PSTAT 131 Project

2024-03-05

Introduction

The goal of my project is to train a model that can predict the action that a poker player takes after all the cards have been shown based on their initial stack, position, all-in status, cards, hand combination, pot sizes, and betting amounts. I am trying to predict the variable, action_river, with the predictors I mentioned in the question. The poker game is called Spin N Go Texas Hold'Em. Spin N Go is a style of tournament poker that is played with only three players. Every player gets dealt two cards. Then three "community" cards are flipped in the middle of the board (called the "flop"). Then another card is turned over in the middle (called the "turn"). And finally one more card is turned over in the middle of the board (called the "river"). All players can combine their hands with the middle board to try and make the best hand. I am trying to predict the action that a player takes once that last card has been flipped (the river card) using the information throughout the hand. This is incredibly important in poker because if you can predict the action of another player you have a significant advantage. There are multiple specific classes of the action_river variable, however I am only interested in cases where the player folds, checks, bets, or calls. These are the four classes I will be trying to predict. For the bets class, this is when the player makes a bet into the pot or re-raises the previous player's bet. I'm combining bets and raises into the same class because the only difference between them is the order of the players. For example, you can't re-raise if you start the betting. Combining these two also simplifies the model and will allow for easier analysis. The data set is very specific and includes whether the player shows their cards or not, however this is not useful for our goal. I will be using a classification approach to answer my question because I am trying to predict the action or class that a player takes at the end of a hand. I think the betting sizes will be the most useful for predicting my response. Larger bets usually means that players are playing aggressively. The goal of my model is inferential and predictive. I want to create a model that is good at predicting my response but I also want to figure out which predictors are most influential in that prediction.

Table position is a very important concept in poker. We are only looking at three positions in this data set: BTN, SB, and BB. These stand for Button, Small Bind, and Big Blind respectively. The Button is the position of the dealer. This is generally considered the best position to be in in poker because this position gets to see everyone else bet before they have to make a decision. This position has the most information before they have to make an action. It will be interesting to see if the Button position usually wins more than the other two positions. It's important to note that this data set is only looking at games with three people. Normal poker games are usually played with 9 people so the data might be a little different than a usual poker game. Stack is also another important variable to clarify. Stack refers to the amount of money a player has before the hand starts. This is important because generally players will larger stacks will play more aggressively and "bully" other players, meaning they will bet more and pressure other players to fold.

There are 102,615 observations and 35 predictors in the original data set. However, I will not be using all of the observations or predictors for my project. I will only be using observations where the player makes it to the river card and either folds, checks, bets, or calls. I will be working with integers, character, and Boolean variables. The only missing values are for the hand combination variables. About 82% of this variable is missing. The null values occur only when the player doesn't show their cards at the end of a round (called "going to showdown").

Data Citation

I got my data set from Kaggle. The data set was built from online poker games in 2020. Below is a citation for the data set and includes the link to the source.

Murilo G.M. Amaral. (2020, July). Online Poker Games Data set. Retrieved March 2, 2024 from Kaggle: https://www.kaggle.com/datasets/murilogmamaral/online-poker-games/data

Codebook

Here I've included a section that identifies and briefly describes every variable in the data set, although I will not be using all of them to train the model.

buyin	character	Amount paid (USD) to play the tournament
tourn_id	integer	Tournament id
table	integer	Table number reference in the tournament
hand_id	integer	Hand id
date	date	Date of a played hand
time	datetime	Time of a played hand
table_size	integer	Maximum number of players per table
level	integer	Blinds levels
playing	integer	Number of players currently in the table
seat	integer	Seat number of each player
name	character	Player login
initial_stack	double	Initial stack of each player
position	integer	Position of each player
action_pre	character	Preflop actions of each player
action_flop	character	Flop actions of each player
action_turn	character	Turn actions of each player
action_river	character	River actions of each player
all_in	boolean	Informs if a player did an all-in bet (TRUE) or not (FALSE)
cards	character	Hand of each player (only available if you see it, obviously)
board_flop	character	Cards on flop
board_turn	character	Cards on turn
board_river	character	Cards on river
combination	character	Cards combination of each player
pot_pre	double	Preflop pot size
pot_flop	double	Flop pot size
pot_turn	double	Turn pot size
pot_river	double	River pot size
ante	double	Ante paid
blinds	double	Blinds paid
bet_pre	double	Bet done on preflop
bet_flop	double	Bet done on flop
bet_turn	double	Bet done on turn
bet_river	double	Bet done on river
result	character	Four categories: 1-won, means a player went to showdown and won; 2-lost, means a player went to showdown and lost; 3-gave up, means a player folded at some point; 4-took chips, means a player took chips without going to showdown.
balance	double	How much a player won or lost after a hand

Exploratory Data Analysis

First, I need to prepare the data for EDA. Most importantly, I need to get rid of the observations that I don't want to include. Again, I will only be using observations where the player makes it to the river card and either folds, checks, bets, raises, or calls after the river card has been flipped.

```
# Read in the data
poker_df <- read.csv("poker.csv")

# Getting rid of certain classes of action_river that we are not interested in
poker_df <- poker_df %>%
  filter(action_river != "shows") %>%
  filter(action_river != "mucks") %>%
  filter(action_river != "x")

# Renaming the variable to only the first action that the player takes
poker_df$action_river <- sub("-.*", "", poker_df$action_river)

# Combines bets and raises class into same class
poker_df <- poker_df %>%
  mutate(action_river = ifelse(action_river %in% c("bets", "raises"), "bets/raises", action_river))
```

I have restricted the data set to the observations that I want to analyze. Now let's select the variables we want to use for the true data set. The most important variables that change for every player on every hand are the actions pre-flop, flop, and turn as well as the betting sizes, and pot sizes for those three stages of the game before the river. The actions, betting size, and pot size all tell a story for what the player will likely do on the river. I don't want to include the hands of the players or hand combinations. First of all, these are categorical variables and there are an incredibly large amount of categories for these variables which likely won't help the model. Second, I want to try and simulate a player guessing at what another player will do without knowing the other player's cards. I don't need to include the ante variable because that stays on 0 for every observation. And I don't need to include the blinds because these are payments that the players are forced to make and don't give any information regarding future actions. I will only be using the following variables:

action_river (response variable)

stack

position

action_pre

action_flop

action_turn

pot_pre

pot_flop

pot_turn

bet_pre

bet_flop

bet_turn

```
poker_df <- select(poker_df, c('action_river', 'stack', 'position', 'action_pre', 'action_flop', 'actio
head(poker_df)
```

```
##   action_river stack position  action_pre action_flop  action_turn pot_pre
## 1       checks   740       BB       calls      checks checks-calls     120
## 2       checks   280      BTN calls-calls      checks         bets     120
## 3  bets/raises   380       SB      raises  bets-calls   bets-calls      80
## 4        calls   490       BB       calls      raises       raises      80
## 5        calls   900       SB      raises       calls        calls      80
## 6  bets/raises   600       BB       calls        bets         bets      80
##   pot_flop pot_turn bet_pre bet_flop bet_turn
## 1      120      240      40        0       60
## 2      120      240      40        0       60
## 3      160      360      40       40      100
## 4      160      360      40       40      100
## 5      160      480      40       40      160
## 6      160      480      40       40      160
```

We've now limited the data set to the variables we are interested in.

We want the variables like action_river, action_pre, action_flop, action_turn, and position to be of a categorical type, so we will factorize those using the code below:

```
poker_df$action_river <- as.factor(poker_df$action_river)
poker_df$position <- as.factor(poker_df$position)
poker_df$action_pre <- as.factor(poker_df$action_pre)
poker_df$action_flop <- as.factor(poker_df$action_flop)
poker_df$action_turn <- as.factor(poker_df$action_turn)
```

Before we are ready to dive into the relationships between variables, lets see if we have any missing data.

```
# Check for missing values in each column
missing_values <- colSums(is.na(poker_df))

# Display columns with missing values and their counts
missing_values[missing_values > 0]
```

```
## named numeric(0)
```

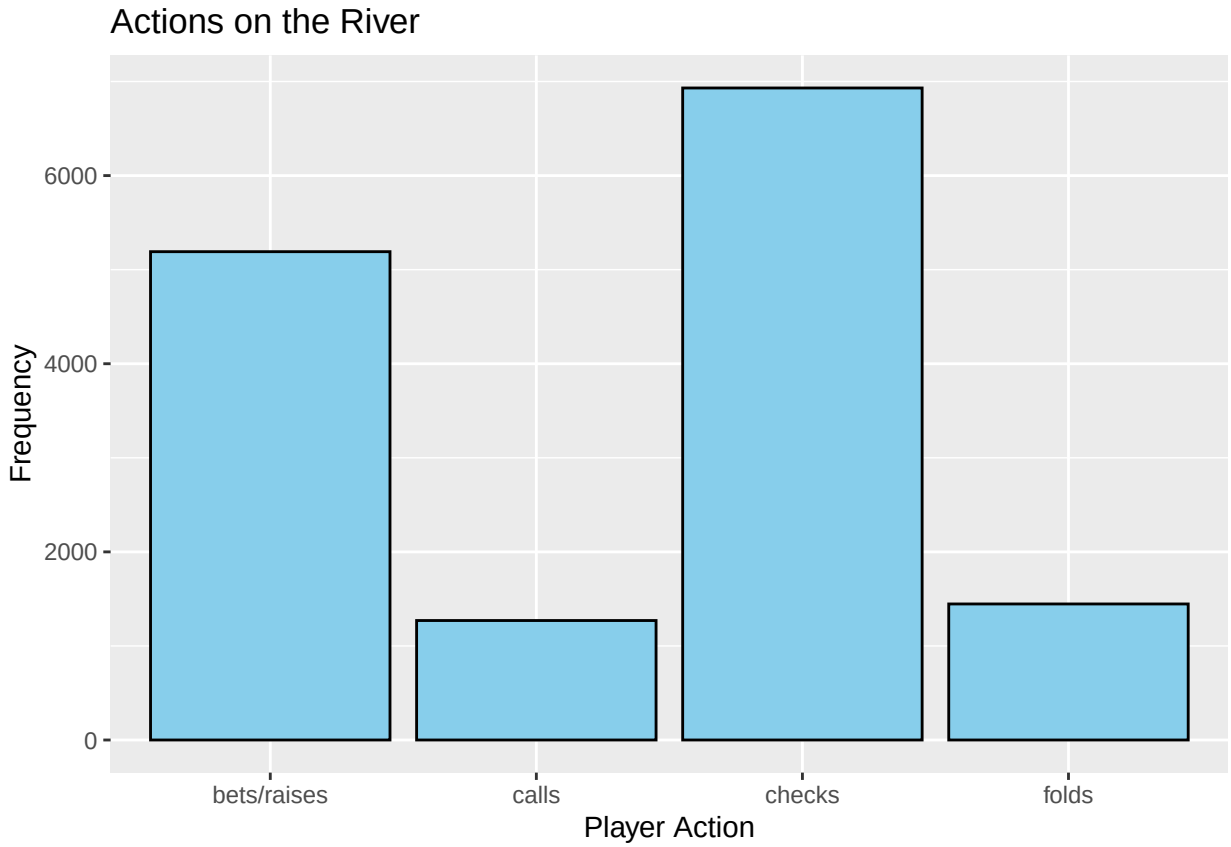
This output indicates an empty numeric vector with no elements, which implies that no columns have missing values so we don't have to do any more data cleaning.

