Homework 6

Jules Merigot (8488256)

November 26, 2022

PSTAT 131/231 Statistical Machine Learning - Fall 2022

Tree-Based Models

Exercise 1

Before we get started, let's load the Pokemon data in into our workspace.

```
Pokemon_data <- read.csv(file = "C:/Users/jules/OneDrive/Desktop/homework-5/data/Pokemon.csv")
```

Let's load the janitor package, and use its `clean_names()` function on the Pokémon data. We'll save the results to work with for the rest of the assignment.

```
library(janitor)
```

```
##
## Attaching package: 'janitor'

## The following objects are masked from 'package:stats':
##
##     chisq.test, fisher.test
```

```
Pokemon_data <- Pokemon_data %>%
  clean_names()
```

For this assignment, we'll handle the rarer classes by simply filtering them out. Let's filter the entire data set to contain only Pokemon whose `type_1` is Bug, Fire, Grass, Normal, Water, or Psychic.

```
Pokemon_data <- Pokemon_data %>%
  filter(grepl("Bug|Fire|Grass|Normal|Water|Psychic", type_1))
```

Now that we're done filtering, let's convert `type_1`, `legendary`, and `generation` to factors.

```
Pokemon_data$type_1 <- factor(Pokemon_data$type_1)
Pokemon_data$legendary <- factor(Pokemon_data$legendary)
Pokemon_data$generation <- factor(Pokemon_data$generation)
```

Let's perform an initial split of the data, and stratify by the outcome variable.

```r
set.seed(8488)

Pokemon_split <- initial_split(Pokemon_data, prop=0.70, strata=type_1)

Pokemon_train <- training(Pokemon_split)
Pokemon_test <- testing(Pokemon_split)
```

For splitting the data, I chose a proportion of 0.70 because it allows for more training data, while retaining enough data to be tested since there is a limited amount of observations. The training data has 559 observations while the testing data has 241 observations.

Next, let's use v-fold cross-validation on the training set, using 5 folds. We'll stratify the folds by `type_1` as well.

```r
Pokemon_folds <- vfold_cv(Pokemon_train, v = 5, strata=type_1)
```

In this case, stratifying the folds is useful to ensure that each fold is representative of all strata of the data.

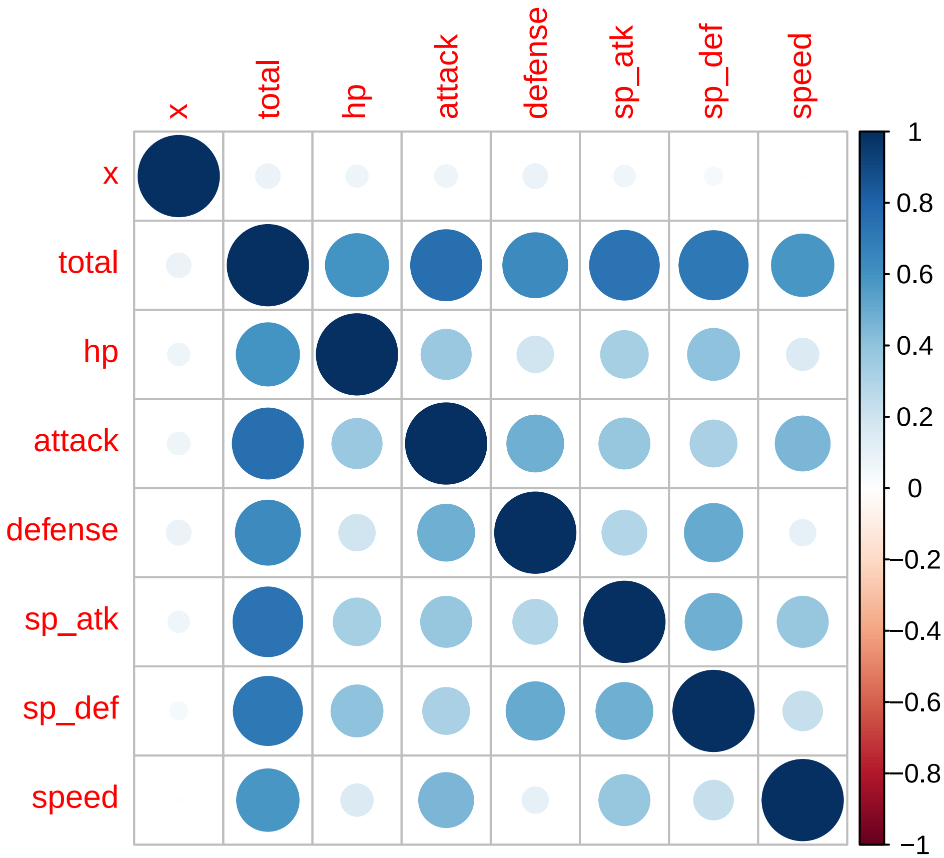
Let's set up a recipe to predict `type_1` with `legendary`, `generation`, `sp_atk`, `attack`, `speed`, `defense`, `hp`, and `sp_def`. We'll also dummy-code `legendary` and `generation`, as well as center and scale all predictors.

```r
Pokemon_recipe <- recipe(type_1 ~ legendary + generation + sp_atk + attack +
                           speed + defense + hp + sp_def, data=Pokemon_train) %>%
  step_dummy(c(legendary, generation)) %>%
  step_normalize(all_predictors())

#Pokemon_recipe %>% prep() %>% juice()
```

