PSTAT 131 Homework 6

Luke Fields (8385924)

May 18, 2022

Below are the packages and libraries we are using in this assignment.

```
library(corrplot)
library(discrim)
library(corrr)
library(knitr)
library(MASS)
library(tidyverse)
library(tidymodels)
library(ggplot2)
library(glmnet)
library("dplyr")
library("yardstick")
library(ISLR)
library(rpart.plot)
library(vip)
library(janitor)
library(randomForest)
library(xgboost)
library(ranger)
library("parsnip")
tidymodels_prefer()
pokemon <- read_csv("Pokemon.csv")
# set global chunk options: images will be 7x5 inches
knitr::opts_chunk$set(
    echo = TRUE,
    fig.height = 5,
    fig.width = 7,
    tidy = TRUE,
    tidy.opts = list(width.cutoff = 60)
)
opts_chunk$set(tidy.opts=list(width.cutoff=60),tidy=TRUE)
options(digits = 4)



## indents are for indenting r code as formatted text
## They may need to be adjusted depending on your OS
# if your output looks odd, increase or decrease indent
indent1 = '    '
indent2 = '        '
indent3 = '            '
```

Exercise 1: Read in the data and set things up as in Homework 5: Use clean_names(), Filter out the rarer Pokémon types, Convert type_1 and legendary to factors, Do an initial split of the data; you can choose the percentage for splitting. Stratify on the outcome variable, Fold the training set using v-fold cross-validation, with v = 5. StratifY, on the outcome variable, Set up a recipe to predict type_1 with legendary, generation, sp_atk, attack, speed, defense, hp, and sp_def: Dummy-code legendary and generation; Center and scale all predictors.

First, we clean_names() on our Pokemon dataset.

```
library(janitor)
pokemon <- pokemon %>%
  clean_names
```

Then, we filter our the rarer pokemon types.

```
pokemon <- filter(pokemon, type_1 %in% c("Bug", "Fire", "Grass", "Normal", "Water", "Psychic"))
```

Next, we will convert type_1, legendary, and generation to factors within our data set.

```
pokemon$type_1 <- as.factor(pokemon$type_1)
pokemon$legendary <- as.factor(pokemon$legendary)
pokemon$generation <- as.factor(pokemon$generation)
```

Then, we will perform our initial split into our training and testing sets, stratifying on on our response variable type_1.

```
set.seed(912)
pokemon_split <- initial_split(pokemon,
                               prop = 0.7, strata = type_1)
pokemon_train <- training(pokemon_split)
pokemon_test <- testing(pokemon_split)
```

After that, we will fold the training data set using 5-fold cross validation.

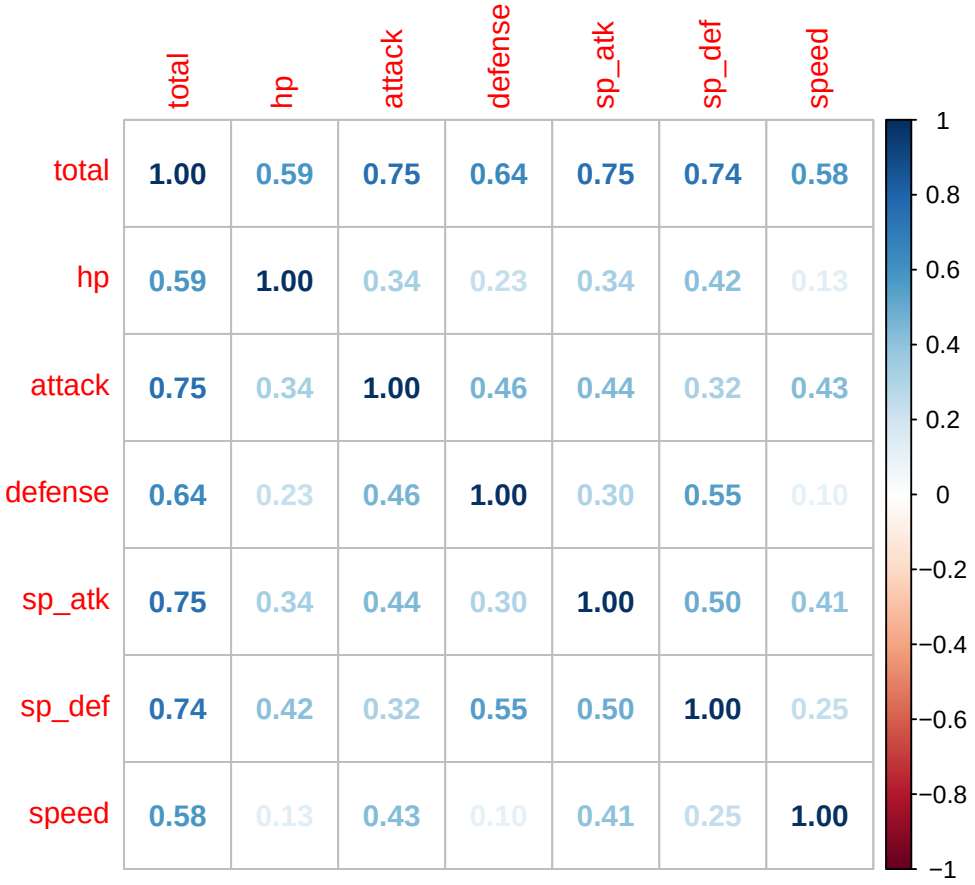
```
pokemon_folds <- vfold_cv(pokemon_train, v = 5, strata = type_1)
```

Finally, we make our recipe for the rest of the assignment.

```
pokemon_recipe <-
  recipe(type_1 ~ legendary + generation + sp_atk + attack +
           speed + defense + hp + sp_def, data = pokemon_train) %>%
  step_dummy(legendary) %>%
  step_dummy(generation) %>%
  step_normalize(all_predictors())
```

