Package 'aorsf'

July 22, 2025

Title Accelerated Oblique Random Forests

Version 0.1.5

Description Fit, interpret, and compute predictions with oblique random forests. Includes support for partial dependence, variable importance, passing customized functions for variable importance and identification of linear combinations of features. Methods for the oblique random survival forest are described in Jaeger et al., (2023) <DOI:10.1080/10618600.2023.2231048>.

License MIT + file LICENSE

URL `https://github.com/ropensci/aorsf`, `https://docs.ropensci.org/aorsf/`

BugReports `https://github.com/ropensci/aorsf/issues/`

Depends R (>= 3.6)

Imports collapse, data.table, lifecycle, R6, Rcpp, utils

Suggests covr, ggplot2, glmnet, knitr, rmarkdown, survival, SurvMetrics, testthat (>= 3.0.0), tibble, units

LinkingTo Rcpp, RcppArmadillo

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

LazyData true

RoxygenNote 7.2.3

NeedsCompilation yes

Author Byron Jaeger [aut, cre] (ORCID: <`https://orcid.org/0000-0001-7399-2299`>),
Nicholas Pajewski [ctb],
Sawyer Welden [ctb],
Christopher Jackson [rev],
Marvin Wright [rev],
Lukas Burk [rev]

Maintainer Byron Jaeger <bjaeger@wakehealth.edu>

Repository CRAN

Date/Publication 2024-05-30 03:40:02 UTC

Contents

`as.data.table.orsf_summary_uni`
Coerce to data.table

Description

Convert an 'orsf_summary' object into a `data.table` object.

Usage

```
## S3 method for class 'orsf_summary_uni'
as.data.table(x, ...)
```

Arguments

`x`	an object of class 'orsf_summary_uni'
`...`	not used

Value

a data.table

Examples

```
## Not run:

library(data.table)

object <- orsf(pbc_orsf, Surv(time, status) ~ . - id, n_tree = 25)

smry <- orsf_summarize_uni(object, n_variables = 2)

as.data.table(smry)


## End(Not run)
```

`orsf` *Oblique Random Forests*

Description

Grow or specify an oblique random forest. While the name `orsf()` implies that this function only works for survival forests, it can be used for classification, regression, or survival forests.

Usage

```
orsf(
  data,
  formula,
  control = NULL,
  weights = NULL,
  n_tree = 500,
  n_split = 5,
  n_retry = 3,
  n_thread = 0,
  mtry = NULL,
  sample_with_replacement = TRUE,
  sample_fraction = 0.632,
  leaf_min_events = 1,
```

```
  leaf_min_obs = 5,
  split_rule = NULL,
  split_min_events = 5,
  split_min_obs = 10,
  split_min_stat = NULL,
  oobag_pred_type = NULL,
  oobag_pred_horizon = NULL,
  oobag_eval_every = NULL,
  oobag_fun = NULL,
  importance = "anova",
  importance_max_pvalue = 0.01,
  group_factors = TRUE,
  tree_seeds = NULL,
  attach_data = TRUE,
  no_fit = FALSE,
  na_action = "fail",
  verbose_progress = FALSE,
  ...
)

orsf_train(object, attach_data = TRUE)
```

Arguments

`data` a data.frame, tibble, or data.table that contains the relevant variables.

`formula` (*formula*) Two sided formula with a single outcome. The terms on the right are names of predictor variables, and the symbol '.' may be used to indicate all variables in the data except the response. The symbol '-' may also be used to indicate removal of a predictor. Details on the response vary depending on forest type:

- *Classification*: The response should be a single variable, and that variable should have type `factor` in `data`.
- *Regression*: The response should be a single variable, and that variable should have typee `double` or `integer` with at least 10 unique numeric values in `data`.
- *Survival*: The response should include a time variable, followed by a status variable, and may be written inside a call to Surv (see examples).

`control` (*orsf_control*) An object returned from one of the `orsf_control` functions: orsf_control_survival, orsf_control_classification, and orsf_control_regression. If `NULL` (the default) will use an accelerated control, which is the fastest available option. For survival and classification, this is Cox and Logistic regression with 1 iteration, and for regression it is ordinary least squares.

`weights` (*numeric vector*) Optional. If given, this input should have length equal to `nrow(data)` for complete or imputed data and should have length equal to `nrow(na.omit(data))` if `na_action` is `"omit"`. As the weights vector is used to count observations and events prior to growing a node for a tree, `orsf()` scales `weights` so that `sum(weights) == nrow(data)`. This helps to make tree depth consistent between weighted and un-weighted fits.

`n_tree`	(*integer*) the number of trees to grow. Default is `n_tree = 500.`
`n_split`	(*integer*) the number of cut-points assessed when splitting a node in decision trees. Default is `n_split = 5`.
`n_retry`	(*integer*) when a node is splittable, but the current linear combination of inputs is unable to provide a valid split, `orsf` will try again with a new linear combination based on a different set of randomly selected predictors, up to `n_retry` times. Default is `n_retry = 3`. Set `n_retry = 0` to prevent any retries.
`n_thread`	(*integer*) number of threads to use while growing trees, computing predictions, and computing importance. Default is 0, which allows a suitable number of threads to be used based on availability.
`mtry`	(*integer*) Number of predictors randomly included as candidates for splitting a node. The default is the smallest integer greater than the square root of the number of total predictors, i.e., `mtry = ceiling(sqrt(number of predictors))`
`sample_with_replacement`	(*logical*) If `TRUE` (the default), observations are sampled with replacement when an in-bag sample is created for a decision tree. If `FALSE`, observations are sampled without replacement and each tree will have an in-bag sample containing `sample_fraction`% of the original sample.
`sample_fraction`	(*double*) the proportion of observations that each trees' in-bag sample will contain, relative to the number of rows in `data`. Only used if `sample_with_replacement` is `FALSE`. Default value is 0.632.
`leaf_min_events`	(*integer*) This input is only relevant for survival analysis, and specifies the minimum number of events in a leaf node. Default is `leaf_min_events = 1`
`leaf_min_obs`	(*integer*) minimum number of observations in a leaf node. Default is `leaf_min_obs = 5`.
`split_rule`	(*character*) how to assess the quality of a potential splitting rule for a node. Valid options for survival are: • 'logrank' : a log-rank test statistic (default). • 'cstat' : Harrell's concordance statistic. For classification, valid options are: • 'gini' : gini impurity (default) • 'cstat' : area underneath the ROC curve (AUC-ROC) For regression, valid options are: • 'variance' : variance reduction (default)
`split_min_events`	(*integer*) minimum number of events required in a node to consider splitting it. Default is `split_min_events = 5`. This input is only relevant for survival trees.
`split_min_obs`	(*integer*) minimum number of observations required in a node to consider splitting it. Default is `split_min_obs = 10`.
`split_min_stat`	(double) minimum test statistic required to split a node. If no splits are found with a statistic exceeding `split_min_stat`, the given node either becomes a leaf or a retry occurs (up to `n_retry` retries). Defaults are

- 3.84 if `split_rule` = `'logrank'`
- 0.55 if `split_rule` = `'cstat'` (see first note below)
- 0.00 if `split_rule` = `'gini'` (see second note below)
- 0.00 if `split_rule` = `'variance'`

Note 1 For C-statistic splitting, if C is < 0.50, we consider the statistic value to be 1 - C to allow for good 'anti-predictive' splits. So, if a C-statistic is initially computed as 0.1, it will be considered as 1 - 0.10 = 0.90.

Note 2 For Gini impurity, a value of 0 and 1 usually indicate the best and worst possible scores, respectively. To make things simple and to avoid introducing a `split_max_stat` input, we flip the values of Gini impurity so that 1 and 0 indicate the best and worst possible scores, respectively.

`oobag_pred_type`

(*character*) The type of out-of-bag predictions to compute while fitting the ensemble. Valid options for any tree type:

- 'none' : don't compute out-of-bag predictions
- 'leaf' : the ID of the predicted leaf is returned for each tree

Valid options for survival:

- 'risk' : probability of event occurring at or before `oobag_pred_horizon` (default).
- 'surv' : 1 - risk.
- 'chf' : cumulative hazard function at `oobag_pred_horizon`.
- 'mort' : mortality, i.e., the number of events expected if all observations in the training data were identical to a given observation.

Valid options for classification:

- 'prob' : probability of each class (default)
- 'class' : class (i.e., which.max(prob))

Valid options for regression:

- 'mean' : mean value (default)

`oobag_pred_horizon`

(*numeric*) A numeric value indicating what time should be used for out-of-bag predictions. Default is the median of the observed times, i.e., `oobag_pred_horizon = median(time)`. This input is only relevant for survival trees that have prediction type of 'risk', 'surv', or 'chf'.

`oobag_eval_every`

(*integer*) The out-of-bag performance of the ensemble will be checked every `oobag_eval_every` trees. So, if `oobag_eval_every = 10`, then out-of-bag performance is checked after growing the 10th tree, the 20th tree, and so on. Default is `oobag_eval_every = n_tree`.

`oobag_fun`

(*function*) to be used for evaluating out-of-bag prediction accuracy every `oobag_eval_every` trees. When `oobag_fun = NULL` (the default), the evaluation statistic is selected based on tree type

- survival: Harrell's C-statistic (1982)
- classification: Area underneath the ROC curve (AUC-ROC)

	• regression: Traditional prediction R-squared if you use your own `oobag_fun` note the following: • `oobag_fun` should have three inputs: `y_mat`, `w_vec`, and `s_vec` • For survival trees, `y_mat` should be a two column matrix with first column named 'time' and second named 'status'. For classification trees, `y_mat` should be a matrix with number of columns = number of distinct classes in the outcome. For regression, `y_mat` should be a matrix with one column. • `s_vec` is a numeric vector containing predictions • `oobag_fun` should return a numeric output of length 1 For more details, see the out-of-bag vignette.
`importance`	(*character*) Indicate method for variable importance: • 'none': no variable importance is computed. • 'anova': compute analysis of variance (ANOVA) importance • 'negate': compute negation importance • 'permute': compute permutation importance For details on these methods, see orsf_vi.
`importance_max_pvalue`	(*double*) Only relevant if `importance` is `"anova"`. The maximum p-value that will register as a positive case when counting the number of times a variable was found to be 'significant' during tree growth. Default is 0.01, as recommended by Menze et al.
`group_factors`	(*logical*) Only relevant if variable importance is being estimated. if `TRUE`, the importance of factor variables will be reported overall by aggregating the importance of individual levels of the factor. If `FALSE`, the importance of individual factor levels will be returned.
`tree_seeds`	(*integer vector*) Optional. if specified, random seeds will be set using the values in `tree_seeds[i]` before growing tree `i`. Two forests grown with the same number of trees and the same seeds will have the exact same out-of-bag samples, making out-of-bag error estimates of the forests more comparable. If `NULL` (the default), seeds are picked at random.
`attach_data`	(*logical*) if `TRUE`, a copy of the training data will be attached to the output. This is required if you plan on using functions like orsf_pd_oob or orsf_summarize_uni to interpret the forest using its training data. Default is `TRUE`.
`no_fit`	(*logical*) if `TRUE`, model fitting steps are defined and saved, but training is not initiated. The object returned can be directly submitted to `orsf_train()` so long as `attach_data` is `TRUE`.
`na_action`	(*character*) what should happen when `data` contains missing values (i.e., `NA` values). Valid options are: • 'fail' : an error is thrown if `data` contains `NA` values • 'omit' : rows in `data` with incomplete data will be dropped • 'impute_meanmode' : missing values for continuous and categorical variables in `data` will be imputed using the mean and mode, respectively.

`verbose_progress`	(*logical*) if `TRUE`, progress messages are printed in the console. If `FALSE` (the default), nothing is printed.
`...`	Further arguments passed to or from other methods (not currently used).
`object`	an untrained 'aorsf' object, created by setting `no_fit = TRUE` in `orsf()`.

Details

Why isn't this function called `orf()`? In its earlier versions, the `aorsf` package was exclusively for *o*blique *r*andom *s*urvival *f*orests.

formula for survival oblique RFs:

- The response in `formula` can be a survival object as returned by the Surv function, but can also just be the time and status variables. I.e., `Surv(time, status) ~ .` works and `time + status ~ .` works
- The response can also be a survival object stored in `data`. For example, `y ~ .` is a valid formula if `data$y` inherits from the `Surv` class.

mtry:

The `mtry` parameter may be temporarily reduced to ensure that linear models used to find combinations of predictors remain stable. This occurs because coefficients in linear model fitting algorithms may become infinite if the number of predictors exceeds the number of observations.

oobag_fun:

If `oobag_fun` is specified, it will be used in to compute negation importance or permutation importance, but it will not have any role for ANOVA importance.

n_thread:

If an R function is to be called from C++ (i.e., user-supplied function to compute out-of-bag error or identify linear combinations of variables), `n_thread` will automatically be set to 1 because attempting to run R functions in multiple threads will cause the R session to crash.

Value

an *obliqueForest* object

What is an oblique decision tree?

