Homework 6

Shiyu Zhang

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.

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.

```
library(tidyverse)
library(tidymodels)
library(corrplot)
library(rpart.plot)
library(vip)
library(janitor)
library(randomForest)
library(xgboost)
library(ranger)
tidymodels_prefer()
```

```
setwd("C:/Users/DELL/Desktop/1126hw6_131")
pokemon <- read.csv("Pokemon.csv")
pokemon_cleannames <- clean_names(pokemon)
colnames(pokemon)
```

```
##  [1] "X."         "Name"       "Type.1"     "Type.2"     "Total"
##  [6] "HP"         "Attack"     "Defense"    "Sp..Atk"    "Sp..Def"
## [11] "Speed"      "Generation" "Legendary"
```

```
colnames(pokemon_cleannames)
```

```
##  [1] "x"          "name"       "type_1"     "type_2"     "total"
##  [6] "hp"         "attack"     "defense"    "sp_atk"     "sp_def"
## [11] "speed"      "generation" "legendary"
```

```
pokemon_filter <- pokemon_cleannames %>%
  filter(type_1 == c("Bug", "Fire", "Grass", "Normal", "Water", "Psychic"))
pokemon1<-pokemon_filter%>%
  mutate(type_1=factor(type_1),
         legendary=factor(legendary))

set.seed(1234)
split_pokemon <- initial_split(pokemon1, prop = 0.70,strata = type_1)
train <- training(split_pokemon)
test <- testing(split_pokemon)

train_folds <- vfold_cv(train, v = 5, strata = type_1)

recipe <- train %>%
  recipe(type_1 ~ legendary + generation + sp_atk + attack+ speed + defense + hp + sp_def) %>%
  step_dummy(legendary)%>%
  step_dummy(generation)%>%
  step_center(all_predictors())%>%
  step_scale(all_predictors())

recipe
```

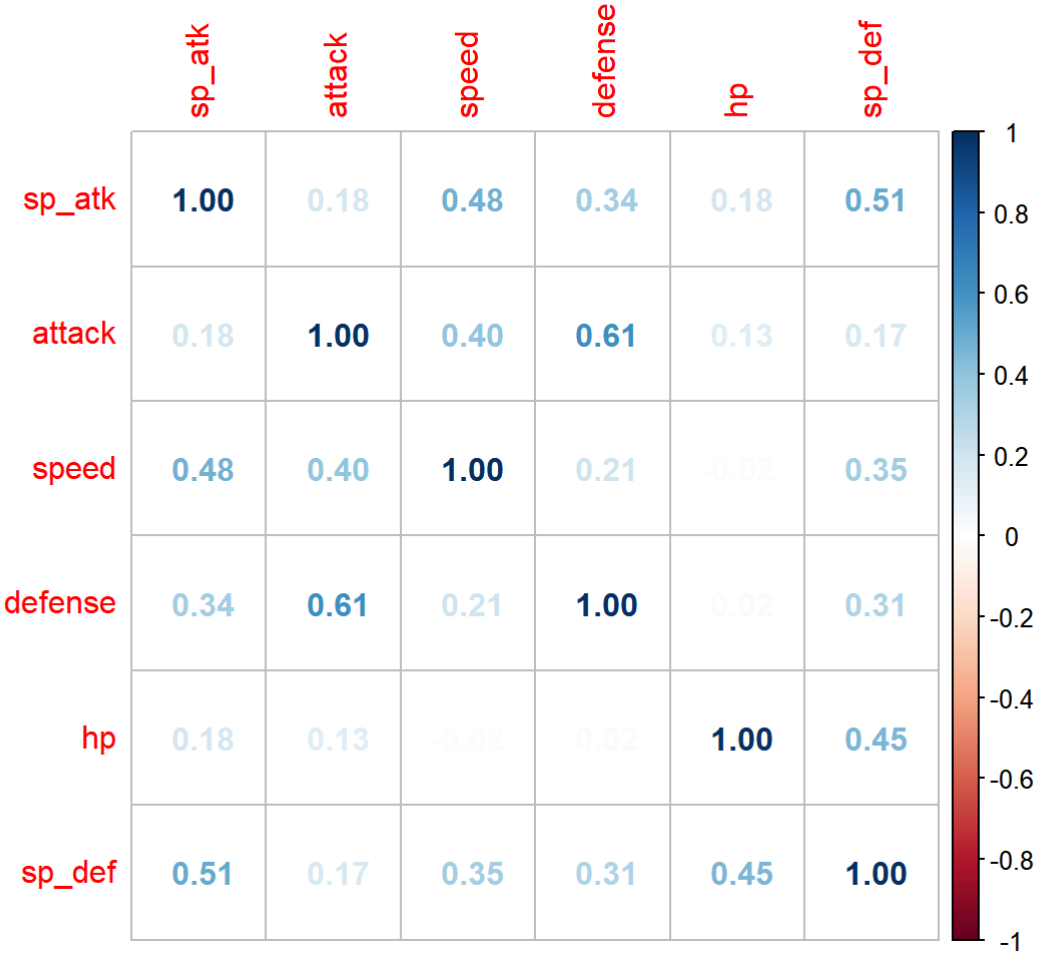
```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          8
##
## Operations:
##
## Dummy variables from legendary
## Dummy variables from generation
## Centering for all_predictors()
## Scaling for all_predictors()
```

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).*

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

```
corr<- train %>%
  select(c(sp_atk, attack, speed, defense, hp, sp_def)) %>%
  cor()

corrplot(corr, method = "number")
```



Answer: We only select numerical independent variables and draw the correlation coefficient matrix. We find that there is a certain degree of positive correlation between variables, and the correlation coefficient of attack and defense is the highest, reaching 0.61.

