Tree-Based Models

For 231 Students

Exercise 11

PSTAT231HW6

Code

Cheng Ye

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

```
Pokemon_types <- read.csv("C:/Cheng Ye/UCSB/PSTAT 231/HW/homework-6/data/Pokemon.csv")
# Pokemon_types
Pokemon <- clean_names(Pokemon_types)
Pokemon <-Pokemon[Pokemon$type_1 %in% c('Bug', 'Fire', 'Grass', 'Normal', 'Water', 'Psychi
c'), ]
Pokemon$type_1 = factor(Pokemon$type_1)
Pokemon$legendary = factor(Pokemon$legendary)
Pokemon$generation = factor(Pokemon$generation)

set.seed(231) # this could be any number
Pokemon_split <- initial_split(Pokemon, strata = "type_1", prop = 0.8)
Pokemon_train <- training(Pokemon_split)
Pokemon_test <- testing(Pokemon_split)
#K-fold Cross Validation
Pokemon_folds<-vfold_cv(Pokemon_train, v = 5, strata = type_1)
#Creating the recipe
Pokemon_recipe <-
  recipe(formula = type_1 ~ legendary + generation + sp_atk + attack + speed + defense + h
p + sp_def, data = Pokemon_train) %>%
  step_dummy(c('legendary', 'generation')) %>%
  step_normalize(all_predictors())
```

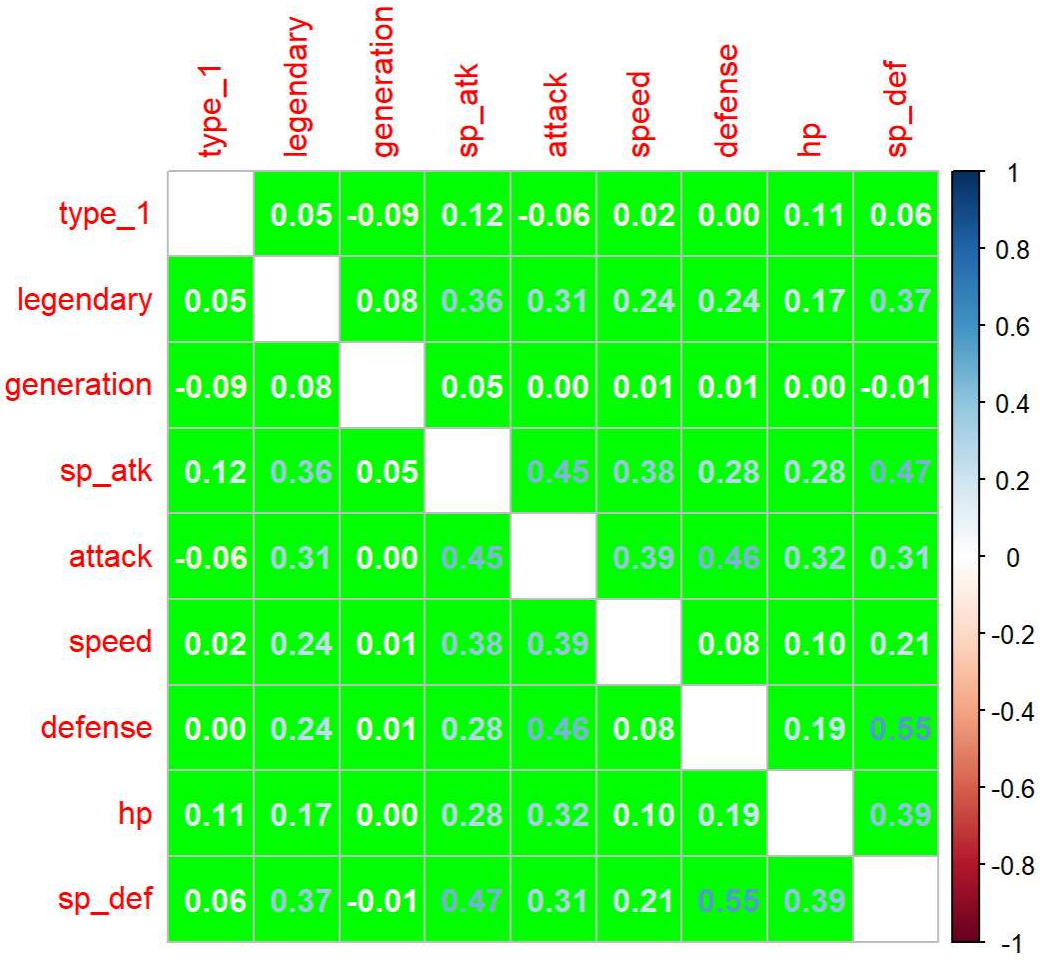
Exercise 2

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?

