Homework 6

PSTAT 131/231

Contents

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.

The Pokémon 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 (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.

```r
set.seed(185)
# load libraries
library(tidyverse)
library(tidymodels)
library(dplyr)
library(discrim)
library(glmnet)
library(janitor)
library(rpart.plot)
library(randomForest)
library(ranger)
library(vip)
library(xgboost)
library(ggfortify)
library(corrr)
library(corrplot)
```

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.

```
pokemon <- read.csv('pokemon.csv')
library(janitor)
pokemon_clean <- pokemon %>% clean_names()
head(pokemon_clean)
```

```
##   x                  name type_1 type_2 total hp attack defense sp_atk sp_def
## 1 1             Bulbasaur  Grass Poison   318 45     49      49     65     65
## 2 2               Ivysaur  Grass Poison   405 60     62      63     80     80
## 3 3              Venusaur  Grass Poison   525 80     82      83    100    100
## 4 3 VenusaurMega Venusaur  Grass Poison   625 80    100     123    122    120
## 5 4            Charmander   Fire          309 39     52      43     60     50
## 6 5            Charmeleon   Fire          405 58     64      58     80     65
##   speed generation legendary
## 1    45          1     False
## 2    60          1     False
## 3    80          1     False
## 4    80          1     False
## 5    65          1     False
## 6    80          1     False
```

```
types <- c("Bug", "Fire", "Grass", "Normal", "Water", "Psychic")
pokemon_filter <- pokemon_clean %>% filter(type_1 %in% types)
pokemon_filter$type_1 <-factor(pokemon_filter$type_1)
pokemon_filter$legendary <-factor(pokemon_filter$legendary)
pokemon_filter$generation <-factor(pokemon_filter$generation) # not in directions, but needed for step_
# initial split
pokemon_split <- initial_split(pokemon_filter, prop = 0.7,  strata = "type_1")
pokemon_train <- training(pokemon_split)
pokemon_test <- testing(pokemon_split)
# check number of observations
dim(pokemon_train)
```

```
## [1] 318  13
```

```
dim(pokemon_test)
```

```
## [1] 140  13
```

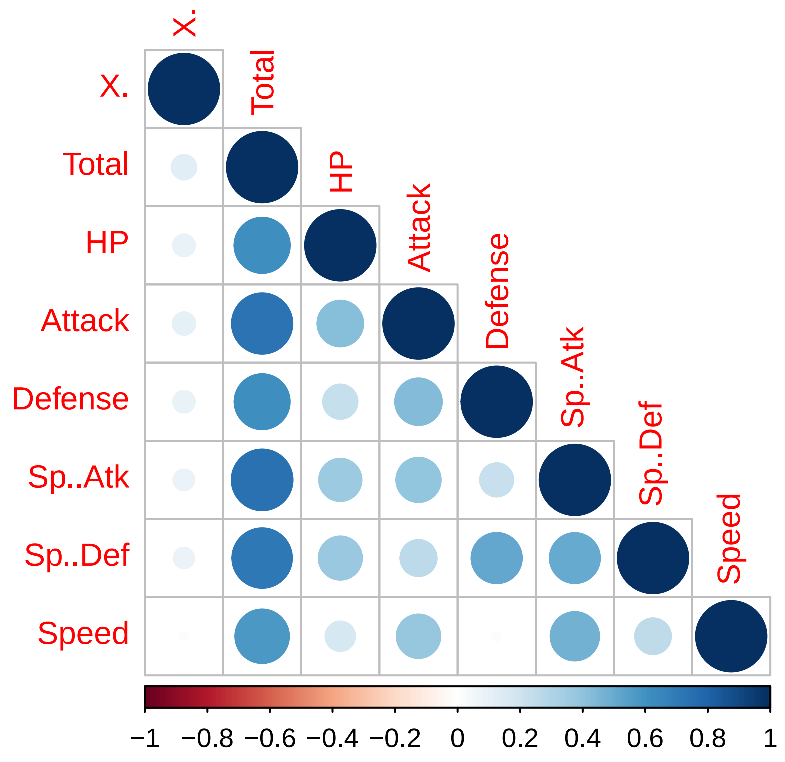
```
# v-fold cross-validation
pokemon_folds <- vfold_cv(pokemon_train, v = 5, strata = "type_1")
# recipe
pokemon_recipe <- recipe(type_1 ~ legendary + generation + sp_atk + attack + speed + defense + hp + sp_
                    step_dummy(legendary) %>%
                    step_dummy(generation) %>%
                    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?

