PSTAT 231 - Homework 6

Code

Yuer Hao

Load Package

Hide

```
library(tinytex)
library(tidyverse)
library(tidymodels)
library(ISLR)
library(ggplot2)
library(rpart.plot)
library(randomForest)
library(ranger)
library(vip)
library(xgboost)
library(corrplot)
library(magrittr)
library(corrr)
library(discrim)
library(poissonreg)
library(klaR)
library(janitor)
library(glmnet)
library(ggthemes)
library(yardstick)
library(dplyr)
tidymodels_prefer()
set.seed(1126)
```

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 (https://www.kaggle.com/abcsds/pokemon).

The Pokémon (https://www.pokemon.com/us/) 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 (https://bulbapedia.bulbagarden.net/wiki/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.

Fig 1. Houndoom, a Dark/Fire-type canine Pokémon from Generation II.

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.

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.

Hide

```
# load the data & clean names
pokemon <- read.csv("/Users/Yuer_Hao/Desktop/PSTAT 231/homework-6/data/Pokemon.c
sv")
#view(pokemon)
pkm <- pokemon %>% clean_names()

#Filter out the rarer Pokémon types and Convert 'type_1' and 'legendary' to fact
ors
pkm2 <- pkm %>%
  filter(type_1 %in% c("Bug","Fire","Grass","Normal","Water","Psychic"))
pkm2$type_1 <- factor(pkm2$type_1)
pkm2$legendary <- factor(pkm2$legendary)
pkm2$generation <- factor(pkm2$generation)

#Do a initial split of the data
pkm_split <- initial_split(pkm2, prop = 0.80,  strata = "type_1")
pkm_train <- training(pkm_split)
pkm_test <- testing(pkm_split)

#Fold the training set using v-fold cv with 'v=5'
pkm_folds <- vfold_cv(pkm_train, v=5, strata = "type_1")

#Set up the recipe
#1)Dummy-code legendary and generation;
#2)Center and scale all predictors.
pkm_recipe <- recipe(type_1 ~ legendary
                     + generation
                     + sp_atk
                     + attack
                     + speed
                     + defense
                     + hp
                     + sp_def,
                    data = pkm_train) %>%
  step_dummy(c("legendary","generation")) %>%
  step_center(all_predictors()) %>%
  step_scale(all_predictors())
```

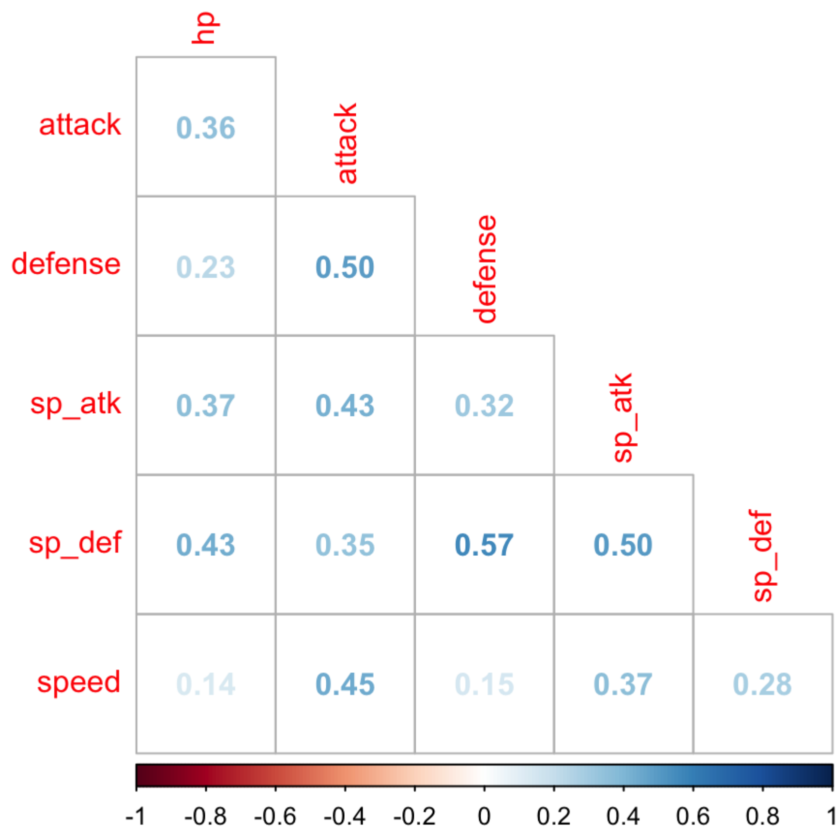
Exercise 2

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?

Hide

```
pkm_train %>%
  select(where(is.numeric)) %>%
  select(-x,-total) %>%
  cor() %>%
  corrplot(type = "lower",method = "number",diag = FALSE)
```



The correlation matrix shows a correlation between the sp_def and defense. Attack and defense are correlated. Sp_atk and Sp_def also have a correlation in between. Attack and sp_atk are correlated with speed. To me, they all make sense.

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`.