Exercise 2: Create a correlation matrix of the training set, using the corrplot package. Note: You can choose how to handle the continuous variables for this plot; justify your decision(s). What relationships, if any, do you notice? Do these relationships make sense to you?

```
pokemon_cont <- pokemon %>%
  select(total, hp, attack, defense, sp_atk, sp_def, speed)
pokemon_cor <- cor(pokemon_cont)
pokemon_cor_mat <- corrplot(pokemon_cor, method = "number")
```

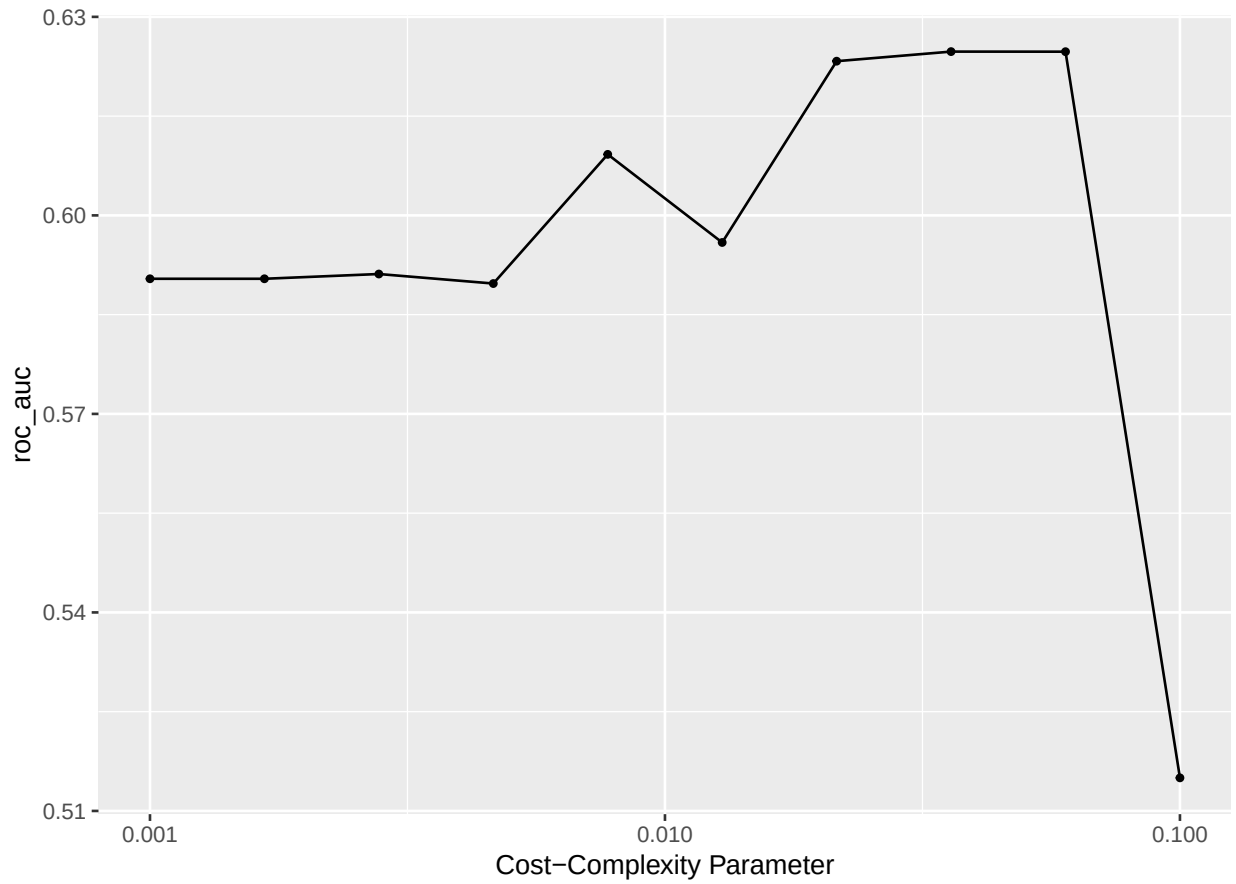


I chose the variables where the number being higher in value means it is greater, so all six of the battle statistics, as well as the sum of all of these statistics. There is obviously a significant relationship between all six of the battle statistics and total, because the higher each of those six values are, the higher the total is going to be. Pokemon with higher speed had less hp and less defense, which makes sense as a player of the game. One that surprised me is that I would think that the speed of attack and speed of defense would have a higher correlation with attack and speed ability.

Exercise 3: First, set up a decision tree model and workflow. Tune the cost_complexity hyperparameter. Use the same levels we used in Lab 7 – that is, range = c(-3, -1). Specify that the metric we want to optimize is roc_auc. Print an autoplot() of the results. What do you observe? Does a single decision tree perform better with a smaller or larger complexity penalty?

```
tree_spec <- decision_tree() %>%
  set_engine("rpart")

pokemon_class_tree_spec <- tree_spec %>%
```

```
  set_mode("classification")

pokemon_class_tree_wf <- workflow() %>%
  add_model(pokemon_class_tree_spec %>% set_args(cost_complexity = tune())) %>%
  add_recipe(pokemon_recipe)

pokemon_fold <- vfold_cv(pokemon_train)

pokemon_grid <- grid_regular(cost_complexity(range = c(-3, -1)), levels = 10)

pokemon_class_tune_res <- tune_grid(
  pokemon_class_tree_wf,
  resamples = pokemon_fold,
  grid = pokemon_grid,
  metrics = metric_set(roc_auc)
)

autoplot(pokemon_class_tune_res)
```



A single decision tree performs better with more complexity (penalty), but the best values are somewhere in the middle, as there is a large drop off at the cost-complexity parameter a little before it reaches 0.1, which is shown in the graph probably for Model 10.

