Practical Predictive Analytics Seminar: Machine learning methods

Talex Diede

Contents

Intro

In the previous sessions we were introduced to the mortality dataset, and we saw how to do some intitial data exploration and cleaning. We looked at fitting a logistic GLM, used stepwise regression to identify significant variables, and assessed overall model fit. We also discussed how to compare candidate models. In this session we will explore the world beyond linear models, moving into machine learning methods.

Load packages

```
# install.packages("e1071", dependencies = TRUE)
library(tidyverse)
library(lubridate)
library(tidymodels)
library(doFuture)
library(vip)
library(rpart)
library(rpart.plot)
library(xgboost)
```

Import cleaned data:

```
data.large <- readRDS("PPASExpandedData.rds")
```

1. Data setup

A. Data exploration

Here we'll take another quick look at the variables in the dataset and their respective summaries. This was already demonstrated in the previous session, but will be helpful information to have around for this analysis

as well.

```
summary(data.large)
```

B. Training, holdout, and testing datasets

Now we'll split the dataset into the same subsets it was split into for the previous analysis. A 50% training sample, a 25% in-time holdout sample, and a 25% out-of-time holdout sample. If you were continuing on from the previous analysis you wouln't need to repeat this process.

```
(test.cut.date <- quantile(data.large$current.date, .75, type = 1))
```

```
            75%
    "2012-04-18"
```

```
set.seed(1)
sample.rand <- data.frame(PolNum = 1:45000,
                          RandNum = runif(45000, 0, 1))
data.large <- left_join(data.large, sample.rand)
data.large <- mutate(data.large,
                     Sample = ifelse(current.date > test.cut.date, "testing",
                                     ifelse(RandNum > 2/3, "holdout", "training")))
```

2. Model fitting

In this section we will begin fitting machine learning models to the dataset which we have loaded and divided into training and holdout samples in the prior section. To run the models in this section we first create the upper bound formula object. This will tell the algorithms what variables to consider for use in better predicting the death indicator.

A. Formula

```
data.large %>%
  filter(Sample == "training") %>%
  mutate(Death = factor(Death)) -> train_data

recipe(Death ~ ., data = train_data) %>%
  update_role(PolYear,
              timeinstudy,
              timetodeath,
              enteredstudydate,
              studyenddate,
              dob,
              dod,
              died,
              age,
              death_ind,
              timetodeath2,
              ht.wt.flag,
              bmisqrerr,
              cancer_ind,
              healthy_ind,
              years,
              finalyearfrac,
              YearFrac,
```

```
              current.date,
              RandNum,
              Sample,
              new_role = "ID") %>%
  step_string2factor(all_nominal(), -all_outcomes()) %>%
  step_normalize(all_numeric(), -all_outcomes()) %>%
  step_dummy(all_predictors(), -all_numeric()) %>%
  step_zv(all_predictors()) %>%
  step_medianimpute(all_numeric()) %>%
  step_naomit(all_predictors()) -> death_recipe

death_recipe
```

```
    Data Recipe

    Inputs:

          role #variables
            ID         21
       outcome          1
     predictor         28

    Operations:

    Factor variables from all_nominal(), -all_outcomes()
    Centering and scaling for all_numeric(), -all_outcomes()
    Dummy variables from all_predictors(), -all_numeric()
    Zero variance filter on all_predictors()
    Median Imputation for all_numeric()
    Removing rows with NA values in all_predictors()
```

```
death_recipe$var_info
```

```
    # A tibble: 50 x 4
       variable         type    role      source
       <chr>            <chr>   <chr>     <chr>
     1 PolNum           numeric predictor original
     2 PolYear          numeric ID        original
     3 timeinstudy      numeric ID        original
     4 timetodeath      numeric ID        original
     5 enteredstudydate date    ID        original
     6 studyenddate     date    ID        original
     7 dob              date    ID        original
     8 dod              date    ID        original
     9 died             numeric ID        original
    10 height           numeric predictor original
    # ... with 40 more rows
```

```
mod_frame <- death_recipe %>%
  prep() %>%
  juice()
```

B. CART

CART stands for classification and regression trees. They are a reasonably simple concept, but we focus on them here because of their utility in more advanced methods. At its core, a tree is just a sequence of yes/no questions or rules used to split the data into subgroups. Then, depending on whether you are building a classification tree or a regression tree, the result will be either a predicted class for each subgroup or a predicted continuous value for each subgroup. For this example, we will use the rpart package to fit a tree to our sample data.

