Homework 6

PSTAT 131 John Wei

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(ggplot2)
library(tidyverse)
library(tidymodels)
library(corrplot)
library(klaR)
library(glmnet)
tidymodels_prefer()
Pokemon <- read_csv("Pokemon.csv")
library(janitor)
library(xgboost)
library(rpart.plot)
library(ranger)
library(vip)
library(pROC)
library("rpart.plot")
set.seed(4167)
```

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 <- clean_names(Pokemon)
rare_pokemon <- pokemon %>%
  filter(type_1 == 'Bug' | type_1 == 'Fire' | type_1 == 'Grass' |
  type_1 == 'Normal' | type_1 == 'Water' | type_1 == 'Psychic') %>%
  mutate(type_1 = factor(type_1), legendary = factor(legendary),
         generation = factor(generation))
pokemon_split <- initial_split(rare_pokemon, strata = type_1, prop = 0.8)
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_def, data = pokemon_train) %>%
  step_dummy(c(legendary, generation)) %>%
  step_normalize(all_predictors())
```

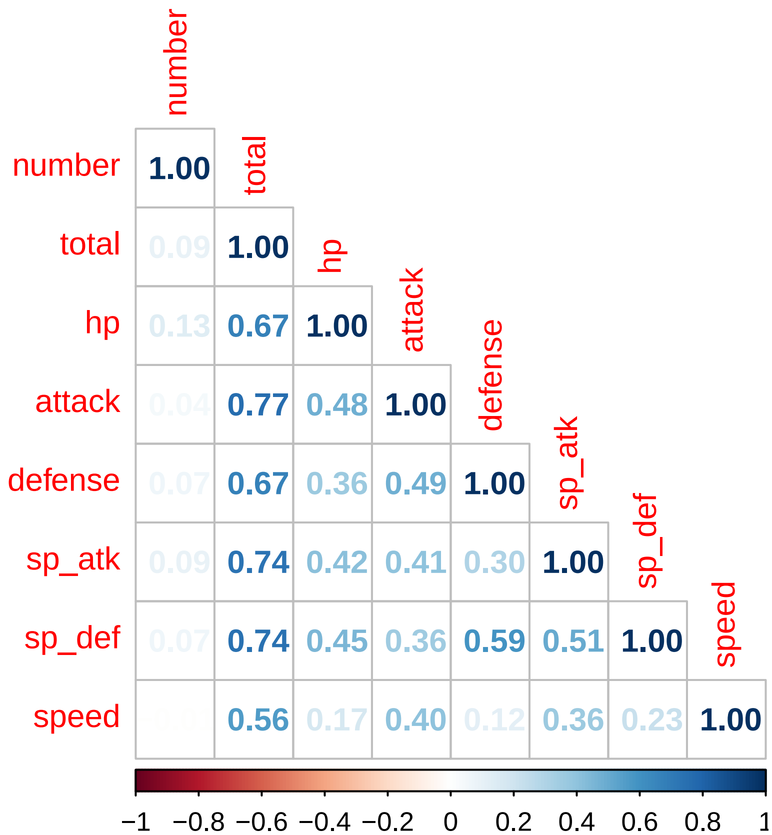
Exercise 2

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).*

```
library(corrplot)
corrmat_training <- pokemon_train[,sapply(pokemon_train,is.numeric)]
head(corrmat_training)
```

```
## # A tibble: 6 x 8
##   number total    hp attack defense sp_atk sp_def speed
##    <dbl> <dbl> <dbl>  <dbl>   <dbl>  <dbl>  <dbl> <dbl>
## 1     10   195    45     30      35     20     20    45
## 2     11   205    50     20      55     25     25    30
## 3     13   195    40     35      30     20     20    50
## 4     14   205    45     25      50     25     25    35
## 5     15   395    65     90      40     45     80    75
## 6     15   495    65    150      40     15     80   145
```

```r
corrplot(cor(corrmat_training), method = 'number', type = 'lower')
```



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

All the variables have positive correlation to each other to varying degrees. They also have correlation to total stats, which does make sense. Special attack, attack, and special defense have especially high correlations to total stats.

