

# PSTAT 131 - Homework Assignment 6

Akshat Ataliwala (7924145)

May 19, 2022

## 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.

Read in the file and familiarize yourself with the variables using pokemon\_codebook.txt.

```
library(tidymodels)
library(tidyverse)
library(ggplot2)
library(corr)
library(klaR)
library(glmnet)
library(MASS)
library(discrim)
library(poissonreg)
library(janitor)
library(corrplot)

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

### Exercise 1

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

```
data <- read_csv("data/pokemon.csv")
data %>% head(5)
```

```
## # A tibble: 5 x 13
##   '# Name' 'Type 1' 'Type 2' Total HP Attack Defense 'Sp. Atk' 'Sp. Def'
##   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 1 Bulbas~ Grass Poison 318 45 49 49 65 65
## 2 2 Ivysaur Grass Poison 405 60 62 63 80 80
## 3 3 Venusa~ Grass Poison 525 80 82 83 100 100
## 4 3 Venusa~ Grass Poison 625 80 100 123 122 120
## 5 4 Charma~ Fire <NA> 309 39 52 43 60 50
## # ... with 3 more variables: Speed <dbl>, Generation <dbl>, Legendary <lgl>
```

- Use `clean_names()`
- Filter out the rarer Pokémon types
- Convert `type_1` and `legendary` to factors

```
# Clean Names
data <- data %>% clean_names()

# Filter out rarer Pokemon types
data <- data %>% filter(type_1 == "Bug" | type_1 == "Fire" |
                        type_1 == "Grass" | type_1 == "Normal" |
                        type_1 == "Water" | type_1 == "Psychic")

# Convert type_1, legendary, generation to factors
data$type_1 <- as.factor(data$type_1)
data$generation <- as.factor(data$generation)
data$legendary <- as.factor(data$legendary)

data %>% head(5)
```

```
## # A tibble: 5 x 13
##   number name      type_1 type_2 total hp attack defense sp_atk sp_def speed
##   <dbl> <chr> <fct> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 1 Bulbasaur Grass Poison 318 45 49 49 65 65 45
## 2 2 Ivysaur Grass Poison 405 60 62 63 80 80 60
## 3 3 Venusaur Grass Poison 525 80 82 83 100 100 80
## 4 3 VenusaurM~ Grass Poison 625 80 100 123 122 120 80
## 5 4 Charmander Fire <NA> 309 39 52 43 60 50 65
## # ... with 2 more variables: generation <fct>, legendary <fct>
```

Do an initial split of the data; you can choose the percentage for splitting. Stratify on the outcome variable.

```
# Setting Seed
set.seed(3478)

# Stratified Initial Split
data_split <- initial_split(data = data,
                             prop = 0.8,
                             strata = type_1)
```

```
train <- training(data_split)
test <- testing(data_split)
```

Fold the training set using  $v$ -fold cross-validation, with  $v = 5$ . Stratify on the outcome variable.

```
folds <- vfold_cv(data = train,
                  v = 5,
                  strata = type_1)
```

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.

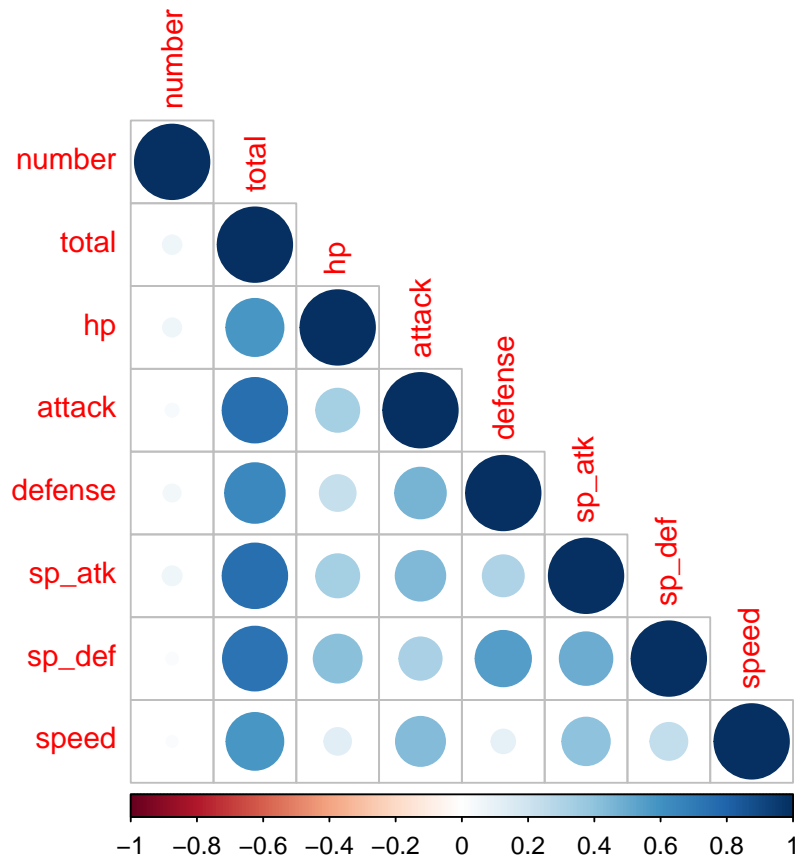
```
pokemon_recipe <- recipe(formula = type_1 ~ legendary + generation + sp_atk + attack + speed + defense +
                          data = train) %>%
  step_dummy(legendary) %>%
  step_dummy(generation) %>%
  step_normalize(all_predictors())
```

## 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).*