The tree model will use the formula object we created previously.

```
tree_mod <- decision_tree(cost_complexity = tune(),
                tree_depth = tune(),
                min_n = tune()) %>%
  set_engine("rpart") %>%
  set_mode("classification")

par_grid <- grid_max_entropy(cost_complexity(),
                             tree_depth(),
                             min_n(range = c(50, 100)),
                             size = 10)

summary(par_grid)
```

```
     cost_complexity       tree_depth        min_n
     Min.   :0.000e+00   Min.   : 1.00   Min.   :50.0
     1st Qu.:1.900e-08   1st Qu.: 2.25   1st Qu.:53.5
     Median :1.217e-05   Median : 5.50   Median :70.5
     Mean   :3.569e-03   Mean   : 7.00   Mean   :70.9
     3rd Qu.:3.071e-03   3rd Qu.:12.25   3rd Qu.:88.5
     Max.   :2.317e-02   Max.   :15.00   Max.   :96.0
```

```
set.seed(1234)
cv_splits <- rsample::vfold_cv(train_data, v = 10, repeats = 1)

tree_workflow <-
  workflow() %>%
  add_model(tree_mod) %>%
  add_recipe(death_recipe)
tree_workflow
```

```
    == Workflow ========================================================================================
    Preprocessor: Recipe
    Model: decision_tree()

    -- Preprocessor ----------------------------------------------------------------------------------
    6 Recipe Steps

    * step_string2factor()
    * step_normalize()
    * step_dummy()
    * step_zv()
    * step_medianimpute()
    * step_naomit()

    -- Model -----------------------------------------------------------------------------------------
```

```
Decision Tree Model Specification (classification)

Main Arguments:
  cost_complexity = tune()
  tree_depth = tune()
  min_n = tune()

Computational engine: rpart
```

```
registerDoFuture()
cl <- makeCluster(10)
plan(cluster, workers = cl)

tree_res <-
  tree_workflow %>%
  tune_grid(resamples = cv_splits,
            grid = par_grid,
            control = control_grid(allow_par = T))
stopCluster(cl)
rm(cl)
gc()
```

```
             used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells  2473322 132.1    4659575 248.9  4659575 248.9
Vcells 34715661 264.9   62101071 473.8 51684066 394.4
```

```
tree_res %>%
  show_best()
```

```
# A tibble: 5 x 9
  cost_complexity tree_depth min_n .metric .estimator  mean     n std_err
            <dbl>      <int> <int> <chr>   <chr>      <dbl> <int>   <dbl>
1  0.0000000170           13    52 roc_auc binary     0.631    10 0.00909
2  0.000115               10    53 roc_auc binary     0.597    10 0.0168
3  0.000000000280          4    77 roc_auc binary     0.550    10 0.0207
4  0.0000000262            7    92 roc_auc binary     0.535    10 0.0179
5  0.00000000498           2    50 roc_auc binary     0.5      10 0
# ... with 1 more variable: .config <chr>
```

```
best <- tree_res %>%
  select_best()

final_wf <-
  tree_workflow %>%
  finalize_workflow(best)

final_wf
```

```
== Workflow ====================================================================================
Preprocessor: Recipe
Model: decision_tree()

-- Preprocessor ----------------------------------------------------------------------------
6 Recipe Steps

* step_string2factor()
```

```
    * step_normalize()
    * step_dummy()
    * step_zv()
    * step_medianimpute()
    * step_naomit()

    -- Model ---------------------------------------------------------------------------------------
    Decision Tree Model Specification (classification)

    Main Arguments:
      cost_complexity = 1.69884143757972e-08
      tree_depth = 13
      min_n = 52

    Computational engine: rpart
```