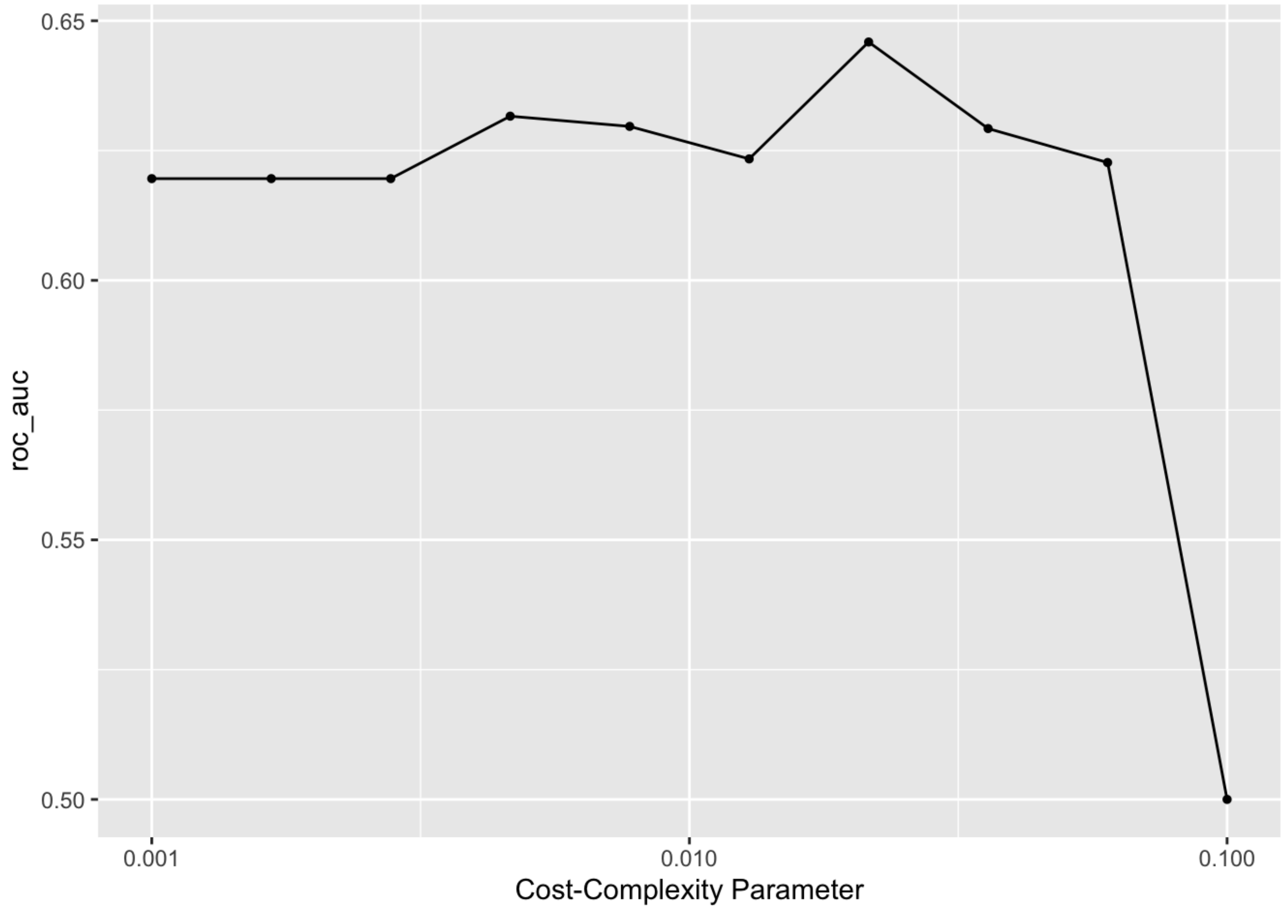
Print an `autoplot()` of the results. What do you observe? Does a single decision tree perform better with a smaller or larger complexity penalty?

Hide

```
#set up a decision tree model and workflow
tree_spec <- decision_tree(cost_complexity = tune()) %>%
  set_engine("rpart") %>%
  set_mode("classification")

tree_wkflow <- workflow() %>%
  add_recipe(pkm_recipe) %>%
  add_model(tree_spec)

#Tune the `cost_complexity` hyperparameter
param_grid <- grid_regular(cost_complexity(range = c(-3, -1)), levels = 10)
tune_res <- tune_grid(tree_wkflow,
                      resamples = pkm_folds,
                      grid = param_grid,
                      metrics = metric_set(roc_auc))
autoplot(tune_res)
```



The performance of the decision tree is optimized with lower complexity penalties. It peaks at 0.05 and drastically declining after that.

Exercise 4

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

Hide

```
collect_metrics(tune_res) %>%
  arrange(desc(mean))
```

```
## # A tibble: 10 × 7
##    cost_complexity .metric .estimator  mean     n std_err .config
##              <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1         0.0215  roc_auc hand_till  0.646     5 0.0117  Preprocessor1_Model
07
##  2         0.00464 roc_auc hand_till  0.632     5 0.00940 Preprocessor1_Model
04
##  3         0.00774 roc_auc hand_till  0.630     5 0.00847 Preprocessor1_Model
05
##  4         0.0359  roc_auc hand_till  0.629     5 0.00910 Preprocessor1_Model
08
##  5         0.0129  roc_auc hand_till  0.623     5 0.00445 Preprocessor1_Model
06
##  6         0.0599  roc_auc hand_till  0.623     5 0.0114  Preprocessor1_Model
09
##  7         0.001   roc_auc hand_till  0.620     5 0.0121  Preprocessor1_Model
01
##  8         0.00167 roc_auc hand_till  0.620     5 0.0121  Preprocessor1_Model
02
##  9         0.00278 roc_auc hand_till  0.620     5 0.0121  Preprocessor1_Model
03
## 10         0.1     roc_auc hand_till  0.5       5 0       Preprocessor1_Model
10
```