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")
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_train)
class_tree_fit %>%
  extract_fit_engine()
```

```
## n= 364
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 364 275 Water (0.15 0.11 0.15 0.21 0.12 0.24)
##    2) sp_atk< 61.5 149  93 Normal (0.26 0.04 0.12 0.38 0.027 0.18)
##      4) speed< 47.5 53  32 Bug (0.4 0.019 0.17 0.21 0.075 0.13)
##        8) hp< 72 46  25 Bug (0.46 0.022 0.15 0.15 0.065 0.15) *
##        9) hp>=72 7   3 Normal (0 0 0.29 0.57 0.14 0) *
##      5) speed>=47.5 96  51 Normal (0.18 0.052 0.094 0.47 0 0.21)
##       10) attack>=35.5 83  40 Normal (0.16 0.06 0.084 0.52 0 0.18) *
##       11) attack< 35.5 13   8 Water (0.31 0 0.15 0.15 0 0.38) *
##    3) sp_atk>=61.5 215 153 Water (0.079 0.16 0.18 0.1 0.19 0.29)
##      6) attack>=51 183 126 Water (0.093 0.17 0.19 0.11 0.13 0.31)
##       12) speed< 109 166 109 Water (0.096 0.18 0.19 0.096 0.09 0.34)
##         24) speed>=79 74  56 Fire (0.12 0.24 0.14 0.14 0.12 0.24)
##           48) generation=1,2 25  13 Fire (0.12 0.48 0.08 0.12 0.04 0.16) *
##           49) generation=3,4,5,6 49  35 Water (0.12 0.12 0.16 0.14 0.16 0.29)
##             98) speed>=87 37  25 Water (0.16 0.081 0.16 0.19 0.081 0.32) *
##             99) speed< 87 12   7 Psychic (0 0.25 0.17 0 0.42 0.17) *
##         25) speed< 79 92  53 Water (0.076 0.13 0.24 0.065 0.065 0.42) *
##       13) speed>=109 17   9 Psychic (0.059 0.059 0.12 0.29 0.47 0) *
##      7) attack< 51 32  14 Psychic (0 0.12 0.12 0.031 0.56 0.16) *
```

```
class_tree_wf <- workflow() %>%
  add_model(class_tree_spec %>% set_args(cost_complexity = tune())) %>%
  add_recipe(pokemon_recipe)
```

```
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)
)
```

With a single decision tree, a lower cost complexity performs better.

Exercise 4

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

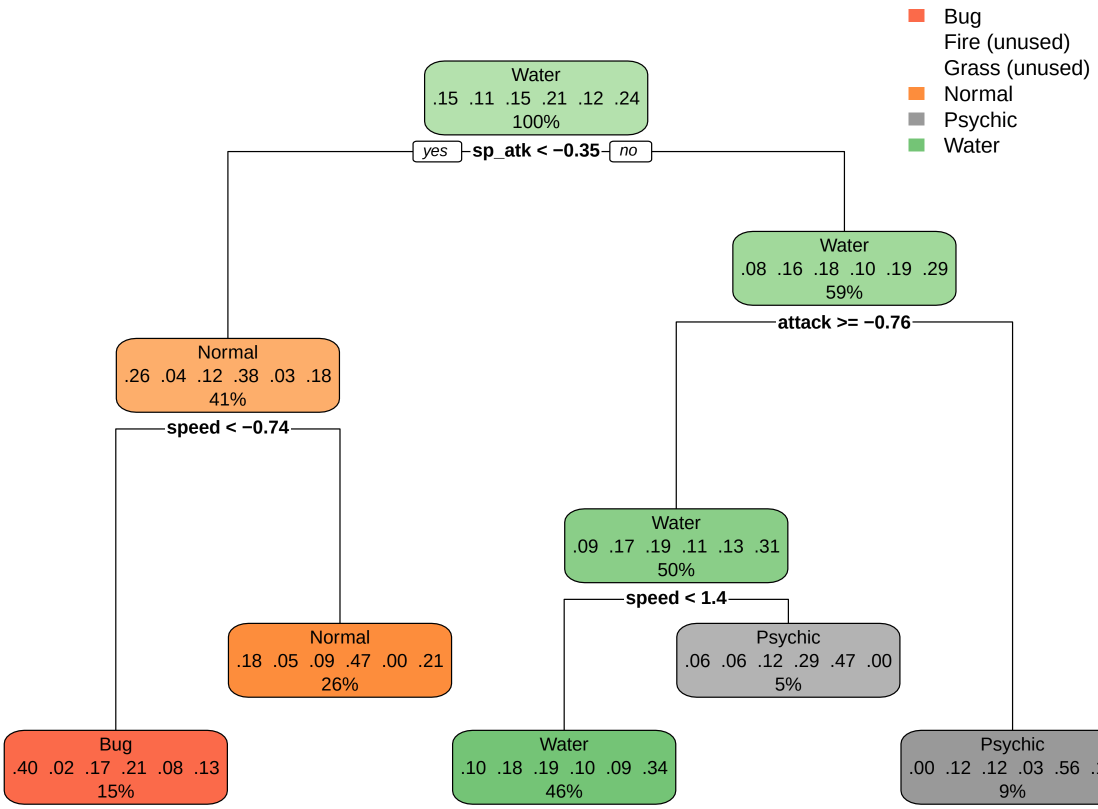
```
metrics_1 <- collect_metrics(tune_res)%>%
  arrange(-mean)
```

Highest mean roc is 0.65792.

Exercise 5

Using `rpart.plot`, fit and visualize your 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()
```