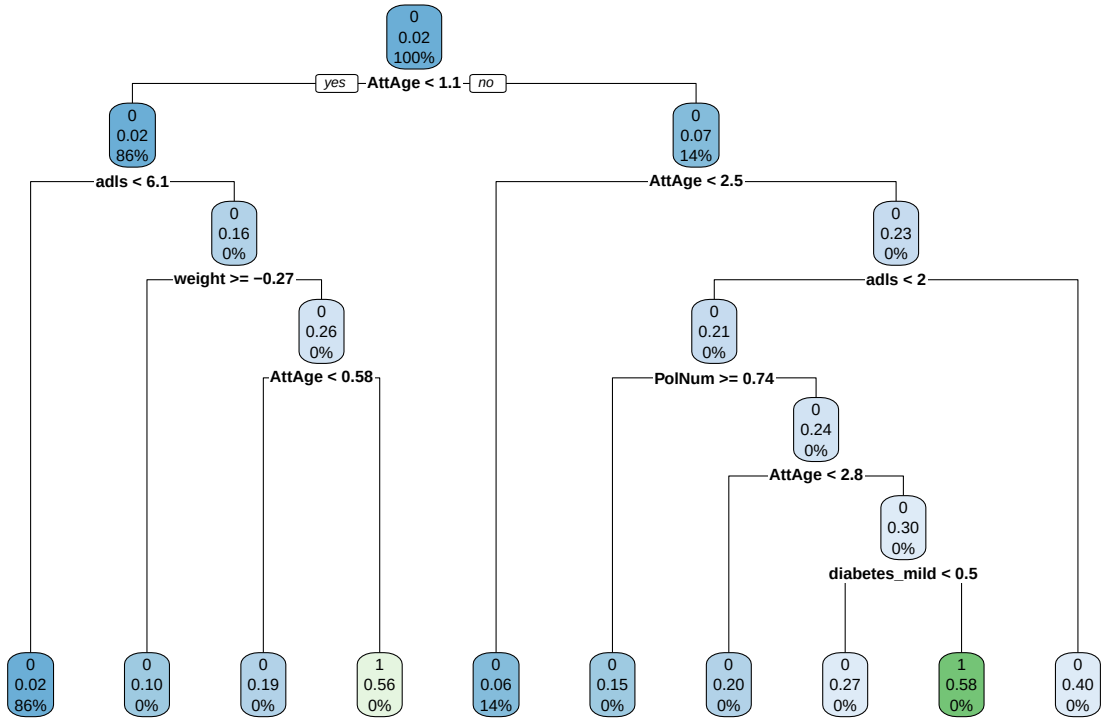
```
final_mod_rpart <-
  final_wf %>%
  fit(data = train_data)
```

```
final_mod_rpart
```

