Package ‘grf’

April 14, 2018

Title Generalized Random Forests (Beta)

Version 0.9.6

Author Julie Tibshirani [aut, cre],
Susan Athey [aut],
Stefan Wager [aut],
Rina Friedberg [ctb],
Luke Miner [ctb],
Marvin Wright [ctb]

BugReports https://github.com/swager/grf/issues

Maintainer Julie Tibshirani <jtibs@cs.stanford.edu>

Description A pluggable package for forest-based statistical estimation and inference. GRF currently provides methods for non-parametric least-squares regression, quantile regression, and treatment effect estimation (optionally using instrumental variables). This package is currently in beta, and we expect to make continual improvements to its performance and usability.

Depends R (>= 3.3.0)

License GPL-3

LinkingTo Rcpp, RcppEigen

Imports DiceKriging, Matrix, methods, Rcpp (>= 0.12.15), sandwich (>= 2.4-0)

RoxygenNote 6.0.1.9000

Suggests testthat

SystemRequirements GNU make

URL https://github.com/swager/grf

NeedsCompilation yes

Repository CRAN

Date/Publication 2018-04-14 14:50:00 UTC

R topics documented:

`average_partial_effect`

Estimate average partial effects using a causal forest

Description

Gets estimates of the average partial effect, in particular the (conditional) average treatment effect (target.sample = all): 1/n sum_i = 1^n Cov[Wi, Yi | X = Xi] / Var[Wi | X = Xi]. Note that for a binary unconfounded treatment, the average partial effect matches the average treatment effect.

Usage

```
average_partial_effect(forest, calibrate.weights = TRUE)
```

Arguments

`forest` The trained forest.

`calibrate.weights`
Whether to force debiasing weights to match expected moments for 1, W, W.hat, and 1/Var[W|X].

Value

An estimate of the average partial effect, along with standard error.

Examples

```
## Not run:
n = 2000; p = 10
X = matrix(rnorm(n*p), n, p)
W = rbinom(n, 1, 1/(1 + exp(-X[,2]))) + rnorm(n)
Y = pmax(X[,1], 0) * W + X[,2] + pmin(X[,3], 0) + rnorm(n)
tau.forest = causal_forest(X, Y, W)
tau.hat = predict(tau.forest)
average_partial_effect(tau.forest)

## End(Not run)
```

`average_treatment_effect` *Estimate average treatment effects using a causal forest*

Description

Gets estimates of one of the following.

- The (conditional) average treatment effect (target.sample = all): sum_i = 1^n E[Y(1) - Y(0) | X = Xi] / n
- The (conditional) average treatment effect on the treated (target.sample = treated): sum_Wi = 1 E[Y(1) - Y(0) | X = Xi] / |i : Wi = 1|
- The (conditional) average treatment effect on the controls (target.sample = control): sum_Wi = 0 E[Y(1) - Y(0) | X = Xi] / |i : Wi = 0|
- The overlap-weighted (conditional) average treatment effect sum_i = 1^n e(Xi) (1 - e(Xi)) E[Y(1) - Y(0) | X = Xi] / sum_i = 1^n e(Xi) (1 - e(Xi)), where e(x) = P[Wi = 1 | Xi = x].

This last estimand is recommended by Li, Morgan, and Zaslavsky (JASA, 2017) in case of poor overlap (i.e., when the propensities e(x) may be very close to 0 or 1), as it doesn't involve dividing by estimated propensities.

Usage

```
average_treatment_effect(forest, target.sample = c("all", "treated",
  "control", "overlap"), method = c("AIPW", "TMLE"))
```

Arguments

`forest`	The trained forest.
`target.sample`	Which sample to aggregate treatment effects over.
`method`	Method used for doubly robust inference. Can be either augmented inverse-propensity weighting (AIPW), or targeted maximum likelihood estimation (TMLE).

Value

An estimate of the average treatment effect, along with standard error.