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?**

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

```r
pokemon_grid <- grid_regular(mtry(range = c(1, 8)), trees(range = c(1,5)), min_n(range = c(3,5)), level
```

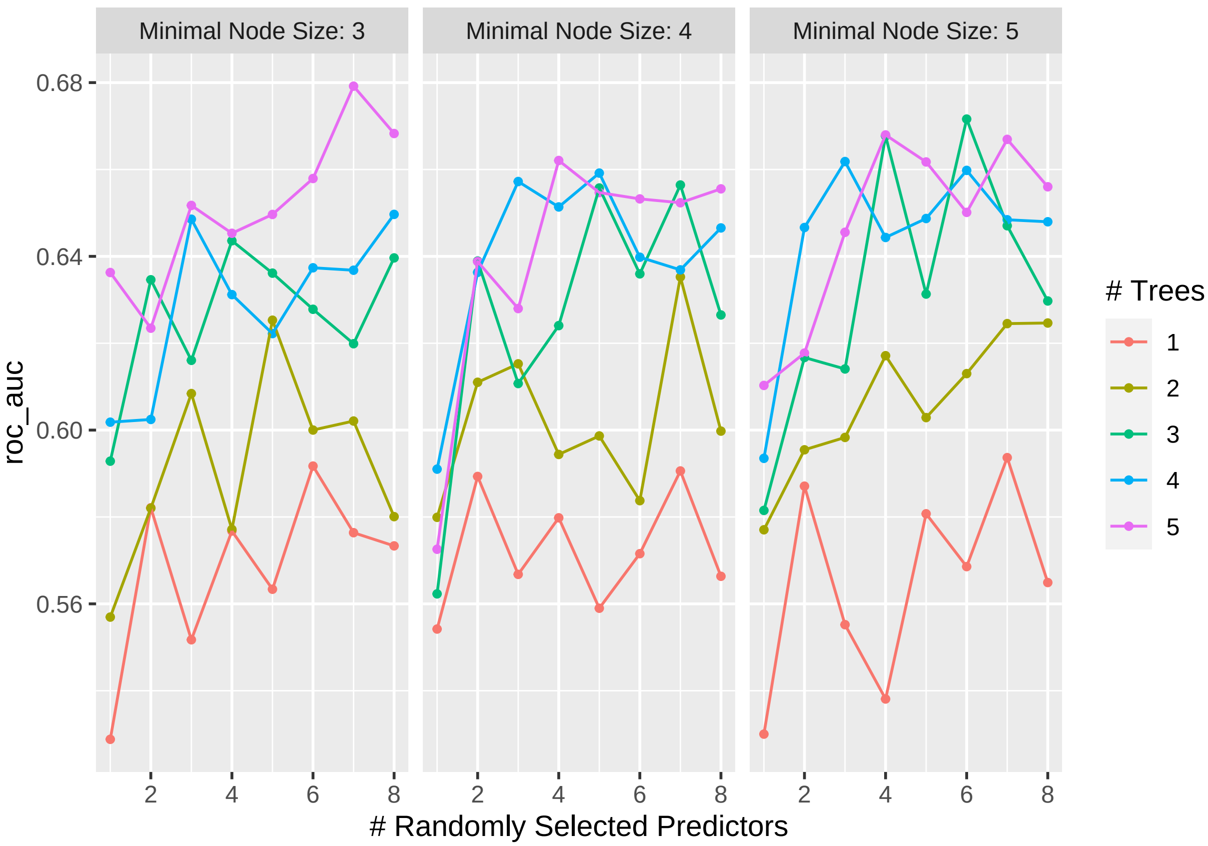
mtry is the number of predictors to be randomly selected in the model at each split, trees is the number of trees, and min_n is the minimal node size.