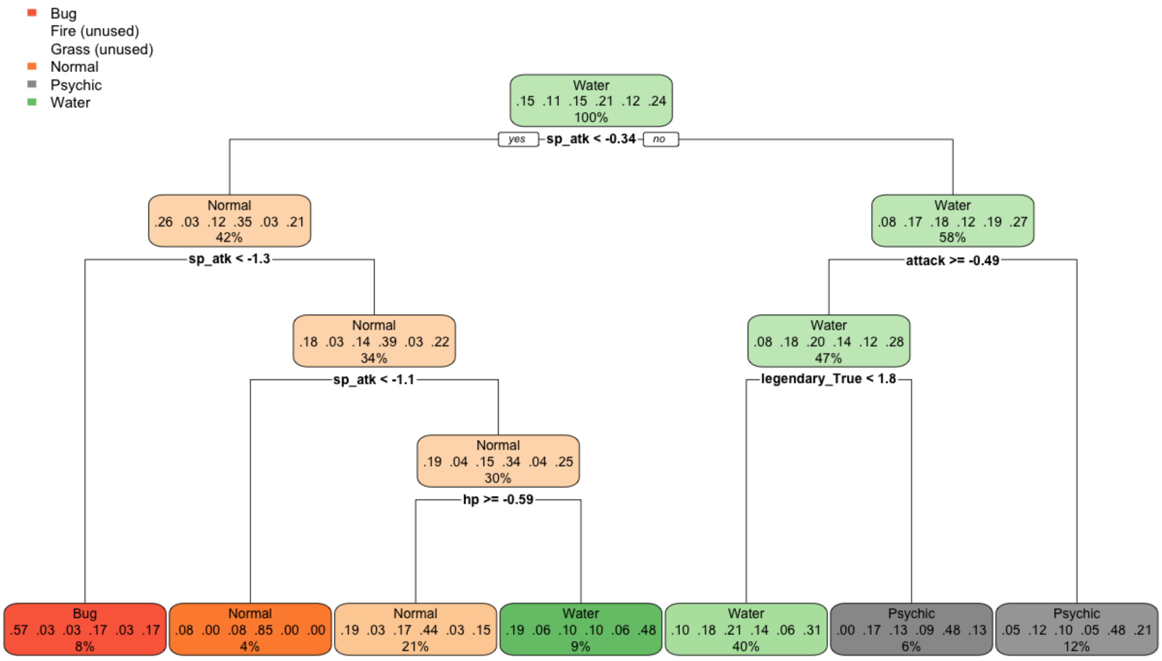
The roc_auc of the best_performing pruned decision tree was 0.6459126.

Exercise 5

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

Hide

```
best_complexity <- select_best(tune_res)
tree_final <- finalize_workflow(tree_wkflow, best_complexity)
tree_fit <- fit(tree_final,pkm_train)
tree_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.

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?**

Hide

```
#mtry: the number of predictors will be randomly chosen while building tree mode
ls.
#trees: the number of trees contained in the ensemble.
#min_n: the minimum amount of data points in a node that are needed for the node
to be split further.

#set up a random forest model and workflow.
rand_forest_spec <- rand_forest() %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("classification")
rand_forest_wf <- workflow() %>%
  add_model(rand_forest_spec %>% set_args(mtry = tune(), trees = tune(), min_n =
tune())) %>%
  add_recipe(pkm_recipe)

