Package ‘grf’

March 12, 2020

Title  Generalized Random Forests
Version  1.1.0
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, and treatment effect estimation (optionally using instrumental variables).
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  7.0.2
Suggests  DiagrammeR, testthat
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 [ctb],
       Stefan Wager [aut],
       Marvin Wright [ctb]
Maintainer  Julie Tibshirani <jtibs@cs.stanford.edu>
Repository  CRAN
Date/Publication  2020-03-12 06:30:28 UTC
R topics documented:

- average_late
- average_partial_effect
- average_treatment_effect
- best_linear_projection
- boosted_regression_forest
- causal_forest
- custom_forest
- get_sample_weights
- get_tree
- grf
- instrumental_forest
- leaf_stats.causal_forest
- leaf_stats.default
- leaf_stats.instrumental_forest
- leaf_stats.regression_forest
- ll_regression_forest
- merge_forests
- plot.grf_tree
- predict.boosted_regression_forest
- predict.causal_forest
- predict.custom_forest
- predict.instrumental_forest
- predict.ll_regression_forest
- predict.quantile_forest
- predict.regression_forest
- print.boosted_regression_forest
- print.grf
- print.grf_tree
- print.tuning_output
- quantile_forest
- regression_forest
- split_frequencies
- test_calibration
- tune_causal_forest
- tune_forest
- tune_instrumental_forest
- tune_ll_causal_forest
- tune_ll_regression_forest
- tune_regression_forest
- variable_importance
Estimate the average (conditional) local average treatment effect using a causal forest.

Description

Given an outcome Y, treatment W and instrument Z, the (conditional) local average treatment effect is \( \tau(x) = \frac{\text{Cov}[Y, Z | X = x]}{\text{Cov}[W, Z | X = x]} \). This is the quantity that is estimated with an instrumental forest. It can be interpreted 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 is about estimating \( \tau = E[\tau(X)] \) which, extending standard nomenclature, should perhaps be called the Average (Conditional) Local Average Treatment Effect (ACLATE).

Usage

```r
average_late(forest, compliance.score = NULL, subset = NULL)
```

Arguments

- `forest`: The trained forest.
- `compliance.score`: An estimate of the causal effect of Z on W, i.e., \( \Delta(X) = E[W | X, Z = 1] - E[W | X, Z = 0] \), for each sample \( i = 1, \ldots, n \).
- `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 \( X_i \), not using the instrument \( Z_i \), treatment \( W_i \) or outcome \( Y_i \).

Details

We estimate the ACLATE using a doubly robust estimator. See Chernozhukov et al. (2016) for a discussion, and Section 5.2 of Athey and Wager (2017) for an example using forests.

If clusters are specified for the forest, then each cluster gets equal weight. 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 0.5 (not 0.05).

Value

An estimate of the average (C)LATE, along with standard error.

References


---

average_partial_effect

*Estimate average partial effects using a causal forest*

---

**Description**

Gets estimates of the average partial effect, in particular the (conditional) average treatment effect (target.sample = all): 1/n \( \sum_{i=1}^{n} \text{Cov}[W_i, Y_i|X = X_i] / \text{Var}[W_i|X = X_i] \). Note that for a binary unconfounded treatment, the average partial effect matches the average treatment effect.

**Usage**

```r
average_partial_effect(
  forest, 
  calibrate.weights = TRUE, 
  subset = NULL, 
  num.trees.for.variance = 500
)
```

**Arguments**

- `forest` The trained forest.
- `calibrate.weights` Whether to force debiasing weights to match expected moments for 1, \( W \), \( W \hat{=} \), and \( 1/\text{Var}[W|X] \).
- `subset` Specifies a subset of the training examples over which we estimate the ATE. WARNING: For valid statistical performance, the subset should be defined only using features \( X_i \), not using the treatment \( W_i \) or the outcome \( Y_i \).
- `num.trees.for.variance` Number of trees used to estimate \( \text{Var}[W_i|X = x] \). Default is 500.

