Package 'aorsf'

January 12, 2023

Title Accelerated Oblique Random Survival Forests

Version 0.0.7

Description Fit, interpret, and make predictions with oblique random survival forests. Oblique decision trees are notoriously slow compared to their axis based counterparts, but 'aorsf' runs as fast or faster than axis-based decision tree algorithms for right-censored time-to-event outcomes. Methods to accelerate and interpret the oblique random survival forest are described in Jaeger et al., (2022) <arXiv:2208.01129>.

License MIT + file LICENSE

Encoding UTF-8

LazyData true

RoxygenNote 7.2.2

LinkingTo Rcpp, RcppArmadillo

Imports Rcpp, data.table, utils, stats, collapse

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

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

Depends R (>= 3.6)

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

Config/testthat/edition 3

VignetteBuilder knitr

NeedsCompilation yes

Author Byron Jaeger [aut, cre] (<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 2023-01-12 11:50:02 UTC

R topics documented:

`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

```
library(data.table)

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

smry <- orsf_summarize_uni(object, n_variables = 3)

as.data.table(smry)
```

`orsf` *Oblique Random Survival Forest (ORSF)*

Description

Fit an oblique random survival forest

Usage

```
orsf(
  data,
  formula,
  control = orsf_control_fast(),
  weights = NULL,
  n_tree = 500,
  n_split = 5,
  n_retry = 3,
  mtry = NULL,
  leaf_min_events = 1,
  leaf_min_obs = 5,
  split_min_events = 5,
  split_min_obs = 10,
  split_min_stat = 3.841459,
  oobag_pred_type = "surv",
  oobag_pred_horizon = NULL,
  oobag_eval_every = n_tree,
  oobag_fun = NULL,
  importance = "anova",
  group_factors = TRUE,
  tree_seeds = NULL,
  attach_data = TRUE,
  no_fit = FALSE,
  na_action = "fail",
  verbose_progress = FALSE,
  ...
)
```

```
orsf_train(object)
```

Arguments

data	a data.frame, tibble, or data.table that contains the relevant variables.
formula	(*formula*) The response on the left hand side should include a time variable, followed by a status variable, and may be written inside a call to Surv (see examples). The terms on the right are names of predictor variables.
control	(*orsf_control*) An object returned from one of the `orsf_control` functions: • orsf_control_fast (the default) uses a single iteration of Newton Raphson scoring to identify a linear combination of predictors. • orsf_control_cph uses Newton Raphson scoring until a convergence criteria is met. • orsf_control_net uses `glmnet` to identify linear combinations of predictors, similar to Jaeger (2019). • orsf_control_custom allows the user to apply their own function to create linear combinations of predictors.
weights	(*numeric vector*) Optional. If given, this input should have length equal to `nrow(data)`. Values in `weights` are treated like replication weights, i.e., a value of 2 is the same thing as having 2 observations in `data`, each containing a copy of the corresponding person's data. *Use* `weights` *cautiously*, as `orsf` will count the number of observations and events prior to growing a node for a tree, so higher values in `weights` will lead to deeper trees.
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 can be split, 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.
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))`
leaf_min_events	(*integer*) 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_min_events	(*integer*) minimum number of events required in a node to consider splitting it. Default is `split_min_events = 5`
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. Default is 3.841459 for the log-rank test, which is roughly a p-value of 0.05

`oobag_pred_type`

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

- 'none' : don't compute out-of-bag predictions
- 'risk' : predict the probability of having an event at or before `oobag_pred_horizon`.
- 'surv' : 1 - risk.
- 'chf' : predict cumulative hazard function

Mortality ('mort')is not implemented for out of bag predictions yet, but it will be in a future update.

`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)`.

`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), Harrell's C-statistic (1982) is used to evaluate accuracy. if you use your own `oobag_fun` note the following:

- `oobag_fun` should have two inputs: `y_mat` and `s_vec`
- `y_mat` is a two column matrix with first column named 'time', second named 'status'
- `s_vec` is a numeric vector containing predicted survival probabilities.
- `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.

`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), no seeds are set during the training process.

attach_data (*logical*) if TRUE, a copy of the training data will be attached to the output. This is helpful 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. Note that is this option is selected and attach_data is TRUE, the data attached to the output will be the imputed version of data.

verbose_progress (*logical*) if TRUE, progress messages are printed in the console.

... 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

This function is based on and similar to the ORSF function in the obliqueRSF R package. The primary difference is that this function runs much faster. The speed increase is attributable to better management of memory (i.e., no unnecessary copies of inputs) and using a Newton Raphson scoring algorithm to identify linear combinations of inputs rather than performing penalized regression using routines in glmnet.The modified Newton Raphson scoring algorithm that this function applies is an adaptation of the C++ routine developed by Terry M. Therneau that fits Cox proportional hazards models (see survival::coxph() and more specifically survival::coxph.fit()).

Value

an accelerated oblique RSF object (aorsf)

Details on inputs

formula:

- 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 just like time + status ~ .
- A . symbol on the right hand side is short-hand for using all variables in data (omitting those on the left hand side of formula) as predictors.
- The order of variables in the left hand side matters. i.e., writing status + time ~ . will make orsf assume your status variable is actually the time variable.
- The response variable can be a survival object stored in data. For example, y ~ . is a valid formula if data$y inherits from the Surv class.

- Although you can fit an oblique random survival forest with 1 predictor variable, your formula should have at least 2 predictors. The reason for this recommendation is that a linear combination of predictors is trivial if there is only one predictor.

mtry:

The `mtry` parameter may be temporarily reduced to ensure there are at least 2 events per predictor variable. This occurs when using orsf_control_cph because coefficients in the Newton Raphson scoring algorithm may become unstable when the number of covariates is greater than or equal to the number of events. This reduction does not occur when using orsf_control_net.