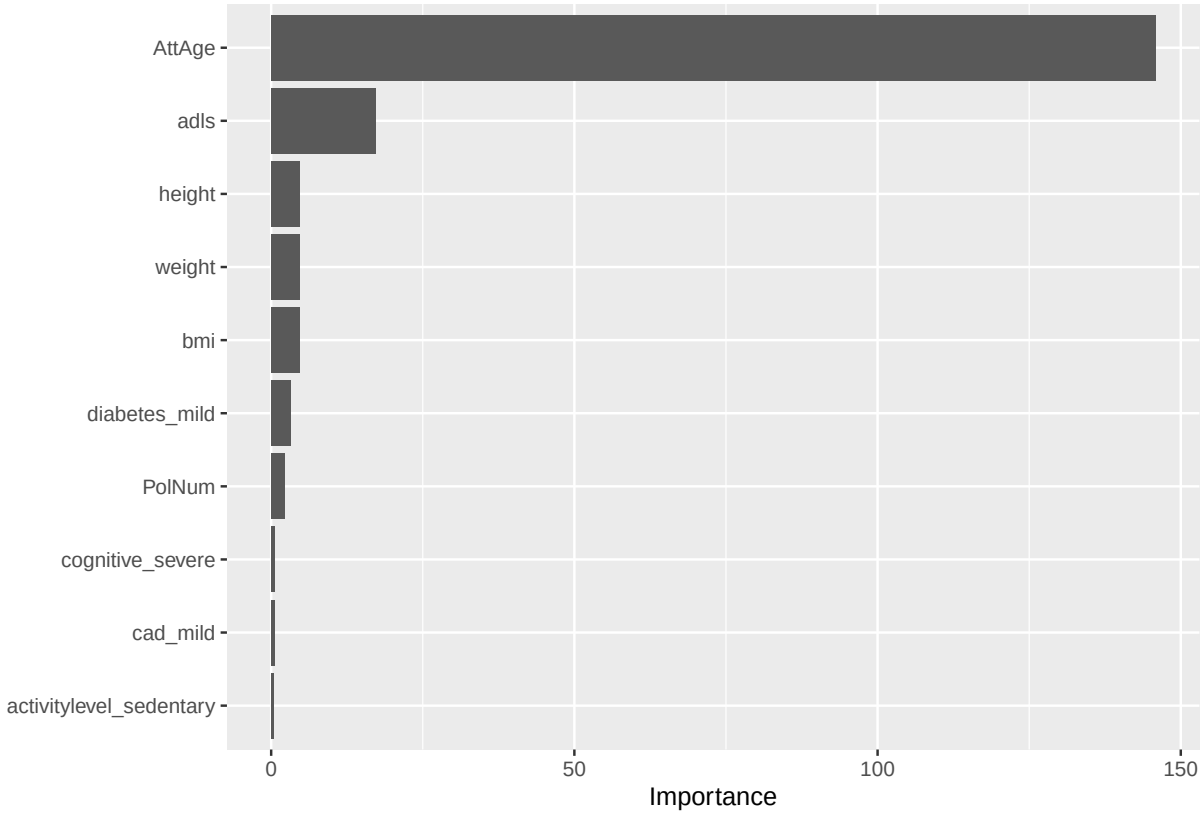
```
    == Workflow [trained] ===============================================================================
    Preprocessor: Recipe
    Model: decision_tree()

    -- Preprocessor ---------------------------------------------------------------------------------
    6 Recipe Steps

    * step_string2factor()
    * step_normalize()
    * step_dummy()
    * step_zv()
    * step_medianimpute()
    * step_naomit()

    -- Model ----------------------------------------------------------------------------------------
    n= 172949

    node), split, n, loss, yval, (yprob)
          * denotes terminal node

      1) root 172949 4144 0 (0.97603918 0.02396082)
        2) AttAge< 1.09973 148755 2531 0 (0.98298545 0.01701455)
          4) adls< 6.056475 148418 2477 0 (0.98331065 0.01668935) *
          5) adls>=6.056475 337   54 0 (0.83976261 0.16023739)
           10) weight>=-0.2670109 212   21 0 (0.90094340 0.09905660) *
           11) weight< -0.2670109 125   33 0 (0.73600000 0.26400000)
             22) AttAge< 0.5815455 100   19 0 (0.81000000 0.19000000) *
             23) AttAge>=0.5815455 25   11 1 (0.44000000 0.56000000) *
        3) AttAge>=1.09973 24194 1613 0 (0.93333058 0.06666942)
          6) AttAge< 2.508973 23531 1462 0 (0.93786919 0.06213081) *
          7) AttAge>=2.508973 663  151 0 (0.77224736 0.22775264)
           14) adls< 1.963958 615  132 0 (0.78536585 0.21463415)
             28) PolNum>=0.7420889 149   22 0 (0.85234899 0.14765101) *
```

```
      29) PolNum< 0.7420889 466  110 0 (0.76394850 0.23605150)
        58) AttAge< 2.757269 305   61 0 (0.80000000 0.20000000) *
        59) AttAge>=2.757269 161   49 0 (0.69565217 0.30434783)
         118) diabetes_mild< 0.5 142   38 0 (0.73239437 0.26760563) *
         119) diabetes_mild>=0.5 19    8 1 (0.42105263 0.57894737) *
      15) adls>=1.963958 48   19 0 (0.60416667 0.39583333) *
```

```
rpart.plot(final_mod_rpart$fit$fit$fit)
```




```
final_mod_rpart %>%
  pull_workflow_fit() %>%
  vip()
```



C. Ensemble (GBM Model)

Ensemble models are combinations of models. In an ensemble model, we run two or more related but different models and aggregate those results into a single prediction. This is done to improve the accuracy of predictions and stability of the model. They are easier to overfit, so proper care should be spent validating the resulting model predictions.

