Tree-Based Models

For 231 Students

Homework 6

Code

PSTAT 131/231

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 (https://www.kaggle.com/abcsds/pokemon).

The Pokémon (https://www.pokemon.com/us/) 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 (https://bulbapedia.bulbagarden.net/wiki/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.

Fig 1. Houndoom, a Dark/Fire-type canine Pokémon from Generation II.

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.

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.

Hide

```
library(ggplot2)
library(dbplyr)
library(tidyverse)
library(tidymodels)
library(corrr)
library(corrplot)
library(ISLR) # For the Smarket data set
library(ISLR2) # For the Bikeshare data set
library(discrim)
library(glmnet)
tidymodels_prefer()
library(pROC)
library(boot)
library(rsample)
```

Hide

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

Hide

```
pokemon_data <- read.csv("pokemon.csv")
pokemon <- clean_names(pokemon_data)
```

Hide

```
pokemon1 <- pokemon %>% filter(type_1 %in% c("Bug","Fire", "Grass", "Normal", "Water","Psychic"))

pokemon1$type_1 <- factor(pokemon1$type_1)
pokemon1$legendary <- factor(pokemon1$legendary)
```

Hide

```
pokemon1_split <- initial_split(pokemon1,prop=0.80,strata=type_1)
pokemon1_training <- training(pokemon1_split)
pokemon1_testing <- testing(pokemon1_split)
dim(pokemon1_testing)
```

```
## [1] 94 13
```

Hide

```
dim(pokemon1_training)
```

```
## [1] 364  13
```

Hide

```
pokemon_folds <- vfold_cv(pokemon1_training,v=5, strata=type_1)
pokemon_folds
```

```
## #  5-fold cross-validation using stratification
## # A tibble: 5 x 2
##   splits           id
##   <list>           <chr>
## 1 <split [289/75]> Fold1
## 2 <split [291/73]> Fold2
## 3 <split [291/73]> Fold3
## 4 <split [292/72]> Fold4
## 5 <split [293/71]> Fold5
```

Hide

```
pokemon_recipe <- recipe(type_1~legendary+generation+sp_atk+attack+speed+defense+hp+sp_def,
                         data=pokemon1_training) %>% step_dummy(legendary,generation) %>% step_normalize(all_predictors())

pokemon_recipe %>% prep() %>% bake(pokemon1_training)
```

```
## # A tibble: 364 x 13
##    sp_atk attack   speed defense     hp sp_def type_1 legendary_True
##     <dbl>  <dbl>   <dbl>   <dbl>  <dbl>  <dbl> <fct>           <dbl>
##  1 -1.61  -1.38  -0.792   -1.19  -0.951 -1.80  Bug            -0.277
##  2 -1.45  -1.68  -1.29    -0.494 -0.763 -1.63  Bug            -0.277
##  3  0.541 -0.929  0.0404  -0.668 -0.388  0.299 Bug            -0.277
##  4 -1.61  -1.23  -0.625   -1.37  -1.14  -1.80  Bug            -0.277
##  5 -1.45  -1.53  -1.12    -0.668 -0.951 -1.63  Bug            -0.277
##  6 -0.839  0.425  0.207   -1.02  -0.200  0.299 Bug            -0.277
##  7 -1.76   2.23   2.54    -1.02  -0.200  0.299 Bug            -0.277
##  8 -0.379  0.576 -1.29     0.379 -0.388  0.299 Bug            -0.277
##  9 -0.992 -0.628 -0.792   -0.668 -0.388 -0.576 Bug            -0.277
## 10 -0.225  2.38   1.21     1.77  -0.200  0.649 Bug            -0.277
## # ... with 354 more rows, and 5 more variables: generation_X2 <dbl>,
## #   generation_X3 <dbl>, generation_X4 <dbl>, generation_X5 <dbl>,
## #   generation_X6 <dbl>
```

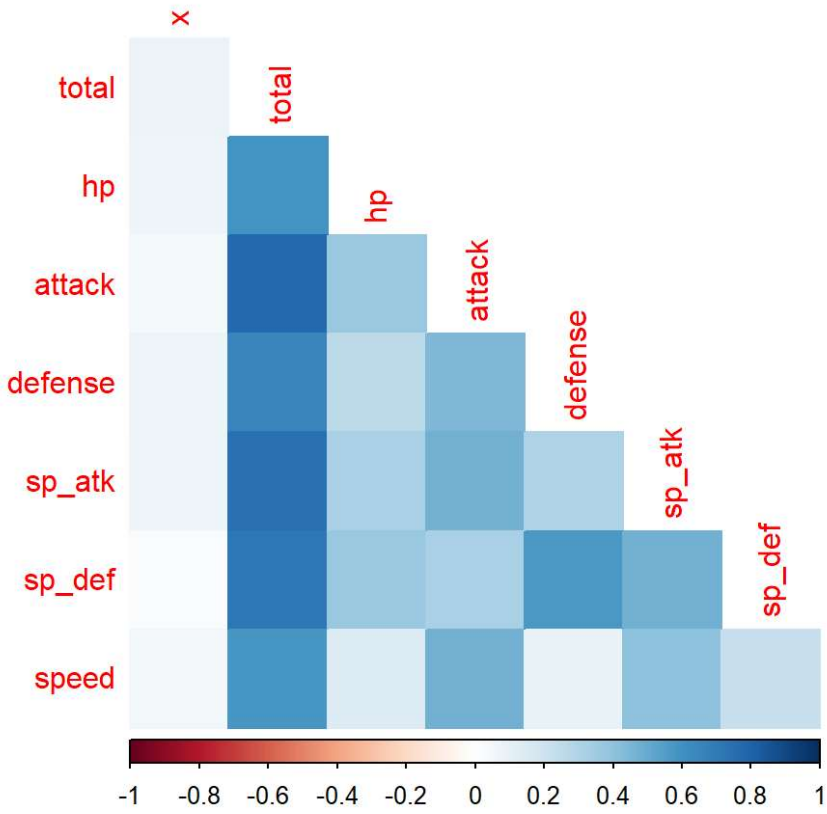
Exercise 2

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?