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.

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.

Missing data

Data passed to aorsf functions are not allowed to have missing values. A user should impute missing values using an R package with that purpose, such as `recipes` or `mlr3pipelines`.

Examples

First we load some relevant packages

```
set.seed(329730)
suppressPackageStartupMessages({
 library(aorsf)
 library(survival)
 library(tidymodels)
 library(tidyverse)
 library(randomForestSRC)
 library(ranger)
 library(riskRegression)
 library(obliqueRSF)
})
```

The entry-point into `aorsf` is the standard call to `orsf()`:

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

printing `fit` provides quick descriptive summaries:

```
fit
```

```
## ---------- Oblique random survival forest
##
##      Linear combinations: Accelerated
##           N observations: 276
##                 N events: 111
##                  N trees: 500
##       N predictors total: 17
##    N predictors per node: 5
##  Average leaves per tree: 24
## Min observations in leaf: 5
##       Min events in leaf: 1
##           OOB stat value: 0.84
##            OOB stat type: Harrell's C-statistic
##      Variable importance: anova
##
## -----------------------------------------
```

Model control:

For these examples we will make use of the `orsf_control_` functions to build and compare models based on their out-of-bag predictions. We will also standardize the out-of-bag samples using the input argument `tree_seeds`

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_fast(),
                  formula = Surv(time, status) ~ . - id,
                  tree_seeds = 1:500)
```

Linear combinations with Cox regression:
`orsf_control_cph` runs Cox regression in each non-terminal node of each survival tree, using the regression coefficients to create linear combinations of predictors:

```
fit_cph <- orsf(pbc_orsf,
                control = orsf_control_cph(),
                formula = Surv(time, status) ~ . - id,
                tree_seeds = 1:500)
```

Linear combinations with penalized cox regression:
`orsf_control_net` runs penalized Cox regression in each non-terminal node of each survival tree, using the regression coefficients to create linear combinations of predictors. This can be really helpful if you want to do feature selection within the node, but it is a lot slower than the other options.

```
fit_net <- orsf(pbc_orsf,
                # select 3 predictors out of 5 to be used in
                # each linear combination of predictors.
                control = orsf_control_net(df_target = 3),
                formula = Surv(time, status) ~ . - id,
                tree_seeds = 1:500)
```

Linear combinations with your own function:
Let's make two customized functions to identify linear combinations of predictors.

- 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[, 2L, drop = FALSE]

  }
  ```

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_custom(beta_fun = f_rando),
                  tree_seeds = 1:500)

fit_pca <- orsf(pbc_orsf,
                Surv(time, status) ~ . - id,
                control = orsf_control_custom(beta_fun = f_pca),
                tree_seeds = 1:500)
```

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 = 1 - fit_accel$pred_oobag,
 cph   = 1 - fit_cph$pred_oobag,
 net   = 1 - fit_net$pred_oobag,
 rando = 1 - fit_rando$pred_oobag,
 pca   = 1 - fit_pca$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
## 1:   net  1788 0.9107925 0.02116880 0.8693024 0.9522826
## 2: accel  1788 0.9106308 0.02178112 0.8679406 0.9533210
## 3:   cph  1788 0.9072690 0.02120139 0.8657150 0.9488229
## 4:   pca  1788 0.8915619 0.02335399 0.8457889 0.9373349
## 5: rando  1788 0.8900944 0.02228487 0.8464168 0.9337719
```

And the indices of prediction accuracy:

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

```
##         model times       IPA
## 1:      accel  1788 0.4891448
## 2:        cph  1788 0.4687734
## 3:        net  1788 0.4652211
## 4:      rando  1788 0.4011573
## 5:        pca  1788 0.3845911
## 6: Null model  1788 0.0000000
```

From inspection,

- the PCA approach has the highest discrimination, showing that you can do very well with just a two line custom function.
- the accelerated ORSF has the highest index of prediction accuracy
- the random coefficients generally don't do that well.

tidymodels:

This example uses `tidymodels` functions but stops short of using an official `tidymodels` workflow. I am working on getting `aorsf` pulled into the `censored` package and I will update this with real workflows if that happens!

Comparing ORSF with other learners:

Start with a recipe to pre-process data

```
imputer <- recipe(pbc_orsf, formula = time + status ~ .) %>%
 step_impute_mean(all_numeric_predictors()) %>%
 step_impute_mode(all_nominal_predictors())
```

Next create a 10-fold cross validation object and pre-process the data:

```
# 10-fold cross validation; make a container for the pre-processed data
analyses <- vfold_cv(data = pbc_orsf, v = 10) %>%
 mutate(recipe = map(splits, ~prep(imputer, training = training(.x))),
        train = map(recipe, juice),
        test = map2(splits, recipe, ~bake(.y, new_data = testing(.x))))

analyses
## #  10-fold cross-validation
## # A tibble: 10 x 5
##    splits           id     recipe   train               test
##    <list>           <chr>  <list>   <list>              <list>
##  1 <split [248/28]> Fold01 <recipe> <tibble [248 x 20]> <tibble [28 x 20]>
##  2 <split [248/28]> Fold02 <recipe> <tibble [248 x 20]> <tibble [28 x 20]>
##  3 <split [248/28]> Fold03 <recipe> <tibble [248 x 20]> <tibble [28 x 20]>
##  4 <split [248/28]> Fold04 <recipe> <tibble [248 x 20]> <tibble [28 x 20]>
##  5 <split [248/28]> Fold05 <recipe> <tibble [248 x 20]> <tibble [28 x 20]>
##  6 <split [248/28]> Fold06 <recipe> <tibble [248 x 20]> <tibble [28 x 20]>
##  7 <split [249/27]> Fold07 <recipe> <tibble [249 x 20]> <tibble [27 x 20]>
##  8 <split [249/27]> Fold08 <recipe> <tibble [249 x 20]> <tibble [27 x 20]>
##  9 <split [249/27]> Fold09 <recipe> <tibble [249 x 20]> <tibble [27 x 20]>
## 10 <split [249/27]> Fold10 <recipe> <tibble [249 x 20]> <tibble [27 x 20]>
```

