Package ‘aorsf’

October 26, 2023

Title Accelerated Oblique Random Survival Forests

Version 0.1.1

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., (2023) <DOI:10.1080/10618600.2023.2231048>.

License MIT + file LICENSE

Encoding UTF-8

LazyData true

RoxygenNote 7.2.3

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, markdown, 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 2023-10-26 16:40:06 UTC
R topics documented:

as.data.table.orsf_summary_uni ........................................ 2
orsf ................................................................. 3
orsf_control_cph ..................................................... 19
orsf_control_custom .................................................. 20
orsf_control_fast ..................................................... 23
orsf_control_net ...................................................... 24
orsf_ice_oob .......................................................... 25
orsf_pd_oob ........................................................... 28
orsf_scale_cph ........................................................ 32
orsf_summarize_uni ..................................................... 33
orsf_time_to_train .................................................... 35
orsf_vi ................................................................. 36
orsf_vs ................................................................. 41
pbc_orsf ............................................................... 42
predict.orsf_fit ......................................................... 43
print.orsf_fit ........................................................ 46
print.orsf_summary_uni .................................................. 47

Index 49

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,
n_thread = 1,
mtry = NULL,
sample_with_replacement = TRUE,
sample_fraction = 0.632,
leaf_min_events = 1,
leaf_min_obs = 5,
split_rule = "logrank",
split_min_events = 5,
split_min_obs = 10,
split_min_stat = switch(split_rule, logrank = 3.841459, cstat = 0.5),
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.

n_thread (integer) number of threads to use while growing trees, computing predictions,
and computing importance. Default is one thread. To use the maximum number
of threads that your system provides for concurrent execution, set n_thread =
0.

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 num-
ber of total predictors, i.e., mtry = ceiling(sqrt(number of predictors))
orsf

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) 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 are
• 'logrank' : a log-rank test statistic.
• 'cstat' : Harrell’s concordance statistic.

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 if split_rule = 'logrank' and 0.50 if split_rule = 'cstat'. 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).

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' : probability of event occurring at or before oobag_pred_horizon.
• '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.

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.
ooabg_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 if 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. 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

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.

**n_thread:**

If an R function must 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.
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 et al., 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

```r
set.seed(329730)
suppressPackageStartupMessages({
  library(aorsf)
  library(survival)
  library(tidymodels)
  library(tidyverse)
  library(randomForestSRC)
  library(ranger)
  library(riskRegression)
  library(obliqueRSF)
})
```
The entry-point into `orsf` is the standard call to `orsf()`:

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

Printing `fit` provides quick descriptive summaries:

```r
fit
```

```r
## ---------- 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: 21
## 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.

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

**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:

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

**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.
# select 3 predictors out of 5 to be used in each linear combination of predictors.
fit_net <- orsf(pbc_orsf,
    control = orsf_control_net(df_target = 3),
    formula = Surv(time, status) ~ . - id,
    tree_seeds = 329)

**Linear combinations with your own function:**

Let’s make two customized functions to identify linear combinations of predictors.

- The first uses random coefficients
  
  ```r
  f_rando <- function(x_node, y_node, w_node) {
    matrix(runif(ncol(x_node)), ncol=1)
  }
  ```

- The second derives coefficients from principal component analysis.
  
  ```r
  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 `orsf()` inside of `orsf()`.
  
  ```r
  f_aorsf <- function(x_node, y_node, w_node) {
    # This approach is known as reinforcement learning trees.
    # some special care is taken to prevent your R session from crashing.
    # Specifically, random coefficients are used when n_obs <= 10
    # or n_events <= 5.
    f_aorsf <- 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 || sum(y_node[, 'status']) <= 5) {
        return(matrix(runif(ncol(x_node)), ncol = 1))
      }

      fit <- orsf(data, time + status ~ .,
          weights = as.numeric(w_node),
          n_tree = 25,
          importance = 'permute')

      out <- orsf_vi(fit)

      # drop the least two important variables
      n_vars <- length(out)
      out[c(n_vars, n_vars-1)] <- 0
  }
  ```
# ensure out has same variable order as input
out <- out[, colnames(x_node)]

matrix(out, ncol = 1)