Exercise 2

Let's create a correlation matrix of the training set, using the `corrplot` package.

```r
Pokemon_train %>%
  select(where(is.numeric)) %>%
  cor() %>%
  corrplot()
```



All predictors that we have kept in our recipe seem to have rather strong positive relationships with each other. This seems to make sense overall because if a Pokemon has a strong defense, attack, or speed, for example, the rest its attributes will be strong as well since it will be a more powerful Pokemon.

Exercise 3

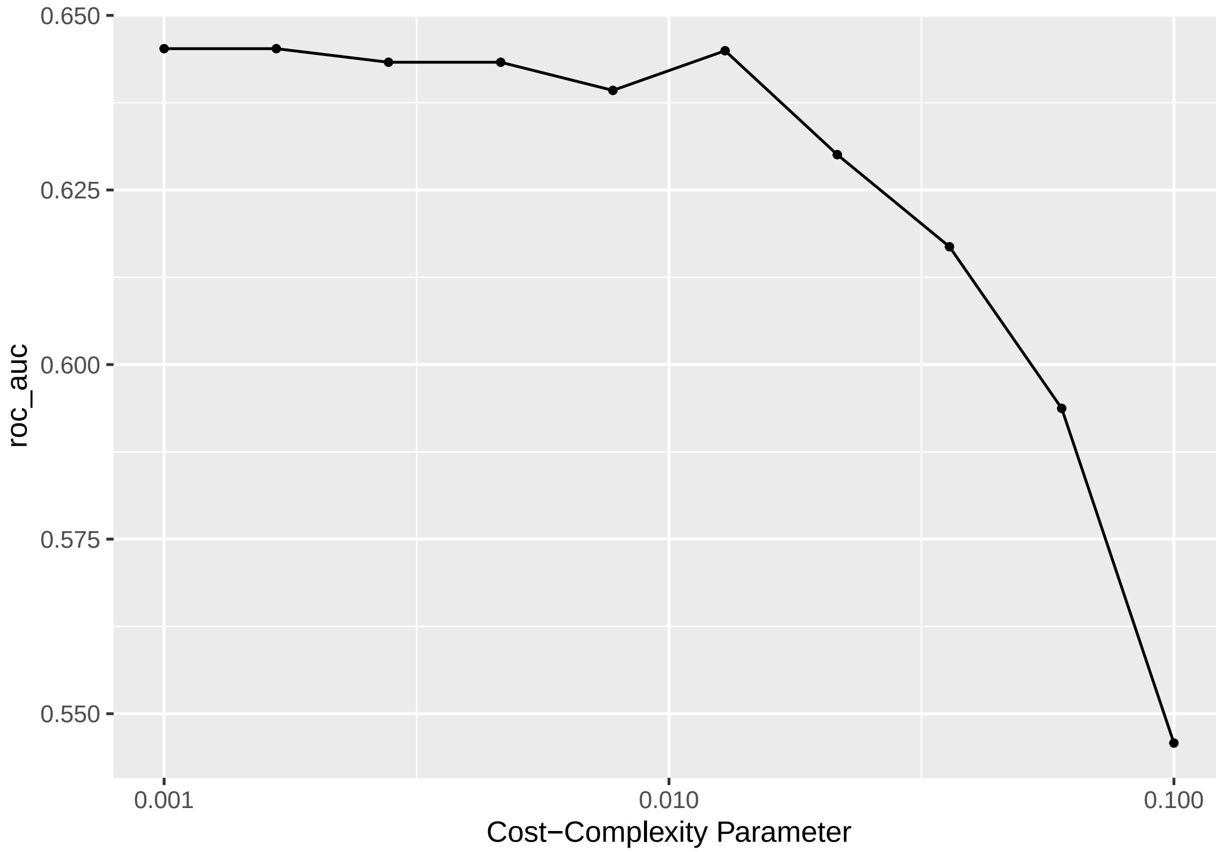
First, we'll set up a decision tree model and workflow. We'll tune the `cost_complexity` hyper parameter.

```
tree_spec <- decision_tree() %>%
  set_engine("rpart")

class_tree_spec <- tree_spec %>%
  set_mode("classification")

class_tree_wf <- workflow() %>%
  add_model(class_tree_spec %>% set_args(cost_complexity = tune())) %>%
  add_formula(type_1 ~ legendary + generation + sp_atk + attack +
                speed + defense + hp + sp_def)
```

Then, we'll use the same levels we used in Lab 7 – that is, `range = c(-3, -1)`. We'll also specify that the metric we want to optimize is `roc_auc`.

```
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(yardstick::roc_auc)
)

autoplot(tune_res)
```



As we can see in the plot above, the ROC AUC value of the single decision tree is consistent for cost-complexity values between 0.001 and 0.010, but decreases significantly between 0.010 and 0.100. We can therefore say that a single decision tree performs better with a smaller complexity penalty, and performs poorer with a larger complexity penalty.