Define functions for a 'workflow' with `randomForestSRC`, `ranger`, and `aorsf`.

```
rfsrc_wf <- function(train, test, pred_horizon){

 # rfsrc does not like tibbles, so cast input data into data.frames
 train <- as.data.frame(train)
 test <- as.data.frame(test)

 rfsrc(formula = Surv(time, status) ~ ., data = train) %>%
  predictRisk(newdata = test, times = pred_horizon) %>%
  as.numeric()

}

ranger_wf <- function(train, test, pred_horizon){

 ranger(Surv(time, status) ~ ., data = train) %>%
  predictRisk(newdata = test, times = pred_horizon) %>%
  as.numeric()

}

aorsf_wf <- function(train, test, pred_horizon){

 train %>%
  orsf(Surv(time, status) ~ .,) %>%
  predict(new_data = test, pred_horizon = pred_horizon) %>%
```

```
  as.numeric()

}
```

Run the 'workflows' on each fold:

```
# 5 year risk prediction
ph <- 365.25 * 5

results <- analyses %>%
 transmute(test,
           pred_aorsf = map2(train, test, aorsf_wf, pred_horizon = ph),
           pred_rfsrc = map2(train, test, rfsrc_wf, pred_horizon = ph),
           pred_ranger = map2(train, test, ranger_wf, pred_horizon = ph))
```

Next unnest each column to get back a `tibble` with all of the testing data and predictions.

```
results <- results %>%
 unnest(everything())

glimpse(results)
```

```
## Rows: 276
## Columns: 23
## $ id        <int> 2, 16, 27, 66, 79, 97, 107, 116, 136, 137, 158, 189, 193, ~
## $ trt      <fct> d_penicill_main, placebo, placebo, d_penicill_main, d_peni~
## $ age      <dbl> 56.44627, 40.44353, 54.43943, 46.45311, 46.51608, 71.89322~
## $ sex       <fct> f, f, f, m, f, m, f, f, f, f, f, f, f, f, f, f, f, f, f~
## $ ascites    <fct> 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
## $ hepato     <fct> 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1~
## $ spiders    <fct> 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1~
## $ edema     <fct> 0, 0, 0.5, 0, 0, 0.5, 0, 0.5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
## $ bili      <dbl> 1.1, 0.7, 21.6, 1.4, 0.8, 2.0, 0.6, 3.0, 0.8, 1.1, 3.4, 1.~
## $ chol      <int> 302, 204, 175, 427, 315, 420, 212, 458, 263, 399, 450, 360~
## $ albumin    <dbl> 4.14, 3.66, 3.31, 3.70, 4.24, 3.26, 4.03, 3.63, 3.35, 3.60~
## $ copper     <int> 54, 28, 221, 105, 13, 62, 10, 74, 27, 79, 32, 52, 267, 76,~
## $ alk.phos   <dbl> 7394.8, 685.0, 3697.4, 1909.0, 1637.0, 3196.0, 648.0, 1588~
## $ ast       <dbl> 113.52, 72.85, 101.91, 182.90, 170.50, 77.50, 71.30, 106.9~
## $ trig      <int> 88, 58, 168, 171, 70, 91, 77, 382, 69, 152, 118, 164, 157,~
## $ platelet   <int> 221, 198, 80, 123, 426, 344, 316, 438, 206, 344, 313, 256,~
## $ protime    <dbl> 10.6, 10.8, 12.0, 11.0, 10.9, 11.4, 17.1, 9.9, 9.8, 10.1, ~
## $ stage      <ord> 3, 3, 4, 3, 3, 3, 1, 3, 2, 2, 2, 3, 4, 4, 2, 2, 3, 3, 4, 4~
## $ time      <int> 4500, 3672, 77, 4191, 3707, 611, 3388, 3336, 3098, 2990, 2~
## $ status     <dbl> 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0~
## $ pred_aorsf  <dbl> 0.21650571, 0.01569191, 0.93095617, 0.36737089, 0.12868206~
## $ pred_rfsrc  <dbl> 0.15202784, 0.01104486, 0.81913559, 0.20173550, 0.13806608~
## $ pred_ranger <dbl> 0.11418963, 0.02130315, 0.77073269, 0.22130305, 0.18419972~
```

