# Package 'grf'

July 14, 2021

**Title** Generalized Random Forests

**Version** 2.0.2

**BugReports** https://github.com/grf-labs/grf/issues

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

**Depends** R (>= 3.5.0)

**License** GPL-3

**LinkingTo** Rcpp, RcppEigen

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

**RoxygenNote** 7.1.1

**Suggests** DiagrammeR, MASS, survival (>= 3.2-8), testthat (>= 3.0.4)

**SystemRequirements** GNU make

**URL** https://github.com/grf-labs/grf

**NeedsCompilation** yes

**Author** Julie Tibshirani [aut, cre],
Susan Athey [aut],
Rina Friedberg [ctb],
Vitor Hadad [ctb],
David Hirshberg [ctb],
Luke Miner [ctb],
Erik Sverdrup [aut],
Stefan Wager [aut],
Marvin Wright [ctb]

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

**Repository** CRAN

**Date/Publication** 2021-07-14 16:00:02 UTC

# R topics documented:

---

`average_late` *Average LATE (removed)*

---

## Description

See the function 'average_treatment_effect'

## Usage

```
average_late(forest, ...)
```

## Arguments

| | |
|---|---|
| `forest` | The forest |
| `...` | Additional arguments (currently ignored). |

## Value

output

---

`average_partial_effect` *Average partial effect (removed)*

---

## Description

See the function 'average_treatment_effect'

## Usage

```
average_partial_effect(forest, ...)
```

## Arguments

| | |
|---|---|
| `forest` | The forest |
| `...` | Additional arguments (currently ignored). |

## Value

output

`average_treatment_effect` *Get doubly robust estimates of average treatment effects.*

## Description

In the case of a causal forest with binary treatment, we provide estimates of one of the following:

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

This last estimand is recommended by Li, Morgan, and Zaslavsky (2018) 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"),
  subset = NULL,
  debiasing.weights = NULL,
  compliance.score = NULL,
  num.trees.for.weights = 500
)
```

## Arguments

| | |
|---|---|
| `forest` | The trained forest. |
| `target.sample` | Which sample to aggregate treatment effects over. Note: Options other than "all" are only currently implemented for causal forests. |
| `method` | Method used for doubly robust inference. Can be either augmented inverse-propensity weighting (AIPW), or targeted maximum likelihood estimation (TMLE). Note: TMLE is currently only implemented for causal forests with a binary treatment. |
| `subset` | Specifies subset of the training examples over which we estimate the ATE. WARNING: For valid statistical performance, the subset should be defined only using features Xi, not using the treatment Wi or the outcome Yi. |
| `debiasing.weights` | A vector of length n (or the subset length) of debiasing weights. If NULL (default) these are obtained via the appropriate doubly robust score construction, e.g., in the case of causal_forests with a binary treatment, they are obtained via inverse-propensity weighting. |

`compliance.score`
: Only used with instrumental forests. An estimate of the causal effect of Z on W, i.e., Delta(X) = E[W | X, Z = 1] - E[W | X, Z = 0], which can then be used to produce debiasing.weights. If not provided, this is estimated via an auxiliary causal forest.

`num.trees.for.weights`
: In some cases (e.g., with causal forests with a continuous treatment), we need to train auxiliary forests to learn debiasing weights. This is the number of trees used for this task. Note: this argument is only used when debiasing.weights = NULL.

## Details

In the case of a causal forest with continuous treatment, we provide estimates of the average partial effect, i.e., E[Cov[W, Y | X] / Var[W | X]]. In the case of a binary treatment, the average partial effect matches the average treatment effect. Computing the average partial effect is somewhat more involved, as the relevant doubly robust scores require an estimate of Var[Wi | Xi = x]. By default, we get such estimates by training an auxiliary forest; however, these weights can also be passed manually by specifying debiasing.weights.

In the case of instrumental forests with a binary treatment, we provide an estimate of the the Average (Conditional) Local Averate Treatment (ACLATE). Specifically, given an outcome Y, treatment W and instrument Z, the (conditional) local average treatment effect is tau(x) = Cov[Y, Z | X = x] / Cov[W, Z | X = x]. This is the quantity that is estimated with an instrumental forest. It can be intepreted causally in various ways. Given a homogeneity assumption, tau(x) is simply the CATE at x. When W is binary and there are no "defiers", Imbens and Angrist (1994) show that tau(x) can be interpreted as an average treatment effect on compliers. This function provides and estimate of tau = E[tau(X)]. See Chernozhukov et al. (2016) for a discussion, and Section 5.2 of Athey and Wager (2021) for an example using forests.

If clusters are specified, then each unit gets equal weight by default. For example, if there are 10 clusters with 1 unit each and per-cluster ATE = 1, and there are 10 clusters with 19 units each and per-cluster ATE = 0, then the overall ATE is 0.05 (additional sample.weights allow for custom weighting). If equalize.cluster.weights = TRUE each cluster gets equal weight and the overall ATE is 0.5.

## Value

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

## References

Athey, Susan, and Stefan Wager. "Policy Learning With Observational Data." Econometrica 89.1 (2021): 133-161.

Chernozhukov, Victor, Juan Carlos Escanciano, Hidehiko Ichimura, Whitney K. Newey, and James M. Robins. "Locally robust semiparametric estimation." arXiv preprint arXiv:1608.00033, 2016.

Imbens, Guido W., and Joshua D. Angrist. "Identification and Estimation of Local Average Treatment Effects." Econometrica 62(2), 1994.

Li, Fan, Kari Lock Morgan, and Alan M. Zaslavsky. "Balancing covariates via propensity score weighting." Journal of the American Statistical Association 113(521), 2018.

Robins, James M., and Andrea Rotnitzky. "Semiparametric efficiency in multivariate regression models with missing data." Journal of the American Statistical Association 90(429), 1995.

## Examples

```
# 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")

# Estimate the conditional average treatment effect on samples with positive X[,1].
average_treatment_effect(c.forest, target.sample = "all", subset = X[, 1] > 0)

# Example for causal forests with a continuous treatment.
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_treatment_effect(tau.forest)
average_treatment_effect(tau.forest, subset = X[, 1] > 0)
```

---

`best_linear_projection` *Estimate the best linear projection of a conditional average treatment effect using a causal forest, or causal survival forest.*

---

## Description

Let tau(Xi) = E[Y(1) - Y(0) | X = Xi] be the CATE, and Ai be a vector of user-provided covariates. This function provides a (doubly robust) fit to the linear model tau(Xi) ~ beta_0 + Ai * beta.

## Usage

```
best_linear_projection(
  forest,
  A = NULL,
  subset = NULL,
  debiasing.weights = NULL,
  num.trees.for.weights = 500,
  vcov.type = "HC3"
)
```

## Arguments

| | |
|---|---|
| forest | The trained forest. |
| A | The covariates we want to project the CATE onto. |
| subset | Specifies subset of the training examples over which we estimate the ATE. WARNING: For valid statistical performance, the subset should be defined only using features Xi, not using the treatment Wi or the outcome Yi. |
| debiasing.weights | A vector of length n (or the subset length) of debiasing weights. If NULL (default) these are obtained via the appropriate doubly robust score construction, e.g., in the case of causal_forests with a binary treatment, they are obtained via inverse-propensity weighting. |
| num.trees.for.weights | In some cases (e.g., with causal forests with a continuous treatment), we need to train auxiliary forests to learn debiasing weights. This is the number of trees used for this task. Note: this argument is only used when debiasing.weights = NULL. |
| vcov.type | Optional covariance type for standard errors. The possible options are HC0, ..., HC3. The default is "HC3", which is recommended in small samples and corresponds to the "shortcut formula" for the jackknife (see MacKinnon & White for more discussion, and Cameron & Miller for a review). For large data sets with clusters, "HC0" or "HC1" are significantly faster to compute. |

## Details

Procedurally, we do so by regressing doubly robust scores derived from the forest against the Ai. Note the covariates Ai may consist of a subset of the Xi, or they may be distinct The case of the null model tau(Xi) ~ beta_0 is equivalent to fitting an average treatment effect via AIPW.

In the event the treatment is continuous the inverse-propensity weight component of the double robust scores are replaced with a component based on a forest based estimate of Var[Wi | Xi = x]. These weights can also be passed manually by specifying debiasing.weights.

## Value

An estimate of the best linear projection, along with coefficient standard errors.

## References

Cameron, A. Colin, and Douglas L. Miller. "A practitioner's guide to cluster-robust inference." Journal of human resources 50, no. 2 (2015): 317-372.

Cui, Yifan, Michael R. Kosorok, Erik Sverdrup, Stefan Wager, and Ruoqing Zhu. "Estimating Heterogeneous Treatment Effects with Right-Censored Data via Causal Survival Forests." arXiv preprint arXiv:2001.09887, 2020.

MacKinnon, James G., and Halbert White. "Some heteroskedasticity-consistent covariance matrix estimators with improved finite sample properties." Journal of Econometrics 29.3 (1985): 305-325.

Semenova, Vira, and Victor Chernozhukov. "Debiased Machine Learning of Conditional Average Treatment Effects and Other Causal Functions". The Econometrics Journal (2020).

## Examples

```
n <- 800
p <- 5
X <- matrix(rnorm(n * p), n, p)
W <- rbinom(n, 1, 0.25 + 0.5 * (X[, 1] > 0))
Y <- pmax(X[, 1], 0) * W + X[, 2] + pmin(X[, 3], 0) + rnorm(n)
forest <- causal_forest(X, Y, W)
best_linear_projection(forest, X[,1:2])
```

---

# boosted_regression_forest

*Boosted regression forest (experimental)*

---

## Description

Trains a boosted regression forest that can be used to estimate the conditional mean function mu(x) = E[Y | X = x]. Selects number of boosting iterations based on cross-validation. This functionality is experimental and will likely change in future releases.

## Usage

```
boosted_regression_forest(
  X,
  Y,
  num.trees = 2000,
  sample.weights = NULL,
  clusters = NULL,
  equalize.cluster.weights = FALSE,
  sample.fraction = 0.5,
  mtry = min(ceiling(sqrt(ncol(X)) + 20), ncol(X)),
  min.node.size = 5,
```

```
  honesty = TRUE,
  honesty.fraction = 0.5,
  honesty.prune.leaves = TRUE,
  alpha = 0.05,
  imbalance.penalty = 0,
  ci.group.size = 2,
  tune.parameters = "none",
  tune.num.trees = 10,
  tune.num.reps = 100,
  tune.num.draws = 1000,
  boost.steps = NULL,
  boost.error.reduction = 0.97,
  boost.max.steps = 5,
  boost.trees.tune = 10,
  num.threads = NULL,
  seed = runif(1, 0, .Machine$integer.max)
)
```

## Arguments

| | |
|---|---|
| X | The covariates used in the regression. |
| Y | The outcome. |
| num.trees | Number of trees grown in the forest. Note: Getting accurate confidence intervals generally requires more trees than getting accurate predictions. Default is 2000. |
| sample.weights | Weights given to each observation in estimation. If NULL, each observation receives the same weight. Default is NULL. |
| clusters | Vector of integers or factors specifying which cluster each observation corresponds to. Default is NULL (ignored). |
| equalize.cluster.weights | If FALSE, each unit is given the same weight (so that bigger clusters get more weight). If TRUE, each cluster is given equal weight in the forest. In this case, during training, each tree uses the same number of observations from each drawn cluster: If the smallest cluster has K units, then when we sample a cluster during training, we only give a random K elements of the cluster to the tree-growing procedure. When estimating average treatment effects, each observation is given weight 1/cluster size, so that the total weight of each cluster is the same. Note that, if this argument is FALSE, sample weights may also be directly adjusted via the sample.weights argument. If this argument is TRUE, sample.weights must be set to NULL. Default is FALSE. |
| 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 variables. |
| 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 algorithm reference.

`honesty.fraction`
The fraction of data that will be used for determining splits if honesty = TRUE. Corresponds to set J1 in the notation of the paper. Default is 0.5 (i.e. half of the data is used for determining splits).

`honesty.prune.leaves`
If TRUE, prunes the estimation sample tree such that no leaves are empty. If FALSE, keep 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.

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

`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. Default is 2.

`tune.parameters`
If true, NULL parameters are tuned by cross-validation; if FALSE NULL parameters are set to defaults. Default is FALSE.

`tune.num.trees` The number of trees in each 'mini forest' used to fit the tuning model. Default is 10.

`tune.num.reps` The number of forests used to fit the tuning model. Default is 100.

`tune.num.draws` The number of random parameter values considered when using the model to select the optimal parameters. Default is 1000.

`boost.steps` The number of boosting iterations. If NULL, selected by cross-validation. Default is NULL.

`boost.error.reduction`
If boost.steps is NULL, the percentage of previous steps' error that must be estimated by cross validation in order to take a new step, default 0.97.

`boost.max.steps`
The maximum number of boosting iterations to try when boost.steps=NULL. Default is 5.

`boost.trees.tune`
If boost.steps is NULL, the number of trees used to test a new boosting step when tuning boost.steps. Default is 10.

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

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

## Value

A boosted regression forest object. $error contains the mean debiased error for each step, and $forests contains the trained regression forest for each step.

## Examples

