Homework 6

PSTAT 131/231

Contents

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`:

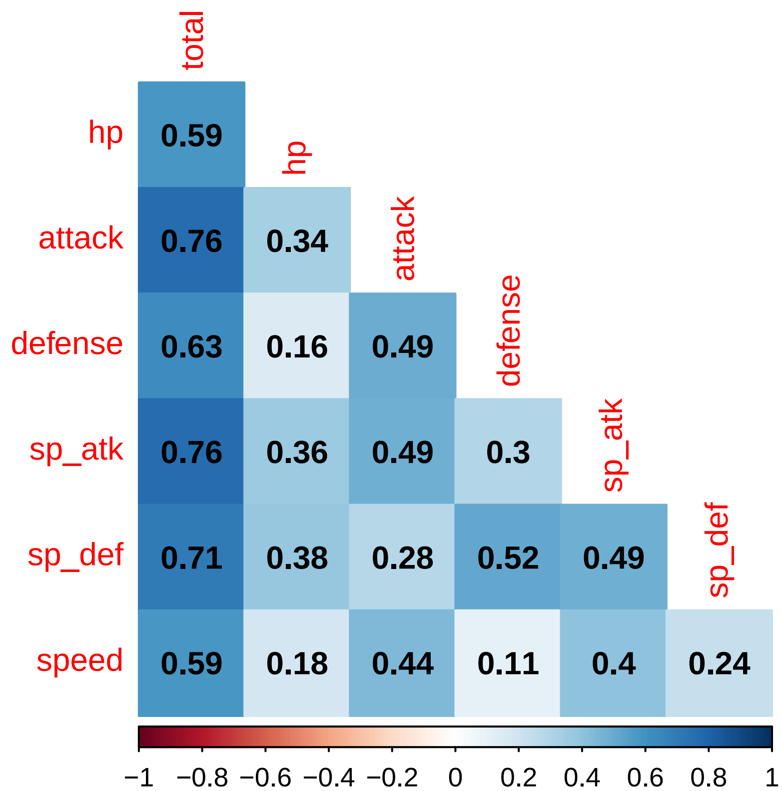
- Dummy-code `legendary` and `generation`;
- Center and scale all predictors.

```
set.seed(1008)
pokemon <- clean_names(pokemon)
type_variables <- c("Bug",'Fire', 'Grass',"Normal","Water","Psychic")
pokemon_edit <- filter(pokemon, pokemon$type_1 %in% type_variables)
pokemon_edit$type_1<-as.factor(pokemon_edit$type_1)
pokemon_edit$legendary<-as.factor(pokemon_edit$legendary)
pokemon_edit$generation<-as.factor(pokemon_edit$generation)
pokemon_split <- initial_split(pokemon_edit, strata = type_1, prop = 0.7)
pokemon_train <- training(pokemon_split)
pokemon_test <- testing(pokemon_split)
pokemon_folds <- vfold_cv(pokemon_train, v = 5,strata=type_1)
pokemon_recipe <- recipe(type_1 ~ legendary + generation + sp_atk + attack + speed + defense + hp + sp_
  step_dummy(generation) %>%
  step_dummy(legendary) %>%
  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).*

