Package: randomForestSGT (via r-universe)

June 10, 2026

Version 1.0.0

Date 2026-05-05

Title Random Forest Super Greedy Trees

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

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

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

Depends R (>= 4.3.0)

Imports randomForestSRC (>= 3.6.2), varPro (>= 3.1.0)

Suggests mlbench, interp, glmnet

Description Implements random forest Super Greedy Trees (SGTs) for regression. SGTs extend classification and regression tree splitting by fitting lasso-penalized local parametric models at tree nodes, producing sparse univariate and multivariate geometric cuts such as axis-aligned splits, hyperplanes, ellipsoids, hyperboloids, and interaction-based cuts. Trees are grown best-split-first by selecting cuts that reduce empirical risk, and ensembles provide out-of-bag error estimation, prediction on new data, variable filtering, tuning of the hcut complexity parameter, coordinate-descent lasso fitting, variable importance, and local coefficient summaries. For the underlying method, see Ishwaran (2026) <doi:10.1007/s10462-026-11541-6>.

License GPL (>= 3)

URL https://ishwaran.org/

NeedsCompilation yes

Repository https://cran.r-universe.dev

Date/Publication 2026-05-11 20:09:30 UTC

RemoteUrl https://github.com/cran/randomForestSGT

RemoteRef HEAD

RemoteSha 88e4c8b2c806aef6f85fccf639c87cb8458b2bc5

Contents

`cdlasso.rfsgt` *Coordinate Descent Lasso*

Description

Fit lasso for regression using coordinate descent.

Usage

```
cdlasso(formula,
      data,
      nfolds = 0,
      weights = NULL,
      nlambda = 100,
      lambda.min.ratio = ifelse(n < n.xvar, 0.01, 1e-04),
      lambda = NULL,
      threshold = 1e-7,
      eps = .0001,
      maxit = 5000,
      efficiency = ifelse(n.xvar < 500, "covariance", "naive"),
      seed = NULL,
      do.trace = FALSE)
```

Arguments

`formula`	Formula describing the model to be fit.
`data`	Data frame containing response and features.
`nfolds`	Number of cross-validation folds where default is 0 corresponding to no cross-validation.
`weights`	Observation weights. Default is 1 for each observation.
`nlambda`	The number of `lambda` values; default is 100.
`lambda.min.ratio`	Smallest value for `lambda`, as a fraction of `lambda.max` which equals smallest value for which all coefficients are zero. A very small value of `lambda.min.ratio` will lead to a saturated fit in if number of observations `n` is less than number of features `n.xvar`.

lambda	Lasso lambda sequence. Default is an internally selected sequence based on nlambda and lambda.min.ratio. For experts only.
threshold	Convergence threshold for coordinate descent. Each inner coordinate-descent loop continues until the maximum change in the objective after any coefficient update is less than threshold times the null deviance.
eps	Multiplication factor applied to lambda.min.ratio used to define the smallest lambda value.
maxit	Maximum number of passes over the data for all lambda values.
efficiency	Switches the algorithm to efficiency or naive mode depending on number of variables. Efficiency covariance saves all inner-products and can be significantly faster in certain settings than naive which loops through all values n each time an inner-product is formed.
seed	Negative integer specifying seed for the random number generator.
do.trace	Number of seconds between updates to the user on approximate time to completion.

Details

Use coordinate descent to fit lasso to a regression model.

Value

A list containing the fitted lasso solution path. The list contains:

convgCount Convergence counter returned by the coordinate-descent routine.

lambdaCount Number of lambda values in the fitted solution path.

lambda The sequence of lambda values used.

beta Matrix of regression coefficients for the lasso solution path. Rows correspond to values in lambda; columns contain the intercept followed by the encoded predictor variables.

xvar Numeric predictor matrix used in the fit, after any internal encoding of the supplied data.

yvar Response vector used in the fit.

yHat Cross-validated fitted values or predictions by lambda and observation. Returned only when cross-validation output is available, such as when nfolds is greater than 1.

lambda.min.indx Index of the lambda value with minimum cross-validation error. Returned only when cross-validation output is available.

lambda.1se.min.indx Index of the minimum lambda value within one standard error of the minimum cross-validation error. Returned only when cross-validation output is available.

lambda.1se.max.indx Index of the maximum lambda value within one standard error of the minimum cross-validation error. Returned only when cross-validation output is available.

lambda.cvm Mean cross-validation error for each lambda. Returned only when cross-validation output is available.

lambda.cvsd Cross-validation standard-error values for each lambda. Returned only when cross-validation output is available.

ctime.internal Timing information reported by the native coordinate-descent routine.

ctime.external Elapsed R-side timing, computed from proc.time().

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Friedman, J., Hastie, T. and Tibshirani, R. (2010) Regularization paths for generalized linear models via coordinate descent, *J. of Statistical Software*, 33(1):1-22.

