Homework 6

Aidan Baker

Contents

Tree-Based Models 1

Tree-Based Models

Exercise 1

Read in the data and set things up as in Homework 5:

```
library(tidyverse)
library(tidymodels)
library(ggplot2)
library(janitor)

pokemon <- read.csv('/Users/aidanbaker/Downloads/homework-5/data/pokemon.csv') %>%
  clean_names()

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

pokemon <- pokemon %>%
 mutate(type_1 = factor(type_1),
       legendary = factor(legendary),
       generation = factor (generation))

pokemon = subset(pokemon, select = -x )
```

- 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.seed(2001)

pokemon_split <- initial_split(pokemon, prop = 0.75,
                                strata = type_1)
pokemon_training <- training(pokemon_split)
pokemon_testing <- testing(pokemon_split)

pokemon_fold <- vfold_cv(pokemon_training, v = 5,
                         strata = type_1)

head(pokemon)
```

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

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_recipe <- recipe(type_1 ~  legendary + generation + sp_atk + attack + speed + defense + hp + sp
  step_novel(all_nominal_predictors()) %>%
  step_dummy(all_nominal_predictors())%>%
  step_zv(all_predictors()) %>%
  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).*

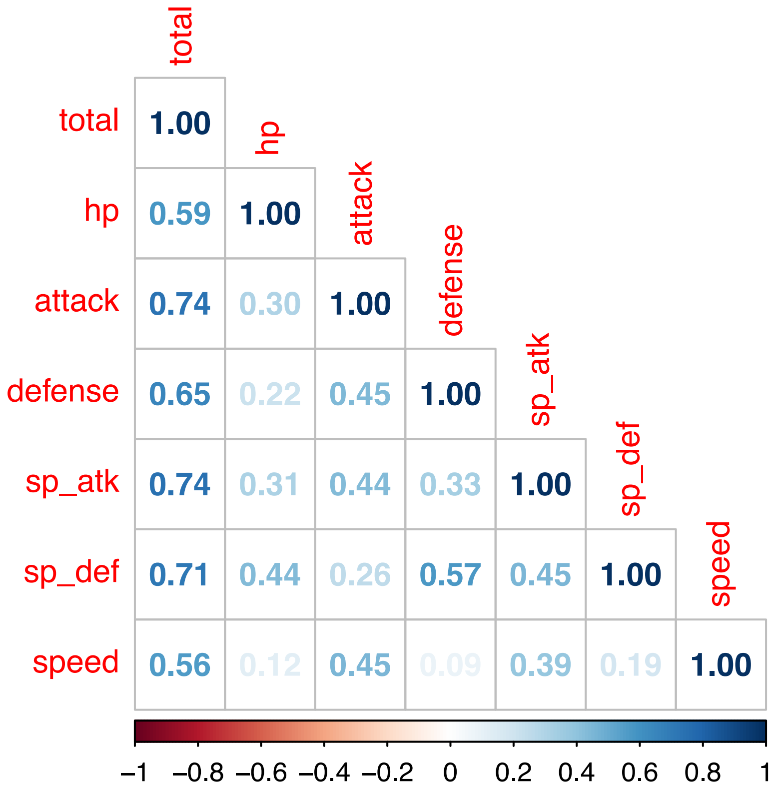
```
library(corrplot)

corrmat_training <- pokemon_training[,sapply(pokemon_training,is.numeric)]

head(corrmat_training)
```

```
##    total hp attack defense sp_atk sp_def speed
## 14   195 45     30      35     20     20    45
## 17   195 40     35      30     20     20    50
## 18   205 45     25      50     25     25    35
## 19   395 65     90      40     45     80    75
## 20   495 65    150      40     15     80   145
## 36   285 35     70      55     45     55    25
```

```
#remove non-numeric numbers
corrplot(cor(corrmat_training), method = 'number', type = 'lower') #normalize the data
```



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

We see the main relationship between total and all the other predictors, since they are directly related. When the pokemon have higher hp or attack, the higher the total will be.

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?

