Hw6

Peter Chu

11/24/2022

Question 1

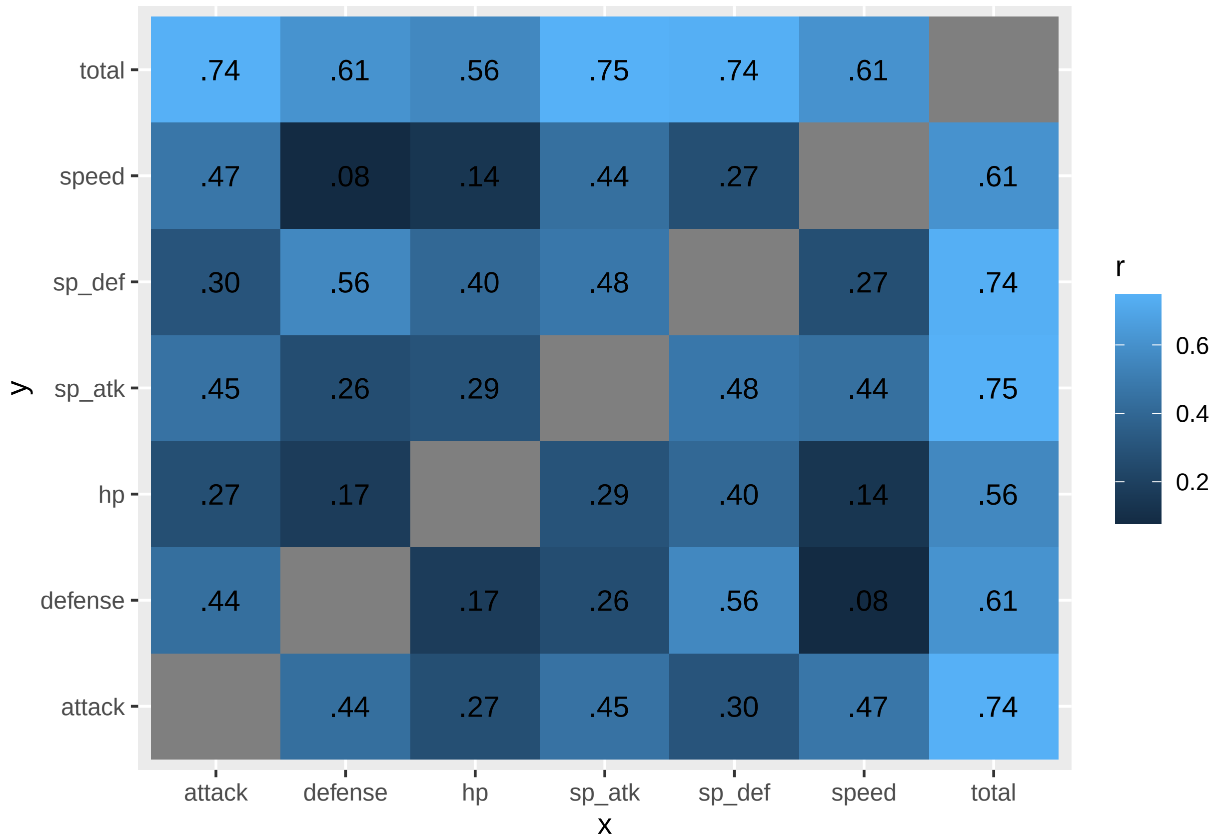
```
data <- read.csv('C:/Users/peter/Desktop/hw6/Pokemon.csv')

data <- clean_names(data)

types <- c('Bug','Fire','Grass','Normal','Water','Psychic')

new_data <- filter(data, data$type_1 %in% types)

data <- new_data

data$type_1 <- as.factor(data$type_1)
data$legendary <- as.factor(data$legendary)

set.seed(104)

data_split <- initial_split(data, strata = type_1, prop = 0.7)
data_train <- training(data_split)
data_test <- testing(data_split)

data_train_fold <- vfold_cv(data_train, v = 5, strata = type_1)

data_rec <- recipe(type_1 ~ legendary + generation + sp_atk + attack + speed + defense + hp + sp_def, da
  step_dummy(all_nominal_predictors()) %>%
  step_center(all_nominal_predictors()) %>%
  step_scale(all_nominal_predictors())
```

Question 2

```
cor_data <- data_train %>%
  select(-type_1) %>%
  select(-name) %>%
  select(-x) %>%
  select(-generation) %>%
  correlate()
```

```
## Non-numeric variables removed from input: `type_2`, and `legendary`
## Correlation computed with
## * Method: 'pearson'
## * Missing treated using: 'pairwise.complete.obs'
```

```
cor_data %>%
  stretch() %>%
  ggplot(aes(x,y, fill = r)) + geom_tile() + geom_text(aes(label = as.character(fashion(r))))
```