Therefore, our data is ready to go!

Now let's take a look at the distribution of the response variable, action_river.

Distribution of Response Variable

```
ggplot(poker_df, aes(x=action_river)) +
  geom_bar(fill = "skyblue", color="black") +
    labs(title = "Actions on the River", x = "Player Action", y = "Frequency")
```

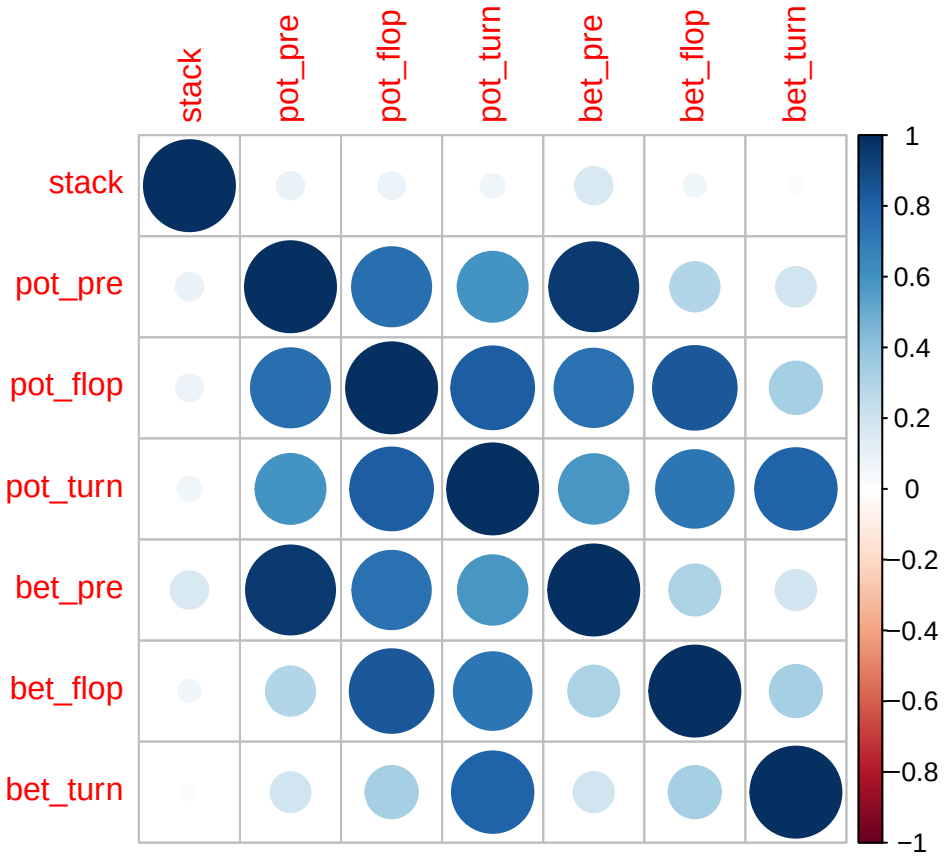


The checks class has the most observations followed by the bets/raises class. Both of these classes contain around 6,000 observations. The calls and folds classes both contain around 1,400 observations each. This is good to know before we re-sample and lets us know that we should probably use machine learning methods that are robust to class imbalances.

Let's look at a correlation heat map of the numeric variables to get an idea of the relationship.

Variable Correlation Plot

```
# To get the numeric data
poker_numeric <- poker_df %>%
  select_if(is.numeric)

# Calculating the correlation between each variable
poker_cor <- cor(poker_numeric)

# Making the correlation plot
poker_cor_plt <- corrplot(poker_cor)
```

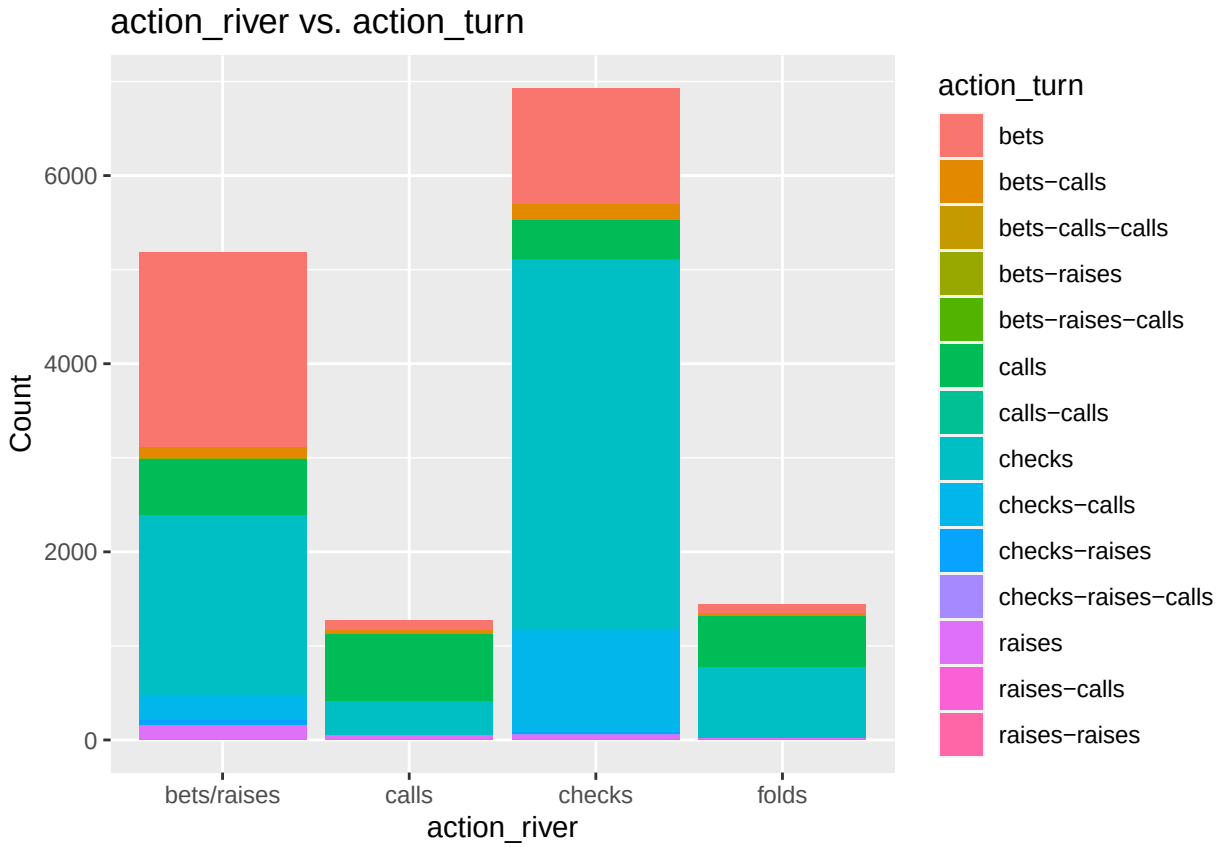


Take a look at the same stages of the game for the pot and bet variables (pot_pre and bet_pre for example). Pot_pre and bet_pre, pot_turn and bet_turn, and pot_flop and bet_flop are all highly positively correlated with each other. This makes sense because at a certain stage of the game, lets say flop, if a player bets during this stage, then the pot will grow. What's more interesting is how post stages of the game impact future stages. Bet_pre is only a little positively correlated with bet_flop and even less with bet_turn. This means that a large bet pre-flop does not necessarily mean the player will also bet large on the flop or turn. Stack isn't really correlated with anything. It is barely positively correlated with bet_pre. This also makes sense because players are usually more likely to make a large bet before the cards come out if they have a large stack.

