Package 'drf'

May 8, 2026

Title Distributional Random Forests

Version 1.3.1

BugReports https://github.com/lorismichel/drf/issues

Description An implementation of distributional random forests as introduced in Cevid & Michel & Naf & Meinshausen & Buhlmann (2022) <doi:10.48550/arXiv.2005.14458>.

License GPL-3

LinkingTo Rcpp, RcppEigen

Depends R (>= 3.4.4)

Imports stats, fastDummies, Matrix, methods, Rcpp (>= 0.12.15)

RoxygenNote 7.3.3

Suggests DiagrammeR

SystemRequirements GNU make

URL https://github.com/lorismichel/drf

NeedsCompilation yes

Author Jeffrey Naf [cre],
Loris Michel [aut],
Domagoj Cevid [aut]

Maintainer Jeffrey Naf <jeffrey.naf@unige.ch>

Repository CRAN

Date/Publication 2026-02-03 08:30:02 UTC

Contents

`drf` *Distributional Random Forests*

Description

Trains a Distributional Random Forest which estimates the full conditional distribution $P(Y|X)$ for possibly multivariate response Y and predictors X. The conditional distribution estimate is represented as a weighted distribution of the training data. The weights can be conveniently used in the downstream analysis to estimate any quantity of interest $\tau(P(Y|X))$.

Usage

```
drf(
  X,
  Y,
  num.trees = 3000,
  splitting.rule = "FourierMMD",
  num.features = 10,
  bandwidth = NULL,
  response.scaling = TRUE,
  node.scaling = FALSE,
  sample.weights = NULL,
  sample.fraction = 0.5,
  mtry = min(ceiling(sqrt(ncol(X)) + 20), ncol(X)),
  min.node.size = 15,
  honesty = TRUE,
  honesty.fraction = 0.5,
  honesty.prune.leaves = TRUE,
  alpha = 0.05,
  imbalance.penalty = 0,
  compute.oob.predictions = FALSE,
  num.threads = NULL,
  seed = stats::runif(1, 0, .Machine$integer.max),
  compute.variable.importance = FALSE,
  ci.group.size = as.integer(num.trees/30)
)
```

Arguments

X	The covariates used in the regression. Can be either a numeric matrix or a data.frame with numeric, factor, or character columns, where the last two will be one-hot-encoded.
Y	The (multivariate) outcome variable. Needs to be a matrix or a data frame consisting of numeric values.
num.trees	Number of trees grown in the forest. Default is 3000.
splitting.rule	A character value. The type of the splitting rule used, can be either "FourierMMD" (MMD splitting criterion with FastMMD approximation for speed) or "CART" (sum of standard CART criteria over the components of Y).
num.features	A numeric value, in case of "FourierMMD", the number of random features to sample.
bandwidth	A numeric value, the bandwidth of the Gaussian kernel used in case of "FourierMMD", the value is set to NULL by default and the square root of the median heuristic is used.
response.scaling	A boolean value, should the responses be standardized before fitting the forest. Default is TRUE.
node.scaling	A boolean value, should the responses be standardized in every node of every tree. Default is FALSE.
sample.weights	(experimental) Weights given to an observation in estimation. If NULL, each observation is given the same weight. Default is NULL.
sample.fraction	Fraction of the data used to build each tree. Note: If honesty = TRUE, these subsamples will further be cut by a factor of honesty.fraction. Default is 0.5.
mtry	Number of variables tried for each split. Default is $\sqrt{p} + 20$, where p is the number of predictors.
min.node.size	A target for the minimum number of observations in each tree leaf. Note that nodes with size smaller than min.node.size can occur, as in the original randomForest package. Default is 5.
honesty	Whether to use honest splitting (i.e., sub-sample splitting). Default is TRUE. For a detailed description of honesty, honesty.fraction, honesty.prune.leaves, and recommendations for parameter tuning, see the GRF reference for more information (the original source).
honesty.fraction	The fraction of data that will be used for determining splits if honesty = TRUE. Default is 0.5 (i.e. half of the data is used for determining splits and the other half for populating the nodes of the tree).
honesty.prune.leaves	If TRUE, prunes the estimation sample tree such that no leaves are empty. If FALSE, keeps the same tree as determined in the splits sample (if an empty leave is encountered, that tree is skipped and does not contribute to the estimate). Setting this to FALSE may improve performance on small/marginally powered data, but requires more trees (note: tuning does not adjust the number of trees). Only applies if honesty is enabled. Default is TRUE.

