Homework 6

Yixin Wang

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.

```
library(tidymodels)
library(tidyverse)
library(ISLR)
library(corrr)
library(janitor)
library(rpart.plot)
library(vip)
library(janitor)
library(randomForest)
library(xgboost)
library(corrplot)
set.seed(3435)
```

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`:

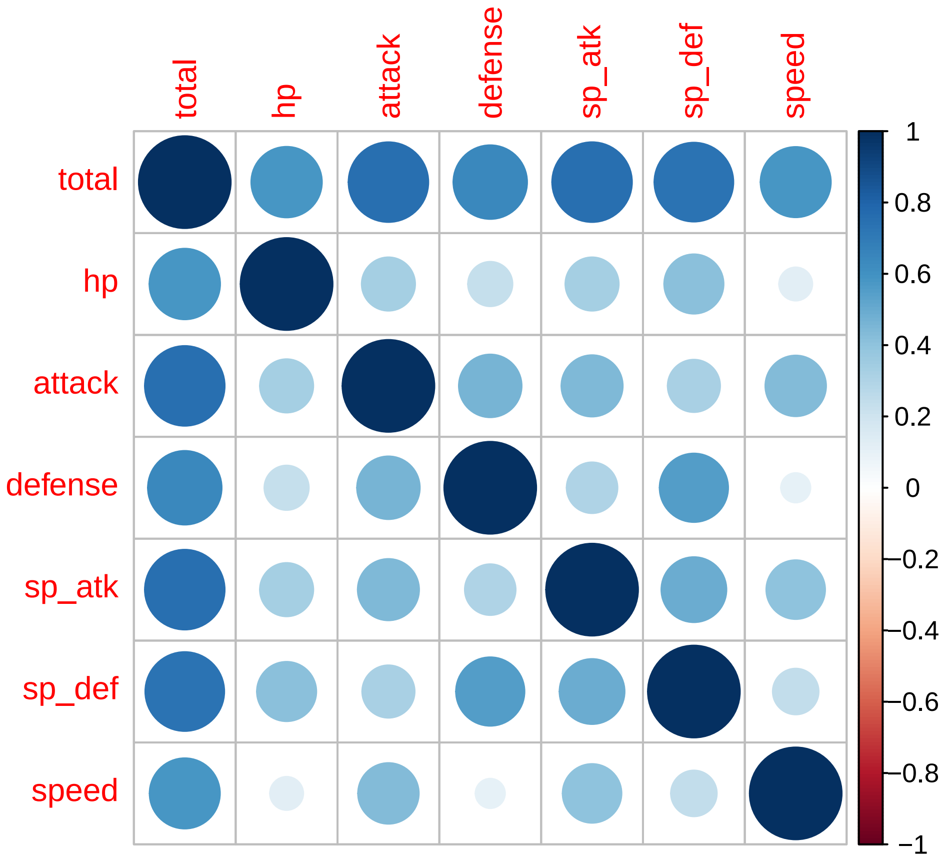
- Dummy-code `legendary` and `generation`;
- Center and scale all predictors.

```
pokemon <- read.csv("Pokemon.csv")
pokemon <- as_tibble(pokemon)
pokemon <- clean_names(pokemon)
pokemon <- pokemon[pokemon$type_1 %in% c("Bug","Fire","Grass","Normal","Water","Psychic"), ]
pokemon$type_1 <- as.factor(pokemon$type_1)
pokemon$legendary <- as.factor(pokemon$legendary)
pokemon_split <- initial_split(pokemon, prop = 0.7, strata = type_1)
pokemon_train <- training(pokemon_split)
pokemon_test <- testing(pokemon_split)
pokemon_fold <- vfold_cv(pokemon_train, v = 5, strata = type_1)
pokemon_recipe <- recipe(type_1 ~ legendary + generation + sp_atk + attack + speed + defense + hp + sp_d
  step_dummy(all_nominal_predictors()) %>%
  step_scale(all_predictors()) %>%
  step_center(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).*