Hide

```
pokemon1_training %>% select(is.numeric) %>%
  cor() %>%
  corrplot(type="lower", diag=FALSE, method="color")
```



Attack, defense, sp_atk, sp_def, hp and speed have relatively high positive correlation with “total”. This makes sense since “total” is determined by the sum of all these characteristics of pokemon. There is also a relatively strong correlation with sp_def, which makes sense since it is likely that a pokemon’s defensive ability is similar for all kind of attacks.

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?

Hide

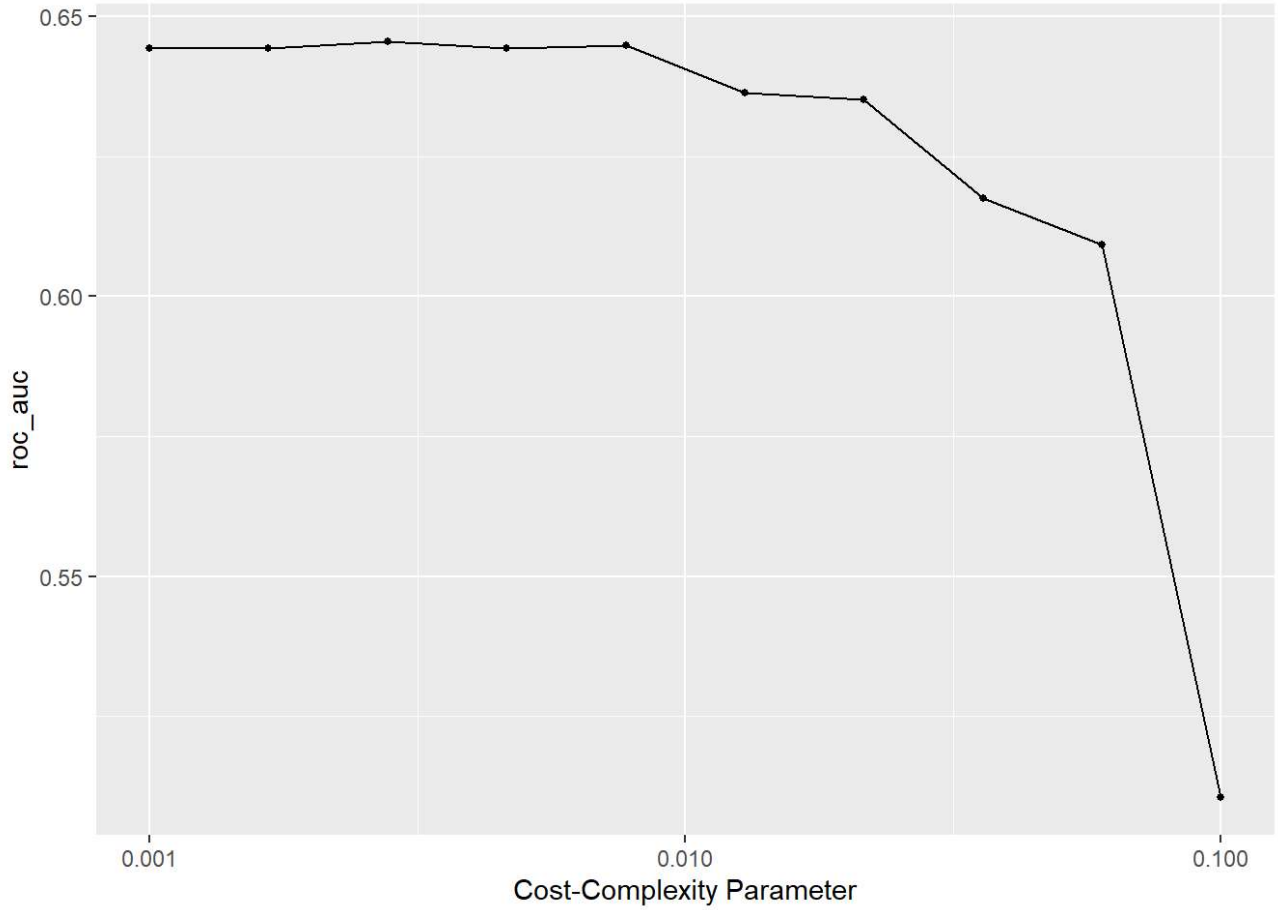
```
tree_spec <- decision_tree() %>%
  set_engine("rpart")
class_tree_spec <- tree_spec %>%
  set_mode("classification")
class_tree_wf <- workflow() %>%
  add_model(class_tree_spec %>% set_args(cost_complexity = tune())) %>%
  add_recipe(pokemon_recipe)
param_grid <- grid_regular(cost_complexity(range = c(-3, -1)), levels = 10)
```

Hide

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

Hide

```
autoplot(tune_res)
```



The roc_auc value is high for smaller cost-complexity parameter between 0.001-0.01. It decreases to 0 when cost-complexity is 0.1.