Action_turn relationship with response

Let's take a look at how action_turn varies with action_river. Remember, the turn is the stage of the hand right before the river.

```
ggplot(poker_df, aes(x = action_river, fill = action_turn)) +
  geom_bar(position = 'stack') +
  labs(title = "action_river vs. action_turn", x = "action_river", y = "Count")
```



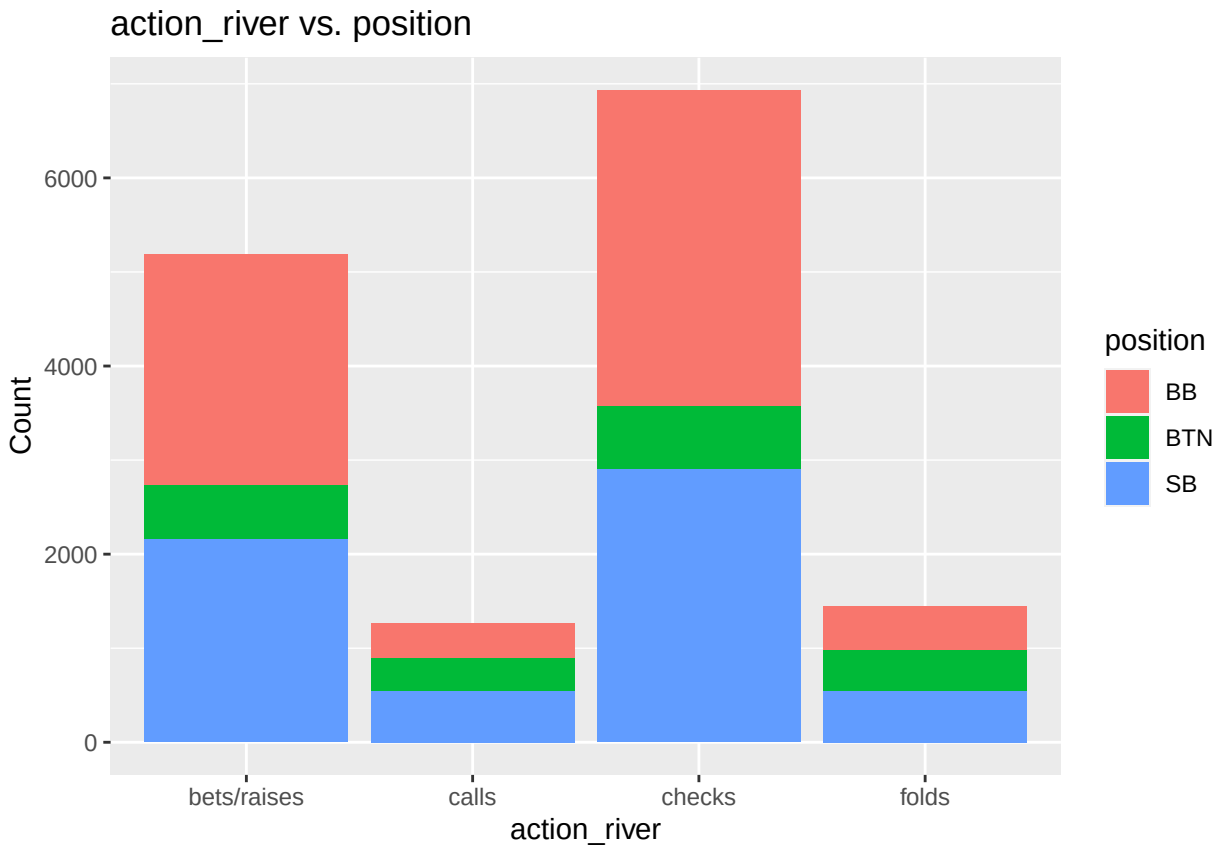
There are 14 classes within the action_turn variable. I considered simplifying this variable as well into just bets, calls, checks, and raises classes but I felt that this would be restricting too much valuable information. For example, raises-raises is significantly different than just raise. Raises-raises means that the player raised the bet, then another player re-raised them, then the original player re-re-raises them once again. This means the player is extremely aggressive and is likely to bet again on the river. If we combined this class into just the raises class, we wouldn't know that the player is this aggressive. This largest takeaways from this stacked bar chart are as follows:

1) Players that bet on the turn usually either bet/raise again on the river or check. Very rarely will they call or fold.
2) Players that check on the turn will usually either bet/raise on the river or check.
3) Players that raise, raises-calls, or raises-raises on the turn almost always will bet again on the river.
4) When a player calls on the turn, their action on the river is evenly spread out between the four response classes. This means that when a player calls a bet on the turn, it is difficult to predict what they will do on the river.

Something important to note is that there might not a large difference in information given between a check and a fold or a call and a bet. For example, if a player doesn't have a good hand but they start off the betting, they will usually check since this means they get to stay in the hand without putting any money into the pot. However, if they are in the second position and a bet has been made before them, they will likely fold since they can no longer check and they have a poor hand. Similarly with a call and a bet, if a player was already planning on betting but someone bet before them, they will likely just call instead of re-raising. These examples are why position matters so much in poker and in this project. Let's take a look at the positions relationships with the response variable.

Position relationship with response

```
ggplot(poker_df, aes(x = action_river, fill = position)) +
  geom_bar(position = 'stack') +
  labs(title = "action_river vs. position", x = "action_river", y = "Count")
```

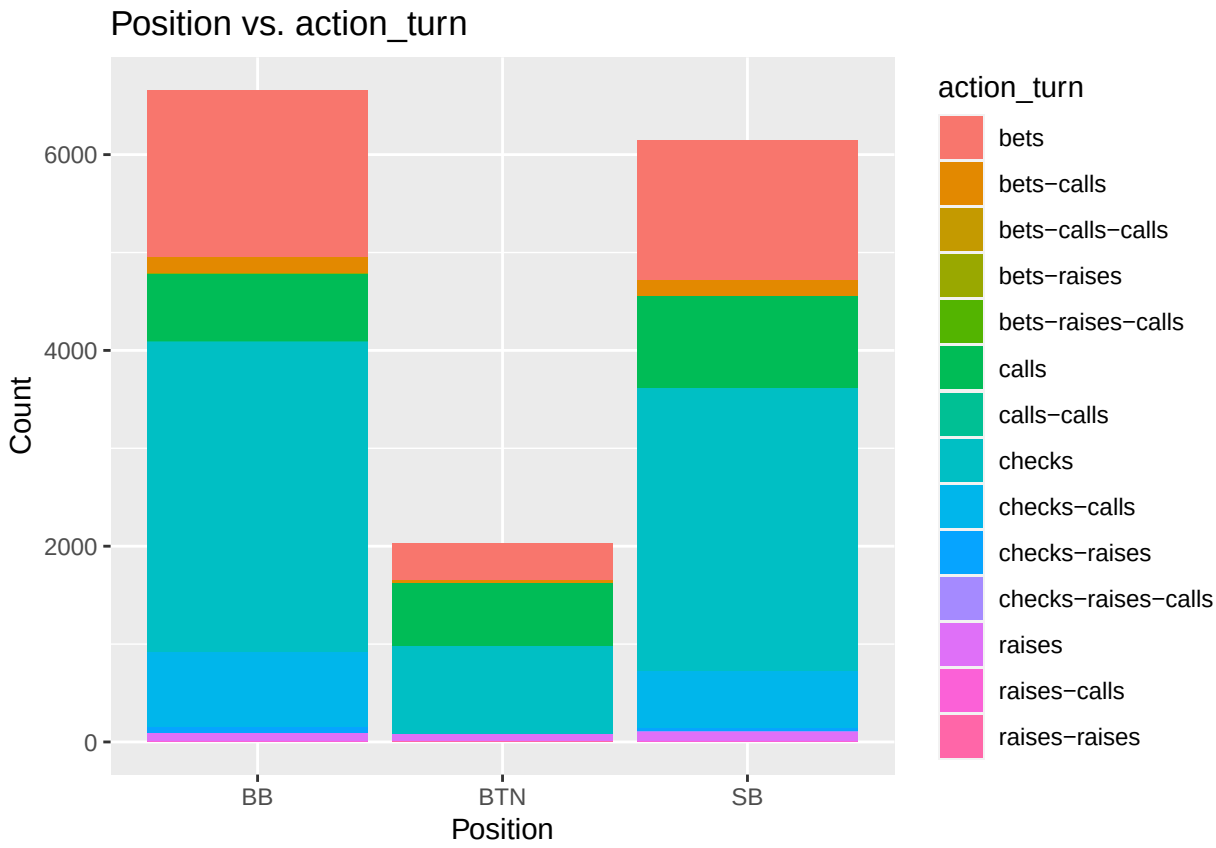


At first glance it looks like players in the BB and SB positions usually bets/raises and checks more often. But this might just be because there are more observations in those two response classes. When we look at the proportion of positions in each response class, the proportions are almost all the same. BB and SB make up around 80% of each class while BTN makes up roughly 20% of each class. This likely means that position might not be a great predictor of action_river. Although there are more observations of BB and SB in the bets/raises and checks classes, there are also more total observations in the bets/raises and checks classes.

Position and action_turn relationship

I also thought it would be interesting to look at the position and action_turn variables plotted together to see if any actions happen more often with certain positions.

```
ggplot(poker_df, aes(x = position, fill = action_turn)) +
  geom_bar(position = 'stack') +
  labs(title = "Position vs. action_turn", x = "Position", y = "Count")
```