```
library(rpart.plot)
library(vip)
library(janitor)
library(randomForest)
library(xgboost)


set.seed(2001)

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
```

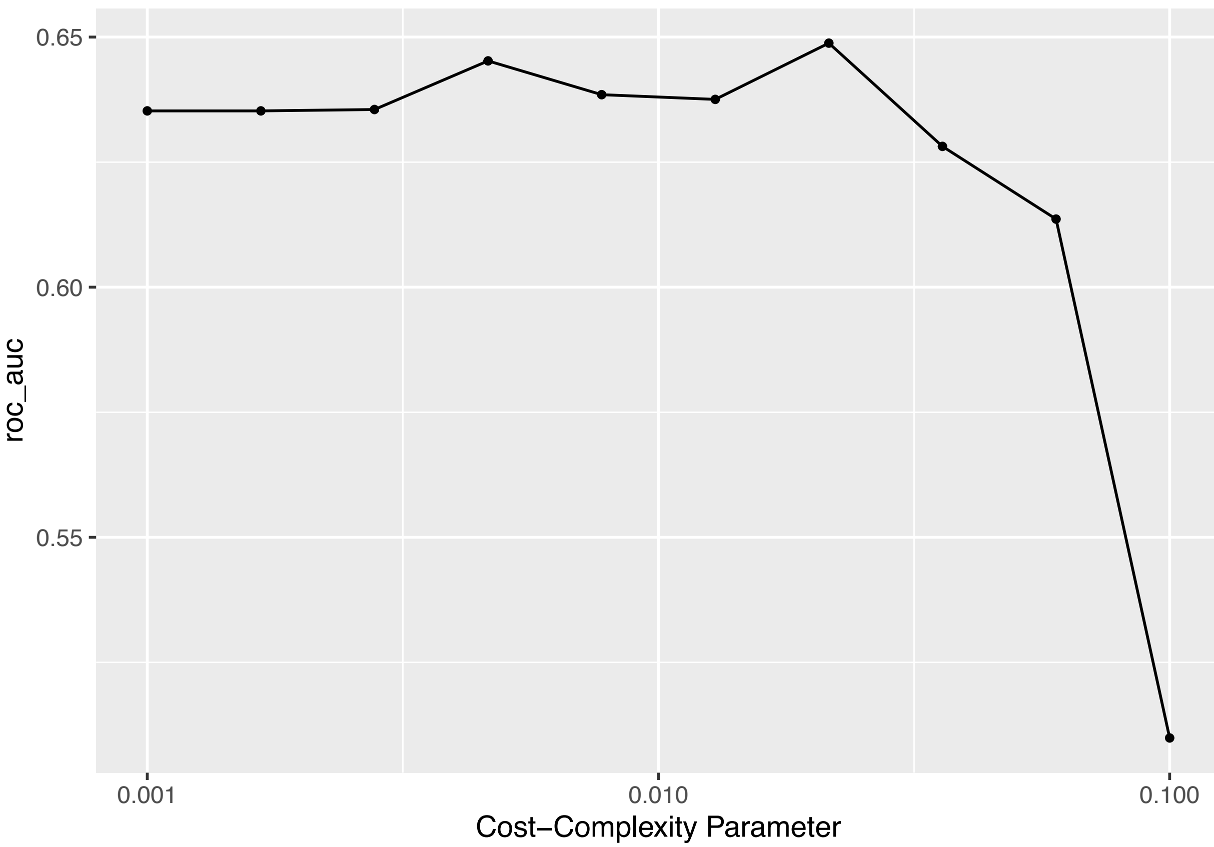
```r
class_tree_workflow <- workflow() %>%
  add_model(class_tree_spec %>% set_args(cost_complexity = tune())) %>%
  add_recipe(pokemon_recipe)

pokemon_folding <- vfold_cv(pokemon_training)

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

tune_res <- tune_grid(class_tree_workflow, resamples = pokemon_folding, grid = parameter_grid, metrics =

autoplot(tune_res)
```




```r
bestcomplexity <- select_best(tune_res, metric = "roc_auc")
bestcomplexity
```

```
## # A tibble: 1 x 2
##   cost_complexity .config
##             <dbl> <chr>
## 1          0.0215 Preprocessor1_Model07
```

Based on the negative overall slope of our curve, we see that a lower cost complexity will yield a higher roc.

Exercise 4

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