Exercise 4: What is the roc_auc of your best-performing pruned decision tree on the folds? Hint: Use collect_metrics() and arrange().

```
pokemon_class_metrics <- collect_metrics(pokemon_class_tune_res)
arrange(pokemon_class_metrics, desc(mean))
```

```
## # A tibble: 10 x 7
##    cost_complexity .metric .estimator  mean     n std_err .config
##              <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1         0.0359  roc_auc hand_till  0.625    10  0.0235 Preprocessor1_Model08
##  2         0.0599  roc_auc hand_till  0.625    10  0.0186 Preprocessor1_Model09
##  3         0.0215  roc_auc hand_till  0.623    10  0.0219 Preprocessor1_Model07
##  4         0.00774 roc_auc hand_till  0.609    10  0.0164 Preprocessor1_Model05
##  5         0.0129  roc_auc hand_till  0.596    10  0.0181 Preprocessor1_Model06
##  6         0.00278 roc_auc hand_till  0.591    10  0.0139 Preprocessor1_Model03
##  7         0.001   roc_auc hand_till  0.590    10  0.0140 Preprocessor1_Model01
##  8         0.00167 roc_auc hand_till  0.590    10  0.0140 Preprocessor1_Model02
##  9         0.00464 roc_auc hand_till  0.590    10  0.0128 Preprocessor1_Model04
## 10         0.1     roc_auc hand_till  0.515    10  0.015  Preprocessor1_Model10
```

```
pokemon_class_best_rocauc <- collect_metrics(pokemon_class_tune_res) %>%
  select(-1) %>%
  arrange(desc(mean)) %>%
  filter(row_number()==1)

pokemon_class_best_rocauc
```

```
## # A tibble: 1 x 6
##   .metric .estimator  mean     n std_err .config
##   <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1 roc_auc hand_till  0.625    10  0.0235 Preprocessor1_Model08
```

The roc_auc of our best performing pruned decision tree is about 0.6094, which is found in Model 09.

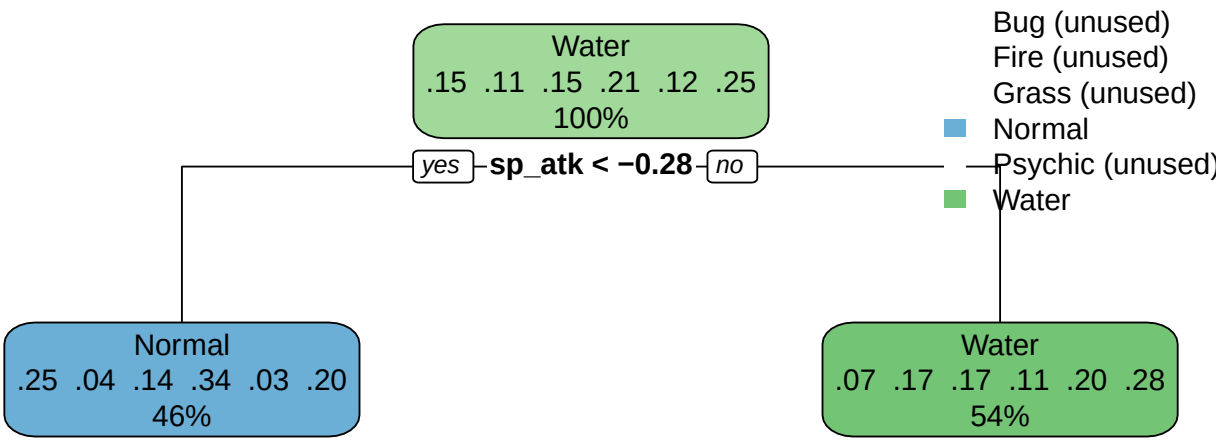
Exercise 5: Using rpart.plot, fit and visualize your best-performing pruned decision tree with the training set.

Below, we can see the visualization of our best performing decision tree with the training set.

```
pokemon_class_best <- select_best(pokemon_class_tune_res)

pokemon_class_tree_final <- finalize_workflow(pokemon_class_tree_wf, pokemon_class_best)

pokemon_class_tree_final_fit <- fit(pokemon_class_tree_final, data = pokemon_train)

pokemon_class_tree_final_fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```

```
## Warning: Cannot retrieve the data used to build the model (so cannot determine roundint and is.binary
```

```
## To silence this warning:
##     Call rpart.plot with roundint=FALSE,
##     or rebuild the rpart model with model=TRUE.
```



Exercise 5: Now set up a random forest model and workflow. Use the ranger engine and set importance = "impurity". Tune mtry, trees, and min_n. Using the documentation for rand_forest(), explain in your own words what each of these hyperparameters represent. Create a regular grid with 8 levels each. You can choose plausible ranges for each hyperparameter. Note that mtry should not be smaller than 1 or larger than 8. Explain why not. What type of model would mtry = 8 represent?