mtry = 8 means to randomly sample 8 predictors in each split when creating the tree models.

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?

```
tune_res_2 <- tune_grid(rf_wf, resamples = pokemon_fold, grid = pokemon_grid, metrics = metric_set(roc_a
autoplot(tune_res_2)
```



Node size does not seem to affect roc_auc much, but the roc_auc for the most part goes up with higher # of randomly selected predictors. Higher # of trees quite clearly create a higher roc_auc.

Exercise 7

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

```
metrics <- collect_metrics(tune_res_2)
arrange(metrics, desc(mean))
```

```
## # A tibble: 120 x 9
```

```
##     mtry trees min_n .metric .estimator  mean     n std_err .config
##    <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1     7     5     3 roc_auc hand_till  0.679     5 0.0106  Preprocessor1_Model~
##  2     6     3     5 roc_auc hand_till  0.672     5 0.00843 Preprocessor1_Model~
##  3     8     5     3 roc_auc hand_till  0.668     5 0.00612 Preprocessor1_Model~
##  4     4     5     5 roc_auc hand_till  0.668     5 0.00921 Preprocessor1_Model~
##  5     4     3     5 roc_auc hand_till  0.668     5 0.0223  Preprocessor1_Model~
##  6     7     5     5 roc_auc hand_till  0.667     5 0.0108  Preprocessor1_Model~
##  7     4     5     4 roc_auc hand_till  0.662     5 0.00617 Preprocessor1_Model~
##  8     3     4     5 roc_auc hand_till  0.662     5 0.0154  Preprocessor1_Model~
##  9     5     5     5 roc_auc hand_till  0.662     5 0.0105  Preprocessor1_Model~
## 10     6     4     5 roc_auc hand_till  0.660     5 0.00959 Preprocessor1_Model~
## # ... with 110 more rows
```

