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

```
pokemon <- pokemon <- read.csv("C:/Users/rocke/Downloads/homework-5/homework-5/data/Pokemon.csv")
pokemon <- clean_names(pokemon)
target <- c("Bug", "Fire", "Grass", "Normal", "Water", "Psychic")
pokemon <- pokemon %>%
  filter(type_1 %in% target)
pokemon$type_1 <- factor(pokemon$type_1)
pokemon$legendary <- factor(pokemon$legendary)
pokemon$generation <- factor(pokemon$generation)
```

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

```
set.seed(1234)
pokemon_split <- initial_split(pokemon, prop = 0.7, strata = type_1)
pokemon_train <- training(pokemon_split)
pokemon_test <- testing(pokemon_split)
dim(pokemon_split)  # verifying training and testing sets have the desired number of observations
```

```
##   analysis assessment          n          p
##        318        140        458         13
```

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

```
pokemon_folds <- vfold_cv(pokemon_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`:

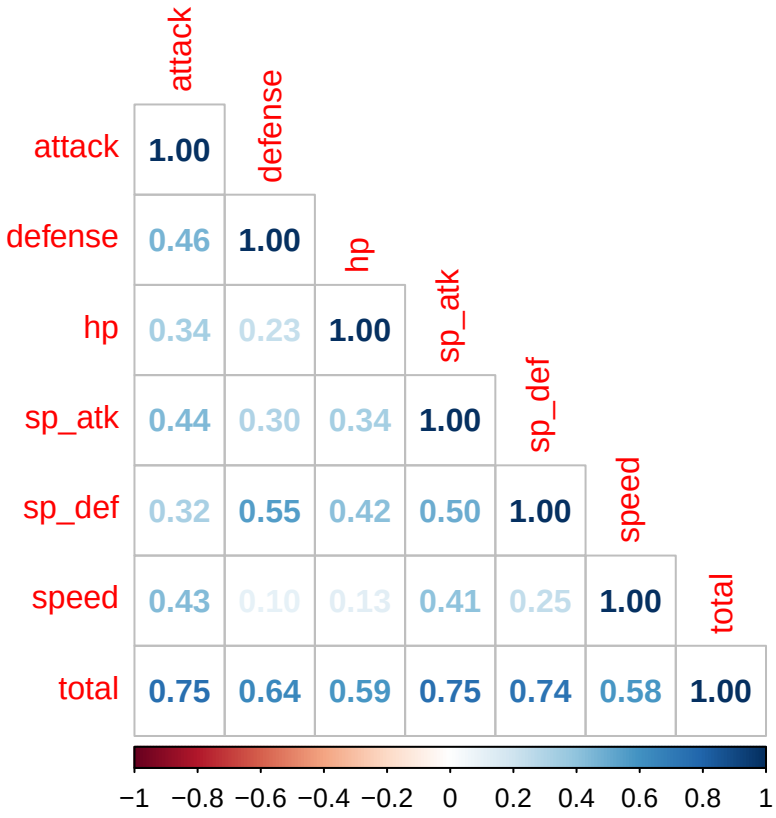
- Dummy-code `legendary` and `generation`;
- Center and scale all predictors.

```
pokemon_recipe <- recipe(type_1 ~ legendary + generation + sp_atk + attack + speed + defense + hp + sp_
  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).*

```
pokemon_continuous <- pokemon[, c(5, 6, 7, 8, 9, 10, 11)]
pokemon_cor <- cor(pokemon_continuous)
corrplot(pokemon_cor, method = "number", order = "alphabet", type = "lower")
```



I kept only the variables that were continuous in nature, so I did not include legendary or generation. I opted not to include the variables that were not in our recipe as well, except for total.

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

I notice that attack and defense, sp_atk and attack, speed and attack, sp_def and defense, hp and sp_def, sp_atk and sp_def, and speed and sp_atk were moderately (positively) correlated. Additionally, total is quite positively associated with all other variables. I agree with most of the relationships, except for attack and defense. I would have thought that attack and defense would be negatively correlated to balance out the pokemon stats, but their positive association is quite strong!