See Also

rfsgt

Examples

```
## ------------------------------------------------------------
## regression example: boston housing
## ------------------------------------------------------------

if (requireNamespace("mlbench", quietly = TRUE)) {
## load the data
data(BostonHousing, package = "mlbench")

## 10-fold validation
o <- cdlasso(medv ~., BostonHousing, nfolds=10)

## lasso solution
bhat <- data.frame(bhat.min=o$beta[o$lambda.min.indx,],
                   bhat.1se=o$beta[o$lambda.1se.max.indx[1],])
print(bhat)

## compare to results from glmnet
if (library("glmnet", logical.return = TRUE)) {

  oo <- cv.glmnet(data.matrix(o$xvar), o$yvar, nfolds=10)
  bhat2 <- cbind(data.matrix(coef(oo, s=oo$lambda.min)),
                 data.matrix(coef(oo, s=oo$lambda.1se)))
  rownames(bhat2) <- rownames(bhat)
  print(bhat2)

}
}
```

---

predict.rfsgt *Prediction on Test Data for Super Greedy Forests*

Description

Obtain predicted values on test data using a trained super greedy forest.

Usage

```
## S3 method for class 'rfsgt'
predict(object, newdata,  get.tree = NULL,
     block.size = 10, seed = NULL, do.trace = FALSE,...)
```

Arguments

object	rfsgt object obtained from previous training call using rfsgt.
newdata	Test data. If not provided the training data is used and the original training forest is restored.
get.tree	Vector of integer(s) identifying trees over which the ensemble is calculated over. By default, uses all trees in the forest.
block.size	Determines how cumulative error rate is calculated. To obtain the cumulative error rate on every nth tree, set the value to an integer between 1 and ntree.
seed	Negative integer specifying seed for the random number generator.
do.trace	Number of seconds between updates to the user on approximate time to completion.
...	Additional options.

Details

Returns the predicted values for a super greedy forest.

Value

An object of class c("rfsgt", "predict", family) containing predictions and prediction-time summaries. When newdata is omitted, the object corresponds to the restored training forest; when newdata is supplied, it corresponds to predictions on the new data. The returned list contains:

call The matched prediction call.

family Model family inherited from the fitted rfsgt object.

n Number of observations predicted.

samptype Sampling type inherited from the fitted forest.

sampsize Tree sample size inherited from the fitted forest.

ntree Number of trees in the fitted forest.

hcut hcut value used by the fitted forest.

splitrule Split rule used by the fitted forest.

yvar Response values used for prediction-error calculation. This is the training response in restore mode, the response from newdata when present, and NULL when newdata has no response column.

yvar.names Response variable name from the fitted forest.

xvar Predictor data used for prediction, after applying the same encoding and filtering as in training.

`xvar.augment` Augmented base-learner data used for prediction, or `NULL` when no augmented terms are used.

`xvar.names` Names of the retained predictor variables.

`xvar.augment.names` Names of augmented base-learner terms, or `NULL` when no augmented terms are used.

`xvar.info` Predictor-encoding metadata used to align new data with the training design.

`term.map` Term map describing generated base-learner terms.

`leaf.count` Number of terminal nodes in each tree.

`forest` Stored forest object used to make the predictions.

`membership` Matrix of terminal-node membership by observation and tree when membership output is requested; otherwise `NULL`.

`inbag` Matrix of bootstrap membership counts in restore mode when membership output is requested; otherwise `NULL`.

`block.size` Block size used for cumulative prediction error calculations.

`perf.type` Internal performance-measure type used for prediction-error calculation.

`predicted` Ensemble predicted values.

`predicted.oob` Out-of-bag predictions in restore mode when available; otherwise `NULL`.

`ambrOffset` Terminal-node offset matrix used internally to recover local terminal-node quantities for new observations; `NULL` in restore mode.

`err.rate` Prediction error when response values are available; otherwise `NULL`.

`ctime.internal` Timing information reported by the native prediction routine.

`ctime.external` Elapsed R-side timing, computed from `proc.time()`.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H. (2025). Super greedy trees. To appear in Artifical Intelligence Review.

See Also

`rfsgt`

Examples

```
## ------------------------------------------------------------
##
##  mtcars: for CRAN testing
##
## ------------------------------------------------------------

o <- rfsgt(mpg~., mtcars[1:20,], ntree=3, treesize=1)
p <- predict(o, mtcars[-(1:20),])
```

```
print(o)
print(p)



## ------------------------------------------------------------
##
## train/test using friedman 3
##
## ------------------------------------------------------------

if (requireNamespace("mlbench", quietly = TRUE)) {

  ## train/test using Friedman 3
  d.trn <- data.frame(mlbench::mlbench.friedman3(100))
  o <- rfsgt(y ~ ., d.trn, hcut = 1, ntree = 3, treesize = 1)
  print(o)

  d.tst <- data.frame(mlbench::mlbench.friedman3(200))
  y.tst <- d.tst$y
  x.tst <- d.tst[, colnames(d.tst) != "y"]
  yhat <- predict(o, x.tst)$predicted
  mean((yhat - y.tst)^2)

## train sgf on friedman 3
d.trn <- data.frame(mlbench::mlbench.friedman3(500))
o <- rfsgt(y~.,d.trn, hcut=1)
print(o)

## test sgf
d.tst <- data.frame(mlbench::mlbench.friedman3(1000))
y.tst <- d.tst$y
x.tst <- d.tst[, colnames(d.tst)!= "y"]
yhat <- predict(o, x.tst)$predicted
cat("test set mse:", mean((yhat - y.tst)^2), "\n")

## ------------------------------------------------------------
##
## restore a trained super greedy forest using boston
##
## ------------------------------------------------------------

## run sgf on boston
data(BostonHousing, package = "mlbench")
o <- rfsgt(medv~., BostonHousing)
print(o)

## restore the forest
print(predict(o))

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