Decision trees are developed by splitting a set of training data into two new subsets, with the goal of having more similarity within the new subsets than between them. This splitting process is repeated on the resulting subsets of data until a stopping criterion is met. When the new subsets of data are formed based on a single predictor, the decision tree is said to be axis-based because the splits of the data appear perpendicular to the axis of the predictor. When linear combinations of variables are used instead of a single variable, the tree is oblique because the splits of the data are neither parallel nor at a right angle to the axis

Figure : Decision trees for classification with axis-based splitting (left) and oblique splitting (right). Cases are orange squares; controls are purple circles. Both trees partition the predictor space defined by variables X1 and X2, but the oblique splits do a better job of separating the two classes.

What is a random forest?

Random forests are collections of de-correlated decision trees. Predictions from each tree are aggregated to make an ensemble prediction for the forest. For more details, see Breiman at el, 2001.

Training, out-of-bag error, and testing

In random forests, each tree is grown with a bootstrapped version of the training set. Because bootstrap samples are selected with replacement, each bootstrapped training set contains about two-thirds of instances in the original training set. The 'out-of-bag' data are instances that are *not* in the bootstrapped training set. Each tree in the random forest can make predictions for its out-of-bag data, and the out-of-bag predictions can be aggregated to make an ensemble out-of-bag prediction. Since the out-of-bag data are not used to grow the tree, the accuracy of the ensemble out-of-bag predictions approximate the generalization error of the random forest. Generalization error refers to the error of a random forest's predictions when it is applied to predict outcomes for data that were not used to train it, i.e., testing data.

Examples

```
library(aorsf)
library(magrittr) # for %>%

##
## Attaching package: 'magrittr'

## The following object is masked from 'package:tidyr':
##
##     extract

## The following objects are masked from 'package:testthat':
##
##     equals, is_less_than, not
```

`orsf()` is the entry-point of the `aorsf` package. It can be used to fit classification, regression, and survival forests.

For classification, we fit an oblique RF to predict penguin species using `penguin` data from the magnificent `palmerpenguins` R package

```
# An oblique classification RF
penguin_fit <- orsf(data = penguins_orsf,
                    n_tree = 5,
                    formula = species ~ .)

penguin_fit

## ---------- Oblique random classification forest
##
##      Linear combinations: Accelerated Logistic regression
##           N observations: 333
```

```
##                N classes: 3
##                  N trees: 5
##       N predictors total: 7
##    N predictors per node: 3
##  Average leaves per tree: 4.6
## Min observations in leaf: 5
##           OOB stat value: 0.99
##            OOB stat type: AUC-ROC
##      Variable importance: anova
##
## -----------------------------------------
```

For regression, we use the same data but predict bill length of penguins:

```
# An oblique regression RF
bill_fit <- orsf(data = penguins_orsf,
                 n_tree = 5,
                 formula = bill_length_mm ~ .)

bill_fit
```

```
## ---------- Oblique random regression forest
##
##      Linear combinations: Accelerated Linear regression
##           N observations: 333
##                  N trees: 5
##       N predictors total: 7
##    N predictors per node: 3
##  Average leaves per tree: 51
## Min observations in leaf: 5
##           OOB stat value: 0.70
##            OOB stat type: RSQ
##      Variable importance: anova
##
## -----------------------------------------
```

My personal favorite is the oblique survival RF with accelerated Cox regression because it was the first type of oblique RF that `aorsf` provided (see ArXiv paper; the paper is also published in *Journal of Computational and Graphical Statistics* but is not publicly available there). Here, we use it to predict mortality risk following diagnosis of primary biliary cirrhosis:

```
# An oblique survival RF
pbc_fit <- orsf(data = pbc_orsf,
                n_tree = 5,
                formula = Surv(time, status) ~ . - id)

pbc_fit
```

```
## ---------- Oblique random survival forest
##
##      Linear combinations: Accelerated Cox regression
##           N observations: 276
##                 N events: 111
##                  N trees: 5
##       N predictors total: 17
##    N predictors per node: 5
##  Average leaves per tree: 22.2
## Min observations in leaf: 5
##       Min events in leaf: 1
##           OOB stat value: 0.78
##            OOB stat type: Harrell's C-index
##      Variable importance: anova
##
## -----------------------------------------
```

More than one way to grow a forest:

You can use `orsf(no_fit = TRUE)` to make a *specification* to grow a forest instead of a fitted forest.

```
orsf_spec <- orsf(pbc_orsf,
                  formula = time + status ~ . - id,
                  no_fit = TRUE)

orsf_spec
```

```
## Untrained oblique random survival forest
##
##      Linear combinations: Accelerated Cox regression
##           N observations: 276
##                 N events: 111
##                  N trees: 500
##       N predictors total: 17
##    N predictors per node: 5
##  Average leaves per tree: 0
## Min observations in leaf: 5
##       Min events in leaf: 1
##           OOB stat value: none
##            OOB stat type: Harrell's C-index
##      Variable importance: anova
##
## -----------------------------------------
```

Why would you do this? Two reasons:

1. For very computational tasks, you may want to check how long it will take to fit the forest before you commit to it:

```
orsf_spec %>%
 orsf_update(n_tree = 10000) %>%
 orsf_time_to_train()
```

```
## Time difference of 2.429678 secs
```

1. If fitting multiple forests, use the blueprint along with `orsf_train()` and `orsf_update()` to simplify your code:

```
orsf_fit <- orsf_train(orsf_spec)
orsf_fit_10 <- orsf_update(orsf_fit, leaf_min_obs = 10)
orsf_fit_20 <- orsf_update(orsf_fit, leaf_min_obs = 20)

orsf_fit$leaf_min_obs
```

```
## [1] 5
```

```
orsf_fit_10$leaf_min_obs
```

```
## [1] 10
```

```
orsf_fit_20$leaf_min_obs
```

```
## [1] 20
```

tidymodels:

`tidymodels` includes support for `aorsf` as a computational engine:

```
library(tidymodels)
library(censored)
library(yardstick)

pbc_tidy <- pbc_orsf %>%
 mutate(event_time = Surv(time, status), .before = 1) %>%
 select(-c(id, time, status)) %>%
 as_tibble()

split  <- initial_split(pbc_tidy)

orsf_spec <- rand_forest() %>%
 set_engine("aorsf") %>%
 set_mode("censored regression")

orsf_fit <- fit(orsf_spec,
                formula = event_time ~ .,
                data = training(split))
```

Prediction with `aorsf` models at different times is also supported:

```
time_points <- seq(500, 3000, by = 500)

test_pred <- augment(orsf_fit,
                     new_data = testing(split),
                     eval_time = time_points)

brier_scores <- test_pred %>%
```

```
  brier_survival(truth = event_time, .pred)

brier_scores

## # A tibble: 6 x 4
##   .metric        .estimator .eval_time .estimate
##   <chr>          <chr>           <dbl>     <dbl>
## 1 brier_survival standard          500    0.0597
## 2 brier_survival standard         1000    0.0943
## 3 brier_survival standard         1500    0.0883
## 4 brier_survival standard         2000    0.102
## 5 brier_survival standard         2500    0.137
## 6 brier_survival standard         3000    0.153

roc_scores <- test_pred %>%
  roc_auc_survival(truth = event_time, .pred)

roc_scores

## # A tibble: 6 x 4
##   .metric          .estimator .eval_time .estimate
##   <chr>            <chr>           <dbl>     <dbl>
## 1 roc_auc_survival standard          500     0.957
## 2 roc_auc_survival standard         1000     0.912
## 3 roc_auc_survival standard         1500     0.935
## 4 roc_auc_survival standard         2000     0.931
## 5 roc_auc_survival standard         2500     0.907
## 6 roc_auc_survival standard         3000     0.889
```

References

1. Harrell, E F, Califf, M R, Pryor, B D, Lee, L K, Rosati, A R (1982). "Evaluating the yield of medical tests." *Jama*, *247*(18), 2543-2546.
2. Breiman, Leo (2001). "Random Forests." *Machine Learning*, *45*(1), 5-32. ISSN 1573-0565.
3. Ishwaran H, Kogalur UB, Blackstone EH, Lauer MS (2008). "Random survival forests." *The Annals of Applied Statistics*, *2*(3).
4. Menze, H B, Kelm, Michael B, Splitthoff, N D, Koethe, Ullrich, Hamprecht, A F (2011). "On oblique random forests." In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2011, Athens, Greece, September 5-9, 2011, Proceedings, Part II 22*, 453-469. Springer.
5. Jaeger BC, Long DL, Long DM, Sims M, Szychowski JM, Min Y, Mcclure LA, Howard G, Simon N (2019). "Oblique random survival forests." *The Annals of Applied Statistics*, *13*(3).
6. Jaeger BC, Welden S, Lenoir K, Speiser JL, Segar MW, Pandey A, Pajewski NM (2023). "Accelerated and interpretable oblique random survival forests." *Journal of Computational and Graphical Statistics*, 1-16.

`orsf_control` *Oblique random forest control*

Description

Oblique random forest control

Usage

```
orsf_control(
  tree_type,
  method,
  scale_x,
  ties,
  net_mix,
  target_df,
  max_iter,
  epsilon,
  ...
)

orsf_control_classification(
  method = "glm",
  scale_x = TRUE,
  net_mix = 0.5,
  target_df = NULL,
  max_iter = 20,
  epsilon = 1e-09,
  ...
)

orsf_control_regression(
  method = "glm",
  scale_x = TRUE,
  net_mix = 0.5,
  target_df = NULL,
  max_iter = 20,
  epsilon = 1e-09,
  ...
)

orsf_control_survival(
  method = "glm",
  scale_x = TRUE,
  ties = "efron",
  net_mix = 0.5,
  target_df = NULL,
```

```
  max_iter = 20,
  epsilon = 1e-09,
  ...
)
```

Arguments

`tree_type`	(*character*) the type of tree. Valid options are - "classification", i.e., categorical outcomes - "regression", i.e., continuous outcomes - "survival", i.e., time-to event outcomes
`method`	(*character* or *function*) how to identify linear linear combinations of predictors. If `method` is a character value, it must be one of: - 'glm': linear, logistic, and cox regression - 'net': same as 'glm' but with penalty terms - 'pca': principal component analysis - 'random': random draw from uniform distribution If `method` is a *function*, it will be used to identify linear combinations of predictor variables. `method` must in this case accept three inputs named `x_node`, `y_node` and `w_node`, and should expect the following types and dimensions: - `x_node` (*matrix*; `n` rows, `p` columns) - `y_node` (*matrix*; `n` rows, `2` columns) - `w_node` (*matrix*; `n` rows, `1` column) In addition, `method` must return a matrix with p rows and 1 column.
`scale_x`	(*logical*) if `TRUE`, values of predictors will be scaled prior to each instance of finding a linear combination of predictors, using summary values from the data in the current node of the decision tree.
`ties`	(*character*) a character string specifying the method for tie handling. Only relevant when modeling survival outcomes and using a method that engages with tied outcome times. If there are no ties, all the methods are equivalent. Valid options are 'breslow' and 'efron'. The Efron approximation is the default because it is more accurate when dealing with tied event times and has similar computational efficiency compared to the Breslow method.
`net_mix`	(*double*) The elastic net mixing parameter. A value of 1 gives the lasso penalty, and a value of 0 gives the ridge penalty. If multiple values of alpha are given, then a penalized model is fit using each alpha value prior to splitting a node.
`target_df`	(*integer*) Preferred number of variables used in each linear combination. For example, with `mtry` of 5 and `target_df` of 3, we sample 5 predictors and look for the best linear combination using 3 of them.
`max_iter`	(*integer*) iteration continues until convergence (see `eps` above) or the number of attempted iterations is equal to `iter_max`.
`epsilon`	(*double*) When using most modeling based method, iteration continues in the algorithm until the relative change in some kind of objective is less than `epsilon`, or the absolute change is less than `sqrt(epsilon)`.
`...`	Further arguments passed to or from other methods (not currently used).

Details

Adjust `scale_x` *at your own risk*. Setting `scale_x = FALSE` will reduce computation time but will also make the `orsf` model dependent on the scale of your data, which is why the default value is `TRUE`.

Value

an object of class `'orsf_control'`, which should be used as an input for the `control` argument of orsf. Components are:

- `tree_type`: type of trees to fit
- `lincomb_type`: method for linear combinations
- `lincomb_eps`: epsilon for convergence
- `lincomb_iter_max`: max iterations
- `lincomb_scale`: to scale or not.
- `lincomb_alpha`: mixing parameter
- `lincomb_df_target`: target degrees of freedom
- `lincomb_ties_method`: method for ties in survival time
- `lincomb_R_function`: R function for custom splits

Examples

First we load some relevant packages

```
set.seed(329730)
suppressPackageStartupMessages({
 library(aorsf)
 library(survival)
 library(ranger)
 library(riskRegression)
})
```

Accelerated linear combinations:

The accelerated ORSF ensemble is the default because it has a nice balance of computational speed and prediction accuracy. It runs a single iteration of Newton Raphson scoring on the Cox partial likelihood function to find linear combinations of predictors.

```
fit_accel <- orsf(pbc_orsf,
                  control = orsf_control_survival(),
                  formula = Surv(time, status) ~ . - id,
                  tree_seeds = 329)
```

Linear combinations with Cox regression:

Setting inputs in `orsf_control_survival` to scale the X matrix and repeat iterations until convergence allows you to run Cox regression in each non-terminal node of each survival tree, using the regression coefficients to create linear combinations of predictors:

```
control_cph <- orsf_control_survival(method = 'glm',
                                     scale_x = TRUE,
                                     max_iter = 20)

fit_cph <- orsf(pbc_orsf,
                control = control_cph,
                formula = Surv(time, status) ~ . - id,
                tree_seeds = 329)
```

Linear combinations with penalized cox regression:

Setting `method == 'net'` runs penalized Cox regression in each non-terminal node of each survival tree. This can be really helpful if you want to do feature selection within the node, but it is a lot slower than the `'glm'` option.

```
# select 3 predictors out of 5 to be used in
# each linear combination of predictors.

control_net <- orsf_control_survival(method = 'net', target_df = 3)

fit_net <- orsf(pbc_orsf,
                control = control_net,
                formula = Surv(time, status) ~ . - id,
                tree_seeds = 329)
```

Linear combinations with your own function:

In addition to the built-in methods, customized functions can be used to identify linear combinations of predictors. We'll demonstrate a few here.