```
# Train a boosted regression forest.
n <- 50
p <- 10
X <- matrix(rnorm(n * p), n, p)
Y <- X[, 1] * rnorm(n)
boosted.forest <- boosted_regression_forest(X, Y)

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

# Predict on out-of-bag training samples.
boost.pred <- predict(boosted.forest)

# Check how many boosting iterations were used
print(length(boosted.forest$forests))
```

---

# causal_forest *Causal forest*

---

## Description

Trains a causal forest that can be used to estimate conditional average treatment effects tau(X). When the treatment assignment 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 partial 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,
  Y.hat = NULL,
  W.hat = NULL,
  num.trees = 2000,
  sample.weights = NULL,
```

```
  clusters = NULL,
  equalize.cluster.weights = FALSE,
  sample.fraction = 0.5,
  mtry = min(ceiling(sqrt(ncol(X)) + 20), ncol(X)),
  min.node.size = 5,
  honesty = TRUE,
  honesty.fraction = 0.5,
  honesty.prune.leaves = TRUE,
  alpha = 0.05,
  imbalance.penalty = 0,
  stabilize.splits = TRUE,
  ci.group.size = 2,
  tune.parameters = "none",
  tune.num.trees = 200,
  tune.num.reps = 50,
  tune.num.draws = 1000,
  compute.oob.predictions = TRUE,
  num.threads = NULL,
  seed = runif(1, 0, .Machine$integer.max)
)
```

## Arguments

| | |
|---|---|
| X | The covariates used in the causal regression. |
| Y | The outcome (must be a numeric vector with no NAs). |
| W | The treatment assignment (must be a binary or real numeric vector with no NAs). |
| Y.hat | Estimates of the expected responses E[Y \| Xi], marginalizing over treatment. If Y.hat = NULL, these are estimated using a separate regression forest. See section 6.1.1 of the GRF paper for further discussion of this quantity. Default is NULL. |
| W.hat | Estimates of the treatment propensities E[W \| Xi]. If W.hat = NULL, these are estimated using a separate regression forest. Default is NULL. |
| num.trees | Number of trees grown in the forest. Note: Getting accurate confidence intervals generally requires more trees than getting accurate predictions. Default is 2000. |
| sample.weights | Weights given to each sample in estimation. If NULL, each observation receives the same weight. Note: To avoid introducing confounding, weights should be independent of the potential outcomes given X. Default is NULL. |
| clusters | Vector of integers or factors specifying which cluster each observation corresponds to. Default is NULL (ignored). |
| equalize.cluster.weights | If FALSE, each unit is given the same weight (so that bigger clusters get more weight). If TRUE, each cluster is given equal weight in the forest. In this case, during training, each tree uses the same number of observations from each drawn cluster: If the smallest cluster has K units, then when we sample a cluster during training, we only give a random K elements of the cluster to the tree-growing procedure. When estimating average treatment effects, each observation is given |

weight 1/cluster size, so that the total weight of each cluster is the same. Note that, if this argument is FALSE, sample weights may also be directly adjusted via the sample.weights argument. If this argument is TRUE, sample.weights must be set to NULL. Default is FALSE.

`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 variables.

`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 algorithm reference.

`honesty.fraction`
: The fraction of data that will be used for determining splits if honesty = TRUE. Corresponds to set J1 in the notation of the paper. Default is 0.5 (i.e. half of the data is used for determining splits).

`honesty.prune.leaves`
: If TRUE, prunes the estimation sample tree such that no leaves are empty. If FALSE, keep 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.

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

`stabilize.splits`
: Whether or not the treatment should be taken into account when determining the imbalance of a split. Default is TRUE.

`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. Default is 2.

`tune.parameters`
: A vector of parameter names to tune. If "all": all tunable parameters are tuned by cross-validation. The following parameters are tunable: ("sample.fraction", "mtry", "min.node.size", "honesty.fraction", "honesty.prune.leaves", "alpha", "imbalance.penalty"). If honesty is FALSE the honesty.* parameters are not tuned. Default is "none" (no parameters are tuned).

`tune.num.trees`
: The number of trees in each 'mini forest' used to fit the tuning model. Default is 200.

`tune.num.reps`
: The number of forests used to fit the tuning model. Default is 50.

`tune.num.draws` The number of random parameter values considered when using the model to select the optimal parameters. Default is 1000.

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

`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.

## Value

A trained causal forest object. If tune.parameters is enabled, then tuning information will be included through the 'tuning.output' attribute.

## Examples

```
# Train a causal forest.
n <- 500
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)

# In some examples, pre-fitting models for Y and W separately may
# be helpful (e.g., if different models use different covariates).
# In some applications, one may even want to get Y.hat and W.hat
# using a completely different method (e.g., boosting).
n <- 2000
p <- 20
X <- matrix(rnorm(n * p), n, p)
TAU <- 1 / (1 + exp(-X[, 3]))
W <- rbinom(n, 1, 1 / (1 + exp(-X[, 1] - X[, 2])))
Y <- pmax(X[, 2] + X[, 3], 0) + rowMeans(X[, 4:6]) / 2 + W * TAU + rnorm(n)

forest.W <- regression_forest(X, W, tune.parameters = "all")
W.hat <- predict(forest.W)$predictions
```

```
forest.Y <- regression_forest(X, Y, tune.parameters = "all")
Y.hat <- predict(forest.Y)$predictions

forest.Y.varimp <- variable_importance(forest.Y)

# Note: Forests may have a hard time when trained on very few variables
# (e.g., ncol(X) = 1, 2, or 3). We recommend not being too aggressive
# in selection.
selected.vars <- which(forest.Y.varimp / mean(forest.Y.varimp) > 0.2)

tau.forest <- causal_forest(X[, selected.vars], Y, W,
  W.hat = W.hat, Y.hat = Y.hat,
  tune.parameters = "all"
)
tau.hat <- predict(tau.forest)$predictions
```

---

`causal_survival_forest` *Causal survival forest (experimental)*

---

## Description

Trains a causal survival forest that can be used to estimate conditional average treatment effects tau(X). When the treatment assignment is unconfounded, we have tau(X) = E[Y(1) - Y(0) | X = x], where Y is the survival time up to a fixed maximum follow-up time. Y(1) and Y(0) are potental outcomes corresponding to the two possible treatment states.

## Usage

```
causal_survival_forest(
  X,
  Y,
  W,
  D,
  W.hat = NULL,
  E1.hat = NULL,
  E0.hat = NULL,
  S.hat = NULL,
  C.hat = NULL,
  lambda.C.hat = NULL,
  failure.times = NULL,
  num.trees = 2000,
  sample.weights = NULL,
  clusters = NULL,
  equalize.cluster.weights = FALSE,
  sample.fraction = 0.5,
```

```
  mtry = min(ceiling(sqrt(ncol(X)) + 20), ncol(X)),
  min.node.size = 5,
  honesty = TRUE,
  honesty.fraction = 0.5,
  honesty.prune.leaves = TRUE,
  alpha = 0.05,
  imbalance.penalty = 0,
  stabilize.splits = TRUE,
  ci.group.size = 2,
  tune.parameters = "none",
  compute.oob.predictions = TRUE,
  num.threads = NULL,
  seed = runif(1, 0, .Machine$integer.max)
)
```

## Arguments

| | |
|---|---|
| X | The covariates. |
| Y | The event time (may be negative). |
| W | The treatment assignment (must be a binary vector with no NAs). |
| D | The event type (0: censored, 1: failure). |
| W.hat | Estimates of the treatment propensities E[W \| Xi]. If W.hat = NULL, these are estimated using a separate regression forest. Default is NULL. |
| E1.hat | Estimates of the expected survival time conditional on being treated E[Y \| X = x, W = 1]. If E1.hat is NULL, then this is estimated with an S-learner using a survival forest. |
| E0.hat | Estimates of the expected survival time conditional on being a control unit E[Y \| X = x, W = 0]. If E0.hat is NULL, then this is estimated with an S-learner using a survival forest. |
| S.hat | Estimates of the conditional survival function S(t, x, w) = P[Y > t \| X = x, W = w]. If S.hat is NULL, this is estimated using a survival forest. If provided: a N*T matrix of survival estimates. The grid should correspond to the T unique events in Y. Default is NULL. |
| C.hat | Estimates of the conditional survival function for the censoring process S_C(t, x, w). If C.hat is NULL, this is estimated using a survival forest. If provided: a N*T matrix of survival estimates. The grid should correspond to the T unique events in Y. Default is NULL. |
| lambda.C.hat | Estimates of the conditional hazard function -d/dt log(S_C(t, x, w)) for the censoring process. If lambda.C.hat is NULL, this is estimated from C.hat using a forward difference. If provided: a matrix of same dimensionality has C.hat. Default is NULL. |
| failure.times | A vector of event times to fit the survival curves at. If NULL, then all the unique event times are used. This speeds up forest estimation by constraining the event grid. Observed event times are rounded down to the last sorted occurance less than or equal to the specified failure time. The time points should be in increasing order. Default is NULL. |

`num.trees` Number of trees grown in the forest. Note: Getting accurate confidence intervals generally requires more trees than getting accurate predictions. Default is 2000.

`sample.weights` Weights given to each sample in estimation. If NULL, each observation receives the same weight. Note: To avoid introducing confounding, weights should be independent of the potential outcomes given X. Sample weights are not used in survival spliting. Default is NULL.

`clusters` Vector of integers or factors specifying which cluster each observation corresponds to. Default is NULL (ignored).

`equalize.cluster.weights`
If FALSE, each unit is given the same weight (so that bigger clusters get more weight). If TRUE, each cluster is given equal weight in the forest. In this case, during training, each tree uses the same number of observations from each drawn cluster: If the smallest cluster has K units, then when we sample a cluster during training, we only give a random K elements of the cluster to the tree-growing procedure. When estimating average treatment effects, each observation is given weight 1/cluster size, so that the total weight of each cluster is the same. Note that, if this argument is FALSE, sample weights may also be directly adjusted via the sample.weights argument. If this argument is TRUE, sample.weights must be set to NULL. Default is FALSE.

`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 variables.

`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 algorithm reference.

`honesty.fraction`
The fraction of data that will be used for determining splits if honesty = TRUE. Corresponds to set J1 in the notation of the paper. Default is 0.5 (i.e. half of the data is used for determining splits).

`honesty.prune.leaves`
If TRUE, prunes the estimation sample tree such that no leaves are empty. If FALSE, keep 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.

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

`stabilize.splits`
: Whether or not the treatment and censoring status should be taken into account when determining the imbalance of a split. The requirement for valid split candidates is the same as in causal_forest with the additional constraint that num.failures(child) >= num.samples(parent) * alpha. Default is TRUE.

`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. Default is 2.

`tune.parameters`
: (Currently only applies to the regression forest used in W.hat estimation) A vector of parameter names to tune. If "all": all tunable parameters are tuned by cross-validation. The following parameters are tunable: ("sample.fraction", "mtry", "min.node.size", "honesty.fraction", "honesty.prune.leaves", "alpha", "imbalance.penalty"). If honesty is FALSE the honesty.* parameters are not tuned. Default is "none" (no parameters are tuned).

`compute.oob.predictions`
: Whether OOB predictions on training set should be precomputed. Default is TRUE.

`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.

## Details

An important assumption for identifying the conditional average treatment effect tau(X) is that there exists a fixed positive constant M such that the probability of observing an event time past the maximum follow-up time Y.max is at least M. This may be an issue with data where most endpoint observations are censored. The suggested resolution is to re-define the estimand as the treatment effect up to some suitable maximum follow-up time Y.max. One can do this in practice by thresholding Y before running causal_survival_forest: ‘D[Y >= Y.max] <- 1‘ and ‘Y[Y >= Y.max] <- Y.max‘. For details see Cui et al. (2020). The computational complexity of this estimator scales with the cardinality of the event times Y. If the number of samples is large and the Y grid dense, consider rounding the event times (or supply a coarser grid with the ‘failure.times‘ argument).

