PSTAT 131 - HW 6

Ephets Head

5/19/2022

Exercise 1: Read in the pokemon data and set up things as in HW 5:

Use `clean_names()`, filter out the rarer Pokémon types, and convert `type_1`, `legendary`, and `generation` to factors.

```
#1. read in the data
pokemon <- read.csv("data/Pokemon.csv")

#2. clean the names
Pokemon <- clean_names(pokemon)

#3. filter out the rare Pokémon types
Pokemon <- Pokemon %>%
  filter(
    type_1 %in% c("Bug","Fire","Grass","Normal","Water","Psychic")
  )

#4. Convert type_1, legendary, generation to factors
Pokemon <- Pokemon %>%
  mutate(
    type_1 = factor(type_1),
    legendary = factor(legendary),
    generation = factor(generation)
  )
```

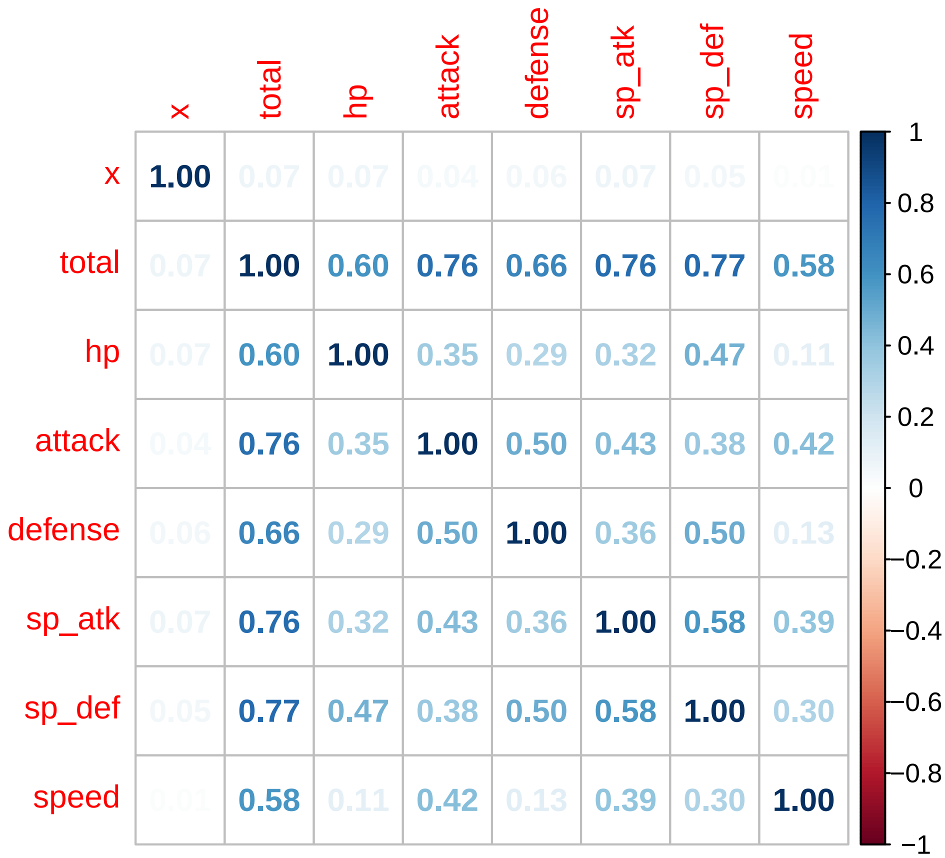
Do an initial split of the data, stratified by the outcome variable. Fold the training set using v-fold cross-validation, with v=5. Stratify on the outcome variable. Set up a recipe to predict `type_1` with `legendary`, `generation`, `sp_attack`, `attack`, `speed`, `defense`, `hp`, and `sp_def`: Dummy-code `legendary` and `generation`, and center/scale all predictors.

```
#first we will do our stratified initial split of the data
set.seed(5555)
pokemon_split <- initial_split(Pokemon, prop=0.75, strata=type_1)
pokemon_train <- training (pokemon_split)
pokemon_test <- testing(pokemon_split)

#next we will fold the training set using v-fold cross-validation with v=5
pokemon_folds <- vfold_cv(pokemon_train, v=5, strata=type_1)

#next we will set up a recipe to predict type_1
pokemon_rec <- recipe(type_1 ~ legendary + generation + sp_atk + attack + speed + defense + hp + sp_def
  step_dummy(legendary) %>%
  step_dummy(generation) %>%
  step_center(all_predictors()) %>%
```

```r
  step_scale(all_predictors())
```