- The first uses random coefficients

```
f_rando <- function(x_node, y_node, w_node){
 matrix(runif(ncol(x_node)), ncol=1)
}
```

- The second derives coefficients from principal component analysis

```
f_pca <- function(x_node, y_node, w_node) {

 # estimate two principal components.
 pca <- stats::prcomp(x_node, rank. = 2)
 # use the second principal component to split the node
 pca$rotation[, 1L, drop = FALSE]

}
```

- The third uses `ranger()` inside of `orsf()`. This approach is very similar to a method known as reinforcement learning trees (see the `RLT` package), although our method of "muting" is very crude compared to the method proposed by Zhu et al.

```
f_rlt <- function(x_node, y_node, w_node){

 colnames(y_node) <- c('time', 'status')
```

```
 colnames(x_node) <- paste("x", seq(ncol(x_node)), sep = '')

 data <- as.data.frame(cbind(y_node, x_node))

 if(nrow(data) <= 10)
  return(matrix(runif(ncol(x_node)), ncol = 1))

 fit <- ranger::ranger(data = data,
                       formula = Surv(time, status) ~ .,
                       num.trees = 25,
                       num.threads = 1,
                       min.node.size = 5,
                       importance = 'permutation')

 out <- sort(fit$variable.importance, decreasing = TRUE)

 # "mute" the least two important variables
 n_vars <- length(out)
 if(n_vars > 4){
   out[c(n_vars, n_vars-1)] <- 0
 }

 # ensure out has same variable order as input
 out <- out[colnames(x_node)]

 # protect yourself
 out[is.na(out)] <- 0

 matrix(out, ncol = 1)

}
```

We can plug these functions into `orsf_control_custom()`, and then pass the result into `orsf()`:

```
fit_rando <- orsf(pbc_orsf,
                  Surv(time, status) ~ . - id,
                  control = orsf_control_survival(method = f_rando),
                  tree_seeds = 329)

fit_pca <- orsf(pbc_orsf,
                Surv(time, status) ~ . - id,
                control = orsf_control_survival(method = f_pca),
                tree_seeds = 329)

fit_rlt <- orsf(pbc_orsf, time + status ~ . - id,
                control = orsf_control_survival(method = f_rlt),
                tree_seeds = 329)
```

So which fit seems to work best in this example? Let's find out by evaluating the out-of-bag survival predictions.

```
risk_preds <- list(
 accel = fit_accel$pred_oobag,
 cph   = fit_cph$pred_oobag,
 net   = fit_net$pred_oobag,
 rando = fit_rando$pred_oobag,
 pca   = fit_pca$pred_oobag,
 rlt   = fit_rlt$pred_oobag
)

sc <- Score(object = risk_preds,
            formula = Surv(time, status) ~ 1,
            data = pbc_orsf,
            summary = 'IPA',
            times = fit_accel$pred_horizon)
```

The AUC values, from highest to lowest:

```
sc$AUC$score[order(-AUC)]
```

```
##     model times       AUC         se     lower     upper
##    <fctr> <num>     <num>      <num>     <num>     <num>
## 1:    net  1788 0.9151649 0.02025057 0.8754745 0.9548553
## 2:    rlt  1788 0.9119200 0.02090107 0.8709547 0.9528854
## 3:  accel  1788 0.9095628 0.02143250 0.8675558 0.9515697
## 4:    cph  1788 0.9095628 0.02143250 0.8675558 0.9515697
## 5:  rando  1788 0.9062197 0.02148854 0.8641029 0.9483365
## 6:    pca  1788 0.8999479 0.02226683 0.8563057 0.9435901
```

And the indices of prediction accuracy:

```
sc$Brier$score[order(-IPA), .(model, times, IPA)]
```

```
##         model times       IPA
##        <fctr> <num>     <num>
## 1:        net  1788 0.4905777
## 2:      accel  1788 0.4806649
## 3:        cph  1788 0.4806649
## 4:        rlt  1788 0.4675228
## 5:        pca  1788 0.4383995
## 6:      rando  1788 0.4302814
## 7: Null model  1788 0.0000000
```

From inspection,

- `net`, `accel`, and `rlt` have high discrimination and index of prediction accuracy.
- `rando` and `pca` do less well, but they aren't bad.

See Also

linear combination control functions `orsf_control_cph()`, `orsf_control_custom()`, `orsf_control_fast()`, `orsf_control_net()`

`orsf_control_cph` *Cox regression ORSF control*

Description

Use the coefficients from a proportional hazards model to create linear combinations of predictor variables while fitting an orsf model.

Usage

```
orsf_control_cph(method = "efron", eps = 1e-09, iter_max = 20, ...)
```

Arguments

`method`	(*character*) a character string specifying the method for tie handling. If there are no ties, all the methods are equivalent. Valid options are 'breslow' and 'efron'. The Efron approximation is the default because it is more accurate when dealing with tied event times and has similar computational efficiency compared to the Breslow method.
`eps`	(*double*) When using Newton Raphson scoring to identify linear combinations of inputs, iteration continues in the algorithm until the relative change in the log partial likelihood is less than `eps`, or the absolute change is less than `sqrt(eps)`. Must be positive. A default value of 1e-09 is used for consistency with survival::coxph.control.
`iter_max`	(*integer*) iteration continues until convergence (see `eps` above) or the number of attempted iterations is equal to `iter_max`.
`...`	Further arguments passed to or from other methods (not currently used).

Details

[Superseded]

code from the survival package was modified to make this routine.

For more details on the Cox proportional hazards model, see coxph and/or Therneau and Grambsch (2000).

Value

an object of class `'orsf_control'`, which should be used as an input for the `control` argument of orsf.

References

Therneau T.M., Grambsch P.M. (2000) The Cox Model. In: Modeling Survival Data: Extending the Cox Model. Statistics for Biology and Health. Springer, New York, NY. DOI: 10.1007/978-1-4757-3294-8_3

See Also

linear combination control functions `orsf_control_custom()`, `orsf_control_fast()`, `orsf_control_net()`, `orsf_control()`

`orsf_control_custom` *Custom ORSF control*

Description

[Superseded]

Usage

```
orsf_control_custom(beta_fun, ...)
```

Arguments

`beta_fun`	(*function*) a function to define coefficients used in linear combinations of predictor variables. `beta_fun` must accept three inputs named `x_node`, `y_node` and `w_node`, and should expect the following types and dimensions: • `x_node` (*matrix*; `n` rows, `p` columns) • `y_node` (*matrix*; `n` rows, `2` columns) • `w_node` (*matrix*; `n` rows, `1` column) In addition, `beta_fun` must return a matrix with p rows and 1 column. If any of these conditions are not met, `orsf_control_custom()` will let you know.
`...`	Further arguments passed to or from other methods (not currently used).

Value

an object of class `'orsf_control'`, which should be used as an input for the `control` argument of orsf.

See Also

linear combination control functions `orsf_control_cph()`, `orsf_control_fast()`, `orsf_control_net()`, `orsf_control()`

`orsf_control_fast` *Accelerated ORSF control*

Description

Fast methods to identify linear combinations of predictors while fitting an orsf model.

Usage

```
orsf_control_fast(method = "efron", do_scale = TRUE, ...)
```

Arguments

`method`	(*character*) a character string specifying the method for tie handling. If there are no ties, all the methods are equivalent. Valid options are 'breslow' and 'efron'. The Efron approximation is the default because it is more accurate when dealing with tied event times and has similar computational efficiency compared to the Breslow method.
`do_scale`	(*logical*) if `TRUE`, values of predictors will be scaled prior to each instance of Newton Raphson scoring, using summary values from the data in the current node of the decision tree.
`...`	Further arguments passed to or from other methods (not currently used).

Details

code from the survival package was modified to make this routine.

Adjust `do_scale` *at your own risk*. Setting `do_scale = FALSE` will reduce computation time but will also make the `orsf` model dependent on the scale of your data, which is why the default value is `TRUE`.

Value

an object of class `'orsf_control'`, which should be used as an input for the `control` argument of orsf.

See Also

linear combination control functions `orsf_control_cph()`, `orsf_control_custom()`, `orsf_control_net()`, `orsf_control()`

`orsf_control_net` *Penalized Cox regression ORSF control*

Description

Use regularized Cox proportional hazard models to identify linear combinations of input variables while fitting an orsf model.

Usage

```
orsf_control_net(alpha = 1/2, df_target = NULL, ...)
```

Arguments

`alpha`	(*double*) The elastic net mixing parameter. A value of 1 gives the lasso penalty, and a value of 0 gives the ridge penalty. If multiple values of alpha are given, then a penalized model is fit using each alpha value prior to splitting a node.
`df_target`	(*integer*) Preferred number of variables used in a linear combination.
`...`	Further arguments passed to or from other methods (not currently used).

Details

[Superseded]

`df_target` has to be less than `mtry`, which is a separate argument in orsf that indicates the number of variables chosen at random prior to finding a linear combination of those variables.

Value

an object of class `'orsf_control'`, which should be used as an input for the `control` argument of orsf.

References

1. Simon, Noah, Friedman, Jerome, Hastie, Trevor, Tibshirani, Rob (2011). "Regularization paths for Cox's proportional hazards model via coordinate descent." *Journal of statistical software*, *39*(5), 1.

See Also

linear combination control functions `orsf_control_cph()`, `orsf_control_custom()`, `orsf_control_fast()`, `orsf_control()`

`orsf_ice_oob` *Individual Conditional Expectations*

Description

Compute individual conditional expectations for an oblique random forest. Unlike partial dependence, which shows the expected prediction as a function of one or multiple predictors, individual conditional expectations (ICE) show the prediction for an individual observation as a function of a predictor. You can compute individual conditional expectations three ways using a random forest:

- using in-bag predictions for the training data
- using out-of-bag predictions for the training data
- using predictions for a new set of data

See examples for more details

Usage

```
orsf_ice_oob(
  object,
  pred_spec,
  pred_horizon = NULL,
  pred_type = NULL,
  expand_grid = TRUE,
  boundary_checks = TRUE,
  n_thread = NULL,
  verbose_progress = NULL,
  ...
)

orsf_ice_inb(
  object,
  pred_spec,
  pred_horizon = NULL,
  pred_type = NULL,
  expand_grid = TRUE,
  boundary_checks = TRUE,
  n_thread = NULL,
  verbose_progress = NULL,
  ...
)

orsf_ice_new(
  object,
  pred_spec,
  new_data,
  pred_horizon = NULL,
```

```
  pred_type = NULL,
  na_action = "fail",
  expand_grid = TRUE,
  boundary_checks = TRUE,
  n_thread = NULL,
  verbose_progress = NULL,
  ...
)
```

Arguments

`object`	(*ObliqueForest*) a trained oblique random forest object (see orsf).
`pred_spec`	(*named list*, *pspec_auto*, or *data.frame*). • If `pred_spec` is a named list, Each item in the list should be a vector of values that will be used as points in the partial dependence function. The name of each item in the list should indicate which variable will be modified to take the corresponding values. • If `pred_spec` is created using `pred_spec_auto()`, all that is needed is the names of variables to use (see pred_spec_auto). • If `pred_spec` is a `data.frame`, columns will indicate variable names, values will indicate variable values, and partial dependence will be computed using the inputs on each row.
`pred_horizon`	(*double*) Only relevent for survival forests. A value or vector indicating the time(s) that predictions will be calibrated to. E.g., if you were predicting risk of incident heart failure within the next 10 years, then `pred_horizon = 10`. `pred_horizon` can be `NULL` if `pred_type` is `'mort'`, since mortality predictions are aggregated over all event times
`pred_type`	(*character*) the type of predictions to compute. Valid Valid options for survival are: • 'risk' : probability of having an event at or before `pred_horizon`. • 'surv' : 1 - risk. • 'chf': cumulative hazard function • 'mort': mortality prediction • 'time': survival time prediction For classification: • 'prob': probability for each class For regression: • 'mean': predicted mean, i.e., the expected value
`expand_grid`	(*logical*) if `TRUE`, partial dependence will be computed at all possible combinations of inputs in `pred_spec`. If `FALSE`, partial dependence will be computed for each variable in `pred_spec`, separately.
`boundary_checks`	(*logical*) if `TRUE`, `pred_spec` will be checked to make sure the requested values are between the 10th and 90th percentile in the object's training data. If `FALSE`, these checks are skipped.

