PSTAT131HW6

Yiting Zhang

2022-05-24

Exercise 1

```
pokemon_data <- read.csv("Pokemon.csv",fileEncoding = "UTF8")
pokemon <- pokemon_data %>%
  clean_names()
```

```
# filter type_1
filtered_pokemon_types <- pokemon %>%
   filter(type_1 == "Bug" | type_1 == "Fire" |
            type_1 == "Grass" | type_1 == "Normal" |
            type_1 == "Water" | type_1 == "Psychic")
```

```
# convert `type_1` and `legendary` to factors
pokemon_factored <- filtered_pokemon_types %>%
  mutate(type_1 = factor(type_1)) %>%
  mutate(legendary = factor(legendary)) %>%
  mutate(generation = factor(generation))
```

```
set.seed(100)
# initial split of the data
pokemon_split <- initial_split(pokemon_factored, strata = type_1, prop = 0.7)
pokemon_train <- training(pokemon_split)
pokemon_test <- testing(pokemon_split)
```

```
# *v*-fold cross-validation
pokemon_fold <- vfold_cv(pokemon_train, strata = type_1, v = 5)
```

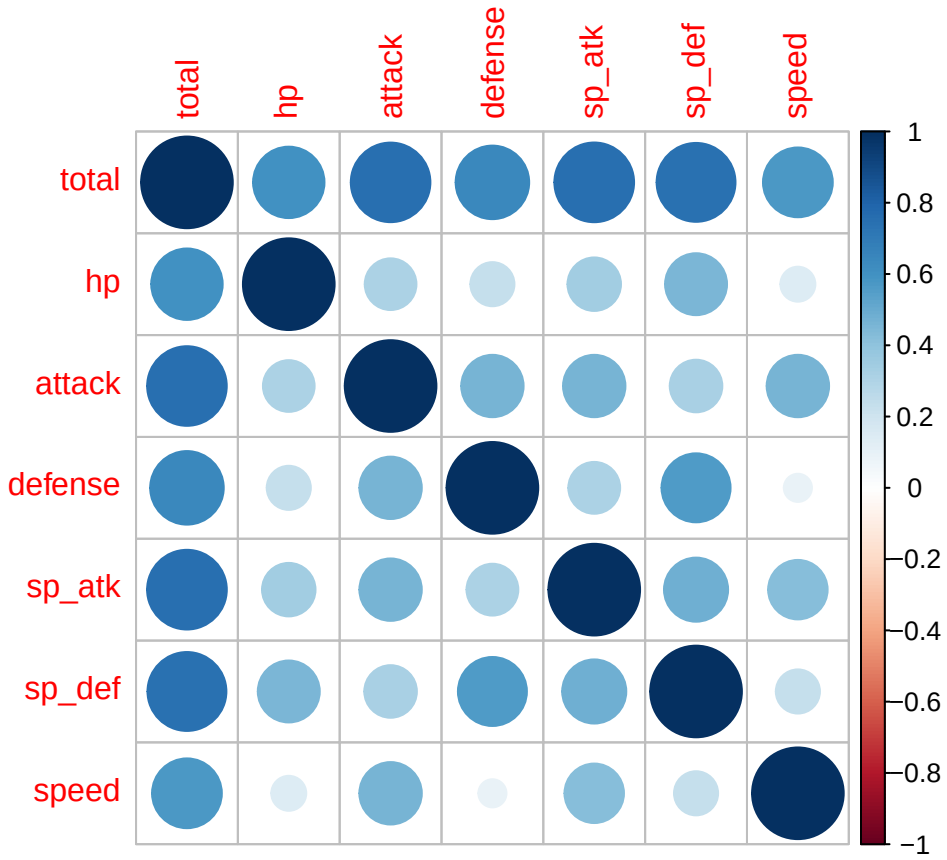
```
# recipe
pokemon_recipe <- recipe(type_1 ~ legendary + generation +
                           sp_atk + attack + speed + defense +
                           hp + sp_def, data = pokemon_train) %>%
  step_dummy(legendary, generation) %>%
  step_center(all_predictors()) %>%
  step_scale(all_predictors())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

```
## Recipe
##
```

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

Exercise 2