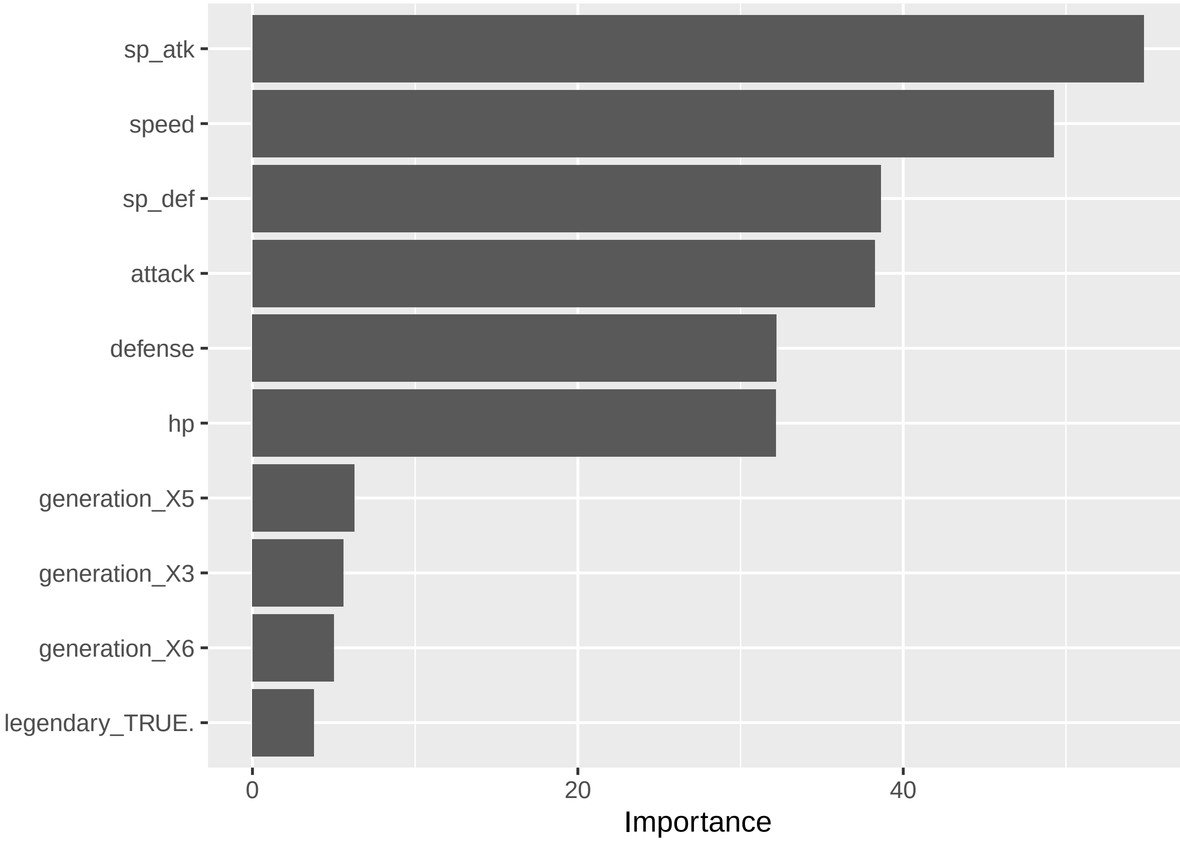
The best performing model has roc_auc at .70616.