`n_thread`	(*integer*) number of threads to use while computing predictions. Default is 0, which allows a suitable number of threads to be used based on availability.
`verbose_progress`	(*logical*) if `TRUE`, progress will be printed to console. If `FALSE` (the default), nothing will be printed.
`...`	Further arguments passed to or from other methods (not currently used).
`new_data`	a data.frame, tibble, or data.table to compute predictions in.
`na_action`	(*character*) what should happen when `new_data` contains missing values (i.e., `NA` values). Valid options are: • 'fail' : an error is thrown if `new_data` contains `NA` values • 'omit' : rows in `new_data` with incomplete data will be dropped

Value

a data.table containing individual conditional expectations for the specified variable(s) and, if relevant, at the specified prediction horizon(s).

Examples

You can compute individual conditional expectation and individual conditional expectations in three ways:

- using in-bag predictions for the training data. In-bag individual conditional expectation indicates relationships that the model has learned during training. This is helpful if your goal is to interpret the model.
- using out-of-bag predictions for the training data. Out-of-bag individual conditional expectation indicates relationships that the model has learned during training but using the out-of-bag data simulates application of the model to new data. This is helpful if you want to test your model's reliability or fairness in new data but you don't have access to a large testing set.
- using predictions for a new set of data. New data individual conditional expectation shows how the model predicts outcomes for observations it has not seen. This is helpful if you want to test your model's reliability or fairness.

Classification:

Begin by fitting an oblique classification random forest:

```
set.seed(329)

index_train <- sample(nrow(penguins_orsf), 150)

penguins_orsf_train <- penguins_orsf[index_train, ]
penguins_orsf_test <- penguins_orsf[-index_train, ]

fit_clsf <- orsf(data = penguins_orsf_train,
                 formula = species ~ .)
```

Compute individual conditional expectation using out-of-bag data for `flipper_length_mm = c(190, 210)`.

```
pred_spec <- list(flipper_length_mm = c(190, 210))

ice_oob <- orsf_ice_oob(fit_clsf, pred_spec = pred_spec)

ice_oob
```

```
## Key: <class>
##      id_variable id_row  class flipper_length_mm       pred
##            <int> <char> <fctr>             <num>      <num>
##   1:           1      1 Adelie               190 0.92169247
##   2:           1      2 Adelie               190 0.80944657
##   3:           1      3 Adelie               190 0.85172955
##   4:           1      4 Adelie               190 0.93559327
##   5:           1      5 Adelie               190 0.97708693
##  ---
## 896:           2    146 Gentoo               210 0.26092984
## 897:           2    147 Gentoo               210 0.04798334
## 898:           2    148 Gentoo               210 0.07927359
## 899:           2    149 Gentoo               210 0.84779971
## 900:           2    150 Gentoo               210 0.11105143
```

There are two identifiers in the output:

- `id_variable` is an identifier for the current value of the variable(s) that are in the data. It is redundant if you only have one variable, but helpful if there are multiple variables.
- `id_row` is an identifier for the observation in the original data.

Note that predicted probabilities are returned for each class and each observation in the data. Predicted probabilities for a given observation and given variable value sum to 1. For example,

```
ice_oob %>%
 .[flipper_length_mm == 190] %>%
 .[id_row == 1] %>%
 .[['pred']] %>%
 sum()
```

```
## [1] 1
```

Regression:

Begin by fitting an oblique regression random forest:

```
set.seed(329)

index_train <- sample(nrow(penguins_orsf), 150)

penguins_orsf_train <- penguins_orsf[index_train, ]
penguins_orsf_test <- penguins_orsf[-index_train, ]

fit_regr <- orsf(data = penguins_orsf_train,
                 formula = bill_length_mm ~ .)
```

Compute individual conditional expectation using new data for `flipper_length_mm = c(190, 210)`.

```
pred_spec <- list(flipper_length_mm = c(190, 210))

ice_new <- orsf_ice_new(fit_regr,
                        pred_spec = pred_spec,
                        new_data = penguins_orsf_test)

ice_new
```

```
##      id_variable id_row flipper_length_mm     pred
##            <int> <char>             <num>    <num>
##   1:           1      1               190 37.94483
##   2:           1      2               190 37.61595
##   3:           1      3               190 37.53681
##   4:           1      4               190 39.49476
##   5:           1      5               190 38.95635
##  ---
## 362:           2    179               210 51.80471
## 363:           2    180               210 47.27183
## 364:           2    181               210 47.05031
## 365:           2    182               210 50.39028
## 366:           2    183               210 48.44774
```

You can also let `pred_spec_auto` pick reasonable values like so:

```
pred_spec = pred_spec_auto(species, island, body_mass_g)

ice_new <- orsf_ice_new(fit_regr,
                        pred_spec = pred_spec,
                        new_data = penguins_orsf_test)

ice_new
```

```
##       id_variable id_row species    island body_mass_g     pred
##             <int> <char>  <fctr>    <fctr>       <num>    <num>
##    1:           1      1  Adelie    Biscoe        3200 37.78339
##    2:           1      2  Adelie    Biscoe        3200 37.73273
##    3:           1      3  Adelie    Biscoe        3200 37.71248
##    4:           1      4  Adelie    Biscoe        3200 40.25782
##    5:           1      5  Adelie    Biscoe        3200 40.04074
##   ---
## 8231:          45    179  Gentoo Torgersen        5300 46.14559
## 8232:          45    180  Gentoo Torgersen        5300 43.98050
## 8233:          45    181  Gentoo Torgersen        5300 44.59837
## 8234:          45    182  Gentoo Torgersen        5300 44.85146
## 8235:          45    183  Gentoo Torgersen        5300 44.23710
```

By default, all combinations of all variables are used. However, you can also look at the variables one by one, separately, like so:

```
ice_new <- orsf_ice_new(fit_regr,
                        expand_grid = FALSE,
```

```
                        pred_spec = pred_spec,
                        new_data = penguins_orsf_test)

ice_new

##       id_variable id_row    variable value  level     pred
##             <int> <char>      <char> <num> <char>    <num>
##    1:           1      1     species    NA Adelie 37.74136
##    2:           1      2     species    NA Adelie 37.42367
##    3:           1      3     species    NA Adelie 37.04598
##    4:           1      4     species    NA Adelie 39.89602
##    5:           1      5     species    NA Adelie 39.14848
##   ---
## 2009:           5    179 body_mass_g  5300   <NA> 51.50196
## 2010:           5    180 body_mass_g  5300   <NA> 47.27055
## 2011:           5    181 body_mass_g  5300   <NA> 48.34064
## 2012:           5    182 body_mass_g  5300   <NA> 48.75828
## 2013:           5    183 body_mass_g  5300   <NA> 48.11020
```

And you can also bypass all the bells and whistles by using your own `data.frame` for a `pred_spec`. (Just make sure you request values that exist in the training data.)

```
custom_pred_spec <- data.frame(species = 'Adelie',
                               island = 'Biscoe')

ice_new <- orsf_ice_new(fit_regr,
                        pred_spec = custom_pred_spec,
                        new_data = penguins_orsf_test)

ice_new

##      id_variable id_row species island     pred
##            <int> <char>  <fctr> <fctr>    <num>
##   1:           1      1  Adelie Biscoe 38.52327
##   2:           1      2  Adelie Biscoe 38.32073
##   3:           1      3  Adelie Biscoe 37.71248
##   4:           1      4  Adelie Biscoe 41.68380
##   5:           1      5  Adelie Biscoe 40.91140
##  ---
## 179:           1    179  Adelie Biscoe 43.09493
## 180:           1    180  Adelie Biscoe 38.79455
## 181:           1    181  Adelie Biscoe 39.37734
## 182:           1    182  Adelie Biscoe 40.71952
## 183:           1    183  Adelie Biscoe 39.34501
```

Survival:

Begin by fitting an oblique survival random forest:

```
set.seed(329)
```

```
index_train <- sample(nrow(pbc_orsf), 150)

pbc_orsf_train <- pbc_orsf[index_train, ]
pbc_orsf_test <- pbc_orsf[-index_train, ]

fit_surv <- orsf(data = pbc_orsf_train,
                 formula = Surv(time, status) ~ . - id,
                 oobag_pred_horizon = 365.25 * 5)
```

Compute individual conditional expectation using in-bag data for `bili = c(1,2,3,4,5)`:

```
ice_train <- orsf_ice_inb(fit_surv, pred_spec = list(bili = 1:5))
ice_train
```

```
##      id_variable id_row pred_horizon  bili      pred
##            <int> <char>        <num> <num>     <num>
##   1:           1      1      1826.25     1 0.1290317
##   2:           1      2      1826.25     1 0.1242352
##   3:           1      3      1826.25     1 0.0963452
##   4:           1      4      1826.25     1 0.1172367
##   5:           1      5      1826.25     1 0.2030256
##  ---
## 746:           5    146      1826.25     5 0.7868537
## 747:           5    147      1826.25     5 0.2012954
## 748:           5    148      1826.25     5 0.4893605
## 749:           5    149      1826.25     5 0.4698220
## 750:           5    150      1826.25     5 0.9557285
```

If you don't have specific values of a variable in mind, let `pred_spec_auto` pick for you:

```
ice_train <- orsf_ice_inb(fit_surv, pred_spec_auto(bili))
ice_train
```

```
##      id_variable id_row pred_horizon  bili       pred
##            <int> <char>        <num> <num>      <num>
##   1:           1      1      1826.25  0.55 0.11728559
##   2:           1      2      1826.25  0.55 0.11728839
##   3:           1      3      1826.25  0.55 0.08950739
##   4:           1      4      1826.25  0.55 0.10064959
##   5:           1      5      1826.25  0.55 0.18736417
##  ---
## 746:           5    146      1826.25  7.25 0.82600898
## 747:           5    147      1826.25  7.25 0.29156437
## 748:           5    148      1826.25  7.25 0.58395919
## 749:           5    149      1826.25  7.25 0.54202021
## 750:           5    150      1826.25  7.25 0.96391985
```

Specify `pred_horizon` to get individual conditional expectation at each value:

```
ice_train <- orsf_ice_inb(fit_surv, pred_spec_auto(bili),
                          pred_horizon = seq(500, 3000, by = 500))
ice_train
```

```
##       id_variable id_row pred_horizon  bili        pred
##             <int> <char>        <num> <num>       <num>
##    1:           1      1          500  0.55 0.008276627
##    2:           1      1         1000  0.55 0.055724516
##    3:           1      1         1500  0.55 0.085091120
##    4:           1      1         2000  0.55 0.123423352
##    5:           1      1         2500  0.55 0.166380739
##   ---
## 4496:           5    150         1000  7.25 0.837774757
## 4497:           5    150         1500  7.25 0.934536379
## 4498:           5    150         2000  7.25 0.967823286
## 4499:           5    150         2500  7.25 0.972059574
## 4500:           5    150         3000  7.25 0.980785643
```

Multi-prediction horizon ice comes with minimal extra computational cost. Use a fine grid of time values and assess whether predictors have time-varying effects.

`orsf_pd_oob` *Partial dependence*

Description

Compute partial dependence for an oblique random forest. Partial dependence (PD) shows the expected prediction from a model as a function of a single predictor or multiple predictors. The expectation is marginalized over the values of all other predictors, giving something like a multivariable adjusted estimate of the model's prediction. You can compute partial dependence three ways using a random forest:

- using in-bag predictions for the training data
- using out-of-bag predictions for the training data
- using predictions for a new set of data

See examples for more details

Usage

```
orsf_pd_oob(
  object,
  pred_spec,
  pred_horizon = NULL,
  pred_type = NULL,
  expand_grid = TRUE,
  prob_values = c(0.025, 0.5, 0.975),
  prob_labels = c("lwr", "medn", "upr"),
  boundary_checks = TRUE,
  n_thread = NULL,
  verbose_progress = NULL,
```

```
  ...
)

orsf_pd_inb(
  object,
  pred_spec,
  pred_horizon = NULL,
  pred_type = NULL,
  expand_grid = TRUE,
  prob_values = c(0.025, 0.5, 0.975),
  prob_labels = c("lwr", "medn", "upr"),
  boundary_checks = TRUE,
  n_thread = NULL,
  verbose_progress = NULL,
  ...
)

orsf_pd_new(
  object,
  pred_spec,
  new_data,
  pred_horizon = NULL,
  pred_type = NULL,
  na_action = "fail",
  expand_grid = TRUE,
  prob_values = c(0.025, 0.5, 0.975),
  prob_labels = c("lwr", "medn", "upr"),
  boundary_checks = TRUE,
  n_thread = NULL,
  verbose_progress = NULL,
  ...
)
```

