Event time prediction with tidymodels

2023-04-13

In June 2022, the censored package was released. This enabled users to fit event time/survival time models using the tidymodels framework. As of this writing, there are now a total of 11 different engines that can be used with 6 different models.

This document is intended as a tutorial for using the broader tidymodels framework for event time analysis, including model tuning, evaluation, and selection.

To reproduce these results, you might need to update some package versions:

```
# Get CRAN versions of
pak::pak(c("parsnip", "censored"), ask = FALSE)

# Get GitHub versions of:
pak::pak(c("tidymodels/tune@ipcw"), ask = FALSE)
pak::pak(c("tidymodels/yardstick"), ask = FALSE)
```

An Example

We'll use the heart value data set in the joineR package (also described in this publication). There are 256 patients in the study that experienced aortic valve replacement surgery. The data has time-dependent covariates, but we will skip those to simplify the analysis here. The outcome is the time to death after surgery.

First, we'll load packages and then get the appropriate event times for the outcome (since there are multiple time points where patients were measured). Then we'll get the predictors that have the same values across the multiple time points and then merge them. We'll use the functions in the joineR package for the first two tasks:

```
library(tidymodels)
library(censored)
library(joineR)

# ------------------------------------------------------------------------------

tidymodels_prefer()
theme_set(theme_bw())
options(pillar.advice = FALSE, pillar.min_title_chars = Inf)

# ------------------------------------------------------------------------------

data(heart.valve, package = "joineR")

outcome_data <-
  UniqueVariables(heart.valve, var.col = c("fuyrs", "status"), id.col = "num")
```

```r
covar_data <-
  UniqueVariables(heart.valve,
                  var.col = c("age", "hs", "sex", "lv", "emergenc", "hc", "sten.reg.mix"),
                  id.col = "num")

heart_data <-
  full_join(outcome_data, covar_data, by = "num") %>%
  select(-num)
```

We'll reformat some of the categorical predictors since they are currently encoded as integers.

Also, tidymodels expects that the event times and corresponding status data are pre-formatted using the `Surv` function in the survival package. We'll do that, then remove the original `fuyrs` and `status` columns.

```r
heart_data <-
  heart_data %>%
  mutate(
    event_time = Surv(fuyrs, status),
    lv =
      case_when(
        lv == 1 ~ "good",
        lv == 2 ~ "moderate",
        lv == 3 ~ "poor"
      ),
    emergenc =
      case_when(
        emergenc == 0 ~ "elective",
        emergenc == 1 ~ "urgent",
        emergenc == 2 ~ "emergency"
      ),
    hc =
      case_when(
        hc == 0 ~ "absent",
        hc == 1 ~ "present_treated",
        hc == 2 ~ "present_untreated"
      ),
    sten.reg.mix =
      case_when(
        sten.reg.mix == 1 ~ "stenosis",
        sten.reg.mix == 2 ~ "regurgitation",
        sten.reg.mix == 3 ~ "mixed"
      ),
    hs =
      case_when(
        hs == "Homograft" ~ "homograft",
        TRUE ~ "stentless_porcine_tissue"
      ),
    across(where(is.character), factor)
  ) %>%
  select(-fuyrs, -status)
```

Since our focus is on prediction, the standard tidymodels methods for data splitting are used to create training and test sets. We'll also make cross-validation folds:

```
set.seed(6941)
valve_split <- initial_split(heart_data)
valve_tr <- training(valve_split)
valve_te <- testing(valve_split)
```

In the training set, the observed time values range from 0.047 years to 11 years and 19.79% of the patients died (i.e. were events).

New Prediction Types

For event time analysis, there are different types of predictions. Dynamic predictions require a specific time to make the prediction (sometimes called a "landmark time"). For example, we might want to know the probability of survival up to some time t. A static prediction is one that is not dependent on time. For example, we might predict the event time from a model.