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_complexity <- select_best(tune_res_2)
rf_final <- finalize_workflow(rf_wf, best_complexity)
rf_final_fit <- fit(rf_final, data = pokemon_train)
rf_final_fit %>%
  extract_fit_engine() %>%
  vip()
```



Special attack and attack are the highest importance, which makes sense because they also correlated with total stats earlier the most. Generations are the least important, which make sense since in a game you want different generations to be similar to be balanced.

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()`.*

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.

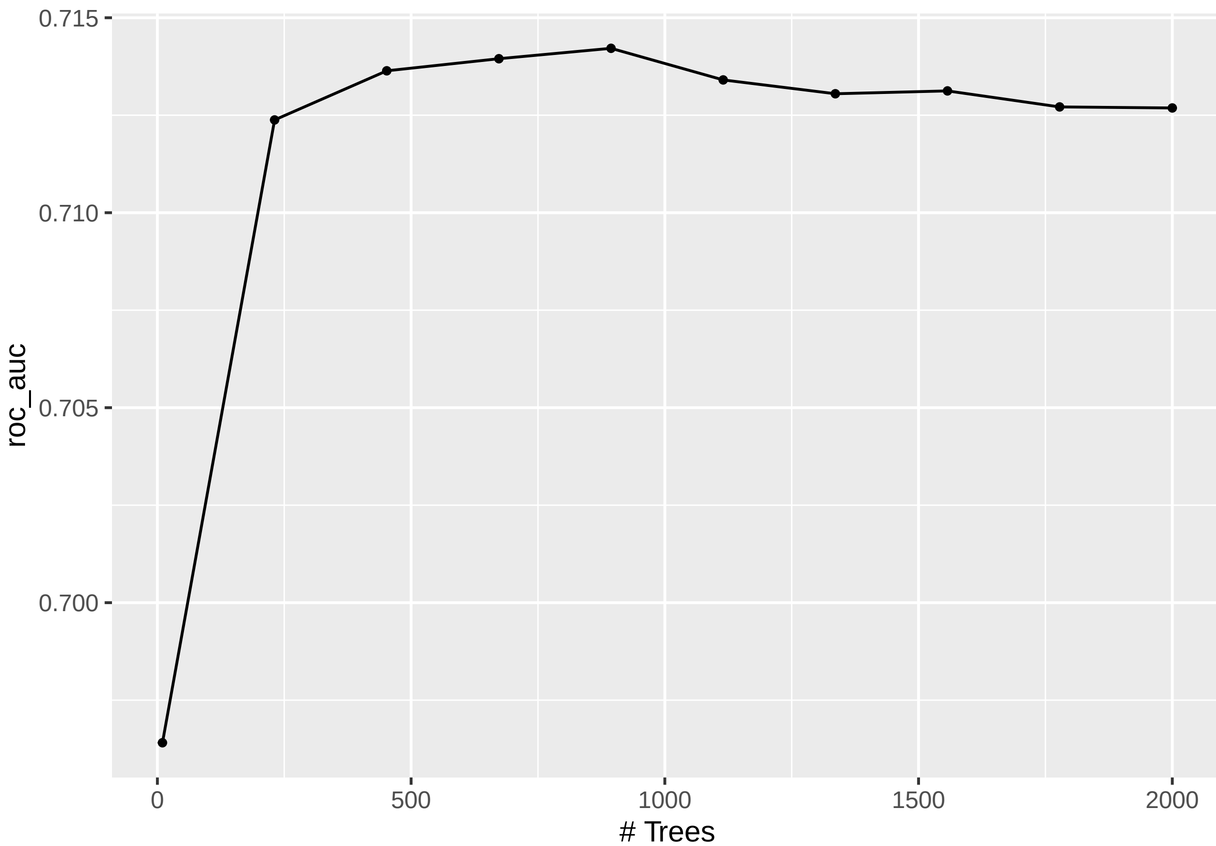
What do you observe?

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

```
boosted_spec <- boost_tree(trees = tune()) %>%
  set_engine("xgboost") %>%
  set_mode("classification")
boosted_workflow <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
```

```
  add_model(boosted_spec)
pokemon_grid_2 <- grid_regular(trees(range = c(10,2000)), levels = 10)
```

```
tune_res_3 <- tune_grid(boosted_workflow, resamples = pokemon_fold, grid = pokemon_grid_2, metrics = met
autoplot(tune_res_3)
```




```
metrics_3 <- collect_metrics(tune_res_3)
arrange(metrics_3, desc(mean))
```

```
## # A tibble: 10 x 7
##    trees .metric .estimator  mean     n std_err .config
##    <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1   894 roc_auc hand_till  0.714     5  0.0145 Preprocessor1_Model05
##  2   673 roc_auc hand_till  0.714     5  0.0149 Preprocessor1_Model04
##  3   452 roc_auc hand_till  0.714     5  0.0151 Preprocessor1_Model03
##  4  1115 roc_auc hand_till  0.713     5  0.0147 Preprocessor1_Model06
##  5  1557 roc_auc hand_till  0.713     5  0.0148 Preprocessor1_Model08
##  6  1336 roc_auc hand_till  0.713     5  0.0144 Preprocessor1_Model07
##  7  1778 roc_auc hand_till  0.713     5  0.0150 Preprocessor1_Model09
##  8  2000 roc_auc hand_till  0.713     5  0.0153 Preprocessor1_Model10
##  9   231 roc_auc hand_till  0.712     5  0.0140 Preprocessor1_Model02
## 10    10 roc_auc hand_till  0.696     5  0.0125 Preprocessor1_Model01
```

The roc_auc of the best performing boosted tree model is .7276 at 231 trees. It peaks here and drops off with more trees. The roc_auc flucuates at most by .0045.

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?