Arguments

`object`	(*ObliqueForest*) a trained oblique random forest object (see orsf).
`pred_spec`	(*named list*, *pspec_auto*, or *data.frame*). • If `pred_spec` is a named list, Each item in the list should be a vector of values that will be used as points in the partial dependence function. The name of each item in the list should indicate which variable will be modified to take the corresponding values. • If `pred_spec` is created using `pred_spec_auto()`, all that is needed is the names of variables to use (see pred_spec_auto). • If `pred_spec` is a `data.frame`, columns will indicate variable names, values will indicate variable values, and partial dependence will be computed using the inputs on each row.
`pred_horizon`	(*double*) Only relevent for survival forests. A value or vector indicating the time(s) that predictions will be calibrated to. E.g., if you were predicting risk

	of incident heart failure within the next 10 years, then `pred_horizon = 10`. `pred_horizon` can be `NULL` if `pred_type` is `'mort'`, since mortality predictions are aggregated over all event times
`pred_type`	(*character*) the type of predictions to compute. Valid Valid options for survival are: - 'risk' : probability of having an event at or before `pred_horizon`. - 'surv' : 1 - risk. - 'chf': cumulative hazard function - 'mort': mortality prediction - 'time': survival time prediction For classification: - 'prob': probability for each class For regression: - 'mean': predicted mean, i.e., the expected value
`expand_grid`	(*logical*) if `TRUE`, partial dependence will be computed at all possible combinations of inputs in `pred_spec`. If `FALSE`, partial dependence will be computed for each variable in `pred_spec`, separately.
`prob_values`	(*numeric*) a vector of values between 0 and 1, indicating what quantiles will be used to summarize the partial dependence values at each set of inputs. `prob_values` should have the same length as `prob_labels`. The quantiles are calculated based on predictions from `object` at each set of values indicated by `pred_spec`.
`prob_labels`	(*character*) a vector of labels with the same length as `prob_values`, with each label indicating what the corresponding value in `prob_values` should be labelled as in summarized outputs. `prob_labels` should have the same length as `prob_values`.
`boundary_checks`	(*logical*) if `TRUE`, `pred_spec` will be checked to make sure the requested values are between the 10th and 90th percentile in the object's training data. If `FALSE`, these checks are skipped.
`n_thread`	(*integer*) number of threads to use while computing predictions. Default is 0, which allows a suitable number of threads to be used based on availability.
`verbose_progress`	(*logical*) if `TRUE`, progress will be printed to console. If `FALSE` (the default), nothing will be printed.
`...`	Further arguments passed to or from other methods (not currently used).
`new_data`	a data.frame, tibble, or data.table to compute predictions in.
`na_action`	(*character*) what should happen when `new_data` contains missing values (i.e., `NA` values). Valid options are: - 'fail' : an error is thrown if `new_data` contains `NA` values - 'omit' : rows in `new_data` with incomplete data will be dropped

Details

Partial dependence has a number of known limitations and assumptions that users should be aware of (see Hooker, 2021). In particular, partial dependence is less intuitive when >2 predictors are examined jointly, and it is assumed that the feature(s) for which the partial dependence is computed are not correlated with other features (this is likely not true in many cases). Accumulated local effect plots can be used (see here) in the case where feature independence is not a valid assumption.

Value

a data.table containing partial dependence values for the specified variable(s) and, if relevant, at the specified prediction horizon(s).

Examples

You can compute partial dependence and individual conditional expectations in three ways:

- using in-bag predictions for the training data. In-bag partial dependence indicates relationships that the model has learned during training. This is helpful if your goal is to interpret the model.
- using out-of-bag predictions for the training data. Out-of-bag partial dependence indicates relationships that the model has learned during training but using the out-of-bag data simulates application of the model to new data. This is helpful if you want to test your model's reliability or fairness in new data but you don't have access to a large testing set.
- using predictions for a new set of data. New data partial dependence shows how the model predicts outcomes for observations it has not seen. This is helpful if you want to test your model's reliability or fairness.

Classification:

Begin by fitting an oblique classification random forest:

```
set.seed(329)

index_train <- sample(nrow(penguins_orsf), 150)

penguins_orsf_train <- penguins_orsf[index_train, ]
penguins_orsf_test <- penguins_orsf[-index_train, ]

fit_clsf <- orsf(data = penguins_orsf_train,
                 formula = species ~ .)
```

Compute partial dependence using out-of-bag data for `flipper_length_mm = c(190, 210)`.

```
pred_spec <- list(flipper_length_mm = c(190, 210))

pd_oob <- orsf_pd_oob(fit_clsf, pred_spec = pred_spec)

pd_oob
```

```
## Key: <class>
##        class flipper_length_mm      mean         lwr       medn       upr
##       <fctr>             <num>     <num>       <num>      <num>     <num>
```

```
## 1:    Adelie               190 0.6176908 0.202278109 0.75856417 0.9810614
## 2:    Adelie               210 0.4338528 0.019173811 0.56489202 0.8648110
## 3: Chinstrap               190 0.2114979 0.017643385 0.15211271 0.7215181
## 4: Chinstrap               210 0.1803019 0.020108201 0.09679464 0.7035053
## 5:    Gentoo               190 0.1708113 0.001334861 0.02769695 0.5750201
## 6:    Gentoo               210 0.3858453 0.068685035 0.20717073 0.9532853
```

Note that predicted probabilities are returned for each class and probabilities in the `mean` column sum to 1 if you take the sum over each class at a specific value of the `pred_spec` variables. For example,

```
sum(pd_oob[flipper_length_mm == 190, mean])
```

```
## [1] 1
```

But this isn't the case for the median predicted probability!

```
sum(pd_oob[flipper_length_mm == 190, medn])
```

```
## [1] 0.9383738
```

Regression:

Begin by fitting an oblique regression random forest:

```
set.seed(329)

index_train <- sample(nrow(penguins_orsf), 150)

penguins_orsf_train <- penguins_orsf[index_train, ]
penguins_orsf_test <- penguins_orsf[-index_train, ]

fit_regr <- orsf(data = penguins_orsf_train,
                 formula = bill_length_mm ~ .)
```

Compute partial dependence using new data for `flipper_length_mm = c(190, 210)`.

```
pred_spec <- list(flipper_length_mm = c(190, 210))

pd_new <- orsf_pd_new(fit_regr,
                      pred_spec = pred_spec,
                      new_data = penguins_orsf_test)

pd_new
```

```
##    flipper_length_mm     mean      lwr     medn      upr
##                <num>    <num>    <num>    <num>    <num>
## 1:               190 42.96571 37.09805 43.69769 48.72301
## 2:               210 45.66012 40.50693 46.31577 51.65163
```

You can also let `pred_spec_auto` pick reasonable values like so:

```
pred_spec = pred_spec_auto(species, island, body_mass_g)

pd_new <- orsf_pd_new(fit_regr,
                      pred_spec = pred_spec,
                      new_data = penguins_orsf_test)

pd_new
```

```
##       species    island body_mass_g     mean      lwr     medn      upr
##        <fctr>    <fctr>       <num>    <num>    <num>    <num>    <num>
##  1:    Adelie    Biscoe        3200 40.31374 37.24373 40.31967 44.22824
##  2: Chinstrap    Biscoe        3200 45.10582 42.63342 45.10859 47.60119
##  3:    Gentoo    Biscoe        3200 42.81649 40.19221 42.55664 46.84035
##  4:    Adelie     Dream        3200 40.16219 36.95895 40.34633 43.90681
##  5: Chinstrap     Dream        3200 46.21778 43.53954 45.90929 49.19173
## ---
## 41: Chinstrap     Dream        5300 48.48139 46.36282 48.25679 51.02996
## 42:    Gentoo     Dream        5300 45.91819 43.62832 45.54110 49.91622
## 43:    Adelie Torgersen        5300 42.92879 40.66576 42.31072 46.76406
## 44: Chinstrap Torgersen        5300 46.59576 44.80400 46.49196 49.03906
## 45:    Gentoo Torgersen        5300 45.11384 42.95190 44.51289 49.27629
```

By default, all combinations of all variables are used. However, you can also look at the variables one by one, separately, like so:

```
pd_new <- orsf_pd_new(fit_regr,
                      expand_grid = FALSE,
                      pred_spec = pred_spec,
                      new_data = penguins_orsf_test)

pd_new
```

```
##        variable value     level     mean      lwr     medn      upr
##          <char> <num>    <char>    <num>    <num>    <num>    <num>
##  1:     species    NA    Adelie 41.90271 37.10417 41.51723 48.51478
##  2:     species    NA Chinstrap 47.11314 42.40419 46.96478 51.51392
##  3:     species    NA    Gentoo 44.37038 39.87306 43.89889 51.21635
##  4:      island    NA    Biscoe 44.21332 37.22711 45.27862 51.21635
##  5:      island    NA     Dream 44.43354 37.01471 45.57261 51.51392
##  6:      island    NA Torgersen 43.29539 37.01513 44.26924 49.84391
##  7: body_mass_g  3200      <NA> 42.84625 37.03978 43.95991 49.19173
##  8: body_mass_g  3550      <NA> 43.53326 37.56730 44.43756 50.47092
##  9: body_mass_g  3975      <NA> 44.30431 38.31567 45.22089 51.50683
## 10: body_mass_g  4700      <NA> 45.22559 39.88199 46.34680 51.18955
## 11: body_mass_g  5300      <NA> 45.91412 40.84742 46.95327 51.48851
```

And you can also bypass all the bells and whistles by using your own `data.frame` for a `pred_spec`. (Just make sure you request values that exist in the training data.)

```
custom_pred_spec <- data.frame(species = 'Adelie',
                               island = 'Biscoe')
```

```
pd_new <- orsf_pd_new(fit_regr,
                      pred_spec = custom_pred_spec,
                      new_data = penguins_orsf_test)

pd_new
```

```
##    species island     mean      lwr     medn      upr
##     <fctr> <fctr>    <num>    <num>    <num>    <num>
## 1:  Adelie Biscoe 41.98024 37.22711 41.65252 48.51478
```

Survival:
Begin by fitting an oblique survival random forest:

```
set.seed(329)

index_train <- sample(nrow(pbc_orsf), 150)

pbc_orsf_train <- pbc_orsf[index_train, ]
pbc_orsf_test <- pbc_orsf[-index_train, ]

fit_surv <- orsf(data = pbc_orsf_train,
                 formula = Surv(time, status) ~ . - id,
                 oobag_pred_horizon = 365.25 * 5)
```

Compute partial dependence using in-bag data for `bili = c(1,2,3,4,5)`:

```
pd_train <- orsf_pd_inb(fit_surv, pred_spec = list(bili = 1:5))
pd_train
```

```
##    pred_horizon  bili      mean        lwr      medn       upr
##           <num> <num>     <num>      <num>     <num>     <num>
## 1:      1826.25     1 0.2566200 0.02234786 0.1334170 0.8918909
## 2:      1826.25     2 0.3121392 0.06853733 0.1896849 0.9204338
## 3:      1826.25     3 0.3703242 0.11409793 0.2578505 0.9416791
## 4:      1826.25     4 0.4240692 0.15645214 0.3331057 0.9591581
## 5:      1826.25     5 0.4663670 0.20123406 0.3841700 0.9655296
```

If you don't have specific values of a variable in mind, let `pred_spec_auto` pick for you:

```
pd_train <- orsf_pd_inb(fit_surv, pred_spec_auto(bili))
pd_train
```

```
##    pred_horizon  bili      mean        lwr      medn       upr
##           <num> <num>     <num>      <num>     <num>     <num>
## 1:      1826.25  0.55 0.2481444 0.02035041 0.1242215 0.8801444
## 2:      1826.25  0.70 0.2502831 0.02045039 0.1271039 0.8836536
## 3:      1826.25  1.50 0.2797763 0.03964900 0.1601715 0.9041584
## 4:      1826.25  3.50 0.3959349 0.13431288 0.2920400 0.9501230
## 5:      1826.25  7.25 0.5351935 0.28064629 0.4652185 0.9783000
```

Specify `pred_horizon` to get partial dependence at each value:

```
pd_train <- orsf_pd_inb(fit_surv, pred_spec_auto(bili),
                        pred_horizon = seq(500, 3000, by = 500))
pd_train
```

```
##     pred_horizon  bili      mean         lwr       medn       upr
##            <num> <num>     <num>       <num>      <num>     <num>
##  1:          500  0.55 0.0617199 0.000443399 0.00865419 0.5907104
##  2:         1000  0.55 0.1418501 0.005793742 0.05572853 0.7360749
##  3:         1500  0.55 0.2082505 0.013609478 0.09174558 0.8556319
##  4:         2000  0.55 0.2679017 0.023047689 0.14574169 0.8910549
##  5:         2500  0.55 0.3179617 0.063797305 0.20254500 0.9017710
## ---
## 26:         1000  7.25 0.3264627 0.135343689 0.25956791 0.8884333
## 27:         1500  7.25 0.4641265 0.218208755 0.38787435 0.9702903
## 28:         2000  7.25 0.5511761 0.293367409 0.48427730 0.9812413
## 29:         2500  7.25 0.6200238 0.371965247 0.56954399 0.9845058
## 30:         3000  7.25 0.6803482 0.425128031 0.64642318 0.9888637
```

vector-valued `pred_horizon` input comes with minimal extra computational cost. Use a fine grid of time values and assess whether predictors have time-varying effects. (see partial dependence vignette for example)

References

1. Hooker, Giles, Mentch, Lucas, Zhou, Siyu (2021). "Unrestricted permutation forces extrapolation: variable importance requires at least one more model, or there is no free variable importance." *Statistics and Computing*, *31*, 1-16.