Exercise 3

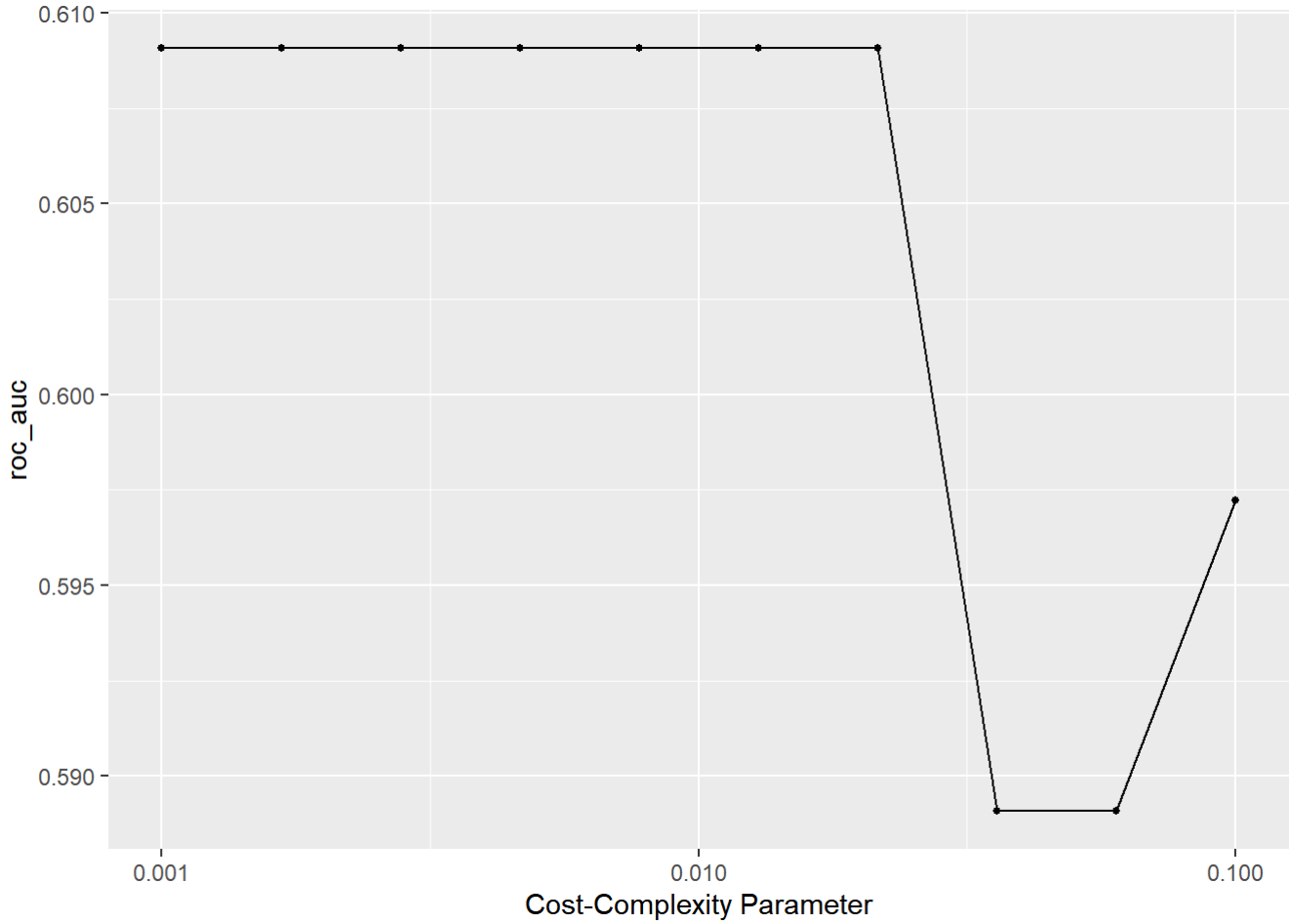
First, set up a decision tree model and workflow. Tune the `cost_complexity` hyperparameter. Use the same levels we used in Lab 7 – that is, `range = c(-3, -1)`. Specify that the metric we want to optimize is `roc_auc`.