```
pokemon %>%
  select(where(is.numeric)) %>%
  cor() %>%
  corrplot(type = "lower")
```



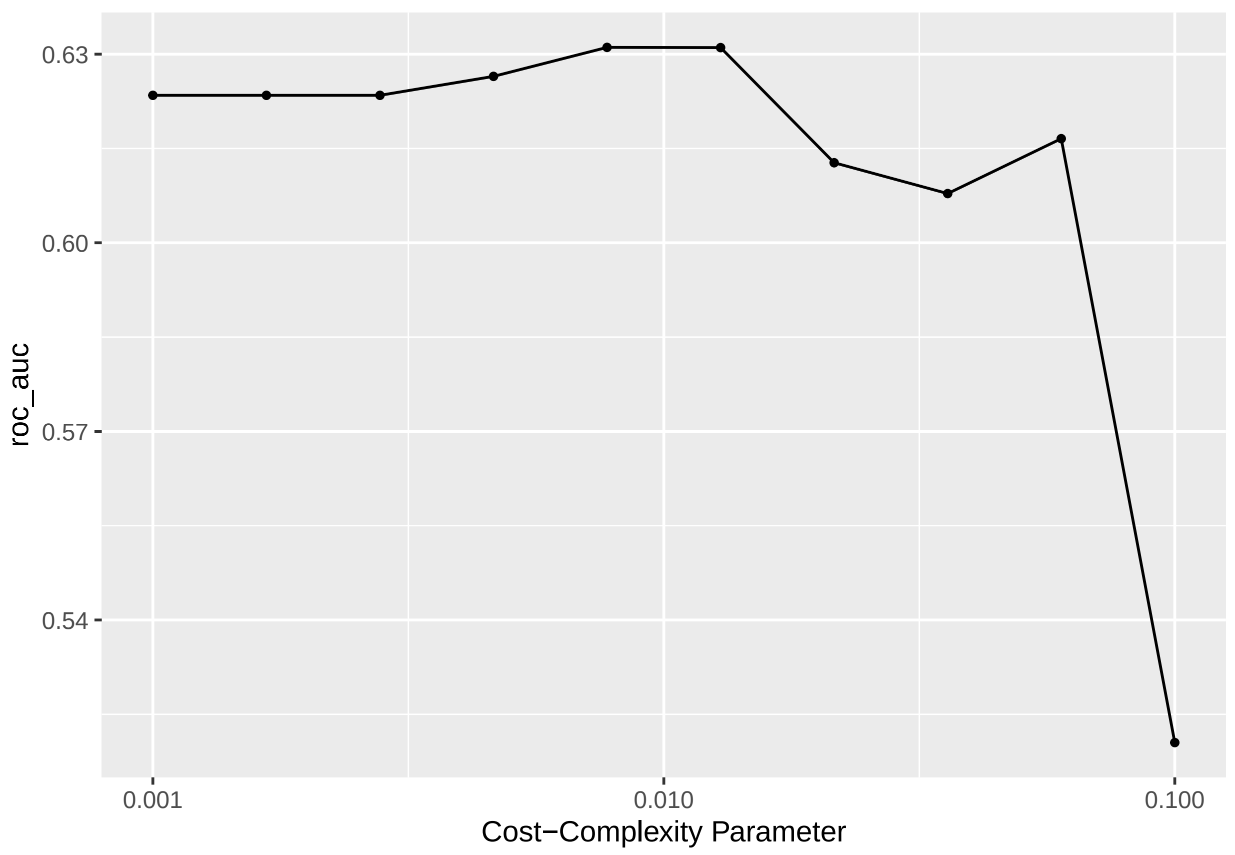
The number of a Pokemon has no relation to any of the attributes of the Pokemon. Speed is slightly positively correlated with attack and special attack. Special defense is slightly correlated with defense and special attack. Special attack is slightly correlated with attack. Defense is slightly correlated with attack. When looking at the influence of individual attributes on the total score, it seems to be that attack, special attack, and special defense are the most important. There are no negatively correlated variables.

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?