#Create a regular grid with 8 levels each
rand_forest_grid <- grid_regular(
  mtry(range = c(1, 8)),
  trees(range = c(10,1000)),
  min_n(range = c(1, 10)),
  levels = 8)
```

Since mtry reflects the number of predictors, and we only have 8, it cannot be greater than 8. If mtry was less than 1, there would be no criterion on which to split. The model employs all 8 predictors if mtry = 8.

Exercise 6

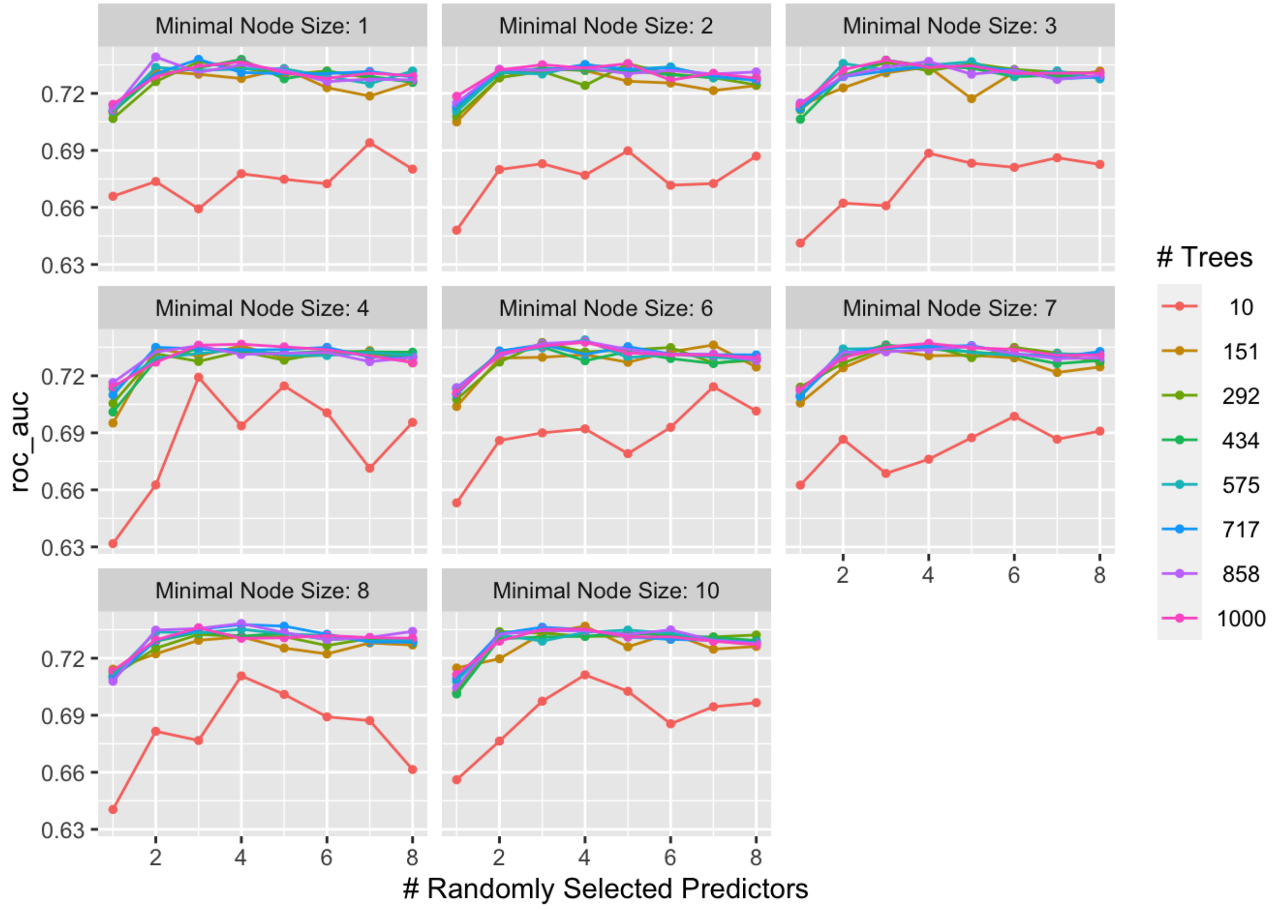
Specify `roc_auc` as a metric. Tune the model and print an `autoplot()` of the results. What do you observe? What values of the hyperparameters seem to yield the best performance?

Hide

```
rand_forest_tune <- tune_grid(
  rand_forest_wf,
  resamples = pkm_folds,
  grid = rand_forest_grid,
  metrics = metric_set(roc_auc)
)
autoplot(rand_forest_tune)
```

Hide

```
autoplot(rand_forest_tune)
```



The accuracy seems to be unaffected by minimal node size. In general, the more trees there are the better, consistent accuracy, especially when the number of trees is more than 10. Additionally, accuracy grows significantly as predictor numbers rise.

Exercise 7

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

Hide