```
## coherence check using boston housing with factors
##
## ------------------------------------------------------------

## boston housing data: make factors
data(BostonHousing, package = "mlbench")
Boston <- BostonHousing[1:40,]
Boston$zn <- factor(Boston$zn)
Boston$chas <- factor(Boston$chas)
Boston$lstat <- factor(round(0.2 * Boston$lstat))
Boston$nox <- factor(round(20 * Boston$nox))
Boston$rm <- factor(round(Boston$rm))

## grow a single tree - save inbag information
o <- rfsgt(medv~., Boston, hcut=2, filter=FALSE, ntree=1, membership=TRUE, nodesize=3)

## coherence matrix
pred <- data.frame(
      inbag=o$inbag,
      pred.inb=o$predicted,
      pred.oob=o$predicted.oob,
      pred.inb.restore=predict(o)$predicted,
      pred.oob.restore=predict(o)$predicted.oob,
      pred.test=predict(o,Boston)$predicted)
print(pred)

## coherence check
cat("coherence for inbag data:", sum(pred$pred.inb-pred$pred.test,na.rm=TRUE)==0, "\n")
cat("  coherence for oob data:", sum(pred$pred.oob-pred$pred.test,na.rm=TRUE)==0, "\n")


## canonical example of train/test with prediction
trn <- sample(1:nrow(Boston), nrow(Boston)/2, replace=FALSE)
o.trn <- rfsgt(medv~., Boston[trn,], hcut=2)
predict(o.trn,Boston[-trn,])


## ------------------------------------------------------------
## prediction using tuning hcut and pre-filtering with tune.hcut
## ------------------------------------------------------------

## fit the forest to the tuned hcut
dta <- data.frame(mlbench::mlbench.friedman3(500))
f <- tune.hcut(y~., dta, hcut=5, verbose=TRUE)
o <- rfsgt(y~., dta, filter=f)
print(o)

## test the tuned forest on new data
print(predict(o, data.frame(mlbench::mlbench.friedman3(25000))))

## over-ride the optimized hcut
o2 <- rfsgt(y~., dta, filter=use.tune.hcut(f, hcut=2))
print(o2)
```

```
print(predict(o2, data.frame(mlbench::mlbench.friedman3(25000))))

}
```

`print.rfsgt` *Print Output from a Random Forest Super Greedy Tree Analysis*

Description

Print summary output from a Random Forest SGT analysis. This is the default print method for the package.

Usage

```
## S3 method for class 'rfsgt'
print(x, ...)
## S3 method for class 'vimp.rfsgt'
print(x, ...)
```

Arguments

`x`	An object of class `"rfsgt"`, `"grow"` or `"vimp.rfsgt"`.
`...`	Further arguments passed to or from other methods.

Value

Called for its side effect of printing a summary of an `rfsgt` grow or predict object. The return value is `NULL`.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

`rfsgt` *Random Forest Super Greedy Trees*

Description

Grow a forest of Super Greedy Trees (SGTs) using lasso. In addition to prediction, the fitted forest supports local beta-value and partial-contribution summaries that show how predictions are assembled.

Usage