Examples

```
## Not run:
# Train a causal forest.
n = 50; p = 10
X = matrix(rnorm(n*p), n, p)
W = rbinom(n, 1, 0.5)
Y = pmax(X[,1], 0) * W + X[,2] + pmin(X[,3], 0) + rnorm(n)
c.forest = causal_forest(X, Y, W)

# Predict using the forest.
X.test = matrix(0, 101, p)
X.test[,1] = seq(-2, 2, length.out = 101)
c.pred = predict(c.forest, X.test)
# Estimate the conditional average treatment effect on the full sample (CATE).
average_treatment_effect(c.forest, target.sample = "all")

# Estimate the conditional average treatment effect on the treated sample (CATT).
# We don't expect much difference between the CATE and the CATT in this example,
# since treatment assignment was randomized.
average_treatment_effect(c.forest, target.sample = "treated")

## End(Not run)
```

---

causal_forest *Causal forest*

Description

Trains a causal forest that can be used to estimate conditional average treatment effects tau(X). When the treatment assignmnet W is binary and unconfounded, we have tau(X) = E[Y(1) - Y(0) | X = x], where Y(0) and Y(1) are potential outcomes corresponding to the two possible treatment states. When W is continuous, we effectively estimate an average partical effect Cov[Y, W | X = x] / Var[W | X = x], and interpret it as a treatment effect given unconfoundedness.

Usage

```
causal_forest(X, Y, W, sample.fraction = 0.5, mtry = NULL,
  num.trees = 2000, num.threads = NULL, min.node.size = NULL,
  honesty = TRUE, ci.group.size = 2, precompute.nuisance = TRUE,
  alpha = NULL, imbalance.penalty = NULL, stabilize.splits = TRUE,
  seed = NULL, clusters = NULL, samples_per_cluster = NULL,
  tune.parameters = FALSE, num.fit.trees = 40, num.fit.reps = 100,
  num.optimize.reps = 1000)
```

Arguments

- `X` The covariates used in the causal regression.
- `Y` The outcome.
- `W` The treatment assignment (may be binary or real).
- `sample.fraction` Fraction of the data used to build each tree. Note: If honesty is used, these subsamples will further be cut in half.
- `mtry` Number of variables tried for each split.
- `num.trees` Number of trees grown in the forest. Note: Getting accurate confidence intervals generally requires more trees than getting accurate predictions.
- `num.threads` Number of threads used in training. If set to NULL, the software automatically selects an appropriate amount.
- `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.
- `honesty` Whether or not honest splitting (i.e., sub-sample splitting) should be used.
- `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.
- `precompute.nuisance` Should we first run regression forests to estimate y(x) = E[Y|X=x] and w(x) = E[W|X=x], and then run a causal forest on the residuals? This approach is recommended, computational resources permitting.
- `alpha` A tuning parameter that controls the maximum imbalance of a split.
- `imbalance.penalty` A tuning parameter that controls how harshly imbalanced splits are penalized.
- `stabilize.splits` Whether or not the treatment should be taken into account when determining the imbalance of a split (experimental).
- `seed` The seed of the C++ random number generator.
- `clusters` Vector of integers or factors specifying which cluster each observation corresponds to.
- `samples_per_cluster` If sampling by cluster, the number of observations to be sampled from each cluster. Must be less than the size of the smallest cluster. If set to NULL software will set this value to the size of the smallest cluster.#'
- `tune.parameters` If true, NULL parameters are tuned by cross-validation; if false NULL parameters are set to defaults.
- `num.fit.trees` The number of trees in each 'mini forest' used to fit the tuning model.
- `num.fit.reps` The number of forests used to fit the tuning model.
- `num.optimize.reps` The number of random parameter values considered when using the model to select the optimal parameters.

Value

A trained causal forest object.

Examples

```
## Not run:
# Train a causal forest.
n = 50; p = 10
X = matrix(rnorm(n*p), n, p)
W = rbinom(n, 1, 0.5)
Y = pmax(X[,1], 0) * W + X[,2] + pmin(X[,3], 0) + rnorm(n)
c.forest = causal_forest(X, Y, W)

# Predict using the forest.
X.test = matrix(0, 101, p)
X.test[,1] = seq(-2, 2, length.out = 101)
c.pred = predict(c.forest, X.test)

# Predict on out-of-bag training samples.
c.pred = predict(c.forest)

# Predict with confidence intervals; growing more trees is now recommended.
c.forest = causal_forest(X, Y, W, num.trees = 4000)
c.pred = predict(c.forest, X.test, estimate.variance = TRUE)

## End(Not run)
```

---

custom_forest *Custom forest*

Description

Trains a custom forest model.