Exercise 4

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

Hide

```
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         0.00278 roc_auc hand_till  0.646     5 0.00868 Preprocessor1_Model03
##  2         0.00774 roc_auc hand_till  0.645     5 0.0154  Preprocessor1_Model05
##  3         0.001   roc_auc hand_till  0.644     5 0.00909 Preprocessor1_Model01
##  4         0.00167 roc_auc hand_till  0.644     5 0.00909 Preprocessor1_Model02
##  5         0.00464 roc_auc hand_till  0.644     5 0.0106  Preprocessor1_Model04
##  6         0.0129  roc_auc hand_till  0.636     5 0.00868 Preprocessor1_Model06
##  7         0.0215  roc_auc hand_till  0.635     5 0.00708 Preprocessor1_Model07
##  8         0.0359  roc_auc hand_till  0.618     5 0.0221  Preprocessor1_Model08
##  9         0.0599  roc_auc hand_till  0.609     5 0.0182  Preprocessor1_Model09
## 10         0.1     roc_auc hand_till  0.510     5 0.0105  Preprocessor1_Model10
```

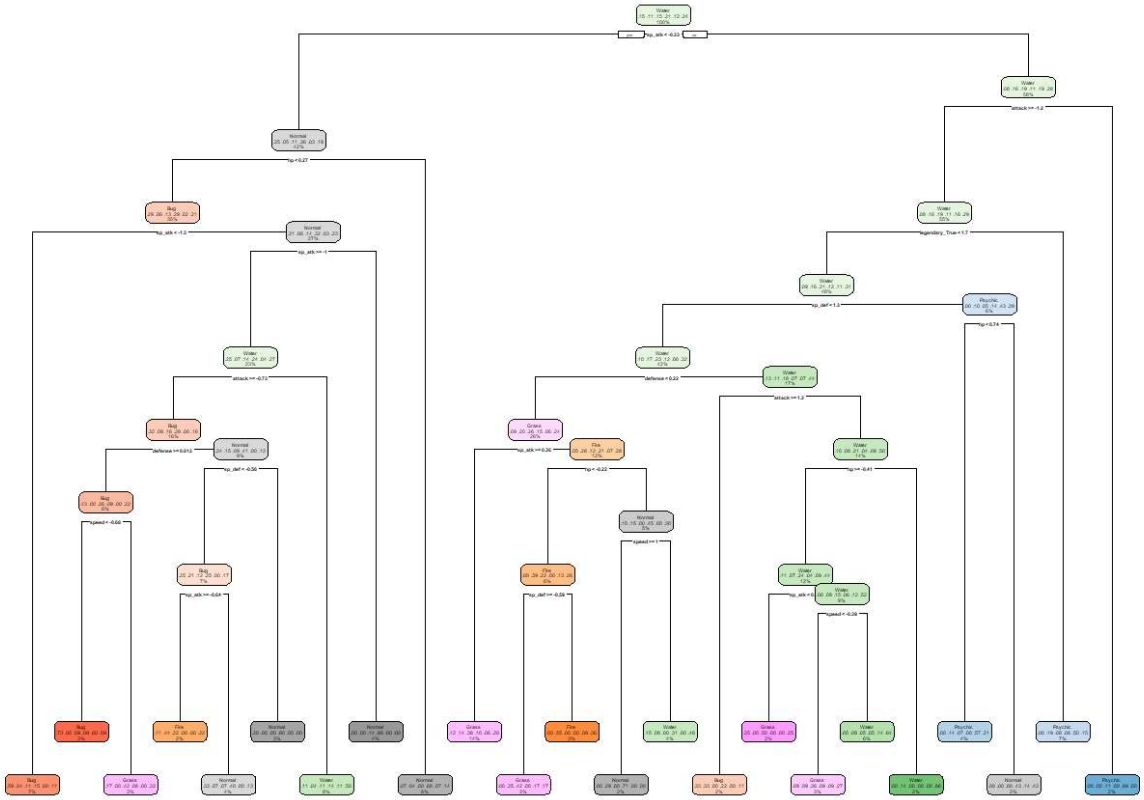
roc_aoc of best-performing decision tree is 0.6472.

Exercise 5

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

Hide

```
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 = pokemon1_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?**

Hide

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

“mtry” represents how many variables as a subset of all predicters will be considered at each split in decision trees. “trees” represents the number of trees that we build. “min_n” represents the number of data points in the leaves. It indicate when the tree stops split.

Hide

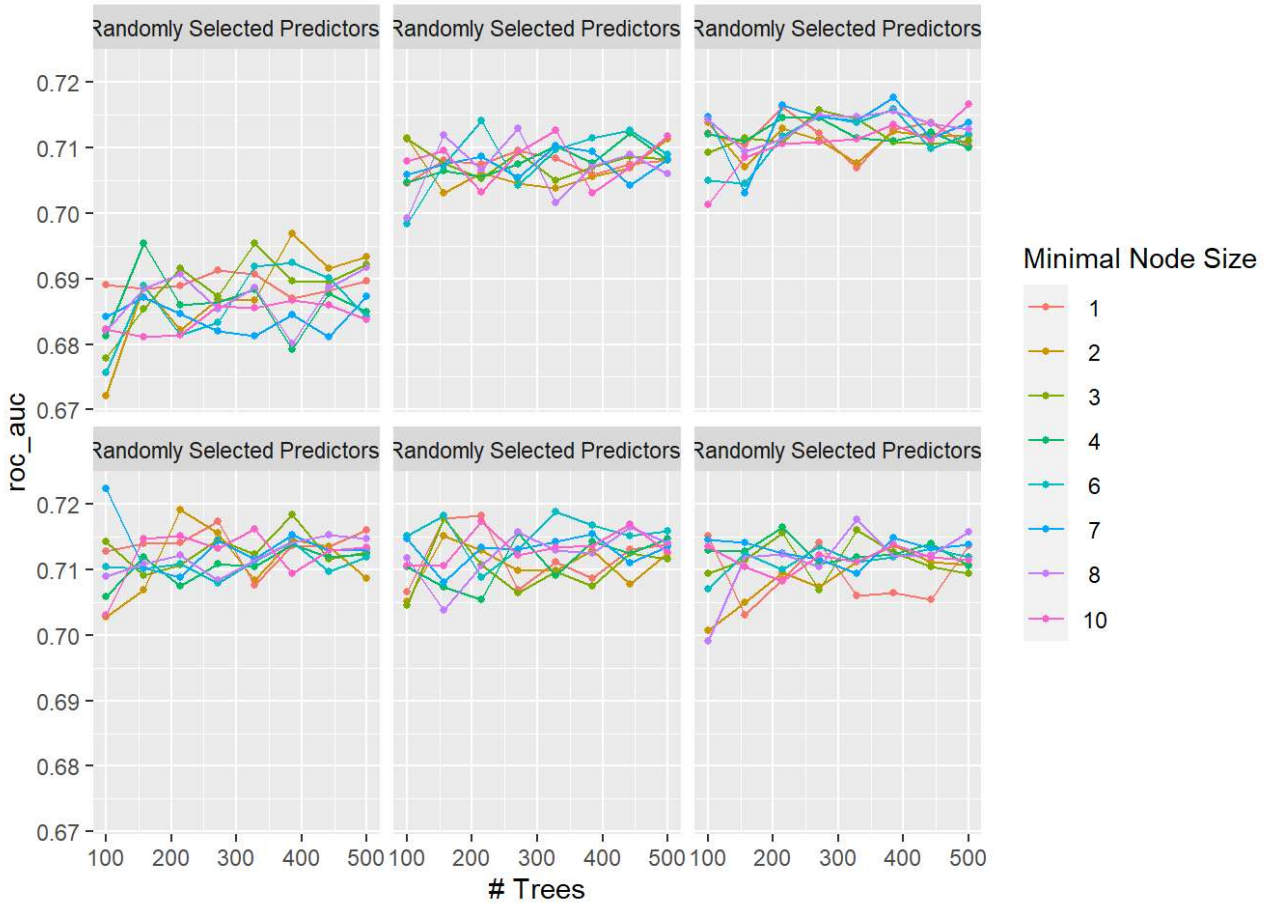
```
reg_grid <- grid_regular(mtry(range = c(1, 6)), trees(range=c(100,500)), min_n(range=c(1,
10)),levels = 8)
```

For classification model, mtry is $\sqrt{p}$ where p is the total number of variables. Here, $p $ is about 3. There fore I choose the range to be 1-6. ### 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?

Hide