```r
tree_spec <- decision_tree() %>% set_engine("rpart") %>%
            set_mode("classification")
tree_workflow <- workflow() %>%
            add_recipe(pokemon_recipe)%>%
            add_model(tree_spec %>% set_args(cost_complexity = tune()))
param_grid <- grid_regular(cost_complexity(range = c(-3, -1)), levels = 10)
tune_res <- tune_grid(
  tree_workflow,
  resamples = pokemon_folds,
  grid = param_grid,
  metrics = metric_set(roc_auc)
)
autoplot(tune_res)
```



In general, as cost-complexity goes up the overall roc_auc of our model goes down. After that the model starts preforming worse and worse until it eventually reaches an roc_auc of 0.52. It seems that a single decision tree performs better with a smaller cost-complexity parameter, as the roc_auc seems to drop significantly when cost-complexity exceeds 0.01.

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

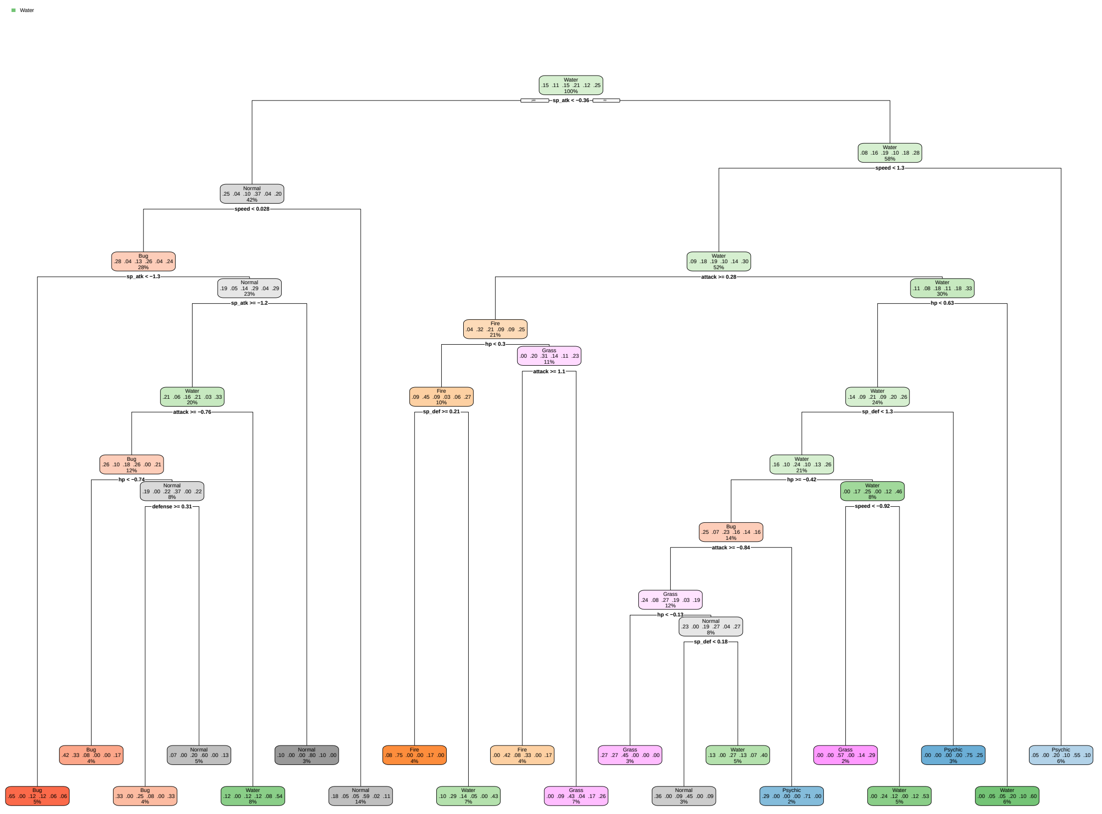
```
## # A tibble: 10 x 7
##    cost_complexity .metric .estimator  mean     n std_err .config
##              <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1         0.00774 roc_auc hand_till  0.631     5  0.0144 Preprocessor1_Model05
##  2         0.0129  roc_auc hand_till  0.631     5  0.0146 Preprocessor1_Model06
##  3         0.00464 roc_auc hand_till  0.626     5  0.0170 Preprocessor1_Model04
##  4         0.001   roc_auc hand_till  0.623     5  0.0176 Preprocessor1_Model01
##  5         0.00167 roc_auc hand_till  0.623     5  0.0176 Preprocessor1_Model02
##  6         0.00278 roc_auc hand_till  0.623     5  0.0176 Preprocessor1_Model03
##  7         0.0599  roc_auc hand_till  0.617     5  0.0126 Preprocessor1_Model09
##  8         0.0215  roc_auc hand_till  0.613     5  0.0148 Preprocessor1_Model07
##  9         0.0359  roc_auc hand_till  0.608     5  0.0131 Preprocessor1_Model08
## 10         0.1     roc_auc hand_till  0.520     5  0.0205 Preprocessor1_Model10
```

The roc_auc of the best-performing pruned decision tree was 0.631, which has a corresponding cost_complexity of 0.00774, 0.0129 (all had the same performance).

Exercise 5

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

```
best_performing <- select_best(tune_res)
tree_final <- finalize_workflow(tree_workflow, best_performing)
tree_final_fit <- fit(tree_final, data = pokemon_train)
tree_final_fit %>%
  extract_fit_engine() %>%
  rpart.plot(roundint=FALSE)
```



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?**

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

regular_grid <- grid_regular(mtry(range = c(1, 8)), trees(range = c(10, 1000)), min_n(range = c(1, 10))
```

mtry refers to the amount of predictors that will be randomly sampled for each split, trees refers to the amount of trees, and min_n refers to a node's minimum amount of data points required for it to split. mtry can't be greater than 8 because we only have 8 predictors, so we can't sample more than 8. mtry = 8 would represent a bagging model.