Hide

```
library(corrplot)
Pokemon_reduce_mutate <- select(Pokemon_train, c('type_1','legendary','generation','sp_at
k','attack','speed','defense','hp','sp_def')) %>%
  mutate_if(is.character, as.factor) %>%
  mutate_if(is.factor, as.numeric)
result <- cor(Pokemon_reduce_mutate)
corrplot(result,diag=FALSE, bg ="green", method = "number")
```



Hide

```
#From the plot, we could observe that there is little correlation between their type and t
heir base stats, however, there is a strong positive correlation between attack and specia
l attack as well as defense and special defense, this makes sense to me as most of the ind
ependent variables show cross correlations besides generation and speed.
```

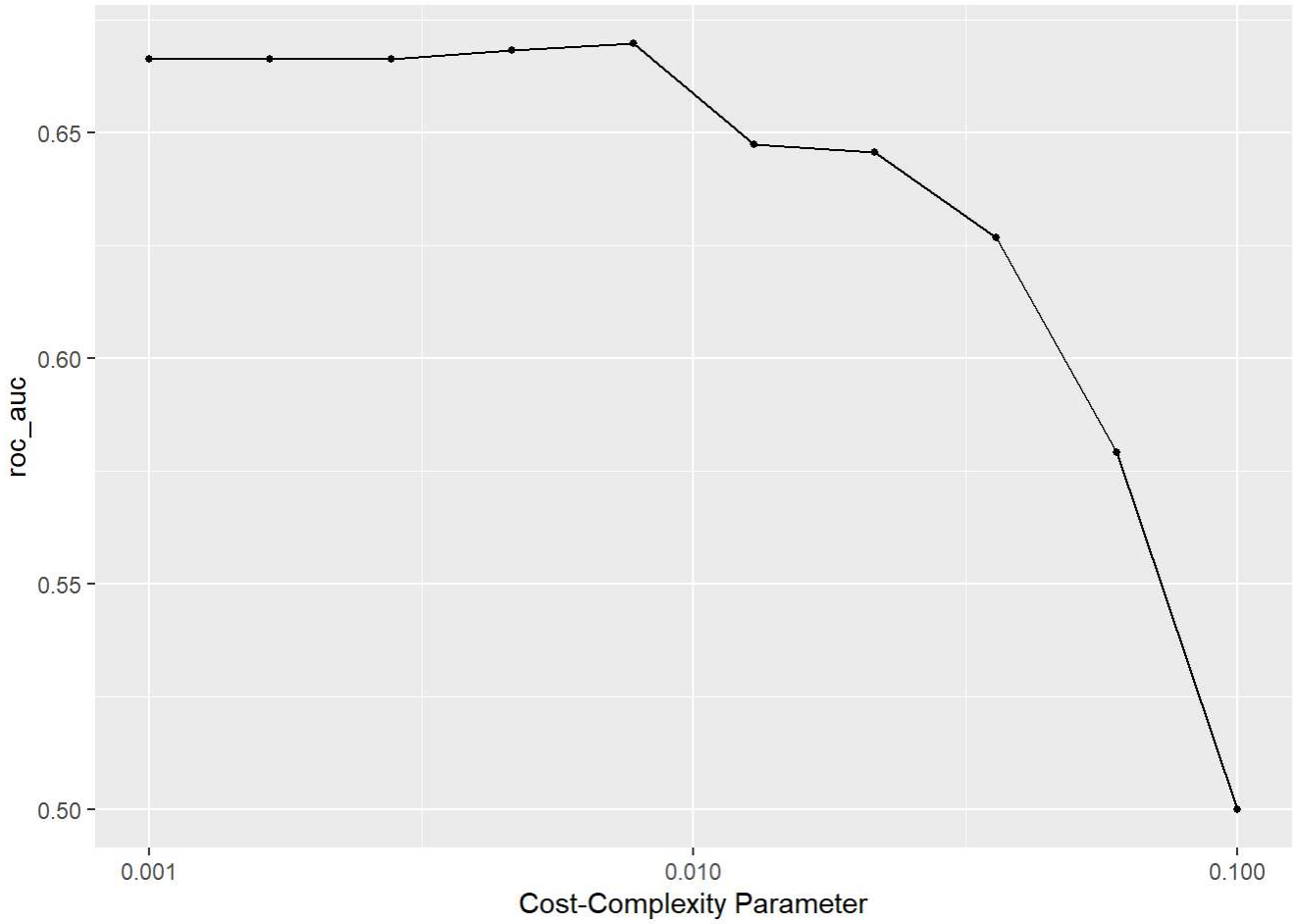
Exercise 3

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

```
decision_tree_model <- decision_tree() %>%
  set_engine("rpart") %>%
  set_mode("classification")

fit_decision_tree <- decision_tree_model %>%
  fit(formula = type_1 ~ legendary + generation + sp_atk + attack + speed + defense + hp +
sp_def, data = Pokemon_train)

decision_tree_wkflow <- workflow() %>%
  add_model(decision_tree_model %>% set_args(cost_complexity = tune())) %>%