## Value

A trained causal_survival_forest forest object.

## References

Cui, Yifan, Michael R. Kosorok, Erik Sverdrup, Stefan Wager, and Ruoqing Zhu. "Estimating Heterogeneous Treatment Effects with Right-Censored Data via Causal Survival Forests." arXiv preprint arXiv:2001.09887, 2020.

## Examples

```
# Train a standard causal survival forest.
n <- 3000
p <- 5
```

```
X <- matrix(runif(n * p), n, p)
W <- rbinom(n, 1, 0.5)
Y.max <- 1
failure.time <- pmin(rexp(n) * X[, 1] + W, Y.max)
censor.time <- 2 * runif(n)
Y <- pmin(failure.time, censor.time)
D <- as.integer(failure.time <= censor.time)
cs.forest <- causal_survival_forest(X, Y, W, D)

# Predict using the forest.
X.test <- matrix(0.5, 10, p)
X.test[, 1] <- seq(0, 1, length.out = 10)
cs.pred <- predict(cs.forest, X.test, estimate.variance = TRUE)

# Plot the estimated CATEs along with 95% confidence bands.
r.monte.carlo <- rexp(5000)
cate <- rep(NA, 10)
for (i in 1:10) {
  cate[i] <- mean(pmin(r.monte.carlo * X.test[i, 1] + 1, Y.max) -
                    pmin(r.monte.carlo * X.test[i, 1], Y.max))
}
plot(X.test[, 1], cate, type = 'l', col = 'red')
points(X.test[, 1], cs.pred$predictions)
lines(X.test[, 1], cs.pred$predictions + 2 * sqrt(cs.pred$variance.estimates), lty = 2)
lines(X.test[, 1], cs.pred$predictions - 2 * sqrt(cs.pred$variance.estimates), lty = 2)

# Compute a doubly robust estimate of the average treatment effect.
average_treatment_effect(cs.forest)

# Compute the best linear projection on the first covariate.
best_linear_projection(cs.forest, X[, 1])

# Train the forest on a less granular grid.
cs.forest.grid <- causal_survival_forest(X, Y, W, D,
                                         failure.times = seq(min(Y), max(Y), length.out = 50))
plot(X.test[, 1], cs.pred$predictions)
points(X.test[, 1], predict(cs.forest.grid, X.test)$predictions, col = "blue")
```

---

`custom_forest` *Custom forest (removed)*

---

## Description

To build a custom forest, see an existing simpler forest, like regression_forest, for a development template.

## Usage

```
custom_forest(X, Y, ...)
```

### Arguments

| | |
|---|---|
| X | X |
| Y | Y |
| ... | Additional arguments (currently ignored). |

---

`generate_causal_data` *Generate causal forest data*

---

### Description

The following DGPs are available for benchmarking purposes:

- "simple": tau = max(X1, 0), e = 0.4 + 0.2 * 1X1 > 0.
- "aw1": equation (27) of https://arxiv.org/pdf/1510.04342.pdf
- "aw2": equation (28) of https://arxiv.org/pdf/1510.04342.pdf
- "aw3": confounding is from "aw1" and tau is from "aw2"
- "aw3reverse": Same as aw3, but HTEs anticorrelated with baseline
- "ai1": "Setup 1" from section 6 of https://arxiv.org/pdf/1504.01132.pdf
- "ai2": "Setup 2" from section 6 of https://arxiv.org/pdf/1504.01132.pdf
- "kunzel": "Simulation 1" from A.1 in https://arxiv.org/pdf/1706.03461.pdf
- "nw1": "Setup A" from Section 4 of https://arxiv.org/pdf/1712.04912.pdf
- "nw2": "Setup B" from Section 4 of https://arxiv.org/pdf/1712.04912.pdf
- "nw3": "Setup C" from Section 4 of https://arxiv.org/pdf/1712.04912.pdf
- "nw4": "Setup D" from Section 4 of https://arxiv.org/pdf/1712.04912.pdf

### Usage

```
generate_causal_data(
  n,
  p,
  sigma.m = 1,
  sigma.tau = 0.1,
  sigma.noise = 1,
 dgp = c("simple", "aw1", "aw2", "aw3", "aw3reverse", "ai1", "ai2", "kunzel", "nw1",
    "nw2", "nw3", "nw4")
)
```

### Arguments

| | |
|---|---|
| n | The number of observations. |
| p | The number of covariates (note: the minimum varies by DGP). |
| sigma.m | The standard deviation of the unconditional mean of Y. Default is 1. |
| sigma.tau | The standard deviation of the treatment effect. Default is 0.1. |
| sigma.noise | The conditional variance of Y. Default is 1. |
| dgp | The kind of dgp. Default is "simple". |

## Details

Each DGP is parameterized by X: observables, m: conditional mean of Y, tau: treatment effect, e: propensity scores, V: conditional variance of Y.

The following rescaled data is returned m = m / sd(m) * sigma.m, tau = tau / sd(tau) * sigma.tau, V = V / mean(V) * sigma.noise^2, W = rbinom(e), Y = m + (W - e) * tau + sqrt(V) + rnorm(n).

## Value

A list consisting of: X, Y, W, tau, m, e, dgp.

## Examples

```
# Generate simple benchmark data
data <- generate_causal_data(100, 5, dgp = "simple")
# Generate data from Wager and Athey (2018)
data <- generate_causal_data(100, 5, dgp = "aw1")
data2 <- generate_causal_data(100, 5, dgp = "aw2")
```

---

# generate_causal_survival_data

*Simulate causal survival data*

---

## Description

The following DGPs are available for benchmarking purposes, T is the failure time and C the censoring time:

- "simple1": T = X1*eps + W, C ~ U(0, 2) where eps ~ Exp(1) and Y.max = 1.
- "type1": T is drawn from an accelerated failure time model and C from a Cox model (scenario 1 in https://arxiv.org/abs/2001.09887)
- "type2": T is drawn from a proportional hazard model and C from a accelerated failure time (scenario 2 in https://arxiv.org/abs/2001.09887)
- "type3": T and C are drawn from a Poisson distribution (scenario 3 in https://arxiv.org/abs/2001.09887)
- "type4": T and C are drawn from a Poisson distribution (scenario 4 in https://arxiv.org/abs/2001.09887)
- "type5": is similar to "type2" but with censoring generated from an accelerated failure time model.

## Usage

```
generate_causal_survival_data(
  n,
  p,
  Y.max = NULL,
  X = NULL,
```

```
  n.mc = 10000,
  dgp = c("simple1", "type1", "type2", "type3", "type4", "type5")
)
```

## Arguments

| | |
|---|---|
| n | The number of samples. |
| p | The number of covariates. |
| Y.max | The maximum follow-up time (optional). |
| X | The covariates (optional). |
| n.mc | The number of monte carlo draws to estimate the treatment effect with. Default is 10000. |
| dgp | The type of DGP. |

## Value

A list with entries: 'X': the covariates, 'Y': the event times, 'W': the treatment indicator, 'D': the censoring indicator, 'cate': the treatment effect estimated by monte carlo, 'cate.sign': the true sign of the cate for ITR comparison, 'dgp': the dgp name, 'Y.max': the maximum follow-up time.

## Examples

```
# Generate data
n <- 1000
p <- 5
data <- generate_causal_survival_data(n, p)
# Get true CATE on a test set
X.test <- matrix(seq(0, 1, length.out = 5), 5, p)
cate.test <- generate_causal_survival_data(n, p, X = X.test)$cate
```

---

# get_forest_weights *Given a trained forest and test data, compute the kernel weights for each test point.*

---

## Description

During normal prediction, these weights (named alpha in the GRF paper) 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_forest_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). |
| 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

```
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)
forest.weights.oob <- get_forest_weights(rrf)

n.test <- 15
X.test <- matrix(2 * runif(n.test * p) - 1, n.test, p)
forest.weights <- get_forest_weights(rrf, X.test)
```

---

## get_leaf_node *Find the leaf node for a test sample.*

---

### Description

Given a GRF tree object, compute the leaf node a test sample falls into. The nodes in a GRF tree are numbered breadth first, and the returned numbers will be the leaf integer according to this ordering. To get kernel weights based on leaf membership, see the function get_forest_weights.

### Usage

```
get_leaf_node(tree, newdata, node.id = TRUE)
```

### Arguments

| | |
|---|---|
| tree | A GRF tree object (retrieved by 'get_tree'). |
| newdata | Points at which leaf predictions should be made. |
| node.id | Boolean indicating whether to return the node.id for each query sample (default), or if FALSE, a list of node numbers with the samples contained. |

## Value

A vector of integers indicating the leaf number for each sample in the given tree.

## Examples

```
p <- 10
n <- 100
X <- matrix(2 * runif(n * p) - 1, n, p)
Y <- (X[, 1] > 0) + 2 * rnorm(n)
r.forest <- regression_forest(X, Y, num.tree = 50)

n.test <- 5
X.test <- matrix(2 * runif(n.test * p) - 1, n.test, p)
tree <- get_tree(r.forest, 1)
# Get a vector of node numbers for each sample.
get_leaf_node(tree, X.test)
# Get a list of samples per node.
get_leaf_node(tree, X.test, node.id = FALSE)
```

---

# get_sample_weights *Retrieve forest weights (renamed to get_forest_weights)*

---

## Description

Retrieve forest weights (renamed to get_forest_weights)

## Usage

```
get_sample_weights(forest, ...)
```

## Arguments

| | |
|---|---|
| forest | The trained forest. |
| ... | Additional arguments (currently ignored). |

## get_scores *Compute doubly robust scores for a GRF forest object*

### Description

Compute doubly robust scores for a GRF forest object

### Usage

```
get_scores(forest, ...)
```

### Arguments

| | |
|---|---|
| forest | A grf forest object |
| ... | Additional arguments |

### Value

A vector of scores

## get_scores.causal_forest *Compute doubly robust scores for a causal forest.*

### Description

Compute doubly robust (AIPW) scores for average treatment effect estimation or average partial effect estimation with continuous treatment, using a causal forest. Under regularity conditions, the average of the DR.scores is an efficient estimate of the average treatment effect.

### Usage

```
## S3 method for class 'causal_forest'
get_scores(
  forest,
  subset = NULL,
  debiasing.weights = NULL,
  num.trees.for.weights = 500,
  ...
)
```

### Arguments

| | |
|---|---|
| forest | A trained causal forest. |
| subset | Specifies subset of the training examples over which we estimate the ATE. WARNING: For valid statistical performance, the subset should be defined only using features Xi, not using the treatment Wi or the outcome Yi. |
| debiasing.weights | A vector of length n (or the subset length) of debiasing weights. If NULL (default) they are obtained via inverse-propensity weighting in the case of binary treatment or by estimating Var[W \| X = x] using a new forest in the case of a continuous treatment. |
| num.trees.for.weights | Number of trees used to estimate Var[W \| X = x]. Note: this argument is only used when debiasing.weights = NULL. |
| ... | Additional arguments (currently ignored). |

### Value

A vector of scores.

### References

Farrell, Max H. "Robust inference on average treatment effects with possibly more covariates than observations." Journal of Econometrics 189(1), 2015.

Graham, Bryan S., and Cristine Campos de Xavier Pinto. "Semiparametrically efficient estimation of the average linear regression function." arXiv preprint arXiv:1810.12511, 2018.

Hirshberg, David A., and Stefan Wager. "Augmented minimax linear estimation." arXiv preprint arXiv:1712.00038, 2017.

Robins, James M., and Andrea Rotnitzky. "Semiparametric efficiency in multivariate regression models with missing data." Journal of the American Statistical Association 90(429), 1995.

---

## get_scores.causal_survival_forest

*Compute doubly robust scores for a causal survival forest.*

---

### Description

For details see section 3.2 and equation (20) in the causal survival forest paper.

### Usage

```
## S3 method for class 'causal_survival_forest'
get_scores(forest, subset = NULL, ...)
```

### Arguments

| | |
|---|---|
| forest | A trained causal survival forest. |
| subset | Specifies subset of the training examples over which we estimate the ATE. WARNING: For valid statistical performance, the subset should be defined only using features Xi, not using the treatment Wi or the outcome Yi. |
| ... | Additional arguments (currently ignored). |

### Value

A vector of scores.