```
data %>%
  select(is.numeric) %>%
  cor() %>%
  corrplot(type = "lower")
```

```
## Warning: Predicate functions must be wrapped in 'where()'.
##
## # Bad
## data %>% select(is.numeric)
##
## # Good
## data %>% select(where(is.numeric))
##
## i Please update your code.
## This message is displayed once per session.
```



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

The number of a Pokemon has no relation to any of the attributes of the Pokemon. Speed is slightly positively correlated with attack and special attack. Special defense is slightly correlated with defense and special attack. Special attack is slightly correlated with attack. Defense is slightly correlated with attack. When looking at the influence of individual attributes on the total score, it seems to be that attack, special attack, and special defense are the most important. There are no negatively correlated variables.

### 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`.

```
pokemon_tree_spec <- decision_tree(cost_complexity = tune()) %>%
  set_mode("classification") %>%
  set_engine("rpart")

pokemon_tree_workflow <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(pokemon_tree_spec)

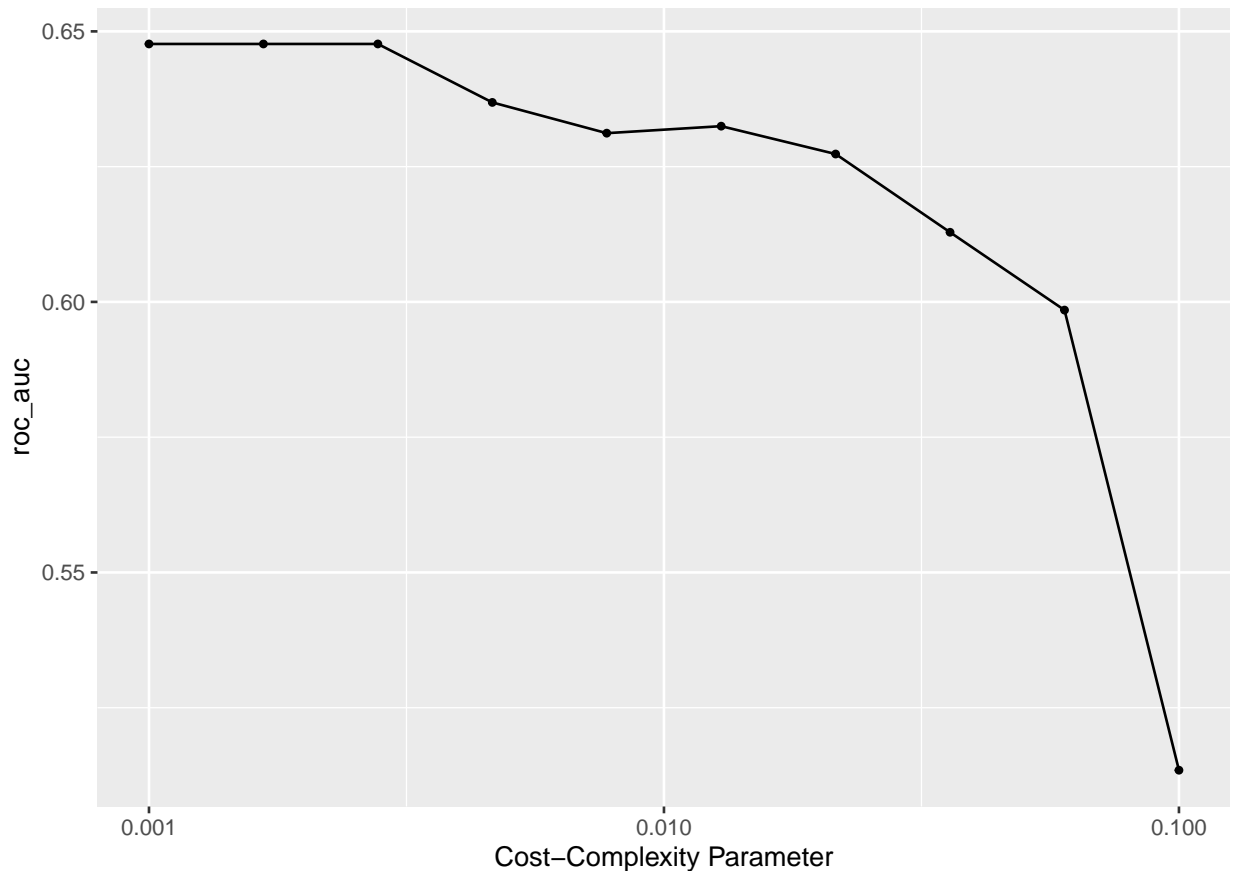
param_grid_tree <- grid_regular(cost_complexity(range = c(-3, -1)), levels = 10)