Exercise 3

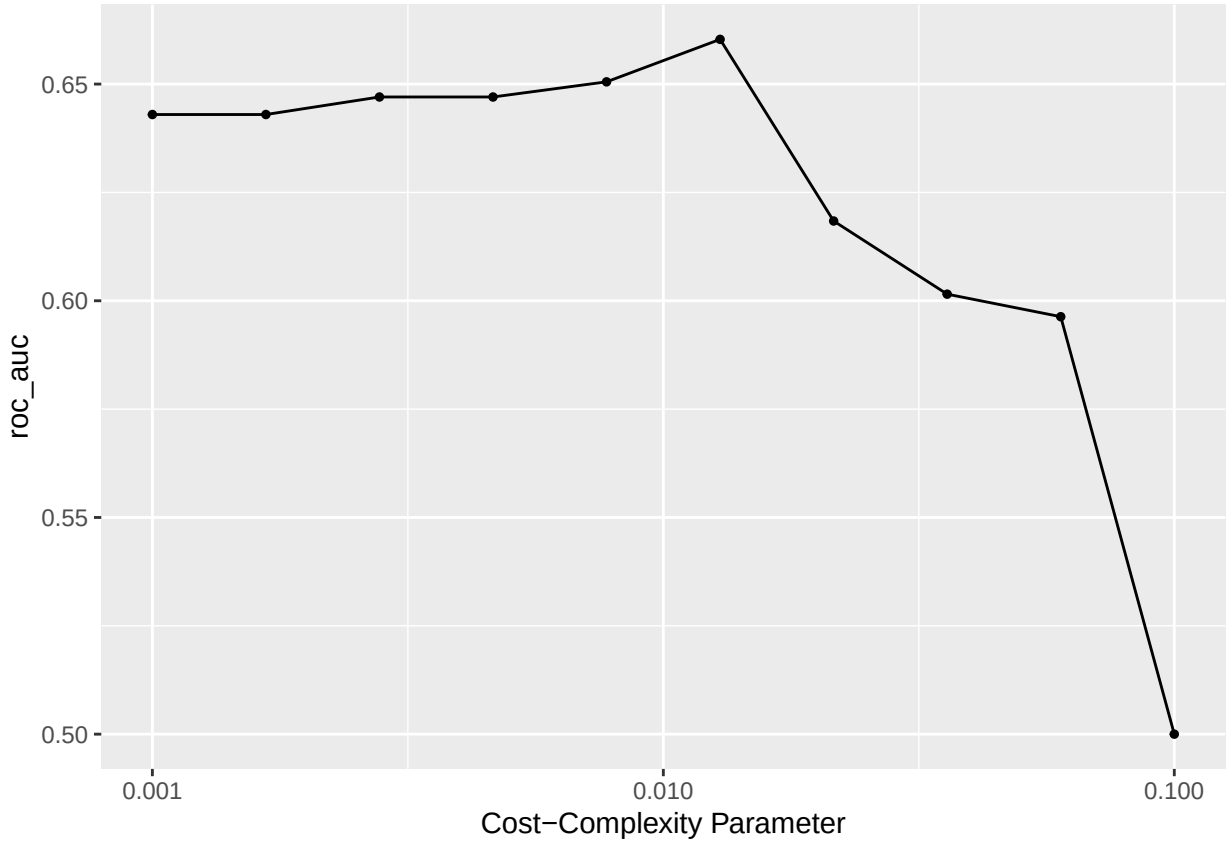
First, set up a decision tree model and workflow. Tune the `cost_complexity` hyperparameter. Use the same levels we used in Lab 7 – that is, `range = c(-3, -1)`. Specify that the metric we want to optimize is `roc_auc`.

```
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_result <- tune_grid(
  class_tree_wf,
  resamples = pokemon_folds,
  grid = param_grid,
  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_result)
```



I observe that our model starts off well and improves until a complexity parameter of about 0.013, then drops significantly. For the most part, a single decision tree performs better with a smaller complexity penalty.

Exercise 4

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

```
decision_tree <- collect_metrics(tune_result) %>%
  arrange(desc(mean))
decision_tree
```