To demonstrate, let's fit a bagged tree to the training data:

```
bag_spec <-
  bag_tree() %>%
  set_mode("censored regression") %>%
  set_engine("rpart", nbagg = 50)

set.seed(29872)
bag_fit <-
  bag_spec %>%
  fit(event_time ~ ., data = valve_tr)
```

Instead of using the training or testing sets, let's make two fake patients by randomly selecting rows from the training set:

```
set.seed(4853)
fake_examples <-
  bind_rows(
    map_dfc(valve_tr, ~ sample(.x, 1)),
    map_dfc(valve_tr, ~ sample(.x, 1))
  )
```

The standard `predict()` machinery can be used to get static (`type = "time"`) or dynamic predictions (`type = "survival"`). We'll create a grid of 101 time points for the latter:

```
time_points <- seq(0, 10, by = .1)
bag_pred <-
  predict(bag_fit, fake_examples, type = "survival", eval_time = time_points) %>%
  bind_cols(
    predict(bag_fit, fake_examples),
    fake_examples %>% select(event_time)
  ) %>%
  add_rowindex()
bag_pred
```

```
## # A tibble: 2 x 4
```

```
##   .pred                .pred_time event_time  .row
##   <list>                    <dbl>     <Surv> <int>
## 1 <tibble [101 x 2]>        5.42  3.800000+      1
## 2 <tibble [101 x 2]>        6.16  5.926027+      2
```

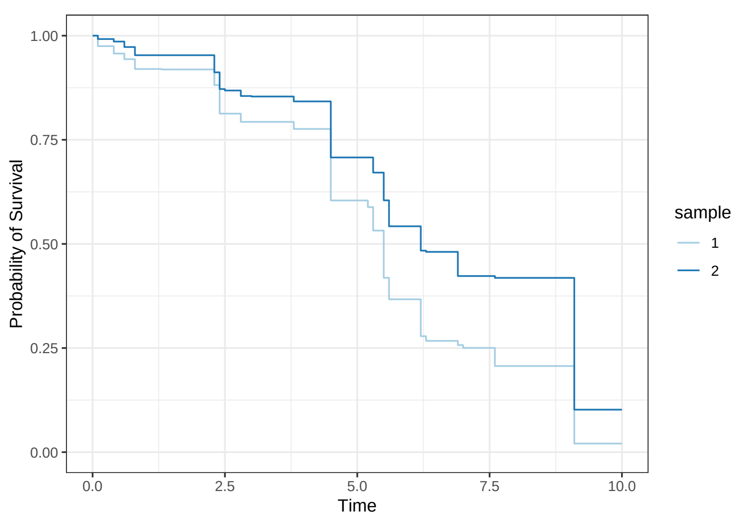
As usual, the prediction columns are prefaced with `.pred_`. What is unusual is that `.pred` is a list column, and each list element is a tibble with 2 columns and 101 rows. They contain the survival estimates for each patient:

```
bag_pred$.pred[[1]] %>% slice(1:5)
```

```
## # A tibble: 5 x 2
##   .eval_time .pred_survival
##        <dbl>          <dbl>
## 1        0            1
## 2        0.1          0.975
## 3        0.2          0.975
## 4        0.3          0.975
## 5        0.4          0.957
```

We can unnest these and plot the per-patient survival curves:

```
bag_pred %>%
  unnest(.pred) %>%
  mutate(sample = format(.row)) %>%
  ggplot(aes(.eval_time, .pred_survival, group = sample, col = sample)) +
  geom_step() +
  lims(y = 0:1) +
  labs(x = "Time", y = "Probability of Survival") +
  scale_color_brewer(palette = "Paired")
```



The static/dynamic prediction types make these models' tuning and evaluations a little more complex. In many tidymodels functions, there is a new argument called `eval_time` that is used to specify the time points for dynamic predictions (as we'll see in a minute).

Measures of Performance

Metrics to measure how well our model performs can also be split into dynamic and static metrics.

For static, a common choice is the concordance statistic, accessible via the `concordance_survival()` function. If we were looking at the test set results for the bagged tree model:

```
test_pred <-
  predict(bag_fit, valve_te, type = "survival", eval_time = time_points) %>%
  bind_cols(
    predict(bag_fit, valve_te),
    valve_te %>% select(event_time)
  )

test_pred %>%
  concordance_survival(truth = event_time, estimate = .pred_time)
```

```
## # A tibble: 1 x 3
##   .metric              .estimator .estimate
##   <chr>                <chr>          <dbl>
## 1 concordance_survival standard       0.547
```

Dynamic metrics usually are classification metrics re-purposed for survival analysis. For example, if we wanted to evaluate the model at $t = 5$, we could use the predicted survival probabilities and try to classify each data point as dead or alive. This ends up being a two class situation, and metrics like the Brier Score or the area under the ROC curve can be used to quantify how well the model works at time t.