Exercise 4

Let's find the `roc_auc` of our best-performing pruned decision tree on the folds. We'll use `collect_metrics()` and `arrange()`.

```
best_pruned_tree <- arrange(collect_metrics(tune_res), desc(mean))
best_pruned_tree
```

```
## # 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  0.645     5  0.0239 Preprocessor1_Model01
```

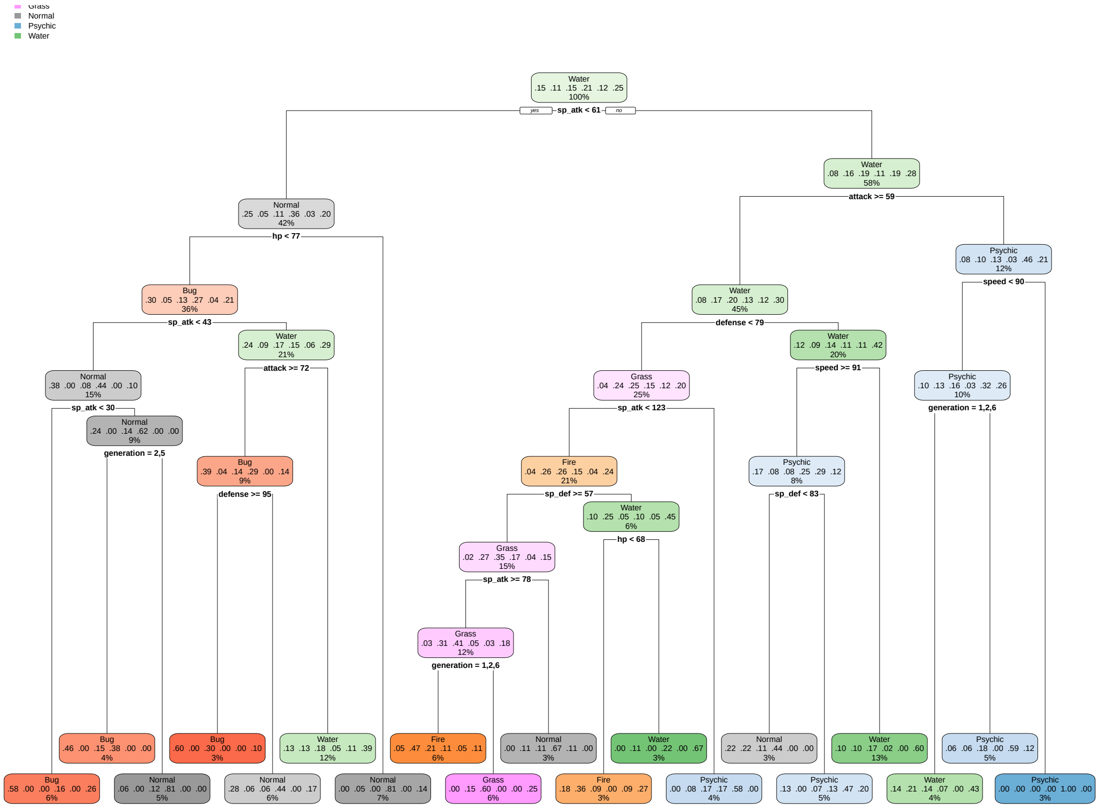
```
##  2         0.00167 roc_auc hand_till  0.645     5  0.0239 Preprocessor1_Model02
##  3         0.0129  roc_auc hand_till  0.645     5  0.0194 Preprocessor1_Model06
##  4         0.00278 roc_auc hand_till  0.643     5  0.0229 Preprocessor1_Model03
##  5         0.00464 roc_auc hand_till  0.643     5  0.0229 Preprocessor1_Model04
##  6         0.00774 roc_auc hand_till  0.639     5  0.0254 Preprocessor1_Model05
##  7         0.0215  roc_auc hand_till  0.630     5  0.0247 Preprocessor1_Model07
##  8         0.0359  roc_auc hand_till  0.617     5  0.0119 Preprocessor1_Model08
##  9         0.0599  roc_auc hand_till  0.594     5  0.0141 Preprocessor1_Model09
## 10         0.1     roc_auc hand_till  0.546     5  0.0203 Preprocessor1_Model10
```

As we can see at the top of the tibble above, the ROC AUC of our best performing model pruned decision tree on the folds is 0.6452386.

Exercise 5

Using `rpart.plot`, we'll fit and visualize our best-performing pruned decision tree with the training set.

```r
best_complexity <- select_best(tune_res)

class_tree_final <- finalize_workflow(class_tree_wf, best_complexity)

class_tree_final_fit <- fit(class_tree_final, data = Pokemon_train)

class_tree_final_fit %>%
  extract_fit_engine() %>%
  rpart.plot(roundint=FALSE)
```



Exercise 5

Now we'll set up a random forest model and workflow. Use the `ranger` engine and set `importance = "impurity"`. Let's also tune `mtry`, `trees`, and `min_n`.