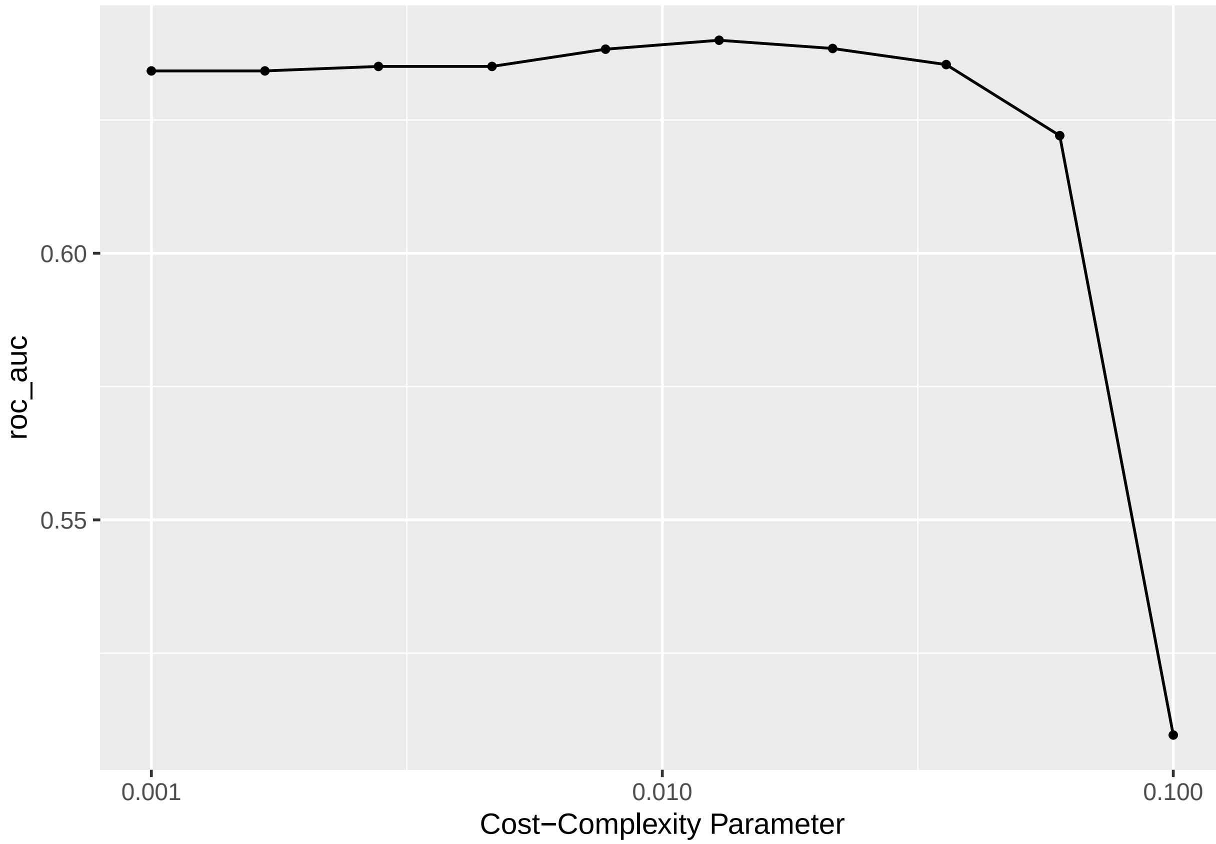


There appears to be a relationship between sp_atk and attack, sp_def and def total and everything, and speed and attack. This makes sense as if a Pokemon has a certain attack value, its special attack is probably based off of that value. sp_def and def are similar to this. In addition the total of all stats summed will clearly be correlated to its components. Speed and attack make sense as generally those with lower speed tend to have higher attack and vise versa.

Question 3

```
tree_spec <- decision_tree() %>%
  set_engine('rpart') %>%
  set_mode('classification')

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

```r
tree_param <- grid_regular(cost_complexity(range = c(-3,-1)), levels = 10)

tune_tree_res <- tune_grid(tree_wf, resamples = data_train_fold, grid = tree_param, metrics = metric_se

autoplot(tune_tree_res)
```



From our autoplot it appears that trees with smaller cost_complexity tend to do better. At 0.1 the roc_auc is around 0.45 while a value of 0.001 has a roc_auc of 0.67.

Question 4

```r
tree_metrics <- collect_metrics(tune_tree_res) %>% dplyr::arrange(desc(mean))

tree_metrics
```

```
## # A tibble: 10 x 7
##    cost_complexity .metric .estimator  mean     n std_err .config
##              <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1         0.0129  roc_auc hand_till  0.640     5 0.0182  Preprocessor1_Model06
##  2         0.0215  roc_auc hand_till  0.638     5 0.0160  Preprocessor1_Model07
##  3         0.00774 roc_auc hand_till  0.638     5 0.0186  Preprocessor1_Model05
##  4         0.0359  roc_auc hand_till  0.635     5 0.0178  Preprocessor1_Model08
##  5         0.00278 roc_auc hand_till  0.635     5 0.0226  Preprocessor1_Model03
##  6         0.00464 roc_auc hand_till  0.635     5 0.0226  Preprocessor1_Model04
##  7         0.001   roc_auc hand_till  0.634     5 0.0220  Preprocessor1_Model01
##  8         0.00167 roc_auc hand_till  0.634     5 0.0220  Preprocessor1_Model02
```

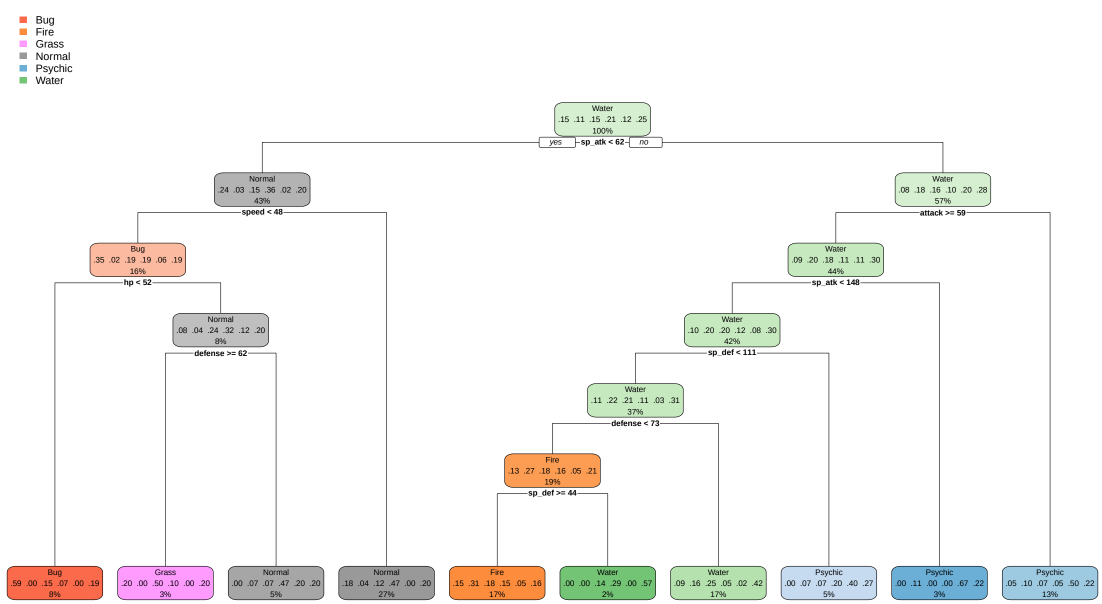
```
## 9          0.0599  roc_auc hand_till  0.622     5 0.0188  Preprocessor1_Model09
## 10         0.1     roc_auc hand_till  0.510     5 0.00964 Preprocessor1_Model10
```

Our best performing pruned decision tree has a roc_auc of 0.6399.

Question 5