```
pokemon_train %>%
  select(-x) %>%
  select(is.numeric) %>%
  cor() %>%
  corrplot()
```



We can first remove the numeric variable x which records the ID number of each pokemon. Because ID is unique to each one thus is helpless to our prediction. And we can also remove all the categorical variables.

From the correlation matrix, we can see the total is positively correlated to all the six battle statistics including hp, attack, defense, sp_atk,sp_def, and speed.

I think the plot makes sense to me. As the total is obviously the sum of the overall power of a pokemon. If the pokemon is better in some of the fields the overall score would be definitely higher.

Exercise 3

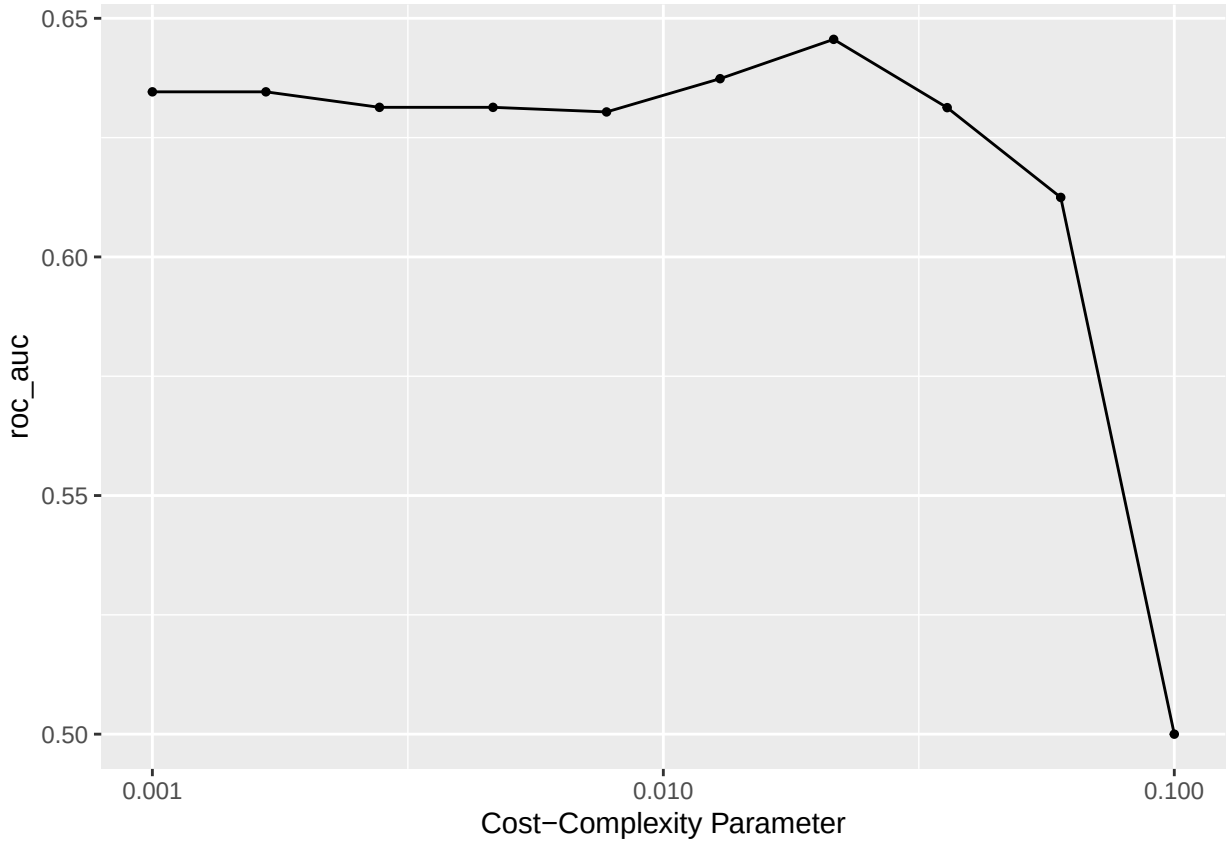
```
# decision tree
tree <- decision_tree() %>%
  set_engine("rpart") %>%
  set_mode("classification") %>%
  set_args(cost_complexity = tune())
```

```
# workflow
tree_wf <- workflow() %>%
  add_model(tree) %>%
  add_recipe(pokemon_recipe)
```

```
# tune the cost_comlexity
pokemon_param_grid <- grid_regular(cost_complexity(range = c(-3, -1)),
                                   levels = 10)
```