```
rfsgt(formula,
      data,
      ntree = if (hcut == 0) 500 else 100,
      hcut = 1,
      treesize = NULL,
      nodesize = NULL,
      filter = (hcut > 1),
      bsf = if (hcut > 0) "oob" else "inbag",
      keep.only = NULL,
      fast = TRUE,
      pure.lasso = FALSE,
      eps = .005,
      maxit = 500,
      nfolds = 10,
      block.size = 10,
      bootstrap = c("by.root", "none", "by.user"),
      samptype = c("swor", "swr"),  samp = NULL, membership = TRUE,
     sampsize = if (samptype == "swor") function(x){x * .632} else function(x){x},
      seed = NULL,
      do.trace = FALSE,
      ...)
```

Arguments

formula	Formula describing the model to be fit.
data	Data frame containing response and features.
ntree	Number of trees to grow.
hcut	Integer value indexing type of parametric regression model to use for splitting. See details below.
treesize	Function specifying upper bound for size of tree (number of terminal nodes) where first input is n sample size and second input is hcut. Can also be supplied as an integer value and defaults to an internal function if unspecified.
nodesize	Minumum size of terminal node. Set internally if not specified.
filter	Logical value specifying whether dimension reduction (filtering) of features should be performed.Can also be specified using the helper function tune.hcut which performs dimension reduction prior to fitting. See examples below.
bsf	Best split first (BSF) empirical risk minimization strategy. Accepted values are "oob" and "inbag". The default is "oob" when hcut>0 and "inbag" otherwise. Using OOB (out-of-bag) empirical risk minimization can improve robustness, but it can also optimistically bias OOB error estimates. Setting bsf="inbag" gives pure inbag empirical risk minimization. If filter is supplied using tune.hcut and bsf is omitted, the stored tuning value is used.
keep.only	Character vector specifying the features of interest. The data is pre-filtered to keep only these requested variables. Ignored if filter is specified using tune.hcut.
fast	Use fast filtering?

pure.lasso	Logical value specifying whether lasso splitting should be strictly adhered to. In general, lasso splits are replaced with CART whenever numerical instability occurs (for example, small node sample sizes may make it impossible to obtain the cross-validated lasso parameter). This option will generally produce shallow trees which may not be appropriate in all settings.
eps	Parameter used by cdlasso.
maxit	Parameter used by cdlasso.
nfolds	Number of cross-validation folds to be used for the lasso.
block.size	Determines how cumulative error rate is calculated. To obtain the cumulative error rate on every nth tree, set the value to an integer between 1 and ntree.
bootstrap	Bootstrap protocol. Default is by.root which bootstraps the data by sampling with or without replacement (without replacement is the default; see the option samptype below). If none, the data is not bootstrapped (it is not possible to return OOB ensembles or prediction error in this case). If by.user, the bootstrap specified by samp is used.
samptype	Type of bootstrap used when by.root is in effect. Choices are swor (sampling without replacement; the default) and swr (sampling with replacement).
samp	Bootstrap specification when by.user is in effect. Array of dim n x ntree specifying how many times each record appears inbag in the bootstrap for each tree.
membership	Should terminal node membership and inbag information be returned?
sampsize	Function specifying bootstrap size when by.root is in effect. For sampling without replacement, it is the requested size of the sample, which by default is .632 times the sample size. For sampling with replacement, it is the sample size. Can also be specified using a number.
seed	Negative integer specifying seed for the random number generator.
do.trace	Number of seconds between updates to the user on approximate time to completion.
...	Further arguments passed to cdlasso and rfsrc.

Details

Super Greedy Trees (SGTs) are tree-based models that extend ordinary CART-style splitting in a fundamental way. In a standard tree, a split typically tests one variable at a time, so tilted, curved, or interaction-driven decision boundaries must be approximated by many small axis-aligned cuts. SGTs instead learn a sparse *score* inside a node and then split the node using that score. This allows a split to depend on several variables at once, so the fitted tree can represent hyperplane, elliptical, hyperboloid, and other higher-order geometric boundaries much more directly.

Operationally, the procedure is organized in stages. First, a family of candidate score functions is chosen through hcut. Second, within each node, a lasso model is fit by coordinate descent. Third, the fitted node-wise score is used to order observations, and an allowable threshold on that score is searched for the best split. The resulting daughter nodes are then re-fit and compared using empirical risk. In this way, the difficult multivariate split problem is converted into a manageable one-dimensional threshold search along a learned score.

Tree growth uses a *best split first* (BSF) strategy. Rather than expanding nodes in strict depth-first or breadth-first order, BSF scans the current terminal nodes, measures the reduction in empirical

risk for each candidate split, and grows the node giving the largest gain. This makes the search aggressive but focused: computation is directed to the part of the tree that is most promising at the current stage of growth. In a forest, repeating this over bootstrap samples yields an ensemble of trees with the flexibility of multivariate cuts and the stabilizing effect of aggregation.

The main user control for split geometry is `hcut`. Smaller values give simpler cut families; larger values allow richer polynomial and interaction structure. Thus `rfsgt` is able to span random-forest-like axis-aligned splits all the way to genuinely multivariate geometric partitioning. When the signal is approximately linear or only mildly nonlinear, smaller `hcut` values may be sufficient. When the signal involves curvature or interactions, larger `hcut` values can reduce bias and often achieve the same structural fit with fewer splits.

An equally important feature of SGTs is that the fitted forest is not only a prediction device. Because each split score and each terminal-node predictor is a sparse lasso model, the forest also carries local coefficient information. For a given observation, each tree contributes the coefficient vector from the terminal node containing that observation, and averaging across trees yields forest-level beta summaries that are usually more stable than the coefficients from any single tree. These beta values are local and data-adaptive: they can change from one region of the feature space to another, so they should be viewed as coefficient functions rather than a single global regression vector.

This gives SGTs a genuinely hybrid character. The partition of the feature space is nonparametric, as in a tree or forest, but within each local region the fitted response is represented by a sparse parametric expansion. Built-in helpers such as `get.beta` expose this structure by returning both beta summaries and corresponding partial term contributions. For a main effect, the contribution is the local beta multiplied by the covariate value; for an interaction, it is the local interaction coefficient multiplied by the associated product term. In many applications, these summaries can be as informative as the prediction itself because they show which variables, nonlinear terms, and interactions are driving the fitted value near the observation of interest.

Parametric models used for splitting are indexed by `hcut` corresponding to the following geometric regions:

1. `hcut`=1 (hyperplane) linear model using all variables.
2. `hcut`=2 (ellipse) plus all quadratic terms.
3. `hcut`=3 (oblique ellipse) plus all pairwise interactions.