```
pokemon_corr <- select_if(pokemon, is.numeric) %>%
  select(-c(x,generation)) %>%
  cor()
corrplot(pokemon_corr)
```



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

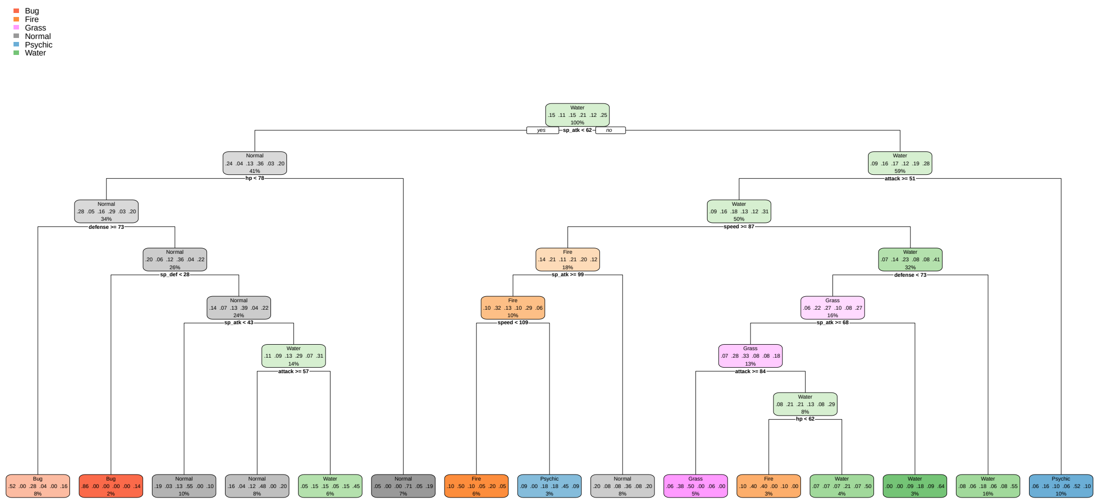
I exclude the x and generation. There is positive correlation between total and other variables. Other than the total variables, there are positive correlation between `defense` and `sp_def`, `attack` and `sp_atk`.

Exercise 3

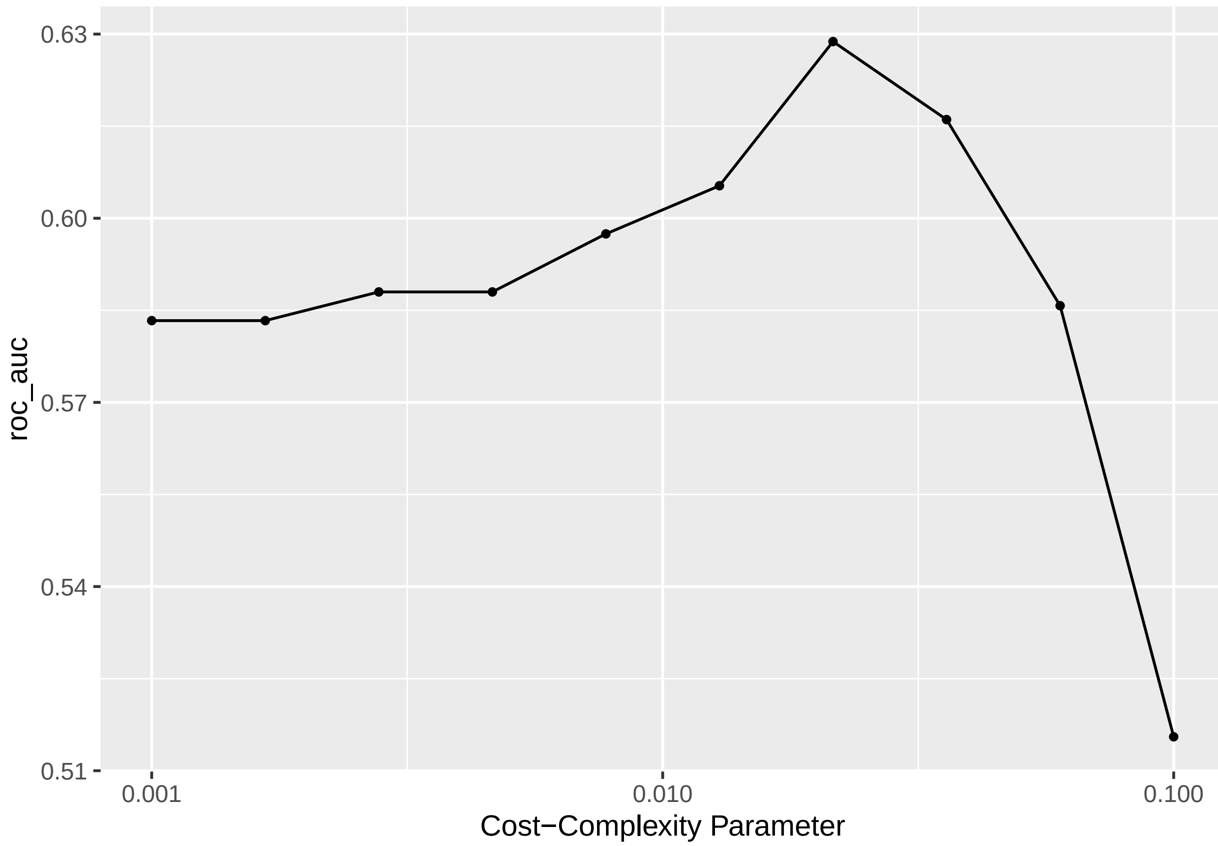
First, set up a decision tree model and workflow. Tune the `cost_complexity` hyperparameter. Use the same levels we used in Lab 7 – that is, `range = c(-3, -1)`. Specify that the metric we want to optimize is `roc_auc`.

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

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

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

class_tree_fit <- class_tree_spec %>%
  fit(type_1 ~ legendary + generation + sp_atk + attack + speed + defense + hp + sp_def, data = pokemon_

class_tree_fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```