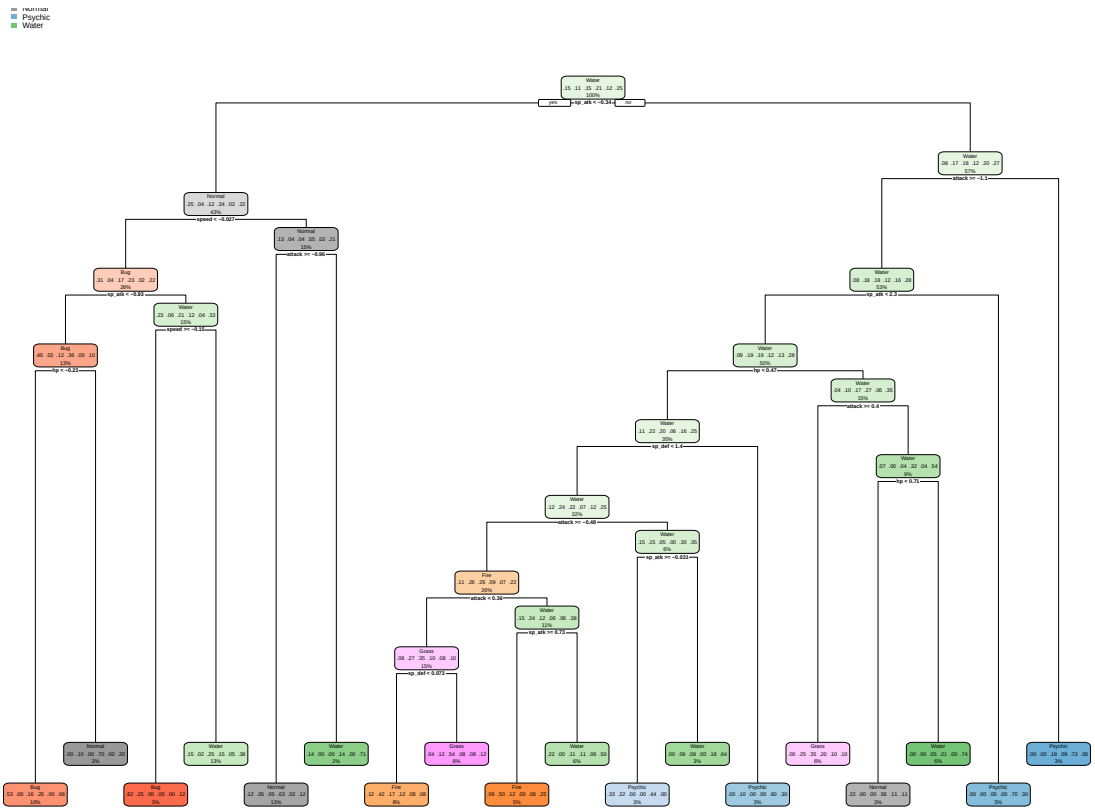
```
## # A tibble: 10 x 7
##    cost_complexity .metric .estimator  mean     n std_err .config
##              <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1         0.0129  roc_auc hand_till  0.660     5 0.0141  Preprocessor1_Model06
##  2         0.00774 roc_auc hand_till  0.651     5 0.0163  Preprocessor1_Model05
##  3         0.00278 roc_auc hand_till  0.647     5 0.0186  Preprocessor1_Model03
##  4         0.00464 roc_auc hand_till  0.647     5 0.0186  Preprocessor1_Model04
##  5         0.001   roc_auc hand_till  0.643     5 0.0183  Preprocessor1_Model01
##  6         0.00167 roc_auc hand_till  0.643     5 0.0183  Preprocessor1_Model02
##  7         0.0215  roc_auc hand_till  0.618     5 0.00683 Preprocessor1_Model07
##  8         0.0359  roc_auc hand_till  0.602     5 0.0132  Preprocessor1_Model08
##  9         0.0599  roc_auc hand_till  0.596     5 0.0146  Preprocessor1_Model09
## 10         0.1     roc_auc hand_till  0.5       5 0       Preprocessor1_Model10
```

The roc_auc of the best performing pruned decision tree is 0.6603222.

Exercise 5

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

```
best_complexity <- select_best(tune_result)
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.

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

```
rf_wf <- workflow() %>%
  add_model(rf_spec) %>%
  add_recipe(pokemon_recipe)
```

Mtry is used to tune the number of predictors that will be sampled randomly at each split– this will aid in lowering the bias from the greedy approach.
Trees is used to tune the number of trees used in each ensemble.
Min_n is used to tune the minimum required number of data points in a node before it is split further.

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?**