```r
pokemon_train2 <- pokemon_train[,sapply(pokemon_train,is.numeric)]
pokemon_train2 %>%
  dplyr::select(-number) %>%
  cor() %>%
  corrplot(type = 'lower', diag = FALSE,
           method = 'color',addCoef.col = 'Black')
```



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

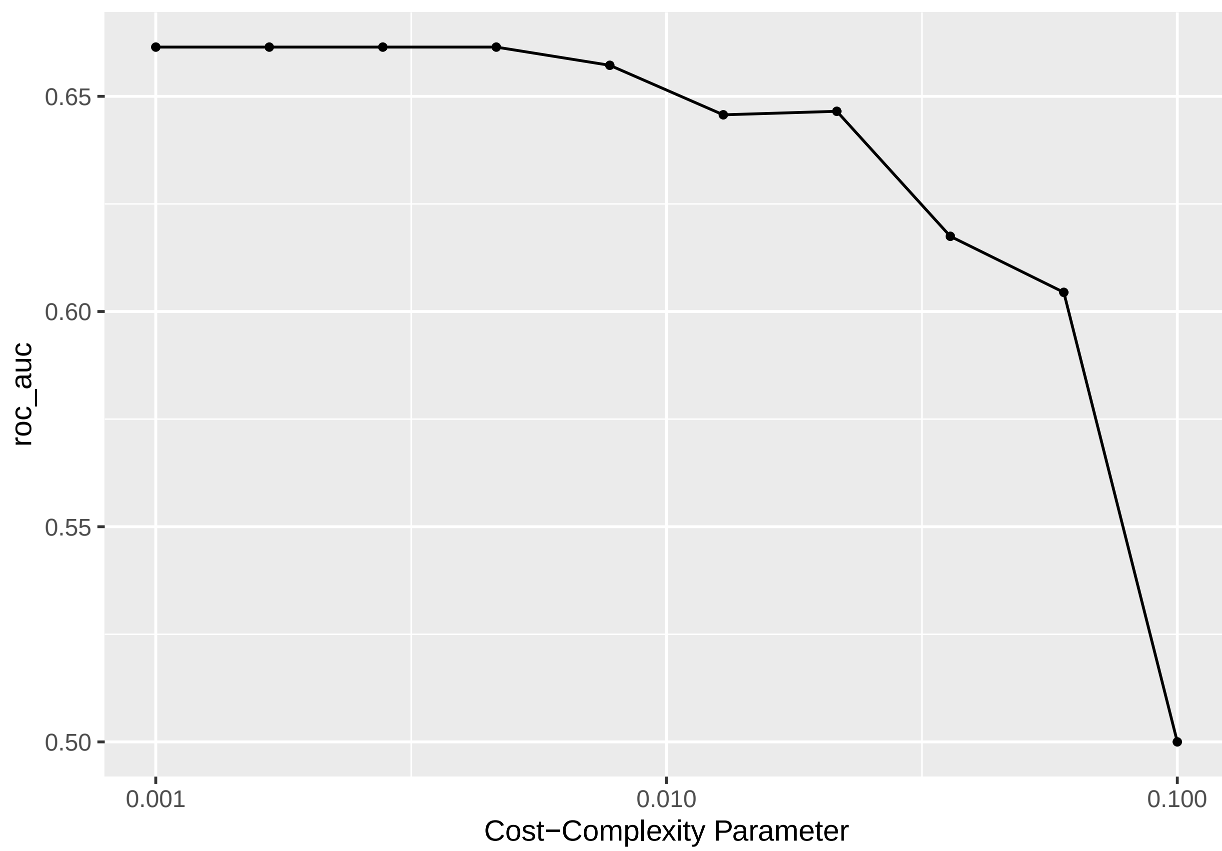
I chose not to include the number and generations because both numbers are more akin to indexes, and therefore, would have no correlation with various stats. Additionally, to my knowledge, corrplot() doesn't allow factors which is why i will also be omitting legendaries from the correlation matrix. As expected, the total stats are highly correlated with all of the other stats. This is especially true for attack, sp_atk, and sp_def. This would indicate that the other stats, hp, defense, and speed, are less influential in a Pokemon's total stats. For example, there may be several Pokemon with a high HP but a low stat total. Reaffirming this, we can see that the correlations between hp/x, defense/x, and sp_def/x seem to be lower on average than attack/x, sp_attack/x, and sp_def/x.

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?

```r
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)

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



The single decision tree seems to perform the same at a cost-complexity parameter of around .001 to .005 but takes a significant drop in performance when approaching .100. From a parameter of .005 to .100, the roc_auc seems to drop around 16%.

Exercise 4

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

```r
collect_metrics(tune_res)%>%
  arrange(-mean)
```