**Details**

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 partial effect, along with standard error.
average_treatment_effect

Examples

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

## End(Not run)
```

average_treatment_effect

*Estimate average treatment effects using a causal forest*

Description

Gets estimates of one of the following.

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

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

Usage

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

- **forest**: The trained forest.
- **target.sample**: Which sample to aggregate treatment effects over.
- **method**: Method used for doubly robust inference. Can be either augmented inverse-propensity weighting (AIPW), or targeted maximum likelihood estimation (TMLE).
- **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 \(X_i\), not using the treatment \(W_i\) or the outcome \(Y_i\).

Details

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.

Examples

```r
## Not run:
# Train a causal forest.
n <- 50
p <- 10
X <- matrix(rnorm(n * p), n, p)
W <- rbinom(n, 1, 0.5)
Y <- pmax(X[, 1], 0) * W + X[, 2] + pmin(X[, 3], 0) + rnorm(n)
c.forest <- causal_forest(X, Y, W)
# Predict using the forest.
X.test <- matrix(0, 101, p)
X.test[, 1] <- seq(-2, 2, length.out = 101)
c.pred <- predict(c.forest, X.test)
# Estimate the conditional average treatment effect on the full sample (CATE).
average_treatment_effect(c.forest, target.sample = "all")
# Estimate the conditional average treatment effect on the treated sample (CATT).
# We don't expect much difference between the CATE and the CATT in this example,
# since treatment assignment was randomized.
average_treatment_effect(c.forest, target.sample = "treated")
# Estimate the conditional average treatment effect on samples with positive X[,1].
average_treatment_effect(c.forest, target.sample = "all", X[, 1] > 0)
## End(Not run)
```
best_linear_projection

Estimate the best linear projection of a conditional average treatment effect using a causal forest.

Description

Let \( \tau(X_i) = E[Y(1) - Y(0) \mid X = X_i] \) be the CATE, and \( A_i \) be a vector of user-provided covariates. This function provides a (doubly robust) fit to the linear model

Usage

```r
best_linear_projection(
  forest,
  A = NULL,
  subset = NULL,
  debiasing.weights = NULL,
  num.trees.for.variance = 500
)
```

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 \( X_i \), not using the treatment \( W_i \) or the outcome \( Y_i \).
- **debiasing.weights**: A vector of length \( n \) of debiasing weights. If NULL (default) and the treatment is binary, then inverse-propensity weighting is used, otherwise, if the treatment is continuous, these are estimated by a variance forest.
- **num.trees.for.variance**: Number of trees used to estimate \( \text{Var}[W_i \mid X_i = x] \). Default is 500. (only applies with continuous treatment and debiasing.weights = NULL)

Details

\( \tau(X_i) \sim \beta_0 + A_i \cdot \beta \)

Procedurally, we do so by regressing doubly robust scores derived from the causal forest against the \( A_i \). Note the covariates \( A_i \) may consist of a subset of the \( X_i \), or they may be distinct. The case of the null model \( \tau(X_i) \sim \beta_0 \) 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 \( \text{Var}[W_i \mid X_i = x] \). 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


Examples

```r
## Not run:
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])
## End(Not run)
```

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

```r
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,
cl.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.
numtry     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 random-Forest 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.
causal_forest

Examples

## Not run:
# Train a boosted regression forest.
\n\n```
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))

## End(Not run)

---

causal_forest  Causal forest

Description

Trains a causal forest that can be used to estimate conditional average treatment effects \( \tau(X) \). When the treatment 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 \( \text{Cov}[Y, W | X = x] / \text{Var}[W | X = x] \), 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,
orthog.boosting = FALSE,
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 $E[Y | X_i]$, 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 $E[W | X_i]$. 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**
(Experimental) 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 random-Forest 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 $J_1$ 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.

orthog.boosting

(experimental) If TRUE, then when Y.hat = NULL or W.hat is NULL, the missing quantities are estimated using boosted regression forests. The number of boosting steps is selected automatically. 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 causal forest object. If tune.parameters is enabled, then tuning information will be included through the ‘tuning.output’ attribute.

Examples

```r
## Not run:
# 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

## End(Not run)

---

custom_forest  

**Custom forest**

**Description**

Trains a custom forest model.

**Usage**

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

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.

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.

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 random-
Forest 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.

clusters
Vector of integers or factors specifying which cluster each observation corre-
sponds 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.

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

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

#### Description

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

#### Usage