`alpha` A tuning parameter that controls the maximum imbalance of a split. Default is 0.05, meaning a child node will contain at most 5% of observations in the parent node.

`imbalance.penalty` A tuning parameter that controls how harshly imbalanced splits are penalized. Default is 0.

`compute.oob.predictions` Whether OOB predictions on training set should be precomputed.

`num.threads` Number of threads used in training. By default, the number of threads is set to the maximum hardware concurrency.

`seed` The seed of the C++ random number generator.

`compute.variable.importance` boolean, should the variable importance be computed in the object.

`ci.group.size` The forest will grow ci.group.size trees on each subsample. In order to provide confidence intervals, ci.group.size must be at least 2. Defaults to `num.trees/30` which yields 30 CI groups.

Value

A trained Distributional Random Forest object.

See Also

See `predict.drf` for how to make predictions, including uncertainty weights.

Examples

```
library(drf)

n = 500
p = 10
d = 3

# Generate training data
X = matrix(rnorm(n * p), nrow=n)
Y = matrix(rnorm(n * d), nrow=n)
Y[, 1] = Y[, 1] + X[, 1]
Y[, 2] = Y[, 2] * X[, 2]
Y[, 3] = Y[, 3] * X[, 1] + X[, 2]

# Fit DRF object
drf.forest = drf(X, Y, num.trees=100)

# Generate test data
X_test = matrix(rnorm(10 * p), nrow=10)

out = predict(drf.forest, newdata=X_test)
# Compute E[Y_1 | X] for all data in X_test directly from
# the weights representing the estimated distribution
out$weights %*% out$y[,1]
```

```
out = predict(drf.forest, newdata=X_test,
              functional='mean')
# Compute E[Y_1 | X] for all data in X_test using built-in functionality
out[,1]

out = predict(drf.forest, newdata=X_test,
              functional='quantile',
              quantiles=c(0.25, 0.75),
              transformation=function(y){y[1] * y[2] * y[3]})
# Compute 25% and 75% quantiles of Y_1*Y_2*Y_3, conditionally on X = X_test[1, ]
out[1,,]

out = predict(drf.forest, newdata=X_test,
              functional='cov',
              transformation=function(y){matrix(1:6, nrow=2) %*% y})
# Compute 2x2 covariance matrix for (1*Y_1 + 3*Y_2 + 5*Y_3, 2*Y_1 + 4*Y_2 + 6*Y_3),
# conditionally on X = X_test[1, ]
out[1,,]

out = predict(drf.forest, newdata=X_test,
              functional='custom',
              custom.functional=function(y, w){c(sum(y[, 1] * w), sum(y[, 2] * w))})
# Compute E[Y_1, Y_2 | X] for all data in X_test by providing custom functional that
# computes it from the weights
out
```

`get_sample_weights` *Given a trained forest and test data, compute the training sample weights for each test point.*

Description

During normal prediction, these weights are computed as an intermediate step towards producing estimates. This function allows for examining the weights directly, so they could be potentially be used as the input to a different analysis.

Usage

```
get_sample_weights(
  forest,
  newdata = NULL,
  estimate.uncertainty = FALSE,
  num.threads = NULL
)
```

Arguments

forest	The trained forest.
newdata	Points at which predictions should be made. If NULL, makes out-of-bag predictions on the training set instead (i.e., provides predictions at Xi using only trees that did not use the i-th training example).
estimate.uncertainty	Whether to return a single weight for each sample or return B weight vectors calculated on B CI groups for each sample. See Details and return value docu.
num.threads	Number of threads used in training. If set to NULL, the software automatically selects an appropriate amount.

Details

To estimate the uncertainty, a set of B=(num.trees)/(ci.group.size) weights is produced for each sample when estimate.uncertainty=TRUE. These B weights arise from B subforests (CI groups) inside the estimation routine and may be seen as bootstrap approximation to the estimation uncertainty of the DRF estimator. As such, they can be used to build confidence intervals for functionals. For instance, for univariate functionals, one may calculate one functional per weight to obtain B estimates, with which the variance can be calculated. Then the usual normal approximation can be used to construct confidence intervals for said functional. Uncertainty weights are not available OOB.