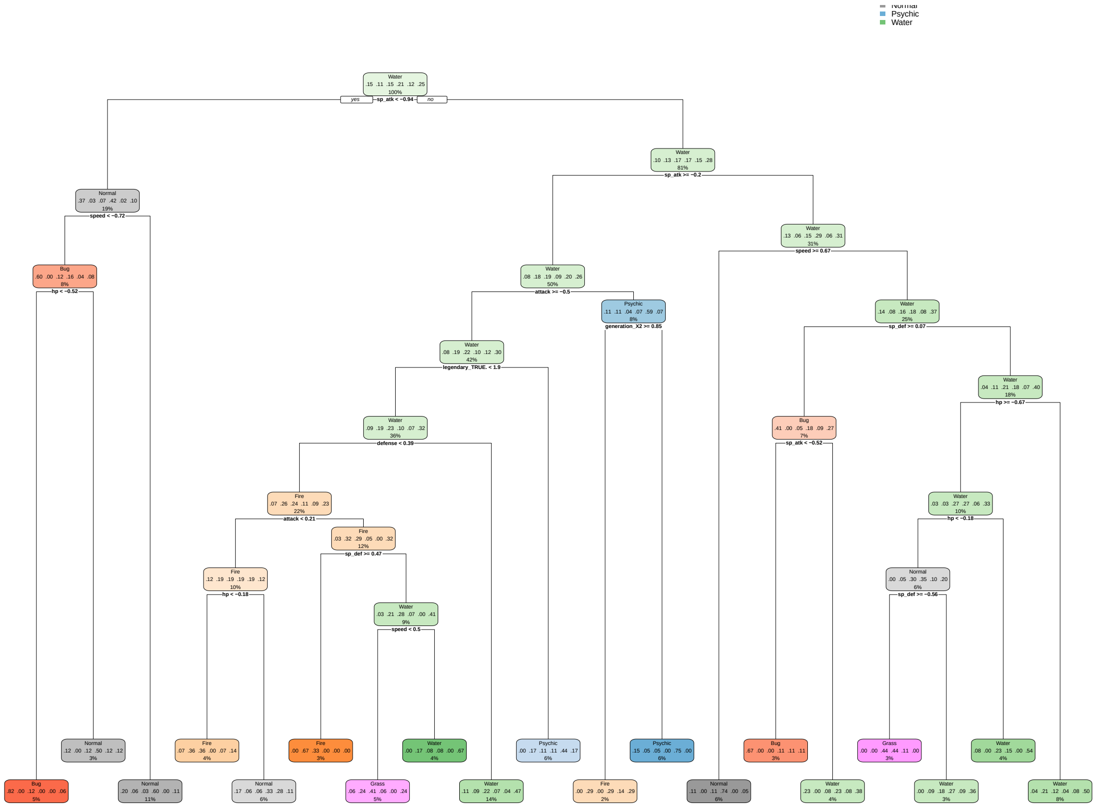
```
## # A tibble: 10 x 7
##    cost_complexity .metric .estimator  mean     n std_err .config
##              <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1         0.001   roc_auc hand_till  0.661     5  0.0195 Preprocessor1_Model01
##  2         0.00167 roc_auc hand_till  0.661     5  0.0195 Preprocessor1_Model02
##  3         0.00278 roc_auc hand_till  0.661     5  0.0195 Preprocessor1_Model03
##  4         0.00464 roc_auc hand_till  0.661     5  0.0195 Preprocessor1_Model04
##  5         0.00774 roc_auc hand_till  0.657     5  0.0206 Preprocessor1_Model05
##  6         0.0215  roc_auc hand_till  0.647     5  0.0224 Preprocessor1_Model07
##  7         0.0129  roc_auc hand_till  0.646     5  0.0156 Preprocessor1_Model06
##  8         0.0359  roc_auc hand_till  0.617     5  0.0160 Preprocessor1_Model08
##  9         0.0599  roc_auc hand_till  0.604     5  0.0193 Preprocessor1_Model09
## 10         0.1     roc_auc hand_till  0.5       5  0      Preprocessor1_Model10
```

The roc_auc of my best performing decision tree was around .661.

