Predicting Pokemon Types

Introduction

The objective for this model is to predict Pokemon type using several battle stats.

Context

The Pokemon handheld series is an adventure game series with a turn based attack system. How well a Pokemon does during battle depends on its type, battle stats, and level. For example, psychic type Pokemon tend to have a very high special attack stat, but a lower defense stat. On the other hand, Bug types tend to be fast, but don't deal much damage. Type has an effect on every battle stat.

Loading and Tidying the Data

The csv file containing the data comes from Kaggle.

```
# Read in csv file and clean names
poke <- read.csv('Pokemon.csv')
pokemon <- clean_names(poke)
```

This model will only consider the 6 main types: Bug, Fire, Grass, Normal, Water, and Psychic. These types are both common, and present in every generation of Pokemon games.

```
# Filter out the types we want
pokemon <- pokemon%>%filter(type_1 == 'Bug' | type_1== 'Fire' | type_1== 'Grass' | type_1== 'Normal' | t

# Factor Nominal Variables
pokemon$legendary <- as.factor(pokemon$legendary)
pokemon$type_1 <- as.factor(pokemon$type_1)
pokemon$generation <- as.factor(pokemon$generation)
```

The resulting data set contains 458 observations.

Creating the Recipe

Now, its time to split the data set into training and testing sets. The split will be 80% training (364 observations) and 20% testing (94 observations), stratifying on type_1.

```
# Split the data, stratifying on type
pokeSplit <- initial_split(pokemon, prop = 0.80,
                                strata = type_1)