```
pokemon_rf_spec <- rand_forest(mtry = tune(), trees = tune(), min_n = tune()) %>%
  set_engine("randomForest", importance = TRUE) %>%
  set_mode("classification")

pokemon_rf_wf <- workflow() %>%
  add_model(pokemon_rf_spec) %>%
  add_recipe(pokemon_recipe)


pokemon_rf_grid <- grid_regular(mtry(range = c(1,8)),
                                trees(range = c(1,10)),
                                min_n(range = c(1,10)),
```

```
                                levels = 8)

pokemon_rf_grid
```

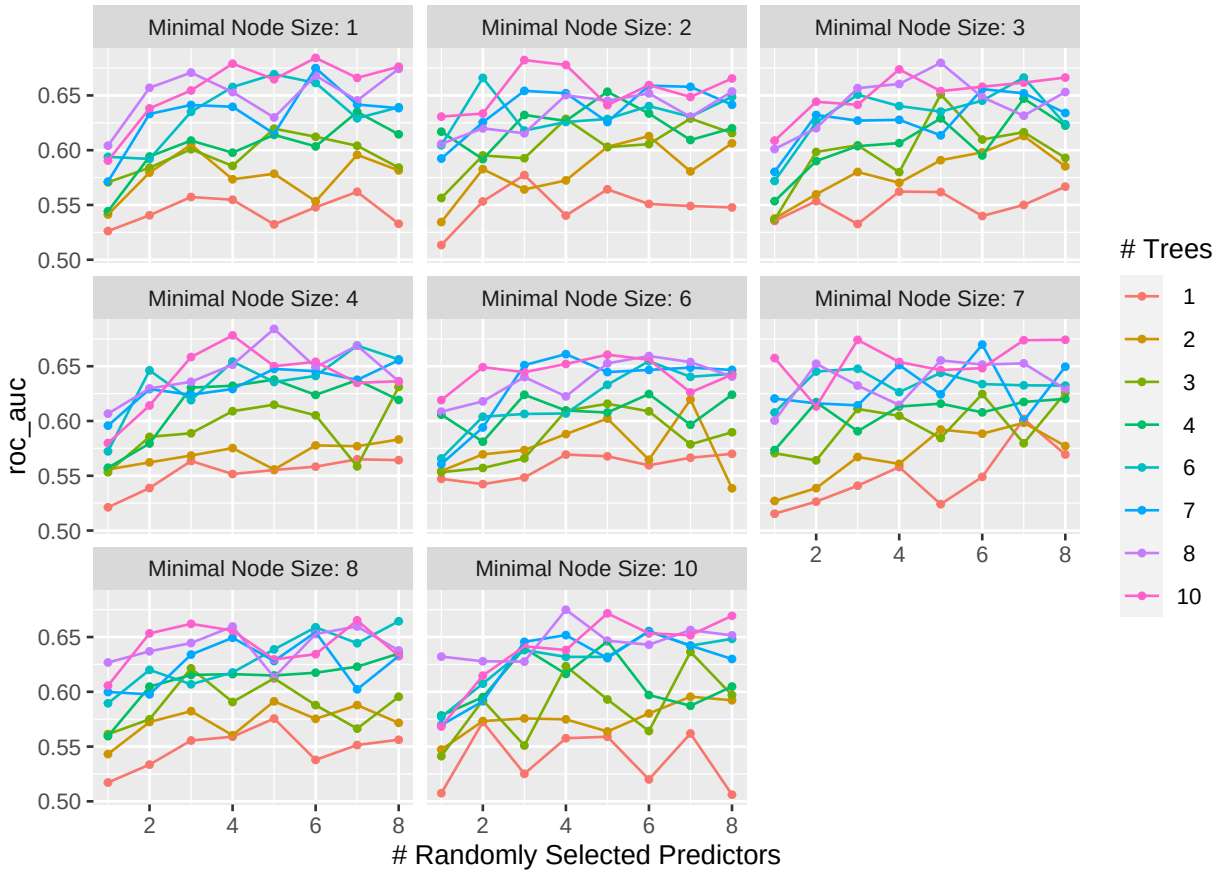
```
## # A tibble: 512 x 3
##     mtry trees min_n
##    <int> <int> <int>
##  1     1     1     1
##  2     2     1     1
##  3     3     1     1
##  4     4     1     1
##  5     5     1     1
##  6     6     1     1
##  7     7     1     1
##  8     8     1     1
##  9     1     2     1
## 10     2     2     1
## # ... with 502 more rows
```

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. min_n: The minimum number of data values needed to create another split.

An mtry of 8 would represent that 8 predictors would be randomly sampled at each creation of a tree split, and mtry should not be smaller than 1 or larger than 8 because it has to be within the levels of the predictors in our grid. If we had mtry = 0, there would be no subset of predictors at all, and if we had an mtry = 9, we would then be bagging instead of using the random forest method.

Exercise 6: Specify roc_auc as a metric. Tune the model and print an autoplot() of the results. What do you observe? What values of the hyperparameters seem to yield the best performance?

```
pokemon_rf_tune_res <- tune_grid(
  pokemon_rf_wf,
  resamples = pokemon_fold,
  grid = pokemon_rf_grid,
  metrics = metric_set(roc_auc)
)

autoplot(pokemon_rf_tune_res)
```



Looking at the plot of the roc_auc performance of our models, it is obvious that the different values of mtry, trees, and min_n play a large role in how well our model does. For mtry, typically roc_auc performance improved with a greater amount of randomly selected predictors ,but the highest values of roc_auc were found when there was somewhere between 5 and 7 predictors being randomly sampled every time. For trees, there is significantly greater roc_auc scores for the models that had a greater number of trees grown in the model, with the best model performances all featuring 8, 9, or 10 trees. For min_n, this is the hyperparameter with the least significance of the three, but the best model came with a minimal node size of 6, so the closer to the median of minimal node size the model contained, the better it performed most of the time.

Exercise 7: What is the roc_auc of your best-performing random forest model on the folds? Hint: Use collect_metrics() and arrange().

