PSTAT 131 HW6

Jenny Do (5669841)

Tree-Based Models

For this assignment, we will continue working with the file `"pokemon.csv"`, found in `/data`. The file is from Kaggle: https://www.kaggle.com/abcsds/pokemon.

The Pokémon franchise encompasses video games, TV shows, movies, books, and a card game. This data set was drawn from the video game series and contains statistics about 721 Pokémon, or "pocket monsters." In Pokémon games, the user plays as a trainer who collects, trades, and battles Pokémon to (a) collect all the Pokémon and (b) become the champion Pokémon trainer.

Each Pokémon has a primary type (some even have secondary types). Based on their type, a Pokémon is strong against some types, and vulnerable to others. (Think rock, paper, scissors.) A Fire-type Pokémon, for example, is vulnerable to Water-type Pokémon, but strong against Grass-type.

The goal of this assignment is to build a statistical learning model that can predict the **primary type** of a Pokémon based on its generation, legendary status, and six battle statistics.

Note: Fitting ensemble tree-based models can take a little while to run. Consider running your models outside of the .Rmd, storing the results, and loading them in your .Rmd to minimize time to knit.

```
# Load in needed packages
library(tidymodels)
library(ISLR)
library(tidyverse)
tidymodels_prefer()
library(corrr)

library(ggplot2)
library(dplyr)
library(glmnet)
library(pROC)
library(rpart)
#visualization
library(rpart.plot)
library(conflicted)
conflict_prefer("prune", "rpart")

library(ranger)

library(vip)

library(xgboost)

# set seed
set.seed(27)
```

Exercise 1

Read in the data and set things up as in Homework 5:

```
# Load in the data
pokemon <- read.csv("Pokemon.csv")

# Set seed
set.seed(27)
```

- Use `clean_names()`

```
# Install and load the 'janitor' package
library(janitor)

# Use clean_names() function on the pokemon data
# save results to work with for rest of assignment
pokemon <- clean_names(pokemon)
```

- Filter out the rarer Pokémon types

```
# Use the filter function and grepl() function to filter rows that contains a certain string
pokemon <- pokemon %>% filter(grepl("Bug|Fire|Grass|Normal|Water|Psychic", type_1))

# Verify that we have filtered the entire data set to
# contain only Pokémon whose `type_1` is Bug, Fire,
# Grass, Normal, Water, or Psychic.

# use table function - we can see that we did filter out the rarer pokemon
table(pokemon$type_1)
```

```
##
##     Bug    Fire   Grass  Normal Psychic   Water
##      69      52      70      98      57     112
```

- Convert `type_1` and `legendary` to factors

```
# use function factor() to convert `type_1` and `legendary` to factors
pokemon$type_1 <- factor(pokemon$type_1)
# verify that type_1 is now a factor
class(pokemon$type_1)
```

```
## [1] "factor"
```

```
pokemon$legendary <- factor(pokemon$legendary)
# verify that type_1 is now a factor
class(pokemon$legendary)
```

```
## [1] "factor"
```

```
# (Per TA's recommendation) Here we also want to make "generation" a factor -
# This is so that #4 does not throw an error
pokemon$generation <- factor(pokemon$generation)
```

Do an initial split of the data; you can choose the percentage for splitting. Stratify on the outcome variable.

```
# Initial split, stratifying on outcome variable type_1
pokemon_split <- initial_split(pokemon, prop = 0.80,
                               strata="type_1")
pokemon_split
```

```
## <Analysis/Assess/Total>
## <364/94/458>
```

```
# training
pokemon_train <- training(pokemon_split)
# testing
pokemon_test <- testing(pokemon_split)

# Verifying that training and test set shave the desired number of obs
dim(pokemon_train);dim(pokemon_test)
```

```
## [1] 364  13
```

```
## [1] 94 13
```

Fold the training set using *v*-fold cross-validation, with `v = 5`. Stratify on the outcome variable.

```
# use the vfold_cv() function, k/v = 5
# use the strata = type_1 argument
pokemon_folds <- vfold_cv(pokemon_train, v = 5, strata = type_1)
pokemon_folds
```

```
## #  5-fold cross-validation using stratification
## # A tibble: 5 x 2
##   splits           id
##   <list>           <chr>
## 1 <split [289/75]> Fold1
## 2 <split [291/73]> Fold2
## 3 <split [291/73]> Fold3
## 4 <split [292/72]> Fold4
## 5 <split [293/71]> Fold5
```

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.