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

rand_tree_wf <- workflow() %>%
  add_model(rf_spec) %>%
  add_formula(type_1 ~ legendary + generation + sp_atk + attack +
                speed + defense + hp + sp_def)
```

In the above model, we are tuning three different hyperparameters. These are:
- `mtry` which represents the amount of predictors that will be sampled randomly during the creation of the models, - `trees` which represents the amount of trees present in the random forest model, - `min_n` which represents the minimum amount of data values required to be in a tree node in order for it to be split further down the tree.

We'll then create a regular grid with 8 levels each. We'll choose plausible ranges for each hyperparameter.

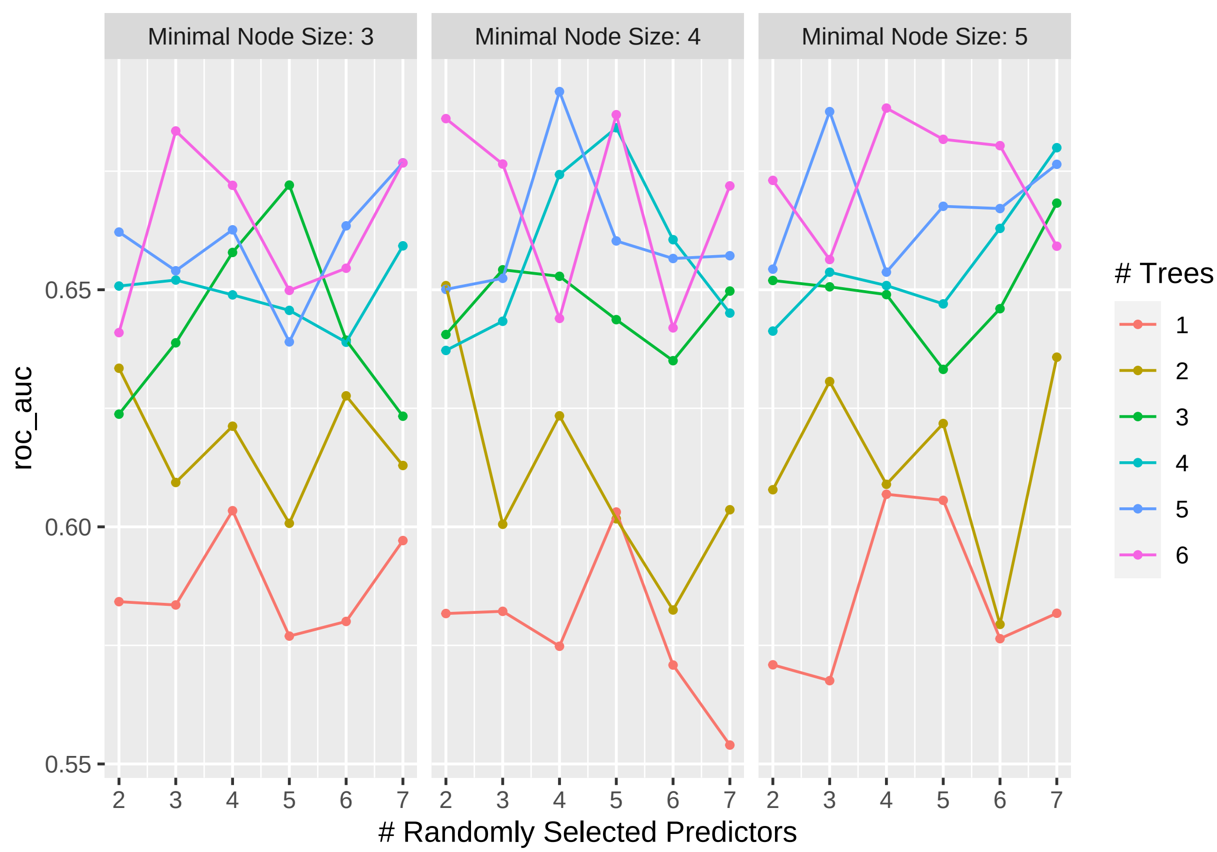
```
forest_param_grid <- grid_regular(mtry(range = c(2, 7)), trees(range = c(1, 6)),
                                  min_n(range = c(3,5)), levels = 8)
```

In this case, we cannot have our hyperparameter `mtry` be smaller than 1 or larger than 8. Having it be smaller than 1 would mean that we are not using any predictors from our recipe, while having it be larger than 8 would mean that we are trying to use more predictors than are available to us in our recipe. A model with `mtry = 8` would be a Bagged model, where predictors would be sampled with replacement and thus used more than once. This reduces variance within the results but increases bias in the sampling and overall.

Exercise 6

Next, let's specify `roc_auc` as a metric, then tune the model and print an `autoplot()` of the results.