```
pokemon_tune_res <- tune_grid(
  tree_wf,
  resamples = pokemon_fold,
  grid = pokemon_param_grid,
  metrics = metric_set(roc_auc)
)
```

```
# autoplot
autoplot(pokemon_tune_res)
```



We can observe that the cost-complexity parameter first increases, and the roc_auc also increases. However, after the value of roc_auc reached the peak value, the roc_auc decreases as the cost-complexity parameter increase. Thus, it seems like a single decision tree perform better with smaller complexity penalty.

Exercise 4

```
best_tree_roc_auc <- collect_metrics(pokemon_tune_res) %>%
  arrange(-mean) %>%
  filter(row_number()==1)
best_tree_roc_auc
```

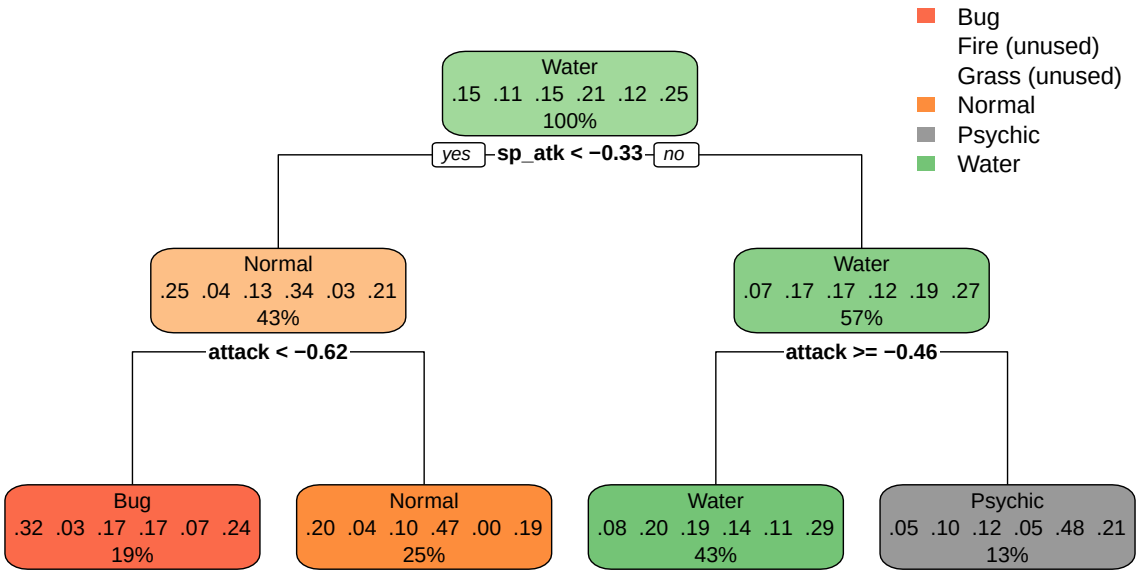
```
## # A tibble: 1 x 7
##   cost_complexity .metric .estimator  mean     n std_err .config
##             <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1          0.0215 roc_auc hand_till  0.646     5  0.0226 Preprocessor1_Model07
```

The roc_auc of the best-performing pruned decision tree on the folds is 0.6455953.

Exercise 5

```r
# fit the best_performing pruned decision tree with the training set
best_complexity <- select_best(pokemon_tune_res)
tree_final <- finalize_workflow(tree_wf, best_complexity)
tree_final_fit <- fit(tree_final, data = pokemon_train)
```

```r
# visualize
tree_final_fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```




```r
# set up a random forest model
pokemon_rf <- rand_forest() %>%
  set_args (mtry = tune(), trees = tune(), min_n = tune()) %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("classification")
```

```r
# set up a random forest workflow
rf_wf <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(pokemon_rf)
```

```r
# create a regular grid
rfp_grid <- grid_regular(mtry(range = c(2,7)),
                         trees(range = c(10,2000)),
```

```
                          min_n(range = c(2, 10)),
                          levels = 8)
```

mtry: the number of predictors that will be randomly chosen for each split of the tree models.