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

```r
fit_rando <- orsf(pbc_orsf, Surv(time, status) ~ . - id, control = orsf_control_custom(beta_fun = f_rando), tree_seeds = 329)
fit_pca <- orsf(pbc_orsf, Surv(time, status) ~ . - id, control = orsf_control_custom(beta_fun = f_pca), tree_seeds = 329)
fit_rlt <- orsf(pbc_orsf, time + status ~ . - id, control = orsf_control_custom(beta_fun = f_aorsf), 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.

```r
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$pred_oobag,
  rlt = 1 - fit_rlt$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:

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

<table>
<thead>
<tr>
<th>model</th>
<th>times</th>
<th>AUC</th>
<th>se</th>
<th>lower</th>
<th>upper</th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>1788</td>
<td>0.9134593</td>
<td>0.02079935</td>
<td>0.8726933</td>
<td>0.9542253</td>
</tr>
<tr>
<td>accel</td>
<td>1788</td>
<td>0.9112315</td>
<td>0.02098077</td>
<td>0.8701099</td>
<td>0.9523530</td>
</tr>
<tr>
<td>rlt</td>
<td>1788</td>
<td>0.9076685</td>
<td>0.02078732</td>
<td>0.8669261</td>
<td>0.9484109</td>
</tr>
<tr>
<td>cph</td>
<td>1788</td>
<td>0.9063871</td>
<td>0.02165434</td>
<td>0.8639453</td>
<td>0.9488288</td>
</tr>
<tr>
<td>rando</td>
<td>1788</td>
<td>0.9023489</td>
<td>0.02218936</td>
<td>0.8588586</td>
<td>0.9458393</td>
</tr>
<tr>
<td>pca</td>
<td>1788</td>
<td>0.8994220</td>
<td>0.02201713</td>
<td>0.8562692</td>
<td>0.9425748</td>
</tr>
</tbody>
</table>
```

And the indices of prediction accuracy:
sc$Brier$score[order(~IPA), .(model, times, IPA)]
## model times IPA
## 1: net 1788 0.4916038
## 2: accel 1788 0.4879683
## 3: cph 1788 0.4751883
## 4: rlt 1788 0.4570179
## 5: pca 1788 0.4370592
## 6: rando 1788 0.4258344
## 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.

**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

```r
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:

```r
# 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))))
```

Define functions for a ‘workflow’ with randomForestSRC, ranger, and aorsf.

```r
rfsrc_wf <- function(train, test, pred_horizon){
  # workflow for randomForestSRC
  rfsrc_fit <- function(train, test, pred_horizon) {
    # function to fit randomForestSRC model
    # ... (code to fit model)
  }

  # workflow for ranger
  ranger_fit <- function(train, test, pred_horizon) {
    # function to fit ranger model
    # ... (code to fit model)
  }

  # workflow for aorsf
  aorsf_fit <- function(train, test, pred_horizon) {
    # function to fit aorsf model
    # ... (code to fit model)
  }

  # run workflows
  result <- map(train, rfsrc_fit, test, pred_horizon)
  # ... (code to extract results)
  return(result)
}
```
orsf 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 %>%
orfs(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> 20, 47, 55, 57, 63, 65, 83, 121, 141, 181, 197, 201, 202, ~
## $ trt <fct> placebo, placebo, d_penicill_main, d_penicill_main, placeb~
## $ age <dbl> 59.95346, 47.42779, 65.76318, 53.57153, 46.62834, 40.20260~
## $ sex <fct> f, f, m, f, f, f, m, f, f, f, f, f, m, f, m, f, f, f, f~
## $ ascites <fct> 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
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 90.2 85.8 94.6
## 2: rfsrc 1826 89.4 84.9 93.9
## 3: ranger 1826 89.8 85.5 94.1
## Results of model comparisons:
##    times model reference delta.AUC lower upper  p
## 1: 1826 rfsrc aorsf -0.8 -2.3 0.6 0.3
## 2: 1826 ranger aorsf -0.4 -1.8 1.0 0.6
## 3: 1826 ranger rfsrc 0.4 -0.7 1.6 0.4
## NOTE: Values are multiplied by 100 and given in %.