```
forest_tune_res <- tune_grid(
  rand_tree_wf,
  resamples = Pokemon_folds,
  grid = forest_param_grid,
  metrics = metric_set(yardstick::roc_auc)
)

autoplot(forest_tune_res)
```



While the plots seem to fluctuate quite a bit, we can pull some conclusions. We can observe that having 5 as the value of `trees` leads to a higher ROC AUC, which means that more trees will lead to a better AUC. Additionally, a `mtry` value (random selection) of 4 predictors tends to output the higher ROC AUC. The optimal `min_n` value is 4, which can be observed in the middle plot with the highest plotted ROC AUC value. These all together yield the best performance.

Exercise 7

Let's once again find the `roc_auc` of our best-performing random forest tree model on the folds. We'll use `collect_metrics()` and `arrange()`.

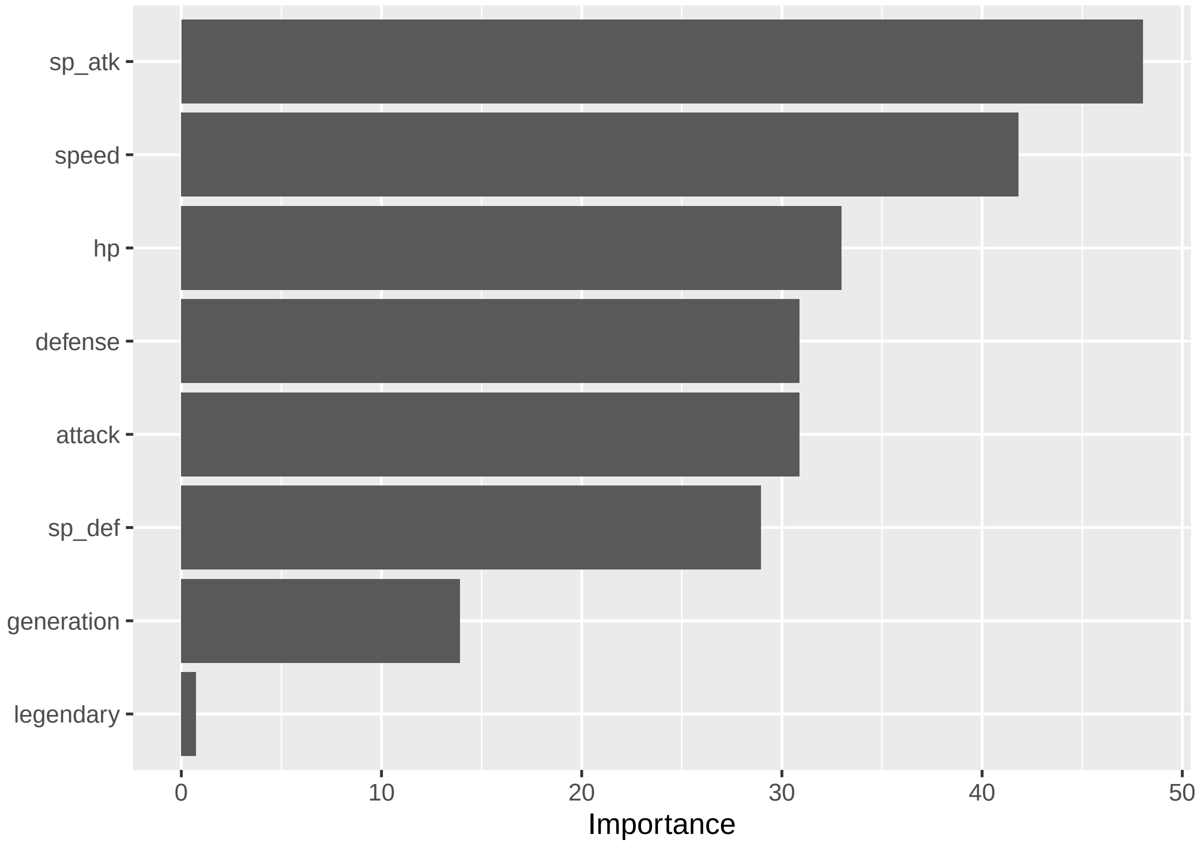
```
best_rd_tree <- arrange(collect_metrics(forest_tune_res), desc(mean))
head(best_rd_tree)
```

```
## # A tibble: 6 x 9
##    mtry trees min_n .metric .estimator  mean     n std_err .config
##   <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1     4     5     4 roc_auc hand_till  0.692     5  0.0201 Preprocessor1_Model0~
## 2     4     6     5 roc_auc hand_till  0.688     5  0.0136 Preprocessor1_Model1~
## 3     3     5     5 roc_auc hand_till  0.688     5  0.0184 Preprocessor1_Model0~
## 4     5     6     4 roc_auc hand_till  0.687     5  0.0116 Preprocessor1_Model0~
## 5     2     6     4 roc_auc hand_till  0.686     5  0.0235 Preprocessor1_Model0~
## 6     5     4     4 roc_auc hand_till  0.684     5  0.0118 Preprocessor1_Model0~
```

As we can see at the top of the tibble above, the ROC AUC of our best-performing random tree model on the folds is 0.6917831.

Exercise 8