4. `hcut`=4 plus all polynomials of degree 3 of two variables.
5. `hcut`=5 plus all polynomials of degree 4 of three variables.
6. `hcut`=6 plus all three-way interactions.
7. `hcut`=7 plus all four-way interactions.

Setting `hcut`=0 gives CART splits where cuts are parallel to the coordinate axis (axis-aligned cuts). Thus, `hcut`=0 is similar to random forests and can be viewed as the baseline case of the SGT framework.

A major part of the implementation is devoted to regularization and stabilization, because richer cut families can otherwise become unstable in small nodes or in the presence of collinearity. The first safeguard is the lasso itself. At each node, the split-defining score is fit with an ℓ_1 penalty, and the penalty is chosen by cross-validation. This induces sparsity, controls local complexity, and lets the procedure adapt to the amount of information available in the node. Near the root, where sample sizes are larger, the fitted score can support richer geometry. Deeper in the tree, lasso sparsity and smaller node sizes tend to simplify the split automatically.

A second safeguard is feature filtering. When the predictor dimension is moderate or large, the candidate dictionary implied by `hcut` can be very large. The implementation can therefore pre-filter variables using shallow pilot fits and retain only variables that appear with nonzero lasso coefficients. This reduces runtime and variance before the final forest is grown. The helper `tune.hcut` is the intended front-end for this step: it can pre-filter variables and also choose a suitable `hcut` value before the final call to `rfsgt`. In practice, this is often the safest workflow when interactions or higher-order terms are being considered.

A third safeguard is automatic simplification of a branch when the lasso modeling is no longer paying off. The algorithm then replaces the lasso-induced split by the best CART coordinate-threshold split at that node and in place of local lasso node estimators, simple CART-style sample-average predictors are used along that branch. In other words, in the presence of numerical instability, the procedure can simplify both the split geometry and the local node model, which prevents unstable or unnecessary parametric structure from being pushed deeper into the tree.

The argument `pure.lasso` controls whether this simplification is allowed. With the default behavior, a branch may switch from the richer lasso-based representation to a simpler CART-style representation when the latter is more stable or gives better empirical risk reduction. Setting `pure.lasso=TRUE` shuts this switch off. This is useful when a user wants a fully lasso-defined tree for methodological reasons, but in difficult data settings it can also lead to shallower trees because branches that would otherwise be stabilized by CART are instead left unsplit.

The argument `bsf`, although related, addresses a different question. It does not turn CART fallback on or off. Instead, it decides which data drive empirical risk minimization during BSF search. With `bsf="inbag"`, the split comparison is based on in-bag data. With `bsf="oob"`, the same comparison uses out-of-bag (OOB) data as a held-out check. This indirectly has implications when deciding on CART fallback, however, because it allows the algorithm to use out of sample data to decide whether the current SGT candidate really improves generalization relative to the best CART candidate at the same node. If the CART candidate wins under the chosen risk criterion, the branch is simplified exactly as described above: CART split geometry is used and CART-style node estimators replace the local lasso fits down that branch. This guard is especially helpful under potentially numerically unstable models when `hcut > 0`. As with any model-selection procedure that reuses held-out data, however, users should remember that the reported OOB error can then be mildly optimistic.

From a practical point of view, the main tuning parameters have clear roles. Use `hcut` to control cut richness, `treesize` and `nodesize` to control tree complexity, `filter` or `tune.hcut` when the feature space is large, and `pure.lasso` only when you explicitly want to over-ride CART fallback. Users new to the method can usually start with the defaults. Users working with interaction-heavy or high-dimensional data will typically benefit from filtering, `tune.hcut`, and OOB-based BSF. Users interested mainly in prediction can simply call `predict.rfsgt`. Users interested in the local parametric structure can query the same fitted forest with `get.beta` to recover beta summaries and partial contributions without fitting a separate surrogate model.

Value

An object of class `c("rfsgt", "grow", family)` containing the trained super greedy forest and associated training-data summaries. The object is a list with the following components:

`family` Model family.

`n` Number of observations in the training data.

`bootstrap` Bootstrap protocol used to grow the forest.

`samptype` Sampling type used for root-node bootstrap samples.

`sampsize` Tree sample size used by the bootstrap protocol.

`ntree` Number of trees grown.

`hcut` Final `hcut` value used to construct the splitting model.

`splitrule` Split rule used by the forest.

`splitinfo` Internal split-rule metadata, including the processed `hcut`, split-rule index, number of random split points, and lasso cross-validation setting.

`yvar` Training response values.

`yvar.names` Response variable name.

`yvar.factor` Factor-level metadata for the response.

`yvar.types` Internal response type code.

`xvar` Training predictor data after hot encoding, optional filtering, and any user-requested variable restriction.

`xvar.augment` Augmented base-learner data used for higher-order `hcut` terms, or `NULL` when no augmented terms are used.

`xvar.names` Names of the retained predictor variables.

`xvar.types` Internal predictor type codes.

`xvar.info` Predictor-encoding metadata used internally; this is `NULL` for standard grow objects.

`xvar.augment.names` Names of augmented base-learner terms, or `NULL` when no augmented terms are used.

`term.map` Term map describing how generated base-learner terms correspond to original variables and powers.

`forest` Stored forest object used by `predict.rfsgt`. This component contains the native node array, per-tree leaf counts, terminal-node offsets, bootstrap membership identifiers, fitting options, and package-version metadata.

`nodeStat` Node statistics returned when empirical-risk output is available; otherwise `NULL`.

`empr.risk` Inbag empirical-risk values by candidate tree size and tree, or `NULL` when unavailable.

`oob.empr.risk` Out-of-bag empirical-risk values by candidate tree size and tree, or `NULL` when unavailable.

`empr.risk.cart` Inbag empirical-risk values for CART fallback splits, or `NULL` when unavailable.

`oob.empr.risk.cart` Out-of-bag empirical-risk values for CART fallback splits, or `NULL` when unavailable.

`bsf` Best-split-first empirical-risk strategy used.

`bsf.order` Best-split-first node expansion order, or `NULL` when unavailable.

`predicted` Training-data ensemble predictions.

`predicted.oob` Out-of-bag training-data predictions when available; otherwise `NULL`.

`membership` Matrix of terminal-node membership by observation and tree when `membership = TRUE`; otherwise `NULL`.

`inbag` Matrix of bootstrap counts by observation and tree when `membership = TRUE`; otherwise `NULL`.

`ensemble` Internal ensemble type used for prediction summaries.

`err.rate` Cumulative prediction error, typically the out-of-bag mean squared error for regression when available; otherwise `NULL`.

`ctime.internal` Timing information reported by the native grow routine.

`ctime.external` Elapsed R-side timing, computed from `proc.time()`.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H. (2025). Super greedy trees. To appear in Artificial Intelligence Review.

See Also

`cdlasso`, `predict.rfsgt`

Examples