tune_res_tree <- tune_grid(
```

```
pokemon_tree_workflow,
  resamples = folds,
  grid = param_grid_tree,
  metrics = metric_set(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?

```
autoplot(tune_res_tree)
```



In general, as cost-complexity goes up the overall `roc_auc` of our model goes down. There seems to be no change in cost complexities of 0.001, 0.0016, 0.0027 (`roc_auc` of 0.6477), but after that the model starts performing worse and worse until it eventually reaches an `roc_auc` of 0.5135 at the largest cost\_complexity of 1.

#### Exercise 4

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

```
# the best cost_complexity in terms of roc_auc is the top result
collect_metrics(tune_res_tree) %>% 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	0.001	roc_auc	hand_till	0.648	5	0.0185	Preprocessor1_Model01
## 2	0.00167	roc_auc	hand_till	0.648	5	0.0185	Preprocessor1_Model02
## 3	0.00278	roc_auc	hand_till	0.648	5	0.0185	Preprocessor1_Model03
## 4	0.00464	roc_auc	hand_till	0.637	5	0.0219	Preprocessor1_Model04
## 5	0.0129	roc_auc	hand_till	0.632	5	0.0177	Preprocessor1_Model06
## 6	0.00774	roc_auc	hand_till	0.631	5	0.0218	Preprocessor1_Model05
## 7	0.0215	roc_auc	hand_till	0.627	5	0.0179	Preprocessor1_Model07
## 8	0.0359	roc_auc	hand_till	0.613	5	0.0151	Preprocessor1_Model08
## 9	0.0599	roc_auc	hand_till	0.598	5	0.0143	Preprocessor1_Model09
## 10	0.1	roc_auc	hand_till	0.513	5	0.0135	Preprocessor1_Model10

The roc\_auc of the best-performing pruned decision tree was 0.6477, which has a corresponding cost\_complexity of 0.001, 0.001668, 0.002783 (all had the same performance).