```
pokemon_rf_metrics <- collect_metrics(pokemon_rf_tune_res)
arrange(pokemon_rf_metrics, desc(mean))
```

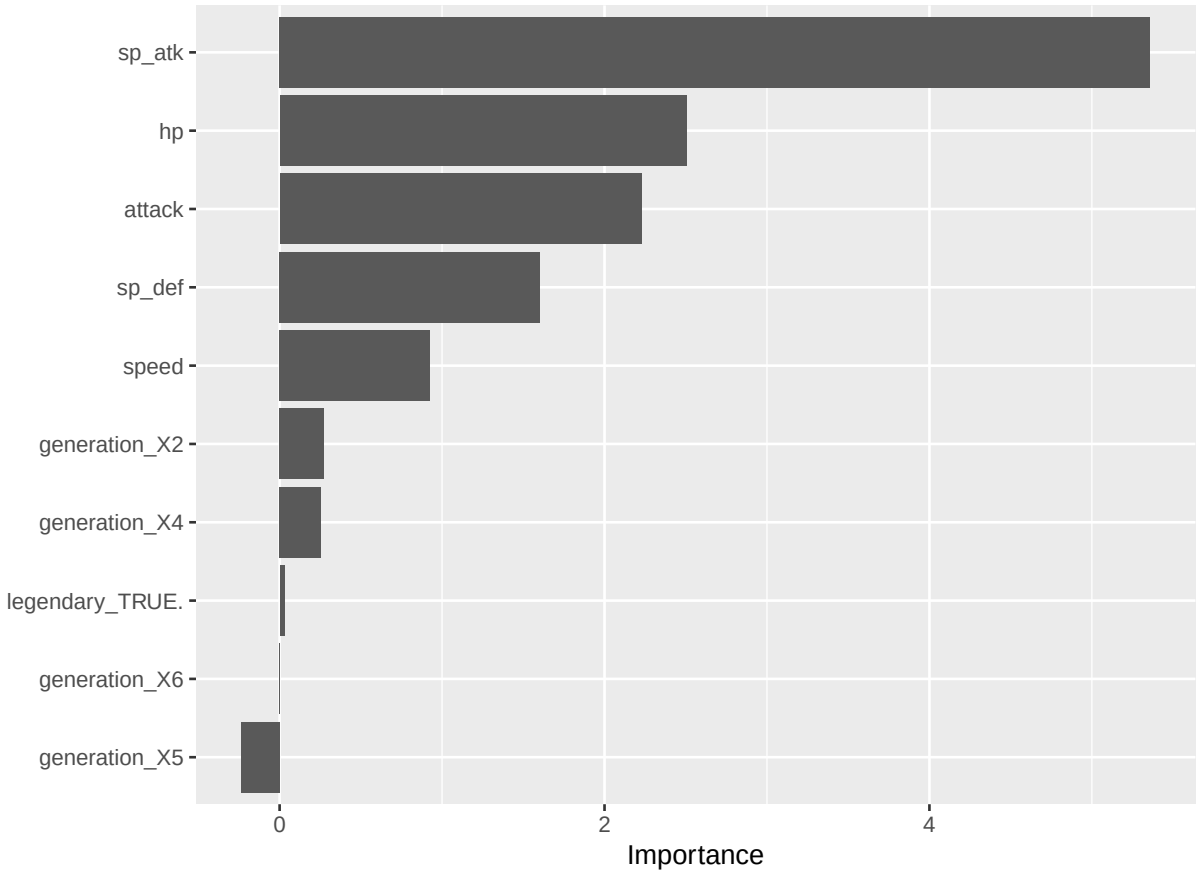
```
## # A tibble: 512 x 9
##    mtry trees min_n .metric .estimator  mean     n std_err .config
##   <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1     6    10     1 roc_auc hand_till  0.684    10  0.0185 Preprocessor1_Model~
## 2     5     8     4 roc_auc hand_till  0.684    10  0.0209 Preprocessor1_Model~
## 3     3    10     2 roc_auc hand_till  0.682    10  0.0231 Preprocessor1_Model~
## 4     5     8     3 roc_auc hand_till  0.680    10  0.0187 Preprocessor1_Model~
## 5     4    10     1 roc_auc hand_till  0.679    10  0.0225 Preprocessor1_Model~
```

```
##  6     4    10     4 roc_auc hand_till  0.678    10  0.0181 Preprocessor1_Model~
##  7     4    10     2 roc_auc hand_till  0.678    10  0.0214 Preprocessor1_Model~
##  8     8    10     1 roc_auc hand_till  0.676    10  0.0176 Preprocessor1_Model~
##  9     6     7     1 roc_auc hand_till  0.675    10  0.0199 Preprocessor1_Model~
## 10     4     8    10 roc_auc hand_till  0.675    10  0.0193 Preprocessor1_Model~
## # ... with 502 more rows
```

Model 303 had the best roc_auc score of the random forest model, with a mean value of 0.0707.

Exercise 8: Create a variable importance plot, using vip(), with your best-performing random forest model fit on the training set. Which variables were most useful? Which were least useful? Are these results what you expected, or not?

```
library(vip)
pokemon_rf_best <- select_best(pokemon_rf_tune_res)

pokemon_rf_final <- finalize_workflow(pokemon_rf_wf, pokemon_rf_best)

pokemon_rf_final_fit <- fit(pokemon_rf_final, data = pokemon_train)