The main difficulty is that, due to censoring, some data can't be cleanly classified. If we have a censored event at time 6, we definitely know that it should not be classified as an event. However, if the observed time were 2 and censored, we don't know if it is an event at $t = 5$ or not.

There are a lot of ways to deal with this issue. We've done an exhaustive reading of the literature and have come to a somewhat opinionated conclusion. Most of the survival metrics in the literature are developed to univariately score a collection of predictors, typically biomarkers, regarding how well they are associated with the event times. That's not what we are doing; we have model predictions.

Our choice for dynamic metrics is to use the inverse probability of censoring weights (IPCW), specifically the scheme used by Graf *et al.* (1999). They compute the probability that every data point might have been censored and uses the inverse of this value as a case weight. If the observed time is a censoring time, and that is before the evaluation time, the data point should make no contribution to the performance metric.

If you were to compute model performance manually (as above), these weights are computed using:

```
ipcw_data <-
  test_pred %>%
  .censoring_weights_graf(bag_fit, .) %>%
  select(-.pred_time)

ipcw_data
```

```
## # A tibble: 64 x 2
##    .pred                event_time
##    <list>                   <Surv>
##  1 <tibble [101 x 5]> 4.95616438+
##  2 <tibble [101 x 5]> 8.81643836+
```

```
##  3 <tibble [101 x 5]> 7.98082192+
##  4 <tibble [101 x 5]> 9.20273973+
##  5 <tibble [101 x 5]> 0.02191781+
##  6 <tibble [101 x 5]> 4.37534247+
##  7 <tibble [101 x 5]> 7.48493151
##  8 <tibble [101 x 5]> 3.20273973
##  9 <tibble [101 x 5]> 6.31780800+
## 10 <tibble [101 x 5]> 9.78630137+
## # ... with 54 more rows
```

```r
# The adjusted data:

ipcw_data$.pred[[1]] %>% slice(1:5)
```

```
## # A tibble: 5 x 5
##   .eval_time .pred_survival .weight_time .pred_censored .weight_censored
##        <dbl>          <dbl>        <dbl>          <dbl>            <dbl>
## 1        0            1            0              1                1
## 2        0.1          0.997        0.100          1                1
## 3        0.2          0.997        0.200          1                1
## 4        0.3          0.997        0.300          0.995            1.01
## 5        0.4          0.997        0.400          0.995            1.01
```

The last column is used as a case weight.

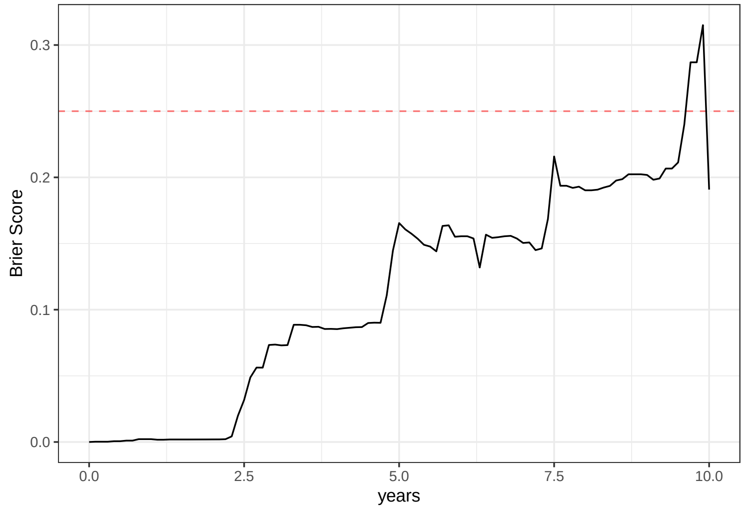
With the data in this format, a function such as `brier_survival()` can be used:

```r
brier_scores <-
  ipcw_data %>%
  # No argument name is used for .pred
  brier_survival(truth = event_time, .pred)
brier_scores %>% slice(1:5)
```

```
## # A tibble: 5 x 4
##   .metric        .estimator .eval_time .estimate
##   <chr>          <chr>           <dbl>     <dbl>
## 1 brier_survival standard          0    0
## 2 brier_survival standard          0.1  0.000207
## 3 brier_survival standard          0.2  0.000207
## 4 brier_survival standard          0.3  0.000208
## 5 brier_survival standard          0.4  0.000599
```