Exercise 2: Create a correlation matrix of the training set, using the `corrplot` package. You can choose how to handle continuous variables for this plot, and justify your decisions. What relationships do you notice? Do these relationships make sense to you?

```r
pokemon_cor <- cor(pokemon_train[sapply(pokemon_train, is.numeric)])
corrplot(pokemon_cor, method='number')
```



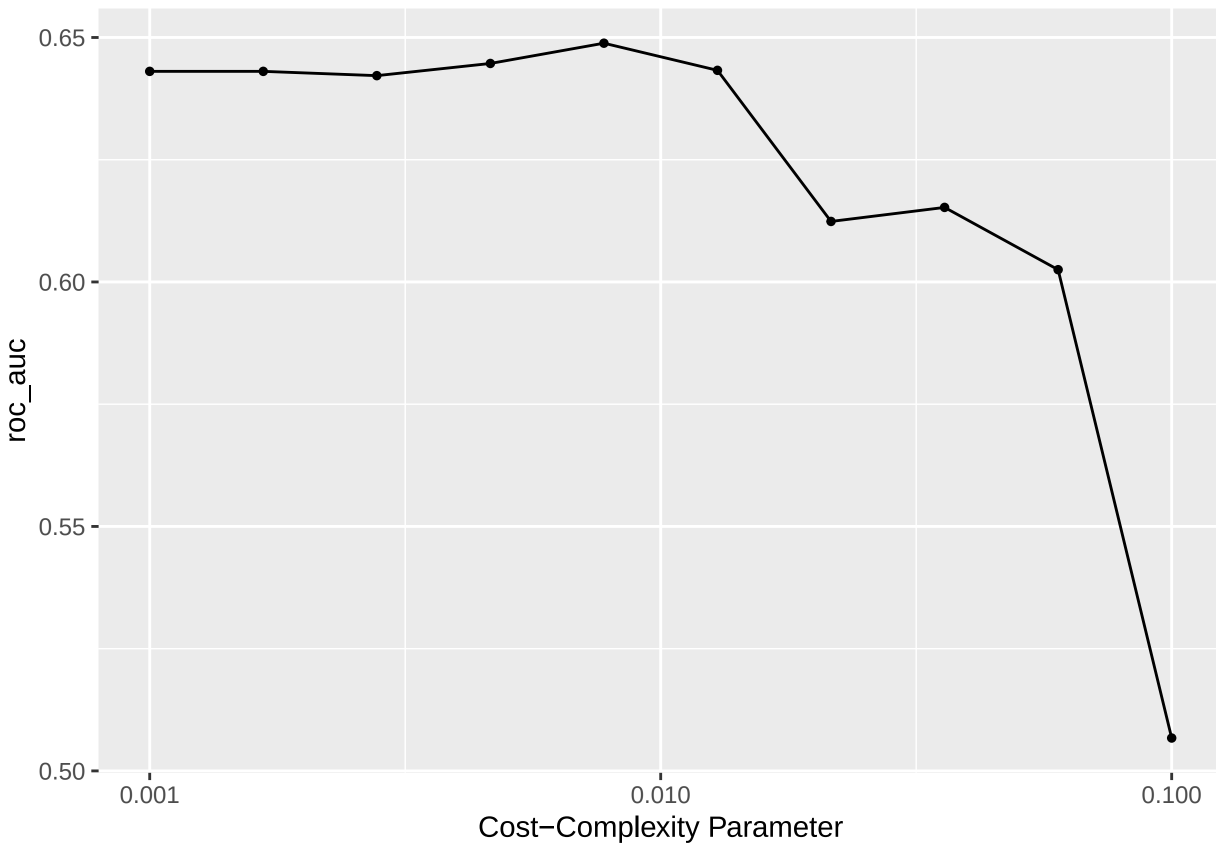
Correlations between continuous variables are represented by correlation coefficients, color coded to show which variables have the strongest correlations. As you might guess, the predictor `total` is the most strongly correlated with all the other variables, as it is the sum of all other statistics and is directly computed from them. `sp_atk` and `sp_def` are pretty positively correlated, as are `defense` and `attack`.

Exercise 3: First, set up a decision tree model and work flow. 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?

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

tree_workflow <- workflow() %>%
  add_model(tree_spec %>% set_args(cost_complexity=tune())) %>%
  add_recipe(pokemon_rec)
```

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

tuned_wf <- tune_grid(
  tree_workflow,
  resamples=pokemon_folds,
  grid=param_grid,
  metrics = metric_set(roc_auc)
)

autoplot(tuned_wf)
```



From the above plot of the tuned/fitted object, it seems like a decision tree performs much better (has a higher ROC AUC) with a smaller cost-complexity parameter. For `cost_complexity` values that are larger than about 0.015, the `roc_auc` decreases steeply.