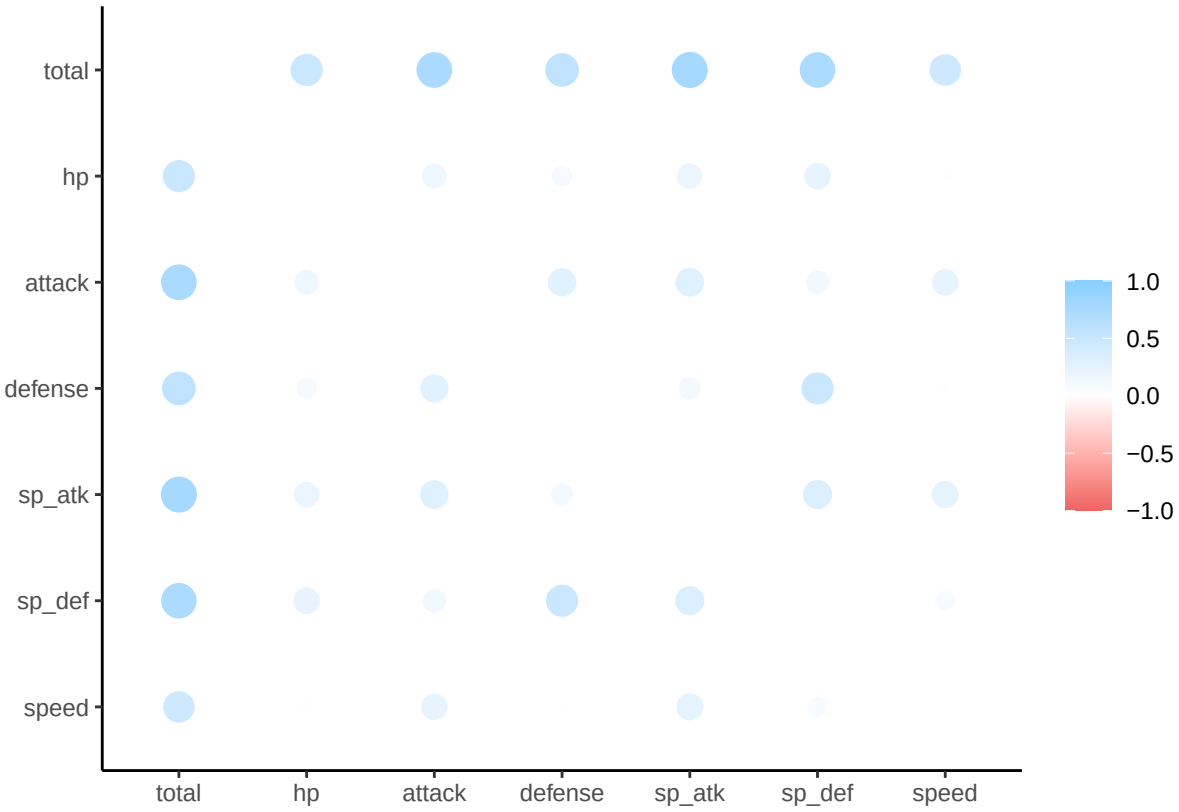
```r
# set up recipe using recipe() function, with `legendary`,
#`generation`, `sp_atk`, `attack`, `speed`, `defense`, `hp`, and `sp_def
pokemon_recipe <- recipe(type_1 ~ legendary + generation + sp_atk +
                           attack + speed + defense + hp
                         + sp_def, data = pokemon_train) %>%
  # step_dummy to dummy code legendary and generation
  step_dummy(all_nominal_predictors())%>%
  # center and scale all predictors
  step_normalize(all_predictors())
pokemon_recipe
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          8
##
## Operations:
##
## Dummy variables from all_nominal_predictors()
## Centering and scaling for 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).*

```r
# Create a correlation matrix - visual
cor_pokemon_train <- pokemon_train %>%
  # remove all discrete variables
  select(-c(x,name, type_1, type_2, legendary,generation)) %>%
  correlate()
  # create a visualization of the matrix
rplot(cor_pokemon_train)
```



What relationships, if any, do you notice? Do these relationships make sense to you?

Answer: The most notable relationship I notice is that total is strongly correlated to a lot of the predictors. This makes sense because total is the sum of all stats.

In addition, I also notice that sp_def and defense is strongly (positive) correlated, this also makes sense because if you are strong against special attacks you should also be strong against normal attacks. This relationship can be seen with other variables (ie. attack and sp_attack).

Another relationship I see is sp_attack and sp_defense. This also makes sense. This clues me in that pokemon that are more evolved with special abilites are stronger all around.

Exercise 3

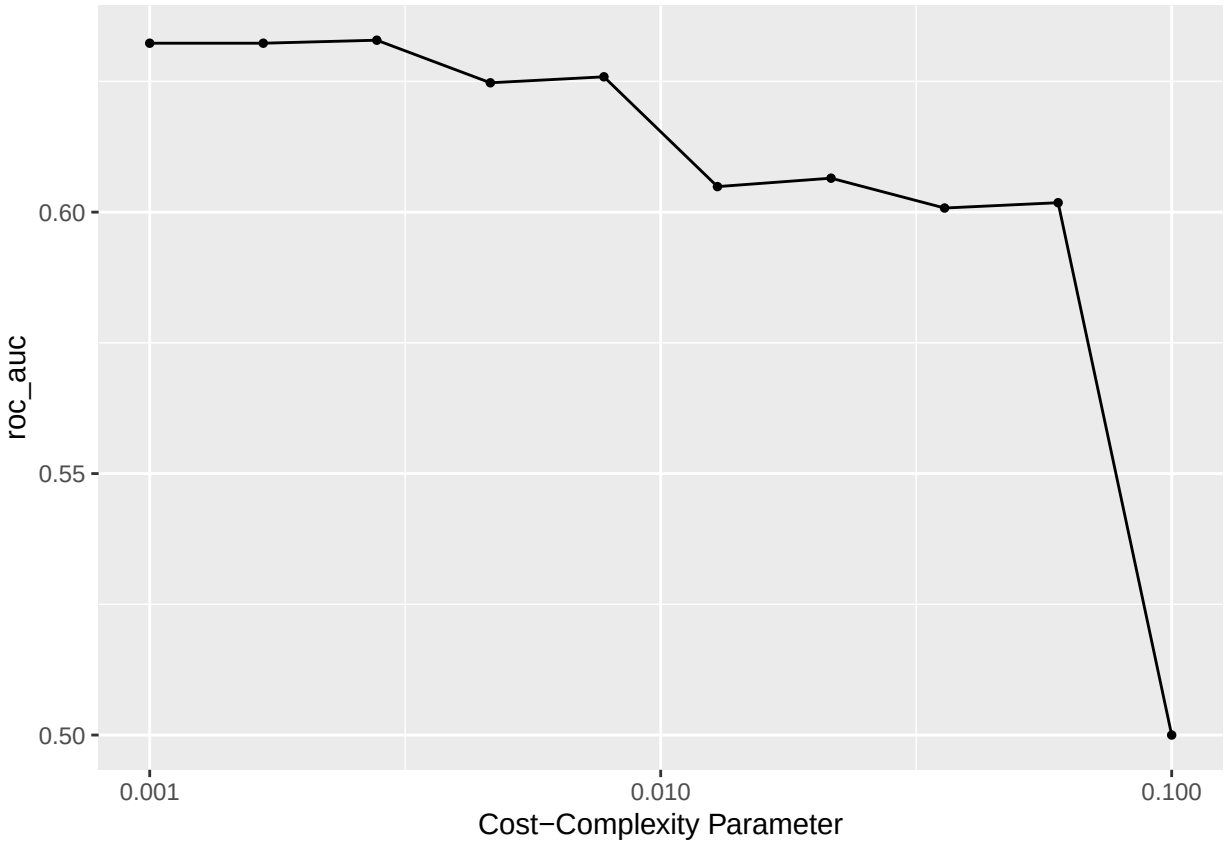
First, set up a decision tree model and workflow. Tune the `cost_complexity` hyperparameter. Use the same levels we used in Lab 7 – that is, `range = c(-3, -1)`. Specify that the metric we want to optimize is `roc_auc`.