And finish by aggregating the predictions and computing performance in the testing data. Note that I am computing one statistic for all predictions instead of computing one statistic for each fold. This approach is fine when you have smaller testing sets and/or small event counts.

```
Score(
 object = list(aorsf = results$pred_aorsf,
```

```
              rfsrc = results$pred_rfsrc,
              ranger = results$pred_ranger),
 formula = Surv(time, status) ~ 1,
 data = results,
 summary = 'IPA',
 times = ph
)

##
## Metric AUC:
##
## Results by model:
##
##     model times  AUC lower upper
## 1:  aorsf  1826 90.1  85.7  94.6
## 2:  rfsrc  1826 89.4  85.0  93.7
## 3: ranger  1826 90.1  85.9  94.3
##
## Results of model comparisons:
##
##    times  model reference delta.AUC lower upper   p
## 1:  1826  rfsrc     aorsf      -0.7  -2.3   0.8 0.4
## 2:  1826 ranger     aorsf      -0.0  -1.7   1.6 1.0
## 3:  1826 ranger     rfsrc       0.7  -0.4   1.8 0.2

##
## NOTE: Values are multiplied by 100 and given in %.

## NOTE: The higher AUC the better.

##
## Metric Brier:
##
## Results by model:
##
##         model   times Brier lower upper  IPA
## 1: Null model 1826.25  20.5  18.1  22.9  0.0
## 2:      aorsf 1826.25  11.1   8.8  13.4 45.8
## 3:      rfsrc 1826.25  12.0   9.8  14.1 41.6
## 4:     ranger 1826.25  11.8   9.7  13.9 42.5
##
## Results of model comparisons:
##
##      times  model  reference delta.Brier lower upper           p
## 1: 1826.25  aorsf Null model        -9.4 -12.1  -6.6 2.423961e-11
## 2: 1826.25  rfsrc Null model        -8.5 -10.8  -6.2 2.104905e-13
## 3: 1826.25 ranger Null model        -8.7 -11.0  -6.4 1.802417e-13
## 4: 1826.25  rfsrc      aorsf         0.9  -0.0   1.7 5.277607e-02
## 5: 1826.25 ranger      aorsf         0.7  -0.1   1.5 1.008730e-01
```

```
## 6: 1826.25 ranger      rfsrc        -0.2  -0.7   0.3 4.550782e-01

##
## NOTE: Values are multiplied by 100 and given in %.

## NOTE: The lower Brier the better, the higher IPA the better.
```

From inspection,

- aorsf obtained slightly higher discrimination (AUC)
- aorsf obtained higher index of prediction accuracy (IPA)
- Way to go, aorsf

mlr3 pipelines:

Warning: this code may or may not run depending on your current version of mlr3proba. First we load some additional mlr3 libraries.

```
suppressPackageStartupMessages({
 library(mlr3verse)
 library(mlr3proba)
 library(mlr3extralearners)
 library(mlr3viz)
 library(mlr3benchmark)
})
```

Next we'll define some tasks for our learners to engage with.

```
# Mayo Clinic Primary Biliary Cholangitis Data
task_pbc <-
 TaskSurv$new(
  id = 'pbc',
  backend = select(pbc_orsf, -id) %>%
   mutate(stage = as.numeric(stage)),
  time = "time",
  event = "status"
 )

# Veteran's Administration Lung Cancer Trial
data(veteran, package = "randomForestSRC")

task_veteran <-
 TaskSurv$new(
  id = 'veteran',
  backend = veteran,
  time = "time",
  event = "status"
 )

# NKI 70 gene signature
data_nki <- OpenML::getOMLDataSet(data.id = 1228)
```

```
task_nki <-
 TaskSurv$new(
  id = 'nki',
  backend = data_nki$data,
  time = "time",
  event = "event"
 )

# Gene Expression-Based Survival Prediction in Lung Adenocarcinoma
data_lung <- OpenML::getOMLDataSet(data.id = 1245)

task_lung <-
 TaskSurv$new(
  id = 'nki',
  backend = data_lung$data %>%
   mutate(OS_event = as.numeric(OS_event) -1),
  time = "OS_years",
  event = "OS_event"
 )


# Chemotherapy for Stage B/C colon cancer
# (there are two rows per person, one for death
#  and the other for recurrence, hence the two tasks)

task_colon_death <-
 TaskSurv$new(
  id = 'colon_death',
  backend = survival::colon %>%
   filter(etype == 2) %>%
   drop_na() %>%
   # drop id, redundant variables
   select(-id, -study, -node4, -etype),
   mutate(OS_event = as.numeric(OS_event) -1),
  time = "time",
  event = "status"
 )

task_colon_recur <-
 TaskSurv$new(
  id = 'colon_death',
  backend = survival::colon %>%
   filter(etype == 1) %>%
   drop_na() %>%
   # drop id, redundant variables
   select(-id, -study, -node4, -etype),
   mutate(OS_event = as.numeric(OS_event) -1),
  time = "time",
```

```
  event = "status"
 )

# putting them all together
tasks <- list(task_pbc,
              task_veteran,
              task_nki,
              task_lung,
              task_colon_death,
              task_colon_recur,
              # add a few more pre-made ones
              tsk("actg"),
              tsk('gbcs'),
              tsk('grace'),
              tsk("unemployment"),
              tsk("whas"))
```

Now we can make a benchmark designed to compare our three favorite learners:

```
# Learners with default parameters
learners <- lrns(c("surv.ranger", "surv.rfsrc", "surv.aorsf"))

# Brier (Graf) score, c-index and training time as measures
measures <- msrs(c("surv.graf", "surv.cindex", "time_train"))

# Benchmark with 5-fold CV
design <- benchmark_grid(
  tasks = tasks,
  learners = learners,
  resamplings = rsmps("cv", folds = 5)
)

benchmark_result <- benchmark(design)

bm_scores <- benchmark_result$score(measures, predict_sets = "test")
```

Let's look at the overall results:

```
bm_scores %>%
 select(task_id, learner_id, surv.graf, surv.cindex, time_train) %>%
 group_by(learner_id) %>%
 filter(!is.infinite(surv.graf)) %>%
 summarize(
  across(
   .cols = c(surv.graf, surv.cindex, time_train),
   .fns = mean,
   na.rm = TRUE
  )
 )
```

```
## # A tibble: 3 x 4
```

```
##   learner_id  surv.graf surv.cindex time_train
##   <chr>           <dbl>       <dbl>      <dbl>
## 1 surv.aorsf      0.151       0.729      0.345
## 2 surv.ranger     0.167       0.706      2.54
## 3 surv.rfsrc      0.156       0.715      0.783
```