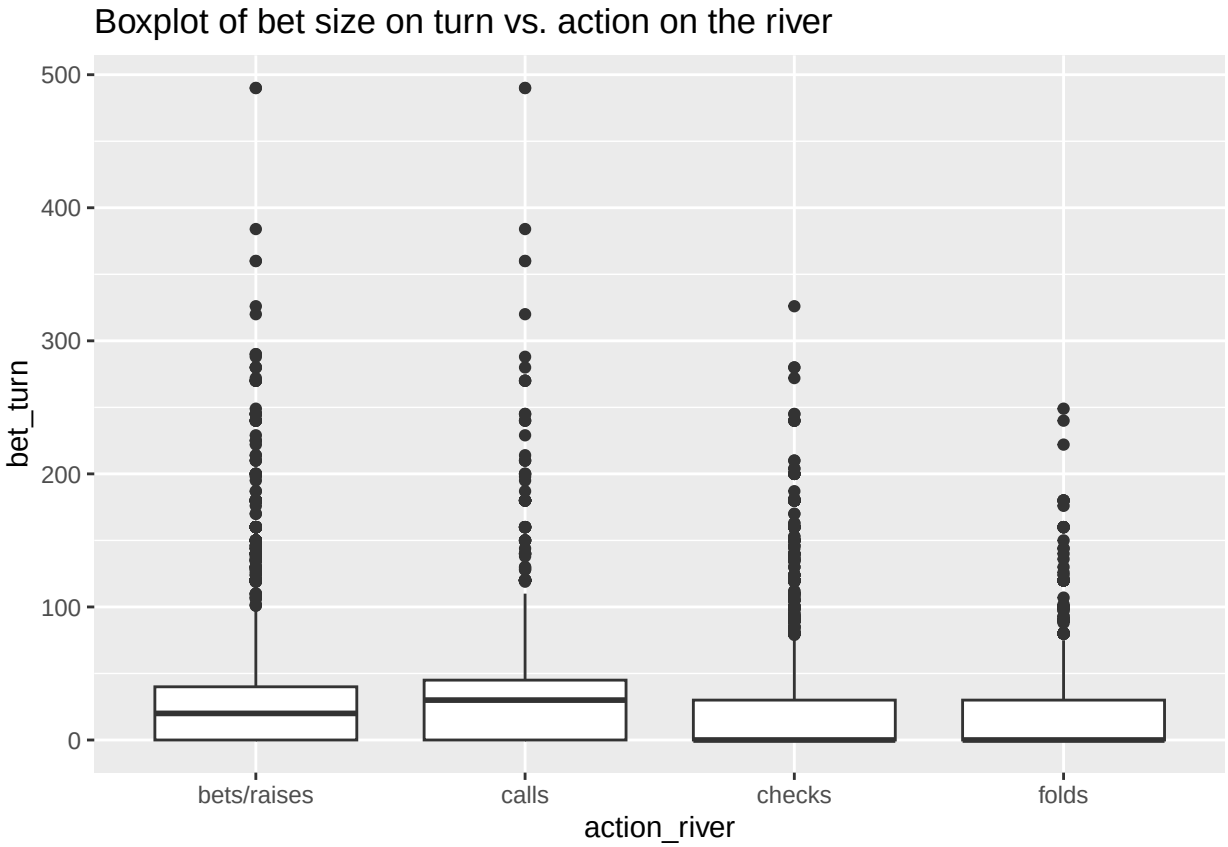


It's important to note that there are far fewer observations for BTN than BB and SB. The distributions of action_turn within the BB and SB classes are almost identical. The BB class checks and bets slightly more than the SB class while the SB class calls a little more than the BB. The BB position also has more observations than the SB position. This is surprising to me that position does not seem to have as much of an effect on actions as I originally thought.

Bet size on the turn relationship with response

I'm interested to look at the sizing of the bet on the turn and how that correlates with the response variable. I would imagine larger bets would lead to more aggressive actions on the river like bets/raises and not a lot of folding.

```
ggplot(data = poker_df, aes(x = action_river, y = bet_turn)) +
  geom_boxplot() +
  labs(title = "Boxplot of bet size on turn vs. action on the river",
       x = "action_river", y = 'bet_turn')
```



It seems like there is not a very large relationship between bet_turn and our response variable. It does seem that smaller bets on the turn lead to more checks and folds on the river while larger bets lead to more bets/raises and calls on the river. I made this observations due to the fact that the median for bets/raises and calls is significantly higher than checks and folds. There are also larger outliers for bets/raises and calls as well.

Model Building

I will be building 5 models: Elastic Net, Gradient Boosted Trees, Pruned Decision Trees, Random Forest, and K-Nearest Neighbors (KNN). Almost all of the models will go through the same process. We specify what type of model, setting up its engine, and setting its classification mode. Then we set up the workflow, add the model, and add the recipe. Then for the models that need parameter tuning, we set up the tuning grid, and how many levels of tuning. We then select the most accurate model from the tuning, and finalize the workflow with those parameters. We will fit the best model with our workflow to the training data set. But before we start building these models, we have to perform a training/testing split of our data. I chose to split the data 80/20 because the testing data set will still have a large portion of data to test on since we have over 14,000 total observations. We will also set a random seed to make sure the training / testing split is the same set every time we run our code. We will also stratify on the response variable, action_river, to make sure that the distribution of the response variable is similar in both the training and testing sets.

```
set.seed(1234)
poker_split <- poker_df %>%
  initial_split(prop = 0.8, strata = "action_river")

poker_train <- training(poker_split)
poker_test <- testing(poker_split)
```

```
dim(poker_train)
```

```
## [1] 11870    12
```

```
dim(poker_test)
```

```
## [1] 2968   12
```

Recipe Building

Since we will be using the same predictors, model conditions, and response variable, we will create one single recipe for all our models to use. This recipe allows us to specify a series of data transformations and pre-processing steps that will be applied to the data set before every model. We will center and scale our data and make all our categorical variables in dummy variables.

```
poker_recipe <- recipe(action_river ~ stack + position + action_pre + action_flop + action_turn + pot_pr
  step_dummy(position) %>%
  step_dummy(action_pre) %>%
  step_dummy(action_flop) %>%
  step_dummy(action_turn) %>%
  step_zv(all_predictors()) %>%
  step_center(all_predictors()) %>%
  step_scale(all_predictors())
```

I added in the step_zv() to remove variables that contain only a single value.

K-Fold Cross Validation

To deal with imbalanced data in our response variable, we will use stratified cross validation. We will stratify on our response variable, action_river. Also, since we have a large data set we will be using 5 folds in order to save computational resources.

```
poker_folds <- vfold_cv(poker_train, v = 5, strata = action_river)
```

I also will save my model results to an RDA file so I don't have to rerun every model whenever I want to see my results.

```
save(poker_folds, poker_recipe, poker_train, poker_test, file = "/Users/caseylinden/Documents/PSTAT 131/
```

Elastic Net

Elastic Net is a method that combines the penalties of both Lasso (L1 regularization) and Ridge (L2 regularization) techniques. It works by adding two penalty terms to the ordinary least squares objective function.

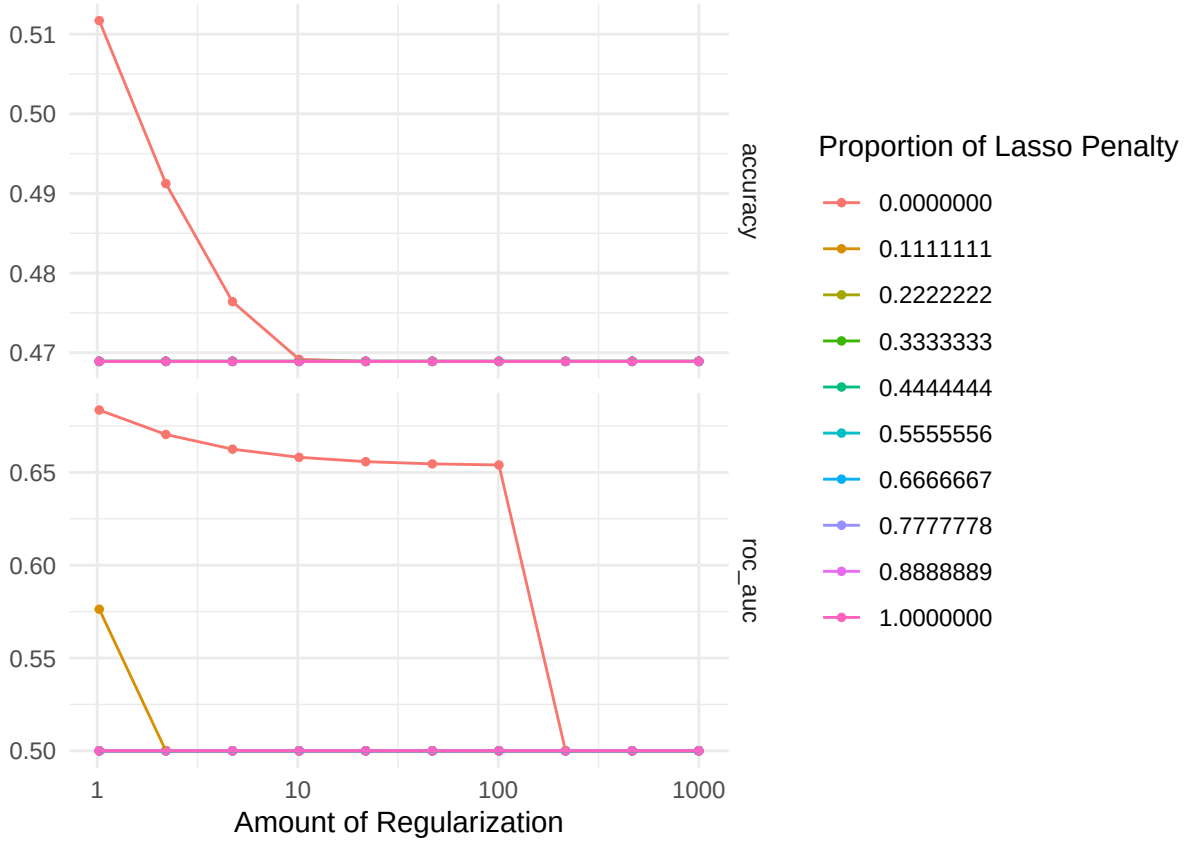
We will set up the penalty term with a range between 0.01 and 3, and the mixture term with a range from 0 to 1, with 10 levels each.

```
# Setting up the Elastic Net Model
en_model <- multinom_reg(mixture = tune(),
                         penalty = tune()) %>%
  set_mode("classification") %>%
  set_engine("glmnet")

# Setting up the workflow
en_wf <- workflow() %>%
  add_recipe(poker_recipe) %>%
  add_model(en_model)
```

```
en_grid <- grid_regular(penalty(range = c(0.01, 3)),
                        mixture(range = c(0, 1)),
                             levels = 10)
```

```
# Tune hyperparameters
tune_en <- tune_grid(
  en_wf,
  resamples = poker_folds,
  grid = en_grid,
)

# Save the results to RDA file
save(tune_en, file = "tune_en.rda")
```

```
load("tune_en.rda")
autoplot(tune_en) + theme_minimal()
```