tree: the number of trees in the tree models

min_n: the minimum number of data points in a nod required for splitting

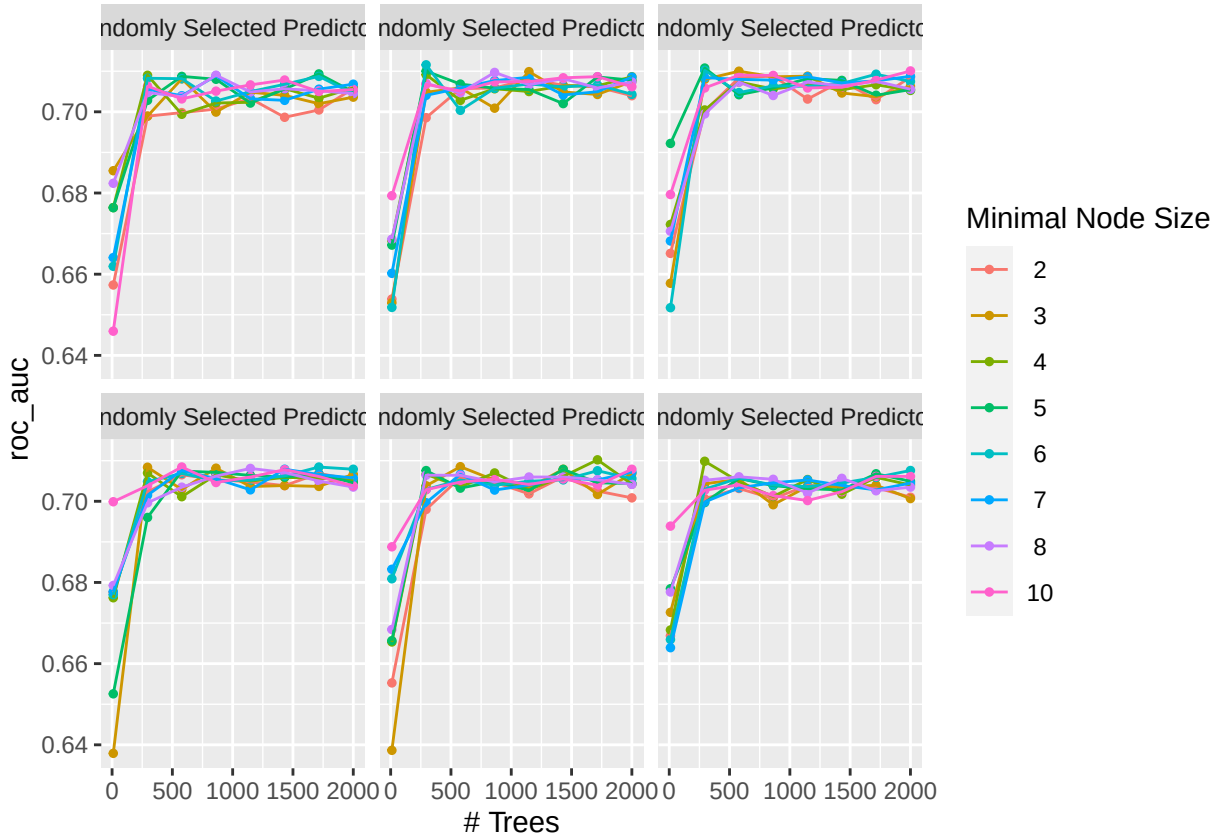
mtry should not be smaller than 1 or larger than 8 We only have 8 predictors in our specified model We can not have 0 predictor in the model

mtry = 8 represents the bagging model

Exercise 6

```
pokemon_tune_rf <- tune_grid(rf_wf,
  resamples = pokemon_fold,
  grid = rfp_grid,
  metrics = metric_set(roc_auc))
```

```
autoplot(pokemon_tune_rf)
```



We can observe that while the number of trees increases, the value of roc_auc also increases. However, when the number of trees is larger than about 125, the value of roc_auc will stop increasing dramatically but rather will fluctuate around. It will not increase significantly anymore even if we add more trees.

The difference among the minimum nod sizes is not very huge. We can find that smaller minimum nod sizes tends to have larger roc_auc, which means perform slightly better. Same as the difference among the the number of randomly selected predictors. We can still find that the number 3,4,5 of randomly selected predictors have larger roc_auc, which means perform slightly better.