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

```
model_tree<- decision_tree() %>%
  set_engine("rpart") %>%
  set_mode("classification") %>%
  set_args(cost_complexity = tune())

workflow_tree <- workflow() %>%
  add_model(model_tree) %>%
  add_recipe(recipe)

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

```
tree_tune_res <- tune_grid(
  workflow_tree,
  resamples = train_folds,
  grid = tree_grid,
  metrics = metric_set(roc_auc)
  )
```

```
#save(tree_tune_res,file = "C:/Users/DELL/Desktop/1126hw6_131/tree_tune_res.rda")
load("C:/Users/DELL/Desktop/1126hw6_131/tree_tune_res.rda")
autoplot(tree_tune_res)
```



Answer: We find that as the parameter cost_complexity increases, the AUC value of the model instead decreases. That means a single decision tree performs 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()`.

```
collect_metrics(tree_tune_res) %>%
  arrange(-mean)
```

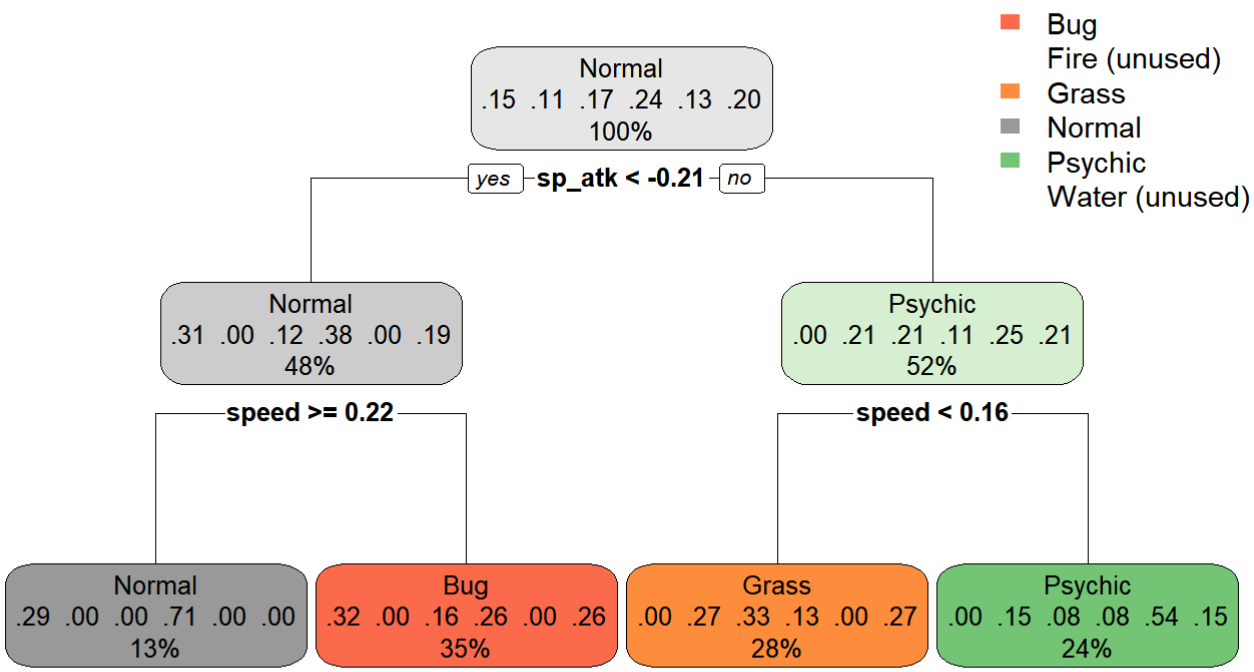
```
## # A tibble: 10 x 7
##    cost_complexity .metric .estimator  mean     n std_err .config
##              <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1         0.001   roc_auc hand_till  0.609     5  0.0503 Preprocessor1_Model01
##  2         0.00167 roc_auc hand_till  0.609     5  0.0503 Preprocessor1_Model02
##  3         0.00278 roc_auc hand_till  0.609     5  0.0503 Preprocessor1_Model03
##  4         0.00464 roc_auc hand_till  0.609     5  0.0503 Preprocessor1_Model04
##  5         0.00774 roc_auc hand_till  0.609     5  0.0503 Preprocessor1_Model05
##  6         0.0129  roc_auc hand_till  0.609     5  0.0503 Preprocessor1_Model06
##  7         0.0215  roc_auc hand_till  0.609     5  0.0503 Preprocessor1_Model07
##  8         0.1     roc_auc hand_till  0.597     5  0.0288 Preprocessor1_Model10
##  9         0.0359  roc_auc hand_till  0.589     5  0.0368 Preprocessor1_Model08
## 10         0.0599  roc_auc hand_till  0.589     5  0.0368 Preprocessor1_Model09
```

Answer: The roc_auc of best-performing pruned decision tree on the folds is 0.6090741.

Exercise 5

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

```
best_param <- select_best(tree_tune_res, metric = "roc_auc")

tree_final <- finalize_workflow(workflow_tree, best_param)

tree_final_fit <- fit(tree_final, data = train)

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.