## NOTE: The higher AUC the better.

### Metric Brier:

### Results by model:

<table>
<thead>
<tr>
<th>model</th>
<th>times</th>
<th>Brier</th>
<th>lower</th>
<th>upper</th>
<th>IPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Null</td>
<td>1826.25</td>
<td>20.5</td>
<td>18.1</td>
<td>22.9</td>
<td>0.0</td>
</tr>
<tr>
<td>2: aorsf</td>
<td>1826.25</td>
<td>11.0</td>
<td>8.7</td>
<td>13.3</td>
<td>46.2</td>
</tr>
<tr>
<td>3: rfsrc</td>
<td>1826.25</td>
<td>12.0</td>
<td>9.8</td>
<td>14.2</td>
<td>41.3</td>
</tr>
<tr>
<td>4: ranger</td>
<td>1826.25</td>
<td>11.8</td>
<td>9.7</td>
<td>14.0</td>
<td>42.3</td>
</tr>
</tbody>
</table>

### Results of model comparisons:

<table>
<thead>
<tr>
<th>times</th>
<th>model</th>
<th>reference</th>
<th>delta.Brier</th>
<th>lower</th>
<th>upper</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1826.25</td>
<td>aorsf</td>
<td>Null model</td>
<td>-9.5</td>
<td>-12.1</td>
<td>-6.8</td>
<td>1.5046E-12</td>
</tr>
<tr>
<td>1826.25</td>
<td>rfsrc</td>
<td>Null model</td>
<td>-8.5</td>
<td>-10.8</td>
<td>-6.1</td>
<td>7.4696E-13</td>
</tr>
<tr>
<td>1826.25</td>
<td>ranger</td>
<td>Null model</td>
<td>-8.7</td>
<td>-11.0</td>
<td>-6.3</td>
<td>1.0228E-12</td>
</tr>
<tr>
<td>1826.25</td>
<td>rfsrc</td>
<td>aorsf</td>
<td>1.0</td>
<td>0.3</td>
<td>1.7</td>
<td>5.8052E-03</td>
</tr>
<tr>
<td>1826.25</td>
<td>ranger</td>
<td>rfsrc</td>
<td>0.8</td>
<td>0.1</td>
<td>1.5</td>
<td>2.1645E-02</td>
</tr>
</tbody>
</table>

## 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)

### mlr3 pipelines:

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

```r
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.rfsrsc", "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 0.152 0.733 1.41
## 2 surv.ranger 0.166 0.712 1.95
## 3 surv.rfsrc 0.155 0.723 0.745

From inspection,

• aorsf has a higher expected value for ‘surv.cindex’ (higher is better)
• aorsf has a lower expected value for ‘surv.graf’ (lower is better)

References


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

See Also

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

Examples

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

### orsf_control_custom

**Custom ORSF control**

**Description**

Custom ORSF control

**Usage**

```r
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:

\( \text{f}_\text{rando}() \) is our function to get the random coefficients:

\[
\text{f}_\text{rando} \left( x\_\text{node}, y\_\text{node}, w\_\text{node} \right) = \begin{bmatrix}
\text{runif}(\text{ncol}(x\_\text{node})) \\
\end{bmatrix}
\]

We can plug \( \text{f}_\text{rando} \) into \text{orsf-control-custom()}, and then pass the result into \text{orsf()}:

\[
\text{library(aorsf)} \\
\text{fit\_rando} <- \text{orsf}(\text{pbc_orfs}, \\
\text{Surv(time, status) ~ . - id,} \\
\text{control = orsf-control-custom(beta.fun = f\_rando),} \\
\text{n\_tree = 500})
\]