```
## ------------------------------------------------------------
##
##  mtcars: for CRAN testing
##
## ------------------------------------------------------------

print(rfsgt(mpg~., mtcars, ntree=3, treesize=1))




## ------------------------------------------------------------
##
##  boston housing
##
## ------------------------------------------------------------

if (requireNamespace("mlbench", quietly = TRUE)) {

## load the data
data(BostonHousing, package = "mlbench")

## default basic call
print(rfsgt(medv~., BostonHousing))

## variable selection
sort(vimp.rfsgt(medv~.,BostonHousing))

## examples of hcut=0 (similar to random forests ... but using BSF)
print(rfsgt(medv~., BostonHousing, hcut=0))
```

```
print(rfsgt(medv~., BostonHousing, hcut=0, nodesize=1))

## hcut=1 with smaller nodesize
print(rfsgt(medv~., BostonHousing, nodesize=1))

## ------------------------------------------------------------
##
## boston housing with factors
##
## ------------------------------------------------------------

## load the data
data(BostonHousing, package = "mlbench")

## make some features into factors
Boston <- BostonHousing
Boston$zn <- factor(Boston$zn)
Boston$chas <- factor(Boston$chas)
Boston$lstat <- factor(round(0.2 * Boston$lstat))
Boston$nox <- factor(round(20 * Boston$nox))
Boston$rm <- factor(round(Boston$rm))

## random forest: hcut=0
print(rfsgt(medv~., Boston, hcut=0, nodesize=1))

## hcut=3
print(rfsgt(medv~., Boston, hcut=3))


## ------------------------------------------------------------
##
##  ozone
##
## ------------------------------------------------------------

## load the data
data(Ozone, package = "mlbench")

print(rfsgt(V4~., na.omit(Ozone), hcut=0, nodesize=1))
print(rfsgt(V4~., na.omit(Ozone), hcut=1))
print(rfsgt(V4~., na.omit(Ozone), hcut=2))
print(rfsgt(V4~., na.omit(Ozone), hcut=3))

}

## ------------------------------------------------------------
##
##  non-linear boundary illustrates hcut using single tree
##
## ------------------------------------------------------------

## simulate non-linear boundary
n <- 500
```

```
p <- 5
signal <- 10
treesize <- 10
ngrid <- 200

## train
x <- matrix(runif(n * p), ncol = p)
fx <- signal * sin(pi * x[, 1] * x[, 2])
nl2d <- data.frame(y = fx, x)

## truth
x1 <- x2 <- seq(0, 1, length = ngrid)
truth <- signal * sin(outer(pi * x1, x2, "*"))

## test
x.tst <- do.call(rbind, lapply(x1, function(x1j) {
  cbind(x1j, x2, matrix(runif(length(x2) * (p-2)), ncol=(p-2)))
}))
colnames(x.tst) <- colnames(x)
fx.tst <- signal * sin(pi * x.tst[, 1] * x.tst[, 2])
nl2d.tst <- data.frame(y = fx.tst, x.tst)