poke_train <- training(pokeSplit)
poke_test <- testing(pokeSplit)
```

This model will use k-fold cross validation, where k=10. The recipe will contain 8 predictor variables, with legendary and generation being nominal.

```
# Create 10 folds, stratifying on type
poke_fold <- vfold_cv(poke_train, v = 10, strata = type_1)

# Create our recipe, accounting for all nominal predictors
pokeRecipe <- recipe(type_1 ~ legendary + generation + sp_atk + attack + speed  + defense + hp + sp_def
```

Exploring the Correlations

Now let's take a look at the correlation plot for the data set.

```
poke_train %>% select(where(is.numeric)) %>% select(-x, -total) %>% cor() %>%
  corrplot(method = 'number')
```

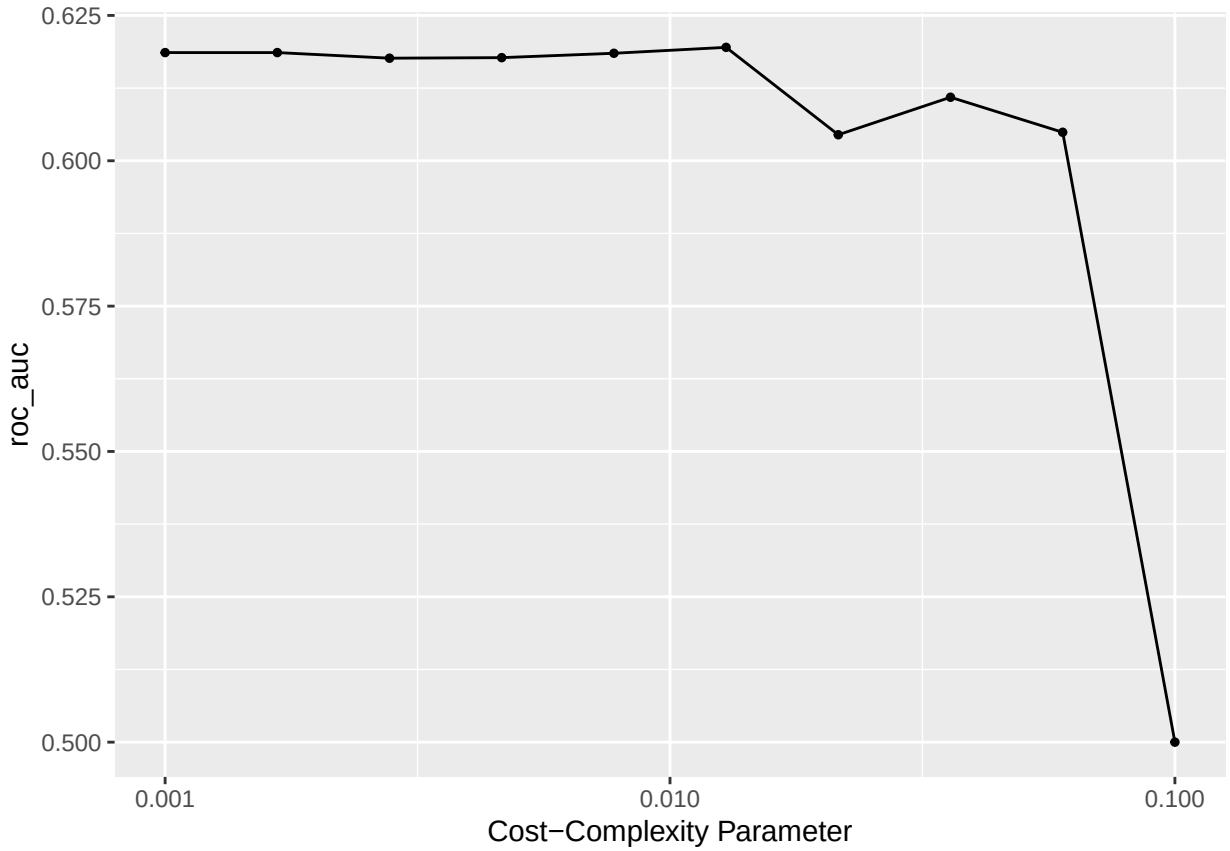
The correlation plot only shows us one variable relationship with a correlation above 0.50, special defense and defense. This makes sense because Pokemon with a high damage resistance to special attacks will often have a high damage resistance to normal attacks as well. Special attack and attack have a high correlation for this same reason. It also shows that the lowest variable relationship is between speed and defense with a correlation of .10. This might only be intuitive for those familiar with the game. Pokemon with a high speed stat tend to be smaller and weigh less, while Pokemon with high defense are usually bulky and weigh more. There is a similar explanation for the relationship between speed and hp. While this correlation plot shows no negative correlations, I suspect that would be different if weight were included as a predictor variable (i.e wieght and speed).