Usage

```
custom_forest(X, Y, sample.fraction = 0.5, mtry = NULL, num.trees = 2000,
  num.threads = NULL, min.node.size = NULL, honesty = TRUE,
  alpha = 0.05, imbalance.penalty = 0, seed = NULL, clusters = NULL,
  samples_per_cluster = NULL)
```

Arguments

`X`	The covariates used in the regression.
`Y`	The outcome.
`sample.fraction`	Fraction of the data used to build each tree. Note: If honesty is used, these subsamples will further be cut in half.

mtry
: Number of variables tried for each split.

num.trees
: Number of trees grown in the forest. Note: Getting accurate confidence intervals generally requires more trees than getting accurate predictions.

num.threads
: Number of threads used in training. If set to NULL, the software automatically selects an appropriate amount.

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.

honesty
: Whether or not honest splitting (i.e., sub-sample splitting) should be used.

alpha
: A tuning parameter that controls the maximum imbalance of a split.

imbalance.penalty
: A tuning parameter that controls how harshly imbalanced splits are penalized.

seed
: The seed for the C++ random number generator.

clusters
: Vector of integers or factors specifying which cluster each observation corresponds to.

samples_per_cluster
: If sampling by cluster, the number of observations to be sampled from each cluster. Must be less than the size of the smallest cluster. If set to NULL software will set this value to the size of the smallest cluster.

Value

A trained regression forest object.

Examples

```
## Not run:
# Train a custom forest.
n = 50; p = 10
X = matrix(rnorm(n*p), n, p)
Y = X[,1] * rnorm(n)
c.forest = custom_forest(X, Y)

# Predict using the forest.
X.test = matrix(0, 101, p)
X.test[,1] = seq(-2, 2, length.out = 101)
c.pred = predict(c.forest, X.test)

## End(Not run)
```

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, 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).#' @param max.depth Maximum depth of splits to consider.
`num.threads`	Number of threads used in training. If set to NULL, the software automatically selects an appropriate amount.

Value

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.

Examples

```
## Not run:
 p = 10
 n = 100
 X = matrix(2 * runif(n * p) - 1, n, p)
 Y = (X[,1] > 0) + 2 * rnorm(n)
 rrf = regression_forest(X, Y, 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 GRF tree object.

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)
```

`grf` *GRF*

Description

A pluggable package for forest-based statistical estimation and inference. GRF currently provides non-parametric methods for least-squares regression, quantile regression, and treatment effect estimation (optionally using instrumental variables).

In addition, GRF supports 'honest' estimation (where one subset of the data is used for choosing splits, and another for populating the leaves of the tree), and confidence intervals for least-squares regression and treatment effect estimation.

This package is currently in beta, and we expect to make continual improvements to its performance and usability.

Examples

```
## Not run:
library(grf)

# The following script demonstrates how to use GRF for heterogeneous treatment
# effect estimation. For examples of how to use other types of forest, as for
# quantile regression and causal effect estimation using instrumental variables,
# please consult the documentation on the relevant forest methods (quantile_forest,
# instrumental_forest, etc.).

# Generate data.
n = 2000; p = 10
X = matrix(rnorm(n*p), n, p)
X.test = matrix(0, 101, p)
X.test[,1] = seq(-2, 2, length.out = 101)

# Perform treatment effect estimation.
W = rbinom(n, 1, 0.5)
Y = pmax(X[,1], 0) * W + X[,2] + pmin(X[,3], 0) + rnorm(n)
tau.forest = causal_forest(X, Y, W)
tau.hat = predict(tau.forest, X.test)
plot(X.test[,1], tau.hat$predictions, ylim = range(tau.hat$predictions, 0, 2),
xlab = "x", ylab = "tau", type = "l")
lines(X.test[,1], pmax(0, X.test[,1]), col = 2, lty = 2)

# Estimate the conditional average treatment effect on the full sample (CATE).
average_treatment_effect(tau.forest, target.sample = "all")

# Estimate the conditional average treatment effect on the treated sample (CATT).
# Here, we don't expect much difference between the CATE and the CATT, since
# treatment assignment was randomized.
average_treatment_effect(tau.forest, target.sample = "treated")

# Add confidence intervals for heterogeneous treatment effects; growing
# more trees is now recommended.
tau.forest = causal_forest(X, Y, W, num.trees = 4000)
tau.hat = predict(tau.forest, X.test, estimate.variance = TRUE)
sigma.hat = sqrt(tau.hat$variance.estimates)
plot(X.test[,1], tau.hat$predictions, ylim = range(tau.hat$predictions + 1.96 * sigma.hat,
tau.hat$predictions - 1.96 * sigma.hat, 0, 2), xlab = "x", ylab = "tau", type = "l")
```

```
lines(X.test[,1], tau.hat$predictions + 1.96 * sigma.hat, col = 1, lty = 2)
lines(X.test[,1], tau.hat$predictions - 1.96 * sigma.hat, col = 1, lty = 2)
lines(X.test[,1], pmax(0, X.test[,1]), col = 2, lty = 1)

## End(Not run)
```

