vb\_matches <- readr::read\_csv('<https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2020/2020-05-19/vb_matches.csv>', guess\_max = 76000)

vb\_matches

## # A tibble: 76,756 x 65

## circuit tournament country year date gender match\_num w\_player1

##

## 1 AVP Huntingto… United… 2002 2002-05-24 M 1 Kevin Wo…

## 2 AVP Huntingto… United… 2002 2002-05-24 M 2 Brad Tor…

## 3 AVP Huntingto… United… 2002 2002-05-24 M 3 Eduardo …

## 4 AVP Huntingto… United… 2002 2002-05-24 M 4 Brent Do…

## 5 AVP Huntingto… United… 2002 2002-05-24 M 5 Albert H…

## 6 AVP Huntingto… United… 2002 2002-05-24 M 6 Jason Ri…

## 7 AVP Huntingto… United… 2002 2002-05-24 M 7 Aaron Bo…

## 8 AVP Huntingto… United… 2002 2002-05-24 M 8 Canyon C…

## 9 AVP Huntingto… United… 2002 2002-05-24 M 9 Dax Hold…

## 10 AVP Huntingto… United… 2002 2002-05-24 M 10 Mark Wil…

## # … with 76,746 more rows, and 57 more variables: w\_p1\_birthdate ,

## # w\_p1\_age , w\_p1\_hgt , w\_p1\_country , w\_player2 ,

## # w\_p2\_birthdate , w\_p2\_age , w\_p2\_hgt , w\_p2\_country ,

## # w\_rank , l\_player1 , l\_p1\_birthdate , l\_p1\_age ,

## # l\_p1\_hgt , l\_p1\_country , l\_player2 , l\_p2\_birthdate ,

## # l\_p2\_age , l\_p2\_hgt , l\_p2\_country , l\_rank ,

## # score , duration , bracket , round ,

## # w\_p1\_tot\_attacks , w\_p1\_tot\_kills , w\_p1\_tot\_errors ,

## # w\_p1\_tot\_hitpct , w\_p1\_tot\_aces , w\_p1\_tot\_serve\_errors ,

## # w\_p1\_tot\_blocks , w\_p1\_tot\_digs , w\_p2\_tot\_attacks ,

## # w\_p2\_tot\_kills , w\_p2\_tot\_errors , w\_p2\_tot\_hitpct ,

## # w\_p2\_tot\_aces , w\_p2\_tot\_serve\_errors , w\_p2\_tot\_blocks ,

## # w\_p2\_tot\_digs , l\_p1\_tot\_attacks , l\_p1\_tot\_kills ,

## # l\_p1\_tot\_errors , l\_p1\_tot\_hitpct , l\_p1\_tot\_aces ,

## # l\_p1\_tot\_serve\_errors , l\_p1\_tot\_blocks , l\_p1\_tot\_digs ,

## # l\_p2\_tot\_attacks , l\_p2\_tot\_kills , l\_p2\_tot\_errors ,

## # l\_p2\_tot\_hitpct , l\_p2\_tot\_aces , l\_p2\_tot\_serve\_errors ,

## # l\_p2\_tot\_blocks , l\_p2\_tot\_digs

This dataset has the match stats like serve errors, kills, and so forth divided out by the two players for each team, but we want those combined together because we are going to make a prediction **per team** (i.e. what makes a team more likely to win). Let’s include predictors like gender, circuit, and year in our model along with the per-match statistics. Let’s omit matches with NA values because we don’t have all kinds of statistics measured for all matches.