  add_formula(type_1 ~ legendary + generation + sp_atk + attack + speed + defense + hp + s
p_def)

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

decision_tree_tune <- tune_grid(
  decision_tree_wkflow,
  resamples = Pokemon_folds,
  grid = param_grid,
  metrics = metric_set(roc_auc)
)

autoplot(decision_tree_tune)
```



Hide

```
#From the results, we could observe that the single decision tree perform better with a re
latively larger complexity penalty within this model while ROC_AUC reaches it's maximum at
0.100.
```

Exercise 4

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

Hide

```
decision_tree_metrics <- collect_metrics(decision_tree_tune) %>%
  arrange(desc(mean))
head(decision_tree_metrics, 1)
```

```
## # A tibble: 1 x 7
##   cost_complexity .metric .estimator  mean     n std_err .config
##             <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1         0.00774 roc_auc hand_till  0.670     5  0.0248 Preprocessor1_Model05
```

Hide

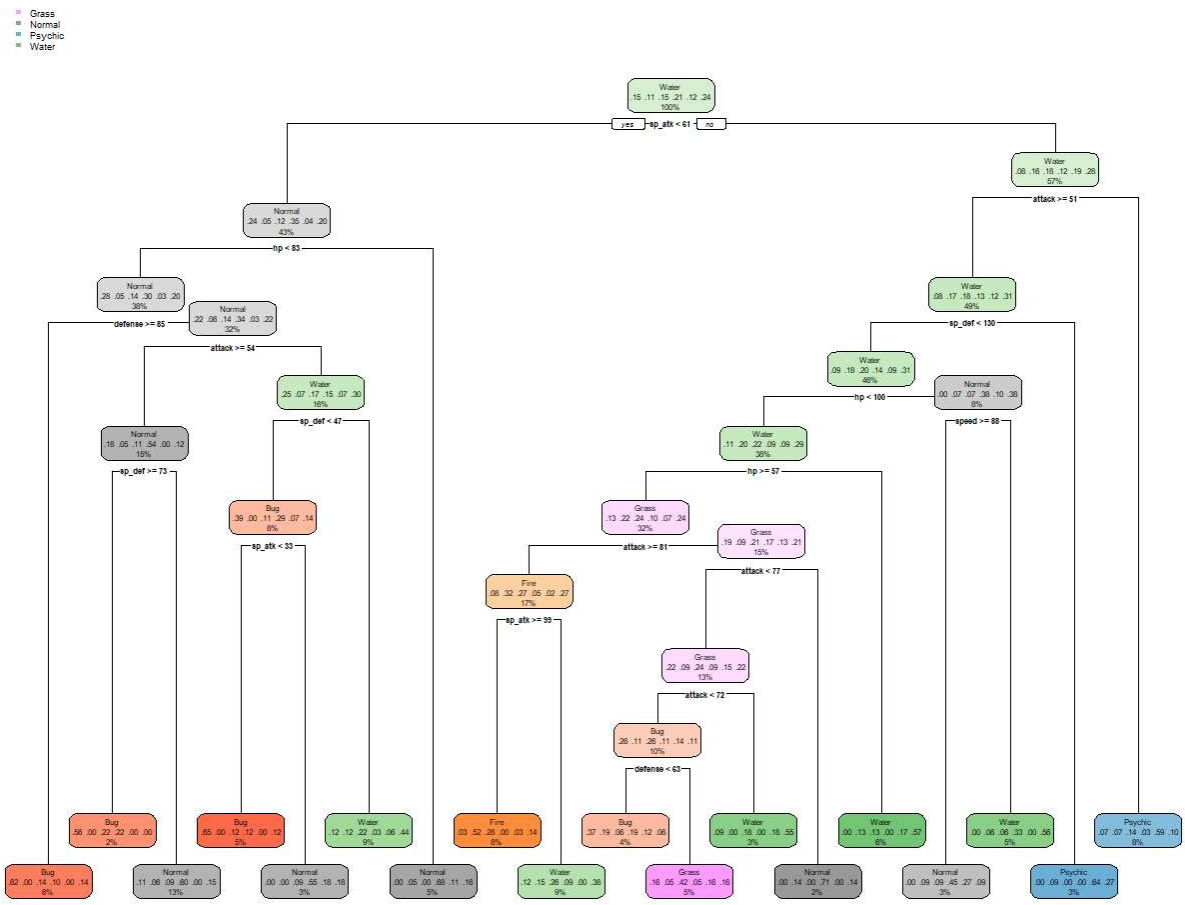
```
# From the results, we could observe that the best cost complexity is 0.007742637 with ROC
_AUC of 0.6696856
```