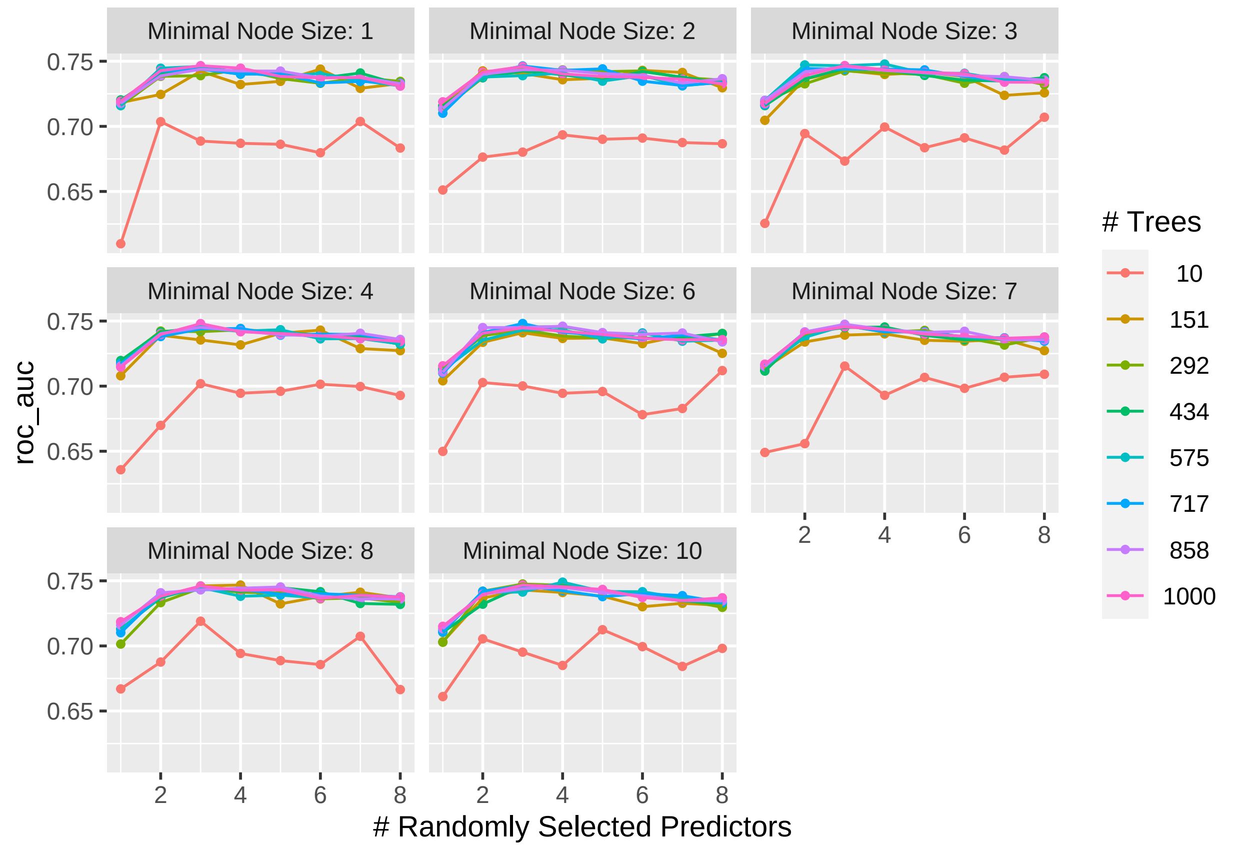
Exercise 6

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?

```
rf_res <- tune_grid(
  rf_workflow,
  resamples = pokemon_folds,
  grid = regular_grid,
  metrics = metric_set(roc_auc)
)
```

```
#save(rf_res, file = "C:/Users/wilson/Desktop/rf_res.rda")
load("C:/Users/wilson/Desktop/rf_res.rda")

autoplot(rf_res)
```