vb\_parsed <- vb\_matches %>%

transmute(

circuit,

gender,

year,

w\_attacks = w\_p1\_tot\_attacks + w\_p2\_tot\_attacks,

w\_kills = w\_p1\_tot\_kills + w\_p2\_tot\_kills,

w\_errors = w\_p1\_tot\_errors + w\_p2\_tot\_errors,

w\_aces = w\_p1\_tot\_aces + w\_p2\_tot\_aces,

w\_serve\_errors = w\_p1\_tot\_serve\_errors + w\_p2\_tot\_serve\_errors,

w\_blocks = w\_p1\_tot\_blocks + w\_p2\_tot\_blocks,

w\_digs = w\_p1\_tot\_digs + w\_p2\_tot\_digs,

l\_attacks = l\_p1\_tot\_attacks + l\_p2\_tot\_attacks,

l\_kills = l\_p1\_tot\_kills + l\_p2\_tot\_kills,

l\_errors = l\_p1\_tot\_errors + l\_p2\_tot\_errors,

l\_aces = l\_p1\_tot\_aces + l\_p2\_tot\_aces,

l\_serve\_errors = l\_p1\_tot\_serve\_errors + l\_p2\_tot\_serve\_errors,

l\_blocks = l\_p1\_tot\_blocks + l\_p2\_tot\_blocks,

l\_digs = l\_p1\_tot\_digs + l\_p2\_tot\_digs

) %>%

na.omit()

Still plenty of data! Next, let’s create separate dataframes for the winners and losers of each match, and then bind them together. I am using functions like rename\_with() from the  
[upcoming dplyr 1.0 release here](https://www.tidyverse.org/blog/2020/05/dplyr-1-0-0-last-minute-additions/).

winners <- vb\_parsed %>%

select(circuit, gender, year,

w\_attacks:w\_digs) %>%

rename\_with(~ str\_remove\_all(., "w\_"), w\_attacks:w\_digs) %>%

mutate(win = "win")

losers <- vb\_parsed %>%

select(circuit, gender, year,

l\_attacks:l\_digs) %>%

rename\_with(~ str\_remove\_all(., "l\_"), l\_attacks:l\_digs) %>%

mutate(win = "lose")

vb\_df <- bind\_rows(winners, losers) %>%

mutate\_if(is.character, factor)

Exploratory data analysis is always important before modeling. Let’s make one plot to explore the relationships in this data.

vb\_df %>%

pivot\_longer(attacks:digs, names\_to = "stat", values\_to = "value") %>%

ggplot(aes(gender, value, fill = win, color = win)) +

geom\_boxplot(alpha = 0.4) +

facet\_wrap(~stat, scales = "free\_y", nrow = 2) +

labs(y = NULL, color = NULL, fill = NULL)

We can see differences in errors and blocks especially.

**Build a model**

We can start by loading the tidymodels metapackage, and splitting our data into training and testing sets.

library(tidymodels)

set.seed(123)

vb\_split <- initial\_split(vb\_df, strata = win)

vb\_train <- training(vb\_split)

vb\_test <- testing(vb\_split)

An XGBoost model is based on trees, so we don’t need to do much preprocessing for our data; we don’t need to worry about the factors or centering or scaling our data. Let’s just go straight to setting up our model specification. Sounds great, right? On the other hand, we are going to tune **a lot** of model hyperparameters.

xgb\_spec <- boost\_tree(

trees = 1000,

tree\_depth = tune(), min\_n = tune(),

loss\_reduction = tune(), ## first three: model complexity

sample\_size = tune(), mtry = tune(), ## randomness

learn\_rate = tune(), ## step size

) %>%

set\_engine("xgboost") %>%

set\_mode("classification")

xgb\_spec

## Boosted Tree Model Specification (classification)

##

## Main Arguments:

## mtry = tune()

## trees = 1000

## min\_n = tune()

## tree\_depth = tune()

## learn\_rate = tune()

## loss\_reduction = tune()

## sample\_size = tune()

##

## Computational engine: xgboost

YIKES. Well, let’s set up possible values for these hyperparameters to try. Let’s use a space-filling design so we can cover the hyperparameter space as well as possible.

xgb\_grid <- grid\_latin\_hypercube(

tree\_depth(),

min\_n(),

loss\_reduction(),

sample\_size = sample\_prop(),

finalize(mtry(), vb\_train),

learn\_rate(),

size = 30

)

xgb\_grid

## # A tibble: 30 x 6

## tree\_depth min\_n loss\_reduction sample\_size mtry learn\_rate

##

## 1 13 9 0.000000191 0.488 6 0.000147

## 2 4 17 0.0000121 0.661 10 0.00000000287

## 3 7 18 0.0000432 0.151 2 0.0713

## 4 12 22 0.00000259 0.298 8 0.0000759

## 5 10 35 16.1 0.676 6 0.00000000111

## 6 4 21 0.673 0.957 7 0.00000000786

## 7 7 25 0.244 0.384 9 0.0000000469

## 8 7 3 8.48 0.775 6 0.000000555

## 9 6 8 0.0000915 0.522 6 0.00000106

## 10 11 37 0.00000109 0.886 9 0.0000000136

## # … with 20 more rows

Notice that we had to treat mtry() differently because it depends on the actual number of predictors in the data.