`orsf_scale_cph` *Scale input data*

Description

These functions are exported so that users may access internal routines that are used to scale inputs when orsf_control_cph is used.

Usage

```
orsf_scale_cph(x_mat, w_vec = NULL)

orsf_unscale_cph(x_mat)
```

Arguments

`x_mat`	(*numeric matrix*) a matrix with values to be scaled or unscaled. Note that `orsf_unscale_cph` will only accept `x_mat` inputs that have an attribute containing transform values, which are added automatically by `orsf_scale_cph`.
`w_vec`	(*numeric vector*) an optional vector of weights. If no weights are supplied (the default), all observations will be equally weighted. If supplied, `w_vec` must have length equal to `nrow(x_mat)`.

Details

The data are transformed by first subtracting the mean and then multiplying by the scale. An inverse transform can be completed using `orsf_unscale_cph` or by dividing each column by the corresponding scale and then adding the mean.

The values of means and scales are stored in an attribute of the output returned by `orsf_scale_cph` (see examples)

Value

the scaled or unscaled `x_mat`.

Examples

```
x_mat <- as.matrix(pbc_orsf[, c('bili', 'age', 'protime')])

head(x_mat)

x_scaled <- orsf_scale_cph(x_mat)

head(x_scaled)

attributes(x_scaled) # note the transforms attribute

x_unscaled <- orsf_unscale_cph(x_scaled)

head(x_unscaled)

# numeric difference in x_mat and x_unscaled should be practically 0
max(abs(x_mat - x_unscaled))
```

`orsf_summarize_uni` *Univariate summary*

Description

Summarize the univariate information from an ORSF object

Usage

```
orsf_summarize_uni(
  object,
  n_variables = NULL,
  pred_horizon = NULL,
  pred_type = NULL,
  importance = NULL,
  class = NULL,
  verbose_progress = FALSE,
  ...
)
```

Arguments

`object`	(*ObliqueForest*) a trained oblique random forest object (see orsf).
`n_variables`	(*integer*) how many variables should be summarized? Setting this input to a lower number will reduce computation time.
`pred_horizon`	(*double*) Only relevent for survival forests. A value or vector indicating the time(s) that predictions will be calibrated to. E.g., if you were predicting risk of incident heart failure within the next 10 years, then `pred_horizon = 10`. `pred_horizon` can be `NULL` if `pred_type` is `'mort'`, since mortality predictions are aggregated over all event times
`pred_type`	(*character*) the type of predictions to compute. Valid Valid options for survival are: • ’risk’ : probability of having an event at or before `pred_horizon`. • ’surv’ : 1 - risk. • ’chf’: cumulative hazard function • ’mort’: mortality prediction • ’time’: survival time prediction For classification: • ’prob’: probability for each class For regression: • ’mean’: predicted mean, i.e., the expected value
`importance`	(*character*) Indicate method for variable importance: • ’none’: no variable importance is computed. • ’anova’: compute analysis of variance (ANOVA) importance • ’negate’: compute negation importance • ’permute’: compute permutation importance
`class`	(*character*) only relevant for classification forests. If `NULL` (the default), summary statistics are returned for all classes in the outcome, and printed summaries will show the last class in the class levels. To specify a single class to summarize, indicate the name of the class with `class`. E.g., if the categorical outcome has class levels A, B, and C, then using `class = "A"` will restrict output to class A. For details on these methods, see orsf_vi.
`verbose_progress`	(*logical*) if `TRUE`, progress will be printed to console. If `FALSE` (the default), nothing will be printed.
`...`	Further arguments passed to or from other methods (not currently used).

Details

If `pred_horizon` is left unspecified, the median value of the time-to-event variable in `object`’s training data will be used. It is recommended to always specify your own prediction horizon, as the median time may not be an especially meaningful horizon to compute predicted risk values at.

If `object` already has variable importance values, you can safely bypass the computation of variable importance in this function by setting importance = ’none’.

Value

an object of class 'orsf_summary', which includes data on

- importance of individual predictors.
- expected values of predictions at specific values of predictors.

See Also

as.data.table.orsf_summary_uni

Examples

```
object <- orsf(pbc_orsf, Surv(time, status) ~ . - id, n_tree = 25)

# since anova importance was used to make object, it is also
# used for ranking variables in the summary, unless we specify
# a different type of importance

orsf_summarize_uni(object, n_variables = 2)

# if we want to summarize object according to variables
# ranked by negation importance, we can compute negation
# importance within orsf_summarize_uni() as follows:

orsf_summarize_uni(object, n_variables = 2, importance = 'negate')

# for multi-category fits, you can specify which class
# you want to summarize:

object =  orsf(species ~ ., data = penguins_orsf, n_tree = 25)

orsf_summarize_uni(object, class = "Adelie", n_variables = 1)
```

`orsf_time_to_train` *Estimate training time*

Description

Estimate training time

Usage

```
orsf_time_to_train(object, n_tree_subset = NULL)
```

Arguments

`object`	an untrained `aorsf` object
`n_tree_subset`	(*integer*) how many trees should be fit in order to estimate the time needed to train `object`. The default value is 10% of the trees specified in `object`. I.e., if `object` has `n_tree` of 500, then the default value `n_tree_subset` is 50.

Value

a difftime object.

Examples

```
# specify but do not train the model by setting no_fit = TRUE.
object <- orsf(pbc_orsf, Surv(time, status) ~ . - id,
               n_tree = 10, no_fit = TRUE)

# approximate the time it will take to grow 10 trees
time_estimated <- orsf_time_to_train(object, n_tree_subset=1)

print(time_estimated)

# let's see how close the approximation was
time_true_start <- Sys.time()
orsf_train(object)
time_true_stop <- Sys.time()

time_true <- time_true_stop - time_true_start

print(time_true)

# error
abs(time_true - time_estimated)
```

`orsf_update` *Update Forest Parameters*

Description

Update Forest Parameters

Usage

```
orsf_update(object, ..., modify_in_place = FALSE, no_fit = NULL)
```

Arguments

`object`	(*ObliqueForest*) an oblique random forest object (see orsf).
`...`	arguments to plug into orsf that will be used to define the update. These arguments include: • `data` • `formula` • `control` • `weights` • `n_tree` • `n_split` • `n_retry` • `n_thread` • `mtry` • `sample_with_replacement` • `sample_fraction` • `leaf_min_events` • `leaf_min_obs` • `split_rule` • `split_min_events` • `split_min_obs` • `split_min_stat` • `pred_type` • `oobag_pred_horizon` • `oobag_eval_every` • `oobag_fun` • `importance` • `importance_max_pvalue` • `group_factors` • `tree_seeds` • `na_action` • `verbose_progress` Note that you can update `control`, but you cannot change the type of forest. For example, you can't go from classification to regression with `orsf_update`.
`modify_in_place`	(*logical*) if `TRUE`, `object` will be modified by the inputs specified in `...`. Be cautious, as modification in place will overwrite existing data. If `FALSE` (the default), `object` will be copied and then the modifications will be applied to the copy, leaving the original `object` unmodified.
`no_fit`	(*logical*) if `TRUE`, model fitting steps are defined and saved, but training is not initiated. The object returned can be directly submitted to `orsf_train()` so long as `attach_data` is `TRUE`.

Details

There are several dynamic inputs in `orsf` with default values of `NULL`. Specifically, these inputs are `control`, `weights`, `mtry`, `split_rule`, `split_min_stat`, `pred_type`, `pred_horizon`, `oobag_eval_function`, `tree_seeds`, and `oobag_eval_every`. If no explicit value is given for these inputs in the call, they *will be re-formed*. For example, if an initial forest includes 17 predictors, the default `mtry` is the smallest integer that is greater than or equal to the square root of 17, i.e., 5. Then, if you make an updated forest with 1 less predictor and you do not explicitly say `mtry = 5`, then `mtry` will be re-initialized in the update based on the available 16 predictors, and the resulting value of `mtry` will be 4. This is done to avoid many potential errors that would occur if the dynamic outputs were not re-initialized.

Value

an `ObliqueForest` object.

Examples

```
## Not run:
# initial fit has mtry of 5
fit <- orsf(pbc_orsf, time + status ~ . -id)

# note that mtry is now 4 (see details)
fit_new <- orsf_update(fit, formula = . ~ . - edema, n_tree = 100)

# prevent dynamic updates by specifying inputs you want to freeze.
fit_newer <- orsf_update(fit_new, mtry = 2)

## End(Not run)
```

`orsf_vi` *Variable Importance*

Description

Estimate the importance of individual predictor variables using oblique random forests.

Usage

```
orsf_vi(
  object,
  group_factors = TRUE,
  importance = NULL,
  oobag_fun = NULL,
  n_thread = NULL,
  verbose_progress = NULL,
  ...
```

```
)

orsf_vi_negate(
  object,
  group_factors = TRUE,
  oobag_fun = NULL,
  n_thread = NULL,
  verbose_progress = NULL,
  ...
)

orsf_vi_permute(
  object,
  group_factors = TRUE,
  oobag_fun = NULL,
  n_thread = NULL,
  verbose_progress = NULL,
  ...
)

orsf_vi_anova(object, group_factors = TRUE, verbose_progress = NULL, ...)
```

Arguments

`object`	(*ObliqueForest*) a trained oblique random forest object (see orsf).
`group_factors`	(*logical*) if `TRUE`, the importance of factor variables will be reported overall by aggregating the importance of individual levels of the factor. If `FALSE`, the importance of individual factor levels will be returned.
`importance`	(*character*) Indicate method for variable importance: • 'anova': compute analysis of variance (ANOVA) importance • 'negate': compute negation importance • 'permute': compute permutation importance
`oobag_fun`	(*function*) to be used for evaluating out-of-bag prediction accuracy after negating coefficients (if importance = 'negate') or permuting the values of a predictor (if importance = 'permute') • When `oobag_fun = NULL` (the default), the evaluation statistic is selected based on tree type • survival: Harrell's C-statistic (1982) • classification: Area underneath the ROC curve (AUC-ROC) • regression: Traditional prediction R-squared • if you use your own `oobag_fun` note the following: – `oobag_fun` should have three inputs: `y_mat`, `w_vec`, and `s_vec` – For survival trees, `y_mat` should be a two column matrix with first column named 'time' and second named 'status'. For classification trees, `y_mat` should be a matrix with number of columns = number of distinct classes in the outcome. For regression, `y_mat` should be a matrix with one column.

	- `s_vec` is a numeric vector containing predictions - `oobag_fun` should return a numeric output of length 1 - the same `oobag_fun` should have been used when you created `object` so that the initial value of out-of-bag prediction accuracy is consistent with the values that will be computed while variable importance is estimated. For more details, see the out-of-bag vignette.
`n_thread`	(*integer*) number of threads to use while computing predictions. Default is 0, which allows a suitable number of threads to be used based on availability.
`verbose_progress`	(*logical*) if `TRUE`, progress messages are printed in the console. If `FALSE` (the default), nothing is printed.
`...`	Further arguments passed to or from other methods (not currently used).

Details

When an `ObliqueForest` object is grown with importance = 'anova', 'negate', or 'permute', the output will have a vector of importance values based on the requested type of importance. However, `orsf_vi()` can be used to compute variable importance after growing a forest or to compute a different type of importance.

`orsf_vi()` is a general purpose function to extract or compute variable importance estimates from an `ObliqueForest` object (see orsf). `orsf_vi_negate()`, `orsf_vi_permute()`, and `orsf_vi_anova()` are wrappers for `orsf_vi()`. The way these functions work depends on whether the `object` they are given already has variable importance estimates in it or not (see examples).

Value

`orsf_vi` functions return a named numeric vector.

- Names of the vector are the predictor variables used by `object`
- Values of the vector are the estimated importance of the given predictor.

The returned vector is sorted from highest to lowest value, with higher values indicating higher importance.

Variable importance methods

negation importance: Each variable is assessed separately by multiplying the variable's coefficients by -1 and then determining how much the model's performance changes. The worse the model's performance after negating coefficients for a given variable, the more important the variable. This technique is promising b/c it does not require permutation and it emphasizes variables with larger coefficients in linear combinations, but it is also relatively new and hasn't been studied as much as permutation importance. See Jaeger, (2023) for more details on this technique.

permutation importance: Each variable is assessed separately by randomly permuting the variable's values and then determining how much the model's performance changes. The worse the model's performance after permuting the values of a given variable, the more important the variable. This technique is flexible, intuitive, and frequently used. It also has several known limitations

analysis of variance (ANOVA) importance: A p-value is computed for each coefficient in each linear combination of variables in each decision tree. Importance for an individual predictor variable is the proportion of times a p-value for its coefficient is < 0.01. This technique is very efficient computationally, but may not be as effective as permutation or negation in terms of selecting signal over noise variables. See Menze, 2011 for more details on this technique.