```r
metrics <- collect_metrics(tune_res)
arrange(metrics, 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  0.649    10  0.0154 Preprocessor1_Model07
##  2         0.00464 roc_auc hand_till  0.645    10  0.0217 Preprocessor1_Model04
##  3         0.00774 roc_auc hand_till  0.638    10  0.0222 Preprocessor1_Model05
##  4         0.0129  roc_auc hand_till  0.638    10  0.0240 Preprocessor1_Model06
##  5         0.00278 roc_auc hand_till  0.636    10  0.0200 Preprocessor1_Model03
##  6         0.001   roc_auc hand_till  0.635    10  0.0196 Preprocessor1_Model01
##  7         0.00167 roc_auc hand_till  0.635    10  0.0196 Preprocessor1_Model02
##  8         0.0359  roc_auc hand_till  0.628    10  0.0141 Preprocessor1_Model08
##  9         0.0599  roc_auc hand_till  0.614    10  0.0168 Preprocessor1_Model09
## 10         0.1     roc_auc hand_till  0.510    10  0.0114 Preprocessor1_Model10
```

We can see the our highest mean roc was about 65%. ### Exercise 5

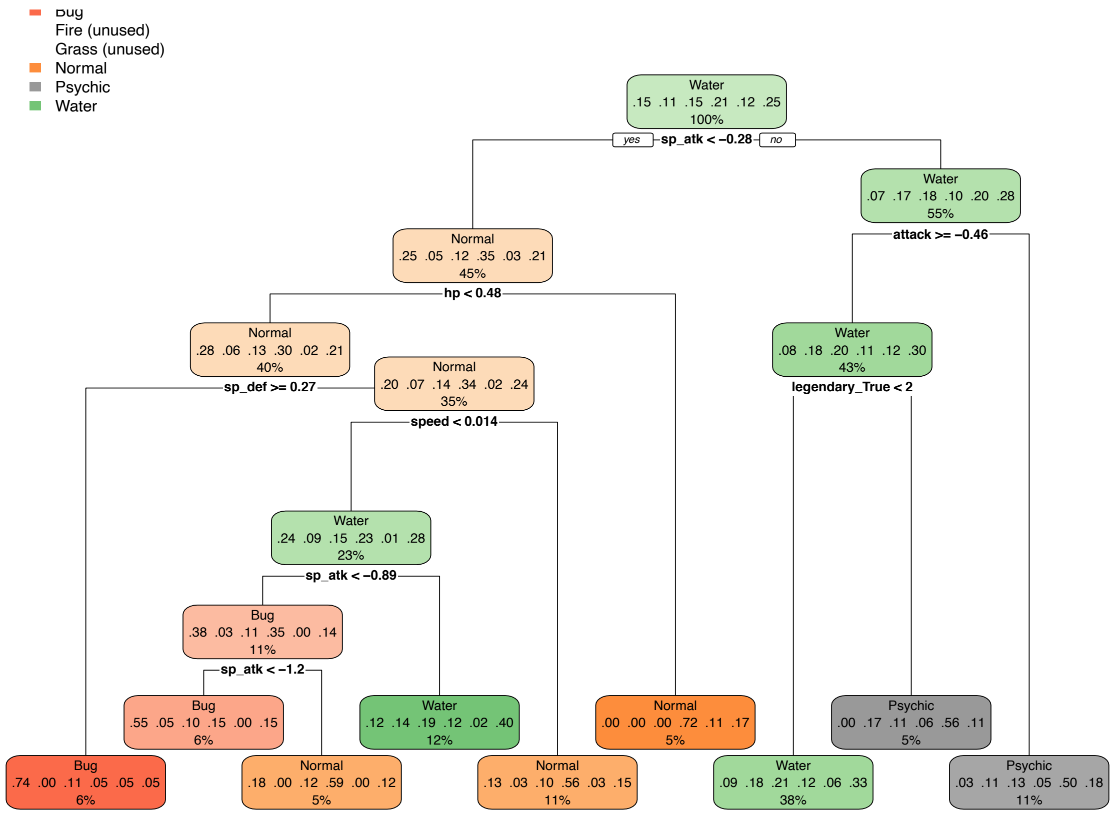
Using `rpart.plot`, fit and visualize your best-performing pruned decision tree with the *training* set.