```r
get_sample_weights(forest, newdata = NULL, num.threads = NULL)
```

#### Arguments

- **forest**: The trained forest.
- **newdata**: Points at which predictions should be made. If NULL, makes out-of-bag predictions on the training set instead (i.e., provides predictions at \( X_i \) 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.
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

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

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

## End(Not run)

---

**grf**  

**GRF**

### Description

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

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

Some helpful links for getting started:


* For community questions and answers around usage, see Github issues labelled 'question' (https://github.com/grf-labs/grf/issues?q=label%3Aquestion).

### Examples

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

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

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

# Train a causal forest.
W = rbinom(n, 1, 0.4 + 0.2 * (X[,1] > 0))
Y = pmax(X[,1], 0) * W + X[,2] + pmin(X[,3], 0) + rnorm(n)
tau.forest = causal_forest(X, Y, W)

# Estimate treatment effects for the training data using out-of-bag prediction.
tau.hat.oob = predict(tau.forest)
hist(tau.hat.oob$predictions)

# Estimate treatment effects for the test sample.
tau.hat = predict(tau.forest, X.test)
plot(X.test[,1], tau.hat$predictions, ylim = range(tau.hat$predictions, 0, 2),
  xlab = "x", ylab = "tau", type = "l")
lines(X.test[,1], pmax(0, X.test[,1]), col = 2, lty = 2)

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

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

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

# 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).

# Generate new data.
n = 4000; 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")
```r
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")

# Check whether causal forest predictions are well calibrated.
test_calibration(tau.forest)

## End(Not run)
```

---

**instrumental_forest**  
**Intrumental forest**

---

**Description**

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

**Usage**

```r
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 $E[Y \mid X_i]$, 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 $E[W \mid X_i]$. If $W.hat = NULL$, these are estimated using a separate regression forest. Default is NULL.

**Z.hat**
Estimates of the instrument propensities $E[Z \mid X_i]$. 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**
(experimental) 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 random-forest 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 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.

**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.

Description

`leaf_stats.causal_forest`

*Calculate summary stats given a set of samples for causal forests.*

Usage

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

Arguments

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

Value

A named vector containing summary stats
A default leaf_stats for forests classes without a leaf_stats method that always returns NULL.

## Default S3 method:

```r
leaf_stats(forest, samples, ...)
```

### Arguments

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

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

## S3 method for class 'instrumental_forest'

```r
leaf_stats(forest, samples, ...)
```

### Arguments

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

### Value

A named vector containing summary stats
leaf_stats.regression_forest

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

Description

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

Usage

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

Arguments

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

Value

A named vector containing summary stats

ll_regression_forest  Local Linear forest

Description

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

Usage

```r
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,
  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,
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.
sample.weights ( experimental) 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 \( 1/\text{cluster size} \), 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 random-Forest 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 \( J_1 \) 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 does not take local linear splits into account.

**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.
merge_forests

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

## Not run:
# 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)

## End(Not run)

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

```r
## Not run:
# 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))

## End(Not run)
```

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

```r
## 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

```r
## Not run:
# Save the plot of a tree in the causal forest.
install.packages("DiagrammeR")
install.packages("DiagrammeRsvg")
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)
```
predict.boosted_regression_forest

Predict with a boosted regression forest.

Description

Gets estimates of \( \mathbb{E}[Y | X=x] \) using a trained regression forest.

Usage

```r
## 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 \( X_i \) 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

```r
## Not run:
# 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)

## End(Not run)
```

predict.causal_forest  

_Predict with a causal forest_

**Description**

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

**Usage**

```r
## 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 $X_i$ 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 co-
   variates 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

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

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

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

# Predict with confidence intervals; growing more trees is now recommended.
```

predict.custom_forest

Predict with a custom forest.

Usage

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

Arguments

object The trained forest.
newdata Points at which predictions should be made. If NULL, makes out-of-bag predictions on the training set instead (i.e., provides predictions at Xi using only trees that did not use the i-th training example). 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.
...

Value

Vector of predictions.

Examples

## Not run:
# Train a custom forest.
N <- 50
P <- 10
X <- matrix(rnorm(N * P), N, P)
Y <- X[, 1] * rnorm(N)
c.forest <- custom_forest(X, Y)

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

## End(Not run)

---

### predict.instrumental_forest

*Predict with an instrumental forest*

---

**Description**

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

**Usage**

```r
## 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.
predict.ll_regression_forest

Predict with a local linear forest

Description

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

Usage

```r
## 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 $X_i$ 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
- **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 $h_{x}(x)$ are desired (for confidence intervals).
- **...**: Additional arguments (currently ignored).
predict.quantile_forest

Value

A vector of predictions.

Examples

```r
## Not run:
# Train the forest.
set.seed(123)
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)

## End(Not run)
```

predict.quantile_forest

**Predict with a quantile forest**

Description

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

Usage

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

Arguments

- **object**: The trained forest.
newdata Points at which predictions should be made. If NULL, makes out-of-bag predictions on the training set instead (i.e., provides predictions at \( X_i \) 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.

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

... Additional arguments (currently ignored).

Value

Predictions at each test point for each desired quantile.

Examples

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

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

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

## End(Not run)
```

predict.regression_forest

*Predict with a regression forest*

Description

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

## S3 method for class 'regression_forest'
predict(
  object,
  newdata = NULL,
  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
- **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.
print.boosted_regression_forest

Print a boosted regression forest

## Examples

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

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

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

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

## End(Not run)
```

## Description

Print a boosted regression forest

## Usage

```r
## 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.