```
prunedtree_roc <- max(metrics_1$mean)
randomforest_roc <- max(metrics$mean)
boosted_roc <- max(metrics_3$mean)
roc_values <- bind_cols(prunedtree_roc,randomforest_roc, boosted_roc)
colnames(roc_values) <-  c('prune','random forest','boosted')
roc_values
```

```
## # A tibble: 1 x 3
##   prune `random forest` boosted
##   <dbl>           <dbl>   <dbl>
## 1 0.668           0.679   0.714
```

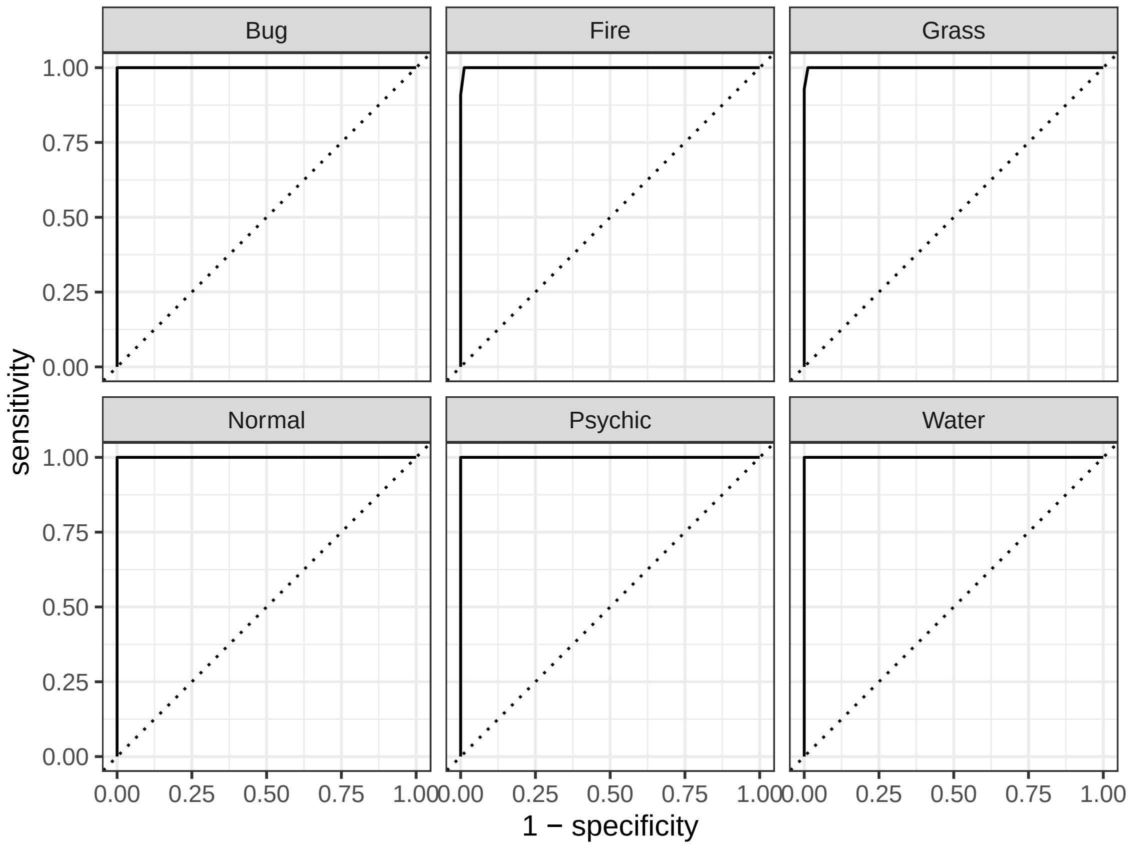
The best performing model is the boosted one.