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?**

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

workflow_rf <- workflow() %>%
  add_model(model_rf ) %>%
  add_recipe(recipe)

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

Answer:
mtry: represents how many independent variables are randomly selected from all independent variables for each decision tree.
trees: the number of decision trees in the random forest model.
min_n: The condition that restricts the continued division of subtree. If the number of samples of a node is less than this value, the division will not continue.

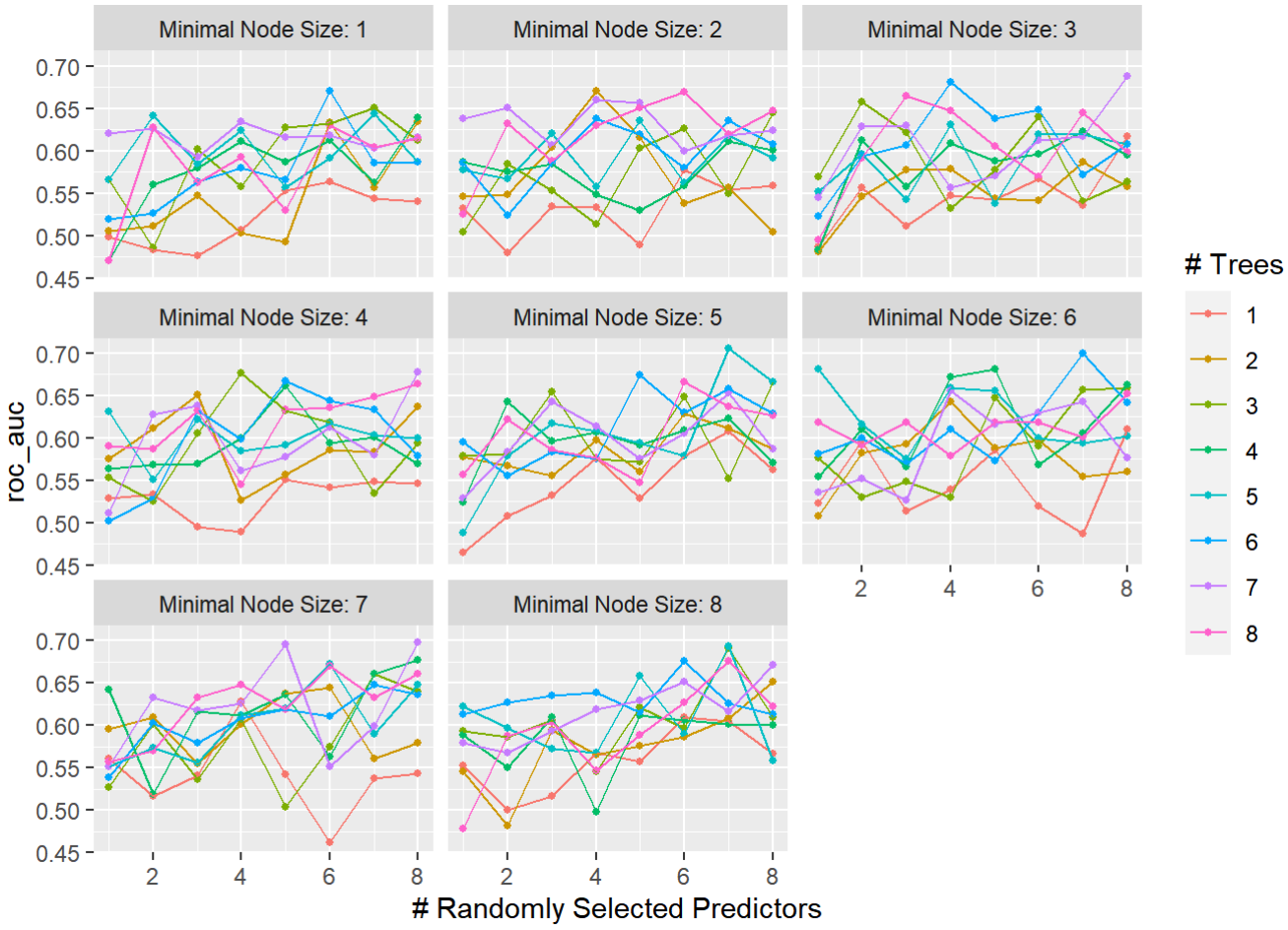
There are only 8 predictors in the model,so mtry should not be smaller than 1 or larger than 8. When mtry is 8 means we use all the predictors for each decision tree.

