Package: randomForestSGT (via r-universe)

June 5, 2026

Version 0.0.1.74

Date 2025-07-16

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 (>= 3.6.0),

Imports randomForestSRC (>= 3.3.8), varPro (>= 0.0.73)

Suggests mlbench, interp, glmnet

Description Super greedy trees (SGTs) with univariate and multivariate geometric cuts are obtained using lasso and coordinate descent.

License GPL (>=3)

URL https://ishwaran.org/ https://kogalur.com/

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

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

Date/Publication 2025-07-16 17:05:59 UTC

RemoteUrl https://github.com/kogalur/randomforestsgt

RemoteRef HEAD

RemoteSha d22825ab1663bb81377fcd666b7b533999dcce84

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

Lasso solution path with the following values.

`beta`	Matrix containing beta values for the lasso solution path.
`lambda`	The sequence of `lambda` values used.
`lambda.min.indx`	Index for value of `lambda` that gives the minimum cross-validation error. Only applies if `nfolds` is greater than 1.
`lambda.1se.min.indx`	Index for minimum `lambda` value within 1 standard error of the minimum cross-validation error. This is more liberal. Only applies if `nfolds` is greater than 1.
`lambda.1se.max.indx`	Index for maximum `lambda` value within 1 standard error of the minimum cross-validation error. This is more conservative. Only applies if `nfolds` is greater than 1.

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

```
## 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.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H. (2025). Super greedy trees.

See Also

rfsgt

Examples

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

## 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, ...)
```

Arguments

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

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.

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 NULL,
      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. When lasso is enabled (hcut>0), using OOB (out-of-bag) improves robustness and is the default. However, this can optimistically bias OOB error estimates.
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 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

A flexible class of parametric models are used for tree splitting using lasso. This includes CART splits, hyperplane, ellipsoid and hyperboloid cuts. Coordinate descent is used for fast calculation of the penalized lasso parametric models. Cross-validation is employed to obtain the lasso regularization parameter.

These trees are called super greedy trees (SGTs) and are constructed using best split first (BSF) where cuts are made sequentially in order of greatest empirical risk reduction.

Parametric linear models used for splitting are indexed by parameter 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.

Value

A forest of SGTs trained on the learning data which can be used for prediction.

Author(s)

Hemant Ishwaran and Udaya B. Kogalur

References

Ishwaran H. (2025). Super greedy trees.

See Also

cdlasso

Examples

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

## 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
rO <- lapply(0:4, function(hcut) {
  cat("hcut", hcut, "\n")
  rfsgt(y~., nl2d, ntree=1, hcut=hcut, treesize=treesize, bootstrap="none",
         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()
  }

  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)

  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)
  })


}

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

## 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
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")


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

## 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 <- data.frame(randomForestSRC:::get.na.roughfix(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
##
## ------------------------------------------------------------

## 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)))
```

`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