```
collect_metrics(rand_forest_tune) %>% arrange(desc(mean))
```

```
## # A tibble: 512 × 9
##     mtry trees min_n .metric .estimator  mean     n std_err .config
##    <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1     2   858     1 roc_auc hand_till  0.739     5  0.0203 Preprocessor1_Mod
el…
##  2     4   575     6 roc_auc hand_till  0.739     5  0.0202 Preprocessor1_Mod
el…
##  3     4   858     8 roc_auc hand_till  0.738     5  0.0204 Preprocessor1_Mod
el…
##  4     4   858     6 roc_auc hand_till  0.738     5  0.0198 Preprocessor1_Mod
el…
##  5     3   717     1 roc_auc hand_till  0.738     5  0.0208 Preprocessor1_Mod
el…
##  6     4   434     1 roc_auc hand_till  0.738     5  0.0196 Preprocessor1_Mod
el…
##  7     4   717     8 roc_auc hand_till  0.738     5  0.0197 Preprocessor1_Mod
el…
##  8     4  1000     6 roc_auc hand_till  0.738     5  0.0182 Preprocessor1_Mod
el…
##  9     3   292     6 roc_auc hand_till  0.738     5  0.0193 Preprocessor1_Mod
el…
## 10     3   434     3 roc_auc hand_till  0.738     5  0.0202 Preprocessor1_Mod
el…
## # … with 502 more rows
```

The best model's roc_auc is 0.7391476, with mtry=2, trees=858, and min_n=1.

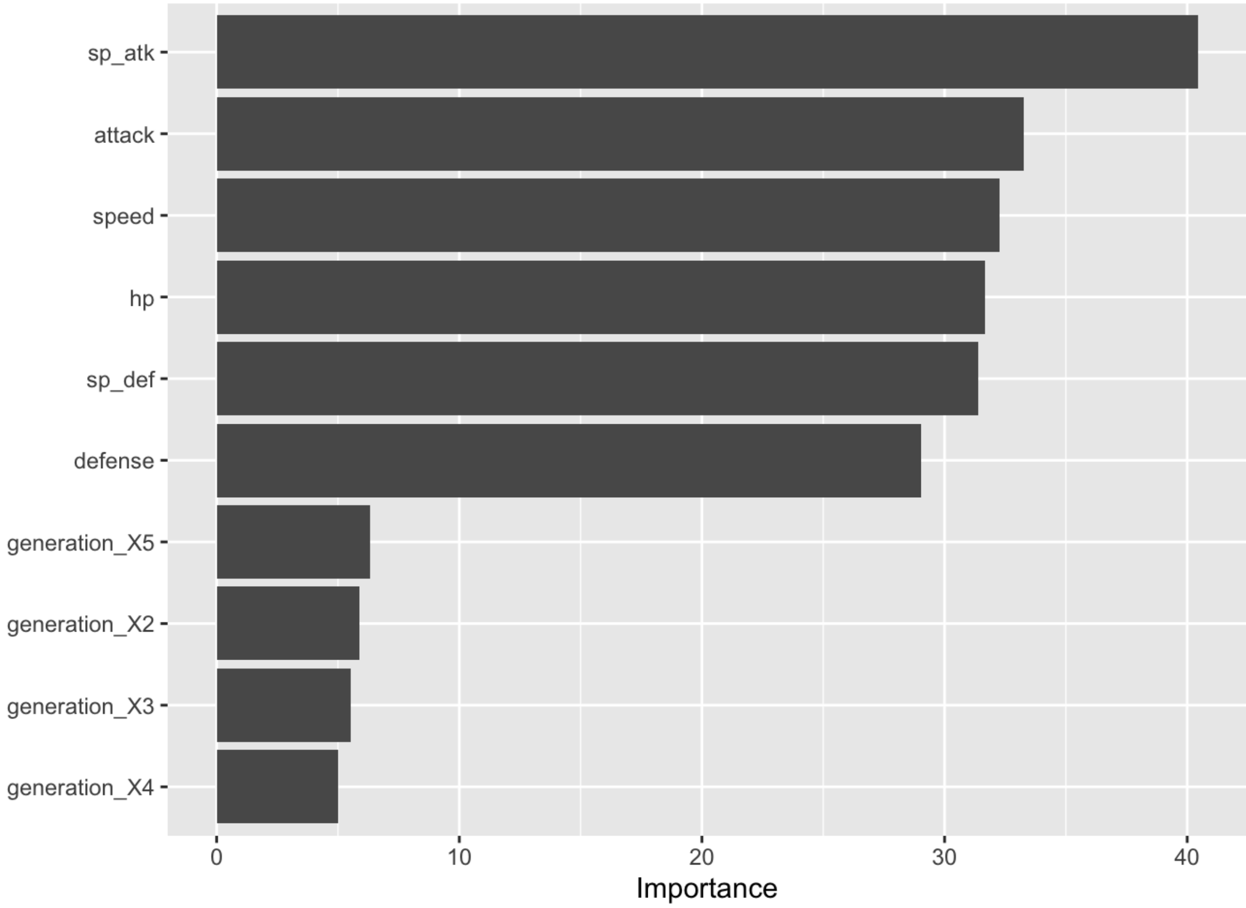
Exercise 8

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?

Hide

```
rand_forest_final <- finalize_workflow(rand_forest_wf,select_best(rand_forest_tu
ne,"roc_auc"))
rand_forest_fit <- fit(rand_forest_final,pkm_train)
rand_forest_fit %>%
  extract_fit_engine() %>%
  vip()
```



The two factors that were most helpful for determining the main Pokemon type were the predictors sp_atk and attack. Besides, the hp, sp_def, speed, and defense are also quite useful. Legendary status, the generation the Pokemon originated from, and defense were the three factors that performed the poorest for the identical forecast. It is hardly surprising that generation and legendary status were the least significant factors. There are many different kinds that the many legendary pokemon may take, so I wouldn't anticipate seeing significantly more of any one type in any one generation.

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.

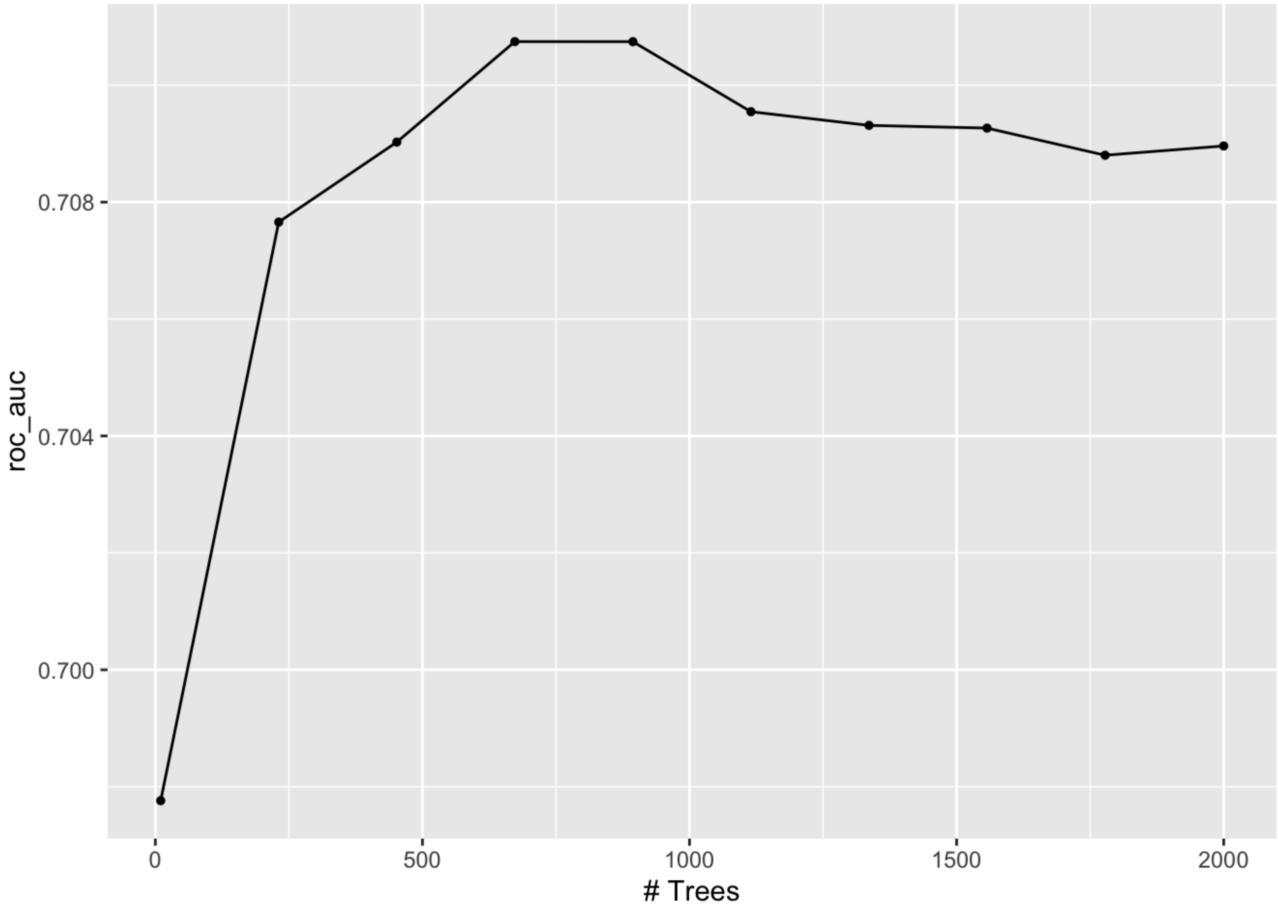
What do you observe?

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

Hide