```
roc <- select_best(tune_res_3, metric = "roc_auc")
boosted_final <- finalize_workflow(boosted_workflow, roc)
boosted_final_fit <- fit(boosted_final, data = pokemon_test)
```

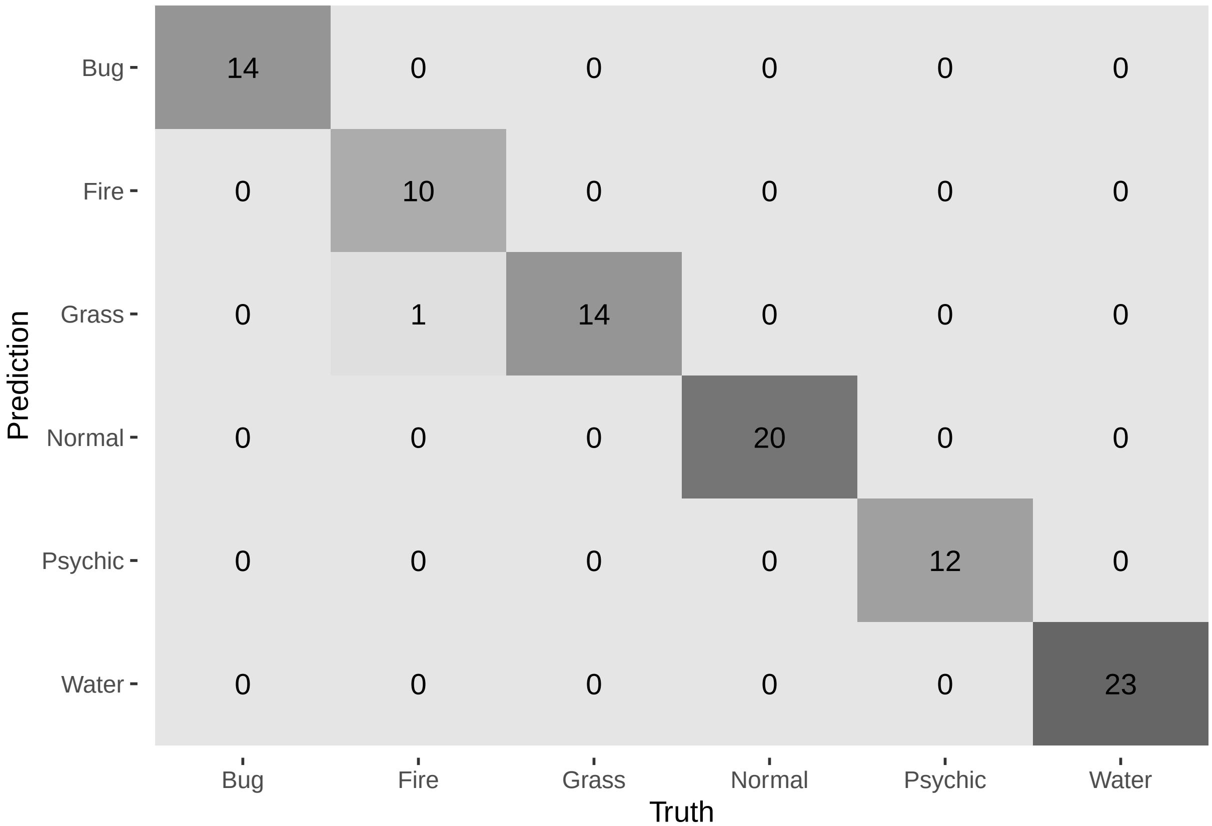
Print the AUC value of your best-performing model on the testing set. Print the ROC curves. Finally, create and visualize a confusion matrix heat map.

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

```
augment(boosted_final_fit, new_data = pokemon_test) %>%
  roc_curve(type_1, estimate = c(.pred_Bug, .pred_Fire,.pred_Grass, .pred_Normal, .pred_Psychic, .pred_W
  autoplot()
```




```
augment(boosted_final_fit, new_data = pokemon_test) %>%
  conf_mat(truth = type_1, estimate = .pred_class) %>%
  autoplot(type = "heatmap")
```



The model is best at predicting normal and water and worst at fire and psychic.