From inspection,

- `aorsf` appears to have a higher expected value for 'surv.cindex' (higher is better)
- `aorsf` appears to have a lower expected value for 'surv.graf' (lower is better)
- `aorsf` has the lowest training time.

the lower training time for `aorsf` is likely due to the fact that there are many unique event times in the benchmark tasks. `ranger` and `rfsrc` create grids of time points based on each unique event time in each leaf of each decision tree, whereas `aorsf` also uses a grid but restricts it to the unique event times among observations in the current leaf.

References

Harrell FE, Califf RM, Pryor DB, Lee KL, Rosati RA. Evaluating the Yield of Medical Tests. *JAMA* 1982; 247(18):2543-2546. DOI: 10.1001/jama.1982.03320430047030

Breiman L. Random forests. *Machine learning* 2001 Oct; 45(1):5-32. DOI: 10.1023/A:1010933404324

Ishwaran H, Kogalur UB, Blackstone EH, Lauer MS. Random survival forests. *Annals of applied statistics* 2008 Sep; 2(3):841-60. DOI: 10.1214/08-AOAS169

Jaeger BC, Long DL, Long DM, Sims M, Szychowski JM, Min YI, Mcclure LA, Howard G, Simon N. Oblique random survival forests. *Annals of applied statistics* 2019 Sep; 13(3):1847-83. DOI: 10.1214/19-AOAS1261

Jaeger BC, Welden S, Lenoir K, Speiser JL, Segar MW, Pandey A, Pajewski NM. Accelerated and interpretable oblique random survival forests. *arXiv e-prints* 2022 Aug; arXiv-2208. URL: https://arxiv.org/abs/2208.01129

`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

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()`

Examples

```
orsf(data = pbc_orsf,
     formula = Surv(time, status) ~ . - id,
     control = orsf_control_cph())
```

`orsf_control_custom` *Custom ORSF control*

Description

Custom ORSF control

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.

Examples

Two customized functions to identify linear combinations of predictors are shown here.

- The first uses random coefficients
- The second derives coefficients from principal component analysis.

Random coefficients:

`f_rando()` is our function to get the random coefficients:

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

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

```
library(aorsf)

fit_rando <- orsf(pbc_orsf,
                  Surv(time, status) ~ . - id,
                  control = orsf_control_custom(beta_fun = f_rando),
                  n_tree = 500)

fit_rando
```

```
## ---------- Oblique random survival forest
##
##      Linear combinations: Custom user function
##           N observations: 276
##                 N events: 111
##                  N trees: 500
##       N predictors total: 17
##    N predictors per node: 5
##  Average leaves per tree: 23
## Min observations in leaf: 5
##       Min events in leaf: 1
##           OOB stat value: 0.82
##            OOB stat type: Harrell's C-statistic
##      Variable importance: anova
##
## ----------------------------------------
```

Principal components:

Follow the same steps as above, starting with the custom function:

```
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[, 2L, drop = FALSE]

}
```

Then plug the function into `orsf_control_custom()` and pass the result into `orsf()`:

```
fit_pca <- orsf(pbc_orsf,
                Surv(time, status) ~ . - id,
                control = orsf_control_custom(beta_fun = f_pca),
                n_tree = 500)
```

Evaluate:

How well do our two customized ORSFs do? Let's compute their indices of prediction accuracy based on out-of-bag predictions:

```
library(riskRegression)
```

```
## riskRegression version 2022.09.23

library(survival)

risk_preds <- list(rando = 1 - fit_rando$pred_oobag,
                   pca = 1 - fit_pca$pred_oobag)

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

The PCA ORSF does quite well! (higher IPA is better)

```
sc$Brier
```

```
##
## Results by model:
##
##         model times  Brier  lower  upper    IPA
## 1: Null model  1788 20.479 18.090 22.868  0.000
## 2:      rando  1788 12.381 10.175 14.588 39.541
## 3:        pca  1788 12.496 10.476 14.515 38.983
##
## Results of model comparisons:
##
##    times model  reference delta.Brier   lower  upper           p
## 1:  1788 rando Null model      -8.098 -10.392 -5.804 4.558033e-12
## 2:  1788   pca Null model      -7.983  -9.888 -6.078 2.142713e-16
## 3:  1788   pca      rando       0.114  -0.703  0.932 7.838255e-01

##
## NOTE: Values are multiplied by 100 and given in %.

## NOTE: The lower Brier the better, the higher IPA the better.
```

See Also

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

`orsf_control_fast` *Accelerated ORSF control*

Description

Accelerated ORSF control

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`. It would be a good idea to center and scale your predictors prior to running `orsf()` if you plan on setting `do_scale = FALSE`.

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()`

Examples

```
orsf(data = pbc_orsf,
     formula = Surv(time, status) ~ . - id,
     control = orsf_control_fast())
```

`orsf_control_net` *Penalized Cox regression ORSF control*

Description

Penalized Cox regression ORSF control

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

`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

Simon N, Friedman J, Hastie T, Tibshirani R. Regularization paths for Cox's proportional hazards model via coordinate descent. *Journal of statistical software* 2011 Mar; 39(5):1. DOI: 10.18637/jss.v039.i05

See Also

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

Examples