```r
best_tree <- select_best(tune_tree_res, metric = 'roc_auc')

tree_final <- finalize_workflow(tree_wf, best_tree)

tree_final_fit <- fit(tree_final, data = data_train)

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

```
## Warning: Cannot retrieve the data used to build the model (model.frame: object '..y' not found).
## To silence this warning:
##     Call rpart.plot with roundint=FALSE,
##     or rebuild the rpart model with model=TRUE.
```



Question 6

```
bagging_spec <- rand_forest() %>%
  set_engine("ranger", importance = 'impurity') %>%
  set_mode("classification")

rf_wf <- workflow() %>%
  add_model(bagging_spec %>% set_args(mtry = tune(), trees = tune(), min_n = tune(), importance = 'impu
  add_formula(type_1 ~ legendary +
        generation +
        sp_atk +
        attack +
        speed +
        defense +
        hp +
        sp_def)

rf_param <- grid_regular(mtry(range = c(1,8)), trees(range = c(1,2000)), min_n(range = c(10,30)),levels
```

The mtry hyperparameter is the number of variables that are randomly sampled at each split. It chooses a number of predictors per split.

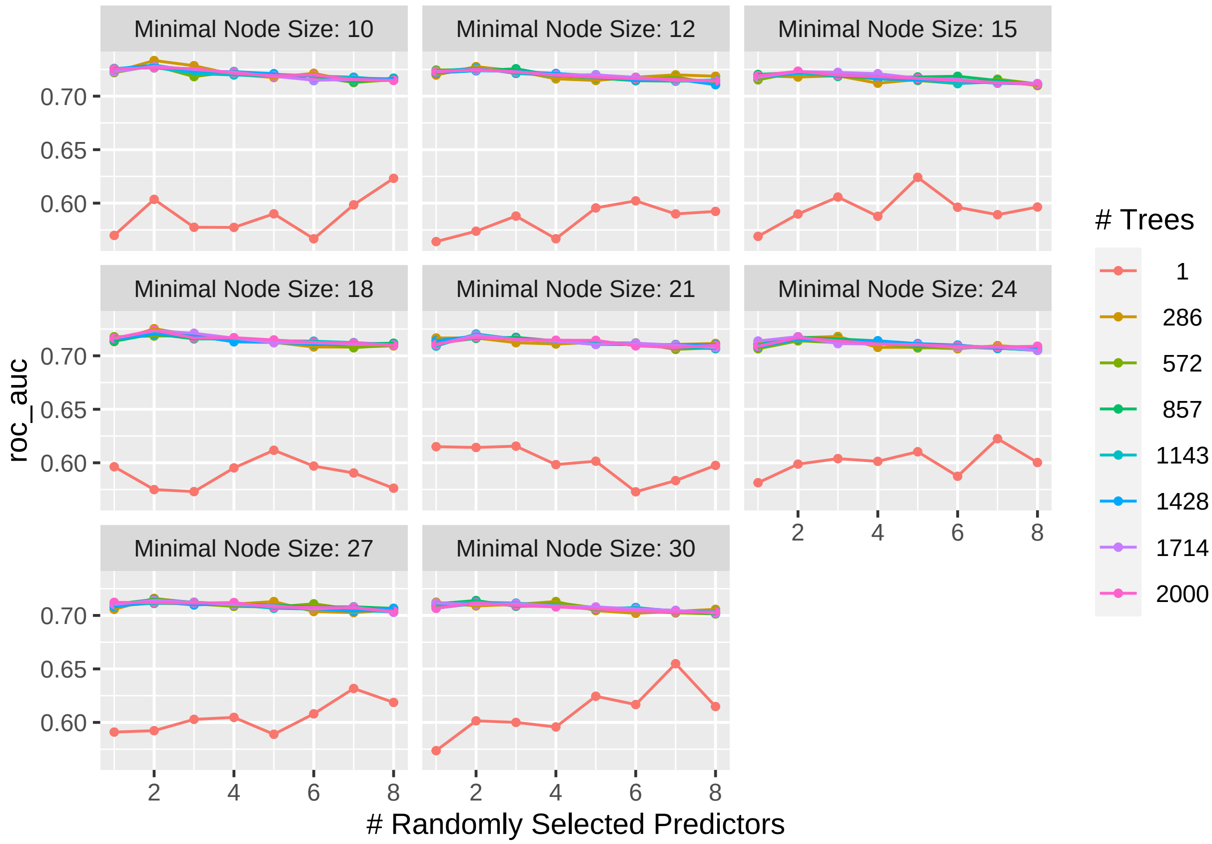
The trees hyperparameter indicates how many trees to grow. The number shouldn't be small so that every input can get predicted at least a couple of times