i. GBM model fit This ensemble example uses a gradient boosted machine (GBM) algorithm. We will use the gbm package to fit the model. The full model formula used is the same as was used for the CART example, GBMs allow for a bernoulli distribution similar to a GLM. This algorithm has two major input parameters, n.trees and shrinkage. We have set these to be something reasonable for this example, but they should both be optimized and validated when creating a final model for use.

```
set.seed(17)
xg_mod <-
  boost_tree(trees = tune(),
             tree_depth = tune(),
             min_n = 100,
             loss_reduction = tune(),
             learn_rate = tune(),
             mtry = tune(),
             sample_size = tune()) %>%
  set_engine("xgboost", nthread = 2) %>%
  set_mode("classification")

xg_reg_grid <- grid_max_entropy(trees(),
                                tree_depth(),
```

```
                          loss_reduction(),
                          learn_rate(),
                          finalize(mtry(), mod_frame),
                          sample_size = sample_prop(),
                          size = 10)
summary(xg_reg_grid)
```

```
      trees         tree_depth     loss_reduction        learn_rate
 Min.   : 212   Min.   : 4.00   Min.   : 0.000000   Min.   :0.000e+00
 1st Qu.: 804   1st Qu.: 5.50   1st Qu.: 0.000000   1st Qu.:1.419e-05
 Median :1354   Median : 8.00   Median : 0.000179   Median :1.930e-03
 Mean   :1214   Mean   : 8.80   Mean   : 2.398180   Mean   :1.015e-02
 3rd Qu.:1577   3rd Qu.:11.75   3rd Qu.: 0.073461   3rd Qu.:1.755e-02
 Max.   :1939   Max.   :14.00   Max.   :18.326048   Max.   :4.432e-02
      mtry        sample_size
 Min.   : 7.00   Min.   :0.1150
 1st Qu.:16.25   1st Qu.:0.4154
 Median :29.50   Median :0.5190
 Mean   :33.50   Mean   :0.5621
 3rd Qu.:52.75   3rd Qu.:0.6860
 Max.   :60.00   Max.   :0.9504
```

```
set.seed(1234)
cv_splits <- rsample::vfold_cv(train_data, v = 10, repeats = 1)

xg_workflow <-
  workflow() %>%
  add_model(xg_mod) %>%
  add_recipe(death_recipe)
xg_workflow
```

```
== Workflow ====================================================================================
Preprocessor: Recipe
Model: boost_tree()

-- Preprocessor ----------------------------------------------------------------------------
6 Recipe Steps

* step_string2factor()
* step_normalize()
* step_dummy()
* step_zv()
* step_medianimpute()
* step_naomit()

-- Model -----------------------------------------------------------------------------------
Boosted Tree Model Specification (classification)

Main Arguments:
  mtry = tune()
  trees = tune()
  min_n = 100
  tree_depth = tune()
  learn_rate = tune()
```

```
  loss_reduction = tune()
  sample_size = tune()

Engine-Specific Arguments:
  nthread = 2

Computational engine: xgboost
```

ii. Parameter tuning In this section we will present how to use the caret package to assist in tuning model parameters. We are using it to tune our GBM parameters but the package generalizes to many other model types as well. In this chunk of code we set up the grid of possible parameters that will be considered when tuning our model. We'll also initialize the parallel backend that will be used for the tuning process.

Note: use at least 1 fewer threads in your cluster than your computer has available, e.g. on a dual-core machine you have 4 threads so don't set the number of cores higher than 3.

```
registerDoFuture()
cl <- makeCluster(10)
plan(cluster, workers = cl)

xg_res <-
  xg_workflow %>%
  tune_grid(resamples = cv_splits,
            grid = xg_reg_grid,
            control = control_grid(allow_par = T))
stopCluster(cl)
rm(cl)
gc()
```

```
             used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells  2652428 141.7    4659575 248.9  4659575 248.9
Vcells 46894176 357.8   74601285 569.2 74557832 568.9
```

In the following chunk of code we will run the train function from the caret package to optimize the tuning parameters for our model. In this case we have set it up to select the best model based on AUC, but there are other options depending on the type of regression you are performing. This process will use cross-validation to determine the best model, and the number of folds for the cross-validation is set to be 5. It's also interesting that this step requires the outcomes to not be 0 and 1 as the typical gbm function prefers, so we manipulate the outcome variable to have character values.

Important note: This is a long process, don't start running this on your local machine unless you're prepared for it to take hours

```
xg_res %>%
  show_best()
```