`instrumental_forest` *Intrumental forest*

Description

Trains an instrumental forest that can be used to estimate conditional local average treatment effects tau(X) identified using instruments. Formally, the forest estimates tau(X) = Cov[Y, Z | X = x] / Cov[W, Z | X = x]. Note that when the instrument Z and treatment assignment W coincide, an instrumental forest is equivalent to a causal forest.

Usage

```
instrumental_forest(X, Y, W, Z, sample.fraction = 0.5, mtry = NULL,
  num.trees = 2000, num.threads = NULL, min.node.size = NULL,
  honesty = TRUE, ci.group.size = 2, precompute.nuisance = TRUE,
  reduced.form.weight = 0, alpha = 0.05, imbalance.penalty = 0,
  stabilize.splits = TRUE, seed = NULL, clusters = NULL,
  samples_per_cluster = NULL)
```

Arguments

`X`	The covariates used in the instrumental regression.
`Y`	The outcome.
`W`	The treatment assignment (may be binary or real).
`Z`	The instrument (may be binary or real).
`sample.fraction`	Fraction of the data used to build each tree. Note: If honesty is used, these subsamples will further be cut in half.
`mtry`	Number of variables tried for each split.
`num.trees`	Number of trees grown in the forest. Note: Getting accurate confidence intervals generally requires more trees than getting accurate predictions.
`num.threads`	Number of threads used in training. If set to NULL, the software automatically selects an appropriate amount.
`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.
`honesty`	Whether or not honest splitting (i.e., sub-sample splitting) should be used.
`ci.group.size`	The forst will grow ci.group.size trees on each subsample. In order to provide confidence intervals, ci.group.size must be at least 2.

precompute.nuisance
: Should we first run regression forests to estimate y(x) = E[Y|X=x], w(x) = E[W|X=x] and z(x) = E[Z|X=x], and then run an instrumental forest on the residuals? This approach is recommended, computational resources permitting.

reduced.form.weight
: Whether splits should be regularized towards a naive splitting criterion that ignores the instrument (and instead emulates a causal forest).

alpha
: A tuning parameter that controls the maximum imbalance of a split.

imbalance.penalty
: A tuning parameter that controls how harshly imbalanced splits are penalized.

stabilize.splits
: Whether or not the instrument should be taken into account when determining the imbalance of a split (experimental).

seed
: The seed for the C++ random number generator.

clusters
: Vector of integers or factors specifying which cluster each observation corresponds to.

samples_per_cluster
: If sampling by cluster, the number of observations to be sampled from each cluster. Must be less than the size of the smallest cluster. If set to NULL software will set this value to the size of the smallest cluster.

Value

A trained instrumental forest object.

---

predict.causal_forest *Predict with a causal forest*

Description

Gets estimates of tau(x) using a trained causal forest.

Usage

```
## S3 method for class 'causal_forest'
predict(object, newdata = NULL, num.threads = NULL,
  estimate.variance = FALSE, ...)
```

Arguments

object
: 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).

num.threads
: Number of threads used in training. If set to NULL, the software automatically selects an appropriate amount.

`estimate.variance`
: Whether variance estimates for hattau(x) are desired (for confidence intervals).