Exercise 5

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

Hide

```
best_complexity <- select_best(decision_tree_tune)
tree_final <- finalize_workflow(decision_tree_wkflow, best_complexity)
tree_final_fit <- fit(tree_final, data = Pokemon_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?**

Hide

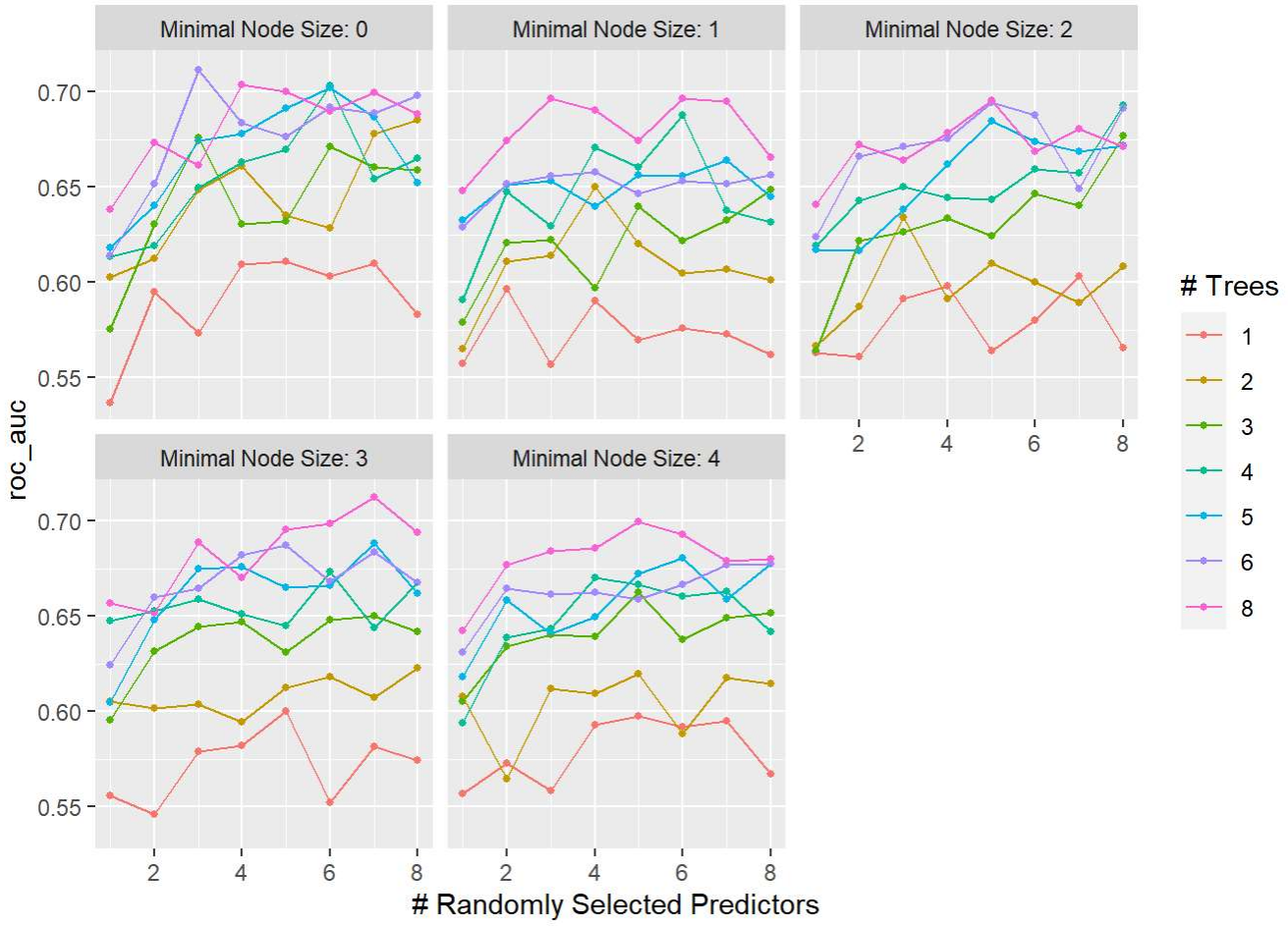
```
bagging_spec <- rand_forest(mtry = .cols()) %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("classification") %>%
  set_args(mtry = tune(),trees = tune(), min_n = tune())