As you can see, the accuracy is better for lower amounts of regularization. The highest accuracy had 0 proportion of Lasso Penalty. The highest AUC ROC was given with a small amount of regularization and 0 proportion of Lasso Penalty.

```
show_best(tune_en, n = 1)
```

```
## Warning: No value of `metric` was given; metric 'roc_auc' will be used.
```

```
## # A tibble: 1 x 8
##   penalty mixture .metric .estimator  mean     n  std_err .config
##     <dbl>   <dbl> <chr>   <chr>      <dbl> <int>    <dbl> <chr>
```

```
## 1    1.02       0 roc_auc hand_till  0.684     5 0.000796 Preprocessor1_Model001
```

```
best_en_model <- select_best(tune_en)
```

```
## Warning: No value of `metric` was given; metric 'roc_auc' will be used.
```

```
en_auc_score <- show_best(tune_en, n = 1)['mean']
```

```
## Warning: No value of `metric` was given; metric 'roc_auc' will be used.
```

The best Elastic Net model has a penalty term of 1.02 and 0 mixture with an AUC ROC of 0.68.

```
final_en_model <- finalize_workflow(en_wf, best_en_model)
final_en_model <- fit(final_en_model, poker_train)
```

Gradient Boosted Trees (GBT)

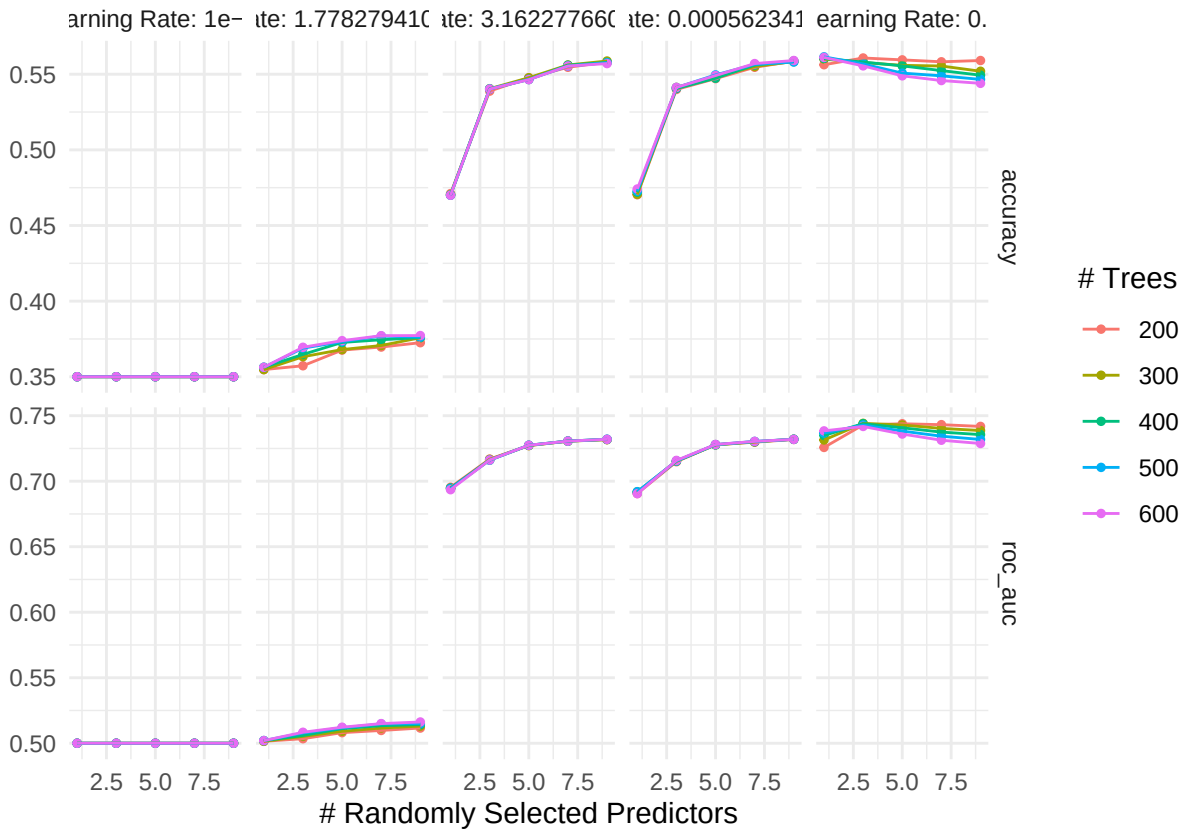
Gradient Boosted Trees combines the predictive power of multiple decision trees to create a robust predictive model. The main idea is to iteratively build a sequence of decision trees, each one focusing on the prediction errors made by the previous trees. The process is guided by a loss function, which measures the difference between the actual and predicted values, and by the gradient descent algorithm, which optimizes the model parameters to minimize this loss function.

We'll tune mtry and trees again; we could also choose to tune min_n again, but the learning rate tends to have a much bigger impact on the performance of gradient-boosted models, and we don't want to increase the size of the grid too much, so we'll add learn_rate instead of min_n. The default engine for gradient-boosted trees is xgboost, which stands for "eXtreme Gradient Boosting."

```
# Set up Gradient Boosted Tree model
gbt_model <- boost_tree(mtry = tune(),
                        trees = tune(),
                        learn_rate = tune()) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

# Set up workflow
gbt_wf <- workflow() %>%
  add_model(gbt_model) %>%
  add_recipe(poker_recipe)

# Set up hyperparameter grid
gbt_grid <- grid_regular(mtry(range = c(1, 9)),
                         trees(range = c(200, 600)),
                         learn_rate(range = c(-10, -1)),
                         levels = 5)
```

```
# Tune hyperparameters
tune_gbt <- tune_grid(
  gbt_wf,
  resamples = poker_folds,
  grid = gbt_grid,
)

# Save the results to RDA file
save(tune_gbt, file = "tune_gbt.rda")
```

```
load("tune_rf.rda")
autoplot(tune_gbt) + theme_minimal()
```



The best accuracy and AUC ROC seem to be when the maximum number of predictors are used, there are 600 trees, and a small Learning Rate.

```
show_best(tune_gbt, n = 1)
```

```
## Warning: No value of `metric` was given; metric 'roc_auc' will be used.
```

```
## # A tibble: 1 x 9
##    mtry trees learn_rate .metric .estimator  mean     n std_err .config
##   <int> <int>      <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1     3   300        0.1 roc_auc hand_till  0.744     5 0.00302 Preprocessor1_M~
```

```
best_gbt_model <- select_best(tune_gbt)
```

```
## Warning: No value of `metric` was given; metric 'roc_auc' will be used.
```

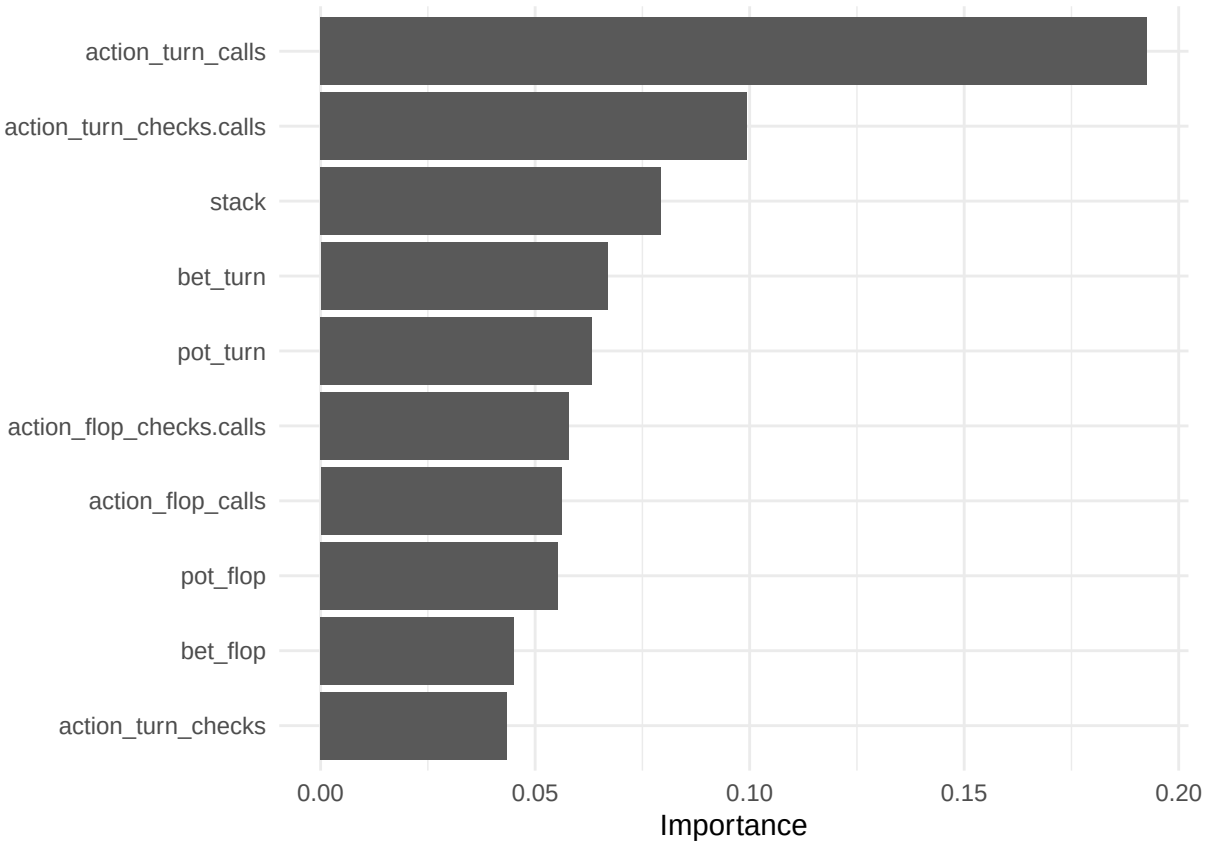
```
gbt_auc_score <- show_best(tune_gbt, n = 1)['mean']
```

```
## Warning: No value of `metric` was given; metric 'roc_auc' will be used.
```

The best Gradient Boosted Tree model has mtry of 3, 300 trees, and a learning rate of 0.1. This model has an AUC ROC of 0.74.

```
final_gbt_model <- finalize_workflow(gbt_wf, best_gbt_model)
final_gbt_model <- fit(final_gbt_model, poker_train)
```

```
final_gbt_model %>% extract_fit_parsnip() %>%
  vip() +
  theme_minimal()
```



This shows that the most important variable in training the best GBT model is the calls class from the action_turn variable.