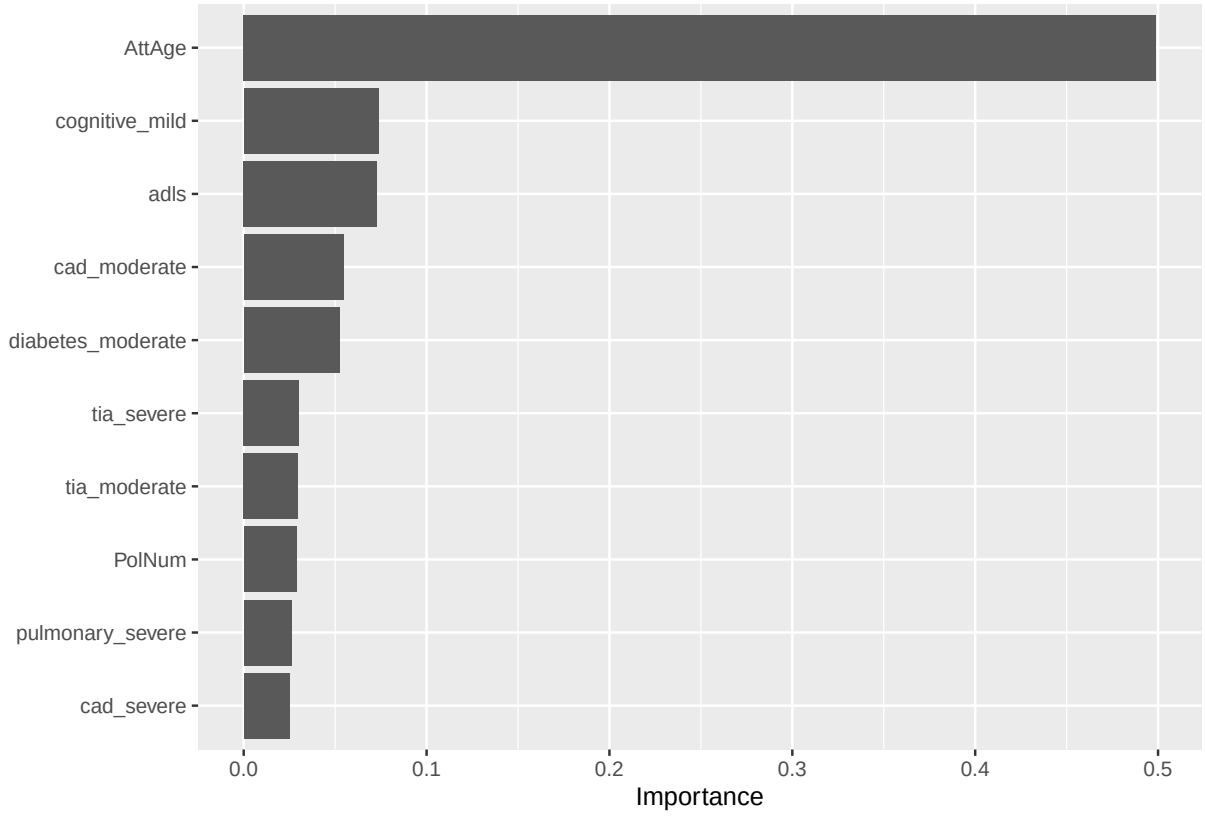
```
# A tibble: 5 x 12
   mtry trees tree_depth learn_rate loss_reduction sample_size .metric
  <int> <int>      <int>      <dbl>          <dbl>       <dbl> <chr>
1     7  1584          4  0.00176      0.0908            0.695 roc_auc
2    27   672          5  0.00210      0.000000140       0.925 roc_auc
3    12  1557         12  0.0138       0.000000487       0.388 roc_auc
4    32  1524         14  0.0000351    0.000347          0.353 roc_auc
5    53   212         14  0.0207       0.0000101         0.496 roc_auc
# ... with 5 more variables: .estimator <chr>, mean <dbl>, n <int>,
#   std_err <dbl>, .config <chr>
```

```
best <-
  xg_res %>%
  select_best()

final_wf <-
  xg_workflow %>%
  finalize_workflow(best)

final_wf
```

```
    == Workflow ========================================================================================
    Preprocessor: Recipe
    Model: boost_tree()

    -- Preprocessor ----------------------------------------------------------------------------------
    6 Recipe Steps

    * step_string2factor()
    * step_normalize()
    * step_dummy()
    * step_zv()
    * step_medianimpute()
    * step_naomit()

    -- Model -----------------------------------------------------------------------------------------
    Boosted Tree Model Specification (classification)

    Main Arguments:
      mtry = 7
      trees = 1584
      min_n = 100
      tree_depth = 4
      learn_rate = 0.00176018861030504
      loss_reduction = 0.0907701166357564
      sample_size = 0.694942815462127

    Engine-Specific Arguments:
      nthread = 2

    Computational engine: xgboost
```

```
final_mod <-
  final_wf %>%
  fit(data = train_data)
```

```
final_mod %>%
  pull_workflow_fit() %>%
  vip()
```