```r
bestcomplexity <- select_best(tune_res)

class_tree_final <- finalize_workflow(class_tree_workflow, bestcomplexity)

class_tree_final_fit <- fit(class_tree_final, data = pokemon_training)

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?**

```
library(ranger)

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


randomforest_workflow <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(randomforest_spec)

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

mtry is the # of variables we sampled at each split we do, trees is the # of trees were doing, min_n is minimum nodes

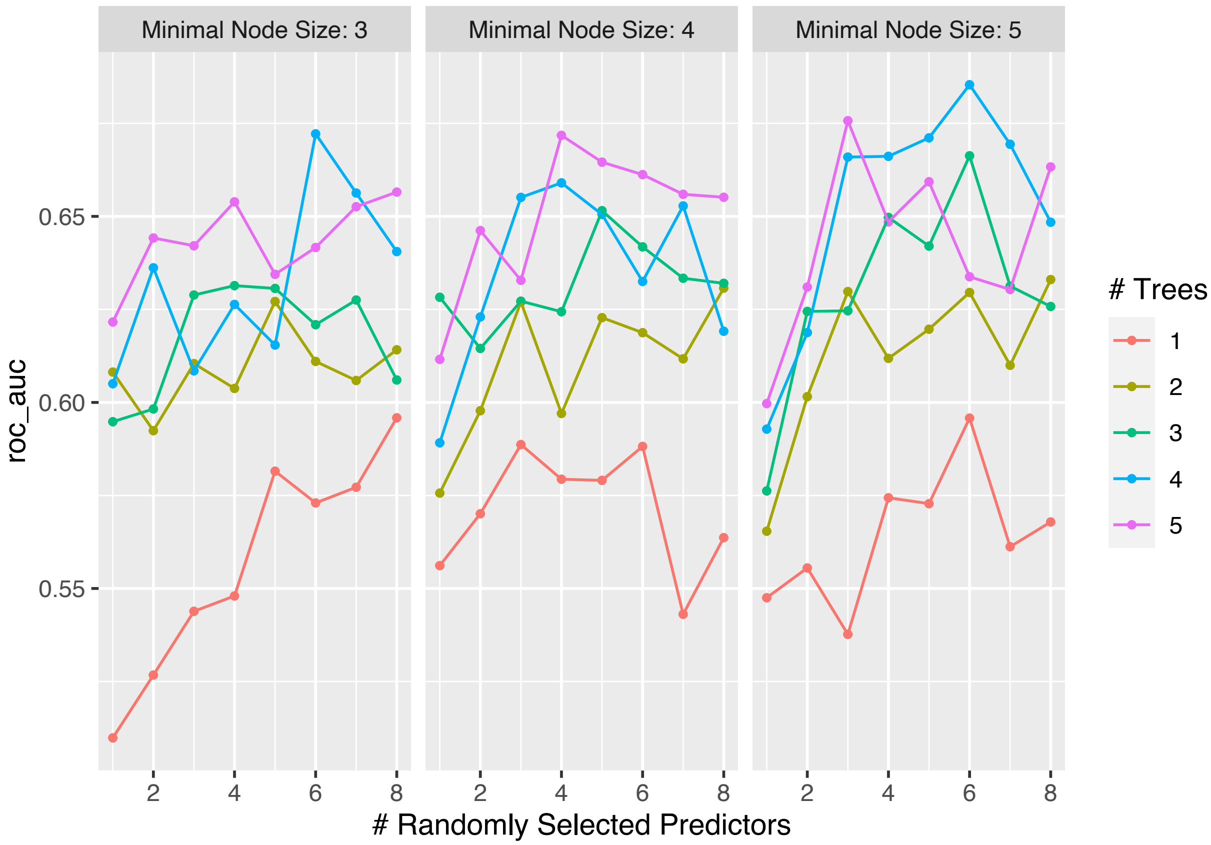
we cannot have a mtry greater than 8 due to only having 8 variables. we must have positive values for our other entries.

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(randomforest_workflow, resamples = pokemon_folding, grid = pokemon_grid, metrics

autoplot(tune_res_2)
```



it looks like minimal node size 5 produced the best roc with 5 trees and 6 predictors.