Building and Running the models

Decision Tree

The first model tested will be a basic decision tree. It will use grid search to tune the cost-complexity parameter. The cost complexity parameter controls the size of the decsion tree.

```
# Create decision tree
tree_spec <- decision_tree() %>%
  set_engine("rpart")

class_tree_spec <- tree_spec %>%
  set_mode("classification")

# Create workflow and add recipe
class_tree_wf <- workflow() %>%
  add_model(class_tree_spec %>% set_args(cost_complexity = tune())) %>%
  add_recipe(pokeRecipe)

# Set up grid and begin tuning
param_grid <- grid_regular(cost_complexity(range = c(-3, -1)), levels = 10)

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

autoplot(tune_res)
```



The AUC is the highest when the cost-complexity parameter is around 0.015. So lets save the best decision tree model.

```
#store the best model and finalize the workflow
best_pruned <- select_best(tune_res)
class_tree_final <- finalize_workflow(class_tree_wf, best_pruned)
class_final_fit <- fit(class_tree_final, data = poke_train)
augment(class_final_fit, new_data = poke_test) %>% roc_auc(type_1, estimate = c(.pred_Bug, .pred_Fire,
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.617
```

This model gave us an AUC of 0.617. Here is a visual showing how predictions were made.

Random Forest Model

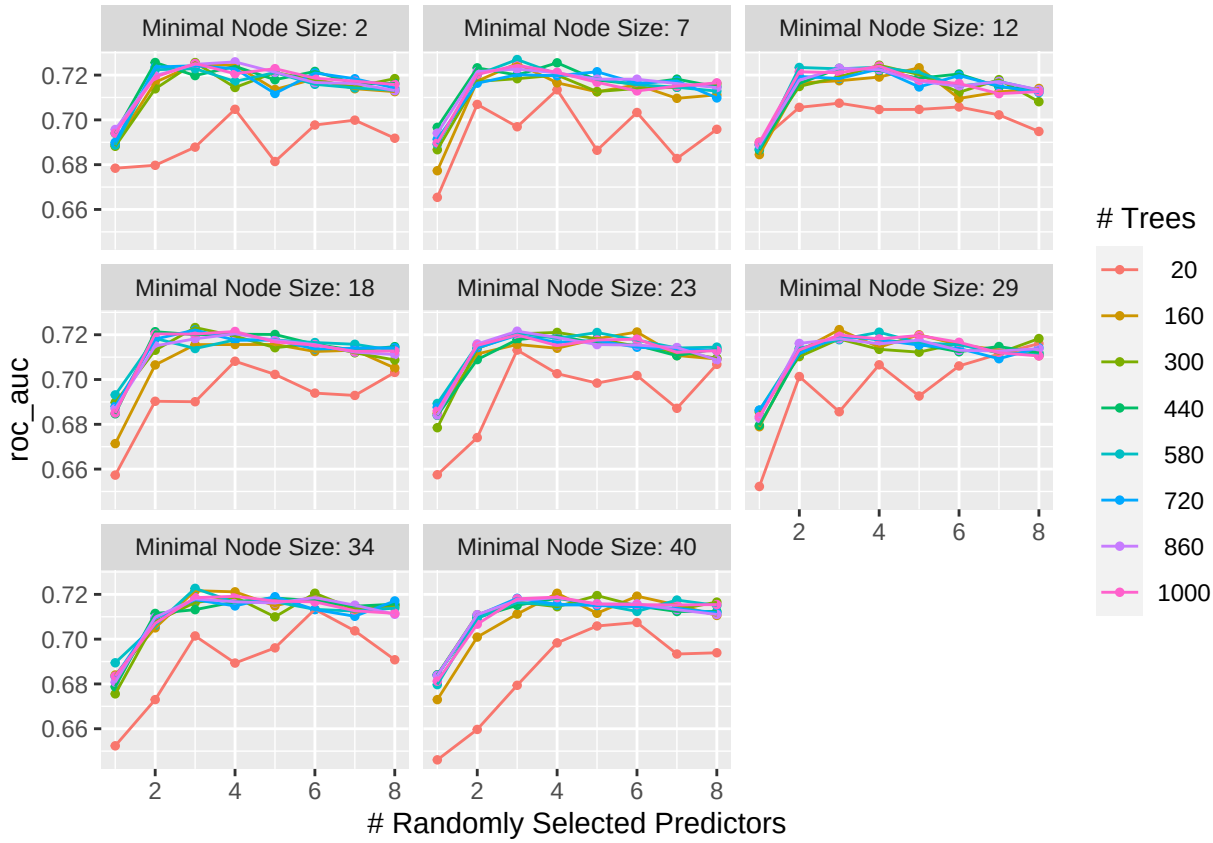
Now lets create a random forest model, tuning the following three parameters:

mtry - number of predictors sampled at each split
trees - number of trees
min_n - min number of observations in a node required for another split

```r
# Create Model
rf_spec <- rand_forest(mtry = tune()) %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("classification")

# Create workflow and add recipe
rf_wf <- workflow() %>%
  add_model(rf_spec %>% set_args(trees = tune(), min_n = tune())) %>%
  add_recipe(pokeRecipe)

# Create tune grid and specify the range for each tuning parameter.
rf_grid <-  grid_regular(mtry(range = c(1, 8)), trees(range = c(20, 1000)), min_n(range = c(2, 40)), le
```

Note: At mtry = 8 we are using all the predictors in each split. Therefore, we can not sample more than 8. Similarly, less than 1 is also not plausible because we won't have a useful model if 0 predictors are sampled at each split. That is why this range was chosen.

```r
# Begin tuning and plot candidate models
tune_res_rf <- tune_grid(
  rf_wf,
  resamples = poke_fold,
  grid = rf_grid,
  metrics = metric_set(roc_auc)
)

# Create plot
autoplot(tune_res_rf)
```