pokemon_rf_final_fit %>% extract_fit_engine() %>%vip()
```

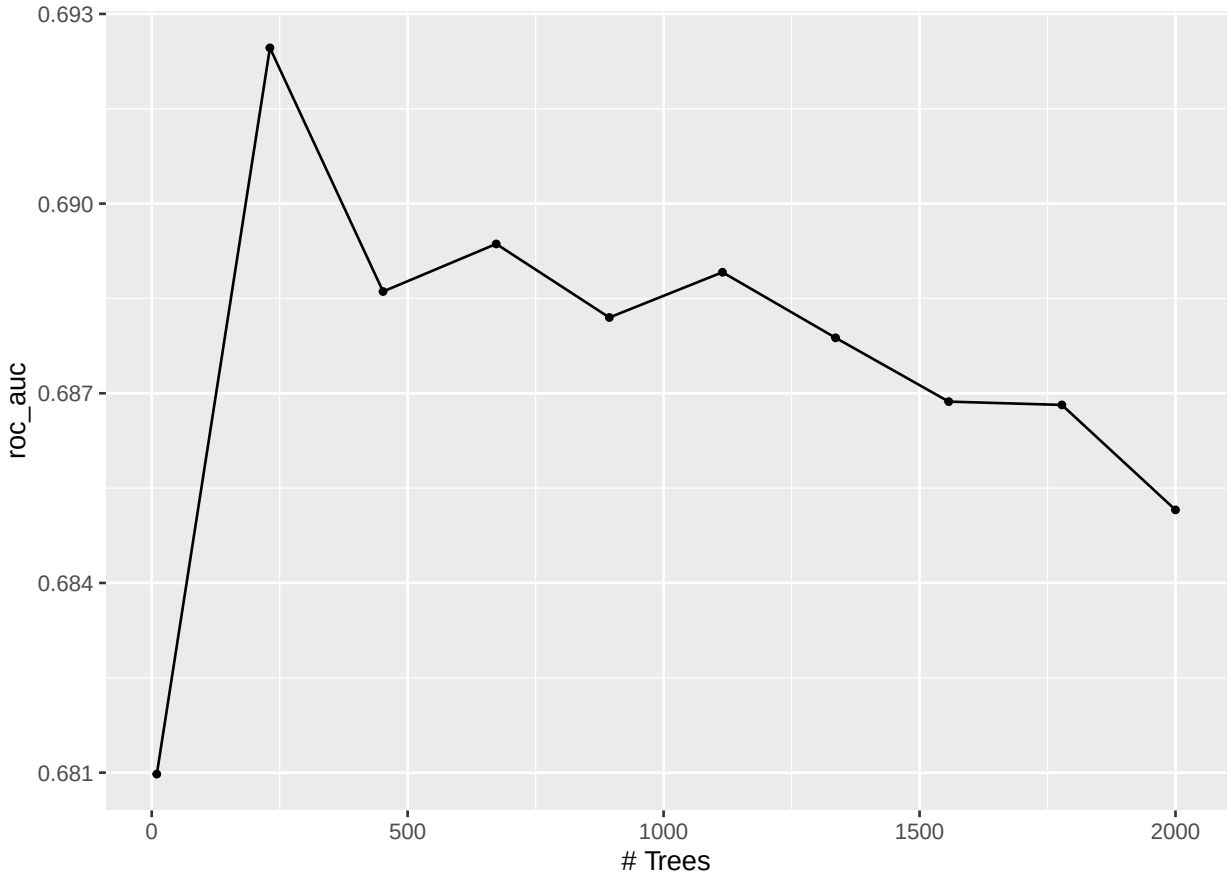


Looking at the variable importance chart, the results remain pretty consistent with the fundamentals of the pokemon game. The variables that were the least useful were the generation of the pokemon, which makes

sense because most of the pokemon games have fairly even power distributions across its generations, so it is a little wodd that a pokemon being in generation 4 played a significant role on the prediction of its type. HP and Speed of Attack are the most important variables in this model, which also makes sense as these statistics vary drastically between the different types of pokemon. I am a little surprised that defense has as little of importance as one would think that such an important battle statistic would set different pokemon types apart from each other.

Exercise 9: Finally, set up a boosted tree model and workflow. Use the xgboost engine. Tune trees. Create a regular grid with 10 levels; let trees range from 10 to 2000. Specify roc_auc and again print an autoplot() of the results. What do you observe? What is the roc_auc of your best-performing boosted tree model on the folds? Hint: Use collect_metrics() and arrange().

```
pokemon_boost_spec <- boost_tree() %>%
  set_engine("xgboost") %>%
  set_mode("classification")

pokemon_boost_wf <- workflow() %>%
  add_model(pokemon_boost_spec %>%
              set_args(trees = tune())) %>%
  add_recipe(pokemon_recipe)

pokemon_boost_grid <- grid_regular(trees(range = c(10,2000)),
                                   levels = 10)

pokemon_boost_tune_res <- tune_grid(pokemon_boost_wf,
                                    resamples = pokemon_fold,
                                    grid = pokemon_boost_grid,
                                    metrics = metric_set(roc_auc))

autoplot(pokemon_boost_tune_res)
```



From this plot, we can see that the number of trees in our boosted model typically caused the roc_auc performance to drop as they increased. However, if you look at the scale on the y-axis, the difference between the lowest roc_auc score of 0.681 with no trees and the highest roc_auc score of 0.693 with about 250 trees shows that there is a maximum of a 0.012 discrepancy between roc_auc scores across the different number of trees. So, overall, roc_auc will tpyically decrease as the number of trees increase, but not in any significant way.

```
pokemon_boost_metrics <- collect_metrics(pokemon_boost_tune_res)
arrange(pokemon_boost_metrics, desc(mean))
```