Let’s put the model specification into a workflow for convenience. Since we don’t have any complicated data preprocessing, we can use add\_formula() as our data preprocessor.

xgb\_wf <- workflow() %>%

add\_formula(win ~ .) %>%

add\_model(xgb\_spec)

xgb\_wf

## ══ Workflow ═══════════════════════════════════════════════════════════

## Preprocessor: Formula

## Model: boost\_tree()

##

## ── Preprocessor ───────────────────────────────────────────────────────

## win ~ .

##

## ── Model ──────────────────────────────────────────────────────────────

## Boosted Tree Model Specification (classification)

##

## Main Arguments:

## mtry = tune()

## trees = 1000

## min\_n = tune()

## tree\_depth = tune()

## learn\_rate = tune()

## loss\_reduction = tune()

## sample\_size = tune()

##

## Computational engine: xgboost

Next, let’s create cross-validation resamples for tuning our model.

set.seed(123)

vb\_folds <- vfold\_cv(vb\_train, strata = win)

vb\_folds

## # 10-fold cross-validation using stratification

## # A tibble: 10 x 2

## splits id

##

## 1 Fold01

## 2 Fold02

## 3 Fold03

## 4 Fold04

## 5 Fold05

## 6 Fold06

## 7 Fold07

## 8 Fold08

## 9 Fold09

## 10 Fold10

IT’S TIME TO TUNE. We use tune\_grid() with our tuneable workflow, our resamples, and our grid of parameters to try. Let’s use control\_grid(save\_pred = TRUE) so we can explore the predictions afterwards.

doParallel::registerDoParallel()

set.seed(234)

xgb\_res <- tune\_grid(

xgb\_wf,

resamples = vb\_folds,

grid = xgb\_grid,

control = control\_grid(save\_pred = TRUE)

)

xgb\_res

## # 10-fold cross-validation using stratification

## # A tibble: 10 x 5

## splits id .metrics .notes .predictions

##

## 1

This takes a while to finish on my computer (and makes my fans run!) but we did it. 

**Explore results**

We can explore the metrics for all these models.

collect\_metrics(xgb\_res)

## # A tibble: 60 x 11

## mtry min\_n tree\_depth learn\_rate loss\_reduction sample\_size .metric

##

## 1 1 23 9 1.64e-3 0.00000854 0.117 accura…

## 2 1 23 9 1.64e-3 0.00000854 0.117 roc\_auc

## 3 2 18 7 7.13e-2 0.0000432 0.151 accura…

## 4 2 18 7 7.13e-2 0.0000432 0.151 roc\_auc

## 5 2 32 3 1.30e-7 1.16 0.497 accura…

## 6 2 32 3 1.30e-7 1.16 0.497 roc\_auc

## 7 2 40 10 3.31e-4 0.0000000486 0.429 accura…

## 8 2 40 10 3.31e-4 0.0000000486 0.429 roc\_auc

## 9 3 5 14 3.56e-3 0.122 0.701 accura…

## 10 3 5 14 3.56e-3 0.122 0.701 roc\_auc

## # … with 50 more rows, and 4 more variables: .estimator , mean ,

## # n , std\_err

We can also use visualization to understand our results.

xgb\_res %>%

collect\_metrics() %>%

filter(.metric == "roc\_auc") %>%

select(mean, mtry:sample\_size) %>%

pivot\_longer(mtry:sample\_size,

values\_to = "value",

names\_to = "parameter"

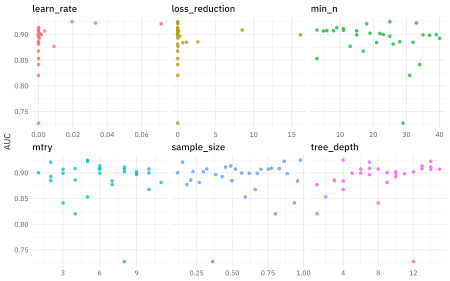
) %>%

ggplot(aes(value, mean, color = parameter)) +

geom\_point(alpha = 0.8, show.legend = FALSE) +

facet\_wrap(~parameter, scales = "free\_x") +

labs(x = NULL, y = "AUC")



Remember that we used a space-filling design for the parameters to try. It looks like higher values for tree depth were better, but other than that, the main thing I take away from this plot is that there are several combinations of parameters that perform well.