```r
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)

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

tune_res <- tune_grid(
  class_tree_wf,
  resamples = pokemon_fold,
  grid = param_grid,
  metrics = metric_set(roc_auc)
)

autoplot(tune_res)
```



From the graph, 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()`*.*

```
arrange(collect_metrics(tune_res), desc(mean))
```

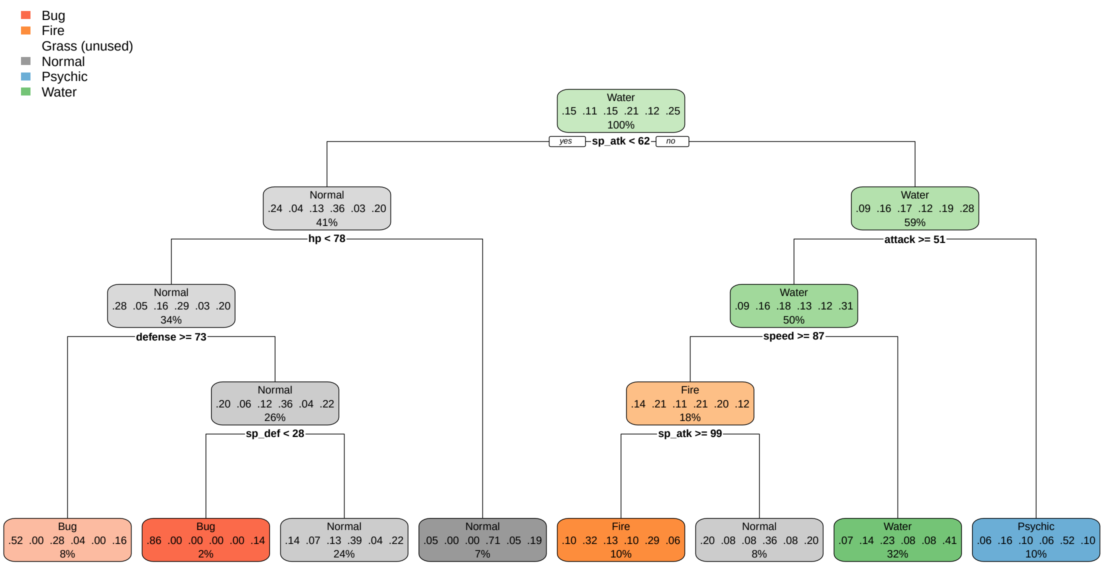
```
## # A tibble: 10 x 7
##    cost_complexity .metric .estimator  mean     n std_err .config
##              <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1         0.0215  roc_auc hand_till  0.629     5  0.0191 Preprocessor1_Model07
##  2         0.0359  roc_auc hand_till  0.616     5  0.0202 Preprocessor1_Model08
##  3         0.0129  roc_auc hand_till  0.605     5  0.0177 Preprocessor1_Model06
##  4         0.00774 roc_auc hand_till  0.597     5  0.0187 Preprocessor1_Model05
##  5         0.00278 roc_auc hand_till  0.588     5  0.0173 Preprocessor1_Model03
##  6         0.00464 roc_auc hand_till  0.588     5  0.0173 Preprocessor1_Model04
##  7         0.0599  roc_auc hand_till  0.586     5  0.0233 Preprocessor1_Model09
##  8         0.001   roc_auc hand_till  0.583     5  0.0157 Preprocessor1_Model01
##  9         0.00167 roc_auc hand_till  0.583     5  0.0157 Preprocessor1_Model02
## 10         0.1     roc_auc hand_till  0.516     5  0.0155 Preprocessor1_Model10
```

The best performing `roc_auc` is 0.6424392 and `cost_complexity` is approximately 0.03594.

Exercise 5

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

```
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()
```



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?**

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

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

rand_tree_grid <- grid_regular(mtry(c(1, 8)), trees(c(10, 100)), min_n(c(1, 8)), levels = 8)
```

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

`mtry` represents the number of predictors that will be randomly sampled at each split when creating the tree models. Since there are 8 predictors, it should not be smaller than 1 or larger than 8. `mtry = 8` could represent a decision tree model with predictors.