```
# Set up the decision tree
tree_spec <- decision_tree() %>%
  set_engine("rpart")
class_tree_spec <- tree_spec %>%
  set_mode("classification")

# Set up the workflow
class_tree_wf <- workflow() %>%
  add_model(class_tree_spec %>% set_args(cost_complexity = tune())) %>%
  add_recipe(pokemon_recipe)
```

```r
# tuning the cost_complexity parameter, using range = c(-3,-1) and 10 levels
# got this from Lab 7

# make the regular grid
param_grid <- grid_regular(cost_complexity(range = c(-3, -1)), levels = 10)

tune_res <- tune_grid(
  class_tree_wf,
  resamples = pokemon_folds,
  grid = param_grid,
  metrics = metric_set(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?

```r
autoplot(tune_res)
```



After printing out an autoplot() of the results, we observe that the model starts to do really poorly as the cost-complexity parameter increases. Therefore a single decision tree preforms better with a smaller complexity penalty.

Exercise 4

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

```r
# First we can use the functions collect_metrics() and arrange() to look at the results
collect_metrics(tune_res) %>% arrange(-mean)
```

```
## # A tibble: 10 x 7
##    cost_complexity .metric .estimator  mean     n std_err .config
##              <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1         0.00278 roc_auc hand_till  0.633     5 0.00873 Preprocessor1_Model03
##  2         0.001   roc_auc hand_till  0.632     5 0.00858 Preprocessor1_Model01
##  3         0.00167 roc_auc hand_till  0.632     5 0.00858 Preprocessor1_Model02
##  4         0.00774 roc_auc hand_till  0.626     5 0.0157  Preprocessor1_Model05
##  5         0.00464 roc_auc hand_till  0.625     5 0.0107  Preprocessor1_Model04
##  6         0.0215  roc_auc hand_till  0.606     5 0.0271  Preprocessor1_Model07
##  7         0.0129  roc_auc hand_till  0.605     5 0.0150  Preprocessor1_Model06
##  8         0.0599  roc_auc hand_till  0.602     5 0.0211  Preprocessor1_Model09
##  9         0.0359  roc_auc hand_till  0.601     5 0.0176  Preprocessor1_Model08
## 10         0.1     roc_auc hand_till  0.5       5 0       Preprocessor1_Model10
```

```r
# Then we can use select_best function to choose the best complexity
best_complexity <- select_best(tune_res, metric = "roc_auc")
class_tree_final <- finalize_workflow(class_tree_wf, best_complexity)
```