Exercise 5

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

```r
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 = pokemon_train)

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?**

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

Mtry represents the number of of predictors that are randomly sampled at each split of the tree model,trees represents the total amount of trees in the ensemble, and min_n is represents the minimum number of observations before splitting nodes.

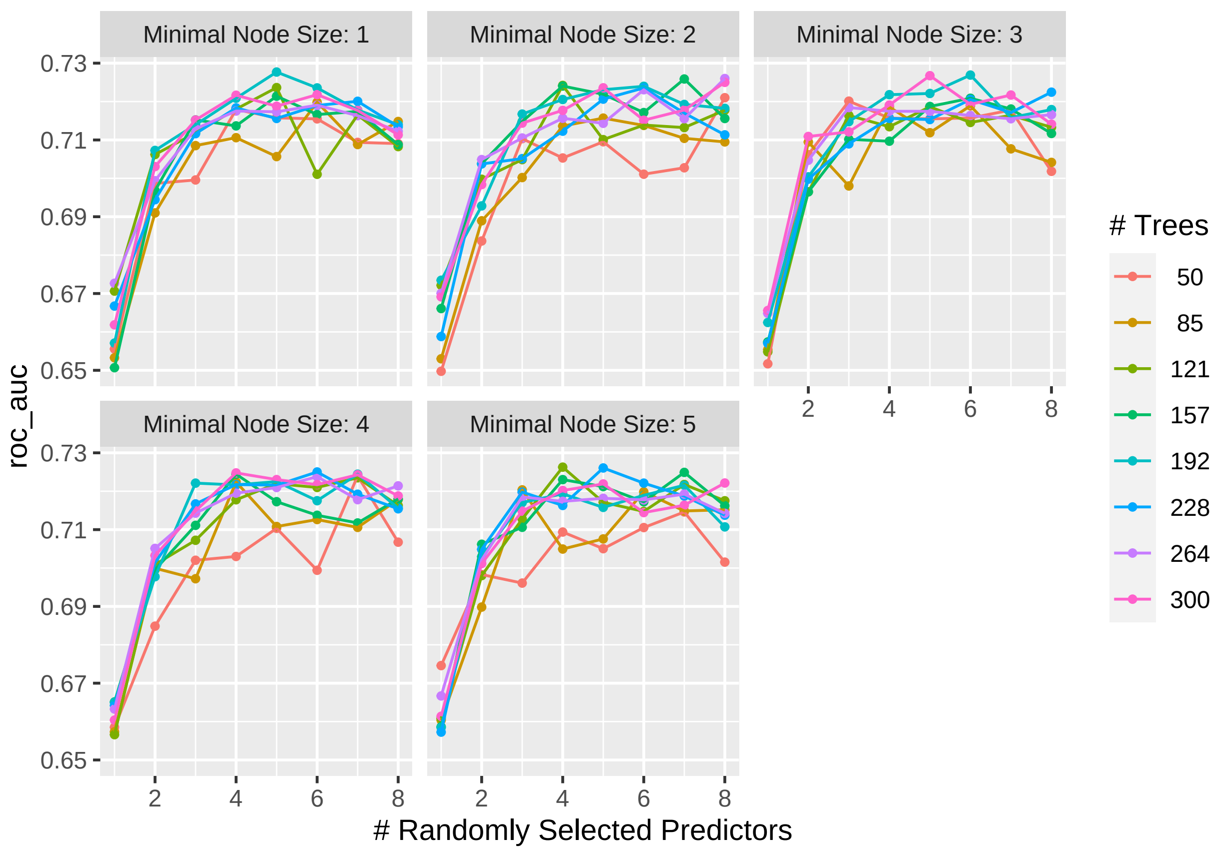
```
param_grid2 <- grid_regular(mtry(range = c(1, 8)),trees(range = c(50,300)),min_n(range=c(1,5)), levels =
```

If mtry is equal to the amount of columns, then it would be a bagging 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 hyperparameters seem to yield the best performance?

```
tune_res2 <- tune_grid(
  rf_wf,
  resamples = pokemon_folds,
  grid = param_grid2,
  metrics = metric_set(roc_auc)
)
autoplot(tune_res2)
```



According to the roc_auc, it seems that 4-6 randomly selected predictors seems to yield the highest accuracy. This is unexpectedly higher than the square root of the maximum number of observations. The optimal amount of trees is different depending on the values of the other hyperparameters. However, the larger amount of trees (192-300) seem to consistently perform better while the smaller amount of trees are more inconsistent. Finally, the minimal node sizes of 4 and 5 seem to yield the best accuracy. However, there is one point with a minimal node size of 1 that is unusually large.

Exercise 7

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

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