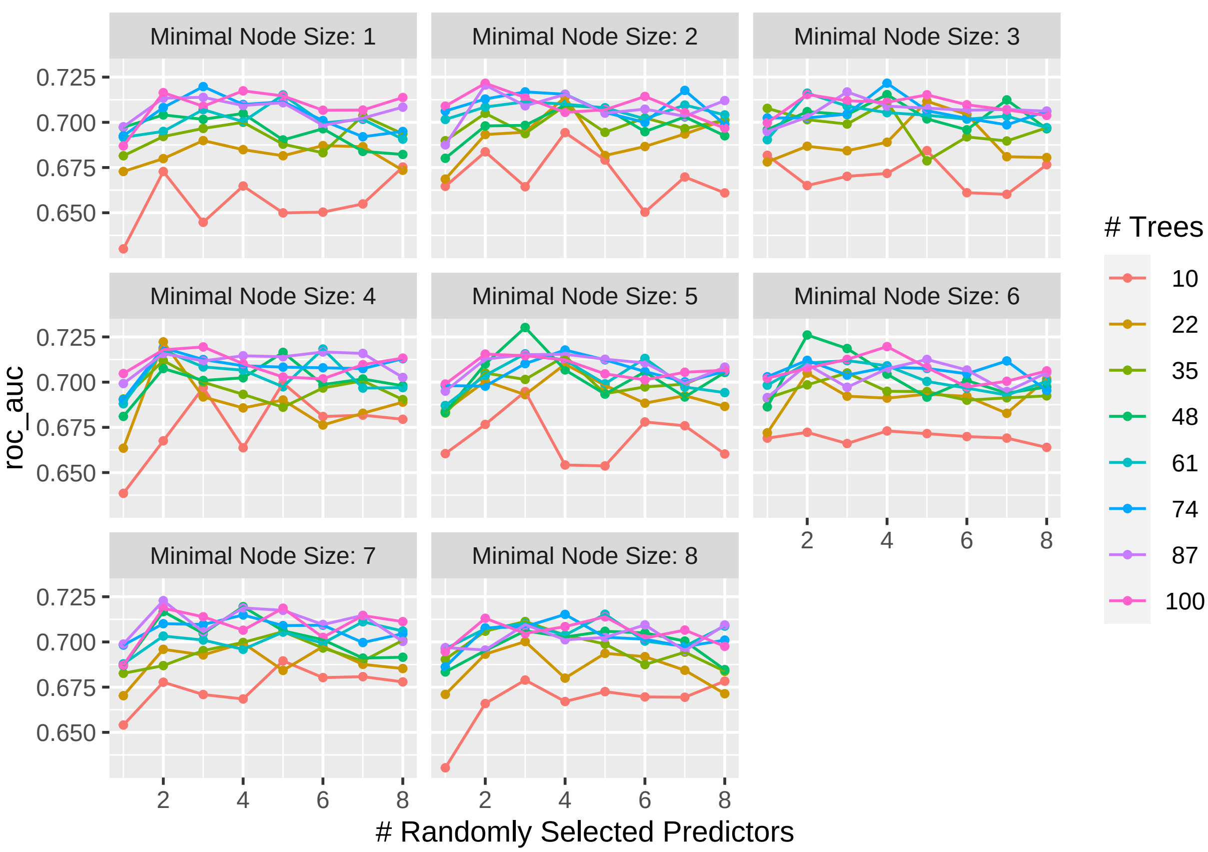
`trees` represents the numbers of trees.

`min_n` represents the minimum number of observations.