Exercise 7

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

```
metrics_2 <- collect_metrics(tune_res_2)
arrange(metrics_2, 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     6     4     5 roc_auc hand_till  0.685    10  0.0147 Preprocessor1_Model~
##  2     3     5     5 roc_auc hand_till  0.676    10  0.0124 Preprocessor1_Model~
##  3     6     4     3 roc_auc hand_till  0.672    10  0.0173 Preprocessor1_Model~
##  4     4     5     4 roc_auc hand_till  0.672    10  0.0126 Preprocessor1_Model~
##  5     5     4     5 roc_auc hand_till  0.671    10  0.0172 Preprocessor1_Model~
##  6     7     4     5 roc_auc hand_till  0.669    10  0.0178 Preprocessor1_Model~
##  7     6     3     5 roc_auc hand_till  0.666    10  0.0197 Preprocessor1_Model~
```

```
## 8      4     4      5 roc_auc hand_till  0.666     10  0.0164 Preprocessor1_Model~
## 9      3     4      5 roc_auc hand_till  0.666     10  0.0272 Preprocessor1_Model~
## 10     5     5      4 roc_auc hand_till  0.665     10  0.0152 Preprocessor1_Model~
## # ... with 110 more rows
```

We see that the ROC % of the best performing random forest is about 69%

Exercise 8

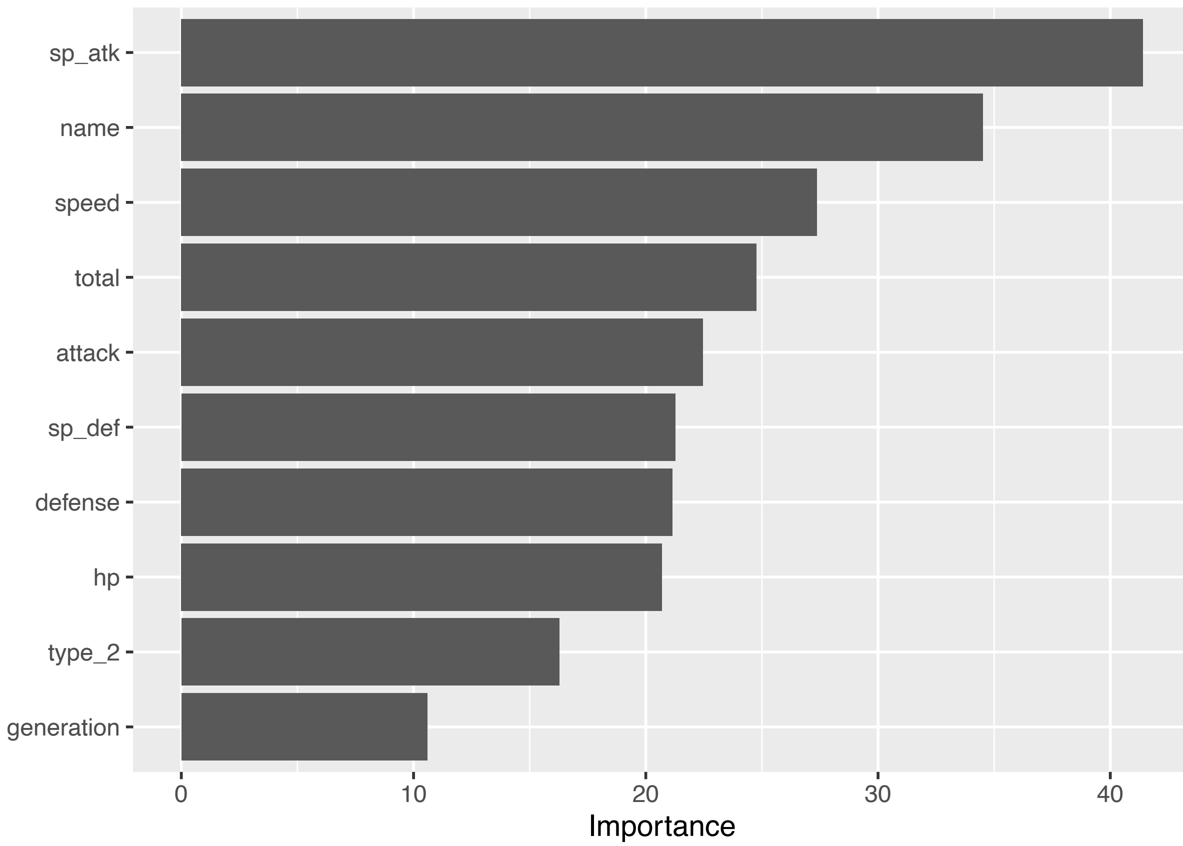
Create a variable importance plot, using `vip()`, with your best-performing random forest model fit on the *training* set.