Value

estimate.uncertainty=FALSE	A sparse matrix where each row represents a test sample, and each column is a sample in the training data. The value at (i, j) gives the weight of training sample j for test sample i.
estimate.uncertainty=TRUE	A list of length nrow(test sample) where each item is a B x w sparse matrix, where B is the number of CI groups and w=nrow(Y). This matrix essentially contains B separate weight vectors, one in each row.

Examples

```
## Not run:
p <- 10
n <- 100
X <- matrix(2 * runif(n * p) - 1, n, p)
Y <- (X[, 1] > 0) + 2 * rnorm(n)
rrf <- drf(X, matrix(Y,ncol=1), mtry = p)
sample.weights.oob <- get_sample_weights(rrf)

n.test <- 15
X.test <- matrix(2 * runif(n.test * p) - 1, n.test, p)
sample.weights <- get_sample_weights(rrf, X.test)

## End(Not run)
```

get_tree *Retrieve a single tree from a trained forest object.*

Description

Retrieve a single tree from a trained forest object.

Usage

```
get_tree(forest, index)
```

Arguments

forest	The trained forest.
index	The index of the tree to retrieve.

Value

A DRF tree object containing the below attributes. drawn_samples: a list of examples that were used in training the tree. This includes examples that were used in choosing splits, as well as the examples that populate the leaf nodes. Put another way, if honesty is enabled, this list includes both subsamples from the split (J1 and J2 in the notation of the paper). num_samples: the number of examples used in training the tree. nodes: a list of objects representing the nodes in the tree, starting with the root node. Each node will contain an 'is_leaf' attribute, which indicates whether it is an interior or leaf node. Interior nodes contain the attributes 'left_child' and 'right_child', which give the indices of their children in the list, as well as 'split_variable', and 'split_value', which describe the split that was chosen. Leaf nodes only have the attribute 'samples', which is a list of the training examples that the leaf contains. Note that if honesty is enabled, this list will only contain examples from the second subsample that was used to 'repopulate' the tree (J2 in the notation of the paper).

Examples

```
## Not run:
# Train a quantile forest.
n <- 50
p <- 10
X <- matrix(rnorm(n * p), n, p)
Y <- X[, 1] * rnorm(n)
q.forest <- quantile_forest(X, Y, quantiles = c(0.1, 0.5, 0.9))

# Examine a particular tree.
q.tree <- get_tree(q.forest, 3)
q.tree$nodes

## End(Not run)
```

`leaf_stats.default` *A default leaf_stats for forests classes without a leaf_stats method that always returns NULL.*

Description

A default leaf_stats for forests classes without a leaf_stats method that always returns NULL.

Usage

```
## Default S3 method:
leaf_stats(forest, samples, ...)
```

Arguments

`forest`	Any forest
`samples`	The samples to include in the calculations.
`...`	Additional arguments (currently ignored).

`leaf_stats.drf` *Calculate summary stats given a set of samples for regression forests.*

Description

Calculate summary stats given a set of samples for regression forests.

Usage

```
## S3 method for class 'drf'
leaf_stats(forest, samples, ...)
```

Arguments

`forest`	The GRF forest
`samples`	The samples to include in the calculations.
`...`	Additional arguments (currently ignored).

Value

A named vector containing summary stats

medianHeuristic *Compute the median heuristic for the MMD bandwidth choice*

Description

Compute the median heuristic for the MMD bandwidth choice

Usage

```
medianHeuristic(Y)
```

Arguments

Y	the response matrix

Value

the median heuristic

plot.drf_tree *Plot a DRF tree object.*

Description

Plot a DRF tree object.

Usage

```
## S3 method for class 'drf_tree'
plot(x, ...)
```

Arguments

x	The tree to plot
...	Additional arguments (currently ignored).

predict.drf *Predict from Distributional Random Forests object*

Description

Obtain predictions from a DRF forest object. For any point x in the predictor space, it returns the estimate of the conditional distribution $P(Y|X = x)$ represented as a weighted distribution $\sum_i w_i y_i$ of the training observations y_i. Additionally, this function also provides support to directly obtain estimates of certain target quantities $\tau(P(Y|X))$, such as e.g. conditional quantiles, variances or correlations.