We compute a score for each landmark time.

```r
brier_scores %>%
  ggplot(aes(.eval_time, .estimate)) +
  geom_hline(yintercept = 0.25, col = "red", alpha = 1 / 2, lty = 2) +
  geom_line() +
  labs(x = "years", y = "Brier Score")
```



The vertical line is the level of performance that you would get with a non-informative model.

The other dynamic metrics that are currently implemented are `brier_survival_integrated()` (for an AUC of the curve above) and `roc_auc_survival()`.

Multiple combinations of static and dynamic metrics can be combined via a metric set.

Resampling the Model

tidymodels strongly focuses on empirical validation via resampling, which is also true of event time models.

We can use the `fit_resamples()` function with an rsample object to compute performance without using the test set. We need to tell the function what times to use for the dynamic metrics:

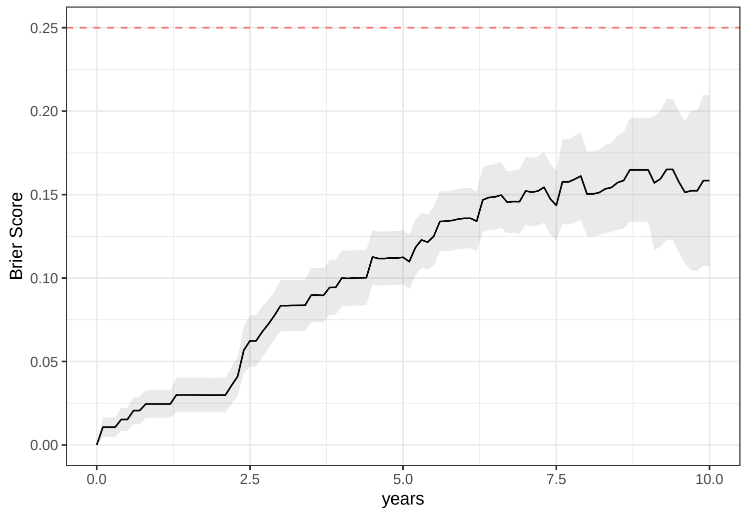
```
# Create resamples
set.seed(12)
valve_rs <- vfold_cv(valve_tr, repeats = 5)

bag_tree_res <-
  bag_spec %>%
  fit_resamples(event_time ~ ., resamples = valve_rs, eval_time = time_points)
```

By default, the Brier score is used.

```
collect_metrics(bag_tree_res) %>% slice(1:5)
```

```
## # A tibble: 5 x 7
##   .eval_time .metric        .estimator   mean     n std_err .config
##        <dbl> <chr>          <chr>       <dbl> <int>   <dbl> <chr>
## 1        0   brier_survival standard   0         50 0       Preprocessor1_Model1
## 2        0.1 brier_survival standard   0.0107    50 0.00298 Preprocessor1_Model1
## 3        0.2 brier_survival standard   0.0107    50 0.00298 Preprocessor1_Model1
## 4        0.3 brier_survival standard   0.0107    50 0.00298 Preprocessor1_Model1
## 5        0.4 brier_survival standard   0.0152    50 0.00350 Preprocessor1_Model1
```

```
bag_tree_res  %>%
  collect_metrics() %>%
  mutate(
    lower = mean - 1.96 * std_err,
    upper = mean + 1.96 * std_err
  ) %>%
  ggplot(aes(.eval_time)) +
  geom_hline(yintercept = 0.25, col = "red", alpha = 1 / 2, lty = 2) +
  geom_line(aes(y = mean)) +
  geom_ribbon(aes(ymin = lower, ymax = upper),
              col = NA,
              alpha = 1 / 10) +
  labs(x = "years", y = "Brier Score")
```