rand_tree_wkflow <- workflow() %>%
  add_recipe(Pokemon_recipe) %>%
  add_model(bagging_spec)

Forest_grid <- grid_regular(mtry(range = c(1, 8)),trees(range = c(0, 8)), min_n(range = c(
0, 4)),levels = 8)

#mtry is number of predictors randomly sampled at each split when creating the tree model
s. trees is the integer for the sample that is tested. min_n is the integer for the minimu
m number of data points in a node to split further.

#mtry = 1 means we randomly select one predictor for the test. Hence in this case mtry = 8
means select all predictors whenever we run the model
```

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

```
tune_res_rand <- tune_grid(
  object = rand_tree_wkflow,
  resamples = Pokemon_folds,
  grid = Forest_grid ,
  metrics = metric_set(... = roc_auc)
)
autoplot(tune_res_rand)
```



Hide

```
#From the results, we could observe that minimum node 4 seems to yield the best performanc
e
```

Exercise 7

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

Hide

```
random_forest_metrics <- collect_metrics(tune_res_rand) %>%
  arrange(desc(mean))
head(random_forest_metrics,1)
```

```
## # 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     7     8     3 roc_auc hand_till  0.713     5  0.0305 Preprocessor1_Model2~
```

Hide

```
#By Observing the results, 0.7093549 seems to be the best performing roc_auc
```

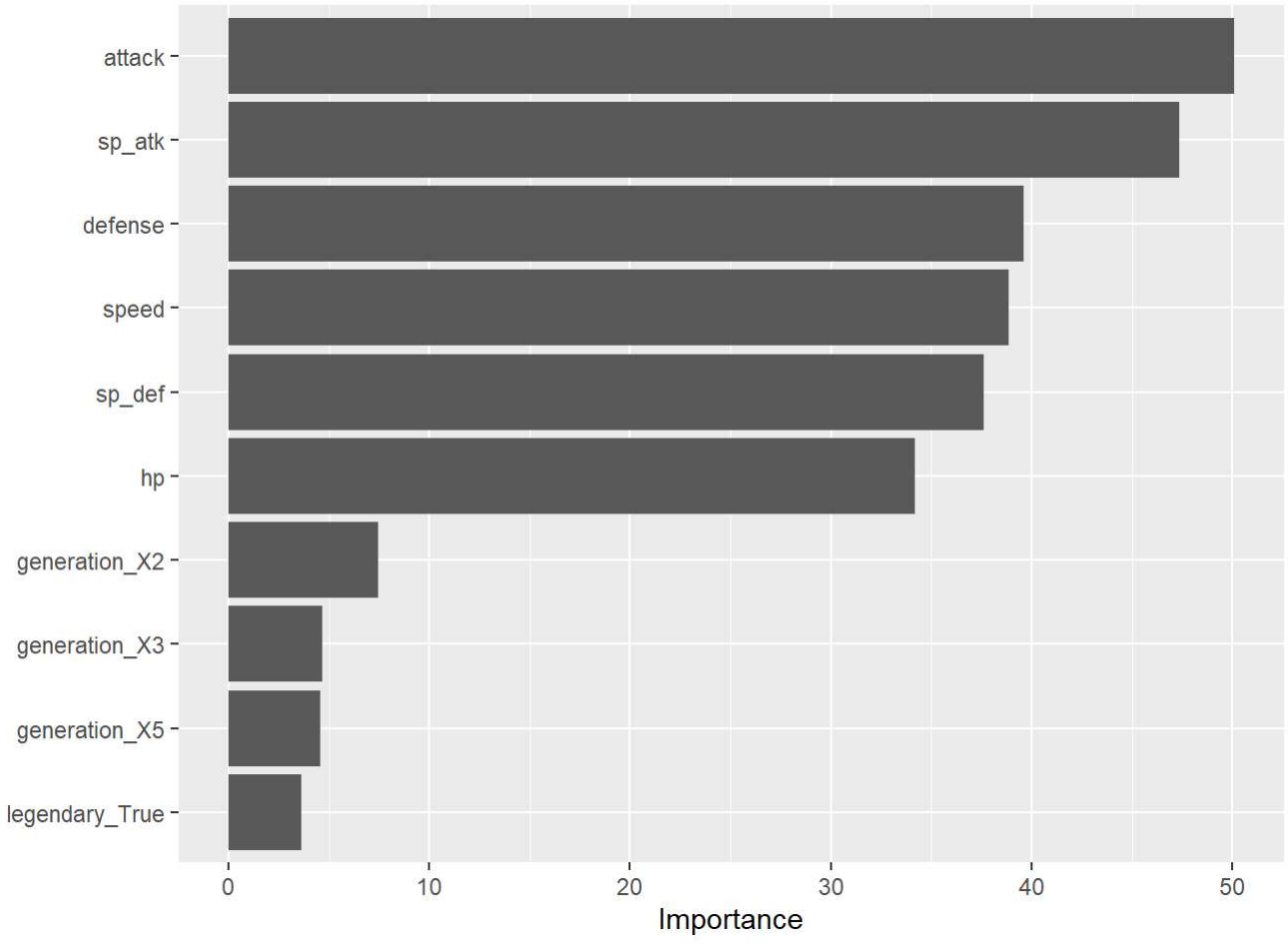
Exercise 8

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

```
best_roc_auc <- select_best(tune_res_rand)

rand_tree_final <- finalize_workflow(rand_tree_wkflow, best_roc_auc)

rf_fit <- fit(rand_tree_final, data = Pokemon_train)