---

## get_scores.instrumental_forest

*Doubly robust scores for estimating the average conditional local average treatment effect.*

---

### Description

Given an outcome Y, treatment W and instrument Z, the (conditional) local average treatment effect is tau(x) = Cov[Y, Z | X = x] / Cov[W, Z | X = x]. This is the quantity that is estimated with an instrumental forest. It can be intepreted causally in various ways. Given a homogeneity assumption, tau(x) is simply the CATE at x. When W is binary and there are no "defiers", Imbens and Angrist (1994) show that tau(x) can be interpreted as an average treatment effect on compliers. This doubly robust scores provided here are for estimating tau = E[tau(X)].

### Usage

```
## S3 method for class 'instrumental_forest'
get_scores(
  forest,
  subset = NULL,
  debiasing.weights = NULL,
  compliance.score = NULL,
  num.trees.for.weights = 500,
  ...
)
```

### Arguments

| | |
|---|---|
| forest | A trained instrumental forest. |
| subset | Specifies subset of the training examples over which we estimate the ATE. WARNING: For valid statistical performance, the subset should be defined only using features Xi, not using the treatment Wi or the outcome Yi. |

`debiasing.weights`
: A vector of length n (or the subset length) of debiasing weights. If NULL (default) these are obtained via the appropriate doubly robust score construction, e.g., in the case of causal_forests with a binary treatment, they are obtained via inverse-propensity weighting.

`compliance.score`
: An estimate of the causal effect of Z on W, i.e., Delta(X) = E[W | X, Z = 1] - E[W | X, Z = 0], which can then be used to produce debiasing.weights. If not provided, this is estimated via an auxiliary causal forest.

`num.trees.for.weights`
: In some cases (e.g., with causal forests with a continuous treatment), we need to train auxiliary forests to learn debiasing weights. This is the number of trees used for this task. Note: this argument is only used when debiasing.weights = NULL.

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

## Value

A vector of scores.

## References

Aronow, Peter M., and Allison Carnegie. "Beyond LATE: Estimation of the average treatment effect with an instrumental variable." Political Analysis 21(4), 2013.

Chernozhukov, Victor, Juan Carlos Escanciano, Hidehiko Ichimura, Whitney K. Newey, and James M. Robins. "Locally robust semiparametric estimation." arXiv preprint arXiv:1608.00033, 2016.

Imbens, Guido W., and Joshua D. Angrist. "Identification and Estimation of Local Average Treatment Effects." Econometrica 62(2), 1994.

---

# `get_scores.multi_arm_causal_forest` *Compute doubly robust scores for a multi arm causal forest.*

---

## Description

Compute doubly robust (AIPW) scores for average treatment effect estimation using a multi arm causal forest. Under regularity conditions, the average of the DR.scores is an efficient estimate of the average treatment effect.

## Usage

```
## S3 method for class 'multi_arm_causal_forest'
get_scores(forest, subset = NULL, ...)
```

### Arguments

| | |
|---|---|
| forest | A trained multi arm causal forest. |
| subset | Specifies subset of the training examples over which we estimate the ATE. WARNING: For valid statistical performance, the subset should be defined only using features Xi, not using the treatment Wi or the outcome Yi. |
| ... | Additional arguments (currently ignored). |

### Value

An array of scores for each contrast and outcome.

---

## 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 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

```
# 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
```

---

`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,
  Y.hat = NULL,
  W.hat = NULL,
  Z.hat = NULL,
  num.trees = 2000,
  sample.weights = NULL,
  clusters = NULL,
  equalize.cluster.weights = FALSE,
  sample.fraction = 0.5,
  mtry = min(ceiling(sqrt(ncol(X)) + 20), ncol(X)),
  min.node.size = 5,
  honesty = TRUE,
  honesty.fraction = 0.5,
  honesty.prune.leaves = TRUE,
  alpha = 0.05,
  imbalance.penalty = 0,
```

```
  stabilize.splits = TRUE,
  ci.group.size = 2,
  reduced.form.weight = 0,
  tune.parameters = "none",
  tune.num.trees = 200,
  tune.num.reps = 50,
  tune.num.draws = 1000,
  compute.oob.predictions = TRUE,
  num.threads = NULL,
  seed = runif(1, 0, .Machine$integer.max)
)
```

## 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). |
| Y.hat | Estimates of the expected responses E[Y \| Xi], marginalizing over treatment. If Y.hat = NULL, these are estimated using a separate regression forest. Default is NULL. |
| W.hat | Estimates of the treatment propensities E[W \| Xi]. If W.hat = NULL, these are estimated using a separate regression forest. Default is NULL. |
| Z.hat | Estimates of the instrument propensities E[Z \| Xi]. If Z.hat = NULL, these are estimated using a separate regression forest. Default is NULL. |
| num.trees | Number of trees grown in the forest. Note: Getting accurate confidence intervals generally requires more trees than getting accurate predictions. Default is 2000. |
| sample.weights | Weights given to each observation in estimation. If NULL, each observation receives equal weight. Default is NULL. |
| clusters | Vector of integers or factors specifying which cluster each observation corresponds to. Default is NULL (ignored). |
| equalize.cluster.weights | If FALSE, each unit is given the same weight (so that bigger clusters get more weight). If TRUE, each cluster is given equal weight in the forest. In this case, during training, each tree uses the same number of observations from each drawn cluster: If the smallest cluster has K units, then when we sample a cluster during training, we only give a random K elements of the cluster to the tree-growing procedure. When estimating average treatment effects, each observation is given weight 1/cluster size, so that the total weight of each cluster is the same. Note that, if this argument is FALSE, sample weights may also be directly adjusted via the sample.weights argument. If this argument is TRUE, sample.weights must be set to NULL. Default is FALSE. |
| 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 variables.

`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 algorithm reference.

`honesty.fraction`
The fraction of data that will be used for determining splits if honesty = TRUE. Corresponds to set J1 in the notation of the paper. Default is 0.5 (i.e. half of the data is used for determining splits).

`honesty.prune.leaves`
If TRUE, prunes the estimation sample tree such that no leaves are empty. If FALSE, keep 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.

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

`stabilize.splits`
Whether or not the instrument should be taken into account when determining the imbalance of a split. Default is TRUE.

`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. Default is 2.

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

`tune.parameters`
(experimental) A vector of parameter names to tune. If "all": all tunable parameters are tuned by cross-validation. The following parameters are tunable: ("sample.fraction", "mtry", "min.node.size", "honesty.fraction", "honesty.prune.leaves", "alpha", "imbalance.penalty"). If honesty is FALSE the honesty.* parameters are not tuned. Default is "none" (no parameters are tuned).

`tune.num.trees` The number of trees in each 'mini forest' used to fit the tuning model. Default is 200.

`tune.num.reps` The number of forests used to fit the tuning model. Default is 50.

`tune.num.draws` The number of random parameter values considered when using the model to select the optimal parameters. Default is 1000.

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

| | |
|---|---|
| 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. |

## Value

A trained instrumental forest object.

## Examples

```
# Train an instrumental forest.
n <- 2000
p <- 5
X <- matrix(rbinom(n * p, 1, 0.5), n, p)
Z <- rbinom(n, 1, 0.5)
Q <- rbinom(n, 1, 0.5)
W <- Q * Z
tau <-  X[, 1] / 2
Y <- rowSums(X[, 1:3]) + tau * W + Q + rnorm(n)
iv.forest <- instrumental_forest(X, Y, W, Z)

# Predict on out-of-bag training samples.
iv.pred <- predict(iv.forest)
```

---

# ll_regression_forest *Local linear forest*

---

## Description

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

## Usage

```
ll_regression_forest(
  X,
  Y,
  enable.ll.split = FALSE,
  ll.split.weight.penalty = FALSE,
  ll.split.lambda = 0.1,
  ll.split.variables = NULL,
  ll.split.cutoff = NULL,
  num.trees = 2000,
  clusters = NULL,
  equalize.cluster.weights = FALSE,
```

```
  sample.fraction = 0.5,
  mtry = min(ceiling(sqrt(ncol(X)) + 20), ncol(X)),
  min.node.size = 5,
  honesty = TRUE,
  honesty.fraction = 0.5,
  honesty.prune.leaves = TRUE,
  alpha = 0.05,
  imbalance.penalty = 0,
  ci.group.size = 2,
  tune.parameters = "none",
  tune.num.trees = 50,
  tune.num.reps = 100,
  tune.num.draws = 1000,
  num.threads = NULL,
  seed = runif(1, 0, .Machine$integer.max)
)
```

## Arguments

| | |
|---|---|
| `X` | The covariates used in the regression. |
| `Y` | The outcome. |
| `enable.ll.split` | (experimental) Optional choice to make forest splits based on ridge residuals as opposed to standard CART splits. Defaults to FALSE. |
| `ll.split.weight.penalty` | If using local linear splits, user can specify whether or not to use a covariance ridge penalty, analogously to the prediction case. Defaults to FALSE. |
| `ll.split.lambda` | Ridge penalty for splitting. Defaults to 0.1. |
| `ll.split.variables` | Linear correction variables for splitting. Defaults to all variables. |
| `ll.split.cutoff` | Enables the option to use regression coefficients from the full dataset for LL splitting once leaves get sufficiently small. Leaf size after which we use the overall beta. Defaults to the square root of the number of samples. If desired, users can enforce no regulation (i.e., using the leaf betas at each step) by setting this parameter to zero. |
| `num.trees` | Number of trees grown in the forest. Note: Getting accurate confidence intervals generally requires more trees than getting accurate predictions. Default is 2000. |
| `clusters` | Vector of integers or factors specifying which cluster each observation corresponds to. Default is NULL (ignored). |
| `equalize.cluster.weights` | If FALSE, each unit is given the same weight (so that bigger clusters get more weight). If TRUE, each cluster is given equal weight in the forest. In this case, during training, each tree uses the same number of observations from each drawn cluster: If the smallest cluster has K units, then when we sample a cluster during training, we only give a random K elements of the cluster to the tree-growing |

procedure. When estimating average treatment effects, each observation is given weight 1/cluster size, so that the total weight of each cluster is the same. Default is FALSE.

`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 variables.

`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 algorithm reference.

`honesty.fraction`
: The fraction of data that will be used for determining splits if honesty = TRUE. Corresponds to set J1 in the notation of the paper. Default is 0.5 (i.e. half of the data is used for determining splits).

`honesty.prune.leaves`
: If TRUE, prunes the estimation sample tree such that no leaves are empty. If FALSE, keep 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.

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

`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. Default is 1.

`tune.parameters`
: If true, NULL parameters are tuned by cross-validation; if FALSE NULL parameters are set to defaults. Default is FALSE. Currently, local linear tuning is based on regression forest fit, and is only supported for 'enable.ll.split = FALSE'.

`tune.num.trees`
: The number of trees in each 'mini forest' used to fit the tuning model. Default is 10.

`tune.num.reps`
: The number of forests used to fit the tuning model. Default is 100.

`tune.num.draws`
: The number of random parameter values considered when using the model to select the optimal parameters. Default is 1000.

`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.

## Value

A trained local linear forest object.

## Examples

```
# Train a standard regression forest.
n <- 50
p <- 10
X <- matrix(rnorm(n * p), n, p)
Y <- X[, 1] * rnorm(n)
forest <- ll_regression_forest(X, Y)
```

---

`merge_forests` *Merges a list of forests that were grown using the same data into one large forest.*

---

## Description

Merges a list of forests that were grown using the same data into one large forest.

## Usage

```
merge_forests(forest_list, compute.oob.predictions = TRUE)
```

## Arguments

| | |
|---|---|
| `forest_list` | A 'list' of forests to be concatenated. All forests must be of the same type, and the type must be a subclass of 'grf'. In addition, all forests must have the same 'ci.group.size'. Other tuning parameters (e.g. alpha, mtry, min.node.size, imbalance.penalty) are allowed to differ across forests. |
| `compute.oob.predictions` | Whether OOB predictions on training set should be precomputed. Note that even if OOB predictions have already been precomputed for the forests in 'forest_list', those predictions are not used. Instead, a new set of oob predictions is computed anew using the larger forest. Default is TRUE. |

## Value

A single forest containing all the trees in each forest in the input list.

## Examples

```
# Train standard regression forests
n <- 50
p <- 10
X <- matrix(rnorm(n * p), n, p)
Y <- X[, 1] * rnorm(n)
r.forest1 <- regression_forest(X, Y, compute.oob.predictions = FALSE, num.trees = 100)
r.forest2 <- regression_forest(X, Y, compute.oob.predictions = FALSE, num.trees = 100)