Model Tuning

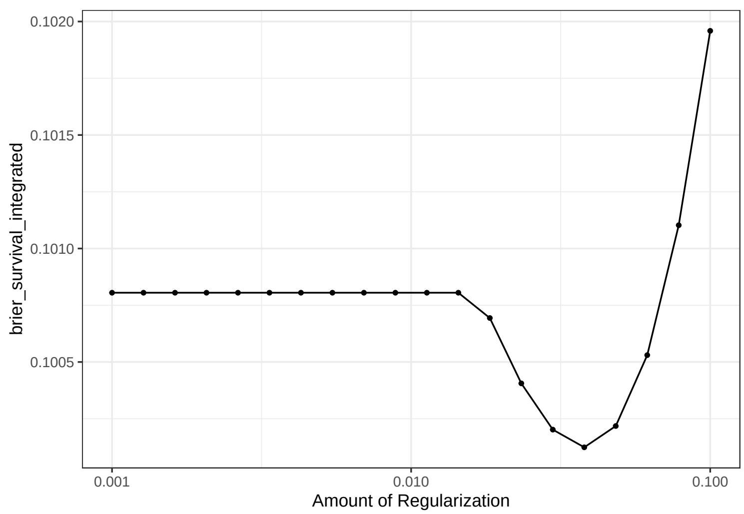
Suppose we try a regularized Cox model for these data. We'll add a recipe to the analysis and tune a lasso model. The code is pretty standard tidymodels syntax, with the added `eval_time` argument. We'll also use a metric set to include the integrated Brier score, which computes the AUC of the Brier/time curve.

```
lasso_spec <-
  proportional_hazards(penalty = tune(), mixture = 0) %>%
  set_engine("glmnet") %>%
  set_mode("censored regression")

lasso_rec <-
  recipe(event_time ~ ., data = valve_tr) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_normalize(all_numeric_predictors())

lasso_wflow <- workflow(lasso_rec, lasso_spec)

surv_metrics <- metric_set(brier_survival_integrated, brier_survival)
```

```
lasso_tune_res <-
  lasso_wflow %>%
  tune_grid(
    resamples = valve_rs,
    eval_time = time_points,
    grid = tibble(penalty = 10^seq(-3, -1, length.out = 20)),
    metrics = surv_metrics
  )
```

We can plot the results for that specific metric:

```
autoplot(lasso_tune_res, metric = "brier_survival_integrated")
```



For these plot methods, `eval_times` can be passed in as shown. If a dynamic metric is used and `eval_time` is not set, the function will pick a time near the middle of the range.

We can also choose the best penalty. If we use an integrated method, no `eval_time` is needed:

```
best_penalty <- select_best(lasso_tune_res, metric = "brier_survival_integrated")
```

Now we can update the workflow and, assuming that this is the model that we want to keep, evaluate it on the test set:

```
lasso_final_wflow <-
  lasso_wflow %>%
  finalize_workflow(best_penalty)

lasso_final_wflow
```

```
## == Workflow ====================================================================
## Preprocessor: Recipe
## Model: proportional_hazards()
##
## -- Preprocessor ----------------------------------------------------------------
## 3 Recipe Steps
```

```
##
## * step_dummy()
## * step_zv()
## * step_normalize()
##
## -- Model -----------------------------------------------------------------------
## Proportional Hazards Model Specification (censored regression)
##
## Main Arguments:
##   penalty = 0.0379269019073225
##   mixture = 0
##
## Computational engine: glmnet
```

For the test set, you can manually predict it or use `last_fit()` with the original split object:

```
test_res <-
  last_fit(lasso_final_wflow, valve_split, eval_time = time_points)
```

As usual, you can get the test set statistics via:

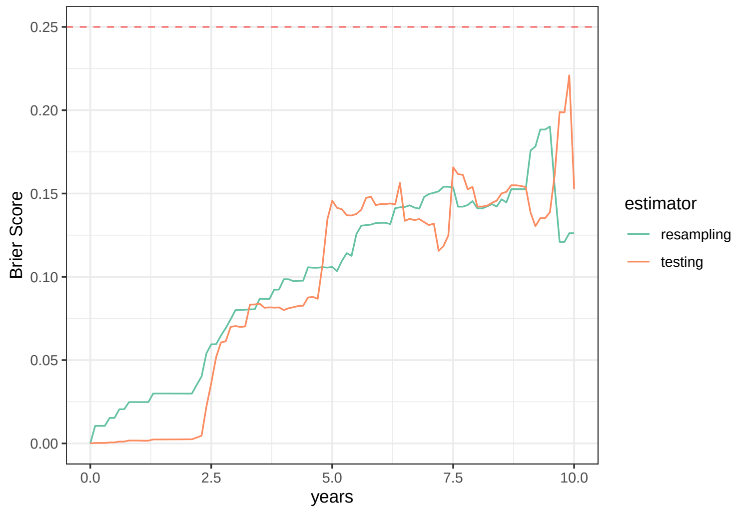
```
collect_metrics(test_res)
```

```
## # A tibble: 101 x 5
##    .metric        .estimator .eval_time .estimate .config
##    <chr>          <chr>           <dbl>     <dbl> <chr>
##  1 brier_survival standard          0    0        Preprocessor1_Model1
##  2 brier_survival standard          0.1  0.000267 Preprocessor1_Model1
##  3 brier_survival standard          0.2  0.000267 Preprocessor1_Model1
##  4 brier_survival standard          0.3  0.000269 Preprocessor1_Model1
##  5 brier_survival standard          0.4  0.000605 Preprocessor1_Model1
##  6 brier_survival standard          0.5  0.000609 Preprocessor1_Model1
##  7 brier_survival standard          0.6  0.00109  Preprocessor1_Model1
##  8 brier_survival standard          0.7  0.00110  Preprocessor1_Model1
##  9 brier_survival standard          0.8  0.00173  Preprocessor1_Model1
## 10 brier_survival standard          0.9  0.00173  Preprocessor1_Model1
## # ... with 91 more rows
```

How do the Brier Score estimates compare between the test set and resampling?

```
collect_metrics(test_res) %>%
  mutate(estimator = "testing") %>%
  select(.eval_time, estimator, Brier = .estimate) %>%
  bind_rows(
    lasso_tune_res %>%
    collect_metrics() %>%
      mutate(estimator = "resampling") %>%
      select(.eval_time, estimator, Brier = mean, penalty) %>%
      inner_join(best_penalty, by = "penalty")
  ) %>%
  ggplot(aes(.eval_time)) +
  geom_hline(yintercept = 0.25, col = "red", alpha = 1 / 2, lty = 2) +
```

```r
  geom_line(aes(y = Brier, col = estimator)) +
  labs(x = "years", y = "Brier Score") +
  scale_color_brewer(palette = "Set2")
```



Good!

Things still to do

- Update finetune to use `eval_time`
- Update Bayesian analysis methods in tidyposterior
- Update parsnip `augment()` to produce IPCW values.