`...`
: Additional arguments (currently ignored).

Value

Vector of predictions, along with (optional) variance estimates.

Examples

```
## Not run:
# Train a causal forest.
n = 100; p = 10
X = matrix(rnorm(n*p), n, p)
W = rbinom(n, 1, 0.5)
Y = pmax(X[,1], 0) * W + X[,2] + pmin(X[,3], 0) + rnorm(n)
c.forest = causal_forest(X, Y, W)

# Predict using the forest.
X.test = matrix(0, 101, p)
X.test[,1] = seq(-2, 2, length.out = 101)
c.pred = predict(c.forest, X.test)

# Predict on out-of-bag training samples.
c.pred = predict(c.forest)

# Predict with confidence intervals; growing more trees is now recommended.
c.forest = causal_forest(X, Y, W, num.trees = 500)
c.pred = predict(c.forest, X.test, estimate.variance = TRUE)

## End(Not run)
```

`predict.custom_forest` *Predict with a custom forest.*

Description

Predict with a custom forest.

Usage

```
## S3 method for class 'custom_forest'
predict(object, newdata = NULL, num.threads = NULL,
  ...)
```

Arguments

object	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).
num.threads	Number of threads used in training. If set to NULL, the software automatically selects an appropriate amount.
...	Additional arguments (currently ignored).

Value

Vector of predictions.

Examples

```
## Not run:
# Train a custom forest.
n = 50; p = 10
X = matrix(rnorm(n*p), n, p)
Y = X[,1] * rnorm(n)
c.forest = custom_forest(X, Y)

# Predict using the forest.
X.test = matrix(0, 101, p)
X.test[,1] = seq(-2, 2, length.out = 101)
c.pred = predict(c.forest, X.test)

## End(Not run)
```

---

predict.instrumental_forest

Predict with an instrumental forest

Description

Gets estimates of tau(x) using a trained instrumental forest.

Usage

```
## S3 method for class 'instrumental_forest'
predict(object, newdata = NULL,
  num.threads = NULL, estimate.variance = FALSE, ...)
```

Arguments

`object`	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).
`num.threads`	Number of threads used in training. If set to NULL, the software automatically selects an appropriate amount.
`estimate.variance`	Whether variance estimates for hattau(x) are desired (for confidence intervals).
`...`	Additional arguments (currently ignored).

Value

Vector of predictions, along with (optional) variance estimates.

`predict.quantile_forest`

Predict with a quantile forest

Description

Gets estimates of the conditional quantiles of Y given X using a trained forest.

Usage

```
## S3 method for class 'quantile_forest'
predict(object, newdata = NULL, quantiles = c(0.1,
  0.5, 0.9), num.threads = NULL, ...)
```

Arguments

`object`	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).
`quantiles`	Vector of quantiles at which estimates are required.
`num.threads`	Number of threads used in training. If set to NULL, the software automatically selects an appropriate amount.
`...`	Additional arguments (currently ignored).

Value

Predictions at each test point for each desired quantile.

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))

# Predict on out-of-bag training samples.
q.pred = predict(q.forest)

# Predict using the forest.
X.test = matrix(0, 101, p)
X.test[,1] = seq(-2, 2, length.out = 101)
q.pred = predict(q.forest, X.test)

## End(Not run)
```

`predict.regression_forest` *Predict with a regression forest*

Description

Gets estimates of E[Y|X=x] using a trained regression forest.

Usage

```
## S3 method for class 'regression_forest'
predict(object, newdata = NULL,
  local.linear = FALSE, lambda = 0, ridge.type = "standardized",
  num.threads = NULL, estimate.variance = FALSE, ...)
```

Arguments

`object`	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).
`local.linear`	Optional local linear prediction correction. If TRUE, code will run a locally weighted ridge regression at each test point. Note that this is a beta feature still in development, and may slow down prediction considerably.
`lambda`	Ridge penalty for local linear predictions
`ridge.type`	Option to standardize ridge penalty by covariance ("standardized"), or penalize all covariates equally ("identity").

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

`estimate.variance`
Whether variance estimates for hattau(x) are desired (for confidence intervals).

`...` Additional arguments (currently ignored).

Value

A vector of predictions.

Examples