Exercise 4: What is the `roc_auc` of your best-performing pruned decision tree on the folds? (Hint: use `collect_metrics()` and `arrange()`)

```r
#using collect_metrics() and arrange(), create a dataset of the top roc_auc decision trees
tuned_metrics <- collect_metrics(tuned_wf)
ordered_metrics <- arrange(tuned_metrics, desc(tuned_metrics$mean))

#output the highest roc_auc
head(ordered_metrics,1)
```

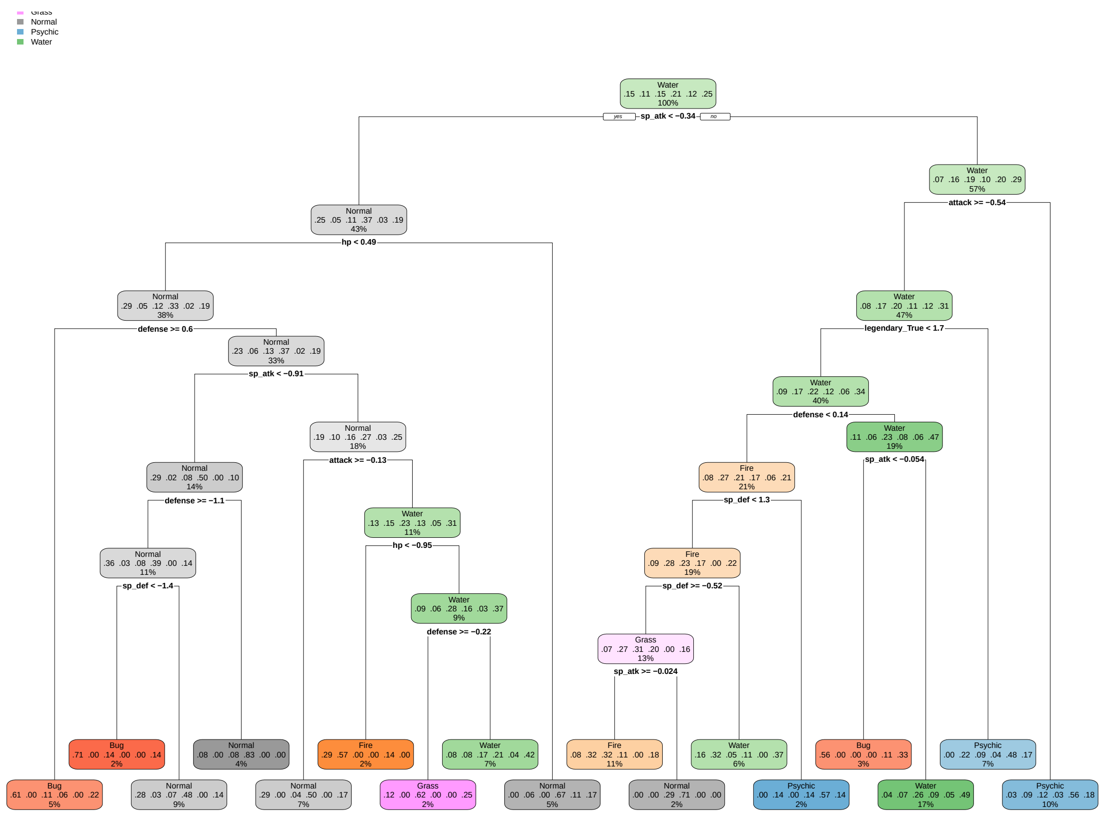
```
## # A tibble: 1 x 7
##   cost_complexity .metric .estimator  mean     n std_err .config
```

```
##             <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1         0.00774 roc_auc hand_till  0.649     5  0.0223 Preprocessor1_Model05
```

The ROC AUC of the best-performing decision tree is about 0.634.

Exercise 5A: Using `rpart.plot`, fit and visualize your best-performing pruned decision tree with the *training* set.

```r
#select the best-performing model
best_costcomplex <- select_best(tuned_wf)

#finalize the workflow using this model

final_wf <- finalize_workflow(tree_workflow, best_costcomplex)

#fit the model to the training set
fitted_cctree <- fit(final_wf, data=pokemon_train)

#using rpart.plot to visualize the fitted decision tree
fitted_cctree %>%
  extract_fit_engine() %>%
  rpart.plot(roundint=FALSE)
```



Exercise 5B: 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.