Session Info

```r
sessioninfo::session_info()
```

```
## - Session info ---------------------------------------------------------------
##  setting  value
##  version  R version 4.2.0 (2022-04-22)
##  os       macOS Monterey 12.6.1
##  system   aarch64, darwin20
##  ui       X11
##  language (EN)
##  collate  en_US.UTF-8
##  ctype    en_US.UTF-8
##  tz       America/New_York
##  date     2023-04-14
##  pandoc   2.19.2 @ /Applications/RStudio.app/Contents/Resources/app/quarto/bin/tools/ (via rmarkdown)
##
## - Packages -------------------------------------------------------------------
##  package      * version    date (UTC) lib source
##  backports      1.4.1      2021-12-13 [1] CRAN (R 4.2.0)
```

```
##  broom        * 1.0.3      2023-01-25 [1] CRAN (R 4.2.0)
##  cachem         1.0.7      2023-02-24 [1] CRAN (R 4.2.0)
##  censored     * 0.1.1.9003 2023-04-12 [1] Github (tidymodels/censored@b107590)
##  class          7.3-21     2023-01-23 [1] CRAN (R 4.2.0)
##  cli            3.6.0      2023-01-09 [1] CRAN (R 4.2.0)
##  codetools      0.2-19     2023-02-01 [1] CRAN (R 4.2.0)
##  colorspace     2.1-0      2023-01-23 [1] CRAN (R 4.2.0)
##  conflicted     1.2.0      2023-02-01 [1] CRAN (R 4.2.0)
##  data.table     1.14.8     2023-02-17 [1] CRAN (R 4.2.0)
##  dials        * 1.1.0      2022-11-04 [1] CRAN (R 4.2.0)
##  DiceDesign     1.9        2021-02-13 [1] CRAN (R 4.2.0)
##  digest         0.6.31     2022-12-11 [1] CRAN (R 4.2.0)
##  doMC         * 1.3.8      2022-02-05 [1] CRAN (R 4.2.0)
##  dplyr        * 1.1.1      2023-03-22 [1] CRAN (R 4.2.0)
##  ellipsis       0.3.2      2021-04-29 [1] CRAN (R 4.2.0)
##  evaluate       0.17       2022-10-07 [1] CRAN (R 4.2.0)
##  fansi          1.0.4      2023-01-22 [1] CRAN (R 4.2.0)
##  farver         2.1.1      2022-07-06 [1] CRAN (R 4.2.0)
##  fastmap        1.1.1      2023-02-24 [1] CRAN (R 4.2.0)
##  foreach      * 1.5.2      2022-02-02 [1] CRAN (R 4.2.0)
##  furrr          0.3.1      2022-08-15 [1] CRAN (R 4.2.0)
##  future         1.31.0     2023-02-01 [1] CRAN (R 4.2.0)
##  future.apply   1.10.0     2022-11-05 [1] CRAN (R 4.2.0)
##  generics       0.1.3      2022-07-05 [1] CRAN (R 4.2.0)
##  ggplot2      * 3.4.1      2023-02-10 [1] CRAN (R 4.2.0)
##  glmnet       * 4.1-6      2022-11-27 [1] CRAN (R 4.2.0)
##  globals        0.16.2     2022-11-21 [1] CRAN (R 4.2.0)
##  glue           1.6.2      2022-02-24 [1] CRAN (R 4.2.0)
##  gower          1.0.1      2022-12-22 [1] CRAN (R 4.2.0)
##  GPfit          1.0-8      2019-02-08 [1] CRAN (R 4.2.0)
##  gtable         0.3.1      2022-09-01 [1] CRAN (R 4.2.0)
##  hardhat        1.3.0.9000 2023-04-06 [1] Github (tidymodels/hardhat@ac2dfd0)
##  htmltools      0.5.3      2022-07-18 [1] CRAN (R 4.2.0)
##  infer        * 1.0.4      2022-12-02 [1] CRAN (R 4.2.0)
##  ipred        * 0.9-13     2022-06-02 [1] CRAN (R 4.2.0)
##  iterators    * 1.0.14     2022-02-05 [1] CRAN (R 4.2.0)
##  joineR       * 1.2.8      2023-01-22 [1] CRAN (R 4.2.0)
##  knitr          1.40       2022-08-24 [1] CRAN (R 4.2.0)
##  labeling       0.4.2      2020-10-20 [1] CRAN (R 4.2.0)
##  lattice        0.20-45    2021-09-22 [1] CRAN (R 4.2.0)
##  lava           1.7.2.1    2023-02-27 [1] CRAN (R 4.2.0)
##  lhs            1.1.6      2022-12-17 [1] CRAN (R 4.2.0)
##  lifecycle      1.0.3      2022-10-07 [1] CRAN (R 4.2.0)
##  listenv        0.9.0      2022-12-16 [1] CRAN (R 4.2.0)
##  lubridate      1.9.2      2023-02-10 [1] CRAN (R 4.2.0)
##  magrittr       2.0.3      2022-03-30 [1] CRAN (R 4.2.0)
##  MASS           7.3-58.2   2023-01-23 [1] CRAN (R 4.2.0)
##  Matrix       * 1.5-3      2022-11-11 [1] CRAN (R 4.2.0)
##  memoise        2.0.1      2021-11-26 [1] CRAN (R 4.2.0)
##  modeldata    * 1.1.0      2023-01-25 [1] CRAN (R 4.2.0)
##  munsell        0.5.0      2018-06-12 [1] CRAN (R 4.2.0)
##  nlme           3.1-162    2023-01-31 [1] CRAN (R 4.2.0)
##  nnet           7.3-18     2022-09-28 [1] CRAN (R 4.2.0)
##  parallelly     1.34.0     2023-01-13 [1] CRAN (R 4.2.0)
```

```
##  parsnip      * 1.0.4.9006 2023-04-06 [1] Github (tidymodels/parsnip@865e18f)
##  pillar         1.8.1      2022-08-19 [1] CRAN (R 4.2.0)
##  pkgconfig      2.0.3      2019-09-22 [1] CRAN (R 4.2.0)
##  prodlim        2022.10.13 2023-01-12 [1] Github (tagteam/prodlim@262971a)
##  purrr        * 1.0.1      2023-01-10 [1] CRAN (R 4.2.0)
##  R6             2.5.1      2021-08-19 [1] CRAN (R 4.2.0)
##  RColorBrewer   1.1-3      2022-04-03 [1] CRAN (R 4.2.0)
##  Rcpp           1.0.10     2023-01-22 [1] CRAN (R 4.2.0)
##  recipes      * 1.0.5      2023-02-20 [1] CRAN (R 4.2.0)
##  rlang          1.1.0.9000 2023-04-06 [1] Github (r-lib/rlang@ea2fe5f)
##  rmarkdown      2.17       2022-10-07 [1] CRAN (R 4.2.0)
##  rpart          4.1.19     2022-10-21 [1] CRAN (R 4.2.0)
##  rsample      * 1.1.1      2022-12-07 [1] CRAN (R 4.2.0)
##  rstudioapi     0.14       2022-08-22 [1] CRAN (R 4.2.0)
##  scales       * 1.2.1      2022-08-20 [1] CRAN (R 4.2.0)
##  sessioninfo    1.2.2      2021-12-06 [1] CRAN (R 4.2.0)
##  shape          1.4.6      2021-05-19 [1] CRAN (R 4.2.0)
##  statmod        1.4.37     2022-08-12 [1] CRAN (R 4.2.0)
##  stringi        1.7.12     2023-01-11 [1] CRAN (R 4.2.0)
##  stringr        1.5.0      2022-12-02 [1] CRAN (R 4.2.0)
##  survival     * 3.5-3      2023-02-12 [1] CRAN (R 4.2.0)
##  tibble       * 3.2.1      2023-03-20 [1] CRAN (R 4.2.0)
##  tidymodels   * 1.0.0      2022-07-13 [1] CRAN (R 4.2.0)
##  tidyr        * 1.3.0      2023-01-24 [1] CRAN (R 4.2.0)
##  tidyselect     1.2.0      2022-10-10 [1] CRAN (R 4.2.0)
##  timechange     0.2.0      2023-01-11 [1] CRAN (R 4.2.0)
##  timeDate       4022.108   2023-01-07 [1] CRAN (R 4.2.0)
##  tune         * 1.0.1.9003 2023-04-14 [1] local
##  utf8           1.2.3      2023-01-31 [1] CRAN (R 4.2.0)
##  vctrs          0.6.1      2023-03-22 [1] CRAN (R 4.2.0)
##  withr          2.5.0      2022-03-03 [1] CRAN (R 4.2.0)
##  workflows    * 1.1.3      2023-02-22 [1] CRAN (R 4.2.0)
##  workflowsets * 1.0.0      2022-07-12 [1] CRAN (R 4.2.0)
##  xfun           0.34       2022-10-18 [1] CRAN (R 4.2.0)
##  yaml           2.3.6      2022-10-18 [1] CRAN (R 4.2.0)
##  yardstick    * 1.1.0.9001 2023-04-13 [1] Github (tidymodels/yardstick@4f1faed)
##
##  [1] /Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/library
##
## ------------------------------------------------------------------------------
```