The n_min hyperparameter indicates the minimum number of inputs before a node can split.

Our mtry value can not be smaller than 1 nor bigger than 8 because if it is smaller than 1, we will have 0 predictors in each split which defeats the whole purpose of what we are doing. If we have more than 8, this is impossible as we can not use more predictors than given. Choosing a value smaller than 1 or bigger than 8 does not make sense. A model with mtry = 8 is a bagging model.

Question 7

```
tune_rf_res <- tune_grid(rf_wf, resamples = data_train_fold, grid = rf_param, metrics = metric_set(roc_



autoplot(tune_rf_res)
```



It appears that a higher number of trees and lower number of minimal node size leads to a higher roc_auc. Overall though a single tree is usually the worst performer. However, after a certain number of trees, adding more seems to yield similar results.

Question 8

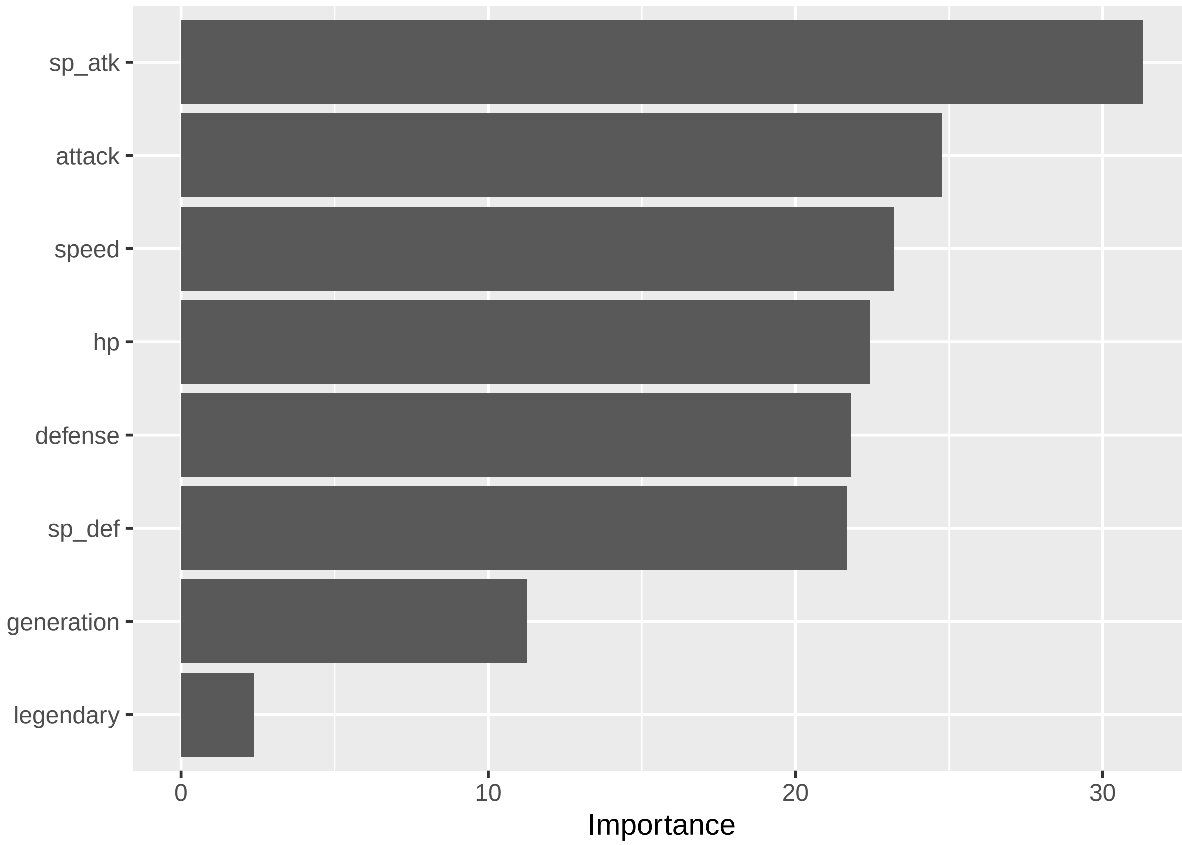
```
rf_metrics <- collect_metrics(tune_rf_res) %>% dplyr::arrange(desc(mean))

rf_metrics
```

```
## # 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     2   286    10 roc_auc hand_till  0.733     5  0.0122 Preprocessor1_Model~
##  2     2  1428    10 roc_auc hand_till  0.729     5  0.0131 Preprocessor1_Model~
##  3     2   572    10 roc_auc hand_till  0.729     5  0.0142 Preprocessor1_Model~
##  4     3   286    10 roc_auc hand_till  0.728     5  0.0136 Preprocessor1_Model~
##  5     2  1714    10 roc_auc hand_till  0.728     5  0.0136 Preprocessor1_Model~
##  6     2   286    12 roc_auc hand_till  0.728     5  0.0161 Preprocessor1_Model~
##  7     2  1143    10 roc_auc hand_till  0.727     5  0.0144 Preprocessor1_Model~
##  8     2  2000    10 roc_auc hand_till  0.727     5  0.0142 Preprocessor1_Model~
##  9     2   857    10 roc_auc hand_till  0.726     5  0.0134 Preprocessor1_Model~
## 10     2  1143    12 roc_auc hand_till  0.726     5  0.0133 Preprocessor1_Model~
## # ... with 502 more rows
```

Our best performing model had a roc_auc of 0.729 and was a combination of low minimal node size and a high number of trees.