extract_fit_engine(rf_fit) %>%
 vip()
```



Hide

```
#By observing the results, we could see that sp_atk is the most useful, generation X2 is t
he least useful. It is somewhat similar to what I predicted in the previous homework (SP_A
TK being most useful).
```

Exercise 9

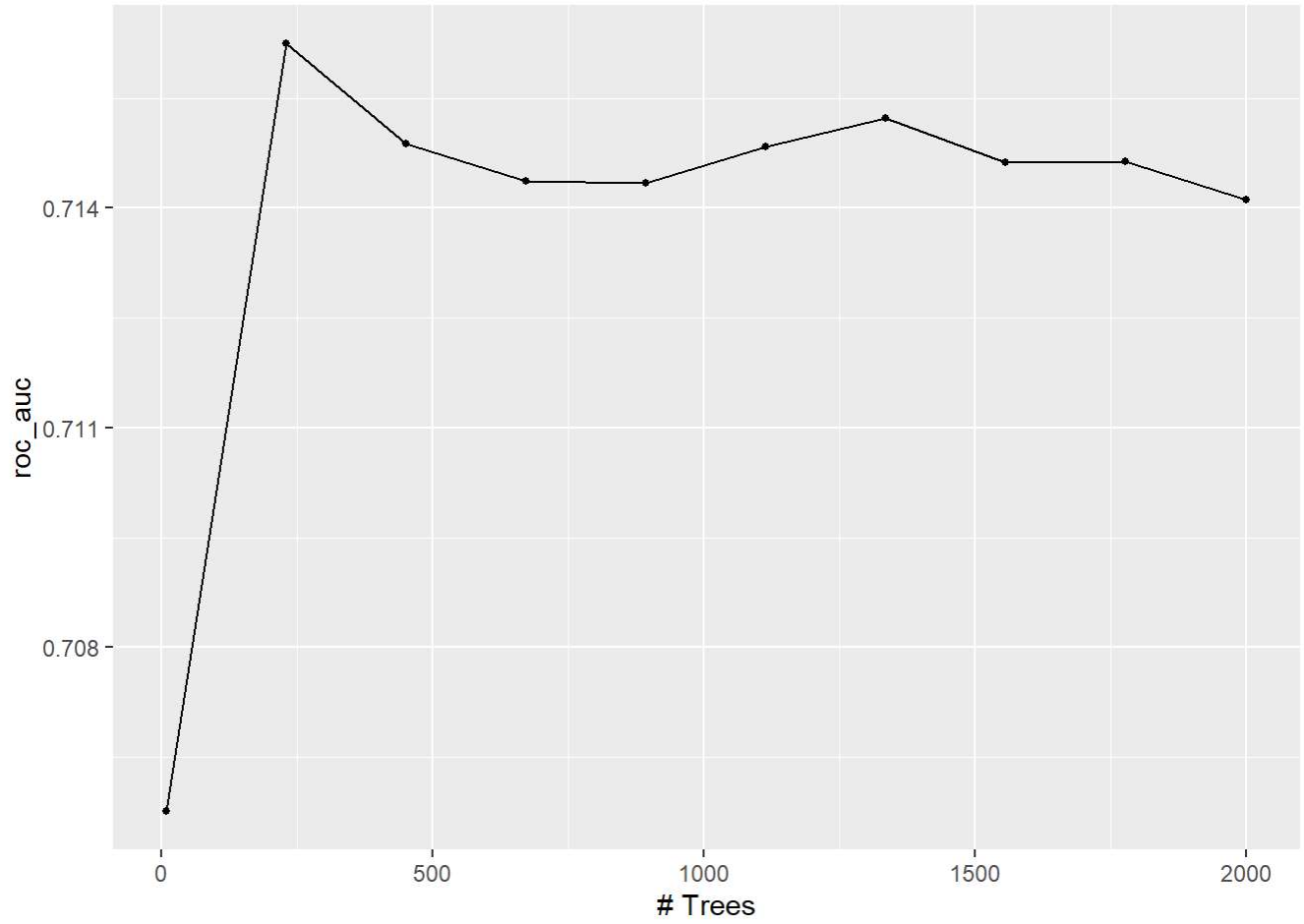
Finally, set up a boosted tree model and workflow. Use the `xgboost` engine. Tune `trees`. Create a regular grid with 10 levels; let `trees` range from 10 to 2000. Specify `roc_auc` and again print an `autoplot()` of the results.

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

```
set.seed(2000)
boost_spec <- boost_tree() %>%
  set_engine("xgboost") %>%
  set_mode("classification")

boost_wkflow <- 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_wkflow,
  resamples = Pokemon_folds,
  grid = boost_grid,
  metrics = metric_set(roc_auc)
)
autoplot(boost_tune_res)
```



Hide

```
collect_metrics(boost_tune_res)%>%
  arrange(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  0.716     5  0.0215 Preprocessor1_Model02
##  2  1336 roc_auc hand_till  0.715     5  0.0202 Preprocessor1_Model07
##  3   452 roc_auc hand_till  0.715     5  0.0211 Preprocessor1_Model03
##  4  1115 roc_auc hand_till  0.715     5  0.0201 Preprocessor1_Model06
##  5  1778 roc_auc hand_till  0.715     5  0.0195 Preprocessor1_Model09
##  6  1557 roc_auc hand_till  0.715     5  0.0198 Preprocessor1_Model08
##  7   673 roc_auc hand_till  0.714     5  0.0210 Preprocessor1_Model04
##  8   894 roc_auc hand_till  0.714     5  0.0202 Preprocessor1_Model05
##  9  2000 roc_auc hand_till  0.714     5  0.0190 Preprocessor1_Model10
## 10    10 roc_auc hand_till  0.706     5  0.0180 Preprocessor1_Model01
```

Hide