```
tune_res2 <- tune_grid(
  rf_wf,
  resamples = pokemon_folds,
  grid = reg_grid,
  metrics = metric_set(roc_auc)
)
autoplot(tune_res2)
```



It seems that trees is about 125 and min_n=1 with mtry=4 yields the best result.

Exercise 7

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

Hide

```
collect_metrics(tune_res2) %>% arrange(desc(mean))
```

```
## # A tibble: 384 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     7 roc_auc hand_till  0.722     5  0.0149 Preprocessor1_Model~
##  2     4   214     2 roc_auc hand_till  0.719     5  0.0135 Preprocessor1_Model~
##  3     5   328     6 roc_auc hand_till  0.719     5  0.0164 Preprocessor1_Model~
##  4     4   385     3 roc_auc hand_till  0.718     5  0.0168 Preprocessor1_Model~
##  5     5   157     6 roc_auc hand_till  0.718     5  0.0128 Preprocessor1_Model~
##  6     5   214     1 roc_auc hand_till  0.718     5  0.0148 Preprocessor1_Model~
##  7     5   157     1 roc_auc hand_till  0.718     5  0.0168 Preprocessor1_Model~
##  8     5   157     3 roc_auc hand_till  0.718     5  0.0156 Preprocessor1_Model~
##  9     6   328     8 roc_auc hand_till  0.718     5  0.0174 Preprocessor1_Model~
## 10     3   385     7 roc_auc hand_till  0.718     5  0.0122 Preprocessor1_Model~
## # ... with 374 more rows
```

The roc_auc of the best performing random forest model is 0.7318639.

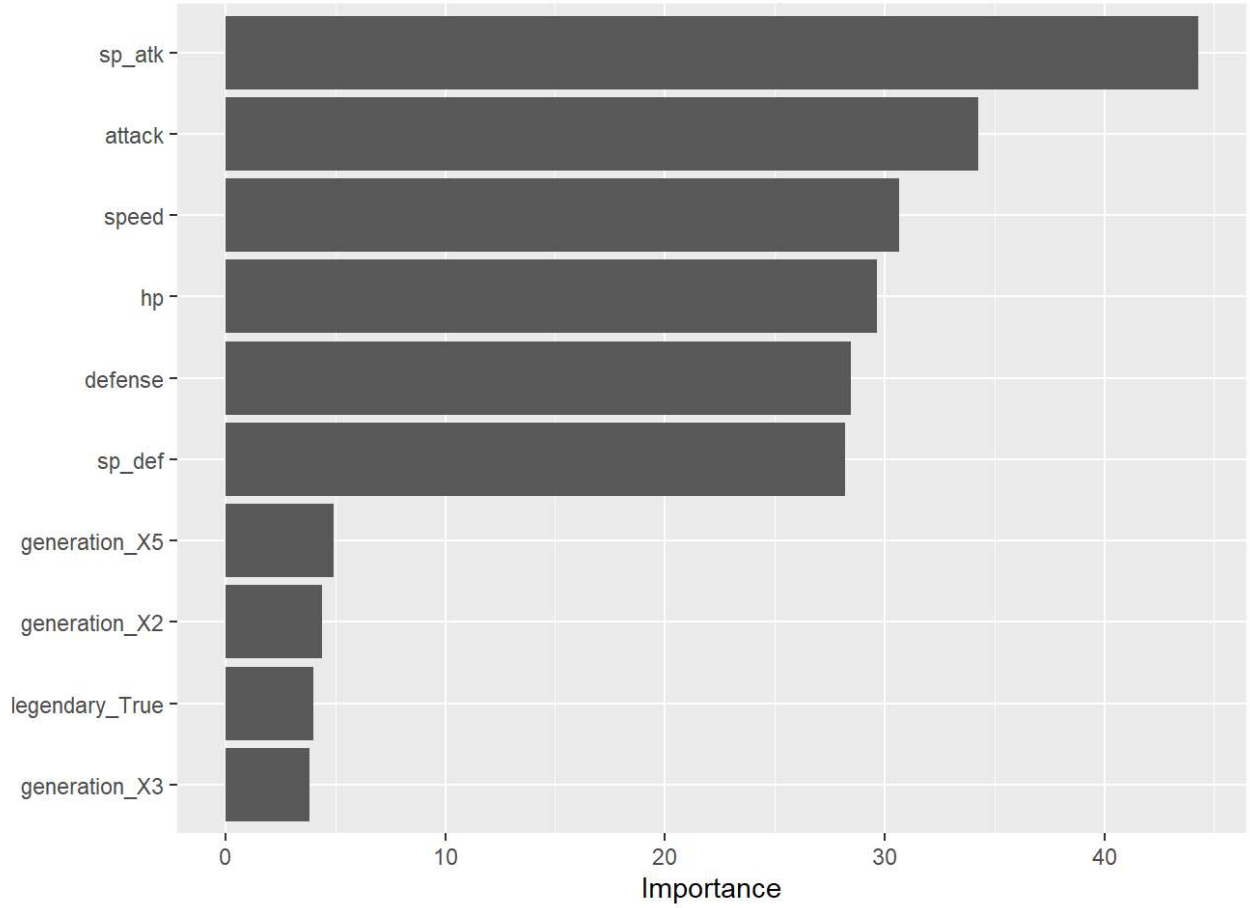
Exercise 8

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?

Hide