## SGT for different hcut values
## Enforce pure lasso
rO <- lapply(0:4, function(hcut) {
  cat("hcut", hcut, "\n")
  rfsgt(y~., nl2d, ntree=1, hcut=hcut, treesize=treesize, bootstrap="none",
         pure.lasso = TRUE, nodesize=1, filter=FALSE)
})

## nice little wrapper for plotting results
if (library("interp", logical.return = TRUE)) {
  ## nice little wrapper for plotting results
  plot.image <- function(x, y, z, linear=TRUE, nlevels=40, points=FALSE) {
    xo <- x; yo <- y
    so <- interp(x=x, y=y, z=z, linear=linear, nx=nlevels, ny=nlevels)
    x <- so$x; y <- so$y; z <- so$z
    xlim <- ylim <- range(c(x, y), na.rm = TRUE, finite = TRUE)
    z[is.infinite(z)] <- NA
    zlim <- q <- quantile(z, c(.01, .99), na.rm = TRUE)
    z[z<=q[1]] <- q[1]
    z[z>=q[2]] <- q[2]
    levels <- pretty(zlim, nlevels)
    col <- hcl.colors(length(levels)-1, "YlOrRd", rev = TRUE)
    plot.new()
    plot.window(xlim, ylim, "", xaxs = "i", yaxs = "i", asp = NA)
    .filled.contour(x, y, z, levels, col)
    axis(1);axis(2)
    if (points)
      points(xo,yo ,pch=16, cex=.25)
    box()
    invisible()
  }
```

```
  oldpar <- par(mfrow=c(3,2))
  image(x1, x2, truth, xlab="", ylab="")
  contour(x1, x2, truth, nlevels = 15, add = TRUE, drawlabels = FALSE)
  mtext(expression(x[1]),1,line=2)
  mtext(expression(x[2]),2,line=2)
  mtext(expression("truth"),3,line=1)

  pO <- lapply(0:4, function(j) {
    plot.image(nl2d.tst[,"X1"],nl2d.tst[,"X2"], predict(rO[[j+1]], nl2d.tst)$predicted)
    contour(x1, x2, truth, nlevels = 15, add = TRUE, drawlabels = FALSE)
    mtext(expression(x[1]),1,line=2)
    mtext(expression(x[2]),2,line=2)
    mtext(paste0("hcut=",j),3,line=1)
    NULL
  })

  par(oldpar)


}

## ------------------------------------------------------------
##
##  friedman illustration of OOB empirical risk
##
## ------------------------------------------------------------

if (requireNamespace("mlbench", quietly = TRUE)) {

## simulate friedman
n <- 500
dta <- data.frame(mlbench::mlbench.friedman1(n, sd=0))

## rf versus rfsgt
o1 <- rfsgt(y~., dta, hcut=0, block.size=1)
o2 <- rfsgt(y~., dta, hcut=3, block.size=1)

## compute running average of OOB empirical risk
runavg <- function(x, lag = 8) {
  x <- c(na.omit(x))
  lag <- min(lag, length(x))
  cx <- c(0,cumsum(x))
  rx <- cx[2:lag] / (1:(lag-1))
  c(rx, (cx[(lag+1):length(cx)] - cx[1:(length(cx) - lag)]) / lag)
}
risk1 <- lapply(data.frame(o1$oob.empr.risk), runavg)
leaf1 <- o1$forest$leafCount
risk2 <- lapply(data.frame(o2$oob.empr.risk),  runavg)
leaf2 <- o2$forest$leafCount

## compare OOB empirical tree risk to OOB forest error
oldpar <- par(mfrow=c(2,2))
```

```
plot(c(1,max(leaf1)), range(c(risk1)), type="n",
     xlab="Tree size", ylab="RF OOB empirical risk")
l1 <- do.call(rbind, lapply(risk1, function(rsk){
  lines(rsk,col=grey(0.8))
  cbind(1:length(rsk), rsk)
}))
lines(tapply(l1[,2], l1[,1], mean), lwd=3)


plot(c(1,max(leaf2)), range(c(risk2)), type="n",
     xlab="Tree size", ylab="SGF OOB empirical risk")
l2 <- do.call(rbind, lapply(risk2, function(rsk){
  lines(rsk,col=grey(0.8))
  cbind(1:length(rsk), rsk)
}))
lines(tapply(l2[,2], l2[,1], mean), lwd=3)

plot(1:o1$ntree, o1$err.rate, type="s", xlab="Trees", ylab="RF OOB error")

plot(1:o2$ntree, o2$err.rate, type="s", xlab="Trees", ylab="SGF OOB error")

par(oldpar)