Usage

```
## S3 method for class 'drf'
predict(
  object,
  newdata = NULL,
  functional = NULL,
  transformation = NULL,
  custom.functional = NULL,
  num.threads = NULL,
  estimate.uncertainty = FALSE,
  ...
)
```

Arguments

object	Trained DRF forest object.
newdata	Points at which predictions should be made. If NULL, returns out-of-bag predictions on the training set (i.e., for every training point X_i, provides predictions using only trees which did not use this point for tree construction). Can be either a data frame, matrix or a vector. Each row represents a data point of interest and the number and ordering of columns is assumed the be the same as in the training set.
functional	Optional. String indicating the statistical functional that we want to compute from the weights. One option between: **"mean"** - Conditional mean, the returned value is a matrix mean of dimension n x f, where n denotes the number of observations in newdata and f the dimension of the transformation. **"sd"** - Conditional standard deviation for each component of the (transformed) response, the returned value is a matrix of dimension n x f, where n denotes the number of observations in newdata and f the dimension of the transformation. **"quantile"** - Conditional quantiles. It requires additional parameter quantiles containing the list of quantile levels we want to compute. The returned value is an array of dimension n x f x q, where n denotes the number of

observations in newdata, f the dimension of the transformation and q the number of desired quantiles.

"cor" - Conditional correlation matrix, the returned value is an array of dimension n x f x f, where n denotes the number of observations in newdata and f the dimension of the transformation.

"cov" - Conditional covariance matrix, the returned value is an array of dimension n x f x f, where n denotes the number of observations in newdata, f the dimension of the transformation.

"custom" - A custom function provided by the user, the returned value is a matrix of dimension n x f, where n denotes the number of observations in newdata and f the dimension of the output of the function custom.functional provided by the user.

`transformation` An optional transformation function that is applied to the responses before computing the target functional. It helps to extend the functionality to a much wider range of targets. The responses are not transformed by default, i.e. the identity function $f(y) = y$ is used.

`custom.functional`
A user-defined function when functional is set to "custom". This should be a function f(y,w) which for a single test point takes the n x f matrix y and the corresponding n-dimensional vector of weights w and returns the quantity of interest given as a list of values. n denotes the number of training observations and f the dimension of the transformation.

`num.threads` Number of threads used for computing. If set to NULL, the software automatically selects an appropriate amount.

`estimate.uncertainty`
Whether to additionally return B weight vectors calculated on B CI groups for each sample. See Details and return value docu.

`...` additional parameters.

Details

To estimate the uncertainty, a set of B=(num.trees)/(ci.group.size) weights is produced for each sample when estimate.uncertainty=TRUE. These B weights arise from B subforests (CI groups) inside the estimation routine and may be seen as bootstrap approximation to the estimation uncertainty of the DRF estimator. As such, they can be used to build confidence intervals for functionals. For instance, for univariate functionals, one may calculate one functional per weight to obtain B estimates, with which the variance can be calculated. Then the usual normal approximation can be used to construct confidence intervals for said functional. Uncertainty weights are not available OOB.

Value

If functional is NULL, returns a list containing

`y` the matrix of training responses

`weights` the matrix of weights, whose number of rows corresponds the number of rows of newdata and the number of columns corresponds to the number of training data points.

If `estimate.uncertainty=TRUE`, additionally

`weights.uncertainty`
: a list of length `nrow(newdata)` where each item is a `B x w` sparse matrix, where `B` is the number of CI groups and `w=nrow(Y)`. This matrix essentially contains `B` separate weight vectors, one in each row.

If `functional` is specified, the desired quantity is returned, in the format described above.