```
best_parameter_tree <- select_best(tune_res_tree, metric = "roc_auc")
best_parameter_tree
```

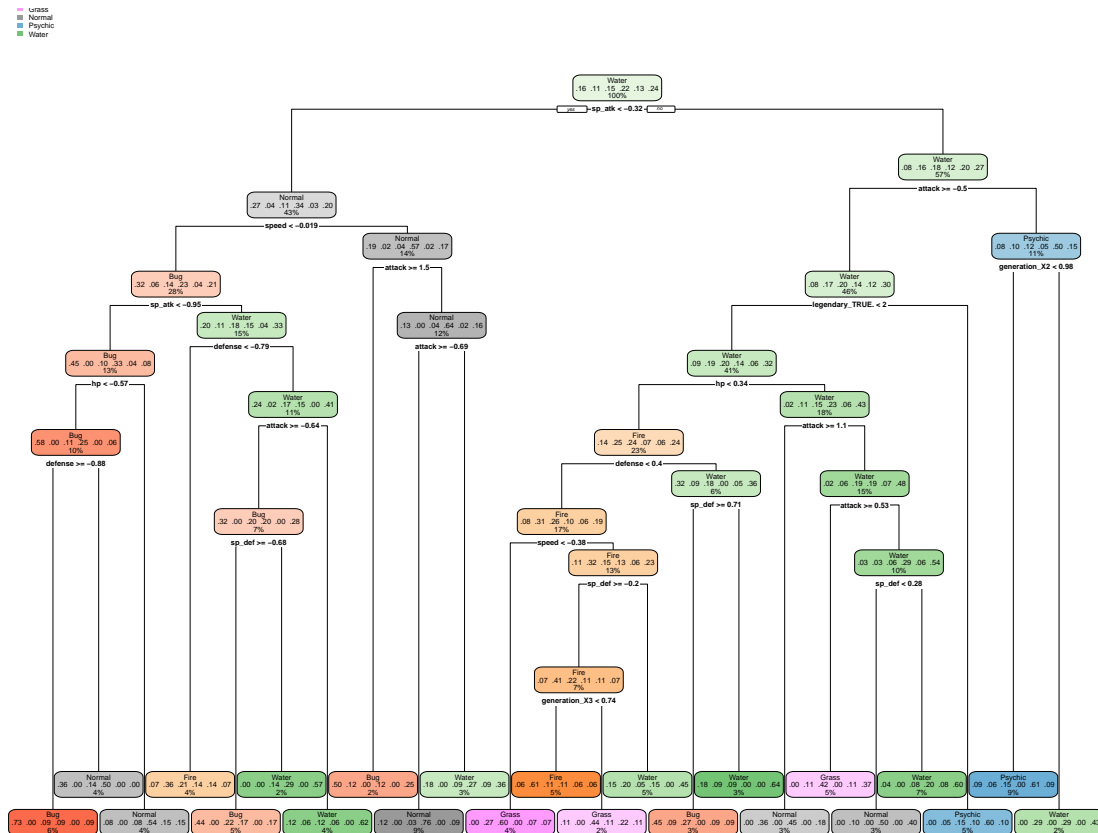
```
## # A tibble: 1 x 2
##   cost_complexity .config
##           <dbl> <chr>
## 1           0.001 Preprocessor1_Model01
```

## Exercise 5

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

```
#finalizing workflow and fitting that model to the training data
pokemon_tree_final <- finalize_workflow(pokemon_tree_workflow, best_parameter_tree)
pokemon_tree_final_fit <- fit(pokemon_tree_final, data = train)

# visualizing best performing pruned decision tree
pokemon_tree_final_fit %>% extract_fit_engine() %>% rpart.plot(roundint = FALSE)
```



## 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.

```
pokemon_rf_spec <- rand_forest(mtry=tune(), trees=tune(), min_n=tune()) %>%
  set_engine("randomForest", importance = TRUE) %>%
  set_mode("classification")

pokemon_rf_workflow <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(pokemon_rf_spec)
```

`mtry` represents the number of randomly selected variables we give each tree to make decisions with.

`trees` represents the number of trees we will create in our forest

`min_n` represents the minimum # data points in each node that are required for it to further split

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?**

```
# create regular grid
param_grid_rf <- grid_regular(mtry(range = c(1, 8)),
                             trees(range = c(1, 10)),
                             min_n(range = c(1, 10)),
                             levels = 8)