```
btr_spec <- boost_tree(trees = tune()) %>%
  set_engine("xgboost") %>%
  set_mode("classification")
btr_grid <- grid_regular(trees(c(10,2000)), levels = 10)
btr_wkflow <- workflow() %>%
  add_model(btr_spec) %>%
  add_recipe(pkm_recipe)

btr_tune_res <- tune_grid(
  btr_wkflow,
  resamples = pkm_folds,
  grid = btr_grid,
  metrics = metric_set(roc_auc)
)
autoplot(btr_tune_res)
```



The roc_auc increases when number of trees is increasing, and reaches the peak at around 690 trees and continuous until around 920. Then the roc_auc keeps decreasing as the trees increasing.

Hide

```
collect_metrics(btr_tune_res) %>% arrange(desc(mean))
```

```
## # A tibble: 10 × 7
##    trees .metric .estimator  mean     n std_err .config
##    <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1   673 roc_auc hand_till  0.711     5  0.0207 Preprocessor1_Model04
##  2   894 roc_auc hand_till  0.711     5  0.0215 Preprocessor1_Model05
##  3  1115 roc_auc hand_till  0.710     5  0.0214 Preprocessor1_Model06
##  4  1336 roc_auc hand_till  0.709     5  0.0216 Preprocessor1_Model07
##  5  1557 roc_auc hand_till  0.709     5  0.0219 Preprocessor1_Model08
##  6   452 roc_auc hand_till  0.709     5  0.0213 Preprocessor1_Model03
##  7  2000 roc_auc hand_till  0.709     5  0.0224 Preprocessor1_Model10
##  8  1778 roc_auc hand_till  0.709     5  0.0224 Preprocessor1_Model09
##  9   231 roc_auc hand_till  0.708     5  0.0209 Preprocessor1_Model02
## 10    10 roc_auc hand_till  0.698     5  0.0165 Preprocessor1_Model01
```

The best_performing boosted tree model's roc_auc is 0.7107454 with 673 trees.

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?

Hide

```
#Display a table of the three ROC AUC values
btr_final <- finalize_workflow(btr_wkflow,select_best(btr_tune_res,"roc_auc"))
btr_fit <- fit(btr_final,pkm_train)
final_class_model <- augment(tree_fit, new_data = pkm_train)
final_random_forest <- augment(rand_forest_fit, new_data = pkm_train)
final_boosted_tree <- augment(btr_fit, new_data = pkm_train)
bind_rows(
  roc_auc(final_class_model, truth = type_1, .pred_Bug,
        .pred_Fire, .pred_Grass, .pred_Normal, .pred_Water, .pred_Psychic),
  roc_auc(final_random_forest, truth = type_1, .pred_Bug,
        .pred_Fire, .pred_Grass, .pred_Normal, .pred_Water, .pred_Psychic),
  roc_auc(final_boosted_tree, truth = type_1, .pred_Bug,
        .pred_Fire, .pred_Grass, .pred_Normal, .pred_Water, .pred_Psychic))
```