```
# orsf_control_net() is considerably slower than orsf_control_cph(),
# The example uses n_tree = 25 so that my examples run faster,
# but you should use at least 500 trees in applied settings.

orsf(data = pbc_orsf,
     formula = Surv(time, status) ~ . - id,
     n_tree = 25,
     control = orsf_control_net())
```

`orsf_ice_oob` *ORSF Individual Conditional Expectations*

Description

Compute individual conditional expectations for an ORSF model. 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 = "risk",
  expand_grid = TRUE,
  boundary_checks = TRUE,
  ...
)

orsf_ice_inb(
  object,
  pred_spec,
  pred_horizon = NULL,
  pred_type = "risk",
  expand_grid = TRUE,
  boundary_checks = TRUE,
  ...
)

orsf_ice_new(
  object,
  pred_spec,
  new_data,
  pred_horizon = NULL,
  pred_type = "risk",
  na_action = "fail",
  expand_grid = TRUE,
  boundary_checks = TRUE,
```

```
  ...
)
```

Arguments

`object`	(*orsf_fit*) a trained oblique random survival forest (see orsf).
`pred_spec`	(*named list* 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 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*) 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 options are • 'risk' : probability of having an event at or before `pred_horizon`. • 'surv' : 1 - risk. • 'chf': cumulative hazard function • 'mort': mortality prediction
`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.
`...`	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) at the specified prediction horizon(s).

Examples

Begin by fitting an ORSF ensemble

```
library(aorsf)

set.seed(329)

fit <- orsf(data = pbc_orsf, formula = Surv(time, status) ~ . - id)

fit
```

```
## ---------- Oblique random survival forest
##
##      Linear combinations: Accelerated
##           N observations: 276
##                 N events: 111
##                  N trees: 500
##       N predictors total: 17
##    N predictors per node: 5
##  Average leaves per tree: 25
## Min observations in leaf: 5
##       Min events in leaf: 1
##           OOB stat value: 0.84
##            OOB stat type: Harrell's C-statistic
##      Variable importance: anova
##
## -----------------------------------------
```

Use the ensemble to compute ICE values using out-of-bag predictions:

```
pred_spec <- list(bili = seq(1, 10, length.out = 25))

ice_oob <- orsf_ice_oob(fit, pred_spec, boundary_checks = FALSE)

ice_oob
```

```
##       pred_horizon id_variable id_row bili      pred
##    1:         1788           1      1    1 0.8935318
##    2:         1788           1      2    1 0.1025087
##    3:         1788           1      3    1 0.6959198
##    4:         1788           1      4    1 0.3465760
##    5:         1788           1      5    1 0.1105536
##   ---
## 6896:         1788          25    272   10 0.4409361
## 6897:         1788          25    273   10 0.4493052
## 6898:         1788          25    274   10 0.4696659
## 6899:         1788          25    275   10 0.3892409
## 6900:         1788          25    276   10 0.4565133
```

Much more detailed examples are given in the vignette

`orsf_pd_oob` *ORSF partial dependence*

Description

Compute partial dependence for an ORSF model. 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 = "risk",
  expand_grid = TRUE,
  prob_values = c(0.025, 0.5, 0.975),
  prob_labels = c("lwr", "medn", "upr"),
  boundary_checks = TRUE,
  ...
)

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

orsf_pd_new(
  object,
```

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

Arguments

`object`	(*orsf_fit*) a trained oblique random survival forest (see orsf).
`pred_spec`	(*named list* 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 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*) 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 options are • 'risk' : probability of having an event at or before `pred_horizon`. • 'surv' : 1 - risk. • 'chf': cumulative hazard function • 'mort': mortality prediction
`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.

...
: 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) at the specified prediction horizon(s).

Examples

Begin by fitting an ORSF ensemble:

```
library(aorsf)

set.seed(329730)

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

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

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

Three ways to compute PD and ICE:

You can compute partial dependence and ICE three ways with aorsf:

- using in-bag predictions for the training data

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

pd_train
```

```
##    pred_horizon bili      mean        lwr      medn       upr
## 1:      1826.25    1 0.2054232 0.01599366 0.0929227 0.8077278
## 2:      1826.25    2 0.2369077 0.02549869 0.1268457 0.8227315
## 3:      1826.25    3 0.2808514 0.05027265 0.1720280 0.8457834
## 4:      1826.25    4 0.3428065 0.09758988 0.2545869 0.8575243
## 5:      1826.25    5 0.3992909 0.16392752 0.3232681 0.8634269
```

- using out-of-bag predictions for the training data

```
pd_train <- orsf_pd_oob(fit, pred_spec = list(bili = 1:5))

pd_train
```

```
##    pred_horizon bili      mean        lwr       medn       upr
## 1:      1826.25    1 0.2068300 0.01479443 0.08824123 0.8053317
## 2:      1826.25    2 0.2377046 0.02469718 0.12623031 0.8258154
## 3:      1826.25    3 0.2810546 0.04080813 0.18721220 0.8484846
## 4:      1826.25    4 0.3417839 0.09076851 0.24968438 0.8611884
## 5:      1826.25    5 0.3979925 0.16098228 0.32147532 0.8554402
```

- using predictions for a new set of data