Examples

```
library(drf)

n = 500
p = 10
d = 3

# Generate training data
X = matrix(rnorm(n * p), nrow=n)
Y = matrix(rnorm(n * d), nrow=n)
Y[, 1] = Y[, 1] + X[, 1]
Y[, 2] = Y[, 2] * X[, 2]
Y[, 3] = Y[, 3] * X[, 1] + X[, 2]

# Fit DRF object
drf.forest = drf(X, Y, num.trees=100, num.threads=1)

# Generate test data
X_test = matrix(rnorm(10 * p), nrow=10)

out = predict(drf.forest, newdata=X_test)
# Compute E[Y_1 | X] for all data in X_test directly from
# the weights representing the estimated distribution
out$weights %*% out$y[,1]

out = predict(drf.forest, newdata=X_test,
              functional='mean')
# Compute E[Y_1 | X] for all data in X_test using built-in functionality
out[,1]

out = predict(drf.forest, newdata=X_test,
              functional='quantile',
              quantiles=c(0.25, 0.75),
              transformation=function(y){y[1] * y[2] * y[3]})
# Compute 25% and 75% quantiles of Y_1*Y_2*Y_3, conditionally on X = X_test[1, ]
out[1,,]

out = predict(drf.forest, newdata=X_test,
              functional='cov',
              transformation=function(y){matrix(1:6, nrow=2) %*% y})
# Compute 2x2 covariance matrix for (1*Y_1 + 3*Y_2 + 5*Y_3, 2*Y_1 + 4*Y_2 + 6*Y_3),
# conditionally on X = X_test[1, ]
out[1,,]
```

```
out = predict(drf.forest, newdata=X_test,
              functional='custom',
              custom.functional=function(y, w){c(sum(y[, 1] * w), sum(y[, 2] * w))})
# Compute E[Y_1, Y_2 | X] for all data in X_test by providing custom functional that
# computes it from the weights
out

## UNCERTAINTY WEIGHTS ########################################################

# Simulate Data that experiences both a mean as well as sd shift
set.seed(10)
n<-500

# Simulate from X
x1 <- runif(n,-1,1)
x2 <- runif(n,-1,1)
x3 <- x1+ runif(n,-1,1)
X0 <- matrix(runif(7*n,-1,1), nrow=n, ncol=7)
X <- cbind(x1,x2, x3, X0)
colnames(X)<-NULL

# Simulate dependent variable Y
Y <- as.matrix(rnorm(n,mean = 0.8*(x1 > 0), sd = 1 + 1*(x2 > 0)))

# Fit DRF with 50 CI groups, each 10 trees large. This results in 50 uncertainty weights
DRF <- drf(X=X, Y=Y,num.trees=500, min.node.size = 5, ci.group.size=500/50, num.threads=1)

# Obtain Test point
x<-matrix(c(0.2, 0.4, runif(8,-1,1)), nrow=1, ncol=10)

# (we only use 2 threads in this example due to cran limitations)
DRFpred<-predict(DRF, newdata=x, estimate.uncertainty=TRUE, num.threads=1)

## Sample from P_{Y| X=x}
Yxs<-Y[sample(1:n, size=n, replace = TRUE, DRFpred$weights[1,])]

# Calculate quantile prediction as weighted quantiles from Y
qx <- quantile(Yxs, probs = c(0.05,0.95))

# Calculate conditional mean prediction
mux <- mean(Yxs)

# True quantiles
q1<-qnorm(0.05, mean=0.8 * (x[1] > 0), sd=(1+(x[2] > 0)))
q2<-qnorm(0.95, mean=0.8 * (x[1] > 0), sd=(1+(x[2] > 0)))
mu<-0.8 * (x[1] > 0)

# Calculate uncertainty
alpha<-0.05
B<-nrow(DRFpred$weights.uncertainty[[1]])
qxb<-matrix(NaN, nrow=B, ncol=2)
muxb<-matrix(NaN, nrow=B, ncol=1)
```

```
for (b in 1:B){
  Yxsb<-Y[sample(1:n, size=n, replace = TRUE, DRFpred$weights.uncertainty[[1]][b,])]
  qxb[b,] <- quantile(Yxsb, probs = c(0.05,0.95))
  muxb[b] <- mean(Yxsb)
}

CI.lower.q1 <- qx[1] - qnorm(1-alpha/2)*sqrt(var(qxb[,1]))
CI.upper.q1 <- qx[1] + qnorm(1-alpha/2)*sqrt(var(qxb[,1]))

CI.lower.q2 <- qx[2] - qnorm(1-alpha/2)*sqrt(var(qxb[,2]))
CI.upper.q2 <- qx[2] + qnorm(1-alpha/2)*sqrt(var(qxb[,2]))

CI.lower.mu <- mux - qnorm(1-alpha/2)*sqrt(var(muxb))
CI.upper.mu <- mux + qnorm(1-alpha/2)*sqrt(var(muxb))

hist(Yxs, prob=TRUE)
z<-seq(-6,7,by=0.01)
d<-dnorm(z, mean=0.8 * (x[1] > 0), sd=(1+(x[2] > 0)))
lines(z,d, col="darkred"  )
abline(v=q1,col="darkred" )
abline(v=q2, col="darkred" )
abline(v=qx[1], col="darkblue")
abline(v=qx[2], col="darkblue")
abline(v=mu, col="darkred")
abline(v=mux, col="darkblue")
abline(v=CI.lower.q1, col="darkblue", lty=2)
abline(v=CI.upper.q1, col="darkblue", lty=2)
abline(v=CI.lower.q2, col="darkblue", lty=2)
abline(v=CI.upper.q2, col="darkblue", lty=2)
abline(v=CI.lower.mu, col="darkblue", lty=2)
abline(v=CI.upper.mu, col="darkblue", lty=2)
```