Question 9

```
best_rf <- select_best(tune_rf_res, metric = 'roc_auc')



rf_final <- finalize_workflow(rf_wf, best_rf)

rf_final_fit <- fit(rf_final, data = data_train)

vip::vip(rf_final_fit%>% extract_fit_parsnip())
```

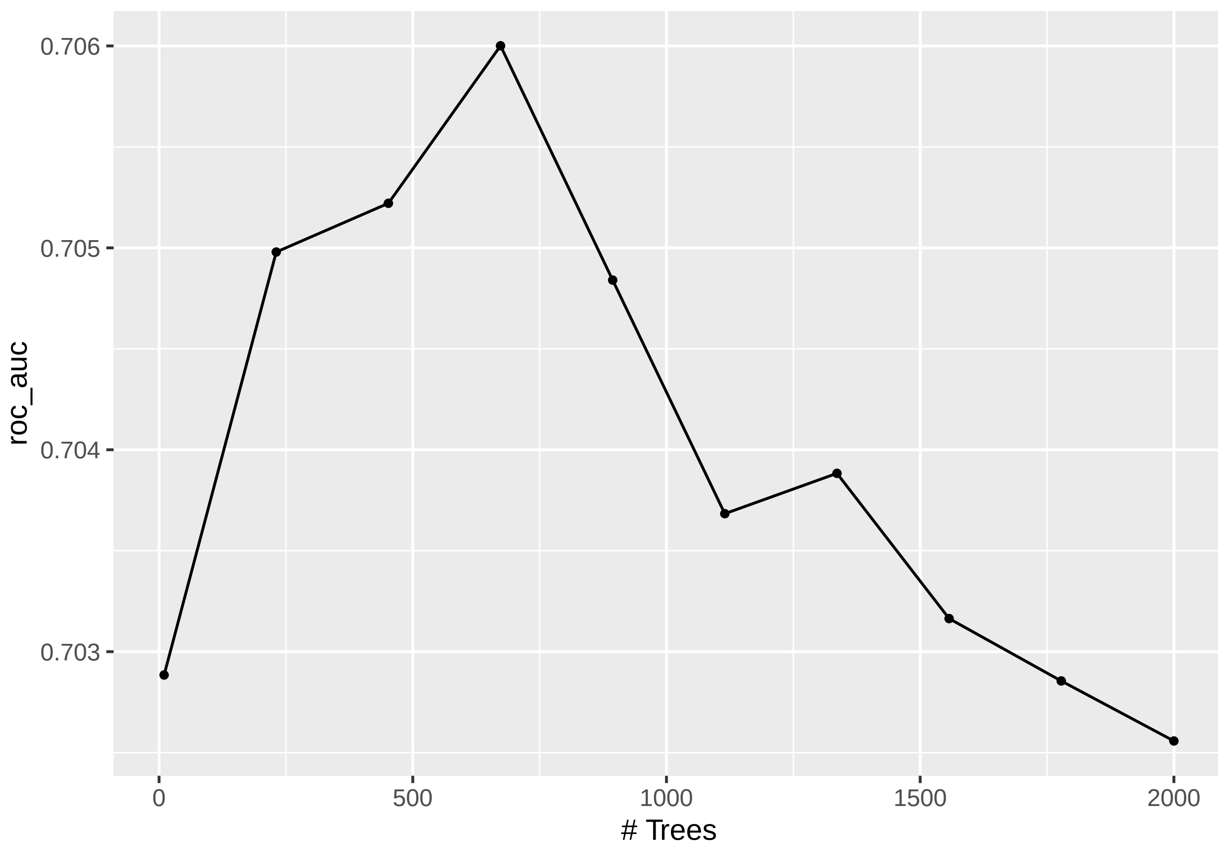


From our vip graph it appears that sp_atk and attack were the 2 most useful predictors. I didn't expect this, but it makes sense as attack is one of the primary attributes and all types of pokemon must have a certain range of both. Legendary was by far the worst predictor which makes sense as just because if a pokemon is a legendary type and it has higher than normal attributes of its type, it may be hard to predict if it is legendary or if it is a certain type of pokemon with similar statistics. If they do have normal attributes then being legendary wouldn't necessarily help predict what type of pokemon it is.