Then, 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 kind of model would `mtry=8` represent?

```r
#first we build the random forest model specification
rf_spec <- rand_forest(mtry=tune(), trees=1000, min_n=tune()) %>%
  set_engine("ranger", importance="impurity") %>%
  set_mode("classification")


#next we set up a workflow
rf_wf <- workflow() %>%
  add_model(rf_spec) %>%
  add_recipe(pokemon_rec)
```

The hyperparameter "mtry" takes an integer value, which represents the number of random predictor variables sampled each time a new model tree is split.

The hyperparameter "trees", also an integer, is simply the number of tree models.

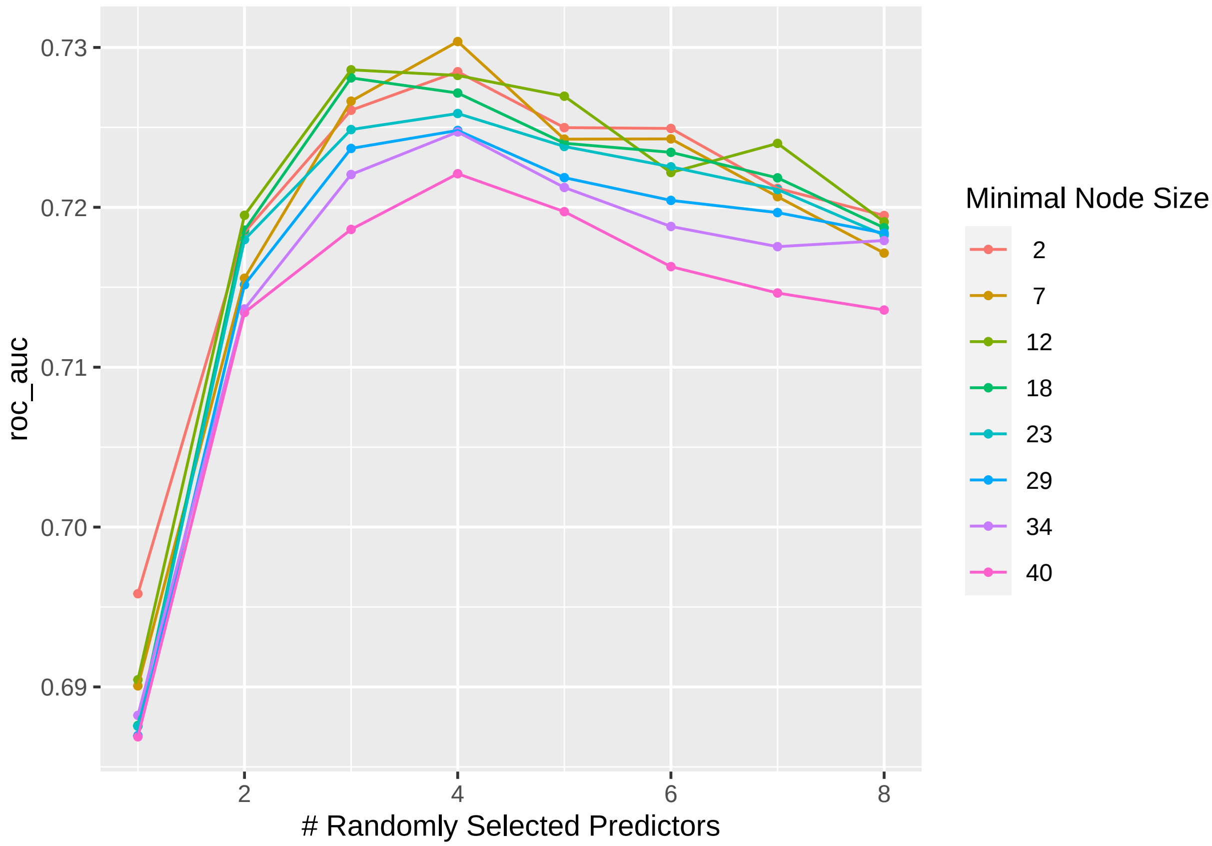
"min_n" is an integer representing the minimum number of data points required to be in one node before it splits.

```r
#now we create a regular grid with 8 levels

rf_grid <- grid_regular(mtry(range=c(1,8)),
                        min_n(range=c(2,40)),
                        levels=8)
#trees(range=c(1,2000)),
```

Since `mtry` represents the number of predictors randomly selected for each tree, it must be at least equal to 1 (you can't build a model with no predictors) and no larger than 8, since there are a total of 8 predictors in our recipe. If `mtry=8`, the model has access to all the predictor variables at every tree, which is a bagged tree model.