# Join the forests together. The resulting forest will contain 200 trees.
big_rf <- merge_forests(list(r.forest1, r.forest2))
```

---

# multi_arm_causal_forest

*Multi-arm causal forest (experimental)*

---

## Description

Trains a causal forest that can be used to estimate conditional average treatment effects tau_k(X). When the treatment assignment W is {1, ..., K} and unconfounded, we have tau_k(X) = E[Y(k) - Y(1) | X = x] where Y(k) and Y(1) are potential outcomes corresponding to the treatment state for arm k and the baseline arm 1.

## Usage

```
multi_arm_causal_forest(
  X,
  Y,
  W,
  Y.hat = NULL,
  W.hat = NULL,
  num.trees = 2000,
  sample.weights = NULL,
  clusters = NULL,
  equalize.cluster.weights = FALSE,
  sample.fraction = 0.5,
  mtry = min(ceiling(sqrt(ncol(X)) + 20), ncol(X)),
  min.node.size = 5,
  honesty = TRUE,
  honesty.fraction = 0.5,
  honesty.prune.leaves = TRUE,
  alpha = 0.05,
  imbalance.penalty = 0,
  stabilize.splits = TRUE,
```

```
  ci.group.size = 2,
  compute.oob.predictions = TRUE,
  num.threads = NULL,
  seed = runif(1, 0, .Machine$integer.max)
)
```

## Arguments

| | |
|---|---|
| X | The covariates used in the causal regression. |
| Y | The outcome (must be a numeric vector or matrix [one column per outcome] with no NAs). |
| W | The treatment assignment (must be a factor vector with no NAs). The reference treatment is set to the first treatment according to the ordinality of the factors, this can be changed with the 'relevel' function in R. |
| Y.hat | Estimates of the expected responses E[Y \| Xi], marginalizing over treatment. If Y.hat = NULL, these are estimated using a separate multi-task regression forest. Default is NULL. |
| W.hat | Matrix with estimates of the treatment propensities E[Wk \| Xi]. If W.hat = NULL, these are estimated using a probability forest. |
| num.trees | Number of trees grown in the forest. Note: Getting accurate confidence intervals generally requires more trees than getting accurate predictions. Default is 2000. |
| sample.weights | Weights given to each sample in estimation. If NULL, each observation receives the same weight. Note: To avoid introducing confounding, weights should be independent of the potential outcomes given X. Default is NULL. |
| clusters | Vector of integers or factors specifying which cluster each observation corresponds to. Default is NULL (ignored). |
| equalize.cluster.weights | If FALSE, each unit is given the same weight (so that bigger clusters get more weight). If TRUE, each cluster is given equal weight in the forest. In this case, during training, each tree uses the same number of observations from each drawn cluster: If the smallest cluster has K units, then when we sample a cluster during training, we only give a random K elements of the cluster to the tree-growing procedure. When estimating average treatment effects, each observation is given weight 1/cluster size, so that the total weight of each cluster is the same. Note that, if this argument is FALSE, sample weights may also be directly adjusted via the sample.weights argument. If this argument is TRUE, sample.weights must be set to NULL. Default is FALSE. |
| 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 variables. |
| 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 algorithm reference.

`honesty.fraction`
The fraction of data that will be used for determining splits if honesty = TRUE. Corresponds to set J1 in the notation of the paper. Default is 0.5 (i.e. half of the data is used for determining splits).

`honesty.prune.leaves`
If TRUE, prunes the estimation sample tree such that no leaves are empty. If FALSE, keep 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.

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

`stabilize.splits`
Whether or not the treatment should be taken into account when determining the imbalance of a split. It is an exact extension of the single-arm constraints (detailed in the causal forest algorithm reference) to multiple arms, where the constraints apply to each treatment arm independently. Default is TRUE.

`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. Default is 2. (Confidence intervals are currently only supported for univariate outcomes Y).

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

`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.

## Details

This forest fits a multi-arm treatment estimate following the multivariate extension of the "R-learner" suggested in Nie and Wager (2021), with kernel weights derived by the GRF algortim (Athey, Tibshirani, and Wager, 2019). In particular, with K arms, and W encoded as {0, 1}^(K-1), we estimate, for a target sample x, and a chosen baseline arm:

$$\hat{\tau}(x) = argmin_\tau \left\{ \sum_{i=1}^n \alpha_i(x) \left( Y_i - \hat{m}^{(-i)}(X_i) - c(x) - \left\langle W_i - \hat{e}^{(-i)}(X_i),\, \tau(X_i) \right\rangle \right)^2 \right\},$$

where the angle brackets indicates an inner product, e(X) = E[W | X = x] is a (vector valued) generalized propensity score, and m(x) = E[Y | X = x]. The forest weights alpha(x) are derived from a generalized random forest splitting on the vector-valued gradient of tau(x). (The intercept c(x) is a nuisance parameter not directly estimated). By default, e(X) and m(X) are estimated using two separate random forests, a probability forest and regression forest respectively (optionally provided

through the arguments W.hat and Y.hat). The k-th element of tau(x) measures the conditional average treatment effect of the k-th treatment arm at X = x for k = 1, ..., K-1. The treatment effect for multiple outcomes can be estimated jointly (i.e. Y can be vector-valued) - in which case the splitting rule takes into account all outcomes simultaneously (specifically, we concatenate the gradient vector for each outcome).

For a single treatment, this forest is equivalent to a causal forest, however, they may produce different results due to differences in numerics.

## Value

A trained multi arm causal forest object.

## References

Athey, Susan, Julie Tibshirani, and Stefan Wager. "Generalized Random Forests". Annals of Statistics, 47(2), 2019.

Nie, Xinkun, and Stefan Wager. "Quasi-Oracle Estimation of Heterogeneous Treatment Effects". Biometrika, 108(2), 2021.

## Examples

```
# Train a multi arm causal forest.
n <- 500
p <- 10
X <- matrix(rnorm(n * p), n, p)
W <- as.factor(sample(c("A", "B", "C"), n, replace = TRUE))
Y <- X[, 1] + X[, 2] * (W == "B") - 1.5 * X[, 2] * (W == "C") + rnorm(n)
mc.forest <- multi_arm_causal_forest(X, Y, W)

# Predict contrasts (out-of-bag) using the forest.
# By default, the first ordinal treatment is used as baseline ("A" in this example),
# giving two contrasts tau_B = Y(B) - Y(A), tau_C = Y(C) - Y(A)
mc.pred <- predict(mc.forest)
# Fitting several outcomes jointly is supported, and the returned prediction array has
# dimension [num.samples, num.contrasts, num.outcomes]. Since num.outcomes is one in
# this example, we can `drop()` this singleton dimension using the `[,,]` shorthand.
tau.hat <- mc.pred$predictions[,,]

plot(X[, 2], tau.hat[, "B - A"], ylab = "tau.contrast")
abline(0, 1, col = "red")
points(X[, 2], tau.hat[, "C - A"], col = "blue")
abline(0, -1.5, col = "red")
legend("topleft", c("B - A", "C - A"), col = c("black", "blue"), pch = 19)

# The average treatment effect of the arms with "A" as baseline.
average_treatment_effect(mc.forest)

# The conditional response surfaces mu_k(X) for a single outcome can be reconstructed from
# the contrasts tau_k(x), the treatment propensities e_k(x), and the conditional mean m(x).
# Given treatment "A" as baseline we have:
```

```
# m(x) := E[Y | X] = E[Y(A) | X] + E[W_B (Y(B) - Y(A))] + E[W_C (Y(C) - Y(A))]
# which given unconfoundedness is equal to:
# m(x) = mu(A, x) + e_B(x) tau_B(X) + e_C(x) tau_C(x)
# Rearranging and plugging in the above expressions, we obtain the following estimates
# * mu(A, x) = m(x) - e_B(x) tau_B(x) - e_C(x) tau_C(x)
# * mu(B, x) = m(x) + (1 - e_B(x)) tau_B(x) - e_C(x) tau_C(x)
# * mu(C, x) = m(x) - e_B(x) tau_B(x) + (1 - e_C(x)) tau_C(x)
Y.hat <- mc.forest$Y.hat
W.hat <- mc.forest$W.hat

muA <- Y.hat - W.hat[, "B"] * tau.hat[, "B - A"] - W.hat[, "C"] * tau.hat[, "C - A"]
muB <- Y.hat + (1 - W.hat[, "B"]) * tau.hat[, "B - A"] - W.hat[, "C"] * tau.hat[, "C - A"]
muC <- Y.hat - W.hat[, "B"] * tau.hat[, "B - A"] + (1 - W.hat[, "C"]) * tau.hat[, "C - A"]

# These can also be obtained with some array manipulations.
# (the first column is always the baseline arm)
Y.hat.baseline <- Y.hat - rowSums(W.hat[, -1, drop = FALSE] * tau.hat)
mu.hat.matrix <- cbind(Y.hat.baseline, c(Y.hat.baseline) + tau.hat)
colnames(mu.hat.matrix) <- levels(W)
head(mu.hat.matrix)

# The reference level for contrast prediction can be changed with `relevel`.
# Fit and predict with treatment B as baseline:
W <- relevel(W, ref = "B")
mc.forest.B <- multi_arm_causal_forest(X, Y, W)
```

---

`multi_regression_forest` *Multi-task regression forest*

---

## Description

Trains a regression forest that can be used to estimate the conditional mean functions mu_i(x) = E[Y_i | X = x]

## Usage

```
multi_regression_forest(
  X,
  Y,
  num.trees = 2000,
  sample.weights = NULL,
  clusters = NULL,
  equalize.cluster.weights = FALSE,
  sample.fraction = 0.5,
  mtry = min(ceiling(sqrt(ncol(X)) + 20), ncol(X)),
  min.node.size = 5,
```

```
  honesty = TRUE,
  honesty.fraction = 0.5,
  honesty.prune.leaves = TRUE,
  alpha = 0.05,
  imbalance.penalty = 0,
  compute.oob.predictions = TRUE,
  num.threads = NULL,
  seed = runif(1, 0, .Machine$integer.max)
)
```

## Arguments

| | |
|---|---|
| X | The covariates used in the regression. |
| Y | The outcomes (must be a numeric vector/matrix with no NAs). |
| num.trees | Number of trees grown in the forest. Note: Getting accurate confidence intervals generally requires more trees than getting accurate predictions. Default is 2000. |
| sample.weights | Weights given to an observation in estimation. If NULL, each observation is given the same weight. Default is NULL. |
| clusters | Vector of integers or factors specifying which cluster each observation corresponds to. Default is NULL (ignored). |
| equalize.cluster.weights | If FALSE, each unit is given the same weight (so that bigger clusters get more weight). If TRUE, each cluster is given equal weight in the forest. In this case, during training, each tree uses the same number of observations from each drawn cluster: If the smallest cluster has K units, then when we sample a cluster during training, we only give a random K elements of the cluster to the tree-growing procedure. When estimating average treatment effects, each observation is given weight 1/cluster size, so that the total weight of each cluster is the same. Note that, if this argument is FALSE, sample weights may also be directly adjusted via the sample.weights argument. If this argument is TRUE, sample.weights must be set to NULL. Default is FALSE. |
| 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 variables. |
| 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 algorithm reference. |
| honesty.fraction | The fraction of data that will be used for determining splits if honesty = TRUE. Corresponds to set J1 in the notation of the paper. Default is 0.5 (i.e. half of the data is used for determining splits). |

honesty.prune.leaves
: If TRUE, prunes the estimation sample tree such that no leaves are empty. If FALSE, keep 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.

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. Default is TRUE.

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.

## Value

A trained multi regression forest object.

## Examples

```
# Train a standard regression forest.
n <- 500
p <- 5
X <- matrix(rnorm(n * p), n, p)
Y <-  X[, 1, drop = FALSE] %*% cbind(1, 2) + rnorm(n)
mr.forest <- multi_regression_forest(X, Y)

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

# Predict on out-of-bag training samples.
mr.pred <- predict(mr.forest)
```

---

# plot.grf_tree *Plot a GRF tree object.*

---

## Description

The direction NAs are sent are indicated with the arrow fill. An empty arrow indicates that NAs are sent that way. If trained without missing values, both arrows are filled.

## Usage

```
## S3 method for class 'grf_tree'
plot(x, include.na.path = NULL, ...)
```

## Arguments

| | |
|---|---|
| x | The tree to plot |
| include.na.path | A boolean toggling whether to include the path of missing values or not. It defaults to whether the forest was trained with NAs. |
| ... | Additional arguments (currently ignored). |

## Examples

```
## Not run:
# Plot a tree in the forest (requires the `DiagrammeR` package).
n <- 500
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)
plot(tree <- get_tree(c.forest, 1))
# Compute the leaf nodes the first five samples falls into.
leaf.nodes <- get_leaf_node(tree, X[1:5, ])

# Saving a plot in .svg can be done with the `DiagrammeRsvg` package.
install.packages("DiagrammeRsvg")
tree.plot = plot(tree)
cat(DiagrammeRsvg::export_svg(tree.plot), file = 'plot.svg')

## End(Not run)
```

---

# predict.boosted_regression_forest

*Predict with a boosted regression forest.*

---

## Description

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

## Usage

```
## S3 method for class 'boosted_regression_forest'
predict(
  object,
  newdata = NULL,
  boost.predict.steps = 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). Note that this matrix should have the number of columns as the training matrix, and that the columns must appear in the same order |
| boost.predict.steps | Number of boosting iterations to use for prediction. If blank, uses the full number of steps for the object given |
| num.threads | the number of threads used in prediction |
| ... | Additional arguments (currently ignored). |

## Value

A vector of predictions.