```
parameter_grid <- grid_regular(mtry(range = c(1, 8)), trees(range = c(50, 150)), min_n(range = c(1, 15)))
```

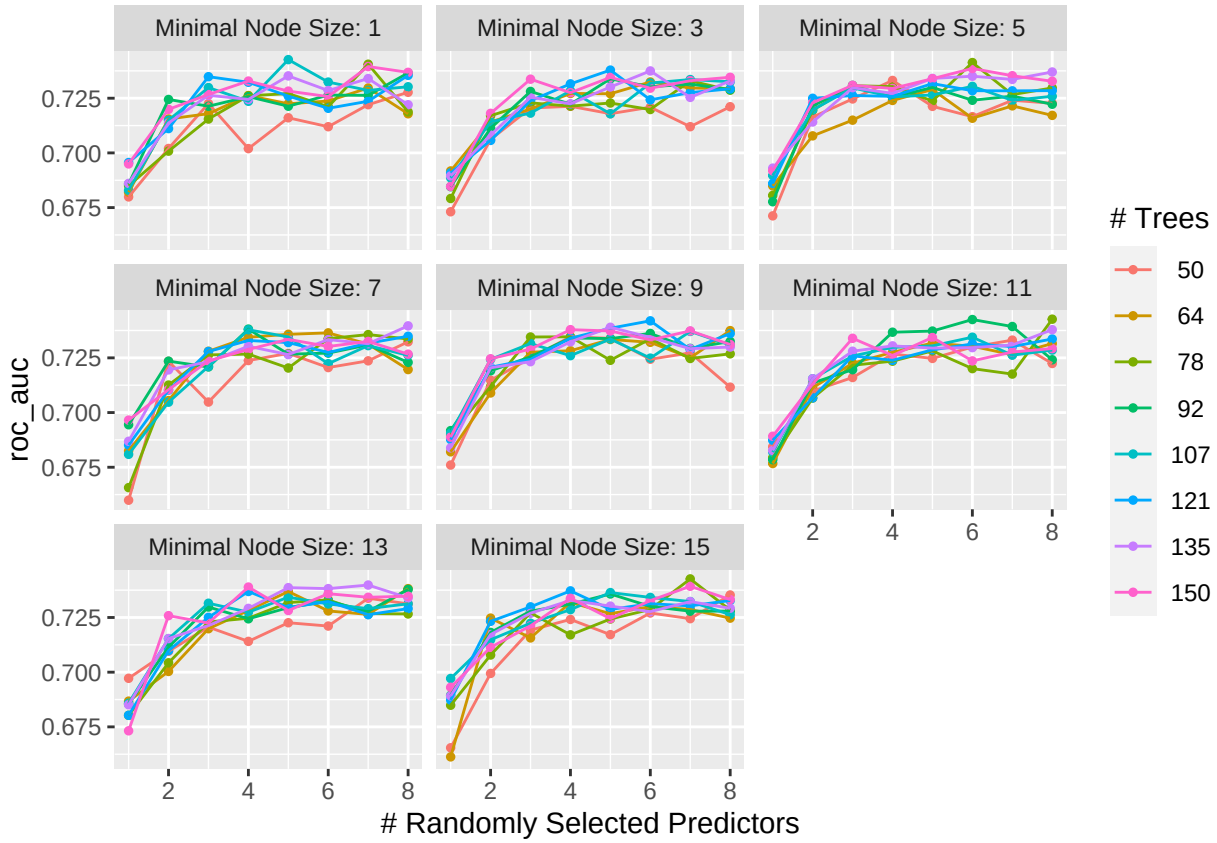
Due to our data, we cannot use mtry values that are less than 1 because then that would mean we will use 0 predictors to split, and we cannot use mtry values that are larger than 8 because we only have 8 predictors to choose from. An mtry value = 8 would represent a bagging model.

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_res <- tune_grid(
  rf_wf,
  resamples = pokemon_folds,
  grid = parameter_grid,
  metrics = metric_set(roc_auc)
)
```

```
autoplot(tune_res)
```



I observe that less trees tends to yield worse results, the results get better as the mtry values increase, and that a higher minimal node size tends to get better results. Though it really depends on the combination of value for the three hyperparameters. More specifically, the best values of hyperparameters that seem to yield the best performance are mtry = 8, trees = 78, and min_n = 11.

Exercise 7

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

```
random_forest <- collect_metrics(tune_res) %>%
  arrange(desc(mean))
random_forest
```

```
## # 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    78    11 roc_auc hand_till  0.743     5 0.0126  Preprocessor1_Model~
## 2     5   107     1 roc_auc hand_till  0.743     5 0.00970 Preprocessor1_Model~
## 3     7    78    15 roc_auc hand_till  0.743     5 0.0198  Preprocessor1_Model~
## 4     6    92    11 roc_auc hand_till  0.743     5 0.0126  Preprocessor1_Model~
## 5     6   121     9 roc_auc hand_till  0.742     5 0.0168  Preprocessor1_Model~
## 6     6    78     5 roc_auc hand_till  0.741     5 0.0133  Preprocessor1_Model~
## 7     7    78     1 roc_auc hand_till  0.741     5 0.0127  Preprocessor1_Model~
## 8     7   135    13 roc_auc hand_till  0.740     5 0.0144  Preprocessor1_Model~
## 9     8   135     7 roc_auc hand_till  0.740     5 0.00726 Preprocessor1_Model~
```