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(
  workflow_rf,
  resamples = train_folds,
  grid = rf_grid,
  metrics = metric_set(roc_auc)
)
```

```
#save(rf_tune_res,file = "C:/Users/DELL/Desktop/1126hw6_131/rf_tune_res.rda")
load("C:/Users/DELL/Desktop/1126hw6_131/rf_tune_res.rda")
autoplot(rf_tune_res)
```



Answer: We found that the model is not the more complex, the better, but overall, when the minimal node size is equal to 8, the individual submodels seem to be better than the other groups. When mtys = 5, trees = 8, min_n = 6, model seems to yield the best performance.

Exercise 7

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

```
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     7     5     5 roc_auc hand_till  0.705     5  0.0351 Preprocessor1_Model~
##  2     7     6     6 roc_auc hand_till  0.700     5  0.0114 Preprocessor1_Model~
##  3     8     7     7 roc_auc hand_till  0.698     5  0.0483 Preprocessor1_Model~
##  4     5     7     7 roc_auc hand_till  0.696     5  0.0392 Preprocessor1_Model~
##  5     7     5     8 roc_auc hand_till  0.693     5  0.0507 Preprocessor1_Model~
##  6     7     3     8 roc_auc hand_till  0.691     5  0.0521 Preprocessor1_Model~
##  7     8     7     3 roc_auc hand_till  0.688     5  0.0357 Preprocessor1_Model~
##  8     5     4     6 roc_auc hand_till  0.681     5  0.0198 Preprocessor1_Model~
##  9     1     5     6 roc_auc hand_till  0.681     5  0.0361 Preprocessor1_Model~
## 10     4     6     3 roc_auc hand_till  0.681     5  0.0385 Preprocessor1_Model~
## # ... with 502 more rows
```

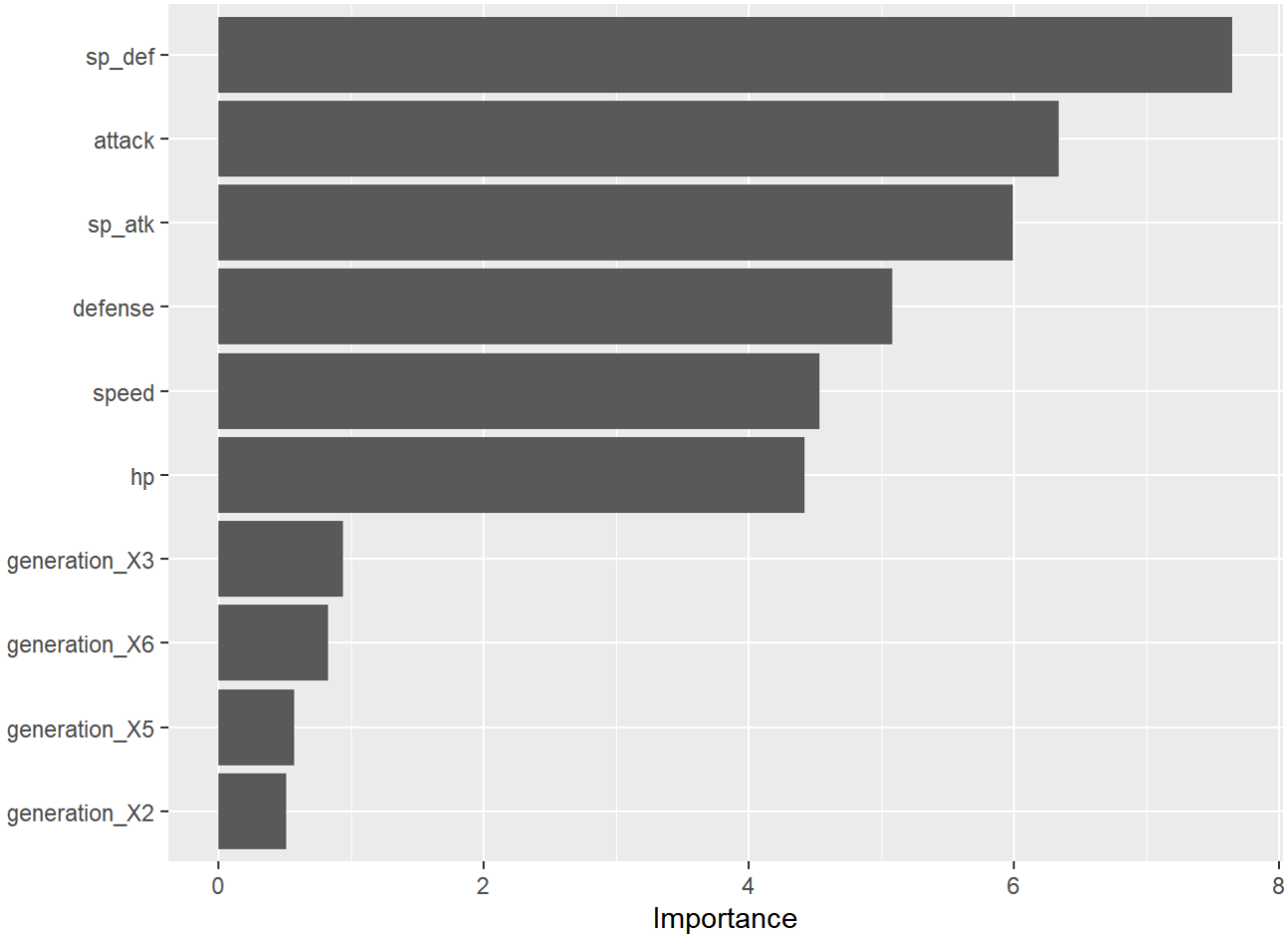
Answer: The roc_auc of my best-performing random forest model on the folds is 0.7185185.

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?