Which variables were most useful? Which were least useful? Are these results what you expected, or not?

```
randomforest_spec1 <- rand_forest(mtry = 6, trees = 5,min_n = 5) %>%
  set_mode("classification") %>%
  set_engine("ranger", importance = 'impurity')

randomforest_fit <- fit(randomforest_spec1, type_1 ~ ., data = pokemon_training)

vip(randomforest_fit)
```



we can see sp_atk is most important and generation is least important.

i'd say the results are expected, even though i dont know much about pokemon in general, it seems like special attack would be the most important while the generation of the pokemon matters less than other statistics.

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()`.*

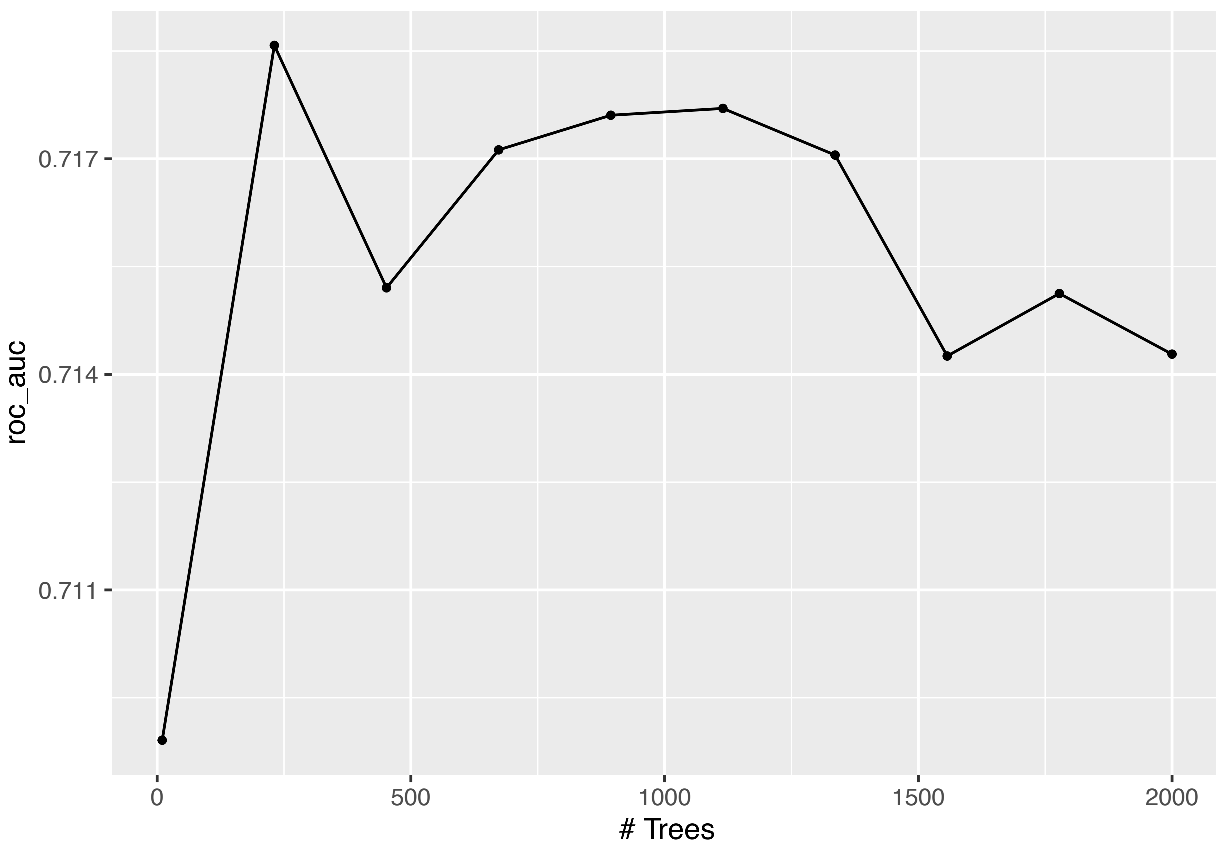
```
boosted_spec <- boost_tree(trees = tune()) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

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_folding, grid = pokemon_grid_2, metrics =

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   231 roc_auc hand_till  0.719    10 0.00805 Preprocessor1_Model02
##  2  1115 roc_auc hand_till  0.718    10 0.00893 Preprocessor1_Model06
##  3   894 roc_auc hand_till  0.718    10 0.00880 Preprocessor1_Model05
##  4   673 roc_auc hand_till  0.717    10 0.00854 Preprocessor1_Model04
##  5  1336 roc_auc hand_till  0.717    10 0.00964 Preprocessor1_Model07
##  6   452 roc_auc hand_till  0.715    10 0.00843 Preprocessor1_Model03
##  7  1778 roc_auc hand_till  0.715    10 0.00999 Preprocessor1_Model09
##  8  2000 roc_auc hand_till  0.714    10 0.0102  Preprocessor1_Model10
##  9  1557 roc_auc hand_till  0.714    10 0.0101  Preprocessor1_Model08
## 10    10 roc_auc hand_till  0.709    10 0.00908 Preprocessor1_Model01
```