Exercise 7

```
rf_roc_auc <- collect_metrics(pokemon_tune_rf) %>%
  arrange(-mean) %>%
  filter(row_number()==1)
rf_roc_auc
```

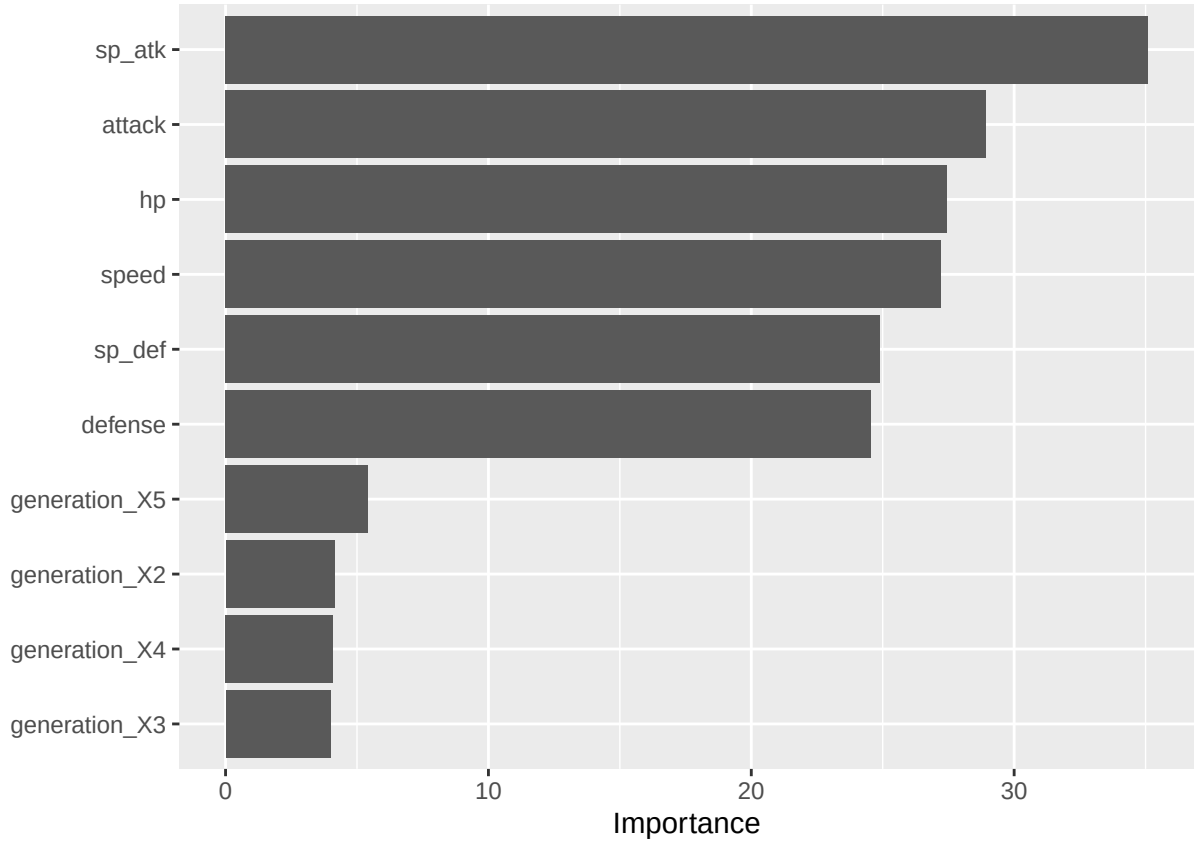
```
## # A tibble: 1 x 9
##    mtry trees min_n .metric .estimator  mean     n std_err .config
##   <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1     3   294     6 roc_auc hand_till  0.712     5  0.0156 Preprocessor1_Model2~
```

The best-performing random forest model roc_auc is 0.7147734

Exercise 8

```
best_rf <- select_best(pokemon_tune_rf, metric = "roc_auc")
rf_final <- finalize_workflow(rf_wf, best_rf)
rf_final_fit <- fit(rf_final, data = pokemon_train)
```

```
rf_final_fit %>%
  extract_fit_engine() %>%
  vip()
```




```
rf_final_fit %>%
  extract_fit_engine() %>%
  vip::vi() %>%
  arrange(Importance)
```

```
## # A tibble: 12 x 2
##    Variable       Importance
##    <chr>               <dbl>
##  1 legendary_True       2.09
##  2 generation_X6        2.60
##  3 generation_X3        3.99
##  4 generation_X4        4.10
##  5 generation_X2        4.14
##  6 generation_X5        5.41
##  7 defense             24.6
##  8 sp_def              24.9
##  9 speed               27.2
## 10 hp                  27.4
## 11 attack              28.9
## 12 sp_atk              35.1
```

The most useful variable is sp_attack, and the least useful variable is legendary_True.