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?

```r
library(ranger)
rand_tune_res <- tune_grid(
  rand_tree_wf,
  resamples = pokemon_fold,
  grid = rand_tree_grid,
  metrics = metric_set(roc_auc)
)

autoplot(rand_tune_res)
```



Exercise 7

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

```
arrange(collect_metrics(rand_tune_res), 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     3    48     5 roc_auc hand_till  0.730     5  0.0137 Preprocessor1_Model~
##  2     2    48     6 roc_auc hand_till  0.726     5  0.0166 Preprocessor1_Model~
##  3     2    87     7 roc_auc hand_till  0.723     5  0.0124 Preprocessor1_Model~
##  4     2    22     4 roc_auc hand_till  0.722     5  0.0142 Preprocessor1_Model~
##  5     4    74     3 roc_auc hand_till  0.722     5  0.0122 Preprocessor1_Model~
##  6     2   100     2 roc_auc hand_till  0.722     5  0.0190 Preprocessor1_Model~
##  7     2    87     2 roc_auc hand_till  0.721     5  0.0126 Preprocessor1_Model~
##  8     3    74     1 roc_auc hand_till  0.720     5  0.0128 Preprocessor1_Model~
##  9     4   100     6 roc_auc hand_till  0.720     5  0.0142 Preprocessor1_Model~
## 10     4    48     7 roc_auc hand_till  0.720     5  0.0159 Preprocessor1_Model~
## # ... with 502 more rows
```

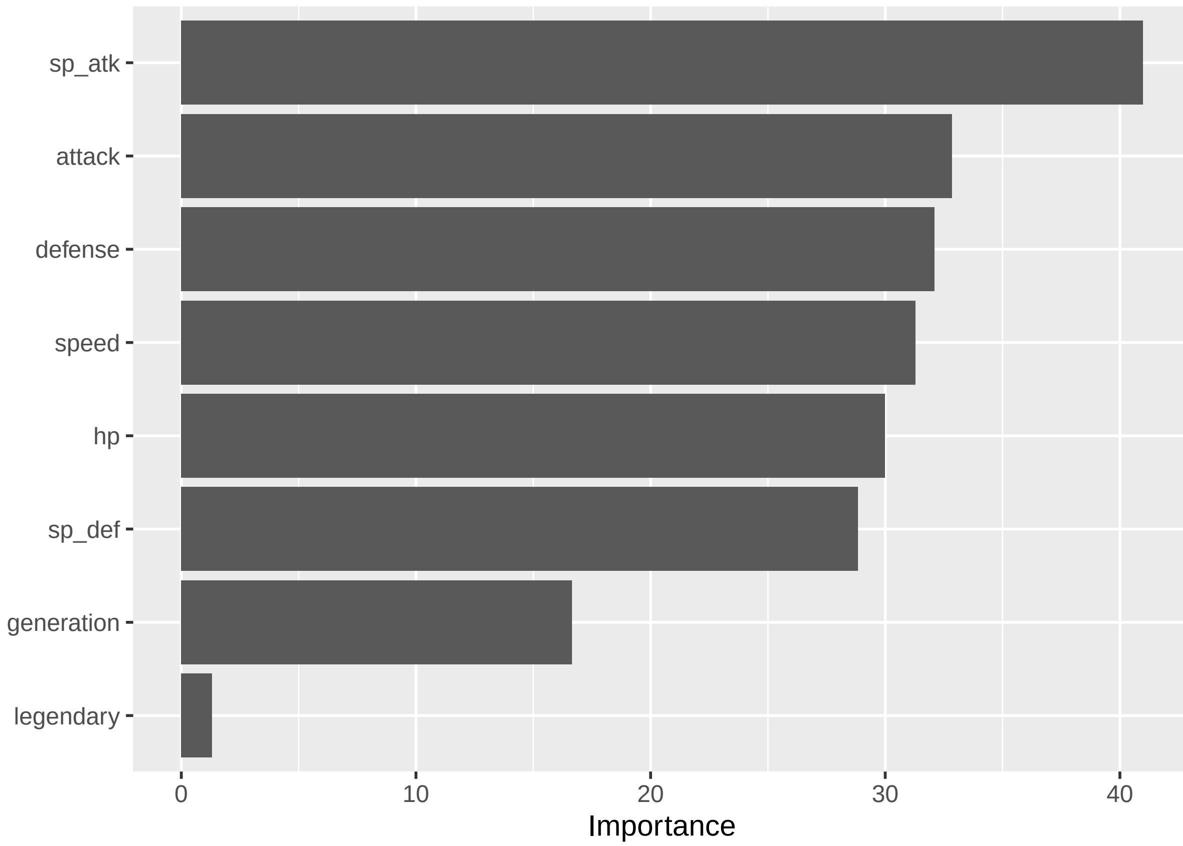
The best-performing random forest is approximately 0.7292. Then `trees` is 87 and `mtry` is 2.