## Examples

```
# Train a boosted regression forest.
n <- 50
p <- 10
X <- matrix(rnorm(n * p), n, p)
Y <- X[, 1] * rnorm(n)
r.boosted.forest <- boosted_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.boosted.forest, X.test)

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

`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,
  linear.correction.variables = NULL,
  ll.lambda = NULL,
  ll.weight.penalty = FALSE,
  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). Note that this matrix should have the number of columns as the training matrix, and that the columns must appear in the same order. |
| `linear.correction.variables` | Optional subset of indexes for variables to be used in local linear prediction. If NULL, standard GRF prediction is used. Otherwise, we run a locally weighted linear regression on the included variables. Please note that this is a beta feature still in development, and may slow down prediction considerably. Defaults to NULL. |
| `ll.lambda` | Ridge penalty for local linear predictions. Defaults to NULL and will be cross-validated. |
| `ll.weight.penalty` | Option to standardize ridge penalty by covariance (TRUE), or penalize all covariates equally (FALSE). Penalizes equally by default. |
| `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 estimates of the error and (optionally) its variance estimates. Column 'predictions' contains estimates of the conditional average treatent effect (CATE). The square-root of column 'variance.estimates' is the standard error of CATE. For out-of-bag estimates, we also output the following error measures. First, column 'debiased.error' contains estimates of the 'R-loss' criterion, (See Nie and Wager 2017 for a justification). Second, column 'excess.error' contains jackknife estimates of the Monte-carlo error (Wager, Hastie, Efron 2014), a measure of how unstable estimates are if we grow forests of the same size on the same data set. The sum of 'debiased.error' and 'excess.error' is the raw error attained by the current forest, and 'debiased.error' alone is an estimate of the error attained by a forest with an infinite number of trees. We recommend that users grow enough forests to make the 'excess.error' negligible.

## Examples

```
# 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)
```

---

# predict.causal_survival_forest

*Predict with a causal survival forest forest*

---

## Description

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

## Usage

```
## S3 method for class 'causal_survival_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). Note that this matrix should have the number of columns as the training matrix, and that the columns must appear in the same order. |
| 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.

## Examples

```
# Train a standard causal survival forest.
n <- 3000
p <- 5
X <- matrix(runif(n * p), n, p)
W <- rbinom(n, 1, 0.5)
Y.max <- 1
failure.time <- pmin(rexp(n) * X[, 1] + W, Y.max)
censor.time <- 2 * runif(n)
Y <- pmin(failure.time, censor.time)
D <- as.integer(failure.time <= censor.time)
cs.forest <- causal_survival_forest(X, Y, W, D)

# Predict using the forest.
X.test <- matrix(0.5, 10, p)
X.test[, 1] <- seq(0, 1, length.out = 10)
cs.pred <- predict(cs.forest, X.test, estimate.variance = TRUE)

# Plot the estimated CATEs along with 95% confidence bands.
```

```
r.monte.carlo <- rexp(5000)
cate <- rep(NA, 10)
for (i in 1:10) {
  cate[i] <- mean(pmin(r.monte.carlo * X.test[i, 1] + 1, Y.max) -
                    pmin(r.monte.carlo * X.test[i, 1], Y.max))
}
plot(X.test[, 1], cate, type = 'l', col = 'red')
points(X.test[, 1], cs.pred$predictions)
lines(X.test[, 1], cs.pred$predictions + 2 * sqrt(cs.pred$variance.estimates), lty = 2)
lines(X.test[, 1], cs.pred$predictions - 2 * sqrt(cs.pred$variance.estimates), lty = 2)

# Compute a doubly robust estimate of the average treatment effect.
average_treatment_effect(cs.forest)

# Compute the best linear projection on the first covariate.
best_linear_projection(cs.forest, X[, 1])

# Train the forest on a less granular grid.
cs.forest.grid <- causal_survival_forest(X, Y, W, D,
                                  failure.times = seq(min(Y), max(Y), length.out = 50))
plot(X.test[, 1], cs.pred$predictions)
points(X.test[, 1], predict(cs.forest.grid, X.test)$predictions, col = "blue")
```

---

`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). Note that this matrix should have the number of columns as the training matrix, and that the columns must appear in the same order. |
| 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

```
# Train an instrumental forest.
n <- 2000
p <- 5
X <- matrix(rbinom(n * p, 1, 0.5), n, p)
Z <- rbinom(n, 1, 0.5)
Q <- rbinom(n, 1, 0.5)
W <- Q * Z
tau <-  X[, 1] / 2
Y <- rowSums(X[, 1:3]) + tau * W + Q + rnorm(n)
iv.forest <- instrumental_forest(X, Y, W, Z)

# Predict on out-of-bag training samples.
iv.pred <- predict(iv.forest)
```

---

# predict.ll_regression_forest

*Predict with a local linear forest*

---

## Description

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

## Usage

```
## S3 method for class 'll_regression_forest'
predict(
  object,
  newdata = NULL,
```

```
  linear.correction.variables = NULL,
  ll.lambda = NULL,
  ll.weight.penalty = FALSE,
  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). Note that this matrix should have the number of columns as the training matrix, and that the columns must appear in the same order. |
| linear.correction.variables | Optional subset of indexes for variables to be used in local linear prediction. If left NULL, all variables are used. We run a locally weighted linear regression on the included variables. Please note that this is a beta feature still in development, and may slow down prediction considerably. Defaults to NULL. |
| ll.lambda | Ridge penalty for local linear predictions. Defaults to NULL and will be cross-validated. |
| ll.weight.penalty | Option to standardize ridge penalty by covariance (TRUE), or penalize all covariates equally (FALSE). Defaults to FALSE. |
| 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

```
# Train the forest.
n <- 50
p <- 5
X <- matrix(rnorm(n * p), n, p)
Y <- X[, 1] * rnorm(n)
forest <- ll_regression_forest(X, Y)

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

```
predictions <- predict(forest, X.test)

# Predict on out-of-bag training samples.
predictions.oob <- predict(forest)
```

---

`predict.multi_arm_causal_forest` *Predict with a multi arm causal forest*

---

## Description

Gets estimates of contrasts tau_k(x) using a trained multi arm causal forest (k = 1,...,K-1 where K is the number of treatments).

## Usage

```
## S3 method for class 'multi_arm_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). Note that this matrix should have the number of columns as the training matrix, and that the columns must appear in the same order. |
| `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). This option is currently only supported for univariate outcomes Y. |
| `...` | Additional arguments (currently ignored). |

## Value

A list with elements ‘predictions‘: a 3d array of dimension [num.samples, K-1, M] with predictions for each contrast, for each outcome 1,..,M (singleton dimensions in this array can be dropped by passing the ‘drop‘ argument to ‘[‘, or with the shorthand ‘$predictions[,,]‘), and optionally ‘variance.estimates‘: a matrix with K-1 columns with variance estimates for each contrast.

## Examples

```
# Train a multi arm causal forest.
n <- 500
p <- 10
X <- matrix(rnorm(n * p), n, p)
W <- as.factor(sample(c("A", "B", "C"), n, replace = TRUE))
Y <- X[, 1] + X[, 2] * (W == "B") - 1.5 * X[, 2] * (W == "C") + rnorm(n)
mc.forest <- multi_arm_causal_forest(X, Y, W)

# Predict contrasts (out-of-bag) using the forest.
# By default, the first ordinal treatment is used as baseline ("A" in this example),
# giving two contrasts tau_B = Y(B) - Y(A), tau_C = Y(C) - Y(A)
mc.pred <- predict(mc.forest)
# Fitting several outcomes jointly is supported, and the returned prediction array has
# dimension [num.samples, num.contrasts, num.outcomes]. Since num.outcomes is one in
# this example, we can `drop()` this singleton dimension using the `[,,]` shorthand.
tau.hat <- mc.pred$predictions[,,]

plot(X[, 2], tau.hat[, "B - A"], ylab = "tau.contrast")
abline(0, 1, col = "red")
points(X[, 2], tau.hat[, "C - A"], col = "blue")
abline(0, -1.5, col = "red")
legend("topleft", c("B - A", "C - A"), col = c("black", "blue"), pch = 19)

# The average treatment effect of the arms with "A" as baseline.
average_treatment_effect(mc.forest)

# The conditional response surfaces mu_k(X) for a single outcome can be reconstructed from
# the contrasts tau_k(x), the treatment propensities e_k(x), and the conditional mean m(x).
# Given treatment "A" as baseline we have:
# m(x) := E[Y | X] = E[Y(A) | X] + E[W_B (Y(B) - Y(A))] + E[W_C (Y(C) - Y(A))]
# which given unconfoundedness is equal to:
# m(x) = mu(A, x) + e_B(x) tau_B(X) + e_C(x) tau_C(x)
# Rearranging and plugging in the above expressions, we obtain the following estimates
# * mu(A, x) = m(x) - e_B(x) tau_B(x) - e_C(x) tau_C(x)
# * mu(B, x) = m(x) + (1 - e_B(x)) tau_B(x) - e_C(x) tau_C(x)
# * mu(C, x) = m(x) - e_B(x) tau_B(x) + (1 - e_C(x)) tau_C(x)
Y.hat <- mc.forest$Y.hat
W.hat <- mc.forest$W.hat

muA <- Y.hat - W.hat[, "B"] * tau.hat[, "B - A"] - W.hat[, "C"] * tau.hat[, "C - A"]
muB <- Y.hat + (1 - W.hat[, "B"]) * tau.hat[, "B - A"] - W.hat[, "C"] * tau.hat[, "C - A"]
muC <- Y.hat - W.hat[, "B"] * tau.hat[, "B - A"] + (1 - W.hat[, "C"]) * tau.hat[, "C - A"]

# These can also be obtained with some array manipulations.
# (the first column is always the baseline arm)
Y.hat.baseline <- Y.hat - rowSums(W.hat[, -1, drop = FALSE] * tau.hat)
mu.hat.matrix <- cbind(Y.hat.baseline, c(Y.hat.baseline) + tau.hat)
colnames(mu.hat.matrix) <- levels(W)
head(mu.hat.matrix)
```

```
# The reference level for contrast prediction can be changed with `relevel`.
# Fit and predict with treatment B as baseline:
W <- relevel(W, ref = "B")
mc.forest.B <- multi_arm_causal_forest(X, Y, W)
```

---

`predict.multi_regression_forest` *Predict with a multi regression forest*

---

## Description

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

## Usage

```
## S3 method for class 'multi_regression_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). Note that this matrix should have the number of columns as the training matrix, and that the columns must appear in the same order. |
| `num.threads` | Number of threads used in training. If set to NULL, the software automatically selects an appropriate amount. |
| `...` | Additional arguments (currently ignored). |

## Value

A list containing ‘predictions‘: a matrix of predictions for each outcome.

## Examples

```
# Train a standard regression forest.
n <- 500
p <- 5
X <- matrix(rnorm(n * p), n, p)
Y <-  X[, 1, drop = FALSE] %*% cbind(1, 2) + rnorm(n)
mr.forest <- multi_regression_forest(X, Y)

# Predict using the forest.
X.test <- matrix(0, 101, p)
```

```
X.test[, 1] <- seq(-2, 2, length.out = 101)
mr.pred <- predict(mr.forest, X.test)

# Predict on out-of-bag training samples.
mr.pred <- predict(mr.forest)
```

---

`predict.probability_forest` *Predict with a probability forest*

---

## Description

Gets estimates of P[Y = k | X = x] using a trained forest.