### Usage

```r
## 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.

### Usage

```r
## 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) (quantiles)
- ...  Additional arguments (currently ignored).

quantile_forest  Quantile forest

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

Usage
quantile_forest(
  X,
  Y,
  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,
quantile_forest

```r
alpha = 0.05,
imbalance.penalty = 0,
um.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.splits**: 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 $J_1$ 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.

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

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

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

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

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

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

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

## End(Not run)
regression_forest

Description

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

Usage

```r
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**: (experimental) 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).
regression_forest

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 random-Forest 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 \(J_1\) 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).
split_frequencies

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

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

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

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

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

The seed of the C++ random number generator.

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

Examples

```r
## Not run:
# Train a standard regression forest.
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)
```

Calculate which features the forest split on at each depth.

Calculate which features the forest split on at each depth.
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

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

# Calculate the split frequencies for this forest.
split_frequencies(q.forest)
## End(Not run)
```

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)`
Arguments

forest The trained forest.

Value

A heteroskedasticity-consistent test of calibration.

References


Examples

```r
## Not run:
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)
## End(Not run)
```
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 = "all",
tune.num.trees = 200,
tune.num.reps = 50,
tune.num.draws = 1000,
um.num.threads = NULL,
seed = runif(1, 0, .Machine$integer.max)
)

Arguments

X  The covariates used in the regression.

Y  The outcome.

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

Y.hat  Estimates of the expected responses E[Y | Xi], marginalizing over treatment. See section 6.1.1 of the GRF paper for further discussion of this quantity.

W.hat  Estimates of the treatment propensities E[W | Xi].

sample.weights (experimental) 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 random-Forest 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 "all".

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.

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

Examples

## Not run:
# Find the optimal tuning parameters.
n <- 500
p <- 10
X <- matrix(rnorm(n * p), n, p)
W <- rbinom(n, 1, 0.5)
Y <- pmax(X[, 1], 0) * W + X[, 2] + pmin(X[, 3], 0) + rnorm(n)
Y.hat <- predict(regression_forest(X, Y))$predictions
W.hat <- rep(0.5, n)
params <- tune_causal_forest(X, Y, W, Y.hat, W.hat)$params

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

## End(Not run)

---

tune_forest

**Tune a forests**

**Description**

Finds the optimal parameters to be used in training a forest.

**Usage**

tune_forest(
  data,
  nrow.X,
  ncol.X,
  args,
  tune.parameters,
  tune.parameters.defaults,
  num.fit.trees,
  num.fit.reps,
  num.optimize.reps,
  train
)

**Arguments**

data
  The data arguments (output from create_data_matrices) for the forest.

nrow.X
  The number of observations.
tune_instrumental_forest

ncol.X The number of variables.
args The remaining call arguments for the forest.
tune.parameters The vector of parameter names to tune.
tune.parameters.defaults The grf default values for the vector of parameter names to tune.
num.fit.trees The number of trees in each 'mini forest' used to fit the tuning model.
num.fit.reps The number of forests used to fit the tuning model.
num.optimize.reps The number of random parameter values considered when using the model to select the optimal parameters.
train The grf forest training function.

Value

tuning output

Description

Instrumental forest tuning

Usage

tune_instrumental_forest(
  X,
  Y,
  W,
  Z,
  Y.hat,
  W.hat,
  Z.hat,
  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,  
cl.group.size = 2,  
reduced.form.weight = 0,  
tune.parameters = "all",  
tune.num.trees = 200,  
tune.num.reps = 50,  
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.
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. See section 6.1.1 of the GRF paper for further discussion of this quantity.
W.hat Estimates of the treatment propensities E[W | Xi].
Z.hat Estimates of the instrument propensities E[Z | Xi].
sample.weights (experimental) 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 random-Forest 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.

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  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 "all".

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.

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 list consisting of the optimal parameter values (‘params’) along with their debiased error (‘error’).
Examples