```
best_penalty2 <- select_best(tune_res2,metric="roc_auc")
rf_final <- finalize_workflow(rf_wf, best_penalty2)

rf_fit <- fit(rf_final, data = pokemon1_training)
vip(rf_fit%>% extract_fit_engine())
```



special attack is the most useful. Generation is the least useful. It makes sense since the ability to attack others reflects the pokemon's specific ability and characteristics. Generation is the least useful, since the generation cannot reflect the pokemon's unique characteristics.

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?

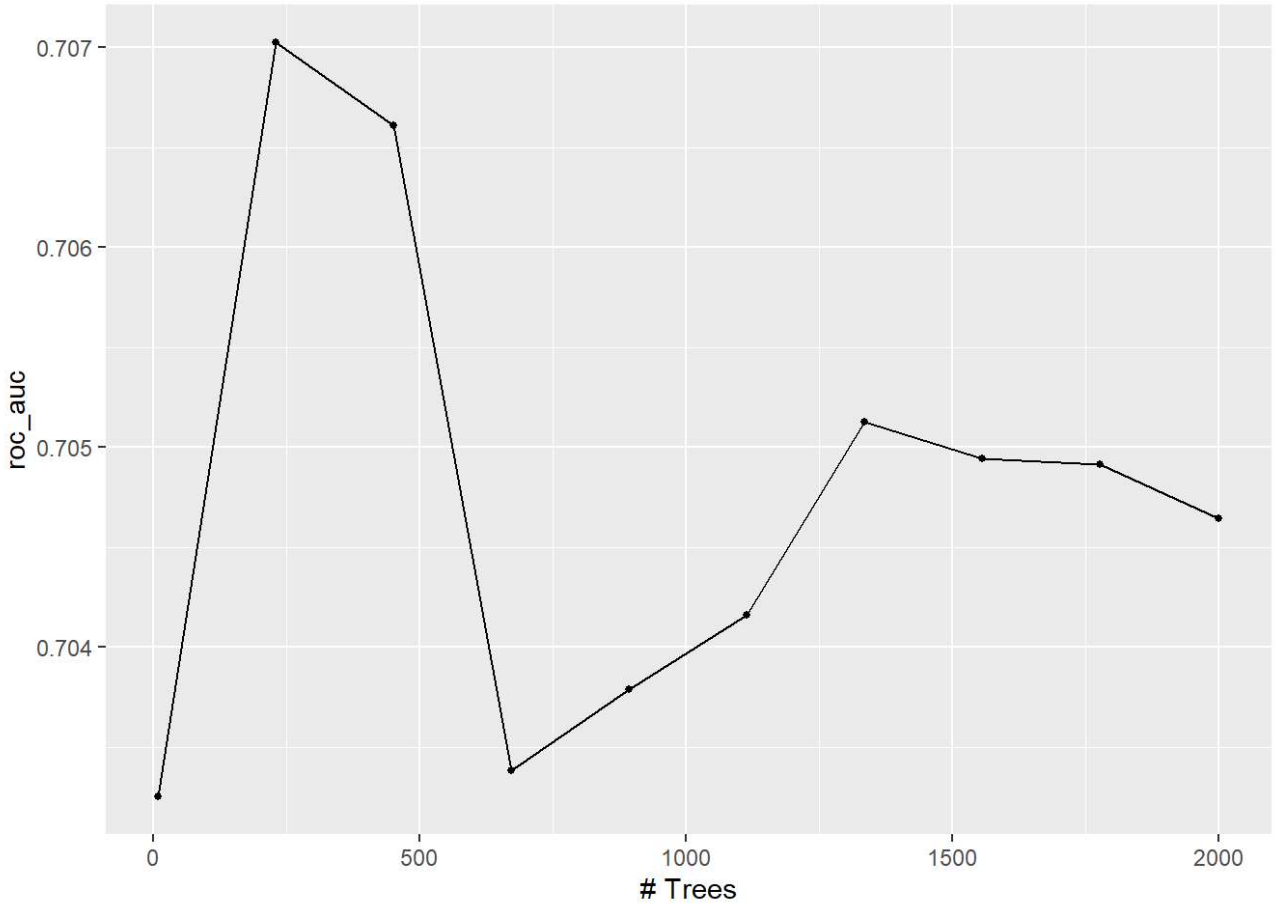
What is the `roc_auc` of your best-performing boosted tree model on the folds? *Hint: Use* `collect_metrics()` *and* `arrange()`.

Hide

```
boost_spec <- boost_tree(trees = tune()) %>%
  set_engine("xgboost") %>%
  set_mode("classification")
boost_wf <- workflow() %>%
  add_model(boost_spec) %>%
  add_recipe(pokemon_recipe)
boost_grid <- grid_regular(trees(range=c(10,2000)),levels = 10)
```

Hide