Question 10

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

```r
boost_wf <- workflow() %>%
  add_model(boost_spec) %>%
  add_recipe(data_rec)

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

tune_boost_res <- tune_grid(boost_wf, resamples = data_train_fold, grid = boost_param, metrics = metric

autoplot(tune_boost_res)
```




```r
boost_metrics <- collect_metrics(tune_boost_res) %>% dplyr::arrange(-mean)

boost_metrics
```

```
## # A tibble: 10 x 7
##    trees .metric .estimator  mean     n std_err .config
##    <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1   673 roc_auc hand_till  0.706     5  0.0151 Preprocessor1_Model04
##  2   452 roc_auc hand_till  0.705     5  0.0144 Preprocessor1_Model03
##  3   231 roc_auc hand_till  0.705     5  0.0139 Preprocessor1_Model02
##  4   894 roc_auc hand_till  0.705     5  0.0149 Preprocessor1_Model05
##  5  1336 roc_auc hand_till  0.704     5  0.0157 Preprocessor1_Model07
##  6  1115 roc_auc hand_till  0.704     5  0.0152 Preprocessor1_Model06
##  7  1557 roc_auc hand_till  0.703     5  0.0155 Preprocessor1_Model08
##  8    10 roc_auc hand_till  0.703     5  0.0138 Preprocessor1_Model01
```

```
## 9 1778 roc_auc hand_till 0.703 5 0.0155 Preprocessor1_Model09
## 10 2000 roc_auc hand_till 0.703 5 0.0157 Preprocessor1_Model10
```

I observed from our roc_auc graph that it peaked around 725 trees before slowly going down and then reaching its lowest value at 2000 trees. Thus adding more trees only helps up to a certain point. Our roc_auc at its peak is 0.706 Question 11

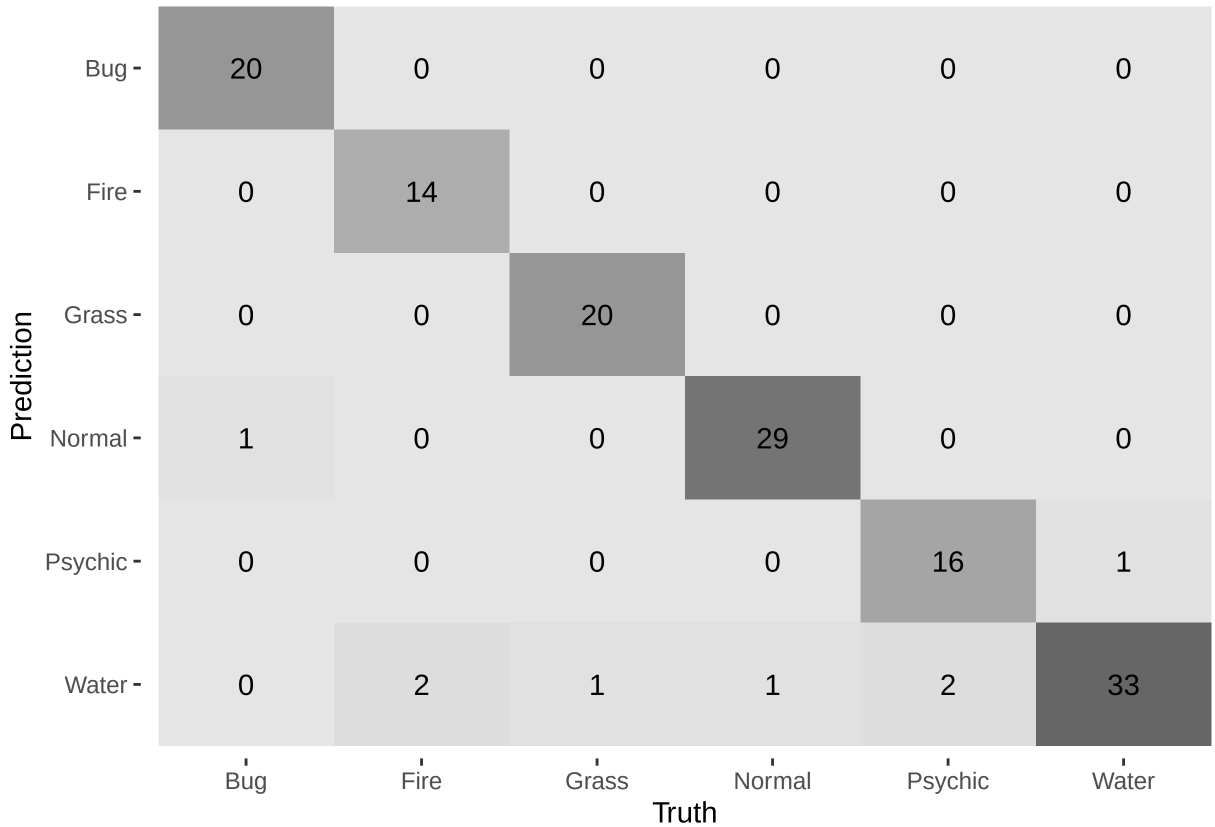
```
models <- c('Decision Tree', 'Random Forest Model', 'Boosted Tree Model')
metric <- c('roc_auc', 'roc_auc', 'roc_auc')
vals <- c(tree_metrics[1,4], rf_metrics[1,6], boost_metrics[1,4])
x <- cbind(models,metric,vals)
x
```

```
##      models                metric    vals
## mean "Decision Tree"       "roc_auc" 0.6399609
## mean "Random Forest Model" "roc_auc" 0.7332956
## mean "Boosted Tree Model"  "roc_auc" 0.7060009
```

```
best_final_vals <- best_rf

best_final <- finalize_workflow(rf_wf, best_final_vals)

best_fit <- fit(best_final, data = data_test)


augment(best_fit, new_data = data_test) %>%
roc_auc(type_1, .pred_Bug:.pred_Water)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.998
```

```
augment(best_fit, new_data = data_test) %>%
  conf_mat(type_1, estimate = .pred_class) %>%
  autoplot(type = 'heatmap')
```



The random forest model performed the best. On the testing set it performed extremely well. It predicted almost every pokemon type accurately, and mispredicted psychic types the worst. There may be some overlap of attributes of pokemon of water and psychic type on the edge which led to them being mis predicted.