param_grid_rf
```

```
## # A tibble: 512 x 3
##   mtry trees min_n
##   <int> <int> <int>
## 1     1     1     1
## 2     2     1     1
## 3     3     1     1
## 4     4     1     1
## 5     5     1     1
## 6     6     1     1
## 7     7     1     1
## 8     8     1     1
## 9     1     2     1
## 10    2     2     1
## # ... with 502 more rows
```

We have 9 predictors in our dataset, so if we set `mtry = 9` we are no longer training a random forest model and instead are utilizing bagging, because there is no subset of the predictors that is randomly chosen (its using all available predictors).

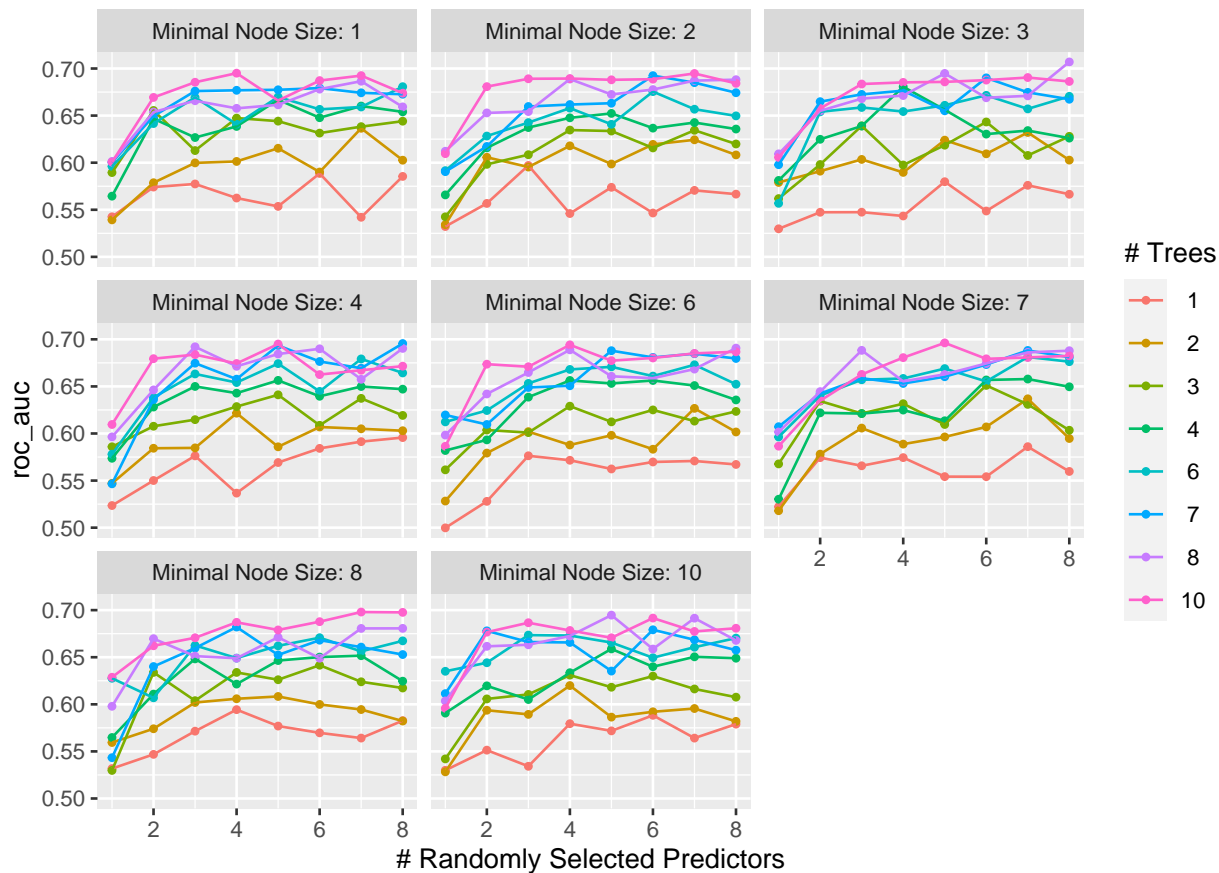
## Exercise 6

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_rf <- tune_grid(
  pokemon_rf_workflow,
  resamples = folds,
  grid = param_grid_rf,
  metrics = metric_set(roc_auc))
```

```
autoplot(tune_res_rf)
```





Across all values of `min_n`, the best performing models had 7, 8, or 10 trees. Minimal node size of 3, 8, and 4 seemed to perform similarly well, with `roc_auc` very close to 0.7. For most models, increasing the # of randomly selected predictors to at least 3 seem to bring the greatest benefit in performance, after which some models flatten out and others go back and forth in terms of `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()`.*

```
# the best parameters are the top results
collect_metrics(tune_res_rf) %>% arrange(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     8     8     3 roc_auc hand_till 0.707     5 0.0149 Preprocessor1_Model~
## 2     7    10     8 roc_auc hand_till 0.698     5 0.0198 Preprocessor1_Model~
## 3     8    10     8 roc_auc hand_till 0.698     5 0.00786 Preprocessor1_Model~
## 4     5    10     7 roc_auc hand_till 0.696     5 0.0141 Preprocessor1_Model~
## 5     8     7     4 roc_auc hand_till 0.696     5 0.0116 Preprocessor1_Model~
## 6     4    10     1 roc_auc hand_till 0.695     5 0.0185 Preprocessor1_Model~
## 7     5    10     4 roc_auc hand_till 0.695     5 0.0108 Preprocessor1_Model~
## 8     5     8     3 roc_auc hand_till 0.695     5 0.0117 Preprocessor1_Model~
## 9     7    10     2 roc_auc hand_till 0.695     5 0.0157 Preprocessor1_Model~
```

```
## 10      5      8      10 roc_auc hand_till 0.695      5 0.0181 Preprocessor1_Model~
## # ... with 502 more rows
```