It seems that the number of trees does not seem to affect performance by that much, as long as it is above 100 trees. It also appears that for predictors and minimal node size, the ones in the middle of the ranges that I chose led to slightly better 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_res) %>% arrange(desc(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     4   575    10 roc_auc hand_till  0.749     5  0.0126 Preprocessor1_Model~
##  2     3   717     6 roc_auc hand_till  0.748     5  0.0126 Preprocessor1_Model~
##  3     3  1000     4 roc_auc hand_till  0.748     5  0.0157 Preprocessor1_Model~
##  4     4   575     3 roc_auc hand_till  0.748     5  0.0144 Preprocessor1_Model~
##  5     3   292    10 roc_auc hand_till  0.748     5  0.0118 Preprocessor1_Model~
##  6     3   858     7 roc_auc hand_till  0.748     5  0.0136 Preprocessor1_Model~
##  7     2   575     3 roc_auc hand_till  0.747     5  0.0164 Preprocessor1_Model~
##  8     3   717     7 roc_auc hand_till  0.747     5  0.0132 Preprocessor1_Model~
##  9     4   151     8 roc_auc hand_till  0.747     5  0.0144 Preprocessor1_Model~
## 10     3  1000    10 roc_auc hand_till  0.747     5  0.0125 Preprocessor1_Model~
## # ... with 502 more rows
```

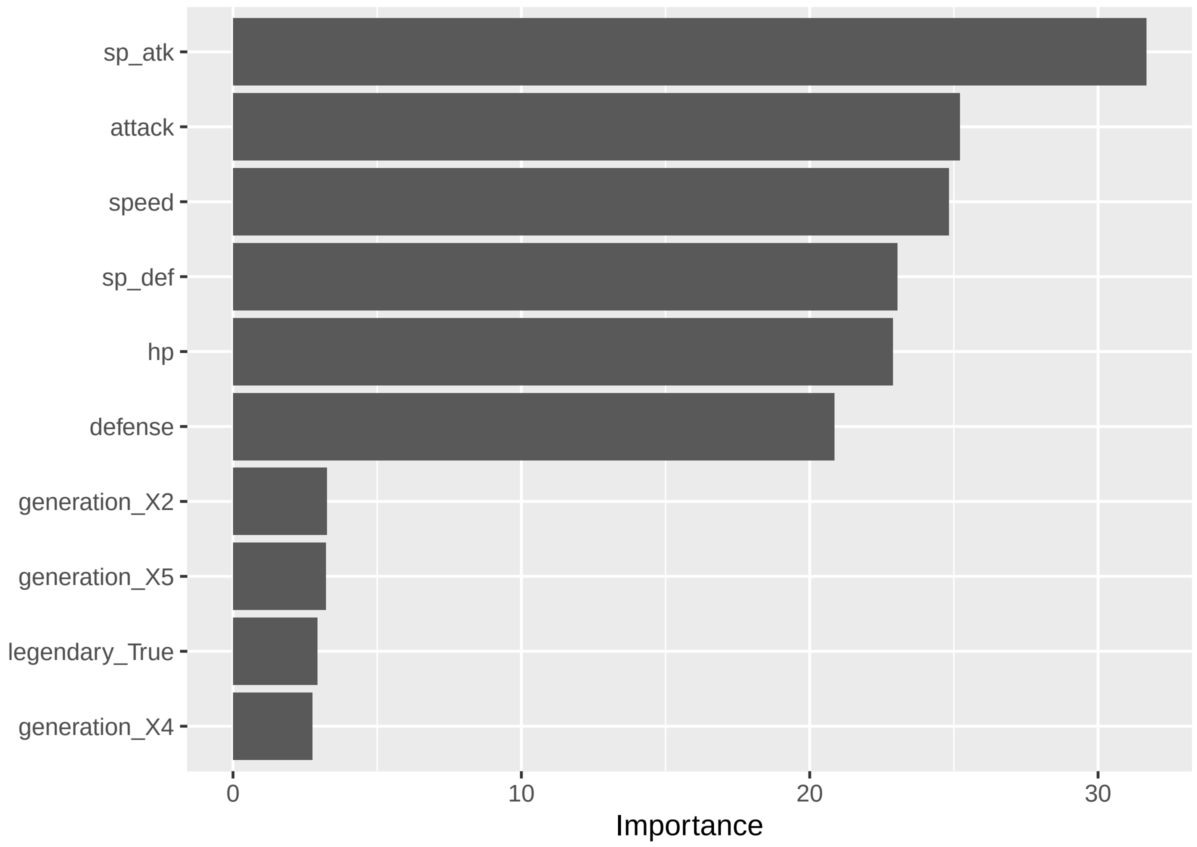
The best-performing random forest model has a roc_auc of 0.749.

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?