```
## # A tibble: 320 x 9
##     mtry trees min_n .metric .estimator  mean     n std_err .config
##    <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1     5   192     1 roc_auc hand_till  0.728     5  0.0253 Preprocessor1_Model~
##  2     6   192     3 roc_auc hand_till  0.727     5  0.0266 Preprocessor1_Model~
##  3     5   300     3 roc_auc hand_till  0.727     5  0.0251 Preprocessor1_Model~
##  4     4   121     5 roc_auc hand_till  0.726     5  0.0226 Preprocessor1_Model~
##  5     5   228     5 roc_auc hand_till  0.726     5  0.0263 Preprocessor1_Model~
##  6     8   264     2 roc_auc hand_till  0.726     5  0.0263 Preprocessor1_Model~
##  7     7   157     2 roc_auc hand_till  0.726     5  0.0258 Preprocessor1_Model~
##  8     6   228     4 roc_auc hand_till  0.725     5  0.0265 Preprocessor1_Model~
##  9     8   300     2 roc_auc hand_till  0.725     5  0.0265 Preprocessor1_Model~
## 10     7   157     5 roc_auc hand_till  0.725     5  0.0237 Preprocessor1_Model~
## # ... with 310 more rows
```

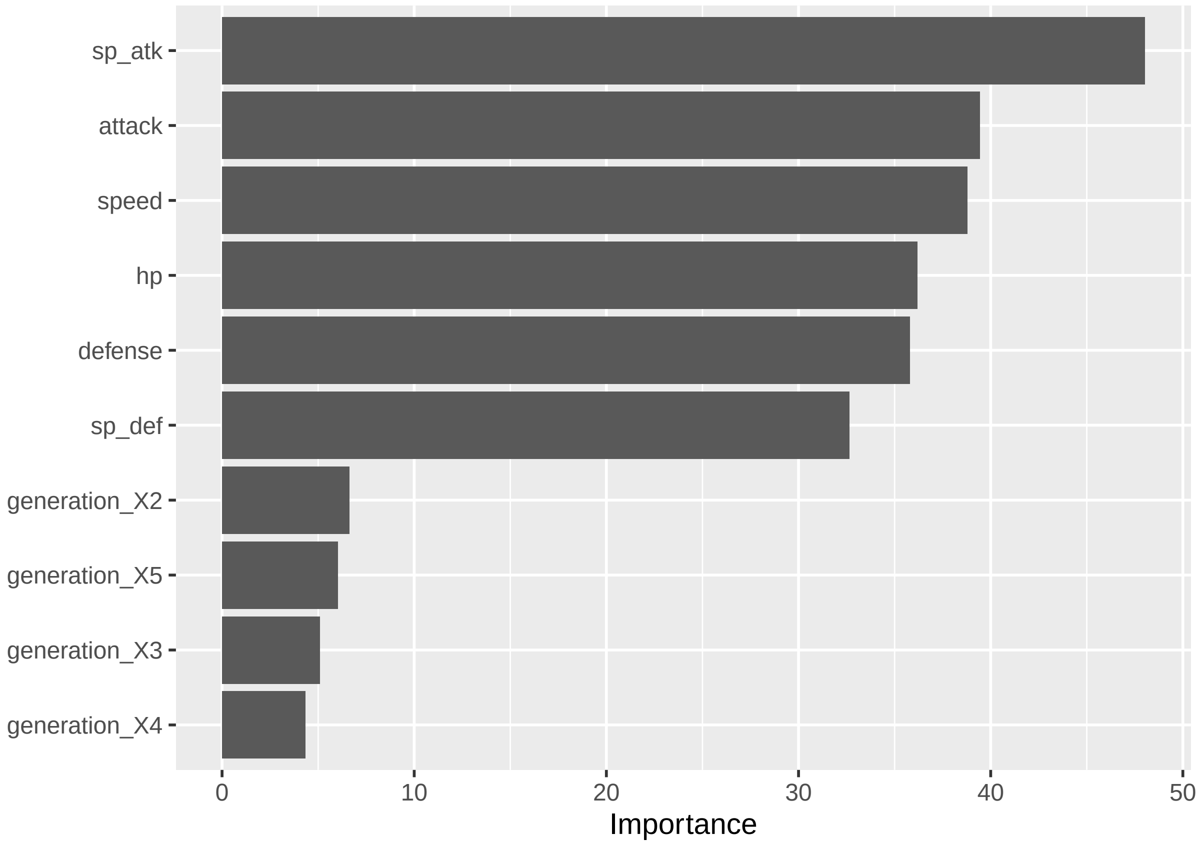
My best performing model has an roc_auc of .728.

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?

```
best_complexity2 <- select_best(tune_res2)

rf_final <- finalize_workflow(rf_wf, best_complexity2)

rf_final_fit <- fit(rf_final, data = pokemon_train)