This is what I expected becauce other not that useful variables are not that relavant to the type_1 of the pokemon which determines the weakness and strongness.

Exercise 9

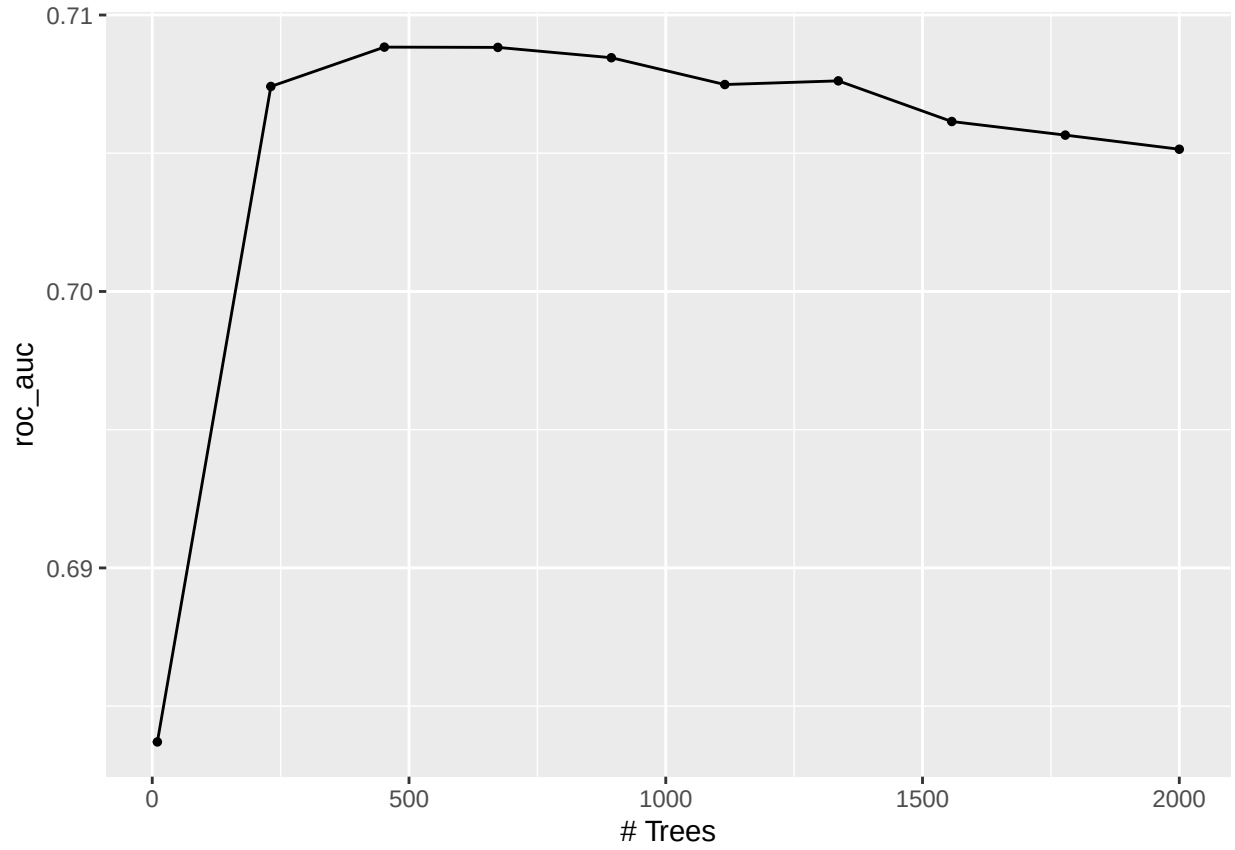
```
# set up a boosted tree model
boost_spec <- boost_tree(trees = tune()) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

#  set up a boosted tree workflow
boost_wf <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(boost_spec)
```

```
# Create a regular grid
boost_grid <- grid_regular(trees(range = c(10,2000)), levels = 10)
```

```
tune_boost <- tune_grid(boost_wf,
                        resamples = pokemon_fold,
                        grid = boost_grid,
                        metrics = metric_set(roc_auc))
```

```
autoplot(tune_boost)
```