The best performing random forest model (`mtry = 8`, `trees = 8`, `min_n = 3`) had an `roc_auc` of 0.7070.

```
best_parameter_rf <- select_best(tune_res_rf, metric = "roc_auc")
best_parameter_rf
```

```
## # A tibble: 1 x 4
##   mtry trees min_n .config
##   <int> <int> <int> <chr>
## 1      8      8      3 Preprocessor1_Model184
```

## 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?

```
#finalizing workflow and fitting that model to the training data
pokemon_rf_final <- finalize_workflow(pokemon_rf_workflow, best_parameter_rf)
pokemon_rf_final_fit <- fit(pokemon_rf_final, data = train)

# visualizing decision tree - FOR SOME REASON THIS ISNT WORKING
# vip(pokemon_rf_final_fit)
```

The variables that are the most useful are . The variables that are the least useful are . This is/isn't what we expected because .

## 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.

```
# Set up boosted tree specification
pokemon_boosted_spec <- boost_tree(trees=tune()) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

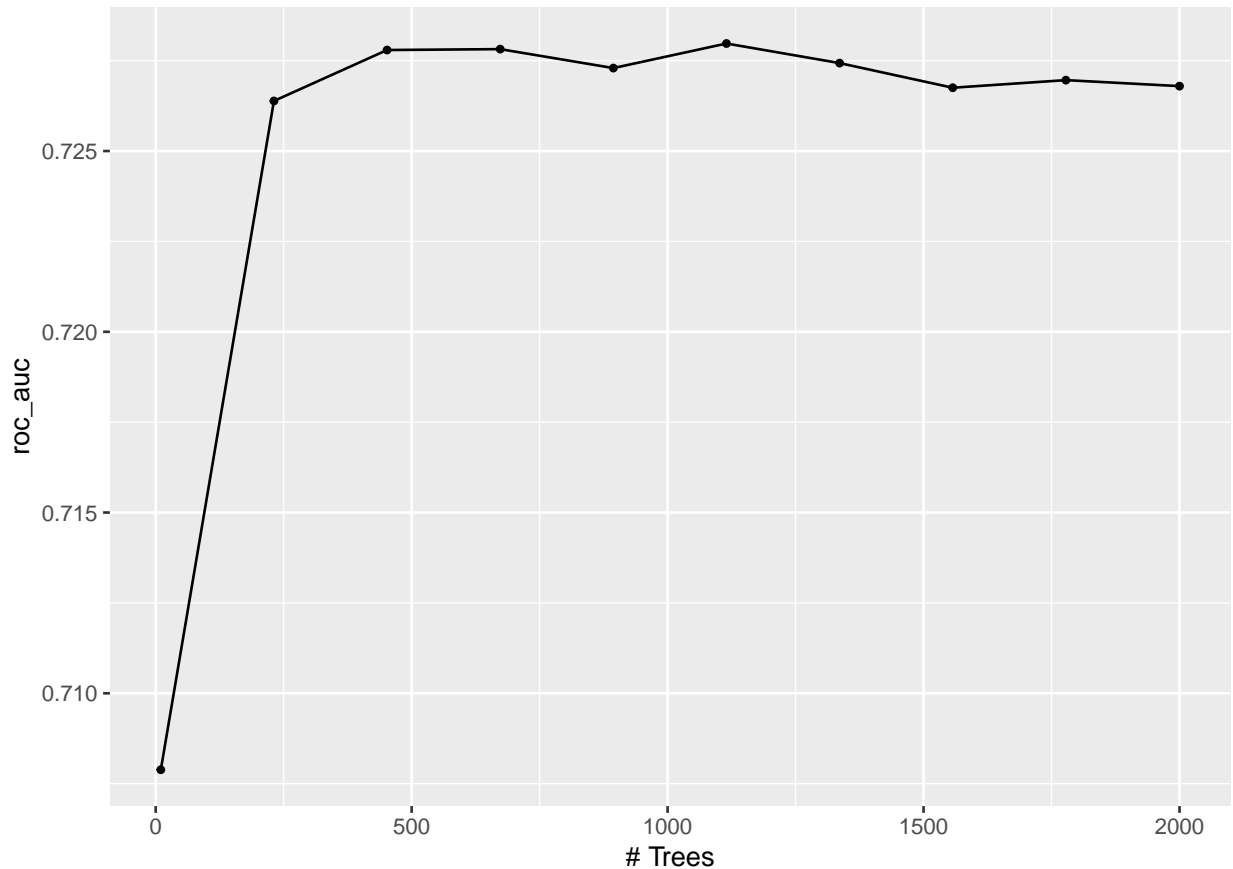
#boosted tree workflow
pokemon_boosted_workflow <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(pokemon_boosted_spec)

#boosted tree tuning grid
param_grid_boosted <- grid_regular(trees(range = c(10, 2000)),
                                   levels = 10)

#resulting values with hyperparameter tuning across CV
tune_res_boosted <- tune_grid(
```

```
pokemon_boosted_workflow,
  resamples = folds,
  grid = param_grid_boosted,
  metrics = metric_set(roc_auc))
```

```
autoplot(tune_res_boosted)
```