rf_final_fit %>%
  extract_fit_engine() %>%
  vip()
```



Like expected, the numeric variables seem to be the most important for our model. Sp_atk, attack and speed, were all especially important in comparison to other stats. It makes sense that special attack and attack are both relatively important since their correlation together was high. Similarly, sp_def and defense, which also had a high correlation, are lower in importance. On the other hand, legendary status and generation both seemed to be rather negligible. This is to be expected since generations seem to be more of an index as opposed to a meaningful classification and the type_1 of legendary/non-legendary pokemon is broad(making legendary status unimportant for determining type).

Exercise 9

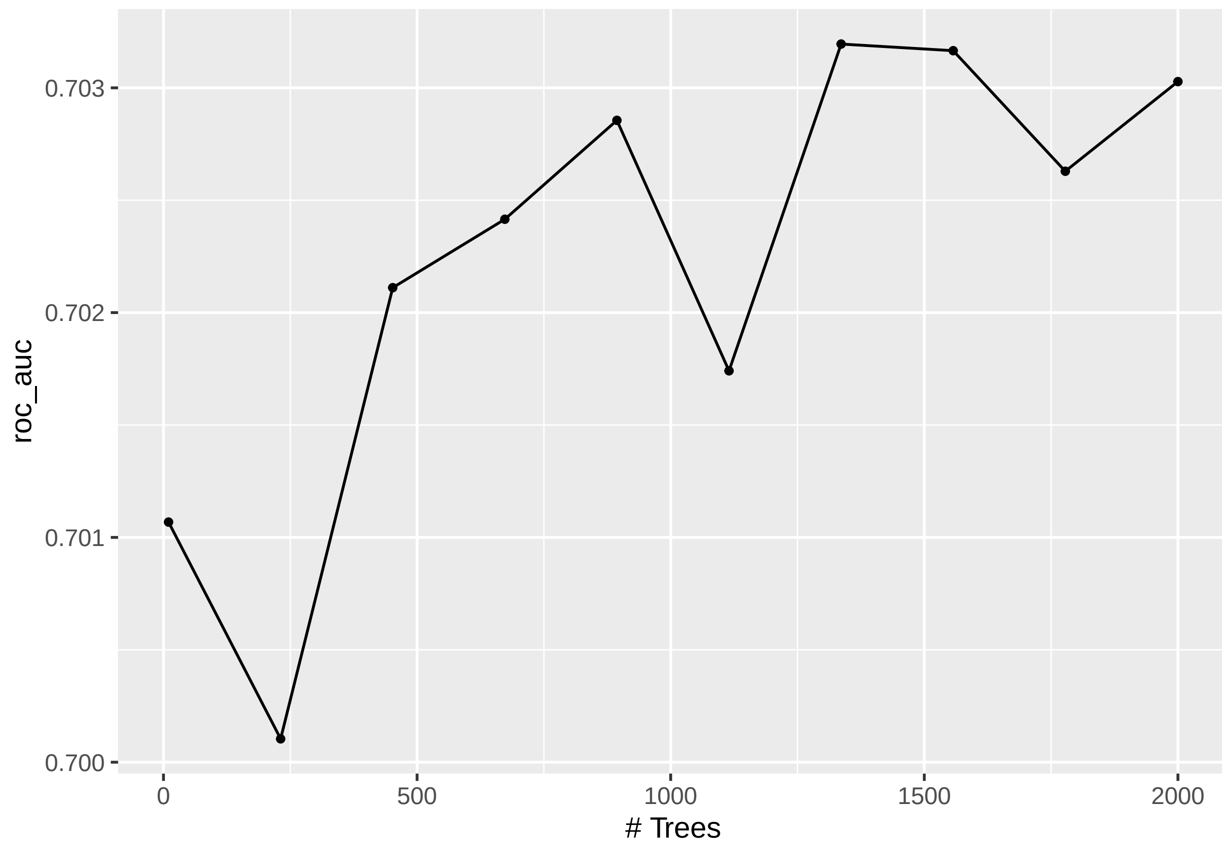
Finally, set up a boosted tree model and workflow. Use the `xgboost` engine. Tune `trees`. Create a regular grid with 10 levels; let `trees` range from 10 to 2000. Specify `roc_auc` and again print an `autoplot()` of the results.

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()`.*

```
bt_spec <- boost_tree() %>%
  set_engine("xgboost") %>%
  set_mode("classification")
bt_wf <- workflow() %>%
  add_model(bt_spec %>% set_args(trees = tune())) %>%
  add_recipe(pokemon_recipe)
param_grid3 <- grid_regular(trees(range = c(10,2000)), levels = 10)
```

```r
tune_res3 <- tune_grid(
  bt_wf,
  resamples = pokemon_folds,
  grid = param_grid3,
  metrics = metric_set(roc_auc)
)
autoplot(tune_res3)
```



While the graph seems to fluctuate significantly, the highest and lowest roc_auc values of the graph differ by only .003. It does seem that a higher amount of trees will increase accuracy very slightly.

```r
collect_metrics(tune_res3)%>%
  arrange(-mean)
```

```
## # A tibble: 10 x 7
##    trees .metric .estimator  mean     n std_err .config
##    <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1  1336 roc_auc hand_till  0.703     5  0.0183 Preprocessor1_Model07
##  2  1557 roc_auc hand_till  0.703     5  0.0184 Preprocessor1_Model08
##  3  2000 roc_auc hand_till  0.703     5  0.0190 Preprocessor1_Model10
##  4   894 roc_auc hand_till  0.703     5  0.0171 Preprocessor1_Model05
##  5  1778 roc_auc hand_till  0.703     5  0.0187 Preprocessor1_Model09
##  6   673 roc_auc hand_till  0.702     5  0.0173 Preprocessor1_Model04
##  7   452 roc_auc hand_till  0.702     5  0.0173 Preprocessor1_Model03
##  8  1115 roc_auc hand_till  0.702     5  0.0174 Preprocessor1_Model06
```