Question 12

```
new_data <- read.csv('C:/Users/peter/Desktop/hw6/abalone.csv')
new_data$age <- new_data$rings + 1.5

set.seed(100)

ab_split <- initial_split(new_data, prop = 0.8, strata = age)
ab_train <- training(ab_split)
ab_test <- testing(ab_split)

ab_train_recipe <- recipe(age~ type + diameter + height + whole_weight + shucked_weight + viscera_weigh
  step_dummy(all_nominal_predictors()) %>%
  step_interact(terms = ~ starts_with('type'):shucked_weight) %>%
  step_interact(terms = ~ shell_weight:shucked_weight) %>%
  step_interact(terms = ~ longest_shell:diameter) %>%
  step_center(all_predictors()) %>%
  step_scale(all_predictors())

ab_train_fold <- vfold_cv(ab_train, v = 5, strata = age)

rf_ab_spec <- rand_forest() %>%
  set_engine('randomForest', importance = TRUE) %>%
```

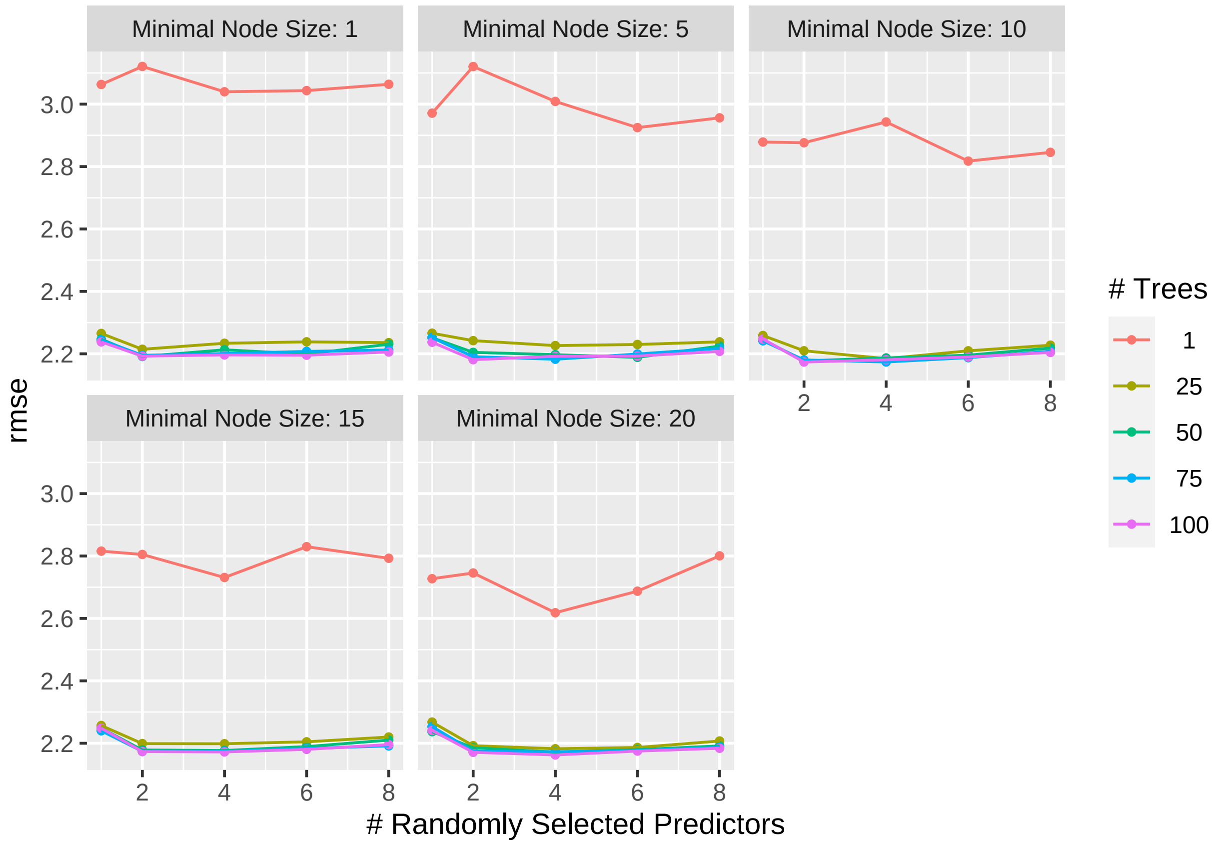
```r
  set_mode('regression')

rf_ab_wf <- workflow() %>%
  add_model(rf_ab_spec %>% set_args(mtry = tune(), trees = tune(), min_n = tune())) %>%
  add_formula(age~ type +
                diameter +
                height +
                whole_weight +
                shucked_weight +
                viscera_weight +
                shell_weight +
                longest_shell)

rf_ab_param <- grid_regular(mtry(range = c(1,8)), trees(range = c(1,100)), min_n(range = c(1,20)),levels

tune_rf_ab_res <- tune_grid(rf_ab_wf, resamples = ab_train_fold, grid = rf_ab_param, metrics = metric_se

rf_ab_metrics <- collect_metrics(tune_rf_ab_res) %>% dplyr::arrange(mean)

rf_ab_metrics
```

```
## # A tibble: 125 x 9
##     mtry trees min_n .metric .estimator  mean     n std_err .config
##    <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1     4   100    20 rmse    standard    2.16     5  0.0245 Preprocessor1_Model~
##  2     2   100    20 rmse    standard    2.17     5  0.0229 Preprocessor1_Model~
##  3     4    50    20 rmse    standard    2.17     5  0.0231 Preprocessor1_Model~
##  4     4   100    15 rmse    standard    2.17     5  0.0244 Preprocessor1_Model~
##  5     4    75    20 rmse    standard    2.17     5  0.0255 Preprocessor1_Model~
##  6     2   100    15 rmse    standard    2.17     5  0.0196 Preprocessor1_Model~
##  7     2   100    10 rmse    standard    2.17     5  0.0204 Preprocessor1_Model~
##  8     4    75    10 rmse    standard    2.17     5  0.0275 Preprocessor1_Model~
##  9     2    75    15 rmse    standard    2.17     5  0.0236 Preprocessor1_Model~
## 10     4    75    15 rmse    standard    2.17     5  0.0281 Preprocessor1_Model~
## # ... with 115 more rows
```