Examples

ANOVA importance:

The default variable importance technique, ANOVA, is calculated while you fit an oblique random forest ensemble.

```
fit <- orsf(pbc_orsf, Surv(time, status) ~ . - id)

fit
```

```
## ---------- Oblique random survival forest
##
##      Linear combinations: Accelerated Cox regression
##           N observations: 276
##                 N events: 111
##                  N trees: 500
##       N predictors total: 17
##    N predictors per node: 5
##  Average leaves per tree: 21.022
## Min observations in leaf: 5
##       Min events in leaf: 1
##           OOB stat value: 0.84
##            OOB stat type: Harrell's C-index
##      Variable importance: anova
##
## -----------------------------------------
```

ANOVA is the default because it is fast, but it may not be as decisive as the permutation and negation techniques for variable selection.

Raw VI values:

the ‘raw’ variable importance values can be accessed from the fit object

```
fit$get_importance_raw()
```

```
##                   [,1]
## trt_placebo 0.06355042
## age         0.23259259
## sex_f       0.14700432
## ascites_1   0.46791708
## hepato_1    0.14349776
## spiders_1   0.17371938
## edema_0.5   0.17459191
## edema_1     0.51197605
```

```
## bili        0.40590758
## chol        0.17666667
## albumin     0.25972156
## copper      0.28840580
## alk.phos    0.10614251
## ast         0.18327491
## trig        0.12815626
## platelet    0.09265648
## protime     0.22656250
## stage       0.20264766
```

these are 'raw' because values for factors have not been aggregated into a single value. Currently there is one value for k-1 levels of a k level factor. For example, you can see edema_1 and edema_0.5 in the importance values above because edema is a factor variable with levels of 0, 0.5, and 1.

Collapse VI across factor levels:

To get aggregated values across all levels of each factor,

- access the `importance` element from the `orsf` fit:

  ```
  # this assumes you used group_factors = TRUE in orsf()
  fit$importance
  ```

  ```
  ##    ascites       bili      edema     copper    albumin        age    protime
  ## 0.46791708 0.40590758 0.31115216 0.28840580 0.25972156 0.23259259 0.22656250
  ##      stage        ast       chol    spiders        sex     hepato       trig
  ## 0.20264766 0.18327491 0.17666667 0.17371938 0.14700432 0.14349776 0.12815626
  ##   alk.phos   platelet        trt
  ## 0.10614251 0.09265648 0.06355042
  ```

- use `orsf_vi()` with group_factors set to `TRUE` (the default)

  ```
  orsf_vi(fit)
  ```

  ```
  ##    ascites       bili      edema     copper    albumin        age    protime
  ## 0.46791708 0.40590758 0.31115216 0.28840580 0.25972156 0.23259259 0.22656250
  ##      stage        ast       chol    spiders        sex     hepato       trig
  ## 0.20264766 0.18327491 0.17666667 0.17371938 0.14700432 0.14349776 0.12815626
  ##   alk.phos   platelet        trt
  ## 0.10614251 0.09265648 0.06355042
  ```

Note that you can make the default returned importance values ungrouped by setting `group_factors` to `FALSE` in the `orsf_vi` functions or the `orsf` function.

Add VI to an oblique random forest:

You can fit an oblique random forest without VI, then add VI later

```
fit_no_vi <- orsf(pbc_orsf,
                  Surv(time, status) ~ . - id,
                  importance = 'none')

# Note: you can't call orsf_vi_anova() on fit_no_vi because anova
# VI can only be computed while the forest is being grown.

orsf_vi_negate(fit_no_vi)
```

```
##        bili      copper         sex     protime         age       stage
## 0.130439814 0.051880867 0.038308025 0.025115249 0.023826061 0.020354822
##     albumin     ascites        chol         ast     spiders      hepato
## 0.019997729 0.015918292 0.013320469 0.010086726 0.007409116 0.007326714
##       edema         trt    alk.phos        trig    platelet
## 0.006844435 0.003214544 0.002517057 0.002469545 0.001056829
```

```
orsf_vi_permute(fit_no_vi)
```

```
##          bili        copper           age       ascites       protime
##  0.0592069141  0.0237362075  0.0136479213  0.0130805894  0.0123091354
##         stage       albumin          chol        hepato           ast
##  0.0117177661  0.0106414724  0.0064501213  0.0058813969  0.0057753740
##         edema       spiders           sex          trig      platelet
##  0.0052171180  0.0048427005  0.0023386947  0.0017883700  0.0013533691
##      alk.phos           trt
##  0.0006492029 -0.0009921507
```

Oblique random forest and VI all at once:

fit an oblique random forest and compute vi at the same time

```
fit_permute_vi <- orsf(pbc_orsf,
                       Surv(time, status) ~ . - id,
                       importance = 'permute')

# get the vi instantly (i.e., it doesn't need to be computed again)
orsf_vi_permute(fit_permute_vi)
```

```
##          bili        copper       ascites       protime       albumin
##  0.0571305446  0.0243657146  0.0138318057  0.0133401675  0.0130746154
##           age         stage          chol           ast       spiders
##  0.0123610374  0.0102963203  0.0077895394  0.0075250059  0.0048628813
##         edema        hepato           sex      platelet          trig
##  0.0046003168  0.0039818730  0.0016891584  0.0012767063  0.0007324402
##      alk.phos           trt
##  0.0005128897 -0.0014443967
```

You can still get negation VI from this fit, but it needs to be computed

```
orsf_vi_negate(fit_permute_vi)
```

```
##        bili      copper         sex     protime       stage         age
## 0.123331760 0.052544318 0.037291358 0.024977898 0.023239189 0.021934511
##     albumin     ascites        chol         ast     spiders       edema
## 0.020586632 0.014229536 0.014053040 0.012227048 0.007643156 0.006832766
##      hepato         trt    alk.phos        trig    platelet
## 0.006301693 0.004348705 0.002371797 0.002309396 0.001347035
```

Custom functions for VI:

The default prediction accuracy functions work well most of the time:

```
fit_standard <- orsf(penguins_orsf, bill_length_mm ~ ., tree_seeds = 1)

# Default method for prediction accuracy with VI is R-squared
orsf_vi_permute(fit_standard)
```

```
##           species flipper_length_mm       body_mass_g     bill_depth_mm
##      0.3725898166      0.3261834607      0.2225730676      0.1026569498
##            island               sex              year
##      0.0876071687      0.0844807334      0.0006978493
```

But sometimes you want to do something specific and the defaults just won't work. For these cases, you can compute VI with any function you'd like to measure prediction accuracy by supplying a valid function to the `oobag_fun` input. For example, we use mean absolute error below. Higher values are considered good when `aorsf` computes prediction accuracy, so we make our function return a pseudo R-squared based on mean absolute error:

```
rsq_mae <- function(y_mat, w_vec, s_vec){

 mae_standard <- mean(abs((y_mat - mean(y_mat)) * w_vec))
 mae_fit <- mean(abs((y_mat - s_vec) * w_vec))

 1 - mae_fit / mae_standard

}

fit_custom <- orsf_update(fit_standard, oobag_fun = rsq_mae)

# not much changes, but the difference between variables shrinks
# and the ordering of sex and island has swapped
orsf_vi_permute(fit_custom)
```

```
##           species flipper_length_mm       body_mass_g     bill_depth_mm
##       0.206951751       0.193248912       0.140899603       0.076759148
##               sex            island              year
##       0.073042331       0.050851073       0.003633365
```

References

1. Harrell, E F, Califf, M R, Pryor, B D, Lee, L K, Rosati, A R (1982). "Evaluating the yield of medical tests." *Jama*, *247*(18), 2543-2546.
2. Breiman, Leo (2001). "Random Forests." *Machine Learning*, *45*(1), 5-32. ISSN 1573-0565.
3. Menze, H B, Kelm, Michael B, Splitthoff, N D, Koethe, Ullrich, Hamprecht, A F (2011). "On oblique random forests." In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2011, Athens, Greece, September 5-9, 2011, Proceedings, Part II 22*, 453-469. Springer.
4. Jaeger BC, Welden S, Lenoir K, Speiser JL, Segar MW, Pandey A, Pajewski NM (2023). "Accelerated and interpretable oblique random survival forests." *Journal of Computational and Graphical Statistics*, 1-16.

`orsf_vint` *Variable Interactions*

Description

Use the variable interaction score described in Greenwell et al (2018). As this method can be computationally demanding, using `n_thread=0` can substantially reduce time needed to compute scores.

Usage

```
orsf_vint(
  object,
  predictors = NULL,
  n_thread = NULL,
  verbose_progress = NULL,
  sep = ".."
)
```

Arguments

`object`	(*ObliqueForest*) a trained oblique random forest object (see orsf)
`predictors`	(*character*) a vector of length 2 or more with names of predictors used by `object`. All pairwise interactions between the predictors will be scored. If `NULL` (the default), all predictors are used.
`n_thread`	(*integer*) number of threads to use while growing trees, computing predictions, and computing importance. Default is 0, which allows a suitable number of threads to be used based on availability.
`verbose_progress`	(*logical*) if `TRUE`, progress messages are printed in the console. If `FALSE` (the default), nothing is printed.
`sep`	(*character*) how to separate the names of two predictors. The default value of `".."` returns names as `name1..name2`

Details

The number of possible interactions grows exponentially based on the number of predictors. Some caution is warranted when using large predictor sets and it is recommended that you supply a specific vector of predictor names to assess rather than a global search. A good strategy is to use `n_tree = 5` to search all predictors, then pick the top 10 interactions, get the unique predictors from them, and re-run on just those predictors with more trees.

Value

a data.table with variable interaction scores and partial dependence values.

References

1. Greenwell, M B, Boehmke, C B, McCarthy, J A (2018). "A simple and effective model-based variable importance measure." *arXiv preprint arXiv:1805.04755*.

Examples

```
set.seed(329)

data <- data.frame(
 x1 = rnorm(500),
 x2 = rnorm(500),
 x3 = rnorm(500)
)

data$y = with(data, expr = x1 + x2 + x3 + 1/2*x1 * x2 + x2 * x3 + rnorm(500))

forest <- orsf(data, y ~ ., n_tree = 5)

orsf_vint(forest)
```

`orsf_vs` *Variable selection*

Description

Variable selection

Usage

```
orsf_vs(object, n_predictor_min = 3, verbose_progress = NULL)
```

Arguments

`object`	(*ObliqueForest*) a trained oblique random forest object (see orsf).
`n_predictor_min`	(*integer*) the minimum number of predictors allowed
`verbose_progress`	(*logical*) not implemented yet. Should progress be printed to the console?

Details

The difference between `variables_included` and `predictors_included` is referent coding. The `variable` would be the name of a factor variable in the training data, while the `predictor` would be the name of that same factor with the levels of the factor appended. For example, if the variable is `diabetes` with `levels = c("no", "yes")`, then the variable name is `diabetes` and the predictor name is `diabetes_yes`.

`tree_seeds` should be specified in `object` so that each successive run of `orsf` will be evaluated in the same out-of-bag samples as the initial run.

Value

a data.table with four columns:

- *n_predictors*: the number of predictors used
- *stat_value*: the out-of-bag statistic
- *variables_included*: the names of the variables included
- *predictors_included*: the names of the predictors included
- *predictor_dropped*: the predictor selected to be dropped

Examples

```
object <- orsf(formula = time + status ~ .,
               data = pbc_orsf,
               n_tree = 25,
               importance = 'anova')

orsf_vs(object, n_predictor_min = 15)
```

`pbc_orsf` *Mayo Clinic Primary Biliary Cholangitis Data*

Description

These data are a light modification of the survival::pbc data. The modifications are:

Usage

```
pbc_orsf
```

Format

A data frame with 276 rows and 20 variables:

id case number

time number of days between registration and the earlier of death, transplantion, or study analysis in July, 1986

status status at endpoint, 0 for censored or transplant, 1 for dead

trt randomized treatment group: D-penicillmain or placebo

age in years

sex m/f

ascites presence of ascites

hepato presence of hepatomegaly or enlarged liver

spiders blood vessel malformations in the skin

edema 0 no edema, 0.5 untreated or successfully treated, 1 edema despite diuretic therapy

bili serum bilirubin (mg/dl)

chol serum cholesterol (mg/dl)

albumin serum albumin (g/dl)

copper urine copper (ug/day)

alk.phos alkaline phosphotase (U/liter)

ast aspartate aminotransferase, once called SGOT (U/ml)

trig triglycerides (mg/dl)

platelet platelet count

protime standardized blood clotting time

stage histologic stage of disease (needs biopsy)

Details

1. removed rows with missing data
2. converted `status` into 0 for censor or transplant, 1 for dead
3. converted `stage` into an ordered factor.
4. converted `trt`, `ascites`, `hepato`, `spiders`, and `edema` into factors.

Source

T Therneau and P Grambsch (2000), Modeling Survival Data: Extending the Cox Model, Springer-Verlag, New York. ISBN: 0-387-98784-3.

`penguins_orsf` *Size measurements for adult foraging penguins near Palmer Station, Antarctica*

Description

These data are copied and lightly modified from the `penguins` data in the palmerpenguins R package. The only modification is removal of rows with missing data. The data include measurements for penguin species, island in Palmer Archipelago, size (flipper length, body mass, bill dimensions), and sex.