```
## 9     10 roc_auc hand_till  0.701     5  0.0175 Preprocessor1_Model01
## 10   231 roc_auc hand_till  0.700     5  0.0172 Preprocessor1_Model02
```

The roc_auc of the best performing boosted tree model is around .703.

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.

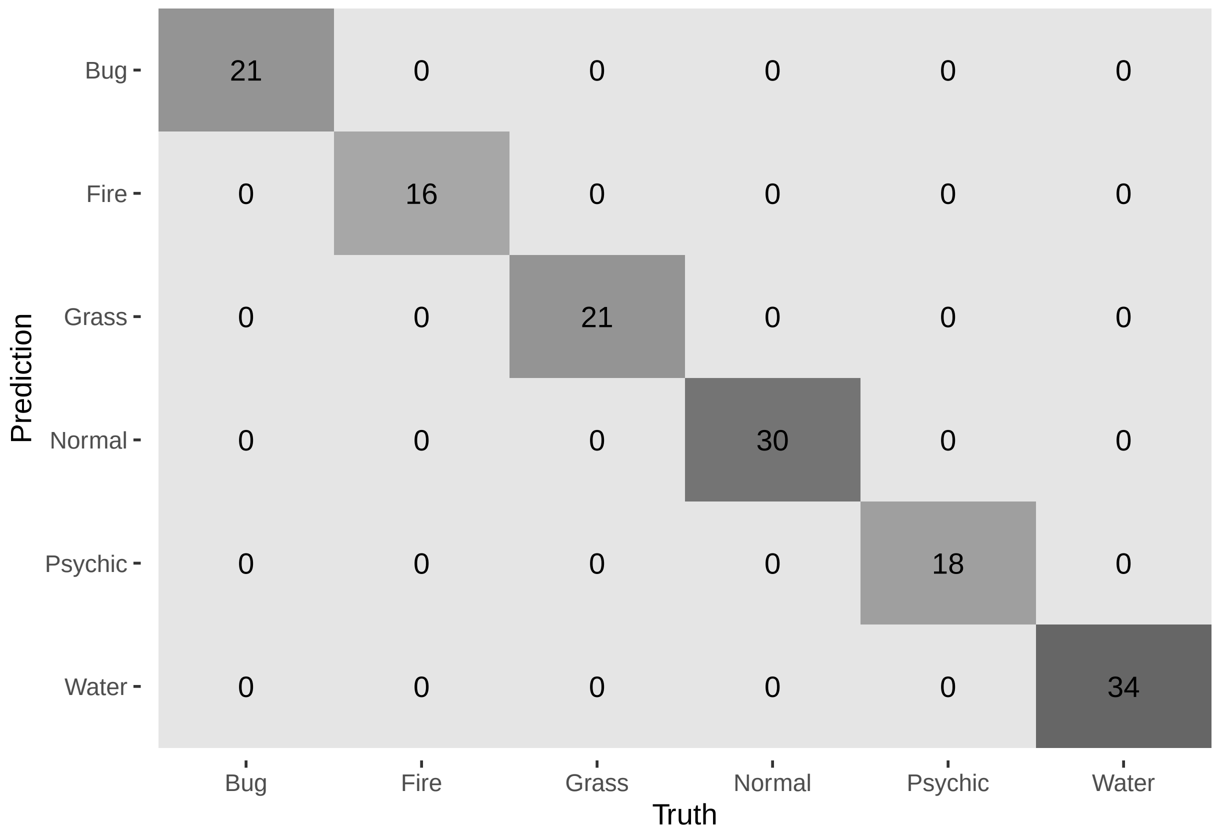
Which classes was your model most accurate at predicting? Which was it worst at?