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: 20
## #  Min observations in leaf: 5
## #  Min events in leaf: 1
## #  OOB stat value: 0.83
## #  OOB stat type: Harrell's C-statistic
## #  Variable importance: anova
## #
## -----------------------------------------

Principal components:

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

\[
\text{f\_pca} \left( x\_\text{node}, y\_\text{node}, w\_\text{node} \right) = \begin{bmatrix}
\text{stats::prcomp}(x\_\text{node})[, 2L, drop = FALSE] \\
\end{bmatrix}
\]

Then plug the function into \text{orsf-control-custom()} and pass the result into \text{orsf()}:

\[
\text{fit\_pca} <- \text{orsf}(\text{pbc_orfs}, \\
\text{Surv(time, status) ~ . - id,}
\]

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 2023.09.08
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 11.748 9.702 13.793 42.636
## 3: pca 1788 12.884 10.906 14.862 37.088

## Results of model comparisons:
##
## times model reference delta.Brier lower upper p
## 1: 1788 rando Null model -8.731 -10.932 -6.531 7.346799e-15
## 3: 1788 pca rando 1.136 0.265 2.008 1.061347e-02

## 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

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()

Examples

orsf(data = pbc_orSF,  
formulan = Surv(time, status) ~ . - id,  
control = orsf_control_fast())
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

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


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_orhf,
    formula = Surv(time, status) ~ . - id,
orsf_ice_oob

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, 
n_thread = 1, 
...
)

dorsf_ice_inb(
oBJECT, 
pred_spec, 
pred_horizon = NULL, 
pred_type = "risk", 
expand_grid = TRUE, 
boundary_checks = TRUE, 
n_thread = 1, 
...
)

dorsf_ice_new(
oBJECT, 
pred_spec, 
new_data,
orsf_ice_oob

```r
pred_horizon = NULL,
pred_type = "risk",
na_action = "fail",
expand_grid = TRUE,
boundary_checks = TRUE,
n_thread = 1,
...
)
```

**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.
- `n_thread` *(integer)* number of threads to use while computing predictions. Default is one thread. To use the maximum number of threads that your system provides for concurrent execution, set `n_thread = 0`.
- `...` 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

```r
library(aorsf)
set.seed(329)
fit <- orsf(data = pbc_orsf, formula = 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
## 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:

```r
pred_spec <- list(bili = seq(1, 10, length.out = 25))
ice_oob <- orsf_ice_oob(fit, pred_spec, boundary_checks = FALSE)
ice_oob
```

```
## id_variable id_row pred_horizon bili  pred
## 1: 1 1 1788 1 167.86459
## 2: 1 2 1788 1 21.77000
## 3: 1 3 1788 1 118.36972
## 4: 1 4 1788 1 63.11360
## 5: 1 5 1788 1 20.65211
## ---
```
## 6896: 25 272 1788 10 61.93365
## 6897: 25 273 1788 10 78.19472
## 6898: 25 274 1788 10 89.93071
## 6899: 25 275 1788 10 56.40274
## 6900: 25 276 1788 10 99.28738

Much more detailed examples are given in the vignette.

### 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

```r
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,
  n_thread = 1,
  ...
)
```

```r
orsf_pd_inb(
  object,
  pred_spec,
  pred_horizon = NULL,
  pred_type = "risk",
  expand_grid = TRUE,
  prob_values = c(0.025, 0.5, 0.975),
```

```
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,
  n_thread = 1,
  ...
)

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.
rorsf_pd_oob_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 one thread. To use the maximum number of threads that your system provides for concurrent execution, set n_thread = 0.