```
## # A tibble: 3 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.631
## 2 roc_auc hand_till      0.797
## 3 roc_auc hand_till      0.800
```

Based on the output, we can tell that the best model is boosted tree model with 0.8002366 roc_auc.

Hide

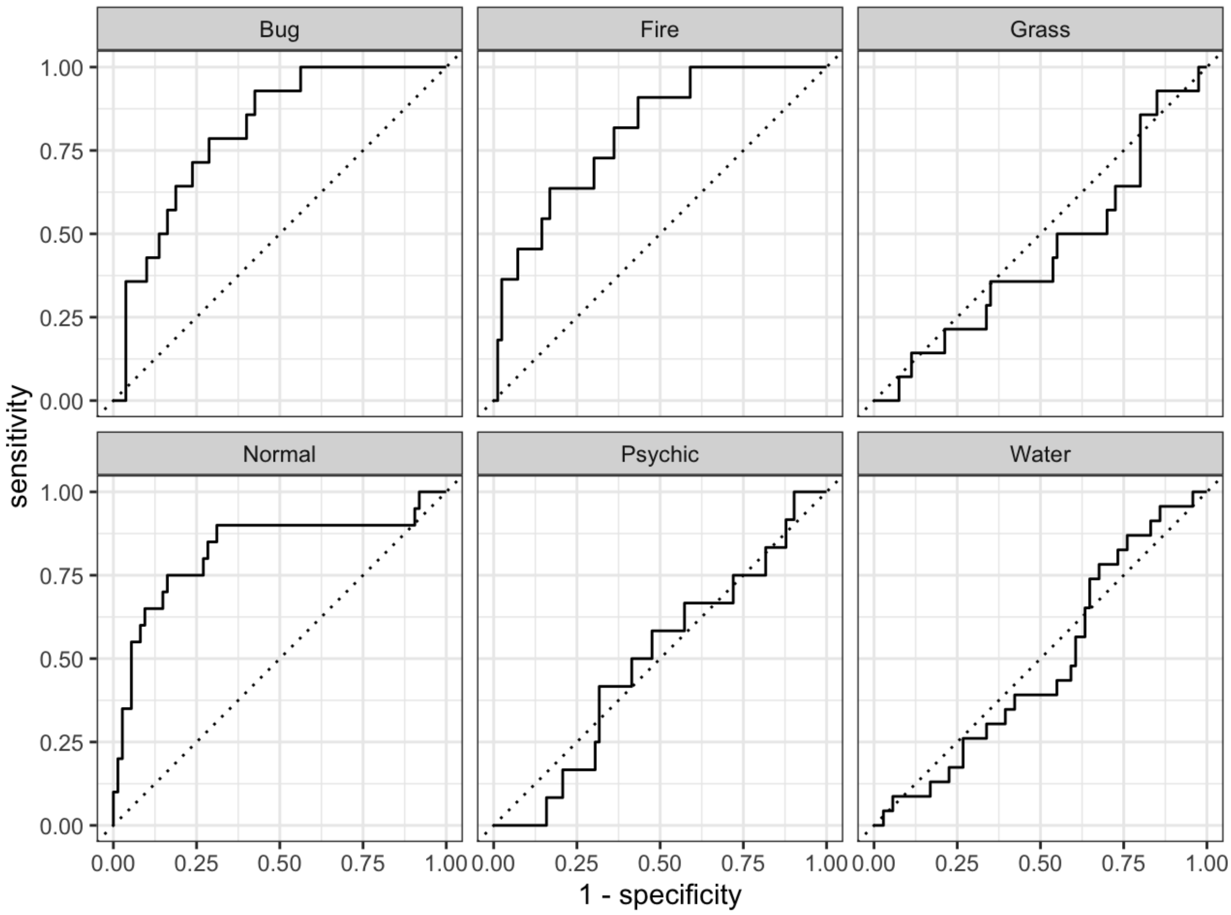
```
final_boosted_tree_test <- augment(btr_fit, new_data = pkm_test)
roc_auc(final_boosted_tree_test, truth = type_1,
        .pred_Bug,
        .pred_Fire,
        .pred_Grass,
        .pred_Normal,
        .pred_Water,
        .pred_Psychic)
```

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.634
```

The roc_auc on the test data set is 0.6343797.