As we can see, 231 trees yields a marginally higher ROC of 72%.

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.

```r
prunedtree_roc <- max(metrics$mean)
randomforest_roc <- max(metrics_2$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 0.649           0.685   0.719
```

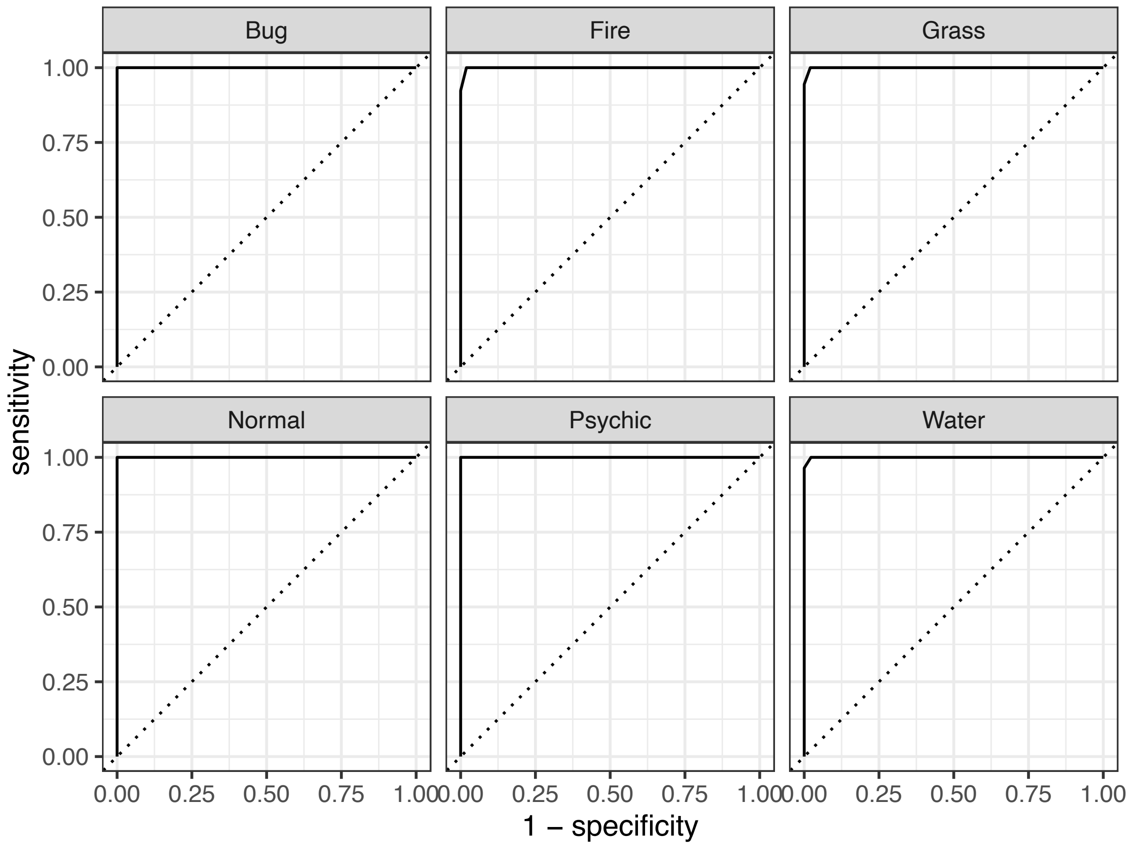
Wee see that the boosted model performed the best