```
best_forest_complexity <- select_best(forest_tune_res)

rand_tree_final <- finalize_workflow(rand_tree_wf, best_forest_complexity)

rand_tree_final_fit <- fit(rand_tree_final, data = Pokemon_train)

rand_tree_final_fit %>%
  extract_fit_parsnip() %>%
  vip()
```



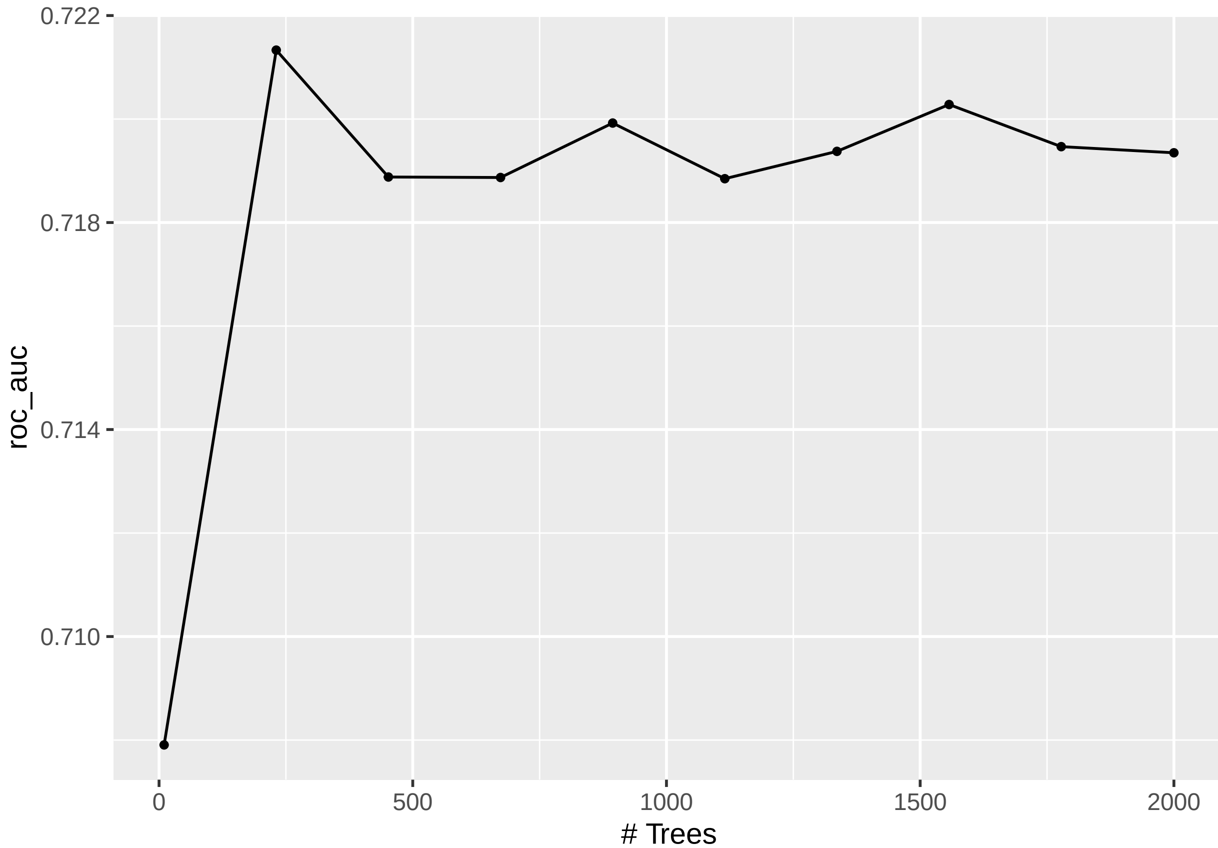
From this plot above, we can see that the variable `sp_atl` was the most useful with `speed` and `defense` close behind. On the other end, the least useful variable was by far `legendary`. This makes sense because these most useful variables tend to vary hugely based on the type of Pokemon, with Fire types, for example, having much more attack strength, etc. On the other hand, the legendary status of a Pokemon has very little to do with its type, and will therefore not effect or predict it well.

Exercise 9

Finally, let's set up a boosted tree model and workflow. We'll use the `xgboost` engine, and tune `trees`. We'll then create a regular grid with 10 levels; letting `trees` range from 10 to 2000. We'll also specify `roc_auc` and again print an `autoplot()` of the results.

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

boost_tree_wf <- workflow() %>%
  add_model(boost_spec) %>%
  add_formula(type_1 ~ legendary + generation + sp_atk + attack +
                speed + defense + hp + sp_def)

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

boost_tune_res <- tune_grid(
  boost_tree_wf,
  resamples = Pokemon_folds,
```

```r
  grid = boost_param_grid,
  metrics = metric_set(yardstick::roc_auc)
)

autoplot(boost_tune_res)
```



As we can see in the plot above, the ROC AUC rapidly and strongly increases at first when the number of Trees is low, which is between 10 and 250 Trees. After that, the ROC AUC seems to decrease a bit and then stays relatively consistent for all other number of Trees until 2000. We cna determine from this graph that the number of Trees that provides the best `roc_auc` is about 230 Trees, just under the 250 mark.