```r
best_rf <- select_best(rf_res, metric='roc_auc')
rf_final <- finalize_workflow(rf_workflow, best_rf)
rf_final_fit <- fit(rf_final, data=pokemon_train)
vip(rf_final_fit %>% extract_fit_parsnip())
```



The variables that is the most useful is by far special attack. Attack and speed are similar important. That is what I expected because in the games certain pokemon are known for their special attack stats so it would be a very useful predictor to be identifying the type of a pokemon based on that stat profile.

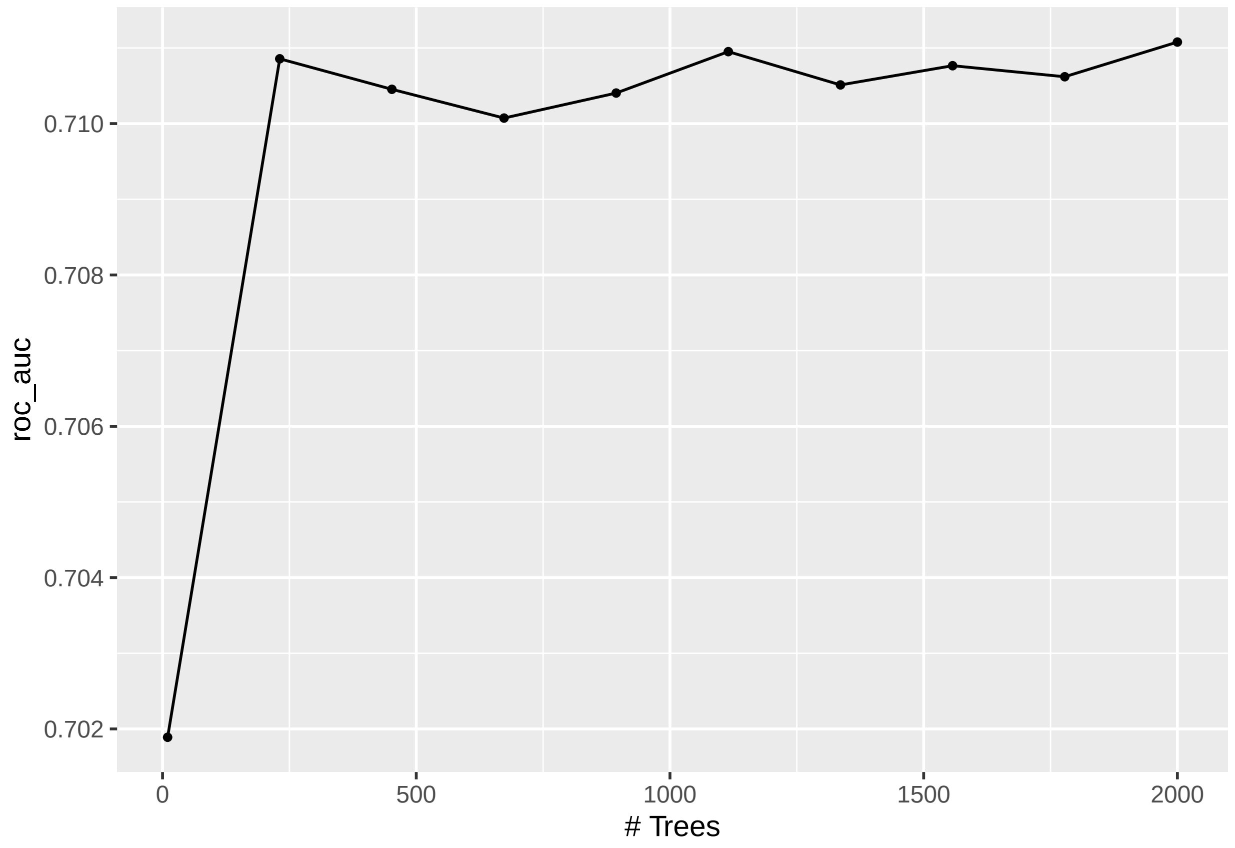
Exercise 9

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()`.*

```r
boost_spec <- boost_tree(trees=tune()) %>%
            set_engine("xgboost") %>%
            set_mode("classification")
boost_workflow <- workflow() %>%
                  add_model(boost_spec) %>%
                  add_recipe(pokemon_recipe)

boost_grid <- grid_regular(trees(range=c(10,2000)), levels=10)
boost_res <- tune_grid(
  boost_workflow,
  resamples = pokemon_folds,
  grid=boost_grid,
  metrics = metric_set(roc_auc)
)
autoplot(boost_res)
```