Pruned Decision Trees

A pruned decision tree is a modified version of a traditional decision tree where some branches and nodes are removed or "pruned" to improve the tree's generalization ability and prevent overfitting. Overfitting occurs when the decision tree captures noise or random fluctuations in the training data rather than the underlying relationships.

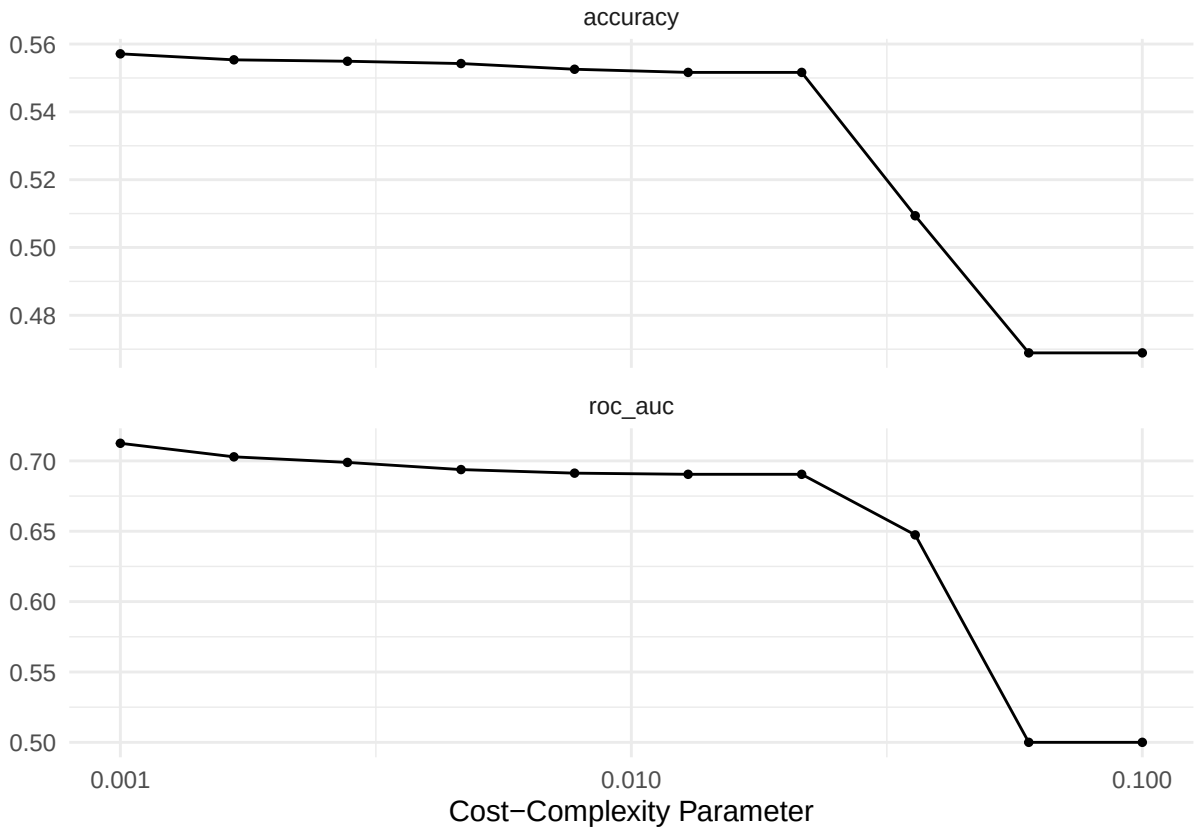
```
# Set up model
pdt_model <- decision_tree(cost_complexity = tune()) %>%
  set_engine("rpart") %>%
  set_mode("classification")

# Set up workflow
pdt_wf <- workflow() %>%
  add_model(pdt_model) %>%
  add_recipe(poker_recipe)

# Set up hyperparameter grid
pdt_grid <- grid_regular(cost_complexity(range = c(-3, -1)), levels = 10)
```

We set up a grid of possible values to consider for the cost-complexity parameter and tune the models, fitting all 10 of them to each of the 5 folds for a total of 50 decision trees.

```
# Tune hyperparameters
tune_pdt <- tune_grid(
```

```
  pdt_wf,
  resamples = poker_folds,
  grid = pdt_grid
)

# Save the results to RDA file
save(tune_rf, file = "tune_pdt.rda")
```

```
load("tune_pdt.rda")
autoplot(tune_pdt) + theme_minimal()
```



The accuracy and AUC ROC are at their highest when the cost-complexity parameter is near 0.

```
show_best(tune_pdt, n = 1)
```

```
## Warning: No value of `metric` was given; metric 'roc_auc' will be used.
```

```
## # A tibble: 1 x 7
##   cost_complexity .metric .estimator  mean     n std_err .config
##             <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1           0.001 roc_auc hand_till  0.712     5 0.00279 Preprocessor1_Model01
```

```
best_pdt_model <- select_best(tune_pdt)
```

```
## Warning: No value of `metric` was given; metric 'roc_auc' will be used.
```

```
pdt_auc_score <- show_best(tune_pdt, n = 1)['mean']
```

```
## Warning: No value of `metric` was given; metric 'roc_auc' will be used.
```

The best Pruned Decision Tree model has a cost complexity of 0.001 and a AUC ROC of 0.70.

```
final_pdt_model <- finalize_workflow(pdt_wf, best_pdt_model)
final_pdt_model <- fit(final_pdt_model, poker_train)
```

Random Forest

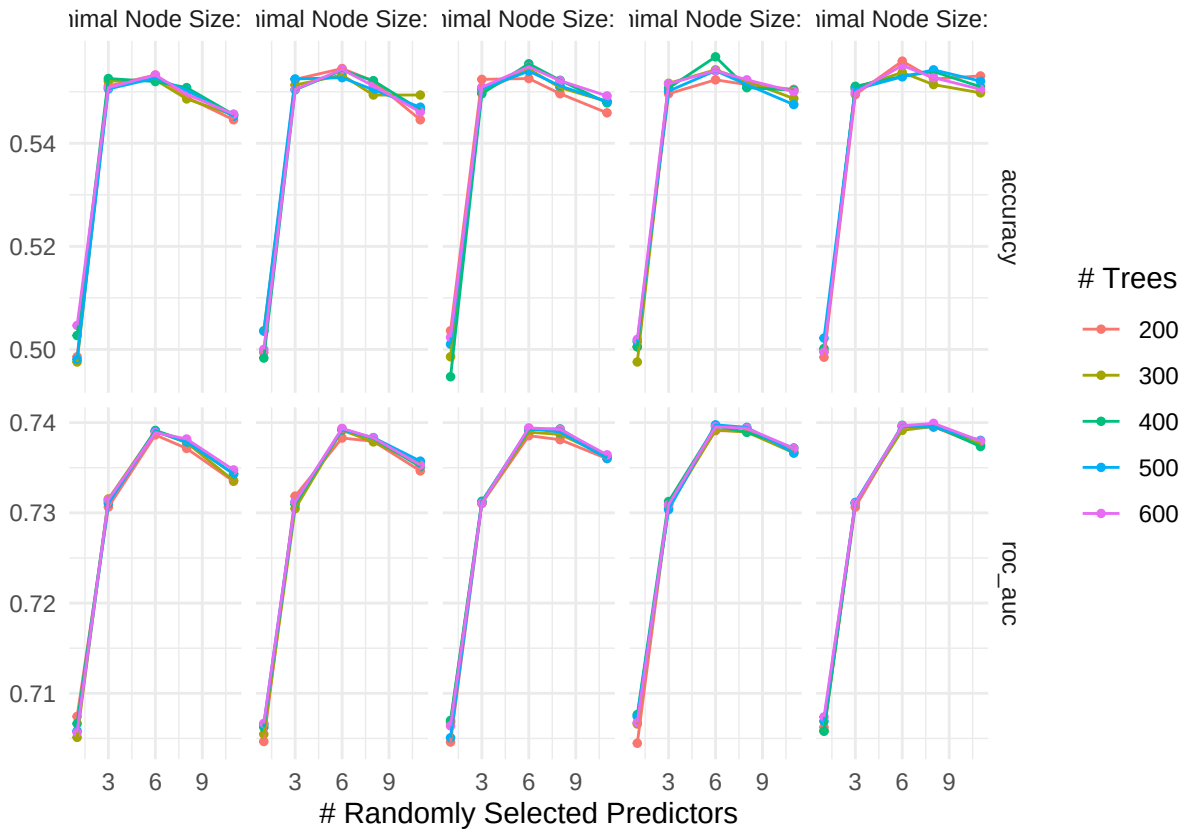
The default engine for random forest models in tidymodels is the ranger package. We will use three hyperparameters for tuning – mtry, trees, and min_n. We also will set up a grid of hyperparameter values to consider. Here we allow mtry to range from 1 to 11, trees from 200 to 600, and min_n from 10 to 20, and we specify five levels of each.

```
# Setting up Random Forest model
rf_model <- rand_forest(mtry = tune(),
                        trees = tune(),
                        min_n = tune()) %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("classification")

# Set up workflow
rf_wf <- workflow() %>%
  add_model(rf_model) %>%
  add_recipe(poker_recipe)

# Set up hyperparameter grid
rf_grid <- grid_regular(mtry(range = c(1, 11)),
                        trees(range = c(200, 600)),
                        min_n(range = c(10, 20)),
                        levels = 5)
```

Now we fit all the random forest models we've specified to each data set. We will save the results to files and load them back in later.

```
# Fit Random Forest models and tune hyperparameters
tune_rf <- tune_grid(
  rf_wf,
  resamples = poker_folds,
  grid = rf_grid
)

# Save the results to RDA file
save(tune_rf, file = "tune_rf.rda")
```

```
load("tune_rf.rda")
autoplot(tune_rf) + theme_minimal()
```



For the Random Forest, we tuned: mtry - The number of predictors that would be randomly sampled and given to the tree to make its decisions, trees - The number of trees to grow in the forest, and min_n - the minimum number of data values needed to create another split. As the number of predictors increased, so did the accuracy as well as the ROC AUC up until it got to predictor 7. There wasn't much change in the ROC AUC as the number of trees increased, but usually the more trees, the higher ROC AUC.