```
## Not run:
# Train a standard regression forest.
n = 50; p = 10
X = matrix(rnorm(n*p), n, p)
Y = X[,1] * rnorm(n)
r.forest = regression_forest(X, Y)

# Predict using the forest.
X.test = matrix(0, 101, p)
X.test[,1] = seq(-2, 2, length.out = 101)
r.pred = predict(r.forest, X.test)

# Predict on out-of-bag training samples.
r.pred = predict(r.forest)

# Predict with confidence intervals; growing more trees is now recommended.
r.forest = regression_forest(X, Y, num.trees = 100)
r.pred = predict(r.forest, X.test, estimate.variance = TRUE)

## End(Not run)
```

`print.grf` *Print a GRF forest object.*

Description

Print a GRF forest object.

Usage

```
## S3 method for class 'grf'
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.grf_tree` *Print a GRF tree object.*

Description

Print a GRF tree object.

Usage

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

Arguments

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

`quantile_forest` *Quantile forest*

Description

Trains a regression forest that can be used to estimate quantiles of the conditional distribution of Y given X = x.

Usage

```
quantile_forest(X, Y, quantiles = c(0.1, 0.5, 0.9),
  regression.splitting = FALSE, sample.fraction = 0.5, mtry = NULL,
  num.trees = 2000, num.threads = NULL, min.node.size = NULL,
  honesty = TRUE, alpha = 0.05, imbalance.penalty = 0, seed = NULL,
  clusters = NULL, samples_per_cluster = NULL)
```

Arguments

- `X` The covariates used in the quantile regression.
- `Y` The outcome.
- `quantiles` Vector of quantiles used to calibrate the forest.
- `regression.splitting` Whether to use regression splits when growing trees instead of specialized splits based on the quantiles (the default). Setting this flag to true corresponds to the approach to quantile forests from Meinshausen (2006).
- `sample.fraction` Fraction of the data used to build each tree. Note: If honesty is used, these subsamples will further be cut in half.
- `mtry` Number of variables tried for each split.
- `num.trees` Number of trees grown in the forest. Note: Getting accurate confidence intervals generally requires more trees than getting accurate predictions.
- `num.threads` Number of threads used in training. If set to NULL, the software automatically selects an appropriate amount.
- `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.
- `honesty` Whether or not honest splitting (i.e., sub-sample splitting) should be used.
- `alpha` A tuning parameter that controls the maximum imbalance of a split.
- `imbalance.penalty` A tuning parameter that controls how harshly imbalanced splits are penalized.
- `seed` The seed for the C++ random number generator.
- `clusters` Vector of integers or factors specifying which cluster each observation corresponds to.
- `samples_per_cluster` If sampling by cluster, the number of observations to be sampled from each cluster. Must be less than the size of the smallest cluster. If set to NULL software will set this value to the size of the smallest cluster.

Value

A trained quantile forest object.

Examples

```
## Not run:
# Generate data.
n = 50; p = 10
X = matrix(rnorm(n*p), n, p)
X.test = matrix(0, 101, p)
X.test[,1] = seq(-2, 2, length.out = 101)
Y = X[,1] * rnorm(n)
```

```
# Train a quantile forest.
q.forest = quantile_forest(X, Y, quantiles=c(0.1, 0.5, 0.9))

# Make predictions.
q.hat = predict(q.forest, X.test)

# Make predictions for different quantiles than those used in training.
q.hat = predict(q.forest, X.test, quantiles=c(0.1, 0.9))

# Train a quantile forest using regression splitting instead of quantile-based
# splits, emulating the approach in Meinshausen (2006).
meins.forest = quantile_forest(X, Y, regression.splitting=TRUE)

# Make predictions for the desired quantiles.
q.hat = predict(meins.forest, X.test, quantiles=c(0.1, 0.5, 0.9))

## End(Not run)
```

---

regression_forest *Regression forest*

Description

Trains a regression forest that can be used to estimate the conditional mean function mu(x) = E[Y | X = x]