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?

```
best_rand_forest <- select_best(rand_tune_res)

rand_tree_final <- finalize_workflow(rand_tree_wf, best_rand_forest)

rand_tree_final_fit <- fit(rand_tree_final, data = pokemon_train)

vip(rand_tree_final_fit %>%
      extract_fit_engine())
```



The `sp_atk` is the most useful and the `legendary` is the least useful. I think the results are approximately the same as I expected.

Exercise 9

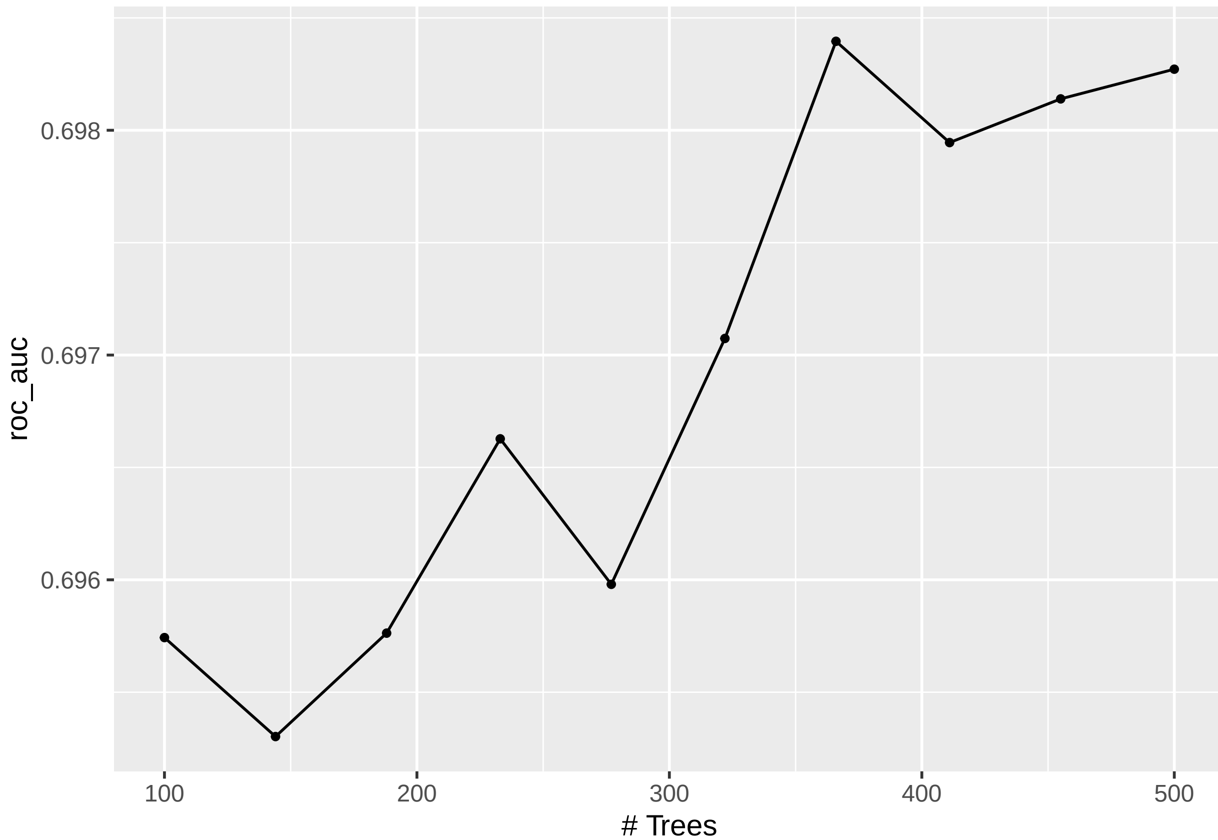
Finally, set up a boosted tree model and workflow. Use the `xgboost` engine. Tune `trees`. Create a regular grid with 10 levels; let `trees` range from 10 to 2000. Specify `roc_auc` and again print an `autoplot()` of the results.

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()`.*

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

boost_grid <- grid_regular(trees(c(100, 500)), levels = 10)

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

boost_tune_res <- tune_grid(
  boost_wf,
  resamples = pokemon_fold,
```

```r
  grid = boost_grid,
  metrics = metric_set(roc_auc)
)

autoplot(boost_tune_res)
```



From the graph, `roc_auc` increases at first sharply, then it starts to decrease.