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

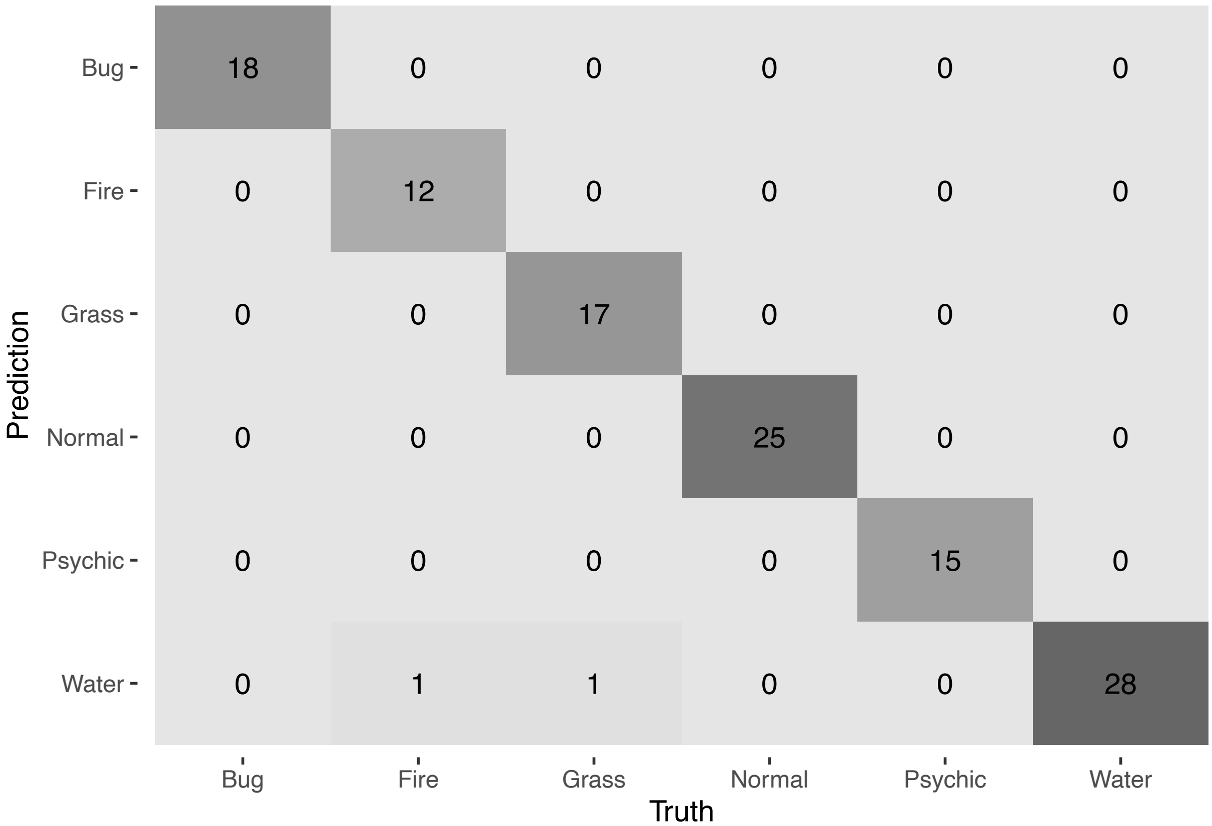
Print the AUC value of your best-performing model on the testing set. Print the ROC curves. Finally, create and visualize a confusion matrix heat map.

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

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




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



Based on the confusion matrix given, it looks like the boosted model is accurate at predicting.