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 hyper parameters seem to yield the best performance?

```r
met <- metric_set(roc_auc)

tuned_rf <- tune_grid(
  rf_wf,
  resamples=pokemon_folds,
  grid=rf_grid,
  metrics = met
)

autoplot(tuned_rf)
```



From the plot of this tuned model, it appears that the highest ROC AUC values were achieved when mtry (the number of randomly selected predictors in each model) was set to be around 3-4, and `min_n` is in the range of 2-12.

Exercise 7: What is the `roc_auc` of your best-performing random forest model on the folds? Use `collect_metrics()` and `arrange()`.

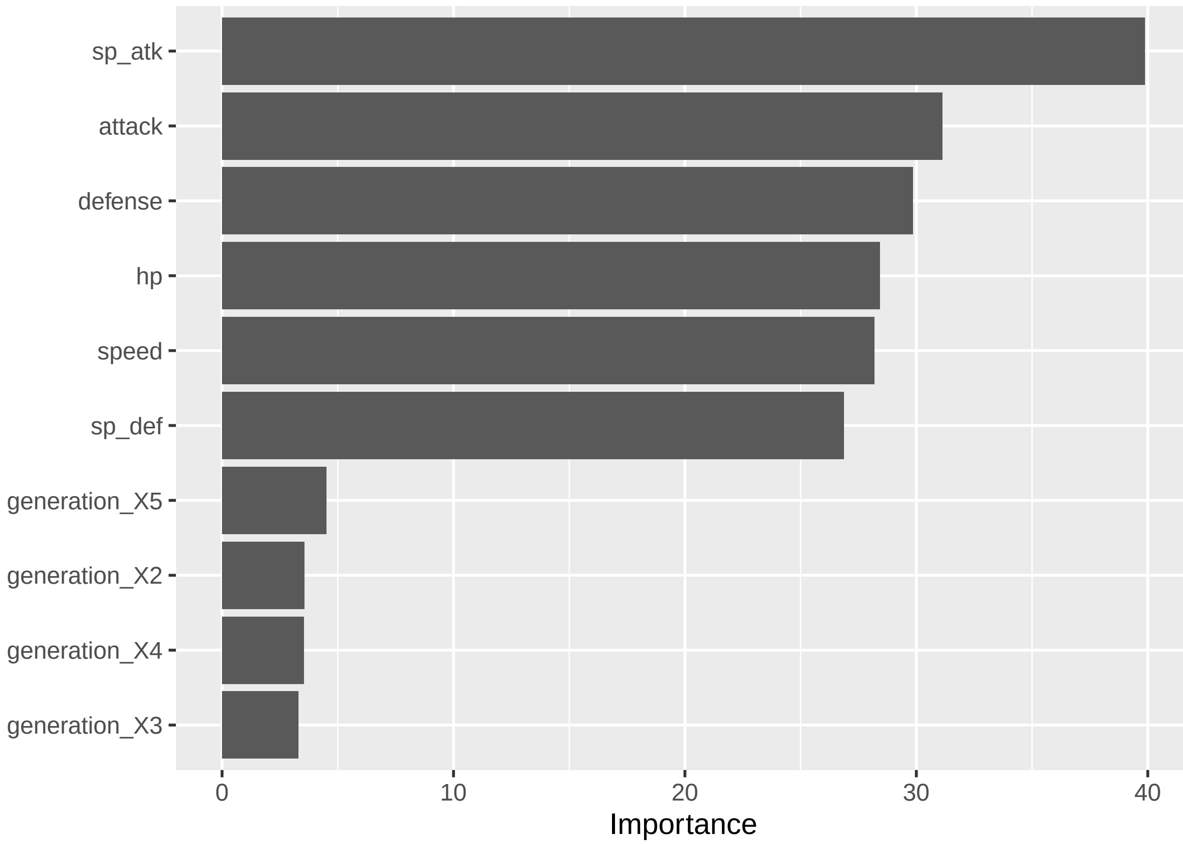
```
#using collect_metrics() and arrange(), create a dataset of the top roc_auc rf models
rf_metrics <- collect_metrics(tuned_rf)
ordered_rf_met <- arrange(rf_metrics, desc(rf_metrics$mean))

#output the highest roc_auc
head(ordered_rf_met,1)
```

```
## # A tibble: 1 x 8
##    mtry min_n .metric .estimator  mean     n std_err .config
##   <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1     4     7 roc_auc hand_till  0.730     5  0.0311 Preprocessor1_Model12
```

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?

```
best_rf <- select_best(tuned_rf)
final_rf_wf <- finalize_workflow(rf_wf, best_rf)
fitted_rf<- fit(final_rf_wf, data=pokemon_train)

fitted_rf %>%
  extract_fit_parsnip() %>%
```

```r
vip()
```