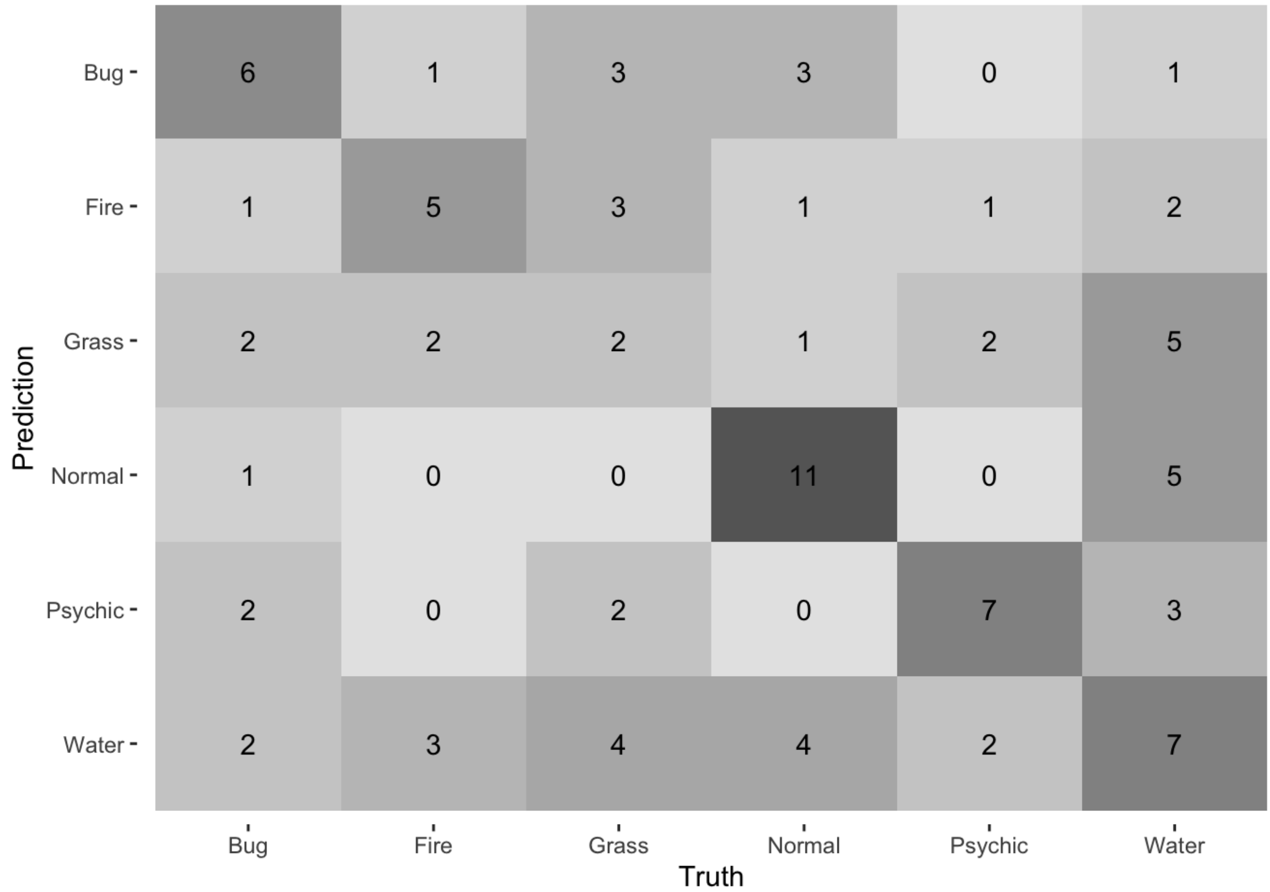
Hide

```
autoplot(roc_curve(final_boosted_tree_test,
                   truth = type_1,
                   .pred_Bug,
                   .pred_Fire,
                   .pred_Grass,
                   .pred_Normal,
                   .pred_Water,
                   .pred_Psychic))
```



Hide

```
conf_mat(final_boosted_tree_test, truth = type_1, estimate = .pred_class) %>%
  autoplot(type = "heatmap")
```



The model is most accurate at predicting Normal, Fire, and Bug; worst at predicting Water, Grass, and Psychic.

For 231 Students

Exercise 11

Using the `abalone.txt` data from previous assignments, fit and tune a random forest model to predict `age`. Use stratified cross-validation and select ranges for `mtry`, `min_n`, and `trees`. Present your results. What was the model's RMSE on your testing set?

Hide

```
# #Load Data
# ab <- read.csv("/Users/Yuer_Hao/Desktop/PSTAT 231/homework-6/data/abalone.cs
v")
# ab["age"] <- ab["rings"]+1.5
# #Data Split
# ab_split <- initial_split(ab,prop=0.80,strata = age)
# ab_train <- training(ab_split)
# ab_test <- testing(ab_split)
#
# ab_folds <- vfold_cv(ab_train, v = 5, strata = age)
# ab_wo_rings <- ab_train %>% select(-rings)
# ab_recipe <- recipe(age ~ ., data = ab_wo_rings) %>%
#   step_dummy(all_nominal_predictors()) %>%
#   step_interact(terms= ~ starts_with("type"):shucked_weight+
#                 longest_shell:diameter+
#                 shucked_weight:shell_weight) %>%
#   step_center(all_predictors()) %>%
#   step_scale(all_predictors())
```

Hide

```
# abalone_rf <- rand_forest(mtry = tune(), trees = tune(), min_n = tune()) %>%
#   set_engine("ranger", importance = "impurity") %>%
#   set_mode("regression")
# abalone_wkflow <- workflow() %>%
#   add_recipe(ab_recipe) %>%
#   add_model(abalone_rf)
# abalone_grid <- grid_regular(
#   mtry(range = c(1,8)),
#   trees(range = c(10,1000)),
#   min_n(range = c(1,10)),
#   levels = 8
#)
```

Hide

```
# abalone_tune <- tune_grid(
#   abalone_wkflow,
#   resamples = ab_folds,
#   grid = abalone_grid,
#   metrics = metric_set(rmse)
# )
```

The model took forever to run.

Hide

```
#autoplot(abalone_tune)
```

Hide

```
# abalone_final <- finalize_workflow(abalone_wkflow,select_best(abalone_tune))
# abalone_fit <- fit(abalone_final, ab_train)
```

Hide

```
# augment(abalone_fit, new_data = ab_test) %>%
#   rmse(truth = age, estimate = .pred)
```

We should see the model's RMSE on the testing set by getting the abalone_tune value and the last two line of augment()and rmse() functions.