One last time, let's find the `roc_auc` of our best-performing boosted tree model on the folds. We'll use `collect_metrics()` and `arrange()`.

```r
best_boost_tree <- arrange(collect_metrics(boost_tune_res), desc(mean))
head(best_boost_tree)
```

```
## # A tibble: 6 x 7
##   trees .metric .estimator  mean     n std_err .config
##   <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1   231 roc_auc hand_till  0.721     5 0.0119  Preprocessor1_Model02
## 2  1557 roc_auc hand_till  0.720     5 0.00796 Preprocessor1_Model08
## 3   894 roc_auc hand_till  0.720     5 0.00947 Preprocessor1_Model05
## 4  1778 roc_auc hand_till  0.719     5 0.00781 Preprocessor1_Model09
## 5  1336 roc_auc hand_till  0.719     5 0.00808 Preprocessor1_Model07
## 6  2000 roc_auc hand_till  0.719     5 0.00763 Preprocessor1_Model10
```

As we can see at the top of the tibble above, the ROC AUC of our best performing-model boosted tree model on the folds is 0.7213321 with 231 Trees.

Exercise 10

Let's display a table of the three ROC AUC values for our best-performing pruned tree, random forest, and boosted tree models.

```
best_roc_auc_trees <- c(best_pruned_tree$mean[1],
                        best_rd_tree$mean[1],
                        best_boost_tree$mean[1])

best_roc_auc_names <- c("Pruned Tree", "Random Forest", "Boosted Tree")

best_roc_auc <- tibble(Model = best_roc_auc_names,
                       ROC_AUC = best_roc_auc_trees)

best_roc_auc <- best_roc_auc %>%
  arrange(-best_roc_auc_trees)

best_roc_auc
```

```
## # A tibble: 3 x 2
##   Model         ROC_AUC
##   <chr>           <dbl>
## 1 Boosted Tree    0.721
## 2 Random Forest   0.692
## 3 Pruned Tree     0.645
```

We can see in the tibble above that the Boosted tree model performed the best with a ROC AUC value of 0.7213321.

Now, we'll select the best of the three and use `select_best()`, `finalize_workflow()`, and `fit()` to fit it to the *testing* set.

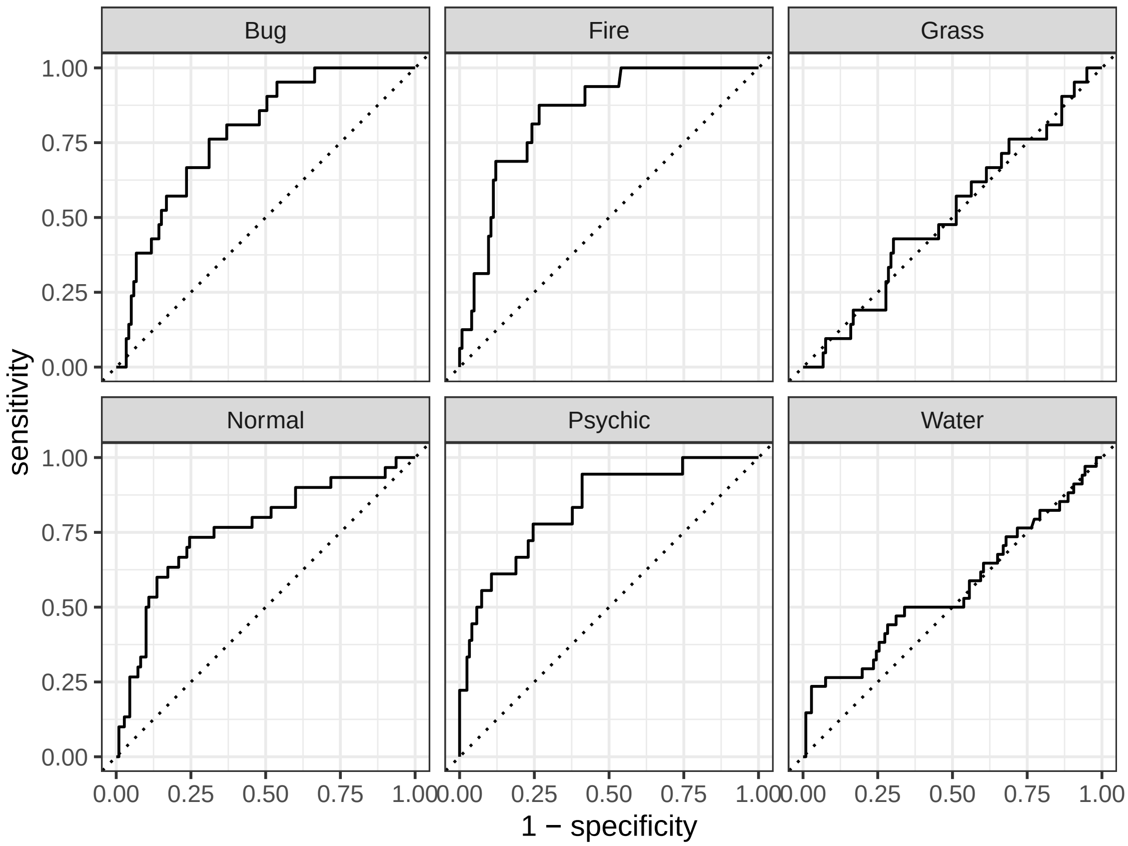
```
best_boost_complexity <- select_best(boost_tune_res)

boost_tree_final <- finalize_workflow(boost_tree_wf, best_boost_complexity)

boost_tree_final_fit <- fit(boost_tree_final, data = Pokemon_train)

roc <- augment(boost_tree_final_fit, new_data = Pokemon_test, type = 'prob')

roc %>%
  roc_auc(type_1, c(.pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal,
                                      .pred_Water, .pred_Psychic))
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.644
```