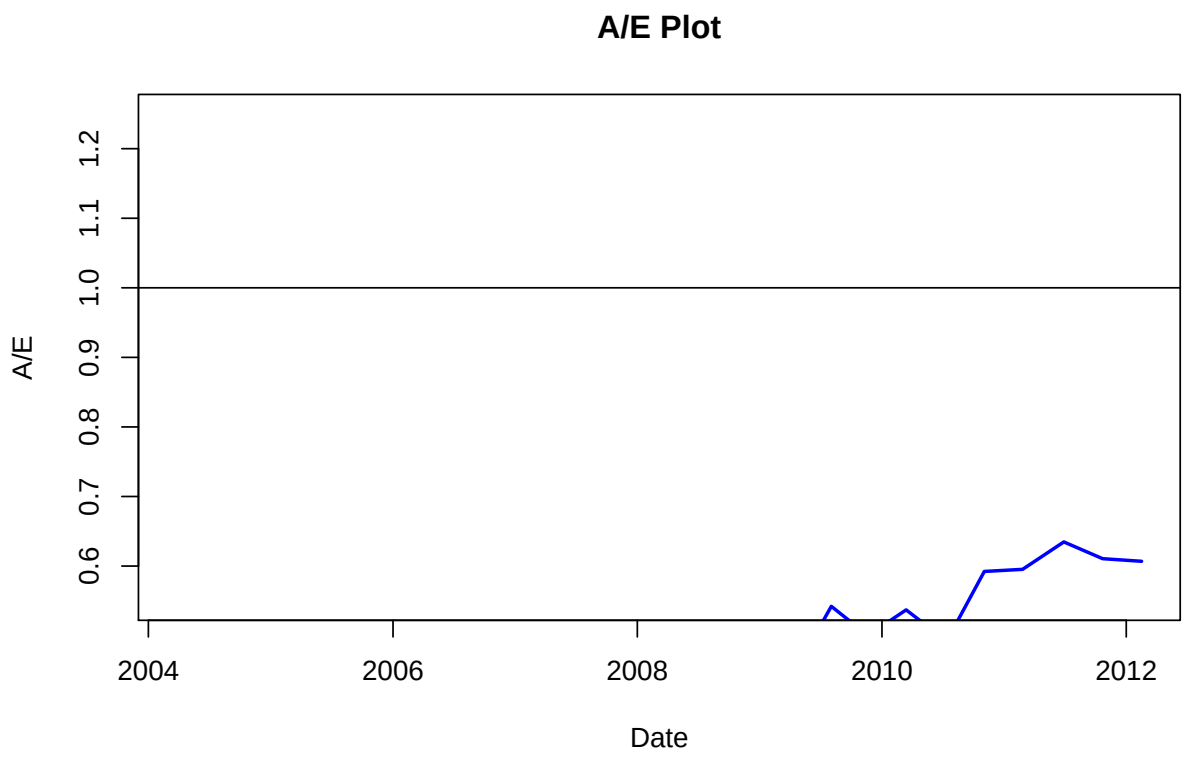


Assess and compare models

Now we may want to bring the model validation skills that we've learned to these machine learning models. So we start by revisiting an A/E plot by date to discover that our machine learning model is not capturing a factor of date. This may indicate a place that we would want to revisit in our model building process.

```
probs <-
  predict(final_mod,
          type = "prob",
          new_data = data.large) %>%
  bind_cols(data.large)


date.plotdata <- filter(probs, Sample == "training") %>%
  group_by(Date.bin = ntile(current.date, 20)) %>%
  summarize(Date = mean(current.date),
            AE.date = mean(as.numeric(as.character(Death)))/mean(.pred_1))

plot(date.plotdata$Date, date.plotdata$AE.date,
     pch = ".", cex = 0,
     main = "A/E Plot",
     xlab = "Date",
     ylab = "A/E",
     ylim = c(0.55, 1.25),
     type = "l",
     lwd = 2,
     col = "blue")
```

```
abline(h = 1)
```