```
## # A tibble: 10 x 7
##    trees .metric .estimator  mean     n std_err .config
##    <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1   231 roc_auc hand_till  0.692    10  0.0176 Preprocessor1_Model02
##  2   673 roc_auc hand_till  0.689    10  0.0166 Preprocessor1_Model04
##  3  1115 roc_auc hand_till  0.689    10  0.0165 Preprocessor1_Model06
##  4   452 roc_auc hand_till  0.689    10  0.0159 Preprocessor1_Model03
##  5   894 roc_auc hand_till  0.688    10  0.0168 Preprocessor1_Model05
##  6  1336 roc_auc hand_till  0.688    10  0.0159 Preprocessor1_Model07
##  7  1557 roc_auc hand_till  0.687    10  0.0155 Preprocessor1_Model08
##  8  1778 roc_auc hand_till  0.687    10  0.0154 Preprocessor1_Model09
##  9  2000 roc_auc hand_till  0.685    10  0.0154 Preprocessor1_Model10
## 10    10 roc_auc hand_till  0.681    10  0.0216 Preprocessor1_Model01
```

Model 2 had the best roc_auc score of the random forest model, with a mean value of 0.6925.

Exercise 10: Display a table of the three ROC AUC values for your best-performing pruned tree, random forest, and boosted tree models. Which performed best on the folds? Select the best of the three and use select_best(), finalize_workflow(), and fit() to fit it to the testing set. Print the AUC value of your best-performing model on the testing set. Print the ROC curves. Finally, create and visualize a confusion matrix heat map. Which classes was your model most accurate at predicting? Which was it worst at?

First, we display the three ROC AUC values for our best performing decision tree, random forest, and boosted tree.

```
pokemon_class_best_rocauc <- collect_metrics(pokemon_class_tune_res) %>%
  select(-1) %>%
  arrange(desc(mean)) %>%
  filter(row_number()==1)

pokemon_rf_best_rocauc <- collect_metrics(pokemon_rf_tune_res) %>%
  select(-1, -2, -3) %>%
  arrange(desc(mean)) %>%
  filter(row_number()==1)

pokemon_boost_best_rocauc <- collect_metrics(pokemon_boost_tune_res) %>%
  select(-1) %>%
  arrange(desc(mean)) %>%
  filter(row_number()==1)

pokemon_rocauc_collect <- rbind(pokemon_class_best_rocauc,
                                pokemon_rf_best_rocauc,
                                pokemon_boost_best_rocauc)

pokemon_model_types <- c("Decision Tree", "Random Forest", "Boosted Tree")
pokemon_rocauc_collect <- cbind(pokemon_model_types, pokemon_rocauc_collect)

pokemon_rocauc_collect %>%
  select(1, 2, 4, 6, 7)
```

```
##   pokemon_model_types .metric   mean std_err                .config
## 1       Decision Tree roc_auc 0.6247 0.02350  Preprocessor1_Model108
## 2       Random Forest roc_auc 0.6842 0.01849 Preprocessor1_Model1062
## 3        Boosted Tree roc_auc 0.6925 0.01755  Preprocessor1_Model102
```

From this table, we will select the random forest model 303 as our best performing model to fit to our testing dataset.

Then, we will fit our best model to the testing set.

```
pokemon_rf_best <- select_best(pokemon_rf_tune_res)

pokemon_rf_final <- finalize_workflow(pokemon_rf_wf, pokemon_rf_best)

pokemon_trees_final_fit <- fit(pokemon_rf_final, data = pokemon_test)
```

After that, let's take a peak at the AUC for our ROC curve for our best performing model.

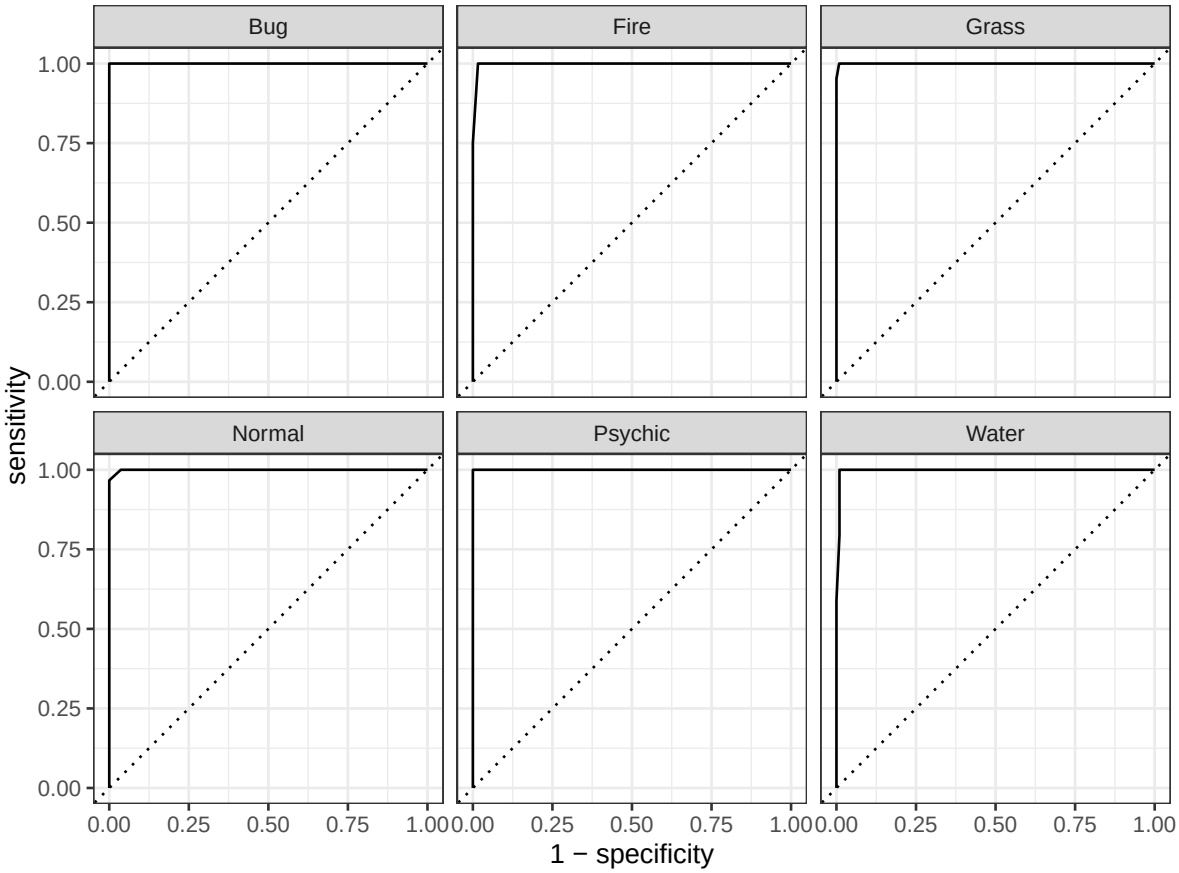
```
augment(pokemon_trees_final_fit, new_data = pokemon_test) %>%
  roc_auc(truth = type_1, estimate = c(".pred_Bug", ".pred_Fire", ".pred_Grass",
                                       ".pred_Normal", ".pred_Psychic", ".pred_Water")) %>%
  select(.estimate)
```