Usage

```
regression_forest(X, Y, sample.fraction = 0.5, mtry = NULL,
  num.trees = 2000, num.threads = NULL, min.node.size = NULL,
  honesty = TRUE, ci.group.size = 2, alpha = NULL,
  imbalance.penalty = NULL, seed = NULL, clusters = NULL,
  samples_per_cluster = NULL, tune.parameters = FALSE, num.fit.trees = 10,
  num.fit.reps = 100, num.optimize.reps = 1000)
```

Arguments

X	The covariates used in the regression.
Y	The outcome.
sample.fraction	Fraction of the data used to build each tree. Note: If honesty is used, these subsamples will further be cut in half.
mtry	Number of variables tried for each split.
num.trees	Number of trees grown in the forest. Note: Getting accurate confidence intervals generally requires more trees than getting accurate predictions.
num.threads	Number of threads used in training. If set to NULL, the software automatically selects an appropriate amount.

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.

honesty Whether or not honest splitting (i.e., sub-sample splitting) should be used.

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.

alpha A tuning parameter that controls the maximum imbalance of a split.

imbalance.penalty A tuning parameter that controls how harshly imbalanced splits are penalized.

seed The seed for the C++ random number generator.

clusters Vector of integers or factors specifying which cluster each observation corresponds to.

samples_per_cluster If sampling by cluster, the number of observations to be sampled from each cluster. Must be less than the size of the smallest cluster. If set to NULL software will set this value to the size of the smallest cluster.

tune.parameters If true, NULL parameters are tuned by cross-validation; if false NULL parameters are set to defaults.

num.fit.trees The number of trees in each 'mini forest' used to fit the tuning model.

num.fit.reps The number of forests used to fit the tuning model.

num.optimize.reps The number of random parameter values considered when using the model to select the optimal parameters.

Value

A trained regression forest object.

Examples

```
## Not run:
# Train a standard regression forest.
n = 50; p = 10
X = matrix(rnorm(n*p), n, p)
Y = X[,1] * rnorm(n)
r.forest = regression_forest(X, Y)

# Predict using the forest.
X.test = matrix(0, 101, p)
X.test[,1] = seq(-2, 2, length.out = 101)
r.pred = predict(r.forest, X.test)

# Predict on out-of-bag training samples.
r.pred = predict(r.forest)

# Predict with confidence intervals; growing more trees is now recommended.
r.forest = regression_forest(X, Y, num.trees = 100)
```

```
r.pred = predict(r.forest, X.test, estimate.variance = TRUE)

## End(Not run)
```

`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)
```

`tune_causal_forest` *Causal forest tuning*

Description

Finds the optimal parameters to be used in training a regression forest. This method currently tunes over min.node.size, mtry, sample.fraction, alpha, and imbalance.penalty. Please see the method 'causal_forest' for a description of the standard causal forest parameters. Note that if fixed values can be supplied for any of the parameters mentioned above, and in that case, that parameter will not be tuned. For example, if this method is called with min.node.size = 10 and alpha = 0.7, then those parameter values will be treated as fixed, and only sample.fraction and imbalance.penalty will be tuned.

Usage

```
tune_causal_forest(X, Y, W, num.fit.trees = 40, num.fit.reps = 100,
  num.optimize.reps = 1000, min.node.size = NULL, sample.fraction = 0.5,
  mtry = NULL, alpha = NULL, imbalance.penalty = NULL,
  stabilize.splits = TRUE, num.threads = NULL, honesty = TRUE,
  seed = NULL, clusters = NULL, samples_per_cluster = NULL)
```

Arguments

`X` The covariates used in the causal regression.

`Y` The outcome.

`W` The treatment assignment (may be binary or real).

`num.fit.trees` The number of trees in each 'mini forest' used to fit the tuning model.

`num.fit.reps` The number of forests used to fit the tuning model.

`num.optimize.reps`
The number of random parameter values considered when using the model to select the optimal parameters.

`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.

`sample.fraction`
Fraction of the data used to build each tree. Note: If honesty is used, these subsamples will further be cut in half.

`mtry` Number of variables tried for each split.

`alpha` A tuning parameter that controls the maximum imbalance of a split.

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

`stabilize.splits`
Whether or not the treatment should be taken into account when determining the imbalance of a split (experimental).

num.threads	Number of threads used in training. If set to NULL, the software automatically selects an appropriate amount.
honesty	Whether or not honest splitting (i.e., sub-sample splitting) should be used.
seed	The seed of the C++ random number generator.
clusters	Vector of integers or factors specifying which cluster each observation corresponds to.
samples_per_cluster	If sampling by cluster, the number of observations to be sampled from each cluster. Must be less than the size of the smallest cluster. If set to NULL software will set this value to the size of the smallest cluster.#'

Value

A list consisting of the optimal parameter values ('params') along with their debiased error ('error').

Examples