... 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, ]
orsf_pd_oob

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 `orsf`:

- using in-bag predictions for the training data
  ```r
  pd_train <- orsf_pd_inb(fit, pred_spec = list(bili = 1:5))
  ```

- using out-of-bag predictions for the training data
  ```r
  pd_train <- orsf_pd_oob(fit, pred_spec = list(bili = 1:5))
  ```

- using predictions for a new set of data
  ```r
  pd_test <- orsf_pd_new(fit,
                         new_data = pbc_orsf_test,
                         pred_spec = list(bili = 1:5))
  ```

- in-bag partial dependence indicates relationships that the model has learned during training.
- 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


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

```r
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

```r
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.
orsf_summarize_uni

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, n_tree = 25)

# 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_orfs, 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 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,
  n_thread = 1,
  verbose_progress = FALSE,
  ...
)

orsf_vi_negate(
  object,
  group_factors = TRUE,
  oobag_fun = NULL,
  n_thread = 1,
  verbose_progress = FALSE,
  ...
)

orsf_vi_permute(
  object,
  group_factors = TRUE,
  oobag_fun = NULL,
  n_thread = 1,
  verbose_progress = FALSE,
  ...
)

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:
orsf_vi

- '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.

**n_thread** *(integer)* number of threads to use while computing predictions. Default is one thread. To use the maximum number of threads that your system provides for concurrent execution, set n_thread = 0.

**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 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, (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 ORSF ensemble.

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

```
## ---------- 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
## 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
orsf_vi

attr(fit, 'importance_values')
## edema_1 ascites_1 bili copper albumin age
## 0.53189300 0.49950642 0.39598881 0.30443254 0.26028060 0.24758399
## protime stage chol edema_0.5 spiders_1 ast
## 0.22874192 0.20974576 0.20353982 0.18401760 0.18090452 0.17457962
## hepato_1 sex_f trig alk.phos platelet trt_placebo
## 0.16402406 0.14803440 0.13009809 0.11627907 0.07853659 0.06939410

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

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

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 sex protime stage albumin
## 0.113868091 0.048934519 0.037084634 0.027045537 0.023883035 0.021168322
## age ascites chol ast hepato edema
## 0.020510253 0.014990139 0.014700024 0.011445420 0.007705758 0.007216976
## spiders trig alk.phos trt platelet
## 0.006373687 0.003230074 0.002825097 0.002487956 0.001558061
orsf_vi_permute(fit_no_vi)
## bili copper protime albumin ascites
## 0.0546262009 0.0154588025 0.0135496100 0.0134187625
## age stage chol edema ast
## 0.0119125415 0.0114086074 0.0074840971 0.0052955192 0.0051264428
## spiders hepato sex trig alk.phos
## 0.0050509057 0.0046277553 0.0039395847 0.0024127938 0.0009590829
## platelet trt
## 0.0004366080 -0.0018115005
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 copper protime albumin ascites
## 5.354589e-02 2.608966e-02 1.286604e-02 1.250721e-02 1.206669e-02
## age stage chol edema ast
## 1.083811e-02 1.056282e-02 8.224237e-03 6.016773e-03 5.831833e-03
## spiders hepato sex trig alk.phos
## 4.736005e-03 3.446193e-03 3.440979e-03 2.277363e-03 2.224480e-03
## platelet trt
## 1.424653e-03 -1.351124e-05
You can still get negation VI from this fit, but it needs to be computed
orsf_vi_negate(fit_permute_vi)
## bili copper sex stage protime albumin
## 0.124731632 0.052319855 0.038669229 0.027453339 0.022757244 0.022220823
## age ast ascites chol spiders edema
## 0.020417138 0.013403277 0.013193815 0.011895708 0.008962837 0.007361044
## spiders hepato sex trig alk.phos platelet
## 0.006428047 0.004366080 -0.0018115005

References
orsf_vs

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

```r
object <- orsf(formula = time + status ~ ., 
                 data = pbc_orsf, 
                 n_tree = 25, 
                 importance = 'anova')
ORSF_vs(object, n_predictor_min = 17)
```

---

**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, transplantation, 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


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,
  n_thread = 1,
  verbose_progress = FALSE,
  pred_aggregate = 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.
predict.orsf_fit

• '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.

n_thread (integer) number of threads to use while computing predictions. Default is one thread. To use the maximum number of threads that your system provides for concurrent execution, set n_thread = 0.

verbose_progress (logical) if TRUE, progress messages are printed in the console. If FALSE (the default), nothing is printed.

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.

... 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:

```r
library(aorsf)
set.seed(329730)
index_train <- sample(nrow(pbc_orbf), 150)
pbc_orbf_train <- pbc_orbf[index_train, ]
pbc_orbf_test <- pbc_orbf[-index_train, ]
fit <- orsf(data = pbc_orbf_train,
formula = Surv(time, status) ~ . - id,
odbag_pred_horizon = 365.25 * 5)
```

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

```r
# predicted risk, the default
predict(fit,
new_data = pbc_orbf_test[1:5, ],
pred_type = 'risk',
pred_horizon = c(500, 1000, 1500))
## [,1] [,2] [,3]
## [1,] 0.458321864 0.72956562 0.89157462
## [2,] 0.032194868 0.08028381 0.15592011
## [3,] 0.115945485 0.24099853 0.38094684
## [4,] 0.000380033 0.02964250 0.06977315
## [5,] 0.009798295 0.01793586 0.04454374

# predicted survival, i.e., 1 - risk
predict(fit,
new_data = pbc_orbf_test[1:5, ],
pred_type = 'surv',
pred_horizon = c(500, 1000, 1500))
## [,1] [,2] [,3]
## [1,] 0.5416781 0.2704344 0.1084254
## [2,] 0.9678051 0.9197162 0.8440799
## [3,] 0.8840545 0.7500015 0.6190532
## [4,] 0.9916220 0.9703575 0.9302269
## [5,] 0.990207 0.9820641 0.9554563

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

```r
```

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

## [,1]  [,2]  [,3]
## [1,] 0.63432745 1.26771489 1.74183802
## [2,] 0.03415809 0.09124550 0.20017014
## [3,] 0.14715014 0.34375274 0.62976148
## [4,] 0.00857621 0.03195771 0.08744159
## [5,] 0.01043219 0.01888677 0.05177019

Predict mortality, defined as the number of events in the forest’s population if all observations had
c 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,] 78.575909
## [2,] 20.872849
## [3,] 37.341745
## [4,] 13.616617
## [5,]  8.798328

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

```r
object <- orsf(pbc_orsf, Surv(time, status) ~ . - id)
smry <- orsf_summarize_uni(object, n_variables = 3)
print(smry)
```
Index

* datasets
  pbc_orsf, 42
* orsf_control
  orsf_control_cph, 19
  orsf_control_custom, 20
  orsf_control_fast, 23
  orsf_control_net, 24
as.data.table.orsf_summary_uni, 2
coxph, 19

data.frame, 4, 26, 30, 43
data.table, 2, 4, 26, 27, 30, 41, 43
difftime, 35

orsf, 3, 19, 20, 23, 24, 26, 29, 33, 36, 37, 41, 43, 44, 46, 47
orsf_control_cph, 4, 7, 19, 22–24, 32
orsf_control_custom, 4, 20, 20, 23, 24
orsf_control_fast, 4, 20, 22, 23, 24
orsf_control_net, 4, 7, 20, 22, 23, 24
orsf_ice_inb(orsf_ice_oob), 25
orsf_ice_new(orsf_ice_oob), 25
orsf_ice_oob, 25
orsf_pd_inb(orsf_pd_oob), 28
orsf_pd_new(orsf_pd_oob), 28
orsf_pd_oob, 6, 28
orsf_scale_cph, 32
orsf_summarize_uni, 6, 33
orsf_time_to_train, 35
orsf_train(orsf), 3
orsf_unscale_cph(orsf_scale_cph), 32
orsf_vi, 6, 34, 36
orsf_vi_anova(orsf_vi), 36
orsf_vi_negate(orsf_vi), 36
orsf_vi_permute(orsf_vi), 36
orsf_vs, 41

pbc_orsf, 42
predict.orsf_fit, 43
print.orsf_fit, 46
print.orsf_summary_uni, 47
Surv, 4, 7
survival::coxph(), 7
survival::coxph.control, 19
survival::coxph.fit(), 7
survival::pbc, 42
tibble, 4, 26, 30, 43