```r
arrange(collect_metrics(boost_tune_res), desc(mean))
```

```
## # A tibble: 10 x 7
##    trees .metric .estimator  mean     n std_err .config
##    <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1   366 roc_auc hand_till  0.698     5  0.0139 Preprocessor1_Model07
##  2   500 roc_auc hand_till  0.698     5  0.0138 Preprocessor1_Model10
##  3   455 roc_auc hand_till  0.698     5  0.0137 Preprocessor1_Model09
##  4   411 roc_auc hand_till  0.698     5  0.0137 Preprocessor1_Model08
##  5   322 roc_auc hand_till  0.697     5  0.0141 Preprocessor1_Model06
##  6   233 roc_auc hand_till  0.697     5  0.0143 Preprocessor1_Model04
##  7   277 roc_auc hand_till  0.696     5  0.0142 Preprocessor1_Model05
##  8   188 roc_auc hand_till  0.696     5  0.0152 Preprocessor1_Model03
##  9   100 roc_auc hand_till  0.696     5  0.0159 Preprocessor1_Model01
## 10   144 roc_auc hand_till  0.695     5  0.0157 Preprocessor1_Model02
```

The best-performing boosted tree is approximately 0.6999 and the trees are 673.

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.

```
best_boost_tree <- select_best(boost_tune_res)

boost_tree_final <- finalize_workflow(boost_wf, best_boost_tree)

boost_tree_final_fit <- fit(boost_tree_final, data = pokemon_train)
```

```
final_rand_model = augment(rand_tree_final_fit, new_data = pokemon_train)
final_class_model = augment(class_tree_final_fit, new_data = pokemon_train)
final_boost_model = augment(boost_tree_final_fit, new_data = pokemon_train)