We'll select the optimal random forest model for the poker data set in terms of ROC AUC:

```
show_best(tune_rf, n = 1)
```

```
## Warning: No value of `metric` was given; metric 'roc_auc' will be used.
```

```
## # A tibble: 1 x 9
##    mtry trees min_n .metric .estimator  mean     n std_err .config
##   <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1     8   600    20 roc_auc hand_till  0.740     5 0.00258 Preprocessor1_Model1~
```

```
best_rf_model <- select_best(tune_rf)
```

```
## Warning: No value of `metric` was given; metric 'roc_auc' will be used.
```

```
rf_auc_score <- show_best(tune_rf, n = 1)['mean']
```

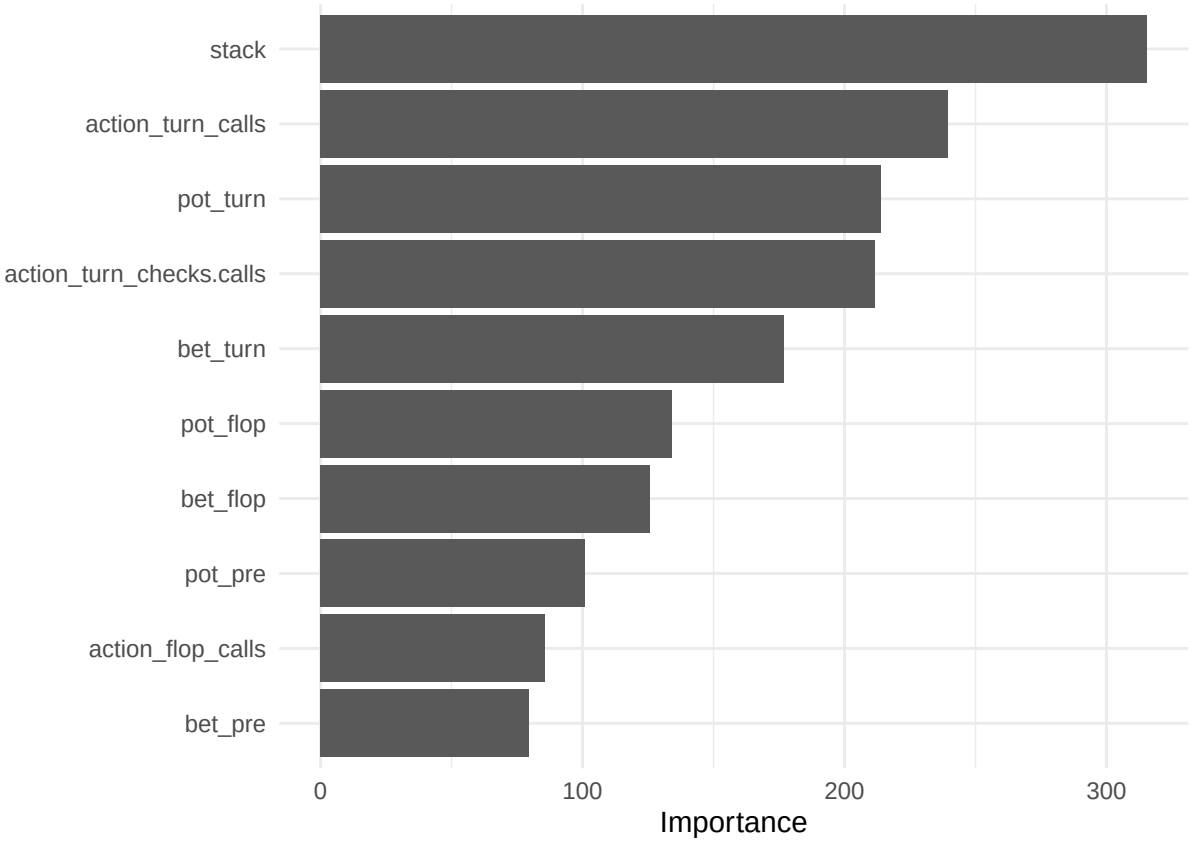
```
## Warning: No value of `metric` was given; metric 'roc_auc' will be used.
```

The best Random Forest Model has a mtry = 6, 600 trees, and min_n = 20 with a mean ROC AUC score of 0.74.

Note that we specify importance = "impurity" in the workflow setup above; this is so that we can use extract_fit_parsnip() and vip() to create and view a variable importance plot.

```
final_rf_model <- finalize_workflow(rf_wf, best_rf_model)
final_rf_model <- fit(final_rf_model, poker_train)
```

```
final_rf_model %>% extract_fit_parsnip() %>%
  vip() +
  theme_minimal()
```



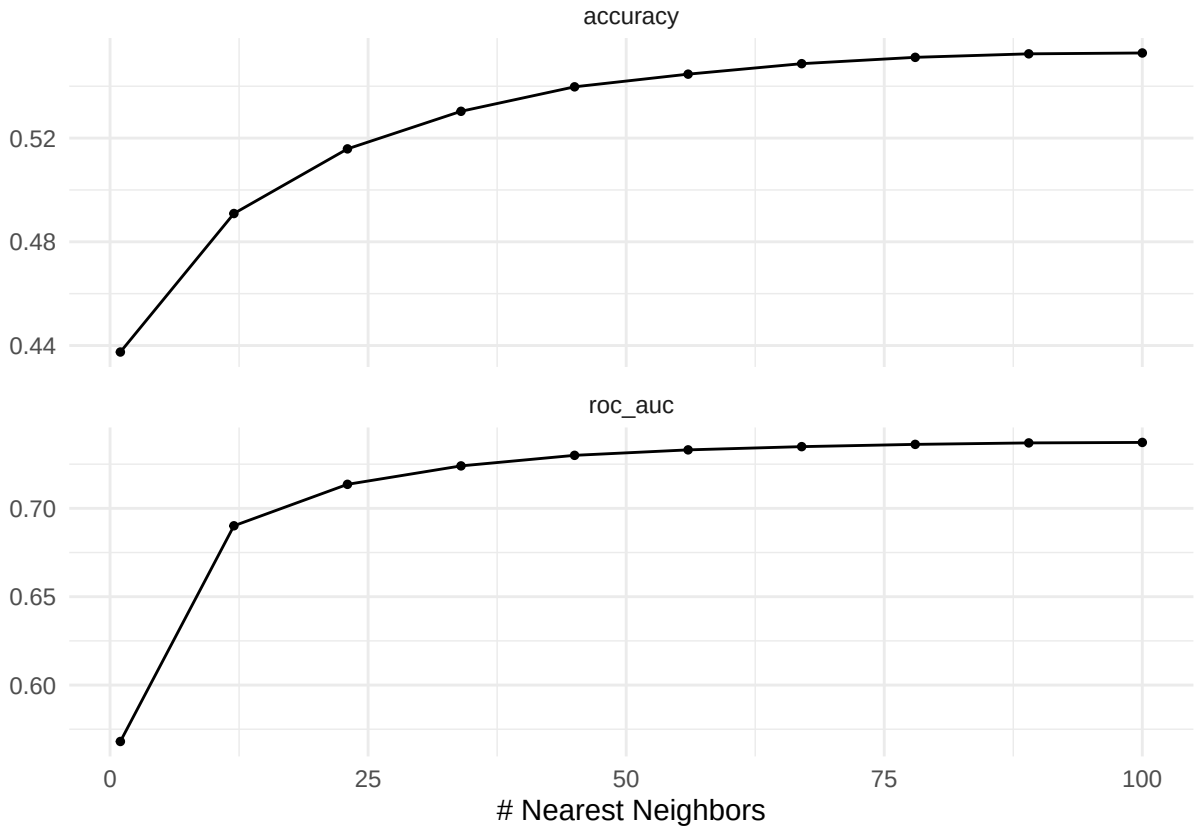
The most important predictor used for the Random Forest model is stack, followed by the action_turn calls class. Stack is the most unique variable out of all the predictors given that there aren't stages of the game associated with it so it makes sense why it'd be the most important.