The roc_auc increases and after reaching at the highest point around 250 trees, then the roc_auc seems to gradually decrease

```
boost_roc_auc <- collect_metrics(tune_boost) %>%
  arrange(-mean) %>%
  filter(row_number()==1)
boost_roc_auc
```

```
## # A tibble: 1 x 7
##   trees .metric .estimator  mean     n std_err .config
##   <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1   452 roc_auc hand_till  0.709     5  0.0118 Preprocessor1_Model03
```

The roc_auc of the best-performing model is 0.7088385

Exercise 10

```
# check best roc_auc
roc_auc <- c(best_tree_roc_auc$mean, rf_roc_auc$mean, boost_roc_auc$mean)
models <- c("Decision Tree", "Random Forest", "Boosted Tree")
results <- tibble(roc_auc = roc_auc, models = models)
results %>%
  arrange(-roc_auc)
```

```
## # A tibble: 3 x 2
##   roc_auc models
##     <dbl> <chr>
## 1   0.712 Random Forest
## 2   0.709 Boosted Tree
## 3   0.646 Decision Tree
```

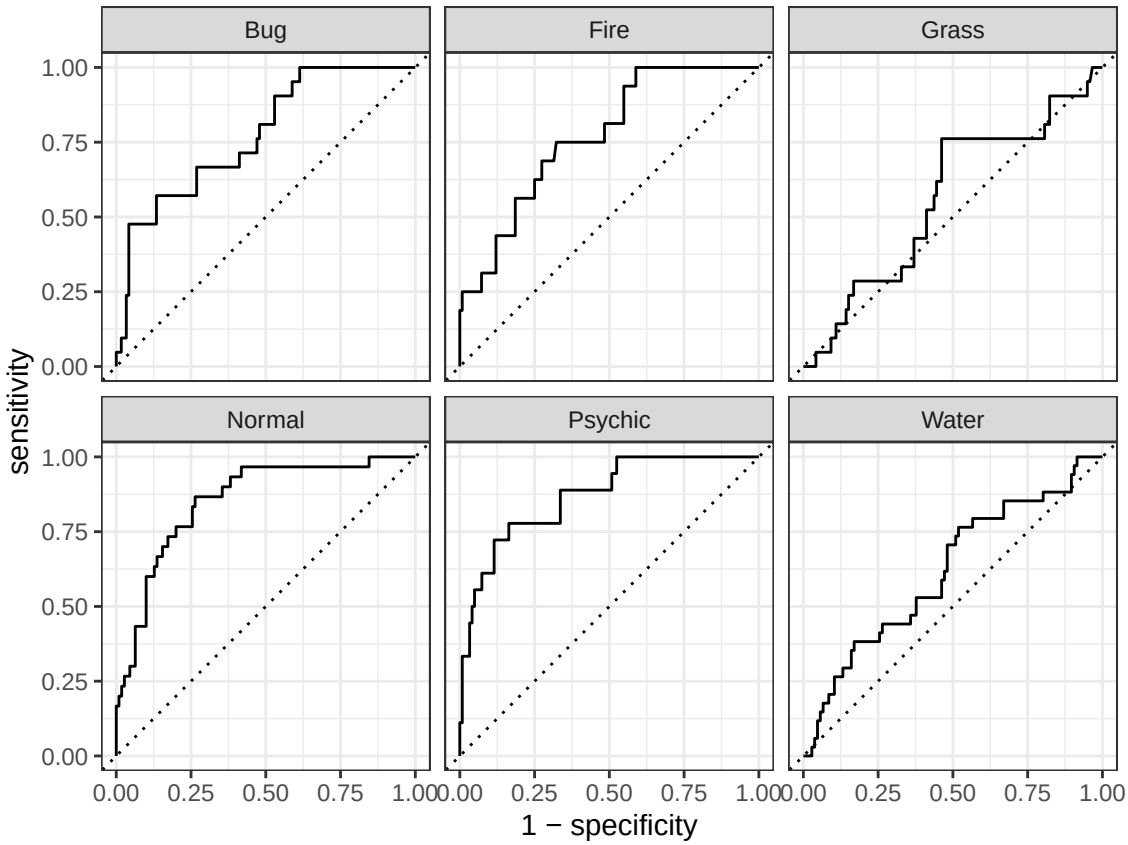
random forest performs the best on the folds

```
# fit to the testing set
best_rf<- select_best(pokemon_tune_rf, metric = "roc_auc")
pokemon_final_test <- finalize_workflow(rf_wf, best_rf)
pokemon_final_fit <- fit(pokemon_final_test, data = pokemon_train)
```