```
## Not run:
# Find the optimal tuning parameters.
n = 50; p = 10
X = matrix(rnorm(n*p), n, p)
W = rbinom(n, 1, 0.5)
Y = pmax(X[,1], 0) * W + X[,2] + pmin(X[,3], 0) + rnorm(n)
params = tune_causal_forest(X, Y, W)$params

# Use these parameters to train a regression forest.
tuned.forest = causal_forest(X, Y, W, num.trees = 1000,
    min.node.size = as.numeric(params["min.node.size"]),
    sample.fraction = as.numeric(params["sample.fraction"]),
    mtry = as.numeric(params["mtry"]),
    alpha = as.numeric(params["alpha"]),
    imbalance.penalty = as.numeric(params["imbalance.penalty"])

## End(Not run)
```

---

tune_regression_forest

Regression forest tuning

Description

Finds the optimal parameters to be used in training a regression forest. This method currently tunes over min.node.size, mtry, sample.fraction, alpha, and imbalance.penalty. Please see the method 'regression_forest' for a description of the standard forest parameters. Note that if fixed values can be supplied for any of the parameters mentioned above, and in that case, that parameter will not be tuned. For example, if this method is called with min.node.size = 10 and alpha = 0.7, then those parameter values will be treated as fixed, and only sample.fraction and imbalance.penalty will be tuned.

Usage

```
tune_regression_forest(X, Y, num.fit.trees = 10, num.fit.reps = 100,
  num.optimize.reps = 1000, min.node.size = NULL, sample.fraction = 0.5,
  mtry = NULL, alpha = NULL, imbalance.penalty = NULL,
  num.threads = NULL, honesty = TRUE, seed = NULL, clusters = NULL,
  samples_per_cluster = NULL)
```

Arguments

`X`	The covariates used in the regression.
`Y`	The outcome.
`num.fit.trees`	The number of trees in each 'mini forest' used to fit the tuning model.
`num.fit.reps`	The number of forests used to fit the tuning model.
`num.optimize.reps`	The number of random parameter values considered when using the model to select the optimal parameters.
`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.
`sample.fraction`	Fraction of the data used to build each tree. Note: If honesty is used, these subsamples will further be cut in half.
`mtry`	Number of variables tried for each split.
`alpha`	A tuning parameter that controls the maximum imbalance of a split.
`imbalance.penalty`	A tuning parameter that controls how harshly imbalanced splits are penalized.
`num.threads`	Number of threads used in training. If set to NULL, the software automatically selects an appropriate amount.
`honesty`	Whether or not honest splitting (i.e., sub-sample splitting) should be used.
`seed`	The seed for the C++ random number generator.
`clusters`	Vector of integers or factors specifying which cluster each observation corresponds to.
`samples_per_cluster`	If sampling by cluster, the number of observations to be sampled from each cluster. Must be less than the size of the smallest cluster. If set to NULL software will set this value to the size of the smallest cluster.

Value

A list consisting of the optimal parameter values ('params') along with their debiased error ('error').

Examples

```
## Not run:
# Find the optimal tuning parameters.
n = 500; p = 10
X = matrix(rnorm(n*p), n, p)
Y = X[,1] * rnorm(n)
params = tune_regression_forest(X, Y)$params

# Use these parameters to train a regression forest.
tuned.forest = regression_forest(X, Y, num.trees = 1000,
    min.node.size = as.numeric(params["min.node.size"]),
    sample.fraction = as.numeric(params["sample.fraction"]),
    mtry = as.numeric(params["mtry"]),
    alpha = as.numeric(params["alpha"]),
    imbalance.penalty = as.numeric(params["imbalance.penalty"]))

## End(Not run)
```

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

Description

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

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)
```

Index