Usage

```
penguins_orsf
```

Format

A tibble with 333 rows and 8 variables:

species	a factor denoting penguin species (Adélie, Chinstrap and Gentoo)
island	a factor denoting island in Palmer Archipelago, Antarctica (Biscoe, Dream or Torgersen)
bill_length_mm	a number denoting bill length (millimeters)
bill_depth_mm	a number denoting bill depth (millimeters)
flipper_length_mm	an integer denoting flipper length (millimeters)
body_mass_g	an integer denoting body mass (grams)
sex	a factor denoting penguin sex (female, male)
year	an integer denoting the study year (2007, 2008, or 2009)

Source

Adélie penguins: Palmer Station Antarctica LTER and K. Gorman. 2020. Structural size measurements and isotopic signatures of foraging among adult male and female Adélie penguins (Pygoscelis adeliae) nesting along the Palmer Archipelago near Palmer Station, 2007-2009 ver 5. Environmental Data Initiative. doi:10.6073/pasta/98b16d7d563f265cb52372c8ca99e60f

Gentoo penguins: Palmer Station Antarctica LTER and K. Gorman. 2020. Structural size measurements and isotopic signatures of foraging among adult male and female Gentoo penguin (Pygoscelis papua) nesting along the Palmer Archipelago near Palmer Station, 2007-2009 ver 5. Environmental Data Initiative. doi:10.6073/pasta/7fca67fb28d56ee2ffa3d9370ebda689

Chinstrap penguins: Palmer Station Antarctica LTER and K. Gorman. 2020. Structural size measurements and isotopic signatures of foraging among adult male and female Chinstrap penguin (Pygoscelis antarcticus) nesting along the Palmer Archipelago near Palmer Station, 2007-2009 ver 6. Environmental Data Initiative. doi:10.6073/pasta/c14dfcfada8ea13a17536e73eb6fbe9e

Originally published in: Gorman KB, Williams TD, Fraser WR (2014) Ecological Sexual Dimorphism and Environmental Variability within a Community of Antarctic Penguins (Genus Pygoscelis). PLoS ONE 9(3): e90081. doi:10.1371/journal.pone.0090081

`predict.ObliqueForest` *Prediction for ObliqueForest Objects*

Description

Compute predicted values from an oblique random forest. Predictions may be returned in aggregate (i.e., averaging over all the trees) or tree-specific.

Usage

```
## S3 method for class 'ObliqueForest'
predict(
  object,
  new_data = NULL,
  pred_type = NULL,
  pred_horizon = NULL,
  pred_aggregate = TRUE,
  pred_simplify = FALSE,
  oobag = FALSE,
  na_action = NULL,
  boundary_checks = TRUE,
  n_thread = NULL,
  verbose_progress = NULL,
  ...
)
```

Arguments

`object`	(*ObliqueForest*) a trained oblique random forest object (see orsf).
`new_data`	a data.frame, tibble, or data.table to compute predictions in.
`pred_type`	(*character*) the type of predictions to compute. Valid options for survival are: • 'risk' : probability of having an event at or before `pred_horizon`. • 'surv' : 1 - risk. • 'chf': cumulative hazard function • 'mort': mortality prediction • 'time': survival time prediction For classification: • 'prob': probability for each class • 'class': predicted class For regression: • 'mean': predicted mean, i.e., the expected value
`pred_horizon`	(*double*) Only relevent for survival forests. A value or vector indicating the time(s) that predictions will be calibrated to. E.g., if you were predicting risk of incident heart failure within the next 10 years, then `pred_horizon = 10`. `pred_horizon` can be `NULL` if `pred_type` is `'mort'`, since mortality predictions are aggregated over all event times
`pred_aggregate`	(*logical*) If `TRUE` (the default), predictions will be aggregated over all trees by taking the mean. If `FALSE`, the returned output will contain one row per observation and one column for each tree. If the length of `pred_horizon` is two or more and `pred_aggregate` is `FALSE`, then the result will be a list of such matrices, with the i'th item in the list corresponding to the i'th value of `pred_horizon`.
`pred_simplify`	(*logical*) If `FALSE` (the default), predictions will always be returned in a numeric matrix or a list of numeric matrices. If `TRUE`, predictions may be simplified to a vector, e.g., if `pred_type` is `'mort'` for survival or `'class'` for classification, or an array of matrices if `length(pred_horizon) > 1`.

`oobag`	(*logical*) If `FALSE` (the default), predictions will be computed using all trees for each observation. If `TRUE`, then out-of-bag predictions will be computed. This input parameter should only be set to `TRUE` if `new_data` is `NULL`.
`na_action`	(*character*) what should happen when `new_data` contains missing values (i.e., `NA` values). Valid options are: • 'fail' : an error is thrown if `new_data` contains `NA` values • 'pass' : the output will have `NA` in all rows where `new_data` has 1 or more `NA` value for the predictors used by `object` • 'omit' : rows in `new_data` with incomplete data will be dropped • 'impute_meanmode' : missing values for continuous and categorical variables in `new_data` will be imputed using the mean and mode, respectively. To clarify, the mean and mode used to impute missing values are from the training data of `object`, not from `new_data`.
`boundary_checks`	(*logical*) if `TRUE`, `pred_horizon` will be checked to make sure the requested values are less than the maximum observed time in `object`'s training data. If `FALSE`, these checks are skipped.
`n_thread`	(*integer*) number of threads to use while computing predictions. Default is 0, which allows a suitable number of threads to be used based on availability.
`verbose_progress`	(*logical*) if `TRUE`, progress messages are printed in the console. If `FALSE` (the default), nothing is printed.
`...`	Further arguments passed to or from other methods (not currently used).

Details

`new_data` must have the same columns with equivalent types as the data used to train `object`. Also, factors in `new_data` must not have levels that were not in the data used to train `object`.

`pred_horizon` values should not exceed the maximum follow-up time in `object`'s training data, but if you truly want to do this, set `boundary_checks = FALSE` and you can use a `pred_horizon` as large as you want. Note that predictions beyond the maximum follow-up time in the `object`'s training data are equal to predictions at the maximum follow-up time, because `aorsf` does not estimate survival beyond its maximum observed time.

If unspecified, `pred_horizon` may be automatically specified as the value used for `oobag_pred_horizon` when `object` was created (see orsf).

Value

a `matrix` of predictions. Column `j` of the matrix corresponds to value `j` in `pred_horizon`. Row `i` of the matrix corresponds to row `i` in `new_data`.

Examples

```
library(aorsf)
```

Classification:

```
set.seed(329)

index_train <- sample(nrow(penguins_orsf), 150)

penguins_orsf_train <- penguins_orsf[index_train, ]
penguins_orsf_test <- penguins_orsf[-index_train, ]

fit_clsf <- orsf(data = penguins_orsf_train,
                 formula = species ~ .)
```

Predict probability for each class or the predicted class:

```
# predicted probabilities, the default
predict(fit_clsf,
        new_data = penguins_orsf_test[1:5, ],
        pred_type = 'prob')
```

```
##         Adelie  Chinstrap      Gentoo
## [1,] 0.9405310 0.04121955 0.018249405
## [2,] 0.9628988 0.03455909 0.002542096
## [3,] 0.9032074 0.08510528 0.011687309
## [4,] 0.9300133 0.05209040 0.017896329
## [5,] 0.7965703 0.16243492 0.040994821
```

```
# predicted class (as a matrix by default)
predict(fit_clsf,
        new_data = penguins_orsf_test[1:5, ],
        pred_type = 'class')
```

```
##      [,1]
## [1,]    1
## [2,]    1
## [3,]    1
## [4,]    1
## [5,]    1
```

```
# predicted class (as a factor if you use simplify)
predict(fit_clsf,
        new_data = penguins_orsf_test[1:5, ],
        pred_type = 'class',
        pred_simplify = TRUE)
```

```
## [1] Adelie Adelie Adelie Adelie Adelie
## Levels: Adelie Chinstrap Gentoo
```

Regression:

```
set.seed(329)

index_train <- sample(nrow(penguins_orsf), 150)

penguins_orsf_train <- penguins_orsf[index_train, ]
```

```
penguins_orsf_test <- penguins_orsf[-index_train, ]

fit_regr <- orsf(data = penguins_orsf_train,
                 formula = bill_length_mm ~ .)
```

Predict the mean value of the outcome:

```
predict(fit_regr,
        new_data = penguins_orsf_test[1:5, ],
        pred_type = 'mean')
```

```
##          [,1]
## [1,] 37.74136
## [2,] 37.42367
## [3,] 37.04598
## [4,] 39.89602
## [5,] 39.14848
```

Survival:

Begin by fitting an oblique survival random forest:

```
set.seed(329)

index_train <- sample(nrow(pbc_orsf), 150)

pbc_orsf_train <- pbc_orsf[index_train, ]
pbc_orsf_test <- pbc_orsf[-index_train, ]

fit_surv <- orsf(data = pbc_orsf_train,
                 formula = Surv(time, status) ~ . - id,
                 oobag_pred_horizon = 365.25 * 5)
```

Predict risk, survival, or cumulative hazard at one or several times:

```
# predicted risk, the default
predict(fit_surv,
        new_data = pbc_orsf_test[1:5, ],
        pred_type = 'risk',
        pred_horizon = c(500, 1000, 1500))
```

```
##             [,1]        [,2]       [,3]
## [1,] 0.013648562 0.058393393 0.11184029
## [2,] 0.003811413 0.026857586 0.04774151
## [3,] 0.030548361 0.100600301 0.14847107
## [4,] 0.040381075 0.169596943 0.27018952
## [5,] 0.001484698 0.006663576 0.01337655
```

```
# predicted survival, i.e., 1 - risk
predict(fit_surv,
        new_data = pbc_orsf_test[1:5, ],
        pred_type = 'surv',
        pred_horizon = c(500, 1000, 1500))
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.9863514 0.9416066 0.8881597
## [2,] 0.9961886 0.9731424 0.9522585
## [3,] 0.9694516 0.8993997 0.8515289
## [4,] 0.9596189 0.8304031 0.7298105
## [5,] 0.9985153 0.9933364 0.9866235
```

```
# predicted cumulative hazard function
# (expected number of events for person i at time j)
predict(fit_surv,
        new_data = pbc_orsf_test[1:5, ],
        pred_type = 'chf',
        pred_horizon = c(500, 1000, 1500))
```

```
##             [,1]        [,2]       [,3]
## [1,] 0.015395388 0.067815817 0.14942956
## [2,] 0.004022524 0.028740305 0.05424314
## [3,] 0.034832754 0.127687156 0.20899732
## [4,] 0.059978334 0.233048809 0.42562310
## [5,] 0.001651365 0.007173177 0.01393016
```

Predict mortality, defined as the number of events in the forest's population if all observations had characteristics like the current observation. This type of prediction does not require you to specify a prediction horizon

```
predict(fit_surv,
        new_data = pbc_orsf_test[1:5, ],
        pred_type = 'mort')
```

```
##           [,1]
## [1,] 23.405016
## [2,] 15.362916
## [3,] 26.180648
## [4,] 36.515629
## [5,]  5.856674
```

`pred_spec_auto` *Automatic variable values for dependence*

Description

For partial dependence and individual conditional expectations, this function allows a variable to be considered without having to specify what values to set the variable at. The values used are based on quantiles for continuous variables (10th, 25th, 50th, 75th, and 90th) and unique categories for categorical variables.

Usage

```
pred_spec_auto(...)
```

Arguments

`...`	names of the variables to use. These can be in quotes or not in quotes (see examples).

Details

This function should only be used in the context of `orsf_pd` or `orsf_ice` functions.

Value

a character vector with the names

Examples

```
fit <- orsf(penguins_orsf, species ~., n_tree = 5)

orsf_pd_oob(fit, pred_spec_auto(flipper_length_mm))
```

`print.ObliqueForest` *Inspect Forest Parameters*

Description

Printing an ORSF model tells you:

- Linear combinations: How were these identified?
- N observations: Number of rows in training data
- N events: Number of events in training data
- N trees: Number of trees in the forest
- N predictors total: Total number of columns in the predictor matrix
- N predictors per node: Number of variables used in linear combinations
- Average leaves per tree: A proxy for the depth of your trees
- Min observations in leaf: See `leaf_min_obs` in orsf
- Min events in leaf: See `leaf_min_events` in orsf
- OOB stat value: Out-of-bag error after fitting all trees
- OOB stat type: How was out-of-bag error computed?
- Variable importance: How was variable importance computed?

Usage

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

Arguments

`x`	(*ObliqueForest*) an oblique random survival forest (ORSF; see orsf).
`...`	Further arguments passed to or from other methods (not currently used).

Value

`x`, invisibly.

Examples

```
object <- orsf(pbc_orsf, Surv(time, status) ~ . - id, n_tree = 5)

print(object)
```

`print.orsf_summary_uni`

Print ORSF summary

Description

Print ORSF summary

Usage

```
## S3 method for class 'orsf_summary_uni'
print(x, n_variables = NULL, ...)
```

Arguments

`x`	an object of class 'orsf_summary'
`n_variables`	The number of variables to print
`...`	Further arguments passed to or from other methods (not currently used).

Value

invisibly, `x`

Examples

```
object <- orsf(pbc_orsf, Surv(time, status) ~ . - id, n_tree = 25)

smry <- orsf_summarize_uni(object, n_variables = 2)

print(smry)
```

Index