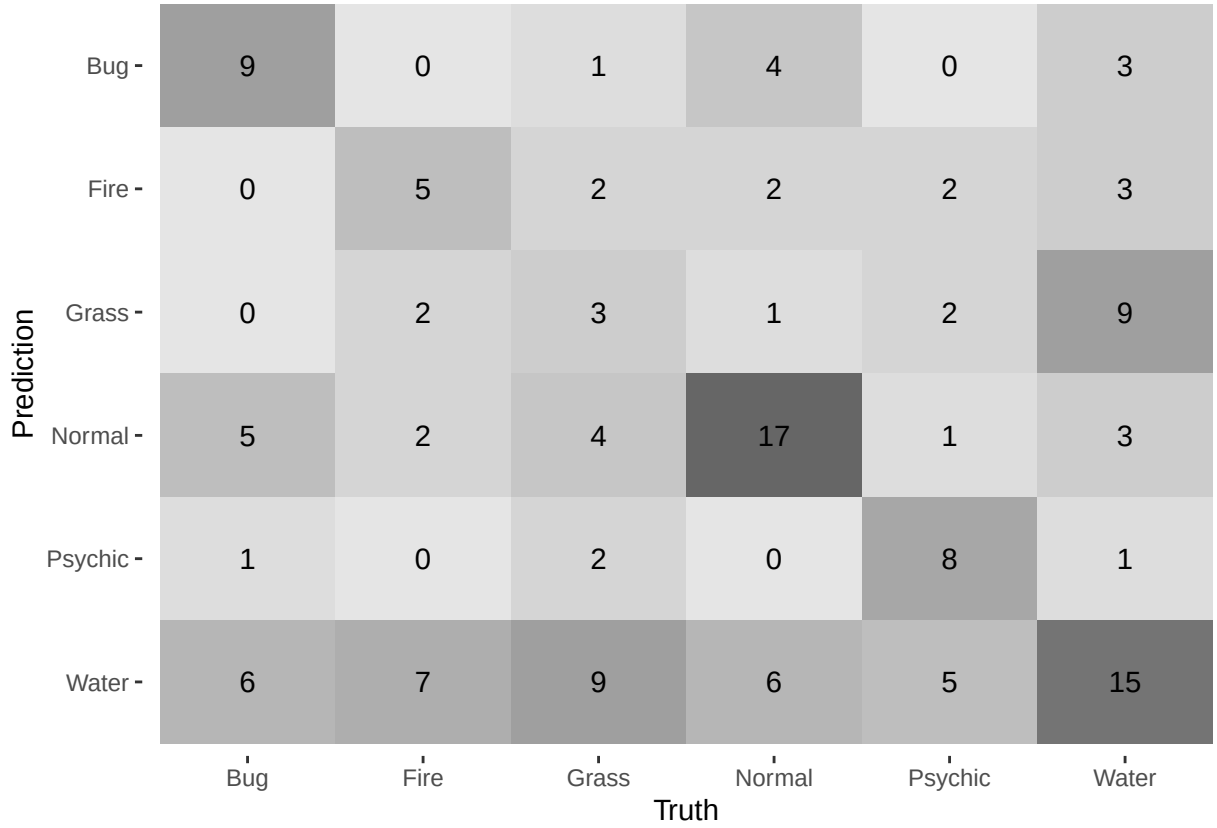
```
# print auc values
roc_auc(augment(pokemon_final_fit,
                new_data = pokemon_test),
        type_1,.pred_Bug, .pred_Fire,
        .pred_Grass,.pred_Normal, .pred_Psychic, .pred_Water)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.742
```

```
# Print the ROC curves.
augment(pokemon_final_fit, new_data = pokemon_test) %>%
roc_curve(type_1, .pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal,
.pred_Psychic,.pred_Water) %>%
autoplot()
```




```
# create and visualize a confusion matrix heat map.
augment(pokemon_final_fit, new_data = pokemon_test) %>%
conf_mat(truth = type_1, estimate = .pred_class) %>%
autoplot(type = "heatmap")
```



Model accurate at predicting Normal and psychic, and Water class was the worst.