## Usage

```
## S3 method for class 'probability_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). Note that this matrix should have the number of columns as the training matrix, and that the columns must appear in the same order. |
| `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 P[Y = k \| X] are desired (for confidence intervals). |
| `...` | Additional arguments (currently ignored). |

## Value

A list with attributes ‘predictions‘: a matrix of predictions for each class, and optionally the attribute ‘variance.estimates‘: a matrix of variance estimates for each class.

## Examples

```
# Train a probability forest.
p <- 5
n <- 2000
X <- matrix(rnorm(n*p), n, p)
prob <- 1 / (1 + exp(-X[, 1] - X[, 2]))
Y <- as.factor(rbinom(n, 1, prob))
p.forest <- probability_forest(X, Y)

# Predict using the forest.
X.test <- matrix(0, 10, p)
X.test[, 1] <- seq(-1.5, 1.5, length.out = 10)
p.hat <- predict(p.forest, X.test, estimate.variance = TRUE)

# Plot the estimated success probabilities with 95 % confidence bands.
prob.test <- 1 / (1 + exp(-X.test[, 1] - X.test[, 2]))
p.true <- cbind(`0` = 1 - prob.test, `1` = prob.test)
plot(X.test[, 1], p.true[, "1"], col = 'red', ylim = c(0, 1))
points(X.test[, 1], p.hat$predictions[, "1"], pch = 16)
lines(X.test[, 1], (p.hat$predictions + 2 * sqrt(p.hat$variance.estimates))[, "1"])
lines(X.test[, 1], (p.hat$predictions - 2 * sqrt(p.hat$variance.estimates))[, "1"])

# Predict on out-of-bag training samples.
p.hat <- predict(p.forest)
```

---

# 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 = 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). Note that this matrix should have the number of columns as the training matrix, and that the columns must appear in the same order. |

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

## Value

A list with elements 'predictions': a matrix with predictions at each test point for each desired quantile.

## Examples

```
# 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)
```

---

# 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,
  linear.correction.variables = NULL,
  ll.lambda = NULL,
  ll.weight.penalty = FALSE,
```

```
  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). Note that this matrix should have the number of columns as the training matrix, and that the columns must appear in the same order. |
| `linear.correction.variables` | Optional subset of indexes for variables to be used in local linear prediction. If NULL, standard GRF prediction is used. Otherwise, we run a locally weighted linear regression on the included variables. Please note that this is a beta feature still in development, and may slow down prediction considerably. Defaults to NULL. |
| `ll.lambda` | Ridge penalty for local linear predictions. Defaults to NULL and will be cross-validated. |
| `ll.weight.penalty` | Option to standardize ridge penalty by covariance (TRUE), or penalize all covariates equally (FALSE). Defaults to FALSE. |
| `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 estimates of the error and (optionally) its variance estimates. Column 'predictions' contains estimates of E[Y|X=x]. The square-root of column 'variance.estimates' is the standard error the test mean-squared error. Column 'excess.error' contains jackknife estimates of the Monte-carlo error. The sum of 'debiased.error' and 'excess.error' is the raw error attained by the current forest, and 'debiased.error' alone is an estimate of the error attained by a forest with an infinite number of trees. We recommend that users grow enough forests to make the 'excess.error' negligible.

## Examples

```
# 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)
```

---

`predict.survival_forest`

*Predict with a survival forest*

---

## Description

Gets estimates of the conditional survival function S(t, x) using a trained survival forest. The curve can be estimated by Kaplan-Meier, or Nelson-Aalen.

## Usage

```
## S3 method for class 'survival_forest'
predict(
  object,
  newdata = NULL,
  failure.times = NULL,
  prediction.type = c("Kaplan-Meier", "Nelson-Aalen"),
  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). Note that this matrix should have the number of columns as the training matrix, and that the columns must appear in the same order. |
| `failure.times` | A vector of failure times to make predictions at. If NULL, then the failure times used for training the forest is used. The time points should be in increasing order. Default is NULL. |

prediction.type
: The type of estimate of the survival function, choices are "Kaplan-Meier" or "Nelson-Aalen". The default is the prediction.type used to train the forest.

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

...
: Additional arguments (currently ignored).

## Value

A list with elements ‘failure.times‘: a vector of event times t for the survival curve, and ‘predictions‘: a matrix of survival curves. Each row is the survival curve for sample X_i: predictions[i, j] = S(failure.times[j], X_i).

## Examples

```
# Train a standard survival forest.
n <- 2000
p <- 5
X <- matrix(rnorm(n * p), n, p)
failure.time <- exp(0.5 * X[, 1]) * rexp(n)
censor.time <- 2 * rexp(n)
Y <- pmin(failure.time, censor.time)
D <- as.integer(failure.time <= censor.time)
s.forest <- survival_forest(X, Y, D)

# Predict using the forest.
X.test <- matrix(0, 3, p)
X.test[, 1] <- seq(-2, 2, length.out = 3)
s.pred <- predict(s.forest, X.test)

# Plot the survival curve.
plot(NA, NA, xlab = "failure time", ylab = "survival function",
     xlim = range(s.pred$failure.times),
     ylim = c(0, 1))
for(i in 1:3) {
  lines(s.pred$failure.times, s.pred$predictions[i,], col = i)
  s.true = exp(-s.pred$failure.times / exp(0.5 * X.test[i, 1]))
  lines(s.pred$failure.times, s.true, col = i, lty = 2)
}

# Predict on out-of-bag training samples.
s.pred <- predict(s.forest)

# Plot the survival curve for the first five individuals.
matplot(s.pred$failure.times, t(s.pred$predictions[1:5, ]),
        xlab = "failure time", ylab = "survival function (OOB)",
        type = "l", lty = 1)

# Train the forest on a less granular grid.
failure.summary <- summary(Y[D == 1])
events <- seq(failure.summary["Min."], failure.summary["Max."], by = 0.1)
```

```
s.forest.grid <- survival_forest(X, Y, D, failure.times = events)
s.pred.grid <- predict(s.forest.grid)
matpoints(s.pred.grid$failure.times, t(s.pred.grid$predictions[1:5, ]),
          type = "l", lty = 2)

# Compute OOB concordance based on the mortality score in Ishwaran et al. (2008).
s.pred.nelson.aalen <- predict(s.forest, prediction.type = "Nelson-Aalen")
chf.score <- rowSums(-log(s.pred.nelson.aalen$predictions))
if (require("survival", quietly = TRUE)) {
 concordance(Surv(Y, D) ~ chf.score, reverse = TRUE)
}
```

---

## print.boosted_regression_forest *Print a boosted regression forest*

---

### Description

Print a boosted regression forest

### Usage

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

### Arguments

| | |
|---|---|
| x | The boosted forest to print. |
| ... | Additional arguments (currently ignored). |

---

## 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). |

---

## print.tuning_output *Print tuning output. Displays average error for q-quantiles of tuned parameters.*

---

### Description

Print tuning output. Displays average error for q-quantiles of tuned parameters.

### Usage

```
## S3 method for class 'tuning_output'
print(x, tuning.quantiles = seq(0, 1, 0.2), ...)
```

### Arguments

| | |
|---|---|
| x | The tuning output to print. |
| tuning.quantiles | vector of quantiles to display average error over. Default: seq(0, 1, 0.2) (quintiles) |
| ... | Additional arguments (currently ignored). |

probability_forest *Probability forest*

## Description

Trains a probability forest that can be used to estimate the conditional class probabilities P[Y = k | X = x]

## Usage

```
probability_forest(
  X,
  Y,
  num.trees = 2000,
  sample.weights = NULL,
  clusters = NULL,
  equalize.cluster.weights = FALSE,
  sample.fraction = 0.5,
  mtry = min(ceiling(sqrt(ncol(X)) + 20), ncol(X)),
  min.node.size = 5,
  honesty = TRUE,
  honesty.fraction = 0.5,
  honesty.prune.leaves = TRUE,
  alpha = 0.05,
  imbalance.penalty = 0,
  ci.group.size = 2,
  compute.oob.predictions = TRUE,
  num.threads = NULL,
  seed = runif(1, 0, .Machine$integer.max)
)
```

## Arguments

| | |
|---|---|
| X | The covariates. |
| Y | The class label (must be a factor vector with no NAs). |
| num.trees | Number of trees grown in the forest. Note: Getting accurate confidence intervals generally requires more trees than getting accurate predictions. Default is 2000. |
| sample.weights | Weights given to an observation in estimation. If NULL, each observation is given the same weight. Default is NULL. |
| clusters | Vector of integers or factors specifying which cluster each observation corresponds to. Default is NULL (ignored). |
| equalize.cluster.weights | If FALSE, each unit is given the same weight (so that bigger clusters get more weight). If TRUE, each cluster is given equal weight in the forest. In this case, during training, each tree uses the same number of observations from each drawn |

cluster: If the smallest cluster has K units, then when we sample a cluster during training, we only give a random K elements of the cluster to the tree-growing procedure. When estimating average treatment effects, each observation is given weight 1/cluster size, so that the total weight of each cluster is the same.

`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 variables.

`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 algorithm reference.

`honesty.fraction`
: The fraction of data that will be used for determining splits if honesty = TRUE. Corresponds to set J1 in the notation of the paper. Default is 0.5 (i.e. half of the data is used for determining splits).

`honesty.prune.leaves`
: If TRUE, prunes the estimation sample tree such that no leaves are empty. If FALSE, keep 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.

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

`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. Default is 2.

`compute.oob.predictions`
: Whether OOB predictions on training set should be precomputed. Default is TRUE.

`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.

## Value

A trained probability forest object.

## Examples

```
# Train a probability forest.
p <- 5
n <- 2000
X <- matrix(rnorm(n*p), n, p)
prob <- 1 / (1 + exp(-X[, 1] - X[, 2]))
Y <- as.factor(rbinom(n, 1, prob))
p.forest <- probability_forest(X, Y)

# Predict using the forest.
X.test <- matrix(0, 10, p)
X.test[, 1] <- seq(-1.5, 1.5, length.out = 10)
p.hat <- predict(p.forest, X.test, estimate.variance = TRUE)

# Plot the estimated success probabilities with 95 % confidence bands.
prob.test <- 1 / (1 + exp(-X.test[, 1] - X.test[, 2]))
p.true <- cbind(`0` = 1 - prob.test, `1` = prob.test)
plot(X.test[, 1], p.true[, "1"], col = 'red', ylim = c(0, 1))
points(X.test[, 1], p.hat$predictions[, "1"], pch = 16)
lines(X.test[, 1], (p.hat$predictions + 2 * sqrt(p.hat$variance.estimates))[, "1"])
lines(X.test[, 1], (p.hat$predictions - 2 * sqrt(p.hat$variance.estimates))[, "1"])

# Predict on out-of-bag training samples.
p.hat <- predict(p.forest)
```

---

# 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,
  num.trees = 2000,
  quantiles = c(0.1, 0.5, 0.9),
  regression.splitting = FALSE,
  clusters = NULL,
  equalize.cluster.weights = FALSE,
  sample.fraction = 0.5,
  mtry = min(ceiling(sqrt(ncol(X)) + 20), ncol(X)),
  min.node.size = 5,
```

```
  honesty = TRUE,
  honesty.fraction = 0.5,
  honesty.prune.leaves = TRUE,
  alpha = 0.05,
  imbalance.penalty = 0,
  compute.oob.predictions = FALSE,
  num.threads = NULL,
  seed = runif(1, 0, .Machine$integer.max)
)
```

## Arguments

| | |
|---|---|
| X | The covariates used in the quantile regression. |
| Y | The outcome. |
| num.trees | Number of trees grown in the forest. Note: Getting accurate confidence intervals generally requires more trees than getting accurate predictions. Default is 2000. |
| quantiles | Vector of quantiles used to calibrate the forest. Default is (0.1, 0.5, 0.9). |
| 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). Default is FALSE. |
| clusters | Vector of integers or factors specifying which cluster each observation corresponds to. Default is NULL (ignored). |
| equalize.cluster.weights | If FALSE, each unit is given the same weight (so that bigger clusters get more weight). If TRUE, each cluster is given equal weight in the forest. In this case, during training, each tree uses the same number of observations from each drawn cluster: If the smallest cluster has K units, then when we sample a cluster during training, we only give a random K elements of the cluster to the tree-growing procedure. When estimating average treatment effects, each observation is given weight 1/cluster size, so that the total weight of each cluster is the same. |
| 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 variables. |
| 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 algorithm reference. |
| honesty.fraction | The fraction of data that will be used for determining splits if honesty = TRUE. Corresponds to set J1 in the notation of the paper. Default is 0.5 (i.e. half of the data is used for determining splits). |

honesty.prune.leaves
: If TRUE, prunes the estimation sample tree such that no leaves are empty. If FALSE, keep 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.

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. Default is FALSE.

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.

## Value

A trained quantile forest object.

## Examples