```
#From the results we could observe that the model roc_auc reaches its maximum at around 23
1 trees.The maximum ROC_AUC is 0.7162454
```

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 best performing prune tree
best_complexity <- select_best(decision_tree_tune)
tree_final <- finalize_workflow(decision_tree_wkflow, best_complexity)
tree_final_fit <- fit(tree_final, data = Pokemon_train)

#Display best performing random forest
best_roc_auc <- select_best(tune_res_rand)
random_final <- finalize_workflow(rand_tree_wkflow,best_roc_auc)
random_fit <- fit(random_final, data = Pokemon_train)

#Display best performing boost model
best_boost <- select_best(boost_tune_res)
boost_final <- finalize_workflow(boost_wkflow,best_boost)
boost_fit <- fit(boost_final, data = Pokemon_train)


#roc_auc
decision_tree_fit <- augment(tree_final_fit,
                         new_data = Pokemon_test,
                         type = "prob") %>%
  mutate(type_1 = as.factor(type_1)) %>%
  roc_auc(truth = type_1, .pred_Bug:.pred_Water)

random_tree_fit <- augment(random_fit,
                         new_data = Pokemon_test,
                         type = "prob") %>%
  mutate(type_1 = as.factor(type_1)) %>%
  roc_auc(truth = type_1, .pred_Bug:.pred_Water)

boost_model_fit <- augment(boost_fit,
                         new_data = Pokemon_test,
                         type = "prob") %>%
  mutate(type_1 = as.factor(type_1)) %>%
  roc_auc(truth = type_1, .pred_Bug:.pred_Water)
table_rocauc <- bind_rows("ROC_AUC:Decision Tree model"= decision_tree_fit,"ROC_AUC:Random
Forest model"=random_tree_fit,"ROC_AUC:Boosted model"=boost_model_fit,.id="model")
table_rocauc
```

```
## # A tibble: 3 x 4
##   model                        .metric .estimator .estimate
##   <chr>                        <chr>   <chr>          <dbl>
## 1 ROC_AUC:Decision Tree model roc_auc hand_till      0.612
## 2 ROC_AUC:Random Forest model roc_auc hand_till      0.711
## 3 ROC_AUC:Boosted model       roc_auc hand_till      0.670
```

Hide

```
#From the results, we could observe that the Boosted model performed the best while the De
cision tree model performed the worst
```

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

```
abalone <- read.csv("C:/Cheng Ye/UCSB/PSTAT 231/HW/homework-6/data/abalone.csv")
abalone$age <- abalone$ring+1.5
set.seed(231)
abalone_split<-initial_split(abalone,prop=0.80,strata=age) #Train 80%, test 20%
abalone_train<-training(abalone_split) #Training dataset
abalone_test<-testing(abalone_split) #Testing dataset
abalone_train_data <-subset(abalone_train,select=-rings)
abalone_fold <- vfold_cv(abalone_train,v=5)
abalone_age_recipe <- recipe(age ~ . , data = abalone_train_data) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_center(all_nominal_predictors()) %>%
  step_scale(all_nominal_predictors()) %>%
  step_interact(terms = ~ starts_with("type"):shucked_weight) %>%
  step_interact(terms = ~ longest_shell:diameter) %>%
  step_interact(terms = ~ shucked_weight:shell_weight)

# Using random forest Model
abalone_random_forest <- rand_forest(mtry = .cols()) %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("regression") %>%
  set_args(mtry = tune(),trees = tune(), min_n = tune())

abalone_rand_tree_wkflow <- workflow() %>%
  add_recipe(abalone_age_recipe) %>%
  add_model(abalone_random_forest )

abalone_grid <- grid_regular(mtry(range = c(1, 8)),trees(range = c(0, 8)), min_n(range = c
(0, 4)),levels = 8)


abalone_random_tune <- tune_grid(
  object = abalone_rand_tree_wkflow,
  resamples = abalone_fold,
  grid = abalone_grid ,
  metrics = metric_set(... = rmse)
)

best_metric <- select_best(abalone_random_tune)
abalone_final <- finalize_workflow(abalone_rand_tree_wkflow, best_metric)
abalone_fit <- fit(abalone_final, data = abalone_train)

abalone_metrics <- metric_set(rmse, rsq, mae)
abalone_test_predict <- predict(abalone_fit,abalone_test %>% select(-age))

abalone_test_res <- bind_cols(abalone_test_predict, abalone_test %>% select(age))

abalone_metrics(abalone_test_res, truth = age, estimate = .pred)
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard       2.30
## 2 rsq     standard       0.519
## 3 mae     standard       1.66
```

Hide

```
# The Results are shown below
```