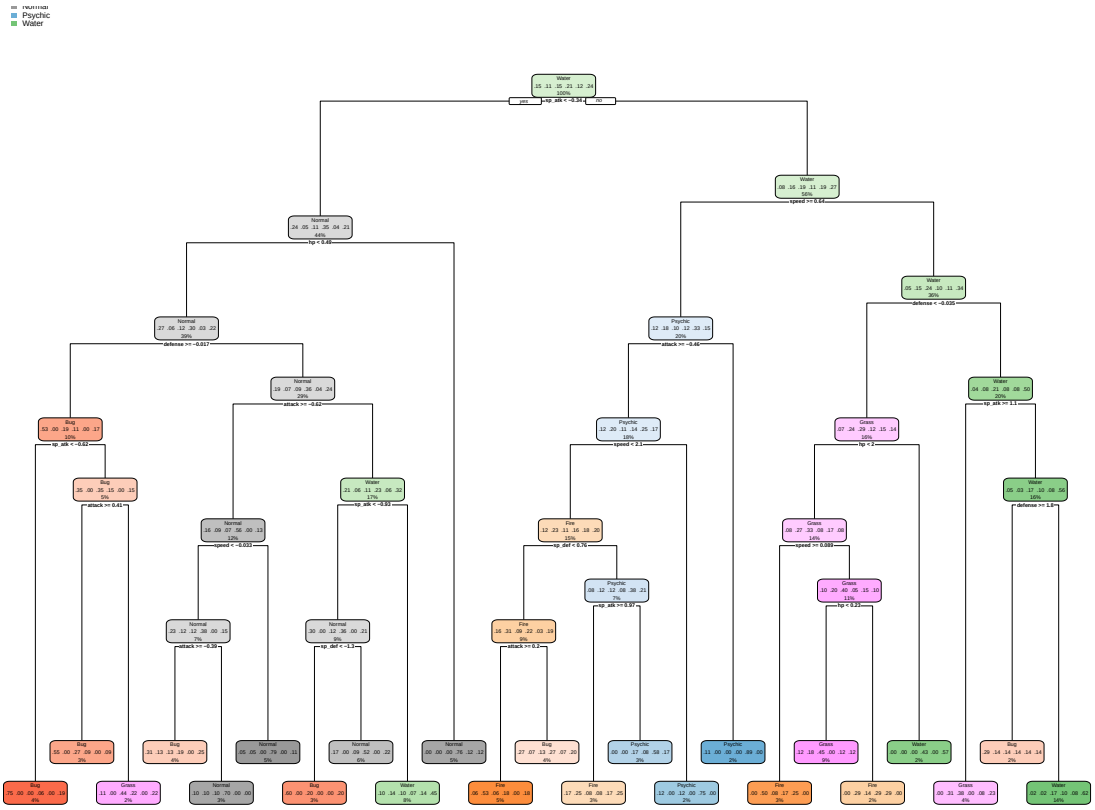
The roc_auc of the best preforming pruned decision tree on the folds has a mean of 0.6328798 (cost_complexity of 0.002782559).

Exercise 5

Using `rpart.plot`, fit and visualize your best-performing pruned decision tree with the *training* set.

```r
class_tree_final_fit <- fit(class_tree_final, data = pokemon_train)
```

```r
class_tree_final_fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```



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.

```
rf_spec <- rand_forest(mtry = tune(),
                       trees = tune(),
                       min_n = tune()) %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("classification")

rf_wf <- workflow() %>%
  add_model(rf_spec %>% set_args(mtry = tune(),
                       trees = tune(),
                       min_n = tune())) %>%
  add_recipe(pokemon_recipe)
```

Answer: Referring to the documentation for rand_forest(), hyperparameter mtry means when we create a tree model this is the amount of predictors that will be randomly sampled at each split, hyperparameter trees is the number of trees contained in the ensemble (machine learning model that combines the predictions for two or more models) and finally hyperparameter min_n is the minimaal amount of datapoints in a node the model needs to continue the split.

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?**

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

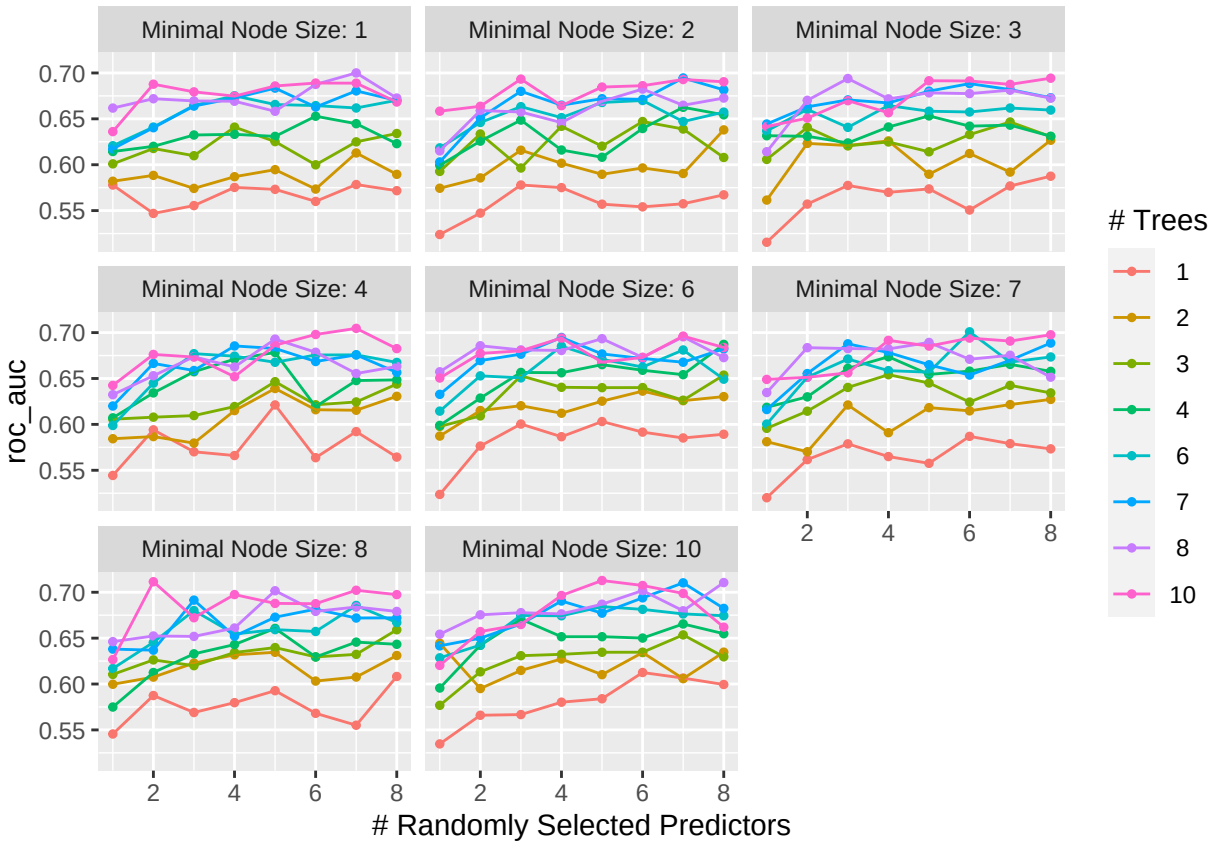
Answer: mtry should not be smaller than 1 or larger than 8 because there are 8 variables at most in the dataset and we should include at least one variable. mtry=8 would represent a pure bagging model not a random forest.

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?