```
pd_test <- orsf_pd_new(fit,
                       new_data = pbc_orsf_test,
                       pred_spec = list(bili = 1:5))

pd_test
```

```
##    pred_horizon bili      mean        lwr      medn       upr
## 1:      1826.25    1 0.2510900 0.01631318 0.1872414 0.8162621
## 2:      1826.25    2 0.2807327 0.02903956 0.2269297 0.8332956
## 3:      1826.25    3 0.3247386 0.05860235 0.2841853 0.8481825
## 4:      1826.25    4 0.3850799 0.10741224 0.3405760 0.8588955
## 5:      1826.25    5 0.4394952 0.17572657 0.4050864 0.8657886
```

- in-bag partial dependence indicates relationships that the model has learned during training. This is helpful if your goal is to interpret the model.
- 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. 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.
- 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.

References

Giles Hooker, Lucas Mentch, Siyu Zhou. Unrestricted Permutation forces Extrapolation: Variable Importance Requires at least One More Model, or There Is No Free Variable Importance. *arXiv e-prints* 2021 Oct; arXiv-1905. URL: https://doi.org/10.48550/arXiv.1905.03151

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` *ORSF summary; univariate*

Description

Summarize the univariate information from an ORSF object

Usage

```
orsf_summarize_uni(
  object,
  n_variables = NULL,
  pred_horizon = NULL,
  pred_type = "risk",
  importance = "negate",
  ...
)
```

Arguments

`object`	(*orsf_fit*) a trained oblique random survival forest (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*) 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 options are • 'risk' : probability of having an event at or before `pred_horizon`. • 'surv' : 1 - risk. • 'chf': cumulative hazard function • 'mort': mortality prediction
`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.
`...`	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)

# since anova importance was used to make object, we can
# safely say importance = 'none' and skip computation of
# variable importance while running orsf_summarize_uni

orsf_summarize_uni(object, n_variables = 3, importance = 'none')

# however, 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 = 3, importance = 'negate')
```

---

orsf_time_to_train *Estimate training time*

Description

Estimate training time

Usage

```
orsf_time_to_train(object, n_tree_subset = 50)
```

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 50, as this usually gives a good enough approximation.

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 = 500, no_fit = TRUE)

# grow 50 trees to approximate the time it will take to grow 500 trees
time_estimated <- orsf_time_to_train(object, n_tree_subset = 50)

print(time_estimated)

# let's see how close the approximation was
time_true_start <- Sys.time()
fit <- 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_vi *ORSF variable importance*

Description

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

Usage

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

orsf_vi_negate(object, group_factors = TRUE, oobag_fun = NULL, ...)
```

```
orsf_vi_permute(object, group_factors = TRUE, oobag_fun = NULL, ...)

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

Arguments

`object`	(*orsf_fit*) a trained oblique random survival forest (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), Harrell's C-statistic (1982) is used to evaluate accuracy. • if you use your own `oobag_fun` note the following: – `oobag_fun` should have two inputs: `y_mat` and `s_vec` – `y_mat` is a two column matrix with first column named 'time', second named 'status' – `s_vec` is a numeric vector containing predicted survival probabilities. – `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.
`...`	Further arguments passed to or from other methods (not currently used).

Details

When an `orsf_fit` object is fitted with importance = 'anova', 'negate', or 'permute', the output will have a vector of importance values based on the requested type of importance. However, you may still want to call `orsf_vi()` on this output if you want to group factor levels into one overall importance value.

`orsf_vi()` is a general purpose function to extract or compute variable importance estimates from an `'orsf_fit'` 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, 2022 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 ORSF ensemble.

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

fit
```

```
## ---------- Oblique random survival forest
##
##      Linear combinations: Accelerated
##           N observations: 276
##                 N events: 111
##                  N trees: 500
##       N predictors total: 17
##    N predictors per node: 5
##  Average leaves per tree: 25
## Min observations in leaf: 5
##       Min events in leaf: 1
##           OOB stat value: 0.84
```

```
##            OOB stat type: Harrell's C-statistic
##      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

```
attr(fit, 'importance_values')
```

```
##     edema_1   ascites_1        bili      copper         age     albumin
##  0.41468531  0.34547820  0.27357335  0.19702602  0.17831563  0.17231851
##   edema_0.5     protime        chol       stage       sex_f   spiders_1
##  0.16100917  0.15265823  0.14529486  0.13818084  0.13186813  0.12881052
##         ast    hepato_1    alk.phos        trig    platelet trt_placebo
##  0.12509496  0.11370348  0.10024752  0.09878683  0.08006941  0.06398488
```

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:

  ```
  fit$importance
  ```

  ```
  ##    ascites      bili     edema    copper       age   albumin   protime
  ## 0.34547820 0.27357335 0.26368761 0.19702602 0.17831563 0.17231851 0.15265823
  ##      chol     stage       sex   spiders       ast    hepato   alk.phos
  ## 0.14529486 0.13818084 0.13186813 0.12881052 0.12509496 0.11370348 0.10024752
  ##       trig   platelet        trt
  ## 0.09878683 0.08006941 0.06398488
  ```

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

  ```
  orsf_vi(fit)
  ```

  ```
  ##    ascites      bili     edema    copper       age   albumin   protime
  ## 0.34547820 0.27357335 0.26368761 0.19702602 0.17831563 0.17231851 0.15265823
  ##      chol     stage       sex   spiders       ast    hepato   alk.phos
  ## 0.14529486 0.13818084 0.13186813 0.12881052 0.12509496 0.11370348 0.10024752
  ##       trig   platelet        trt
  ## 0.09878683 0.08006941 0.06398488
  ```

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 ORSF:

You can fit an ORSF 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           age       protime       albumin
##  0.0717336945  0.0288601792  0.0253698687  0.0110960617  0.0100020838
##          chol       ascites       spiders           ast         stage
##  0.0075015628  0.0060950198  0.0045321942  0.0044280058  0.0025526151
##         edema           sex        hepato      platelet      alk.phos
##  0.0024856369  0.0015628256  0.0004688477  0.0003646593 -0.0007293186
##          trig           trt
## -0.0020316733 -0.0061471140
```