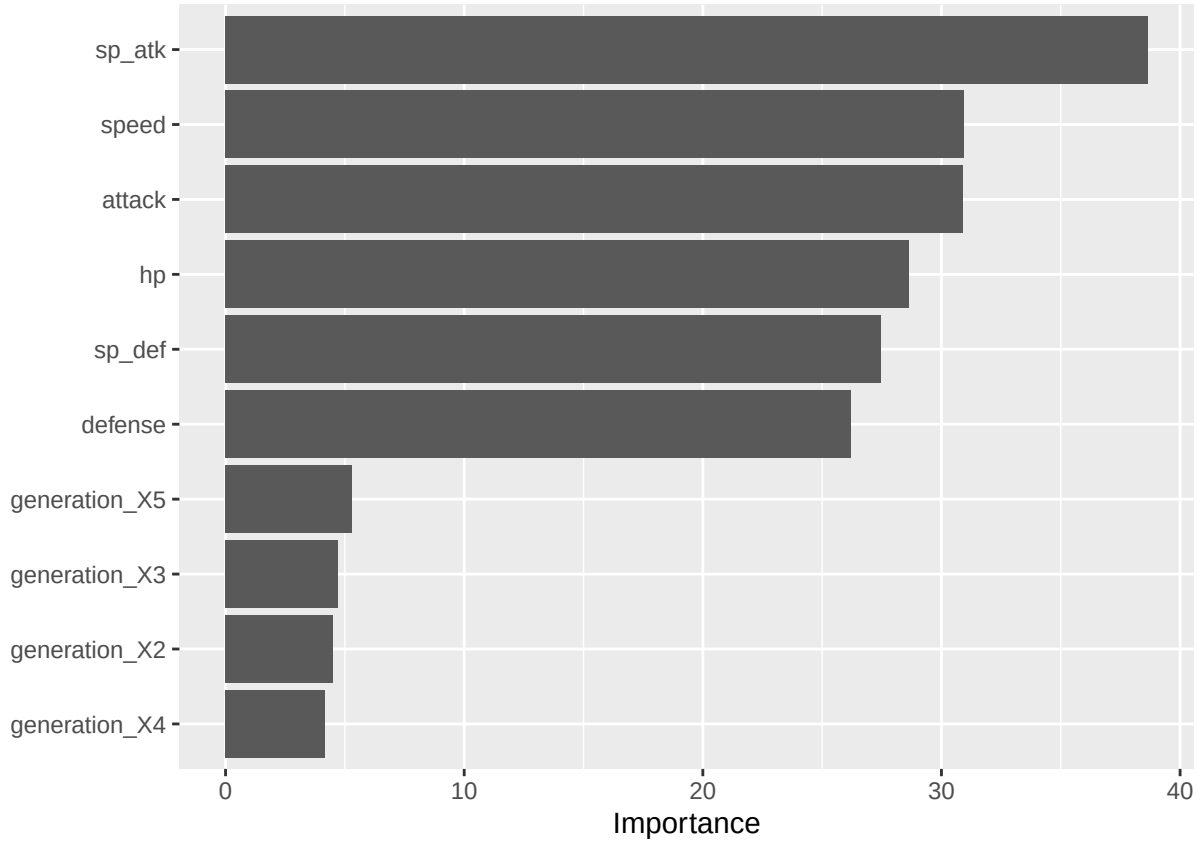
It is difficult to tell from the plot, but the best random forest model has mtry = 4, min_n = 7, and trees = 160. Now lets store that model and finalize the workflow.

```
# Select Best model and stroe in variable
best_rf<- select_best(tune_res_rf)

# Finalize random forest model
rf_final <- finalize_workflow(rf_wf, best_rf)
rf_final_fit <- fit(rf_final, data = poke_train)
augment(rf_final_fit, new_data = poke_test) %>% roc_auc(type_1, estimate = c(.pred_Bug, .pred_Fire, .pr
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimator
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.683
```

This model gives us an AUC of 0.683, which is noticeably better. Now, lets see which variables were the most and least useful in prediction.