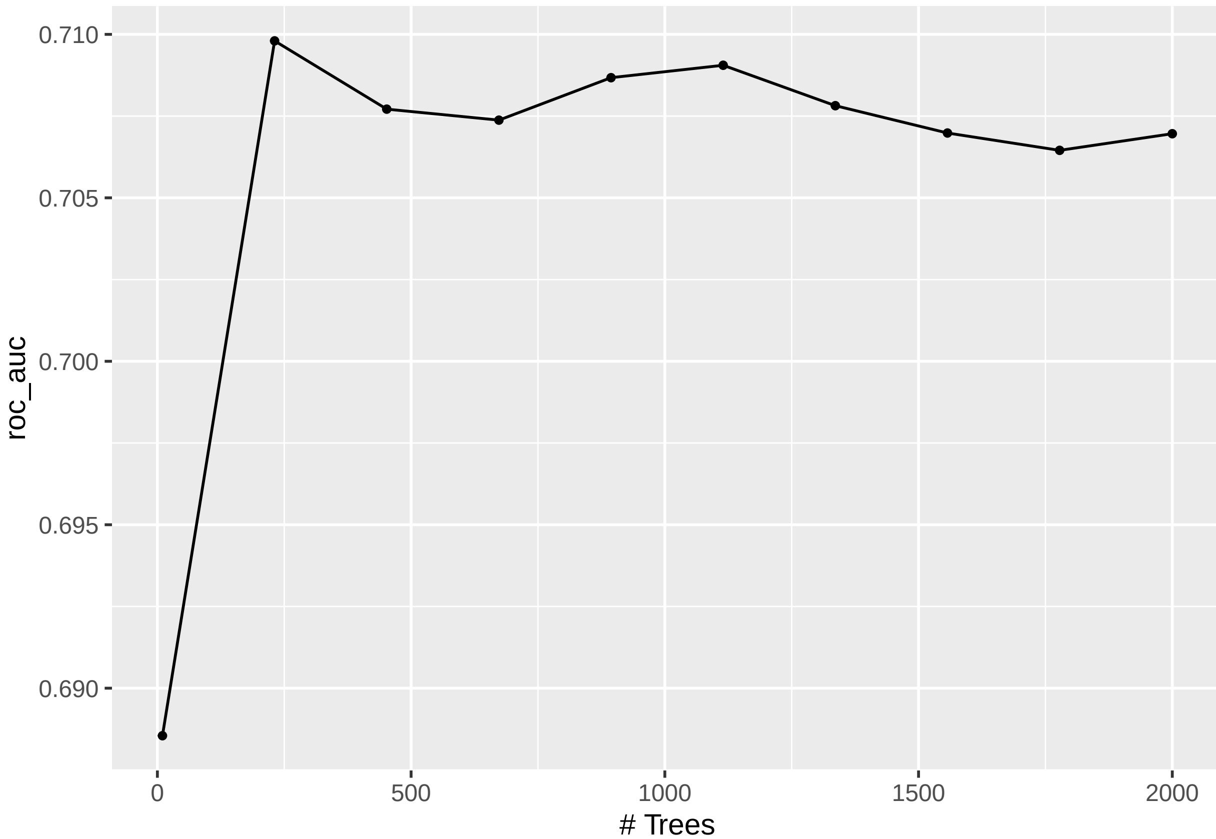
According to the variable importance plot, the most important (significant) variables in our random forest model are the Pokémon's attack and defense statistics: `sp_atk`, `attack`, `speed`, and `hp`. The least important variables (by a large margin) are the factored levels of `generation`.

Exercise 9: Finally, set up a boosted tree model and workflow. Use the `xgboost` engine. Tune `trees`. Create a regular grid with 10 levels, and 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?

```r
#first we set up a boosted tree model specification
bt_spec <- boost_tree(trees=tune()) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

#then we create a workflow and add our model and recipe
bt_wf <- workflow() %>%
  add_model(bt_spec) %>%
  add_recipe(pokemon_rec)

#now we create a regular grid for the trees hyperparameter, then tune our workflow
bt_grid <- grid_regular(trees(range=c(10,2000)), levels=10)

bt_tuned <- tune_grid(
  bt_wf,
  resamples=pokemon_folds,
```

```
  grid=bt_grid,
  metrics = metric_set(roc_auc)
  )

autoplot(bt_tuned)
```



From our outputted `autoplot()`, it appears that the `roc_auc` value peaks just before `trees = 250`, and then fluctuates as the number of trees continues to increase.

```
#using collect_metrics() and arrange(), we can create a set of the top roc auc models
bt_roc <- collect_metrics(bt_tuned)
ordered_bt_roc <- arrange(bt_roc, desc(bt_roc$mean))

#output the highest roc auc
head(ordered_bt_roc,1)
```

```
## # A tibble: 1 x 7
##   trees .metric .estimator  mean     n std_err .config
##   <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1   231 roc_auc hand_till  0.710     5  0.0269 Preprocessor1_Model02
```

According to the ordered metrics dataset above, the highest `roc_auc` is approximately 0.702.