```
rf_tune_res <- tune_grid(
  rf_wf,
  resamples = pokemon_folds,
  grid = rf_grid,
  metrics = metric_set(roc_auc)
)
```

```
autoplot(rf_tune_res)
```



Answer: After tuning the model and printing out an autoplot(), we observe that the model does best with larger amounts of trees, mtry, and min_n. Hyperparameter values of mtry around 7 or 8, trees around 10 and min_n around 6 or 7 seems to yield the best performance. It is a bit hard to see as there are so many lines and dots.

Exercise 7

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

```
# First we can use the functions collect_metrics() and arrange() to look at the results
collect_metrics(rf_tune_res) %>% arrange(-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     5    10    10 roc_auc hand_till  0.713     5 0.0126  Preprocessor1_Model~
##  2     2    10     8 roc_auc hand_till  0.711     5 0.0181  Preprocessor1_Model~
##  3     8     8    10 roc_auc hand_till  0.711     5 0.0169  Preprocessor1_Model~
##  4     7     7    10 roc_auc hand_till  0.710     5 0.0224  Preprocessor1_Model~
##  5     6    10    10 roc_auc hand_till  0.707     5 0.0120  Preprocessor1_Model~
##  6     7    10     4 roc_auc hand_till  0.705     5 0.0129  Preprocessor1_Model~
##  7     7    10     8 roc_auc hand_till  0.702     5 0.0147  Preprocessor1_Model~
##  8     6     8    10 roc_auc hand_till  0.702     5 0.0182  Preprocessor1_Model~
##  9     5     8     8 roc_auc hand_till  0.702     5 0.0149  Preprocessor1_Model~
## 10     6     6     7 roc_auc hand_till  0.701     5 0.00702 Preprocessor1_Model~
## # ... with 502 more rows
```

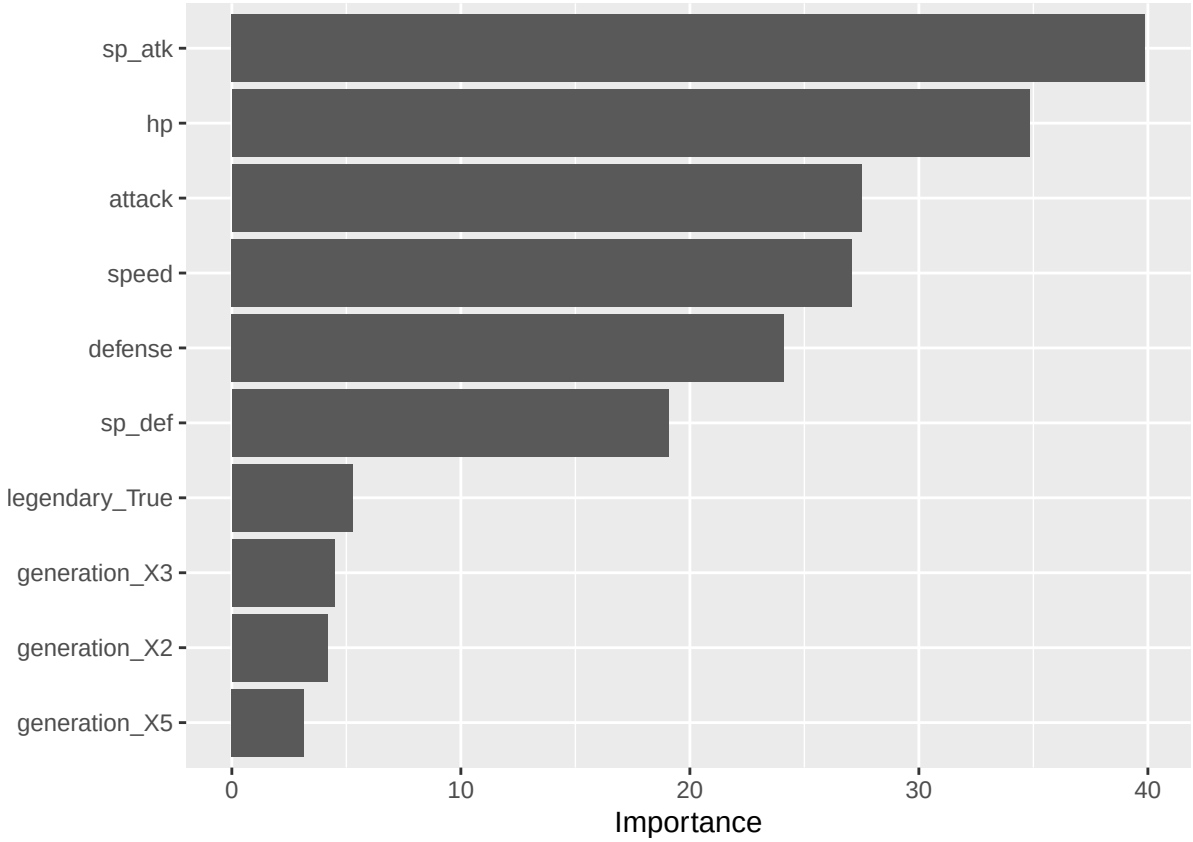
```
# Then we can use select_best function to choose the best complexity
rf_best <- select_best(rf_tune_res, metric = "roc_auc")
class_rf_final <- finalize_workflow(rf_wf, rf_best)
```

The roc_auc of the best-performing random forest model on the folds has a mean of 0.7134003 (mtry=8, trees=10 and min_n =7).

Exercise 8

Create a variable importance plot, using `vip()`, with your best-performing random forest model fit on the *training* set.