What do you observe?

We observe that

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

```
# the best parameters are the top results
collect_metrics(tune_res_boosted) %>% arrange(desc(mean))
```

```
## # A tibble: 10 x 7
##   trees .metric .estimator mean      n std_err .config
##   <int> <chr>    <chr>    <dbl> <int>  <dbl> <chr>
## 1  1115 roc_auc hand_till  0.728     5 0.00851 Preprocessor1_Model106
## 2   673 roc_auc hand_till  0.728     5 0.00865 Preprocessor1_Model104
## 3   452 roc_auc hand_till  0.728     5 0.00830 Preprocessor1_Model103
## 4  1336 roc_auc hand_till  0.727     5 0.00859 Preprocessor1_Model107
## 5   894 roc_auc hand_till  0.727     5 0.00821 Preprocessor1_Model105
```

```
## 6 1778 roc_auc hand_till 0.727 5 0.00838 Preprocessor1_Model09
## 7 2000 roc_auc hand_till 0.727 5 0.00833 Preprocessor1_Model10
## 8 1557 roc_auc hand_till 0.727 5 0.00860 Preprocessor1_Model08
## 9 231 roc_auc hand_till 0.726 5 0.00645 Preprocessor1_Model02
## 10 10 roc_auc hand_till 0.708 5 0.00965 Preprocessor1_Model01
```

```
best_paramater_boosted <- select_best(tune_res_boosted, metric = "roc_auc")
best_paramater_boosted
```

```
## # A tibble: 1 x 2
##   trees .config
##   <int> <chr>
## 1 1115 Preprocessor1_Model06
```

## 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.

```
roc_aucs <- bind_rows(best_parameter_tree, best_parameter_rf, best_paramater_boosted)
roc_aucs <- roc_aucs %>% add_column('model' = c("Pruned Decision Tree", "Random Forest", "Boosted Tree"),
                                   'roc_auc' = c(0.6477, 0.7070, 0.7280))
roc_aucs[, c("model", ".config", "cost_complexity", "mtry", "trees", "min_n", "roc_auc")]
```

```
## # A tibble: 3 x 7
##   model          .config          cost_complexity mtry trees min_n roc_auc
##   <chr>          <chr>          <dbl> <int> <int> <int> <dbl>
## 1 Pruned Decision Tree Preprocessor1_~ 0.001  NA  NA  NA  0.648
## 2 Random Forest      Preprocessor1_~ NA      8   8   3  0.707
## 3 Boosted Tree       Preprocessor1_~ NA      NA 1115 NA  0.728
```

```
pokemon_final <- finalize_workflow(pokemon_boosted_workflow, best_paramater_boosted)
pokemon_final_fit <- fit(pokemon_final, data = train)
```