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.

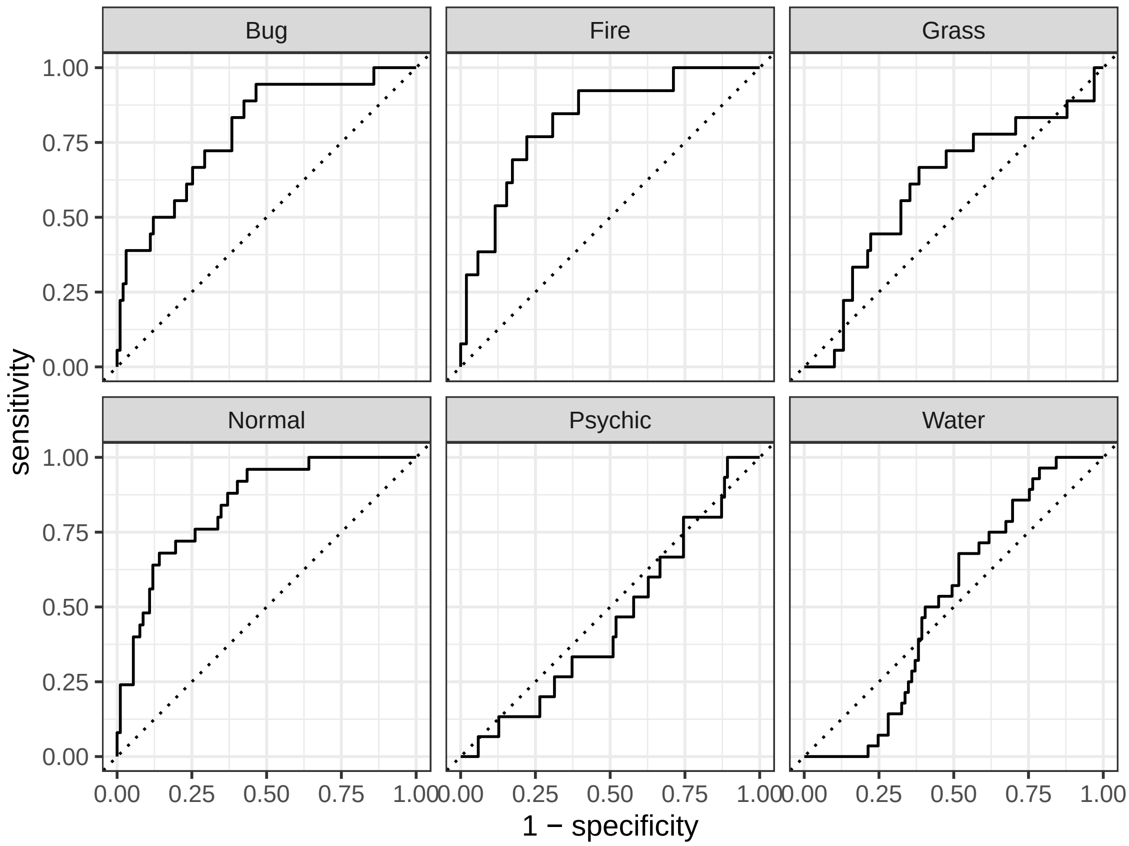
Which classes was your model most accurate at predicting? Which was it worst at?

```
best_roc_bt <- ordered_bt_roc$mean
best_roc_bt <- best_roc_bt[1]

best_roc_dt <- ordered_metrics$mean
best_roc_dt <- best_roc_dt[1]

best_roc_rf <- ordered_rf_met$mean
best_roc_rf <- best_roc_rf[1]


rocauc <- c(best_roc_dt, best_roc_rf,
                best_roc_bt)
models <- c("Pruned Tree", "Random Forest", "Boosted Tree")
results <- tibble(ROC_AUC = rocauc, models = models)

results
```

```
## # A tibble: 3 x 2
##   ROC_AUC models
##     <dbl> <chr>
## 1   0.649 Pruned Tree
## 2   0.730 Random Forest
## 3   0.710 Boosted Tree
```