```
set.seed(1234)
best_rf <- select_best(rf_tune_res, metric = "roc_auc")

rf_final <- finalize_workflow(workflow_rf, best_rf)

rf_final_fit <- fit(rf_final, data = train)

rf_final_fit %>%
  pull_workflow_fit() %>%
  vip()
```



Answer: From the above figure, we find that sp_atk is most useful and legendary is least useful. These results are we expected.

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.

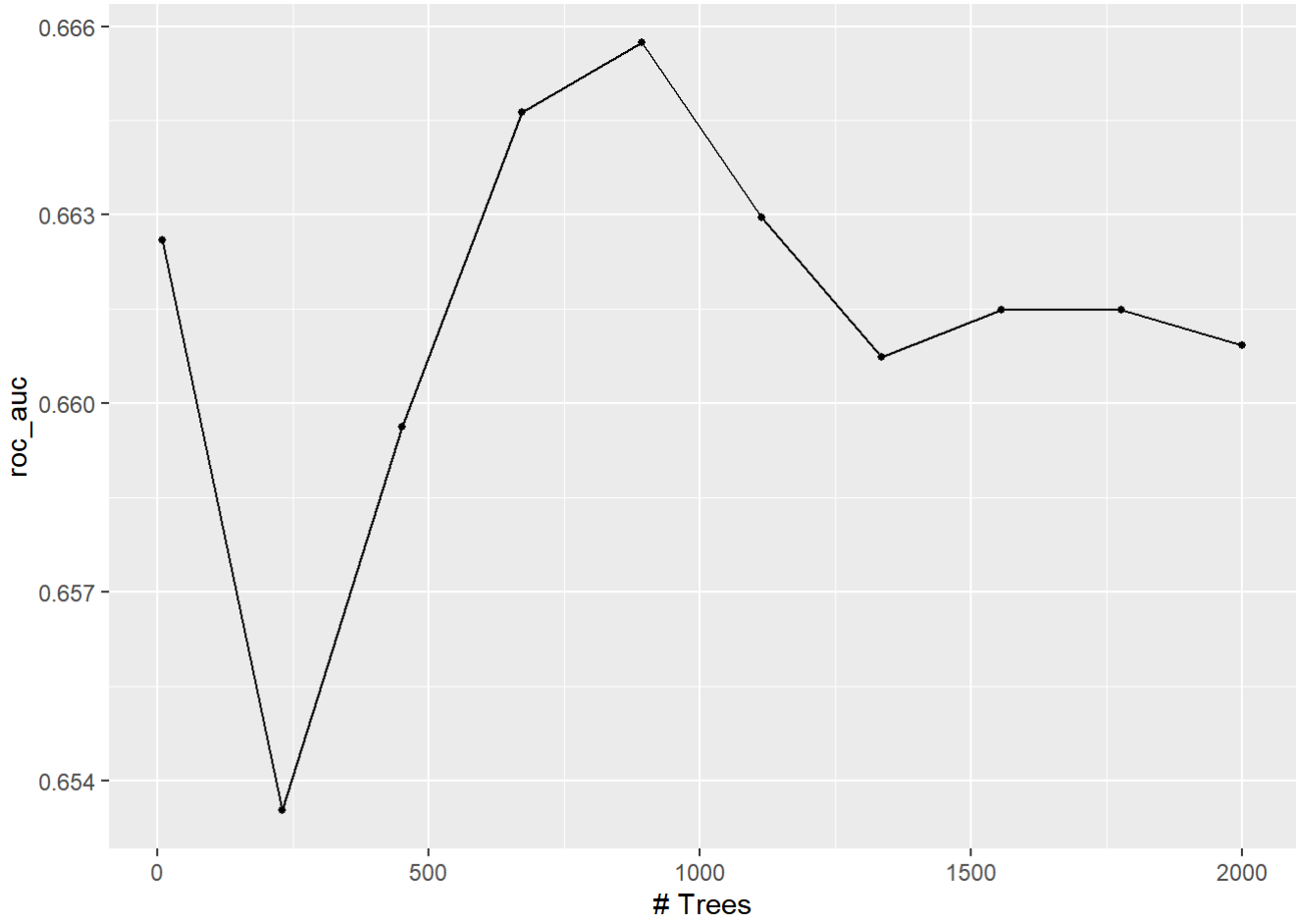
What do you observe?

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

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

workflow_boost <- workflow() %>%
  add_model(model_boost ) %>%
  add_recipe(recipe)

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

```
boost_tune_res <- tune_grid(
  workflow_boost,
  resamples = train_folds,
  grid = boost_grid,
  metrics = metric_set(roc_auc)
  )
```