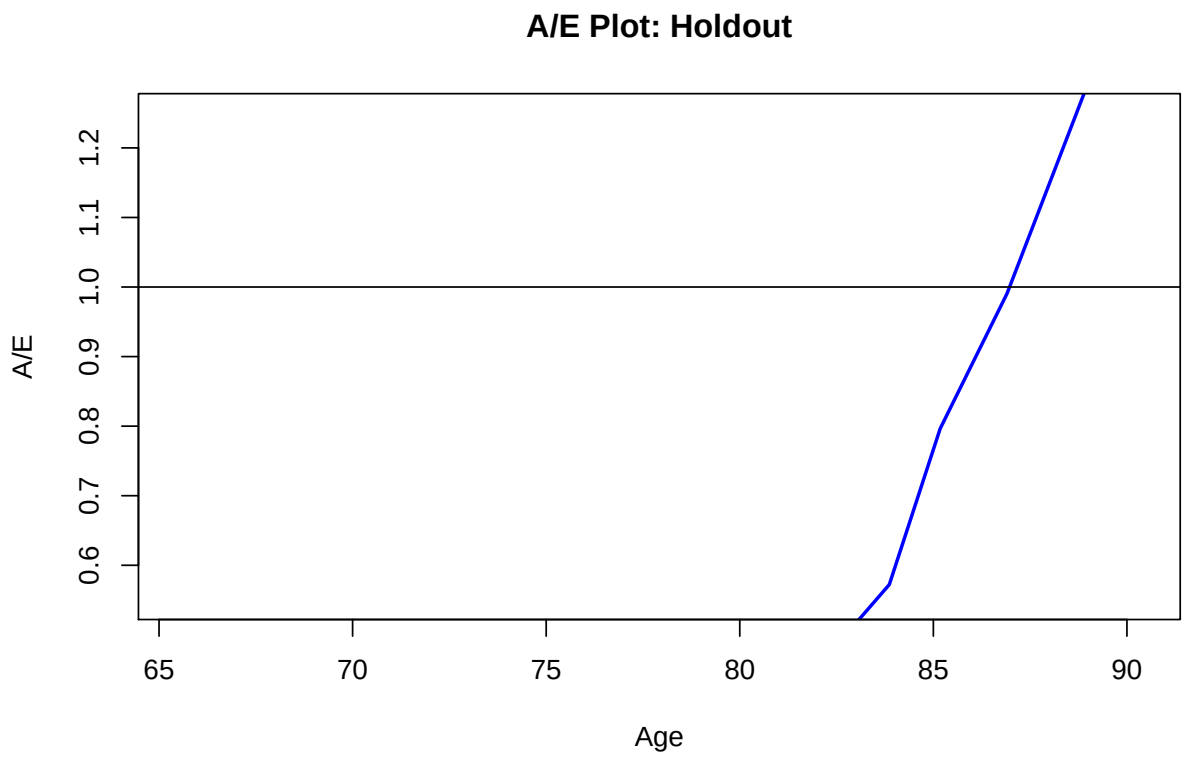


The plot below shows the A/E chart for attained age on the training data, we can see that the model performs quite well for most ages above about 75.

```
age.plotdata <- filter(probs, Sample == "training") %>%
  group_by(age.bin = ntile(AttAge, 20)) %>%
  summarize(Age = mean(AttAge),
            AE.date = mean(as.numeric(as.character(Death)))/mean(.pred_1))

plot(age.plotdata$Age, age.plotdata$AE.date,
     pch = ".", cex = 0,
     main = "A/E Plot: Training",
     xlab = "Age",
     ylab = "A/E",
     ylim = c(0.55, 1.25),
     type = "l",
     lwd = 2,
     col = "blue")
abline(h = 1)
```

Below we consider the same view as the above plot but on the holdout dataset, to check for overfitting issues on the attained age variable. It's predictably less tight around 1 but is still centered about 1 without too much variance or definable pattern.

```
age.plotdata <- filter(probs, Sample == "holdout") %>%
  group_by(age.bin = ntile(AttAge, 20)) %>%
  summarize(Age = mean(AttAge),
            AE.date = sum(Death)/sum(.pred_1))

plot(age.plotdata$Age, age.plotdata$AE.date,
     pch = ".", cex = 0,
     main = "A/E Plot: Holdout",
     xlab = "Age",
     ylab = "A/E",
     ylim = c(0.55, 1.25),
     type = "l",
     lwd = 2,
     col = "blue")
abline(h = 1)
```



Conclusion

The material included here only scratches the surface of all that is considered machine learning. Even for the algorithms demonstrated here, this isn't the end of the road. These models haven't been validated or fully parameterized, so they are not to be taken as "best" or even necessarily good models for this dataset. More information on these packages, and more that perform similar algorithms, can be found on CRAN. There are also many good Coursera and other MOOC courses that provide more in-depth training on maching learning algorithms.