}

## ------------------------------------------------------------
##
## synthetic regression examples with different hcut
##
## ------------------------------------------------------------

if (requireNamespace("mlbench", quietly = TRUE)) {

## simulation functions
sim <- list(
     friedman1=function(n){data.frame(mlbench::mlbench.friedman1(n))},
     friedman2=function(n){data.frame(mlbench::mlbench.friedman2(n))},
     friedman3=function(n){data.frame(mlbench::mlbench.friedman3(n))},
     peak=function(n){data.frame(mlbench::mlbench.peak(n, 10))},
     linear=function(n, sd=.1){
           x=matrix(runif(n*10), n)
           y=3*x[,1]^3-2*x[,2]^2+3*x[,3]+rnorm(n,sd=sd)
           data.frame(y=y,x)
})

## run rfsgt on the simulations
n <- 500
max.hcut <- 3
results <- setNames(lapply(names(sim), function(nm) {
  cat("simulation:", nm, "\n")
  d <- sim[[nm]](n=n)
  r0 <- data.frame(do.call(rbind, lapply(0:max.hcut, function(hcut) {
```

```
   cat("     hcut:", hcut, "\n")
   o <- rfsgt(y~.,d,hcut=hcut)
   c(hcut, tail(o$err.rate, 1), tail(o$err.rate, 1) / var(o$yvar))
  }))
  colnames(r0) <- c("hcut", "mse", "smse")
  r0
}), names(sim))

## print results
print(results)

## ------------------------------------------------------------
##
## synthetic regression example showing how to tune hcut
##
## ------------------------------------------------------------

hcut.opt <- setNames(sapply(names(sim), function(nm) {
  cat("optimize hcut for simulation:", nm, "\n")
  f <- tune.hcut(y~., sim[[nm]](n=n), hcut=4)
  attr(f, "hcut")
}), names(sim))

## print the optimal hcut
print(hcut.opt)

}

## ------------------------------------------------------------
##
## iowa housing data
##
## ------------------------------------------------------------

data(housing, package = "randomForestSRC")

## remove PID
housing$PID <- NULL

## rough missing data imputation
d <- randomForestSRC::impute(data = data.frame(data.matrix(housing)))

d$SalePrice <- log(d$SalePrice)
d <- data.frame(data.matrix(d))

print(rfsgt(SalePrice~.,d))

## ------------------------------------------------------------
##
## high-dimensional model with variable selection
##
## ------------------------------------------------------------
```

```
## simulate big p small n data
n <- 50
p <- 500
d <- data.frame(y = rnorm(n), x = matrix(rnorm(n * p), n))

## we have a big p small n pure noise setting: let's see how well we do
cat("variables selected by vimp.rfsgt:\n")
vmp <- sort(vimp.rfsgt(y~.,d))
print(vmp[vmp > .05])

## internal filtering function can also be used
cat("variables selected by filter.rfsgt:\n")
print(filter.rfsgt(y~.,d, method="conserve"))

## ------------------------------------------------------------
##
## pre-filtering using keep.only
##
## ------------------------------------------------------------

if (requireNamespace("mlbench", quietly = TRUE)) {

## simulate the data
n <- 100
p <- 50
noise <- matrix(runif(n * p), ncol=p)
dta <- data.frame(mlbench::mlbench.friedman1(n, sd=0), noise=noise)

## filter the variables
f <- filter.rfsgt(y~., dta)

## use keep.only to pre-filter the features
print(rfsgt(y~.,dta, keep.only=f, hcut=1))
print(rfsgt(y~.,dta, keep.only=f, hcut=2))
print(rfsgt(y~.,dta, keep.only=f, hcut=3))


## ------------------------------------------------------------
##
## tuning hcut and pre-filtering using tune.hcut
##
## ------------------------------------------------------------

## simulate the data
n <- 100
p <- 50
noise <- matrix(runif(n * p), ncol=p)
dta <- data.frame(mlbench::mlbench.friedman1(n, sd=0), noise=noise)

## tune hcut
f <- tune.hcut(y~., dta, hcut=3)

## use the optimized hcut
```

```
print(rfsgt(y~.,dta, filter=f))

## over-ride the tuned hcut value
print(rfsgt(y~.,dta, filter=use.tune.hcut(f, hcut=1)))
print(rfsgt(y~.,dta, filter=use.tune.hcut(f, hcut=2)))
print(rfsgt(y~.,dta, filter=use.tune.hcut(f, hcut=3)))


## ------------------------------------------------------------
##
## get local beta values and partial contributions
##
## SGTs are not only predictive; the same fit can be queried
## for local parametric summaries.  We use friedman 1 for
## illustration.
##
## ------------------------------------------------------------

n <- 100
dta <- data.frame(mlbench::mlbench.friedman1(n))
o <- rfsgt(y~., dta, hcut=3, pure.lasso=TRUE, treesize=10)
bO <- get.beta(o)
print(str(bO$beta))
print(str(bO$partial))

}
```

`rfsgt.news` *Show the NEWS file*

Description

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

Usage

```
rfsgt.news(...)
```

Arguments

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

Value

None.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

Index