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



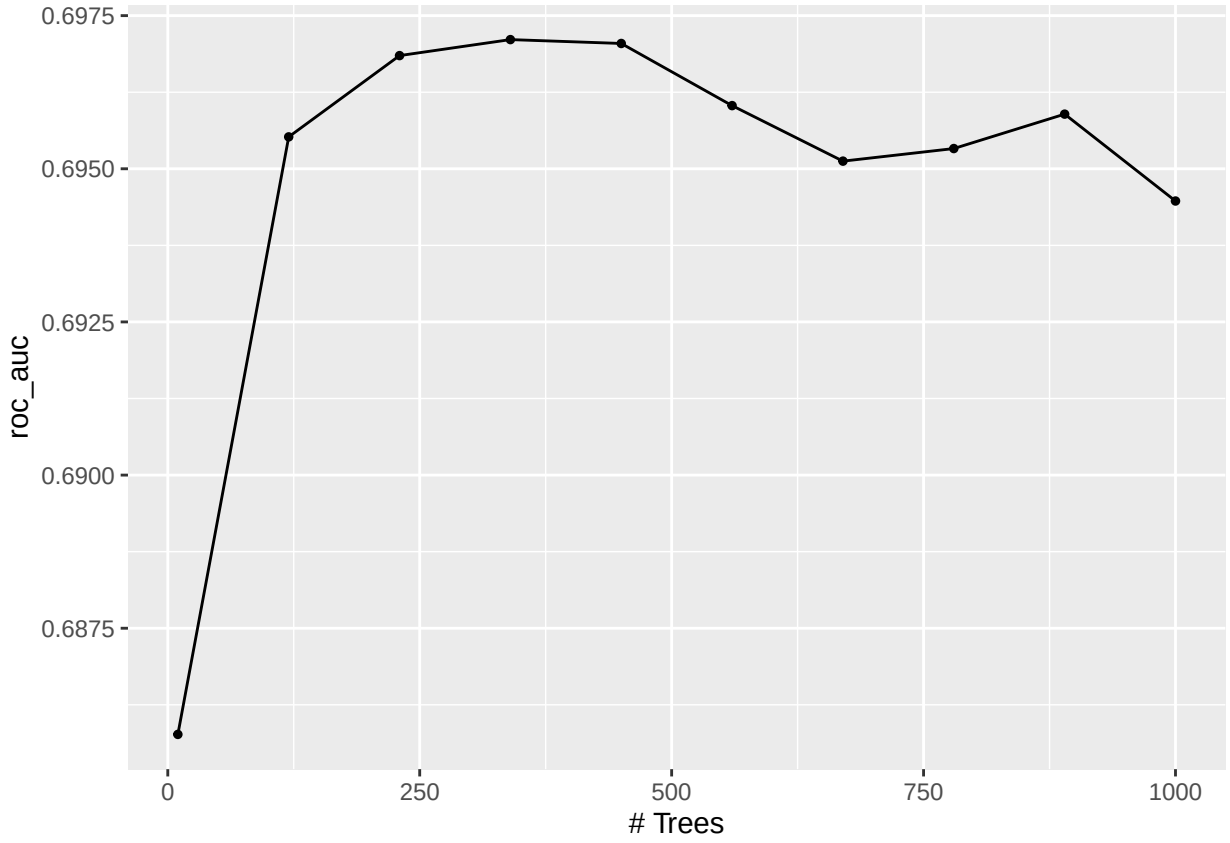
This shows that special attack, speed, attack were the most useful variables, whiles generation was the least useful. This makes sense, because there tends to be a roughly equal distribution of each type of Pokemon from generation to generation, thus it likely isn't useful for predicting type. Special attack is heavily dependent on type. Fire types have and extremely high special attack while bug and normal types have a much lower one. The same can be said, albeit to a lesser extent, for the attack and speed variables.

Boosted Tree Model

Lastly, we will fit a boosted tree model. For this model, the only parameter we will be tuning is trees. This is because the boosted tree model builds trees one at a time, using gradient boosting to improve each subsequent tree. Therefore, we really only need to tune trees, as the model already learns and grows with each tree.

```
# Create boosted tree model
boost_spec <- boost_tree(trees = tune(), tree_depth = 4) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

# Create workflow and add recipe
boost_wf <- workflow() %>%
  add_model(boost_spec %>% set_args(trees = tune())) %>%
  add_recipe(pokeRecipe)

# Create grid and tune trees
boost_grid <-  grid_regular(trees(range = c(10, 1000)), levels = 10)
```

```
# Begin tuning and plot candidate models
boost_tune <- tune_grid(
  boost_wf,
  resamples = poke_fold,
  grid = boost_grid,
  metrics = metric_set(roc_auc)
)

autoplot(boost_tune)
```



This graph clearly shows that our best boosted tree model has around 650 trees. Lets store and finalize this model, and see how it performs relative to the previous two.

```
best_boost<- select_best(boost_tune)
boost_final <- finalize_workflow(boost_wf, best_boost)
boost_final_fit <- fit(boost_final, data = poke_train)
augment(boost_final_fit, new_data = poke_test) %>% roc_auc(type_1, estimate = c(.pred_Bug, .pred_Fire,
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.671
```

The boosted tree model gave us an AUC of 0.671.

Conclusion

The random forest model performed the best, giving an Area Under Curve (AUC) of 0.683. This shows that of the models we ran, the random forest model with 580 trees, 3 predictors randomly sampled at each split, and a minimum of 7 data points required for a node to be split further, fit the data the best.