```r
collect_metrics(boost_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  2000 roc_auc hand_till  0.711     5  0.0163 Preprocessor1_Model10
##  2  1115 roc_auc hand_till  0.711     5  0.0160 Preprocessor1_Model06
##  3   231 roc_auc hand_till  0.711     5  0.0148 Preprocessor1_Model02
##  4  1557 roc_auc hand_till  0.711     5  0.0161 Preprocessor1_Model08
## 5  1778 roc_auc hand_till  0.711     5  0.0164 Preprocessor1_Model09
## 6  1336 roc_auc hand_till  0.711     5  0.0164 Preprocessor1_Model07
## 7   452 roc_auc hand_till  0.710     5  0.0152 Preprocessor1_Model03
## 8   894 roc_auc hand_till  0.710     5  0.0159 Preprocessor1_Model05
## 9   673 roc_auc hand_till  0.710     5  0.0152 Preprocessor1_Model04
## 10    10 roc_auc hand_till  0.702     5  0.0177 Preprocessor1_Model01
```

```
best_boosted <-  select_best(boost_res, metric='roc_auc')
```

We observe that there is an initial jump of roc_auc from the 0-300 tree range, after which we essentially get the same or even potentially worse roc_auc from 300-2000 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?

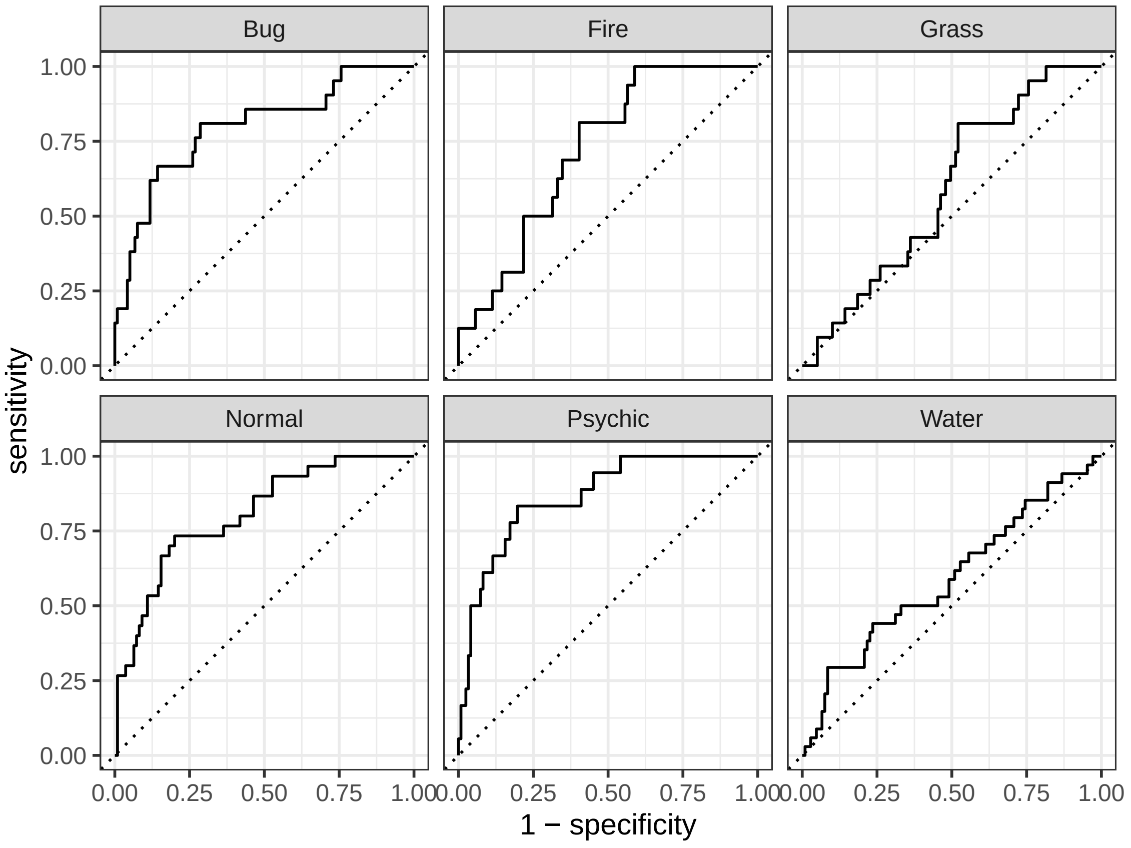
```
boost <- (collect_metrics(boost_res) %>% arrange(desc(mean)))[1,c('.metric','mean')]
rf <- (collect_metrics(rf_res) %>% arrange(desc(mean)))[1,c('.metric','mean')]
decision <- (collect_metrics(tune_res) %>% arrange(desc(mean)))[1,c('.metric','mean')]
table <- rbind(decision, rf, boost)
table %>% add_column(model = c("boosted tree", "random forest", "decision tree"), .before = ".metric")
```