What are the best performing sets of parameters?

show\_best(xgb\_res, "roc\_auc")

## # A tibble: 5 x 11

## mtry min\_n tree\_depth learn\_rate loss\_reduction sample\_size .metric

##

## 1 5 25 4 0.0195 0.00112 0.977 roc\_auc

## 2 5 33 14 0.0332 0.0000000159 0.864 roc\_auc

## 3 2 18 7 0.0713 0.0000432 0.151 roc\_auc

## 4 6 9 13 0.000147 0.000000191 0.488 roc\_auc

## 5 8 11 14 0.0000135 0.000570 0.453 roc\_auc

## # … with 4 more variables: .estimator , mean , n , std\_err

There may have been lots of parameters, but we were able to get good performance with several different combinations. Let’s choose the best one.

best\_auc <- select\_best(xgb\_res, "roc\_auc")

best\_auc

## # A tibble: 1 x 6

## mtry min\_n tree\_depth learn\_rate loss\_reduction sample\_size

##

## 1 5 25 4 0.0195 0.00112 0.977

Now let’s finalize our tuneable workflow with these parameter values.

final\_xgb <- finalize\_workflow(

xgb\_wf,

best\_auc

)

final\_xgb

## ══ Workflow ═══════════════════════════════════════════════════════════

## Preprocessor: Formula

## Model: boost\_tree()

##

## ── Preprocessor ───────────────────────────────────────────────────────

## win ~ .

##

## ── Model ──────────────────────────────────────────────────────────────

## Boosted Tree Model Specification (classification)

##

## Main Arguments:

## mtry = 5

## trees = 1000

## min\_n = 25

## tree\_depth = 4

## learn\_rate = 0.019501844932014

## loss\_reduction = 0.00112048286512169

## sample\_size = 0.977300804650877

##

## Computational engine: xgboost

Instead of tune() placeholders, we now have real values for all the model hyperparameters.

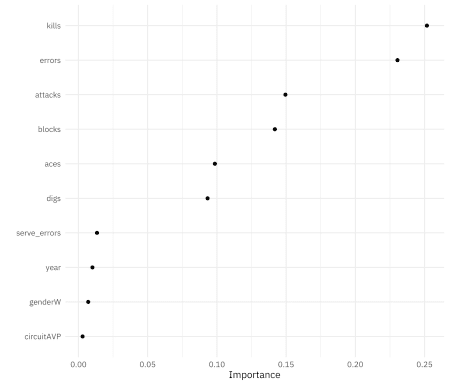
library(vip)

final\_xgb %>%

fit(data = vb\_train) %>%

pull\_workflow\_fit() %>%

vip(geom = "point")



The predictors that are most important in a team winning vs. losing their match are the number of kills, errors, and attacks. There is almost no difference between the two circuits, and very little difference by gender.

It’s time to go back to the testing set! Let’s use last\_fit() to *fit* our model one last time on the training data and *evaluate* our model one last time on the testing set. Notice that this is the first time we have used the testing data during this whole modeling analysis.

final\_res <- last\_fit(final\_xgb, vb\_split)

collect\_metrics(final\_res)

## # A tibble: 2 x 3

## .metric .estimator .estimate

##

## 1 accuracy binary 0.843

## 2 roc\_auc binary 0.928

Our results here indicate that we did not overfit during the tuning process. We can also create a ROC curve for the testing set.

final\_res %>%

collect\_predictions() %>%

roc\_curve(win, .pred\_win) %>%

ggplot(aes(x = 1 - specificity, y = sensitivity)) +

geom\_line(size = 1.5, color = "midnightblue") +

geom\_abline(

lty = 2, alpha = 0.5,

color = "gray50",

size = 1.2

)