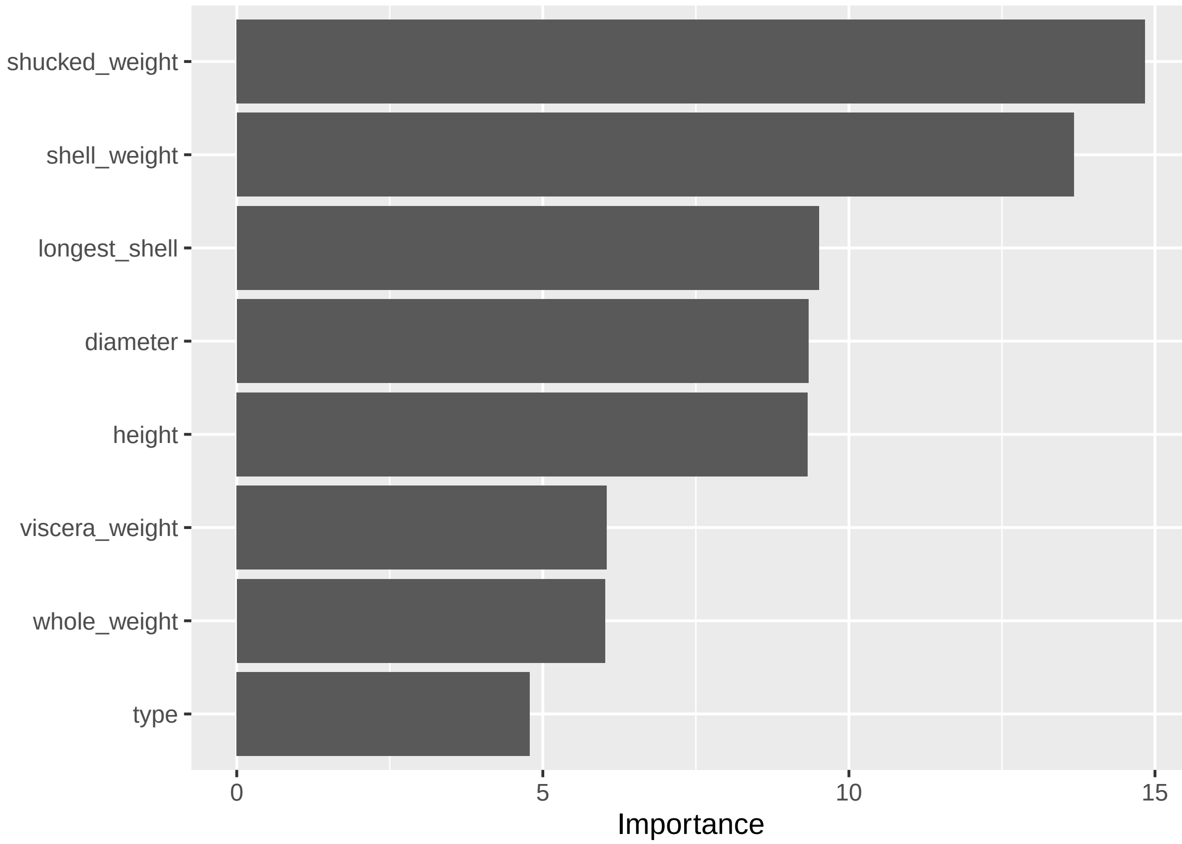
```r
best_rf_ab_vals <- select_best(tune_rf_ab_res, metric = 'rmse')

final_rf_ab <- finalize_workflow(rf_ab_wf, best_rf_ab_vals)

final_rf_ab_fit <- fit(final_rf_ab, data = ab_test)

augment(final_rf_ab_fit, new_data = ab_test) %>%
  rmse(age, .pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        1.48
```

```r
autoplot(tune_rf_ab_res)
```




```
vip::vip(final_rf_ab_fit %>% extract_fit_parsnip())
```



Our RMSE was 1.5 which shows that on average our prediction is somewhat close to the actual value. This is a similar value to what we found in Homework 2. However, this may be because of the range of values I chose for the hyperparameters. Generally from our autoplot, the more minimal node size the lower RMSE value we get. Therefore, increasing this could lead to a RMSE lower than 1.5. It also appears that the more trees we put, the lower RMSE we get when compared to just using 1 tree. In addition, our vip graph shows us that the shucked_weight and shell_weight of the eggs are the most important predictors. Therefore manipulating how many predictors we use. number of trees, and minimal node size, we can possibly achieve a very good / lower RMSE value.