From the values of `roc_auc` above, it seems that the Random Forest model performed best on the folds. We will now fit it to the testing set.

```
set.seed(4567)
finalized_model <- select_best(tuned_rf, metric="roc_auc")
finalized_workflow <- finalize_workflow(rf_wf, finalized_model)
finalized_fit <- fit(finalized_workflow, data=pokemon_train)
final_fit_test <- augment(finalized_fit, new_data = pokemon_test)

#first we will print the ROC curves for our model
final_fit_test %>%
  roc_curve(type_1,
            c(.pred_Bug,.pred_Fire,.pred_Grass,.pred_Normal,.pred_Water,.pred_Psychic)) %>%
  autoplot()
```

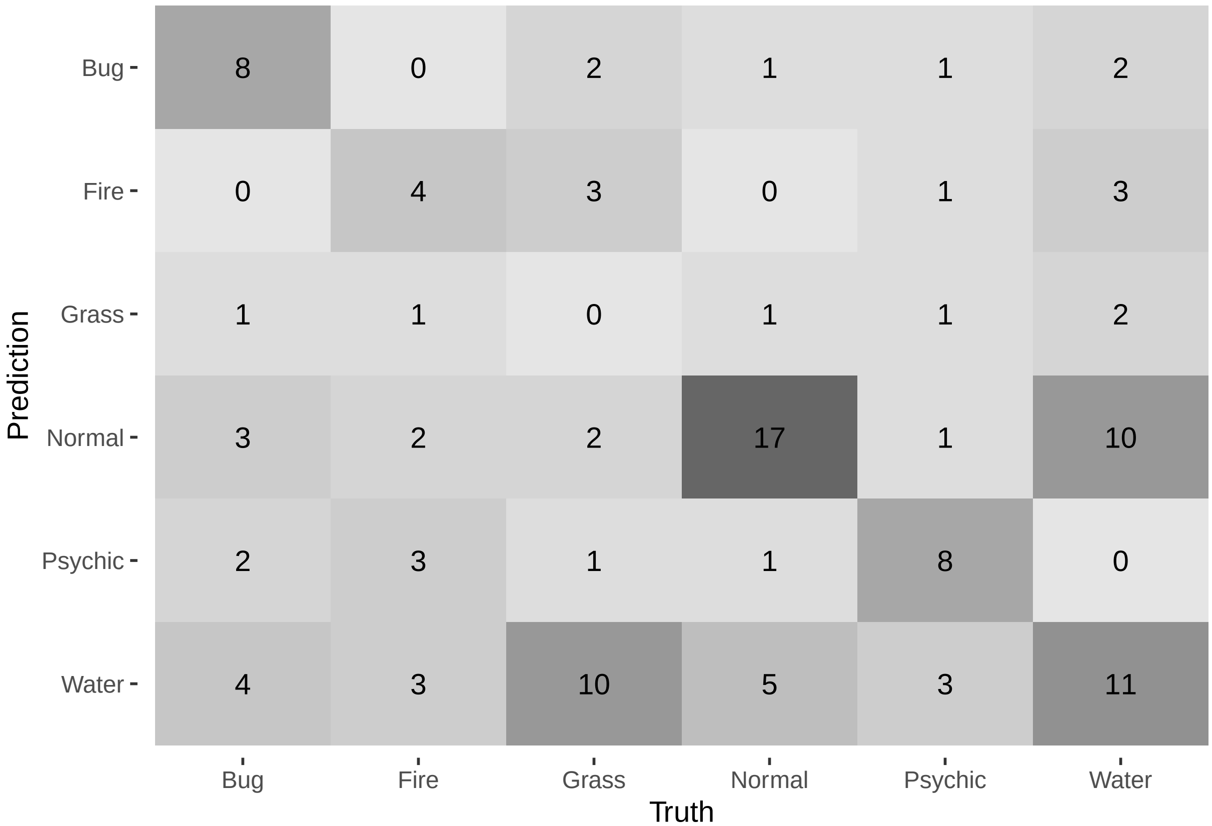



```r
#now we will print the AUC value of our best performing model on the testing set
final_fit_test %>%
  roc_auc(truth=type_1,
          estimate=c(.pred_Bug,.pred_Fire,.pred_Grass,.pred_Normal,.pred_Water,.pred_Psychic))
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.668
```

As expected, the ROC AUC drops significantly when the model trained on the training data is applied to new testing data, but it is still fairly effective.

```r
#finally we will create a confusion matrix heat map
final_fit_test %>%
  conf_mat(truth=type_1, estimate= .pred_class) %>%
  autoplot(type="heatmap")
```



As we can see, the model is very good at successfully predicting Pokémon of types "Normal", "Psychic", and "Bug". It is the worst at predicting Pokémon of type "Grass" (no Grass-type Pokémon from the testing set were successfully predicted as such).