```
tune_res3 <- tune_grid(
  boost_wf,
  resamples = pokemon_folds,
  grid = boost_grid,
  metrics = metric_set(roc_auc)
)
autoplot(tune_res3)
```



The roc_auc value decreases as number of tree increase, it stays relatively low and does not change much between 500 to 1000.

Hide

```
collect_metrics(tune_res3) %>% arrange(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  0.707     5  0.0168 Preprocessor1_Model02
##  2   452 roc_auc hand_till  0.707     5  0.0183 Preprocessor1_Model03
##  3  1336 roc_auc hand_till  0.705     5  0.0178 Preprocessor1_Model07
##  4  1557 roc_auc hand_till  0.705     5  0.0177 Preprocessor1_Model08
##  5  1778 roc_auc hand_till  0.705     5  0.0175 Preprocessor1_Model09
##  6  2000 roc_auc hand_till  0.705     5  0.0175 Preprocessor1_Model10
##  7  1115 roc_auc hand_till  0.704     5  0.0183 Preprocessor1_Model06
##  8   894 roc_auc hand_till  0.704     5  0.0179 Preprocessor1_Model05
##  9   673 roc_auc hand_till  0.703     5  0.0176 Preprocessor1_Model04
## 10    10 roc_auc hand_till  0.703     5  0.0118 Preprocessor1_Model01
```

The roc_auc of the best-performing model is 0.6979390.

Hide

```
best_penalty3<- select_best(tune_res3,metric="roc_auc")
boost_final <- finalize_workflow(boost_wf, best_penalty3)
boost_fit <- fit(boost_final, data = pokemon1_training)
```

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?

Hide

```
roc_auc <- data.frame(Model=c("decision tree", "random forest", "boosted tree"),
                      rocauc = c(0.6670845, 0.7318639,0.6979390))
roc_auc
```

```
##           Model    rocauc
## 1 decision tree 0.6670845
## 2 random forest 0.7318639
## 3  boosted tree 0.6979390
```

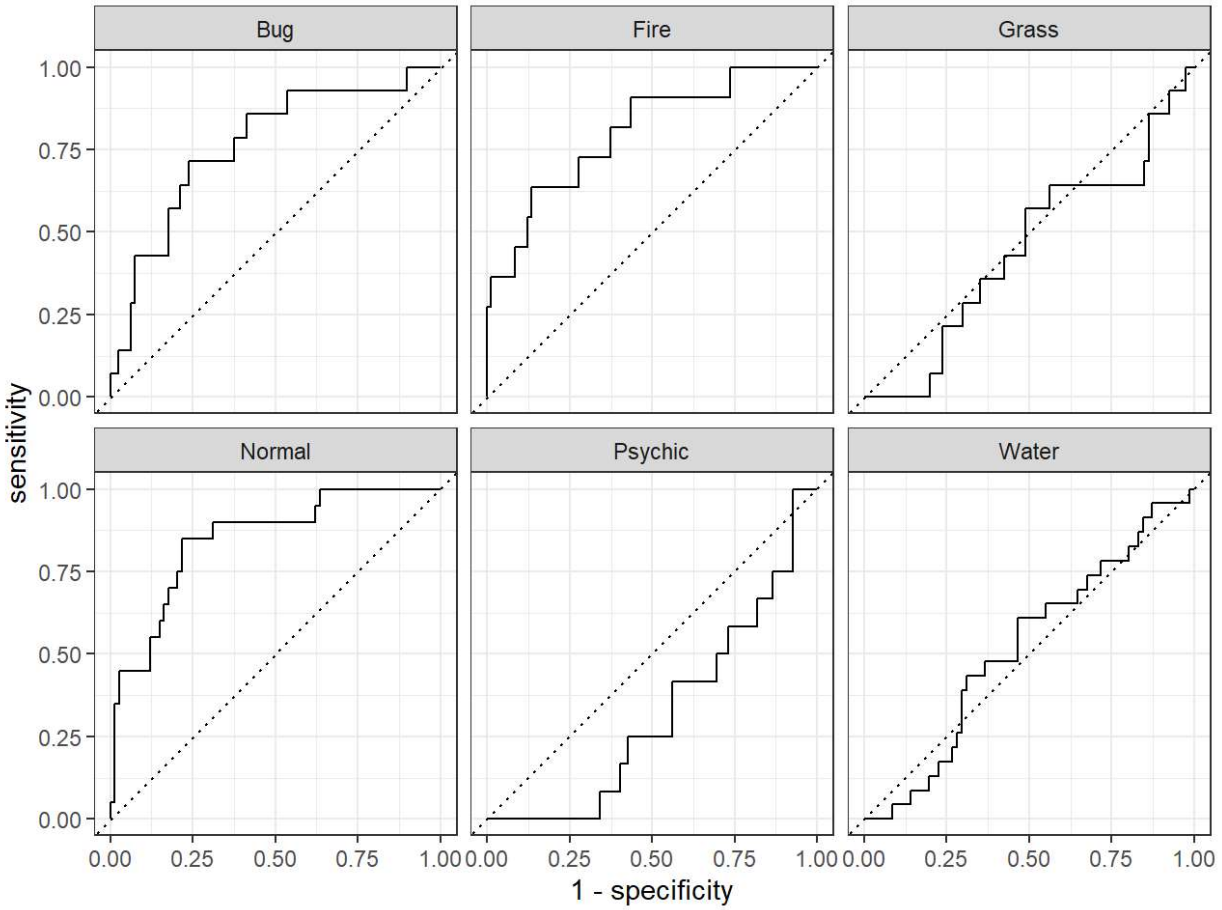
Hide