```
#save(boost_tune_res,file = "C:/Users/DELL/Desktop/1126hw6_131/boost_tune_res.rda")
load("C:/Users/DELL/Desktop/1126hw6_131/boost_tune_res.rda")

autoplot(boost_tune_res)
```




```
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   894 roc_auc hand_till  0.666     5  0.0506 Preprocessor1_Model105
##  2   673 roc_auc hand_till  0.665     5  0.0515 Preprocessor1_Model104
##  3  1115 roc_auc hand_till  0.663     5  0.0511 Preprocessor1_Model106
##  4    10 roc_auc hand_till  0.663     5  0.0516 Preprocessor1_Model101
##  5  1557 roc_auc hand_till  0.661     5  0.0492 Preprocessor1_Model108
##  6  1778 roc_auc hand_till  0.661     5  0.0492 Preprocessor1_Model109
##  7  2000 roc_auc hand_till  0.661     5  0.0500 Preprocessor1_Model110
##  8  1336 roc_auc hand_till  0.661     5  0.0491 Preprocessor1_Model107
##  9   452 roc_auc hand_till  0.660     5  0.0536 Preprocessor1_Model103
## 10   231 roc_auc hand_till  0.654     5  0.0557 Preprocessor1_Model102
```

Answer: From the above figure, we can see that with the increase of the number of trees, the AUC value first decreases greatly, then increases, and finally decreases gradually. The roc_auc of the best-performing boosted tree model on the folds is 0.6657407.

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?

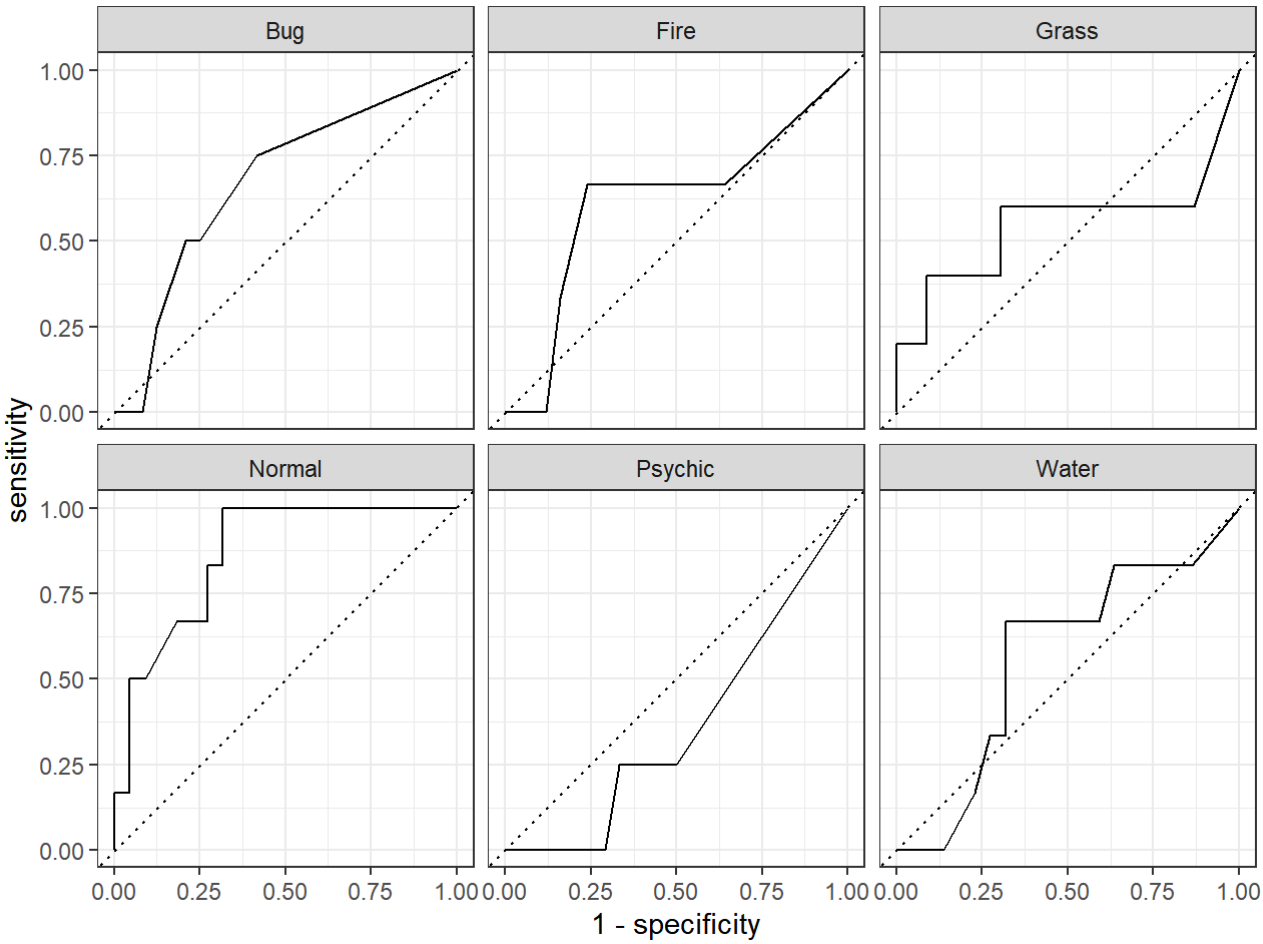
```
roc_auc <- c(0.6090741, 0.7185185, 0.6657407)
models <- c("decision tree model", "random forest model", "boosted tree model")
results <- tibble(roc_auc = roc_auc, models = models)

