quantile.crps, *79*
get.rfq.threshold *(*imbalanced.rfsrc*)*, 29
get.tree, *3*
get.tree *(*get.tree.rfsrc*)*, 20
get.tree.rfsrc, *6*, 20, *127*

hd, 23, *75*
holdout.vimp, *3*, *99*, *102*, *123*, *127*, *144*
holdout.vimp *(*holdout.vimp.rfsrc*)*, 24
holdout.vimp.rfsrc, *6*, 24, *59*, *156*, *164*
housing, 28

imbalanced, *4*
imbalanced *(*imbalanced.rfsrc*)*, 29
imbalanced.rfsrc, *4*, *6*, *10*, 29, *102*, *127*
impute, *4*, *138*
impute *(*impute.rfsrc*)*, 53
impute.learn *(*impute.learn.rfsrc*)*, 38
impute.learn.rfsrc, 38, *56*, *102*, *124*, *127*
impute.ood *(*impute.learn.rfsrc*)*, 38
impute.rfsrc, *4*, *6*, *49*, 53, *120*, *124*, *127*

load.impute.learn, *13*
load.impute.learn *(*impute.learn.rfsrc*)*, 38

max.subtree *(*max.subtree.rfsrc*)*, 57
max.subtree.rfsrc, *6*, 57, *127*
multivariate.values, 60, *64*, *100*, *102*, *123*, *125*, *127*

nutrigenomic, 63

partial, *4*
partial *(*partial.rfsrc*)*, 65
partial.rfsrc, *4*, *6*, 65, *94*, *96*, *127*

pbc, 73
peakVO2, 73
plot.competing.risk (plot.competing.risk.rfsrc), 74
plot.competing.risk.rfsrc, 6, 74, 87, 90, 102, 127
plot.default, 77, 79
plot.quantreg, 115
plot.quantreg (plot.quantreg.rfsrc), 76
plot.quantreg.rfsrc, 76, 102, 127
plot.rfsrc, 6, 79, 102, 127
plot.subsample (plot.subsample.rfsrc), 81
plot.subsample.rfsrc, 81, 123, 127, 156
plot.survival (plot.survival.rfsrc), 84
plot.survival.rfsrc, 6, 84, 102, 127
plot.variable (plot.variable.rfsrc), 92
plot.variable.rfsrc, 6, 66, 68, 69, 92, 102, 127
plotBrierAUC, 102, 123, 127
plotBrierAUC (plot.survival.rfsrc), 84
predict.impute.learn (impute.learn.rfsrc), 38
predict.rfsrc, 3, 6, 10, 12, 13, 49, 62, 64, 90, 96, 98, 115, 127
print.bootsample (subsample.rfsrc), 150
print.imbalanced.performance (imbalanced.rfsrc), 29
print.impute.learn (impute.learn.rfsrc), 38
print.rfsrc, 6, 102, 107, 127
print.subsample (subsample.rfsrc), 150

quantreg, 3, 31, 76, 79, 108
quantreg (quantreg.rfsrc), 108
quantreg.rfsrc, 3, 6, 100, 102, 108, 122, 127

randomForestSRC (rfsrc), 118
randomForestSRC-package, 3
readRDS, 11, 13
rfsrc, 3, 6, 10, 13, 24, 30, 31, 34, 35, 49, 56, 62, 64, 75, 90, 96, 100, 102, 115, 118, 127, 137, 138, 141, 144, 145, 148, 156, 161, 164
rfsrc.anonymous, 127, 137
rfsrc.cart, 6, 127
rfsrc.fast, 3, 6, 30, 31, 34, 35, 56, 102, 122, 127, 140, 144, 148, 158, 159, 161
rfsrc.news, 142

save.impute.learn (impute.learn.rfsrc), 38
saveRDS, 11, 13
sid.perf.metric (sidClustering.rfsrc), 143
sidClustering, 122
sidClustering (sidClustering.rfsrc), 143
sidClustering.rfsrc, 3, 6, 127, 143
subsample, 3, 32, 35, 62
subsample (subsample.rfsrc), 150
subsample.rfsrc, 6, 81, 82, 84, 123, 127, 150

tune (tune.rfsrc), 158
tune.rfsrc, 6, 127, 158

vdv, 162
veteran, 163
vimp, 3, 32, 35, 61, 62, 99
vimp (vimp.rfsrc), 163
vimp.rfsrc, 6, 26, 59, 102, 127, 156, 163

wihs, 75, 166
wine, 167