test <- bind_rows(
  roc_auc(final_rand_model, truth = type_1, .pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal, .pred_Wate
  roc_auc(final_class_model, truth = type_1, .pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal, .pred_Wat
  roc_auc(final_boost_model, truth = type_1, .pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal, .pred_Wat
)
test
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.805
## 2 roc_auc hand_till      0.615
## 3 roc_auc hand_till      0.799
```

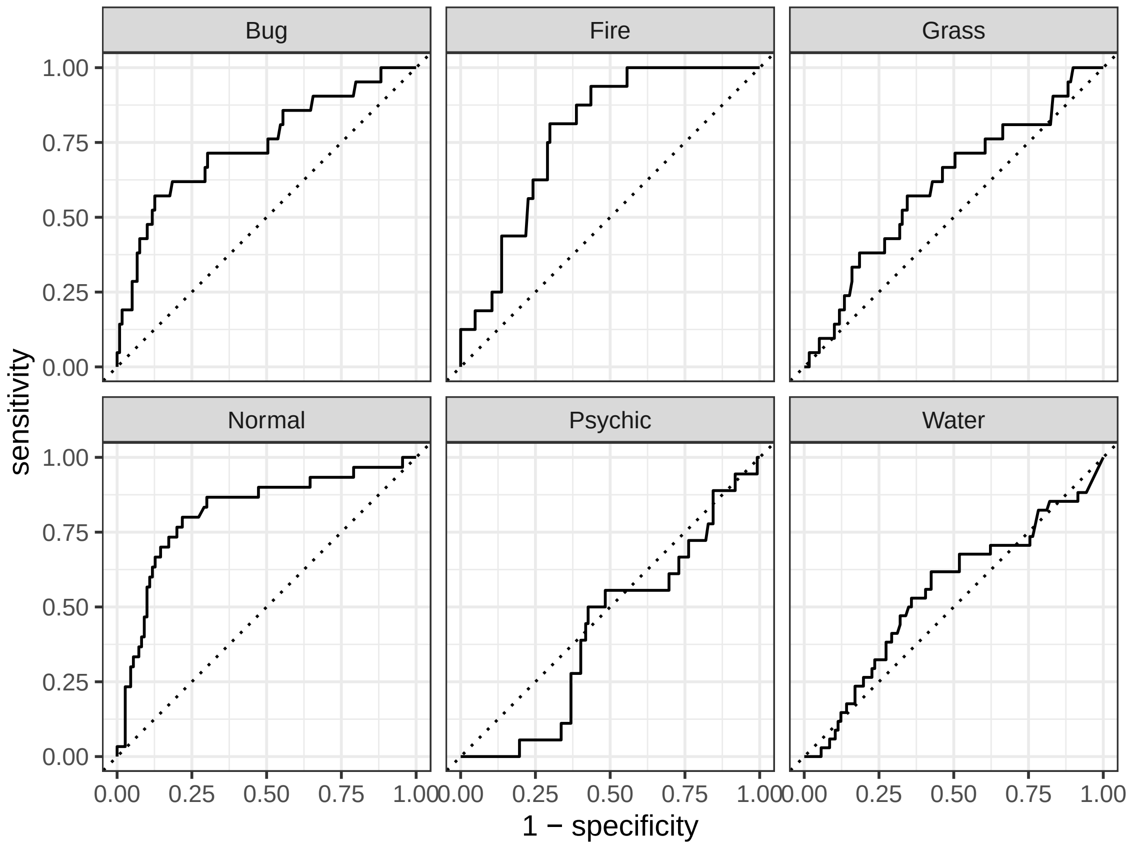
The best model `roc_auc` has approximately 0.791. It is performed by the random forest model.

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.

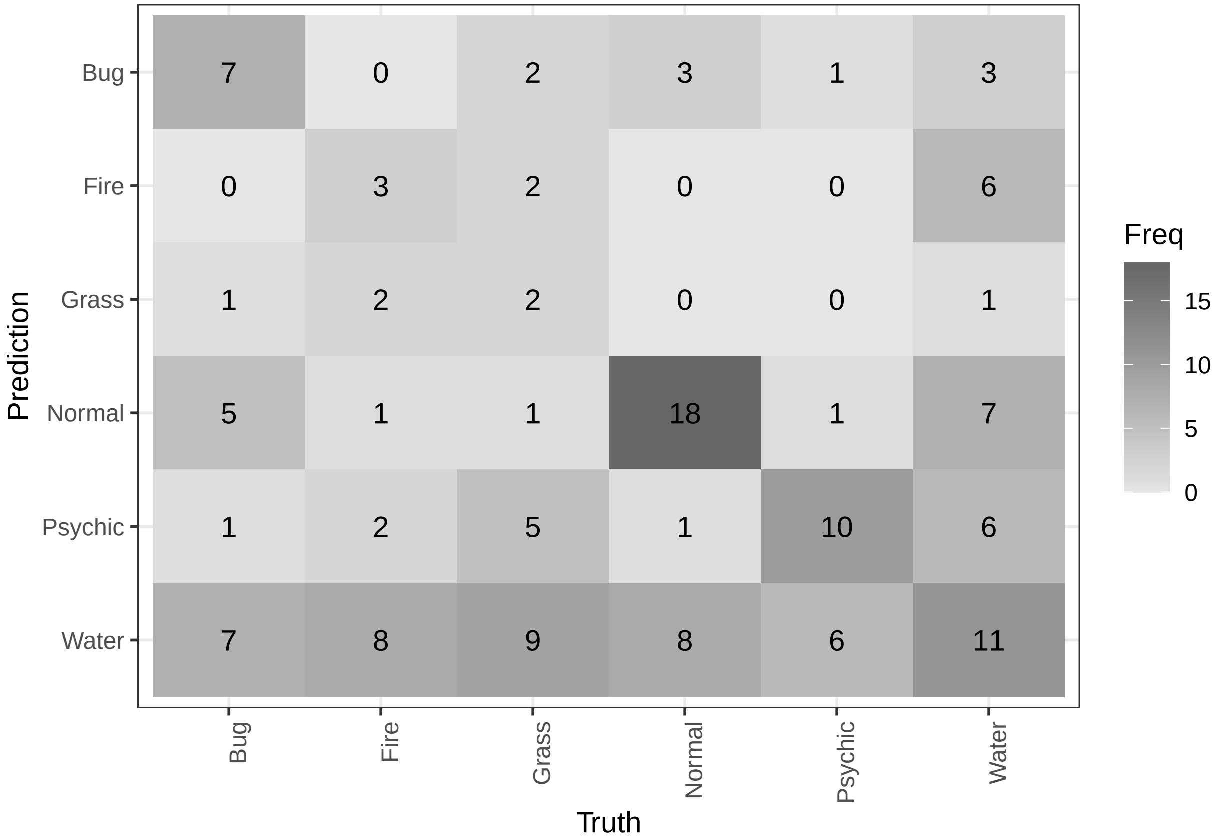
```
final_rand_model_testing = augment(rand_tree_final_fit, new_data = pokemon_test)
roc_auc(final_rand_model_testing, truth = type_1, .pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal, .pre
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.648
```

```
autoplot(roc_curve(final_rand_model_testing, truth = type_1, .pred_Bug, .pred_Fire, .pred_Grass, .pred_N
```




```
conf_mat(final_rand_model_testing, truth = type_1, estimate = .pred_class) %>%
  autoplot(type = "heatmap") + theme_bw() + theme(axis.text.x = element_text(angle = 90, hjust = 1))
```



Which classes was your model most accurate at predicting? Which was it worst at?

The model is most accurate at predicting normal Pokemons. It is worst at predicting psychic Pokemons.