The AUC value of our best-performing model on the testing set was 0.6441098.

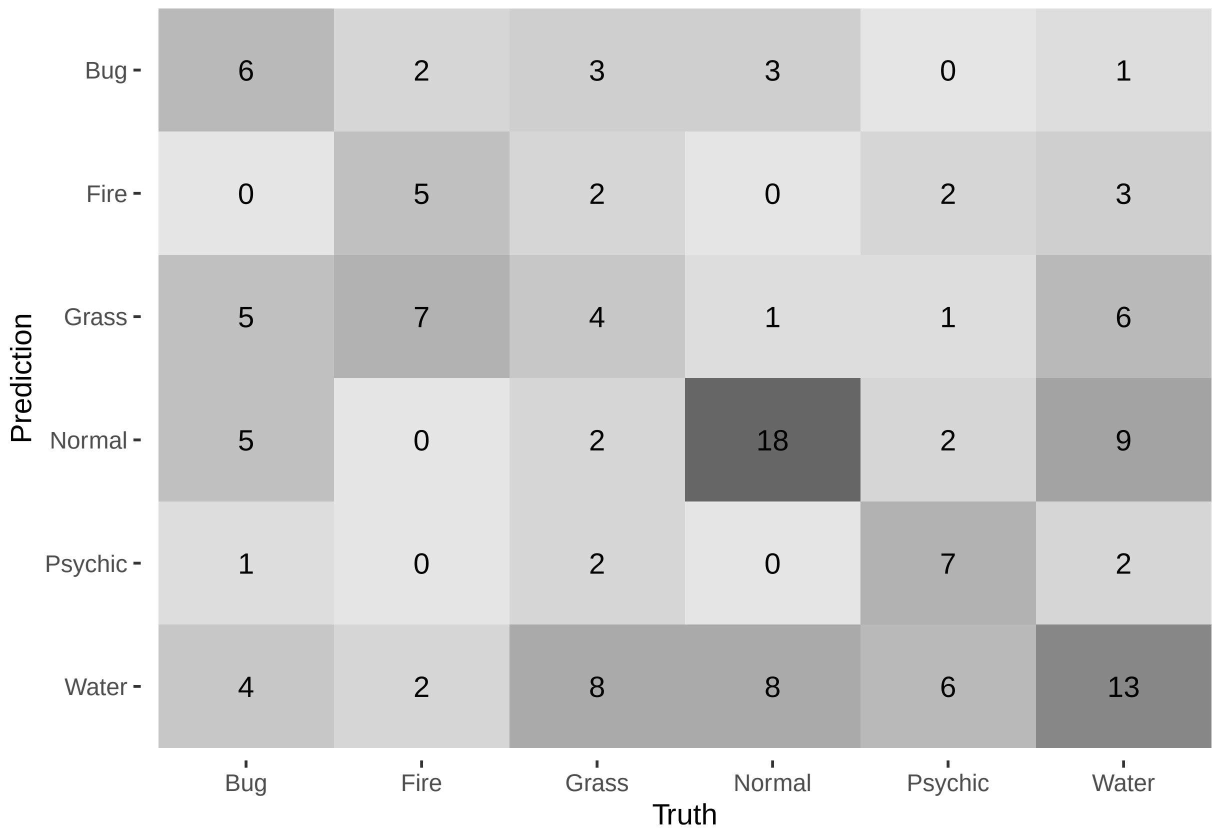
Now, we'll create plots of the different ROC curves, one per level of the outcome.

```
roc %>%
  roc_curve(type_1, c(.pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal,
                      .pred_Psychic, .pred_Water)) %>%
  autoplot()
```



Finally, we'll also make a heat map of the confusion matrix.

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



As can be seen in the ROC curves as well as the heatmap, our model was best at predicting Pokemon types Fire and Psychic, as well as fairly strong at predicting Bug and Normal types. However, our model was by far worst at predicting Grass and Water types of Pokemon.