K-Nearest Neighbors (KNN)

```
# Set up the model
knn_model <- nearest_neighbor(neighbors = tune()) %>%
  set_mode("classification") %>%
  set_engine("kknn")

# Set up workflow
knn_wf <- workflow() %>%
  add_model(knn_model) %>%
  add_recipe(poker_recipe)

# Set up grid
knn_grid <- grid_regular(neighbors(range = c(1, 100)), levels = 10)
```

```r
# Tune hyperparameters
tune_knn <- tune_grid(
  object = knn_wf,
  resamples = poker_folds,
  grid = knn_grid,
)

# Save the results to RDA file
save(tune_knn, file = "tune_knn.rda")
```

```r
load("tune_knn.rda")
autoplot(tune_knn) + theme_minimal()
```



Both the accuracy and ROC AUC are highest when the maximum number of neighbors are used at 100.

```r
show_best(tune_knn, n = 1)
```

```
## Warning: No value of `metric` was given; metric 'roc_auc' will be used.
```

```
## # A tibble: 1 x 7
##   neighbors .metric .estimator  mean     n std_err .config
##       <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1       100 roc_auc hand_till  0.737     5 0.00281 Preprocessor1_Model10
```

```r
best_knn_model <- select_best(tune_knn)
```

```
## Warning: No value of `metric` was given; metric 'roc_auc' will be used.
```

```r
knn_auc_score <- show_best(tune_knn, n = 1)['mean']
```

```
## Warning: No value of `metric` was given; metric 'roc_auc' will be used.
```

The best K-Nearest Neighbors Model is the one with 100 neighbors which produces a mean ROC AUC of 0.73.

```
final_knn_model <- finalize_workflow(knn_wf, best_knn_model)
final_knn_model <- fit(final_knn_model, poker_train)
```

Model Evaluation

To compare the five best ROC AUC scores for each model, I created a data frame that gives the estimate for the optimal model's AUC ROC score.

```
model_names <- c("Elastic Net", "Gradient Boosted Tree", "Pruned Decision Tree", "Random Forest", "K-Nea
combined_auc_table <- bind_rows(en_auc_score, gbt_auc_score, pdt_auc_score, rf_auc_score, knn_auc_score)

auc_table <- bind_cols(model_names, combined_auc_table)
```

```
## New names:
## * `` -> `...1`
```

```
auc_table <- auc_table %>%
  rename(Models = ...1,
         AUC_ROC = mean)
print(auc_table)
```

```
## # A tibble: 5 x 2
##   Models                AUC_ROC
##   <chr>                   <dbl>
## 1 Elastic Net             0.684
## 2 Gradient Boosted Tree   0.744
## 3 Pruned Decision Tree    0.712
## 4 Random Forest           0.740
## 5 K-Nearest Neighbors     0.737
```

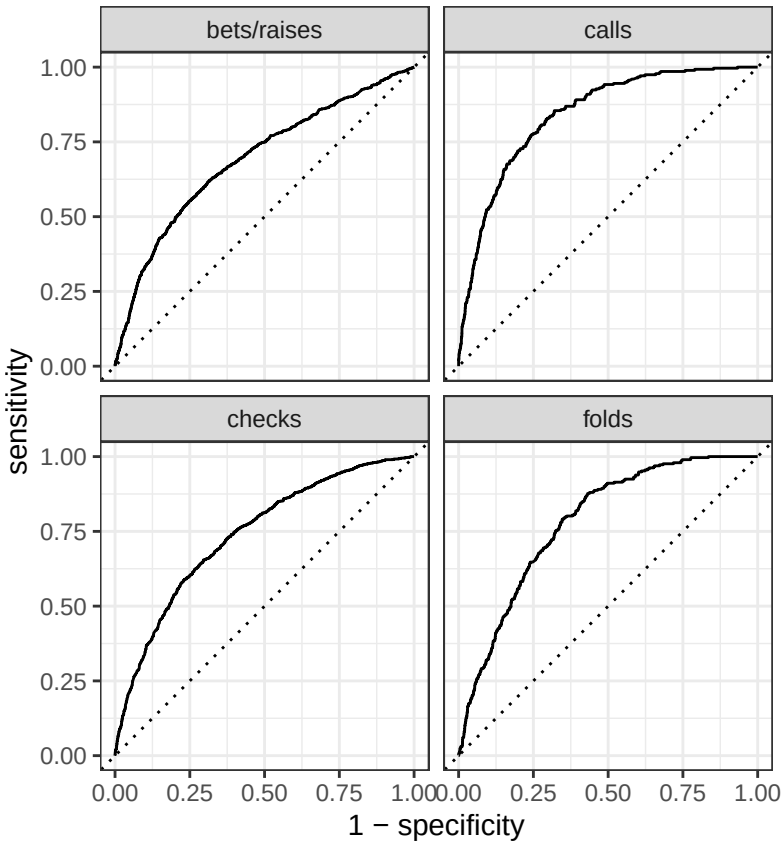
As you can see, the best model based off of AUC ROC is the Gradient Boosted Tree model with a score of 0.74. All of the other scores were around the same score except for Elastic Net which performed the worst. We will now fit our Gradient Boosted Model to our testing data and analyze the results.

```
poker_predict <- predict(final_gbt_model,  # fitting our model to testing data
                         new_data = poker_test,
                         type = "class")

poker_predict_with_actual <- poker_predict %>%
  bind_cols(poker_test)  # adding the actual values side by side to our predicted values

poker_augmented <- augment(final_gbt_model, new_data = poker_test) # for ROC
```

ROC Curve

Let's take a look at the ROC curve before we look at the final results.

```
poker_roc_curve <- augment(final_gbt_model, new_data = poker_test) %>%
  roc_curve(action_river, '.pred_bets/raises', .pred_calls, .pred_checks, .pred_folds)  # computing the

autoplot(poker_roc_curve)
```



These graphs show which classes the model is best at predicting. It seems that the GBT model does a good job of predicting the calls and folds classes as their graphs have the most area under the curve. The model is worst at predicting the bets/raises class.

Testing AUC ROC Results

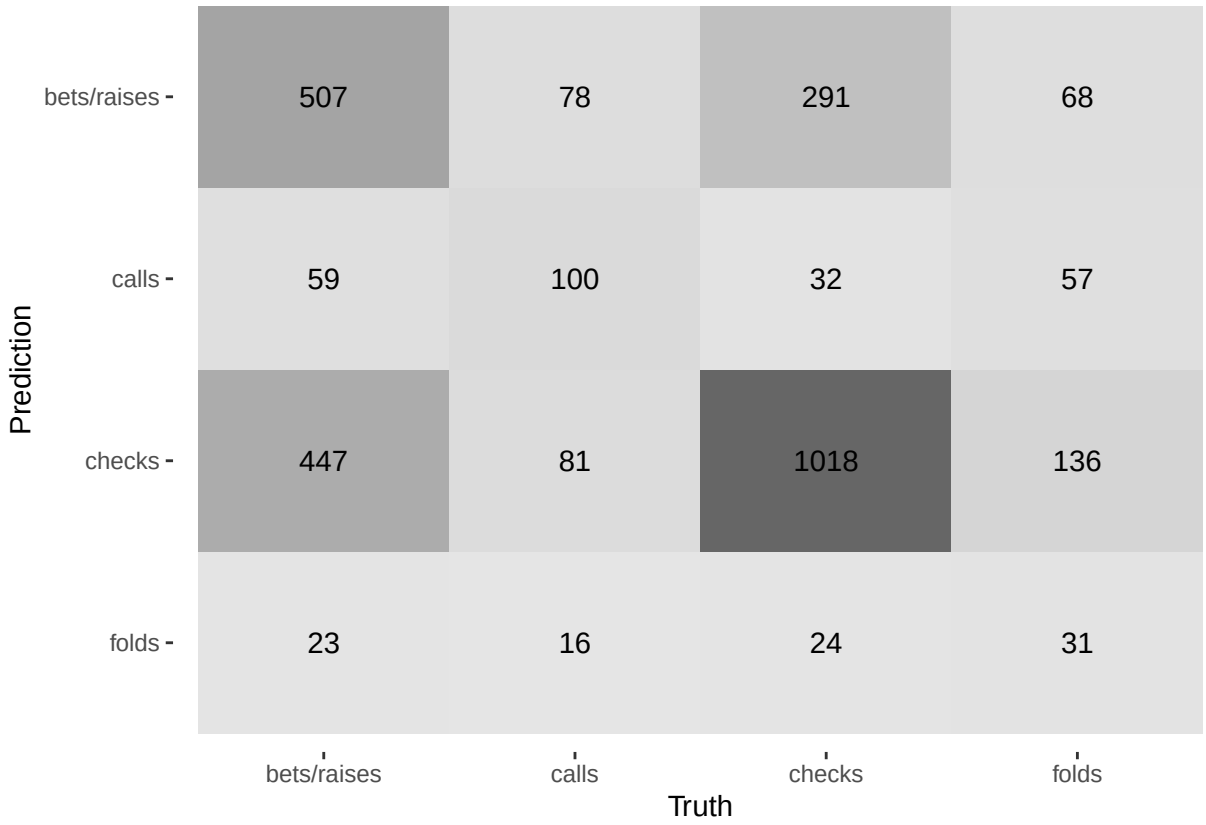
```
poker_roc_auc <- augment(final_gbt_model, new_data = poker_test) %>%
  roc_auc(action_river, '.pred_bets/raises', .pred_calls, .pred_checks, .pred_folds) %>%
  select(.estimate)  # computing the AUC for the ROC curve

poker_roc_auc
```

```
## # A tibble: 1 x 1
##   .estimate
##       <dbl>
## 1     0.741
```

The testing AUC ROC for the best Gradient Boosting Tree model is 0.75, just one percentage point better than from the training data. Not a great score compared to general Machine Learning models, but it is a good sign that our score got better when using the testing data.

Confusion Matrix of Final Model

```
conf_mat(poker_augmented, truth = action_river,
         .pred_class) %>%
  autoplot(type = "heatmap")
```



The model performs best at predicting checks correctly which differs from the ROC graphs. It's not very good at predicting folds at all.

Conclusion

The best model based off of the AUC ROC metric is the Gradient Boosted Tree model which achieved a AUC ROV of .75 on the testing data. This model only performed slightly better than the others during the training portion. The model seems to be very good at predicting checks correctly and very bad at predicting folds. This could be due to the fact that there was less data which had the class of folds to train on. Checks and folds are also similar in terms of poker play, meaning that they are not aggressive actions, so it makes sense why the model would predict check when in reality it was a fold most of the time. The model seemed to confuse checks with bets/raises a lot of the time as well which does not make sense as these actions are essentially opposites of one another.

I had a difficult time trying to figure out how to configure my response variable as well as the other action variables. I decided on condensing the response variable into four classes which seemed to simplify the analysis of results. However, I didn't do this for the other action variables (action_turn, action_flop, action_pre). All of these variables had at least 8 classes which may have added too much complexity for the models. So in the future I would like to try condesning those variables as well to see if we get different results. I would also like to train models on poker data with more than 3 people. This would allow me to study more traditional poker games. Overall, I was surprised most of the models were able to achieve on AUC ROC above .70. When I was first thinking about what I wanted my project to be about, I was nervous that the variables weren't going to provide enough information to the models, however I was happily proven wrong.