```
# use vip() function
class_rf_final  %>%
  fit(data = pokemon_train) %>%
  pull_workflow_fit() %>%
  vip()
```



Which variables were most useful? Which were least useful? Are these results what you expected, or not?

Answer: It seems that sp_atk, speed, attack, hp, sp_def and defense were the most useful variables while the least useful were generation_X2, generation_X3, generation_X5, and generation_X4 were the last useful. These results are expected in theory, the stats that each pokemon has should be useful in predicting the primary type.

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.

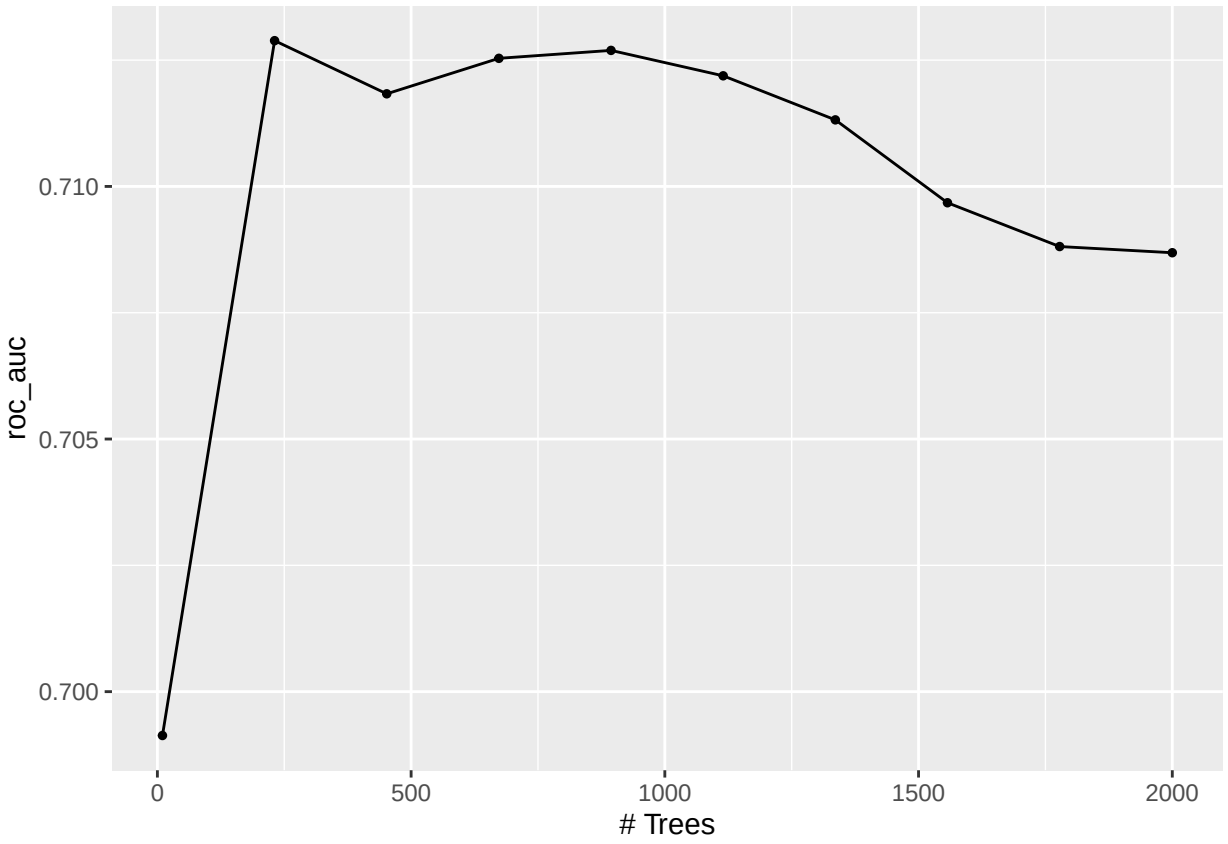
```
boost_spec <- boost_tree(trees = tune()) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

boost_wf <- workflow() %>%
  add_model(boost_spec %>% set_args(trees = tune())) %>%
  add_recipe(pokemon_recipe)
```

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

```
boost_tune_res <- tune_grid(
  boost_wf ,
  resamples = pokemon_folds,
```

```
  grid = boost_grid,
  metrics = metric_set(roc_auc)
)
```

```
autoplot(boost_tune_res)
```



What do you observe?

Answer: We observe that the model foes really well with a smaller amount of trees (around 250). The performance then dips, stabilizes then dips again.

What is the `roc_auc` of your best-performing boosted tree model on the folds? *Hint: Use* `collect_metrics()` *and* `arrange()`.