```r
## Not run:
# Find the optimal tuning parameters.
n <- 3000; p <- 5
X <- matrix(rbinom(n*p, 1, 0.5), n, p)
Z <- rbinom(n, 1, 0.5)
Q <- rbinom(n, 1, 0.5)
T <- Q * Z
ees <- rnorm(n)
TAU <- X[,1] / 2
Y <- rowSums(X[,1:3]) + TAU * T + Q + eps
Y.hat <- predict(regression_forest(X, Y, num.trees = 500))$predictions
W.hat <- predict(regression_forest(X, T, num.trees = 500))$predictions
Z.hat <- predict(regression_forest(X, Z, num.trees = 500))$predictions
tuned.iv.forest <- instrumental_forest(X, Y, T, Z, Y.hat, W.hat, Z.hat, tune.parameters = "all")
## End(Not run)
```

tune_ll_causal_forest  Local linear forest tuning

Description

Finds the optimal ridge penalty for local linear causal prediction.

Usage

```r
tune_ll_causal_forest(
  forest,
  linear.correction.variables = NULL,
  ll.weight.penalty = FALSE,
  num.threads = NULL,
  lambda.path = NULL
)
```

Arguments

- **forest**: The forest used for prediction.
- **linear.correction.variables**: Variables to use for local linear prediction. If left null, all variables are used. Default is NULL.
- **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.
- **lambda.path**: Optional list of lambdas to use for cross-validation.
Value

A list of lambdas tried, corresponding errors, and optimal ridge penalty lambda.

Examples

```r
## Not run:
# Find the optimal tuning parameters.
N <- 50
p <- 10
X <- matrix(rnorm(n * p), n , p)
W <- rbinom(n, 1 , 0.5)
Y <- pmax(X[, 1] , 0) * W + X[, 2] + pmin(X[, 3] , 0) + rnorm(n)

forest <- causal_forest(X, Y, W)
tuned.lambda <- tune_ll_causal_forest(forest)

# Use this parameter to predict from a local linear causal forest.
predictions <- predict(forest, linear.correction.variables = 1:p, lambda = tuned.lambda)
## End(Not run)
```

---

tune_ll_regression_forest

Local linear forest tuning

Description

Finds the optimal ridge penalty for local linear prediction.

Usage

```r
tune_ll_regression_forest(
  forest,
  linear.correction.variables = NULL,
  ll.weight.penalty = FALSE,
  num.threads = NULL,
  lambda.path = NULL
)
```

Arguments

- `forest`: The forest used for prediction.
- `linear.correction.variables`: Variables to use for local linear prediction. If left null, all variables are used. Default is NULL.
ll.weight.penalty
Option to standardize ridge penalty by covariance (TRUE), or penalize all co-
variates equally (FALSE). Defaults to FALSE.
num.threads
Number of threads used in training. If set to NULL, the software automatically
selects an appropriate amount.
lambda.path
Optional list of lambdas to use for cross-validation.

Value
A list of lambdas tried, corresponding errors, and optimal ridge penalty lambda.

Examples
## Not run:
# Find the optimal tuning parameters.
n <- 500
p <- 10
X <- matrix(rnorm(n * p), n, p)
Y <- X[, 1] * rnorm(n)
forest <- regression_forest(X, Y)
tuned.lambda <- tune_ll_regression_forest(forest)

# Use this parameter to predict from a local linear forest.
predictions <- predict(forest, linear.correction.variables = 1:p, lambda = tuned.lambda)
## End(Not run)
tune_regression_forest