results %>%
  arrange(-roc_auc)
```

```
## # A tibble: 3 x 2
##   roc_auc models
##     <dbl> <chr>
## 1   0.719 random forest model
## 2   0.666 boosted tree model
## 3   0.609 decision tree model
```

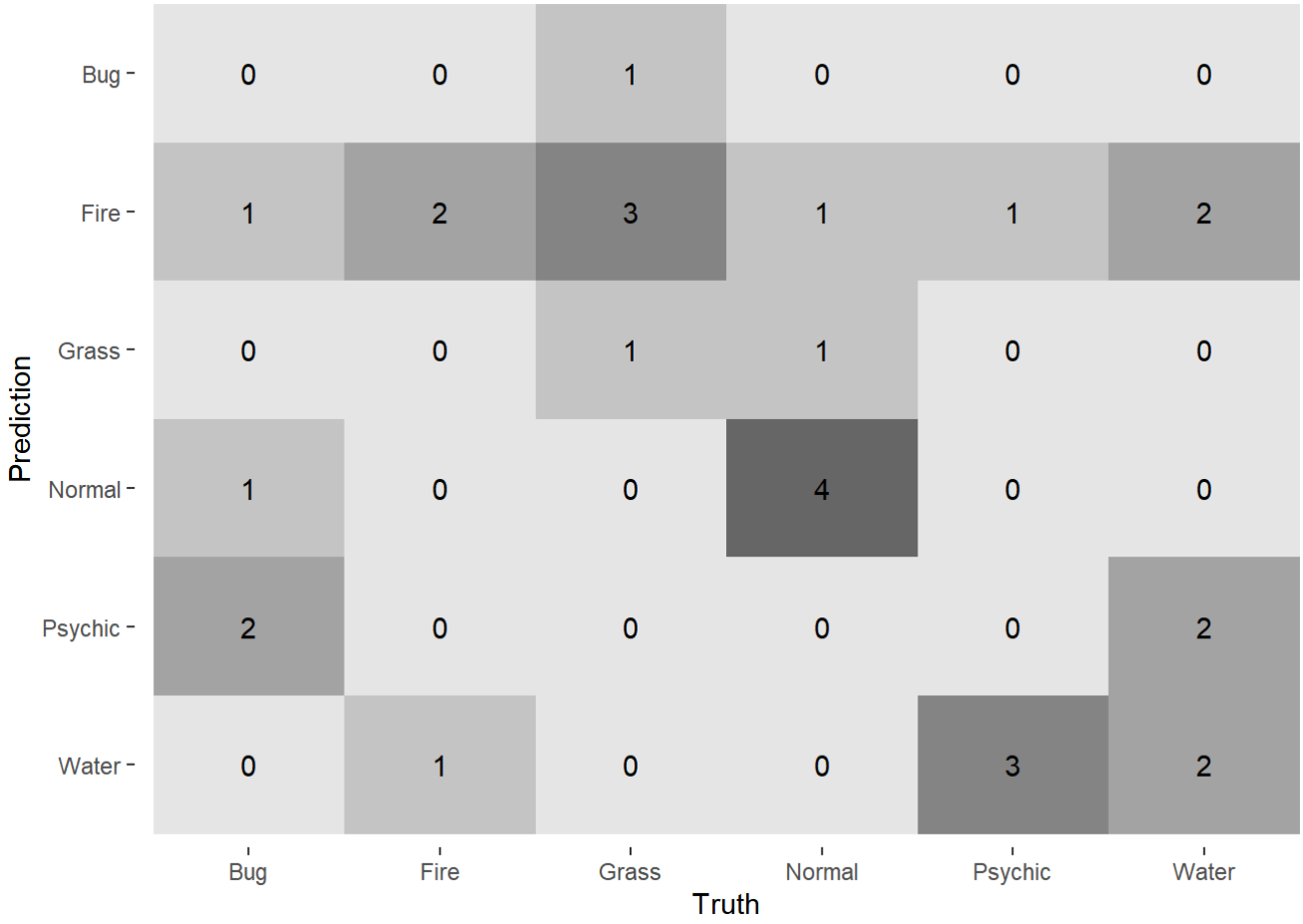
```
best_rf <- select_best(rf_tune_res, metric = "roc_auc")

rf_final <- finalize_workflow(workflow_rf, best_rf)

rf_final_fit <- fit(rf_final, data = train)

augment(rf_final_fit, new_data = test) %>%
  roc_auc(type_1, .pred_Bug:.pred_Water)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.603
```

```
augment(rf_final_fit, new_data = test) %>%
  roc_curve(type_1, .pred_Bug:.pred_Water) %>%
  autoplot()
```




```
augment(rf_final_fit , new_data = test) %>%
  conf_mat(type_1, .pred_class) %>%
  autoplot(type = "heatmap")
```



Answer: The random forest model performed best on the folds. The model is best at predicting Norma and Water and is worst at predicting Grass.