```
## # A tibble: 3 x 3
##   model         .metric  mean
##   <chr>         <chr>   <dbl>
## 1 boosted tree  roc_auc 0.631
## 2 random forest roc_auc 0.749
## 3 decision tree roc_auc 0.711
```

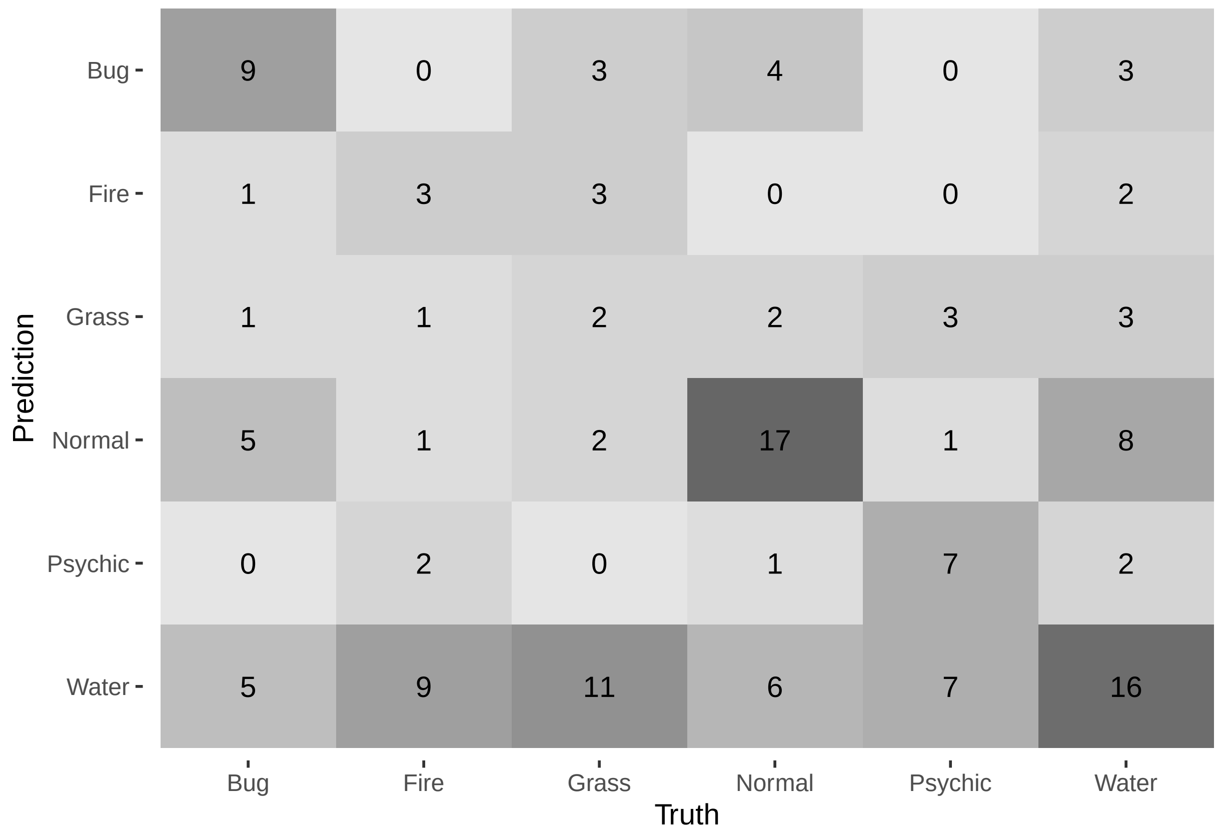
```
best_rf <- select_best(rf_res, metric='roc_auc')
rf_final <- finalize_workflow(rf_workflow, best_rf)
rf_final_fit <- fit(rf_final, data=pokemon_train)
# auc value
roc_auc(augment(rf_final_fit, new_data = pokemon_test), type_1, .pred_Bug, .pred_Fire, .pred_Grass, .pr
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.727
```

```
# plot
augment(rf_final_fit, new_data = pokemon_test) %>%
  roc_curve(type_1, .pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal, .pred_Psychic, .pred_Water) %>%
  autoplot()
```



```r
# confusion matrix heat map
augment(rf_final_fit, new_data = pokemon_test) %>%
  conf_mat(truth = type_1, estimate = .pred_class)  %>% autoplot(type = "heatmap")
```



Prediction \ Truth	Bug	Fire	Grass	Normal	Psychic	Water
Bug	9	0	3	4	0	3
Fire	1	3	3	0	0	2
Grass	1	1	2	2	3	3
Normal	5	1	2	17	1	8
Psychic	0	2	0	1	7	2
Water	5	9	11	6	7	16

It seems that, once again, our model was best at predicting normal and water types. It is the worst at predicting grass and fire.