```
# First we can use the functions collect_metrics() and arrange() to look at the results
collect_metrics(boost_tune_res) %>% arrange(-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  0.713     5 0.0127  Preprocessor1_Model02
##  2   894 roc_auc hand_till  0.713     5 0.0106  Preprocessor1_Model05
##  3   673 roc_auc hand_till  0.713     5 0.0113  Preprocessor1_Model04
##  4  1115 roc_auc hand_till  0.712     5 0.0108  Preprocessor1_Model06
##  5   452 roc_auc hand_till  0.712     5 0.0113  Preprocessor1_Model03
##  6  1336 roc_auc hand_till  0.711     5 0.0105  Preprocessor1_Model07
##  7  1557 roc_auc hand_till  0.710     5 0.0104  Preprocessor1_Model08
```

```
##  8  1778 roc_auc hand_till  0.709     5 0.00999 Preprocessor1_Model09
##  9  2000 roc_auc hand_till  0.709     5 0.00999 Preprocessor1_Model10
## 10    10 roc_auc hand_till  0.699     5 0.0181  Preprocessor1_Model01
```

```
# Then we can use select_best function to choose the best complexity
boost_best <- select_best(boost_tune_res, metric = "roc_auc")
class_tree_final <- finalize_workflow(boost_wf, boost_best)
```

The roc_auc of the best-performing boosted tree model on the folds has a mean of 0.7128855 (trees=231).

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.

```
# Use this to look at the values to create the table
#(collect_metrics(tune_res) %>% arrange(-mean))[1,]
#(collect_metrics(rf_tune_res) %>% arrange(-mean))[1,]
#(collect_metrics(boost_tune_res) %>% arrange(-mean))[1,]

# Create table of three ROC AUC values for each model using as.table() function
as.table(matrix(c(0.6328798,0.7134003,0.7128855), ncol=1, byrow=FALSE,
 dimnames=list(Model = c("Pruned Decision Tree Model", "Random Forest Model","Boosted Tree Model"),
  c("Best ROC_AUC Value"))))
```

```
##
## Model                         Best ROC_AUC Value
##   Pruned Decision Tree Model          0.6328798
##   Random Forest Model                 0.7134003
##   Boosted Tree Model                  0.7128855
```

Answer: Here we can see that the Random Forest Model preformed best on the folds (ROC_AUC Value, mean of 0.7134003 - larger than all the other values). We will be using the Random Forest Model to fit the final model to the testing set.

```
best <- select_best(rf_tune_res, metric = "roc_auc")
# use `finalize_workflow()`, `fit()`, and `augment()` to fit the model to
# the training set and evaluate its performance on the testing set
final_wf <- finalize_workflow(rf_wf, best)
final_fit <- fit(final_wf, data = pokemon_test)
# predictions for each observation
predicted_data <- augment(final_fit, new_data = pokemon_test)
```

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.

Answer: The AUC value of the best-preforming model on the testing set:

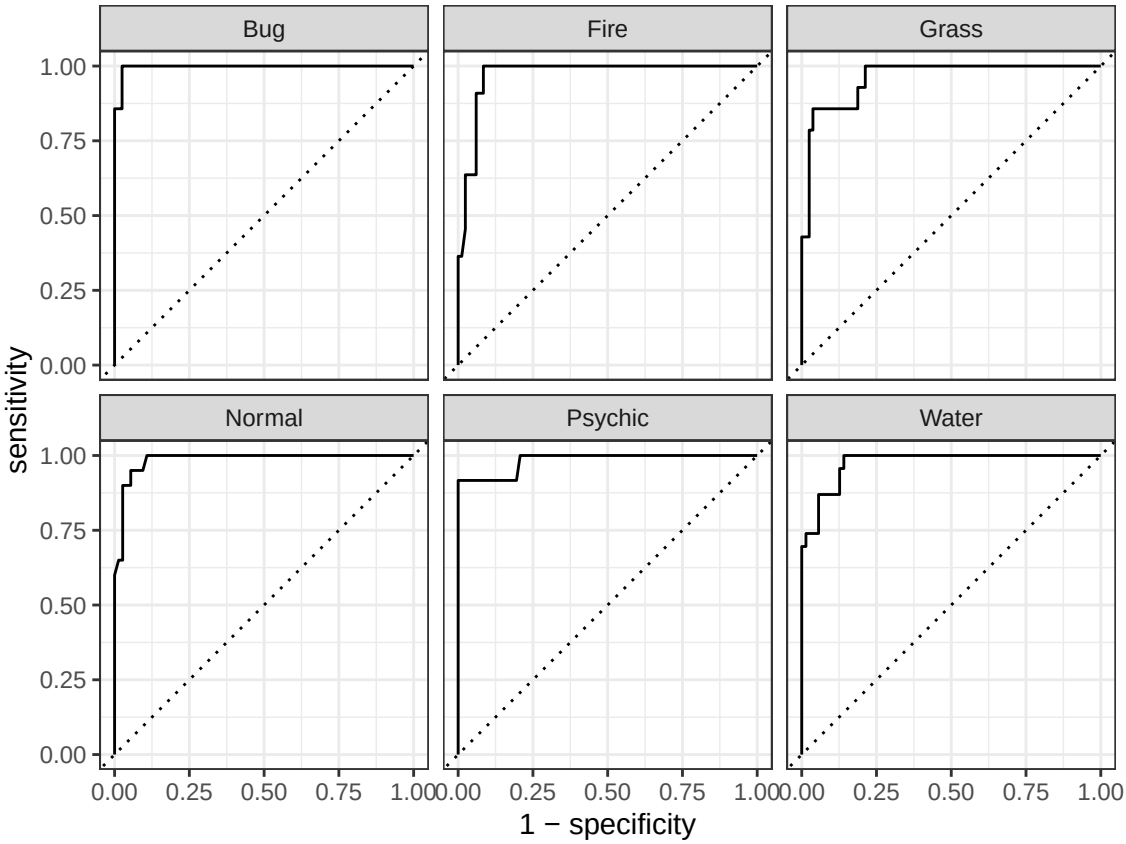
```
# use the functions augment() and roc_auc() to calculate the overall ROC AUC
# on the testing set
predicted_data <- augment(final_fit, new_data = pokemon_test) %>%
```