```
a<-collect_metrics(tune_res)%>%
  arrange(-mean)%>%
  filter(row_number()==1)
a['type']<-'pt'
b<-collect_metrics(tune_res2)%>%
  arrange(-mean)%>%
  filter(row_number()==1)
b['type']<-'rf'
c<-collect_metrics(tune_res3)%>%
  arrange(-mean)%>%
  filter(row_number()==1)
c['type']<-'bt'
inner1<-rbind(a[2:8],b[4:10])
inner2<-rbind(inner1,c[2:8])
inner2
```

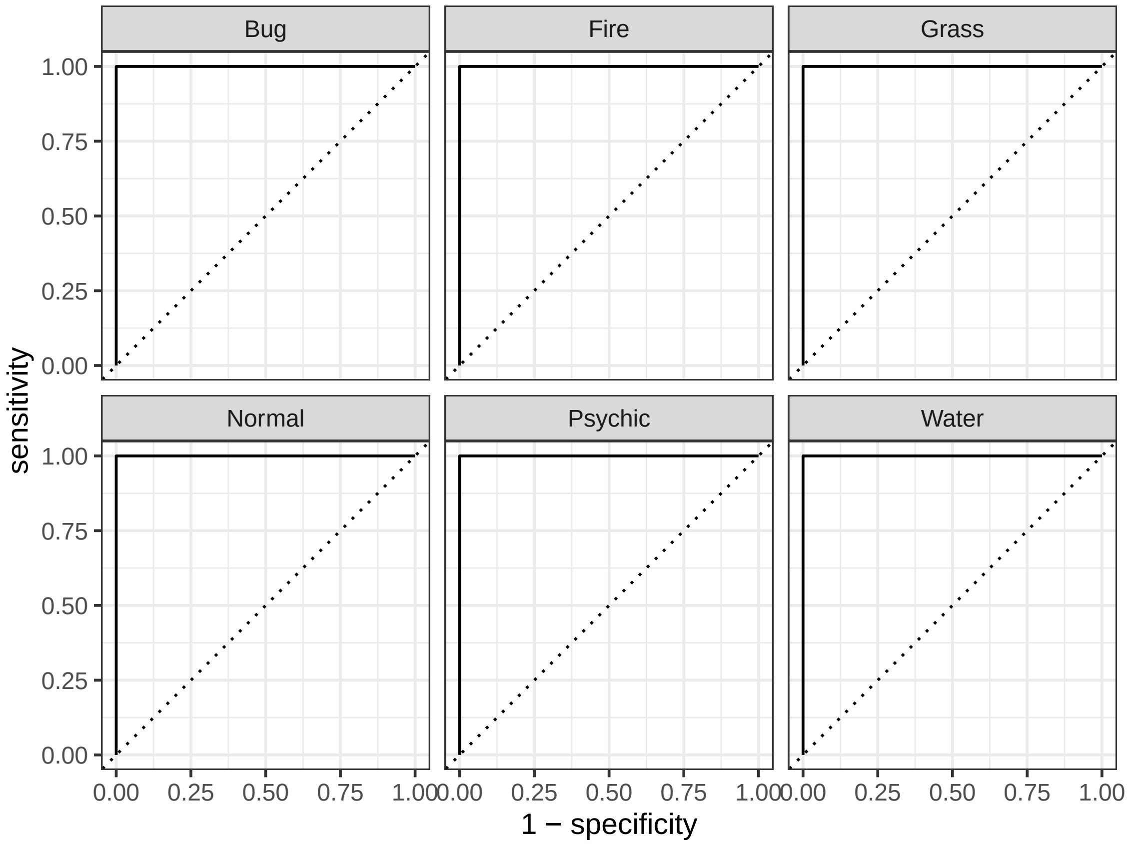
```
## # A tibble: 3 x 7
##   .metric .estimator  mean     n std_err .config                type
##   <chr>   <chr>      <dbl> <int>   <dbl> <chr>                  <chr>
## 1 roc_auc hand_till  0.661     5  0.0195 Preprocessor1_Model01  pt
## 2 roc_auc hand_till  0.728     5  0.0253 Preprocessor1_Model037 rf
## 3 roc_auc hand_till  0.703     5  0.0183 Preprocessor1_Model07  bt
```

The random forest performed best on the folds.

```
best_complexity2 <- select_best(tune_res2)

rf_final2 <- finalize_workflow(rf_wf, best_complexity2)

rf_final_fit2 <- fit(rf_final, data = pokemon_test)

augment(rf_final_fit2, new_data = pokemon_test) %>%
  conf_mat(truth = type_1, estimate = .pred_class) %>%
  autoplot(type = "heatmap")
```




```
augment(rf_final_fit2, new_data = pokemon_test) %>%
  roc_curve(type_1,.pred_Bug,.pred_Fire,.pred_Grass,.pred_Normal,.pred_Psychic,.pred_Water) %>%
  autoplot()
```




```r
augment(rf_final_fit2, new_data = pokemon_test) %>%
  roc_auc(type_1,.pred_Bug,.pred_Fire,.pred_Grass,.pred_Normal,.pred_Psychic,.pred_Water)
```

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

Because the model was fit to the testing set, all of the classes were identified with 100% accuracy.