```
wk_final <- finalize_workflow(rf_final, best_penalty2)

final_fit <- fit(wk_final, data = pokemon1_training)
augment(final_fit, new_data = pokemon1_testing) %>%
   roc_auc(type_1, .pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal,.pred_Water,.pred_Psy
chic)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.610
```

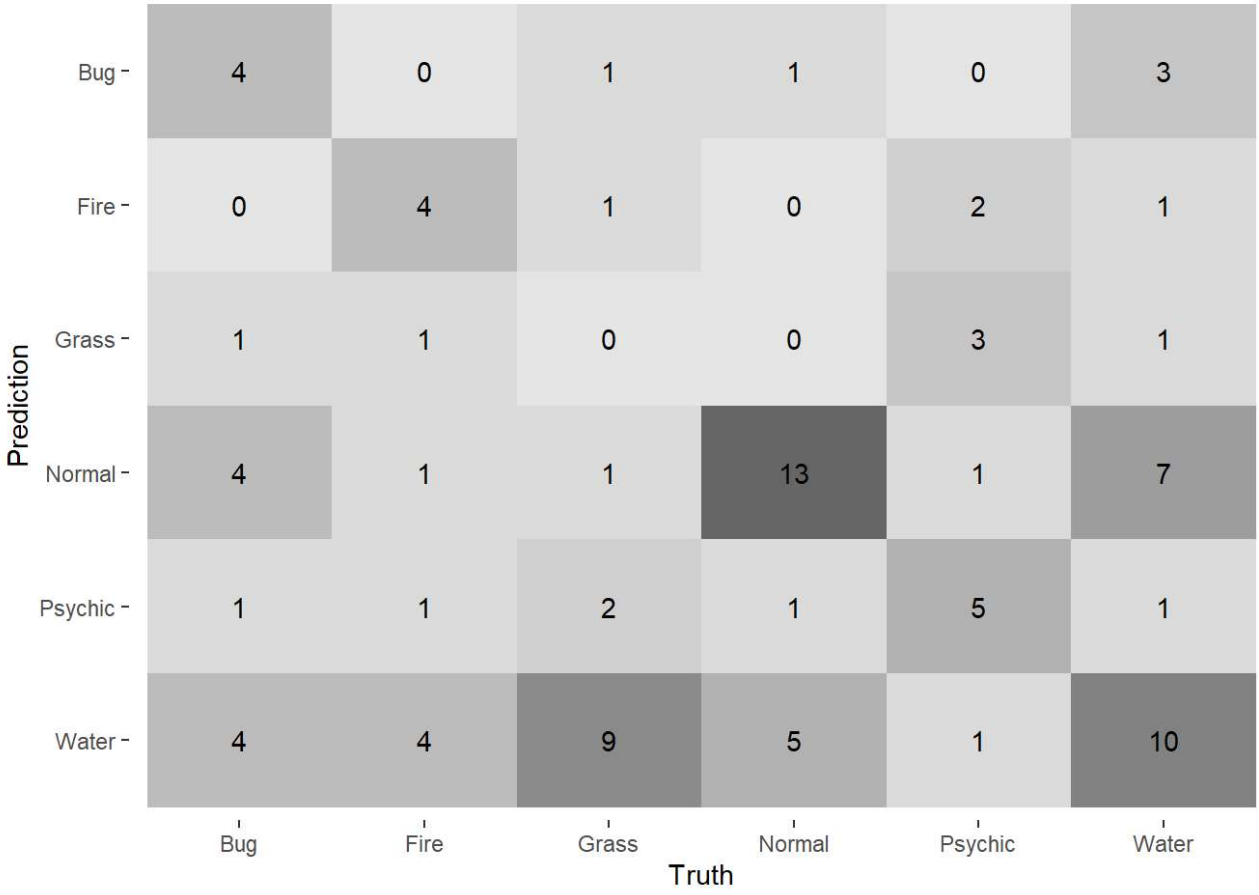
Hide

```
augment(final_fit, new_data = pokemon1_testing) %>% roc_curve(type_1, .pred_Bug, .pred_Fi
re, .pred_Grass, .pred_Normal,.pred_Water,.pred_Psychic) %>%
  autoplot()
```



Hide

```
augment(final_fit, new_data = pokemon1_testing) %>% conf_mat(truth=type_1, .pred_class) %
>%
  autoplot(type="heatmap")
```



The random forest model performs best on folds. The model predicts the most accurate for normal type, and predicts worst on grass.

For 231 Students

Exercise 11

Using the `abalone.txt` data from previous assignments, fit and tune a random forest model to predict `age`. Use stratified cross-validation and select ranges for `mtry`, `min_n`, and `trees`. Present your results. What was the model's RMSE on your testing set?

Hide

```
abalone_data <- read.csv('abalone.csv')
head(abalone_data)
```

```
##   type longest_shell diameter height whole_weight shucked_weight viscera_weight
## 1    M         0.455    0.365  0.095       0.5140         0.2245         0.1010
## 2    M         0.350    0.265  0.090       0.2255         0.0995         0.0485
## 3    F         0.530    0.420  0.135       0.6770         0.2565         0.1415
## 4    M         0.440    0.365  0.125       0.5160         0.2155         0.1140
## 5    I         0.330    0.255  0.080       0.2050         0.0895         0.0395
## 6    I         0.425    0.300  0.095       0.3515         0.1410         0.0775
##   shell_weight rings
## 1        0.150    15
## 2        0.070     7
## 3        0.210     9
## 4        0.155    10
## 5        0.055     7
## 6        0.120     8
```

Hide

```
abalone <- abalone_data %>%
  mutate(abalone_data, age=rings+1.5)
```

Hide

```
set.seed(3435)

abalone_split <- initial_split(abalone, prop = 0.80,
                               strata = age)
abalone_train <- training(abalone_split)
abalone_test <- testing(abalone_split)
abalone_folds <- vfold_cv(abalone_train,v=5, strata=age)
head(abalone_test)
```