```
orsf_vi_permute(fit_no_vi)
```

```
##           age          bili        copper       albumin          chol
##  1.109606e-02  1.083559e-02  7.032715e-03  5.157324e-03  4.636383e-03
##       protime       ascites       spiders           ast      platelet
##  4.011252e-03  3.854970e-03  2.396333e-03  1.146072e-03  5.209419e-04
##      alk.phos         edema           sex        hepato          trig
##  2.083767e-04  1.959734e-04  5.209419e-05 -4.688477e-04 -1.719108e-03
##           trt
## -3.698687e-03
```

ORSF and VI all at once:

fit an ORSF 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           age        copper         stage       ascites
##  0.0114086268  0.0094811419  0.0055219837  0.0043238175  0.0032298395
##       albumin        hepato       protime           ast         edema
##  0.0031256512  0.0030214628  0.0029172744  0.0021358616  0.0019051588
##       spiders          chol      alk.phos      platelet           trt
##  0.0017712023  0.0013023547  0.0008335070 -0.0009376954 -0.0016149198
##           sex          trig
## -0.0020837675 -0.0022921442
```

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

```
orsf_vi_negate(fit_permute_vi)
```

```
##          bili        copper           age       protime       albumin
##  0.0773598666  0.0272452594  0.0258387164  0.0115649094  0.0084392582
##           sex          chol           ast       ascites         stage
##  0.0081787872  0.0074494686  0.0060429256  0.0058866431  0.0043238175
##        hepato         edema       spiders      platelet          trig
##  0.0040112523  0.0027684339  0.0026047093  0.0005730360  0.0002083767
##           trt      alk.phos
## -0.0003125651 -0.0016149198
```

References

Harrell FE, Califf RM, Pryor DB, Lee KL, Rosati RA. Evaluating the Yield of Medical Tests. *JAMA* 1982; 247(18):2543-2546. DOI: 10.1001/jama.1982.03320430047030

Breiman L. Random forests. *Machine learning* 2001 Oct; 45(1):5-32. DOI: 10.1023/A:1010933404324

Menze BH, Kelm BM, Splitthoff DN, Koethe U, Hamprecht FA. On oblique random forests. *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* 2011 Sep 4; pp. 453-469. DOI: 10.1007/978-3-642-23783-6_29

Jaeger BC, Welden S, Lenoir K, Speiser JL, Segar MW, Pandey A, Pajewski NM. Accelerated and interpretable oblique random survival forests. *arXiv e-prints* 2022 Aug; arXiv-2208. URL: https://arxiv.org/abs/2208.01129

`orsf_vs` *Variable selection*

Description

Variable selection

Usage

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

Arguments

`object`	(*orsf_fit*) a trained oblique random survival forest (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

`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
- *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',
               tree_seeds = 1:25)

orsf_vs(object)
```

---

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.

`predict.orsf_fit` *Compute predictions using ORSF*

Description

Predicted risk, survival, hazard, or mortality from an ORSF model.

Usage

```
## S3 method for class 'orsf_fit'
predict(
  object,
  new_data,
  pred_horizon = NULL,
  pred_type = "risk",
  na_action = "fail",
  boundary_checks = TRUE,
  ...
)
```

Arguments

object	(*orsf_fit*) a trained oblique random survival forest (see orsf).
new_data	a data.frame, tibble, or data.table to compute predictions in.
pred_horizon	(*double*) 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 options are • 'risk' : probability of having an event at or before pred_horizon. • 'surv' : 1 - risk. • 'chf': cumulative hazard function • 'mort': mortality prediction
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.
...	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

Begin by fitting an ORSF ensemble:

```
library(aorsf)

set.seed(329730)

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

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

fit <- 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,
        new_data = pbc_orsf_test[1:5, ],
        pred_type = 'risk',
        pred_horizon = c(500, 1000, 1500))
```

```
##            [,1]       [,2]       [,3]
## [1,] 0.48792661 0.75620281 0.90618133
## [2,] 0.04293829 0.09112952 0.18602887
## [3,] 0.12147573 0.27784498 0.41600114
## [4,] 0.01136075 0.03401092 0.08236831
## [5,] 0.01294947 0.02070625 0.05645823
```

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

```
##           [,1]      [,2]       [,3]
## [1,] 0.5120734 0.2437972 0.09381867
## [2,] 0.9570617 0.9088705 0.81397113
## [3,] 0.8785243 0.7221550 0.58399886
## [4,] 0.9886393 0.9659891 0.91763169
## [5,] 0.9870505 0.9792937 0.94354177
```

```
# predicted cumulative hazard function
# (expected number of events for person i at time j)
predict(fit,
```

```
        new_data = pbc_orsf_test[1:5, ],
        pred_type = 'chf',
        pred_horizon = c(500, 1000, 1500))
```

```
##            [,1]       [,2]       [,3]
## [1,] 0.68107429 1.28607479 1.70338193
## [2,] 0.04519460 0.10911618 0.24871482
## [3,] 0.14686474 0.41252079 0.69005048
## [4,] 0.01149952 0.03951923 0.10628942
## [5,] 0.01338978 0.02214232 0.06644605
```

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

```
##           [,1]
## [1,] 68.394152
## [2,] 12.299344
## [3,] 28.208251
## [4,]  6.475339
## [5,]  4.247305
```

`print.orsf_fit` *Inspect your ORSF model*

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 'orsf_fit'
print(x, ...)
```

Arguments

x	(*orsf_fit*) 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)

smry <- orsf_summarize_uni(object, n_variables = 3)

print(smry)
```

Index