`print.drf` *Print a DRF forest object.*

Description

Print a DRF forest object.

Usage

```
## S3 method for class 'drf'
print(x, decay.exponent = 2, max.depth = 4, ...)
```

Arguments

x	The tree to print.
decay.exponent	A tuning parameter that controls the importance of split depth.
max.depth	The maximum depth of splits to consider.
...	Additional arguments (currently ignored).

---

print.drf_tree *Print a DRF tree object.*

Description

Print a DRF tree object.

Usage

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

Arguments

x	The tree to print.
...	Additional arguments (currently ignored).

---

split_frequencies *Calculate which features the forest split on at each depth.*

Description

Calculate which features the forest split on at each depth.

Usage

```
split_frequencies(forest, max.depth = 4)
```

Arguments

forest	The trained forest.
max.depth	Maximum depth of splits to consider.

Value

A matrix of split depth by feature index, where each value is the number of times the feature was split on at that depth.

Examples

```
## Not run:
# Train a quantile forest.
n <- 50
p <- 10
X <- matrix(rnorm(n * p), n, p)
Y <- X[, 1] * rnorm(n)
q.forest <- quantile_forest(X, Y, quantiles = c(0.1, 0.5, 0.9))

# Calculate the split frequencies for this forest.
split_frequencies(q.forest)

## End(Not run)
```

---

variableImportance *Variable importance based on MMD*

Description

compute an mmd-based variable importance for the drf fit.

Usage

```
variableImportance(
  object,
  h = NULL,
  response.scaling = TRUE,
  type = "difference"
)
```

Arguments

object	an S3 object of class drf.
h	the bandwidth parameter, default to NULL using then the median heuristic.
response.scaling	a boolean value indicating if the responses should be scaled globally beforehand.
type	the type of importance, could be either "raw", the plain MMD values, "relative", the ratios to the observed MMD or "difference", the excess to the observed MMD

Value

a vector of variable importance values.

variable_importance *Calculate a simple measure of 'importance' for each feature.*

Description

A simple weighted sum of how many times feature i was split on at each depth in the forest.

Usage

```
variable_importance(forest, decay.exponent = 2, max.depth = 4)
```

Arguments

forest	The trained forest.
decay.exponent	A tuning parameter that controls the importance of split depth.
max.depth	Maximum depth of splits to consider.

Value

A list specifying an 'importance value' for each feature.

Examples

```
## Not run:
# Train a quantile forest.
n <- 50
p <- 10
X <- matrix(rnorm(n * p), n, p)
Y <- X[, 1] * rnorm(n)
q.forest <- quantile_forest(X, Y, quantiles = c(0.1, 0.5, 0.9))

# Calculate the 'importance' of each feature.
variable_importance(q.forest)

## End(Not run)
```

weighted.quantile *Weighted quantiles*

Description

Weighted quantiles

Usage

```
weighted.quantile(x, w, probs = seq(0, 1, 0.25), na.rm = TRUE)
```

Arguments

x	a vector of observations
w	a vector of weights
probs	the given probabilities for which we want to get quantiles
na.rm	should we remove missing values.

Index