```
##    type longest_shell diameter height whole_weight shucked_weight
## 2     M         0.350    0.265  0.090       0.2255         0.0995
## 6     I         0.425    0.300  0.095       0.3515         0.1410
## 13    M         0.490    0.380  0.135       0.5415         0.2175
## 28    M         0.590    0.445  0.140       0.9310         0.3560
## 30    M         0.575    0.425  0.140       0.8635         0.3930
## 39    F         0.575    0.445  0.135       0.8830         0.3810
##    viscera_weight shell_weight rings  age
## 2          0.0485         0.07     7  8.5
## 6          0.0775         0.12     8  9.5
## 13         0.0950         0.19    11 12.5
## 28         0.2340         0.28    12 13.5
## 30         0.2270         0.20    11 12.5
## 39         0.2035         0.26    11 12.5
```

Hide

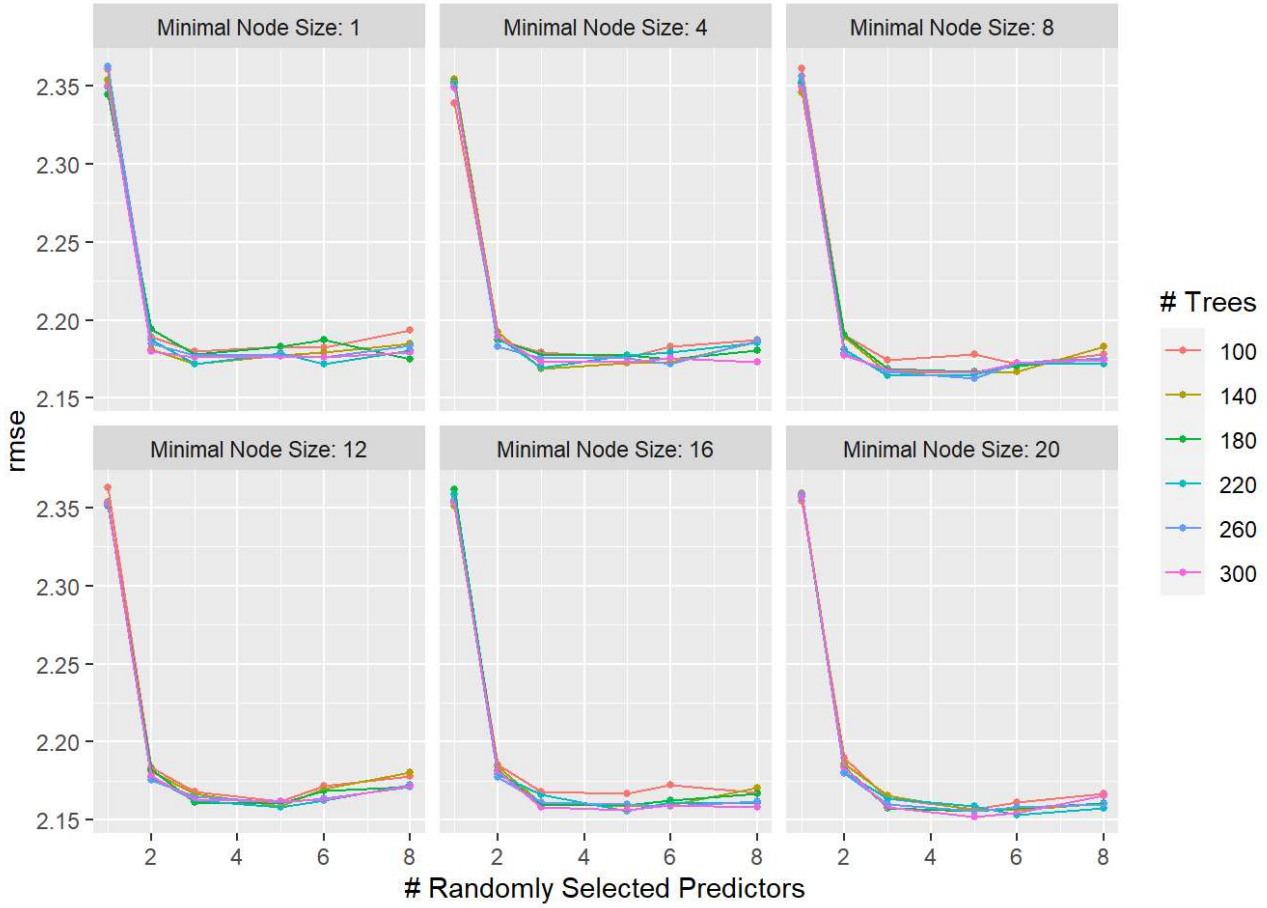
```
abalone_train1 <- abalone_train %>% select(-rings)

abalone_recipe <- recipe(age ~ ., data = abalone_train1) %>%
  step_dummy(all_nominal_predictors())

int_mod_1 <- abalone_recipe %>%
  step_interact(terms = ~ starts_with("type"):shucked_weight) %>%
  step_interact(terms = ~ longest_shell:diameter) %>%
  step_interact(terms = ~ shucked_weight:shell_weight)
norm_trans <- int_mod_1 %>%
  step_normalize(all_predictors())
```

Hide

```
rf_spec1 <- rand_forest(mtry = tune(),trees = tune(), min_n=tune()) %>%
  set_engine("randomForest") %>%
  set_mode("regression")
rf_wf1 <- workflow() %>%
  add_model(rf_spec1) %>%
  add_recipe(norm_trans)
```

Hide

```
reg_grid2 <- grid_regular(mtry(range = c(1, 8)), trees(range=c(100,300)), min_n(range=c(1
,20)),levels = 6)
tune_res4 <- tune_grid(
  rf_wf1,
  resamples = abalone_folds,
  grid = reg_grid2,
  metrics = metric_set(rmse)
)
autoplot(tune_res4)
```



Hide

```
best_abalone <- select_best(tune_res4,metric="rmse")
abalone_final <- finalize_workflow(rf_wf1, best_abalone)

rf_fit_abalone <- fit(abalone_final, data = abalone_train1)
augment(rf_fit_abalone, new_data = abalone_test) %>%
  rmse(truth = age, estimate = .pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        2.09
```