```
# 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))
```

`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,
  num.trees = 2000,
  sample.weights = NULL,
  clusters = NULL,
  equalize.cluster.weights = FALSE,
  sample.fraction = 0.5,
  mtry = min(ceiling(sqrt(ncol(X)) + 20), ncol(X)),
  min.node.size = 5,
  honesty = TRUE,
  honesty.fraction = 0.5,
  honesty.prune.leaves = TRUE,
  alpha = 0.05,
  imbalance.penalty = 0,
  ci.group.size = 2,
  tune.parameters = "none",
  tune.num.trees = 50,
  tune.num.reps = 100,
  tune.num.draws = 1000,
  compute.oob.predictions = TRUE,
  num.threads = NULL,
  seed = runif(1, 0, .Machine$integer.max)
)
```

## Arguments

| | |
|---|---|
| `X` | The covariates used in the regression. |
| `Y` | The outcome. |
| `num.trees` | Number of trees grown in the forest. Note: Getting accurate confidence intervals generally requires more trees than getting accurate predictions. Default is 2000. |
| `sample.weights` | Weights given to an observation in estimation. If NULL, each observation is given the same weight. Default is NULL. |
| `clusters` | Vector of integers or factors specifying which cluster each observation corresponds to. Default is NULL (ignored). |

`equalize.cluster.weights`
: If FALSE, each unit is given the same weight (so that bigger clusters get more weight). If TRUE, each cluster is given equal weight in the forest. In this case, during training, each tree uses the same number of observations from each drawn cluster: If the smallest cluster has K units, then when we sample a cluster during training, we only give a random K elements of the cluster to the tree-growing procedure. When estimating average treatment effects, each observation is given weight 1/cluster size, so that the total weight of each cluster is the same. Note that, if this argument is FALSE, sample weights may also be directly adjusted via the sample.weights argument. If this argument is TRUE, sample.weights must be set to NULL. Default is FALSE.

`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 variables.

`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 algorithm reference.

`honesty.fraction`
: The fraction of data that will be used for determining splits if honesty = TRUE. Corresponds to set J1 in the notation of the paper. Default is 0.5 (i.e. half of the data is used for determining splits).

`honesty.prune.leaves`
: If TRUE, prunes the estimation sample tree such that no leaves are empty. If FALSE, keep 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.

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

`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. Default is 2.

`tune.parameters`
: A vector of parameter names to tune. If "all": all tunable parameters are tuned by cross-validation. The following parameters are tunable: ("sample.fraction", "mtry", "min.node.size", "honesty.fraction", "honesty.prune.leaves", "alpha", "imbalance.penalty"). If honesty is FALSE the honesty.* parameters are not tuned. Default is "none" (no parameters are tuned).

tune.num.trees The number of trees in each 'mini forest' used to fit the tuning model. Default is 50.

tune.num.reps The number of forests used to fit the tuning model. Default is 100.

tune.num.draws The number of random parameter values considered when using the model to select the optimal parameters. Default is 1000.

compute.oob.predictions
Whether OOB predictions on training set should be precomputed. Default is TRUE.

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.

## Value

A trained regression forest object. If tune.parameters is enabled, then tuning information will be included through the 'tuning.output' attribute.

## Examples

```
# Train a standard regression forest.
n <- 500
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)
```

---

# 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

```
# Train a quantile forest.
n <- 250
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)
```

---

# survival_forest *Survival forest*

---

## Description

Trains a forest for right-censored surival data that can be used to estimate the conditional survival function S(t, x) = P[Y > t | X = x]

## Usage

```
survival_forest(
  X,
  Y,
  D,
  failure.times = NULL,
  num.trees = 1000,
  sample.weights = NULL,
  clusters = NULL,
  equalize.cluster.weights = FALSE,
  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,
  prediction.type = c("Kaplan-Meier", "Nelson-Aalen"),
  compute.oob.predictions = TRUE,
  num.threads = NULL,
  seed = runif(1, 0, .Machine$integer.max)
)
```

## Arguments

| | |
|---|---|
| X | The covariates. |
| Y | The event time (may be negative). |
| D | The event type (0: censored, 1: failure). |
| failure.times | A vector of event times to fit the survival curve at. If NULL, then all the observed failure times are used. This speeds up forest estimation by constraining the event grid. Observed event times are rounded down to the last sorted occurance less than or equal to the specified failure time. The time points should be in increasing order. Default is NULL. |
| num.trees | Number of trees grown in the forest. Default is 1000. |
| sample.weights | Weights given to an observation in prediction. If NULL, each observation is given the same weight. Default is NULL. |
| clusters | Vector of integers or factors specifying which cluster each observation corresponds to. Default is NULL (ignored). |
| equalize.cluster.weights | If FALSE, each unit is given the same weight (so that bigger clusters get more weight). If TRUE, each cluster is given equal weight in the forest. In this case, during training, each tree uses the same number of observations from each drawn cluster: If the smallest cluster has K units, then when we sample a cluster during training, we only give a random K elements of the cluster to the tree-growing procedure. When estimating average treatment effects, each observation is given weight 1/cluster size, so that the total weight of each cluster is the same. Note that, if this argument is FALSE, sample weights may also be directly adjusted via the sample.weights argument. If this argument is TRUE, sample.weights must be set to NULL. Default is FALSE. |
| 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 variables. |
| 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 15. |

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 algorithm reference.

honesty.fraction
The fraction of data that will be used for determining splits if honesty = TRUE. Corresponds to set J1 in the notation of the paper. Default is 0.5 (i.e. half of the data is used for determining splits).

honesty.prune.leaves
If TRUE, prunes the estimation sample tree such that no leaves are empty. If FALSE, keep 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. The number of failures in each child has to be at least one or ‘alpha‘ times the number of samples in the parent node. Default is 0.05.

prediction.type
The type of estimate of the survival function, choices are "Kaplan-Meier" or "Nelson-Aalen". Only relevant if ‘compute.oob.predictions‘ is TRUE. Default is "Kaplan-Meier".

compute.oob.predictions
Whether OOB predictions on training set should be precomputed. Default is TRUE.

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.

## Value

A trained survival_forest forest object.

## References

Cui, Yifan, Michael R. Kosorok, Erik Sverdrup, Stefan Wager, and Ruoqing Zhu. "Estimating Heterogeneous Treatment Effects with Right-Censored Data via Causal Survival Forests." arXiv preprint arXiv:2001.09887, 2020.

Ishwaran, Hemant, Udaya B. Kogalur, Eugene H. Blackstone, and Michael S. Lauer. "Random survival forests." The Annals of Applied Statistics 2.3 (2008): 841-860.

## Examples

```
# Train a standard survival forest.
n <- 2000
p <- 5
X <- matrix(rnorm(n * p), n, p)
failure.time <- exp(0.5 * X[, 1]) * rexp(n)
```

```
censor.time <- 2 * rexp(n)
Y <- pmin(failure.time, censor.time)
D <- as.integer(failure.time <= censor.time)
s.forest <- survival_forest(X, Y, D)

# Predict using the forest.
X.test <- matrix(0, 3, p)
X.test[, 1] <- seq(-2, 2, length.out = 3)
s.pred <- predict(s.forest, X.test)

# Plot the survival curve.
plot(NA, NA, xlab = "failure time", ylab = "survival function",
     xlim = range(s.pred$failure.times),
     ylim = c(0, 1))
for(i in 1:3) {
  lines(s.pred$failure.times, s.pred$predictions[i,], col = i)
  s.true = exp(-s.pred$failure.times / exp(0.5 * X.test[i, 1]))
  lines(s.pred$failure.times, s.true, col = i, lty = 2)
}

# Predict on out-of-bag training samples.
s.pred <- predict(s.forest)

# Plot the survival curve for the first five individuals.
matplot(s.pred$failure.times, t(s.pred$predictions[1:5, ]),
        xlab = "failure time", ylab = "survival function (OOB)",
        type = "l", lty = 1)

# Train the forest on a less granular grid.
failure.summary <- summary(Y[D == 1])
events <- seq(failure.summary["Min."], failure.summary["Max."], by = 0.1)
s.forest.grid <- survival_forest(X, Y, D, failure.times = events)
s.pred.grid <- predict(s.forest.grid)
matpoints(s.pred.grid$failure.times, t(s.pred.grid$predictions[1:5, ]),
          type = "l", lty = 2)

# Compute OOB concordance based on the mortality score in Ishwaran et al. (2008).
s.pred.nelson.aalen <- predict(s.forest, prediction.type = "Nelson-Aalen")
chf.score <- rowSums(-log(s.pred.nelson.aalen$predictions))
if (require("survival", quietly = TRUE)) {
 concordance(Surv(Y, D) ~ chf.score, reverse = TRUE)
}
```

---

`test_calibration` *Omnibus evaluation of the quality of the random forest estimates via calibration.*

---

## Description

Test calibration of the forest. Computes the best linear fit of the target estimand using the forest prediction (on held-out data) as well as the mean forest prediction as the sole two regressors. A coefficient of 1 for 'mean.forest.prediction' suggests that the mean forest prediction is correct, whereas a coefficient of 1 for 'differential.forest.prediction' additionally suggests that the forest has captured heterogeneity in the underlying signal. The p-value of the 'differential.forest.prediction' coefficient also acts as an omnibus test for the presence of heterogeneity: If the coefficient is significantly greater than 0, then we can reject the null of no heterogeneity.

## Usage

```
test_calibration(forest, vcov.type = "HC3")
```

## Arguments

| | |
|---|---|
| forest | The trained forest. |
| vcov.type | Optional covariance type for standard errors. The possible options are HC0, ..., HC3. The default is "HC3", which is recommended in small samples and corresponds to the "shortcut formula" for the jackknife (see MacKinnon & White for more discussion, and Cameron & Miller for a review). For large data sets with clusters, "HC0" or "HC1" are significantly faster to compute. |

## Value

A heteroskedasticity-consistent test of calibration.

## References

Cameron, A. Colin, and Douglas L. Miller. "A practitioner's guide to cluster-robust inference." Journal of human resources 50, no. 2 (2015): 317-372.

Chernozhukov, Victor, Mert Demirer, Esther Duflo, and Ivan Fernandez-Val. "Generic Machine Learning Inference on Heterogenous Treatment Effects in Randomized Experiments." arXiv preprint arXiv:1712.04802 (2017).

MacKinnon, James G., and Halbert White. "Some heteroskedasticity-consistent covariance matrix estimators with improved finite sample properties." Journal of Econometrics 29.3 (1985): 305-325.

## Examples

```
n <- 800
p <- 5
X <- matrix(rnorm(n * p), n, p)
W <- rbinom(n, 1, 0.25 + 0.5 * (X[, 1] > 0))
Y <- pmax(X[, 1], 0) * W + X[, 2] + pmin(X[, 3], 0) + rnorm(n)
forest <- causal_forest(X, Y, W)
test_calibration(forest)
```

## tune_causal_forest *Causal forest tuning (removed)*

### Description

To tune a causal forest, see the function ‘causal_forest‘

### Usage

```
tune_causal_forest(X, Y, W, Y.hat, W.hat, ...)
```

### Arguments

| | |
|---|---|
| X | X |
| Y | Y |
| W | W |
| Y.hat | Y.hat |
| W.hat | W.hat |
| ... | Additional arguments (currently ignored). |

### Value

output

## tune_instrumental_forest *Instrumental forest tuning (removed)*

### Description

To tune a instrumental forest, see the function ‘instrumental_forest‘

### Usage

```
tune_instrumental_forest(X, Y, W, Z, Y.hat, W.hat, Z.hat, ...)
```

### Arguments

| | |
|---|---|
| X | X |
| Y | Y |
| W | W |
| Z | Z |
| Y.hat | Y.hat |
| W.hat | W.hat |
| Z.hat | Z.hat |
| ... | Additional arguments (currently ignored). |

### Value

output

---

`tune_regression_forest` *Regression forest tuning (removed)*

---

### Description

To tune a regression forest, see the function ‘regression_forest‘

### Usage

```
tune_regression_forest(X, Y, ...)
```

### Arguments

| | |
|---|---|
| `X` | X |
| `Y` | Y |
| `...` | Additional arguments (currently ignored). |

### Value

output

---

`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

```
# Train a quantile forest.
n <- 250
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)
```

# Index