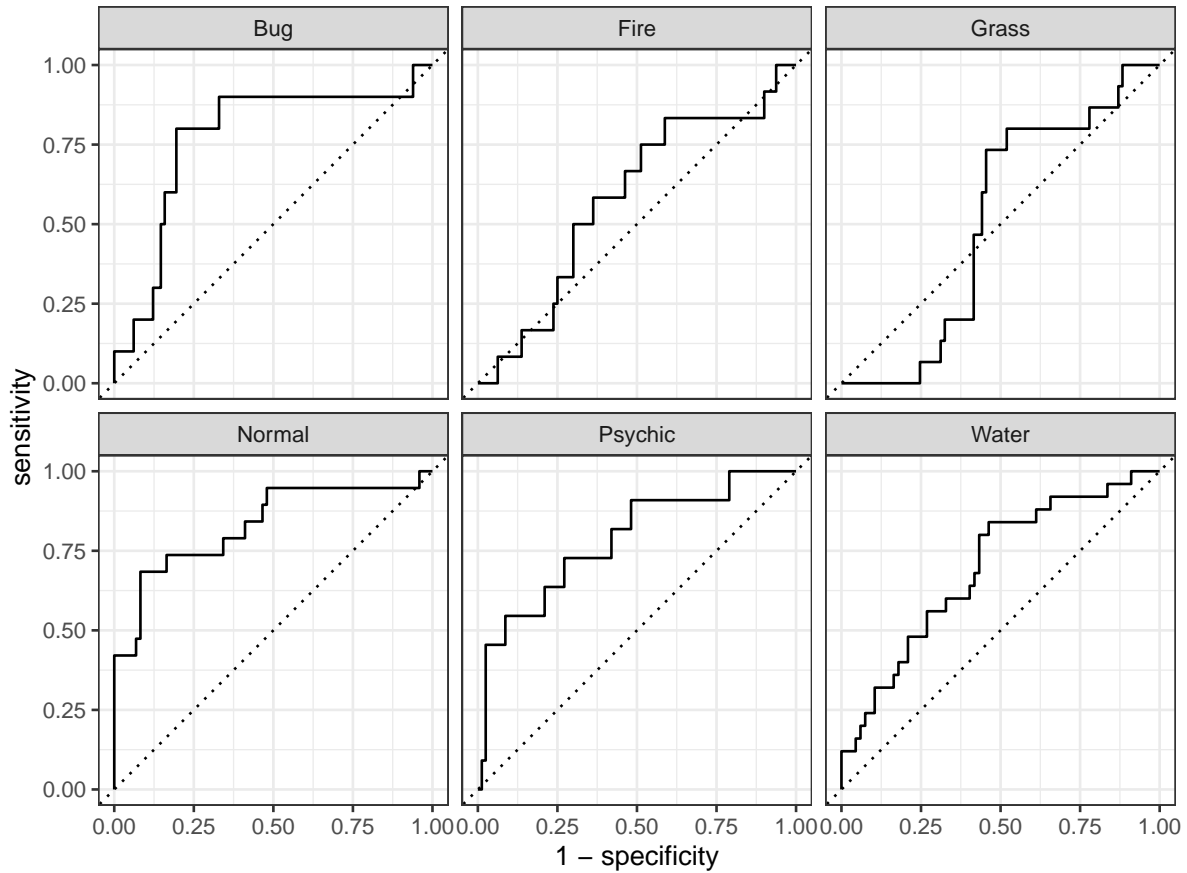
Of all the methods we trained, the Boosted Tree model with 1115 trees performed the best on the folds with an `roc_auc` of 0.7280.

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.

```
testing_roc_auc <- augment(pokemon_final_fit, new_data = test) %>%
  roc_auc(truth = type_1, estimate = c(.pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal, .pred_Psychic),
testing_roc_auc
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>      <dbl>
## 1 roc_auc hand_till 0.692
```

```
roc_curves <- augment(pokemon_final_fit, new_data = test) %>%
  roc_curve(truth = type_1, estimate = c(.pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal, .pred_Psychic))
autoplot()
roc_curves
```



```
final_model_conf <- augment(pokemon_final_fit, new_data = test) %>%
  conf_mat(truth = type_1, estimate = .pred_class) %>%
  autoplot(type = "heatmap")
final_model_conf
```

Prediction	Bug -	3	2	5	2	0	2
	Fire -	0	2	4	0	3	5
	Grass -	2	1	0	1	1	3
	Normal -	3	1	0	13	1	3
	Psychic -	0	2	0	0	5	0
	Water -	2	4	6	3	1	12
		Bug	Fire	Grass	Normal	Psychic	Water
		Truth					

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

The model was extremely good at predicting Normal types with 6 misclassifications and 13 correct ones. This was followed by water with 13 misclassifications and 12 correct ones. Psychic was also very similar with 6 misclassifications and 5 correct ones. Bug types had 3 correct classifications with 7 misclassifications, and Fire had 2 correct classifications with 10 misclassifications. Grass was by far the worst class to predict because there were 0 correct classifications, and 15 misclassifications.