```
  select(type_1, starts_with(".pred"))

predicted_data%>%roc_auc(type_1,.pred_Bug:.pred_Water)
```

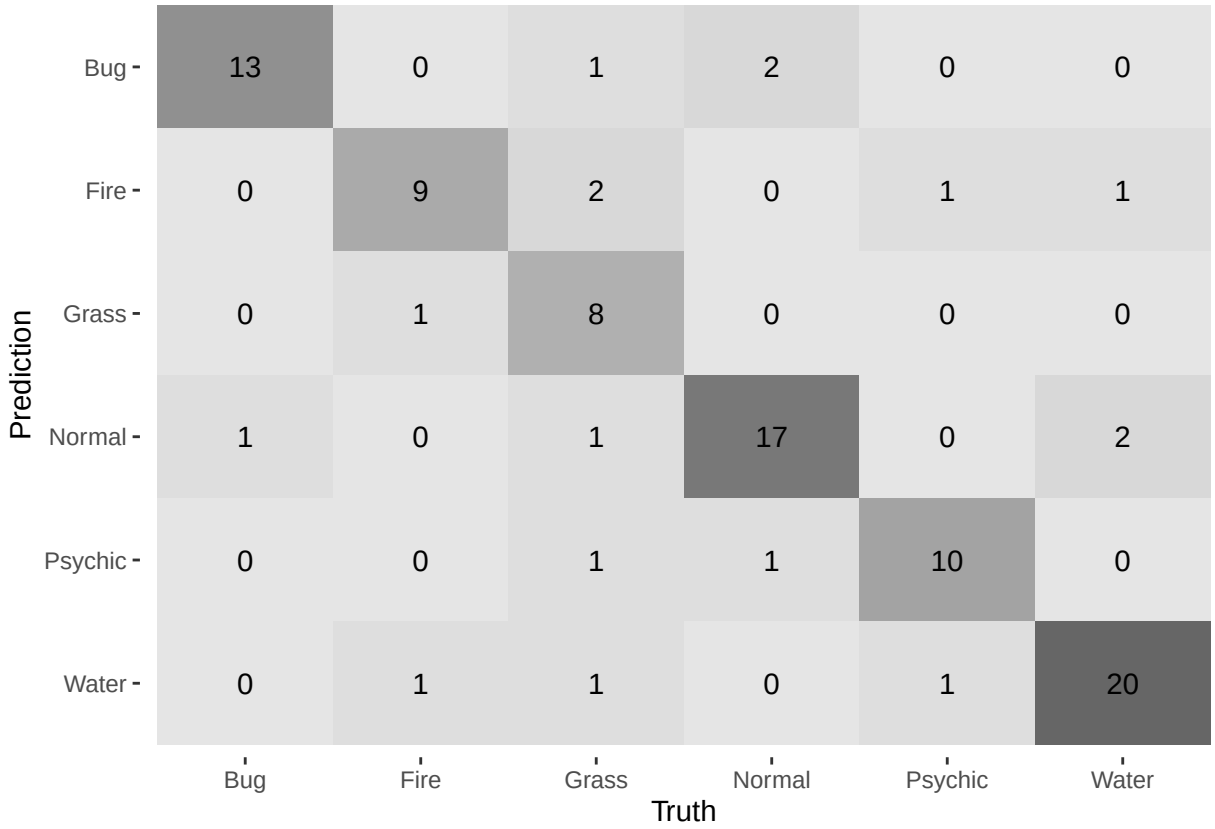
```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.977
```

Answer: Printed ROC curves and created and visualized confusion matrix heat map:

```
# use function roc_curve() function to create different ROC curves,
# one per level of the outcome
predicted_data%>%roc_curve(type_1,.pred_Bug:.pred_Water)%>%
  autoplot()
```




```
# use conf_map() and autoplot() to make a heat map of the confusion matrix
predicted_data%>%
  conf_mat(truth=type_1,estimate=.pred_class)%>%
  autoplot(type="heatmap")
```



Which classes was your model most accurate at predicting? Which was it worst at?

Answer: The model seems to have done really well. The model was the most accurate at predicting Water and Normal. It was the worst at predicting Fire.