```r
honesty = TRUE,
honesty.fraction = 0.5,
honesty.prune.leaves = TRUE,
alpha = 0.05,
imbalance.penalty = 0,
ci.group.size = 2,
tune.parameters = "all",
tune.num.trees = 50,
tune.num.reps = 100,
tune.num.draws = 1000,
num.threads = NULL,
seed = runif(1, 0, .Machine$integer.max)
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>The covariates used in the regression.</td>
</tr>
<tr>
<td>Y</td>
<td>The outcome.</td>
</tr>
<tr>
<td>sample.weights</td>
<td>(experimental) Weights given to an observation in estimation. If NULL, each</td>
</tr>
<tr>
<td></td>
<td>observation is given the same weight. Default is NULL.</td>
</tr>
<tr>
<td>clusters</td>
<td>Vector of integers or factors specifying which cluster each observation cor-</td>
</tr>
<tr>
<td></td>
<td>responds to. Default is NULL (ignored).</td>
</tr>
<tr>
<td>equalize.cluster.weights</td>
<td>If FALSE, each unit is given the same weight (so that bigger clusters get</td>
</tr>
<tr>
<td></td>
<td>more weight). If TRUE, each cluster is given equal weight in the forest.</td>
</tr>
<tr>
<td></td>
<td>In this case, during training, each tree uses the same number of observa-</td>
</tr>
<tr>
<td></td>
<td>tions from each drawn cluster: If the smallest cluster has ( K ) units,</td>
</tr>
<tr>
<td></td>
<td>then when we sample a cluster during training, we only give a random ( K</td>
</tr>
<tr>
<td></td>
<td>) elements of the cluster to the tree-growing procedure. When estimating</td>
</tr>
<tr>
<td></td>
<td>average treatment effects, each observation is given weight ( 1/\text{cluster</td>
</tr>
<tr>
<td></td>
<td>size} ), so that the total weight of each cluster is the same. Note that,</td>
</tr>
<tr>
<td></td>
<td>if this argument is FALSE, sample weights may also be directly adjusted via</td>
</tr>
<tr>
<td></td>
<td>the sample.weights argument. If this argument is TRUE, sample.weights</td>
</tr>
<tr>
<td></td>
<td>must be set to NULL. Default is FALSE.</td>
</tr>
<tr>
<td>sample.fraction</td>
<td>Fraction of the data used to build each tree. Note: If honesty = TRUE, these</td>
</tr>
<tr>
<td></td>
<td>subsamples will further be cut by a factor of honesty.fraction. Default is</td>
</tr>
<tr>
<td></td>
<td>0.5.</td>
</tr>
<tr>
<td>mtry</td>
<td>Number of variables tried for each split. Default is ( \sqrt{p} + 20 )</td>
</tr>
<tr>
<td></td>
<td>where ( p ) is the number of variables.</td>
</tr>
<tr>
<td>min.node.size</td>
<td>A target for the minimum number of observations in each tree leaf. Note that</td>
</tr>
<tr>
<td></td>
<td>nodes with size smaller than min.node.size can occur, as in the original</td>
</tr>
<tr>
<td></td>
<td>random-forest package. Default is 5.</td>
</tr>
<tr>
<td>honesty</td>
<td>Whether to use honest splitting (i.e., sub-sample splitting). Default is TRUE.</td>
</tr>
<tr>
<td></td>
<td>For a detailed description of honesty, honesty.fraction, honesty.prune.leaves,</td>
</tr>
<tr>
<td></td>
<td>and recommendations for parameter tuning, see the grf algorithm reference.</td>
</tr>
<tr>
<td>honesty.fraction</td>
<td>The fraction of data that will be used for determining splits if honesty = TRUE.</td>
</tr>
<tr>
<td></td>
<td>Corresponds to set ( J_1 ) in the notation of the paper. Default is 0.5 (i.e. half of the data is used for determining splits).</td>
</tr>
</tbody>
</table>
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 "all".

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.

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

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

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

variable_importance

mtry = as.numeric(params["mtry"]),
alpha = as.numeric(params["alpha"]),
imbalance.penalty = as.numeric(params["imbalance.penalty"])
}

## End(Not run)

---

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

**Description**

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

**Usage**

```r
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**

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

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

## End(Not run)
```
Index

average_late, 3
average_partial_effect, 4
average_treatment_effect, 5

best_linear_projection, 7
boosted_regression_forest, 8

causal_forest, 11
custom_forest, 15

get_sample_weights, 17
get_tree, 18
grf, 19

instrumental_forest, 21

leaf_stats.causal_forest, 24
leaf_stats.default, 25
leaf_stats.instrumental_forest, 25
leaf_stats.regression_forest, 26
ll_regression_forest, 26

merge_forests, 29

plot.grf_tree, 30
predict.boosted_regression_forest, 31
predict.causal_forest, 32
predict.custom_forest, 34
predict.instrumental_forest, 35
predict.ll_regression_forest, 36
predict.regression_forest, 37
predict.quantile_forest, 38

print.boosted_regression_forest, 40
print.grf, 41
print.grf_tree, 41
print.tuning_output, 42

quantile_forest, 42

regression_forest, 45

split_frequencies, 47

test_calibration, 48
tune_causal_forest, 49
tune_forest, 52
tune_instrumental_forest, 53
tune_ll_causal_forest, 56
tune_ll_regression_forest, 57
tune_regression_forest, 58

variable_importance, 61