```
## # A tibble: 1 x 1
##   .estimate
##       <dbl>
## 1     0.999
```

The AUC value for our best performing model when applied to the dataset is about 0.9615.

Next, we will get the ROC curves for our best model's performance in predicting each Pokemon type.

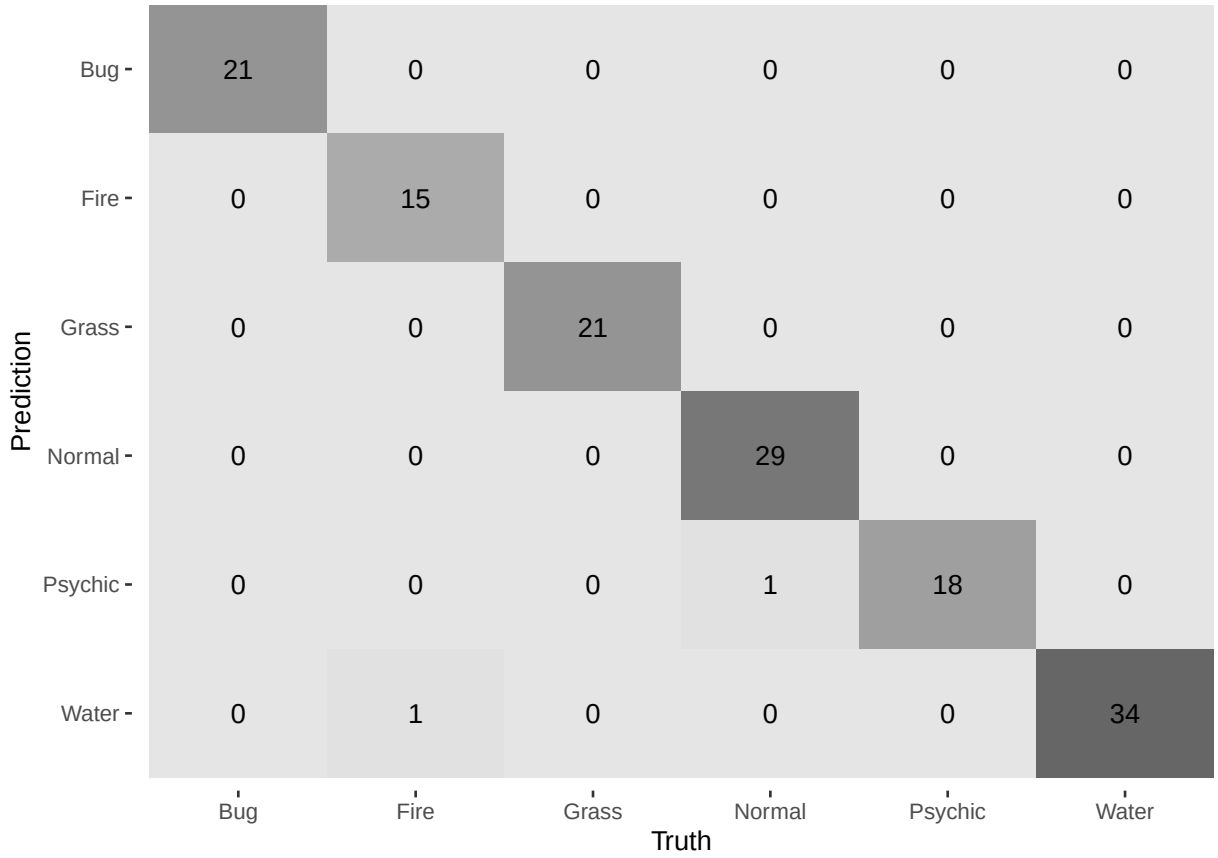
```
pokemon_roc_best_tree <- augment(pokemon_trees_final_fit, new_data = pokemon_test) %>%
  roc_curve(truth = type_1, estimate = c(".pred_Bug", ".pred_Fire", ".pred_Grass",
                                       ".pred_Normal", ".pred_Psychic", ".pred_Water"))

autoplot(pokemon_roc_best_tree)
```



Finally, we print the confusion matrix heatmap for our best model.

```
pokemon_tree_confus_mat <- augment(pokemon_trees_final_fit, new_data = pokemon_test) %>%
  conf_mat(truth = type_1, estimate = .pred_class)
```

```
autoplot(pokemon_tree_confus_mat, type = "heatmap")
```



Overall, our random forest model worked excellently in trying to predict the type of a Pokemon. Right off the bat, an AUC score for the ROC curve of 0.9615 is fantastic. Looking at the curves, we can see all of the Pokemon types have their ROC curve pretty much as up and to the left as it could be. Finally, taking a look at the heatmap for the confusion matrix, the darkest shades are along the diagonal, meaning that the most predictions made for a Pokemon on its type were actually correct. The model worked best at predicting Pokemon with Normal type, and was least effective (but still quite effective!) when evaluating Bug type Pokemon.

END OF HOMEWORK 6