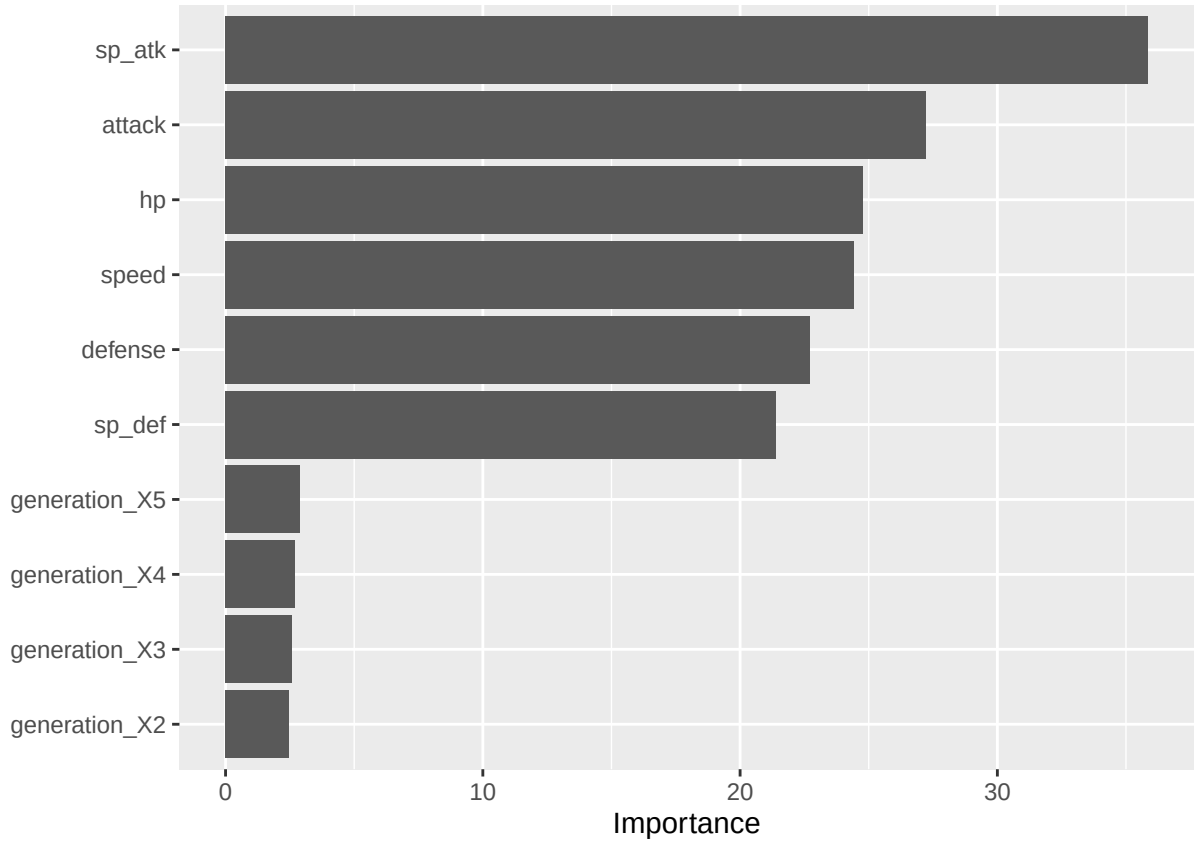
```
## 10     7   150     1 roc_auc hand_till  0.739     5 0.00548 Preprocessor1_Model~
## # ... with 502 more rows
```

The roc_auc of the best-performing random forest model is 0.7426766.

Exercise 8

Create a variable importance plot, using `vip()`, with your best-performing random forest model fit on the *training* set.

```
best_complexity <- select_best(tune_res)
rf_final <- finalize_workflow(rf_wf, best_complexity)
rf_final_fit <- fit(rf_final, data = pokemon_train)

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



Which variables were most useful? Which were least useful? Are these results what you expected, or not?

The most useful variable was sp_atk, followed by attack, hp, and speed. The least useful were the generation_x variables. I think these results are expected because certain types may be better predicted by their attributes (ex. fire type maybe has higher attack on average). Additionally, the generation that a pokemon was from isn't very telling for the pokemon type.

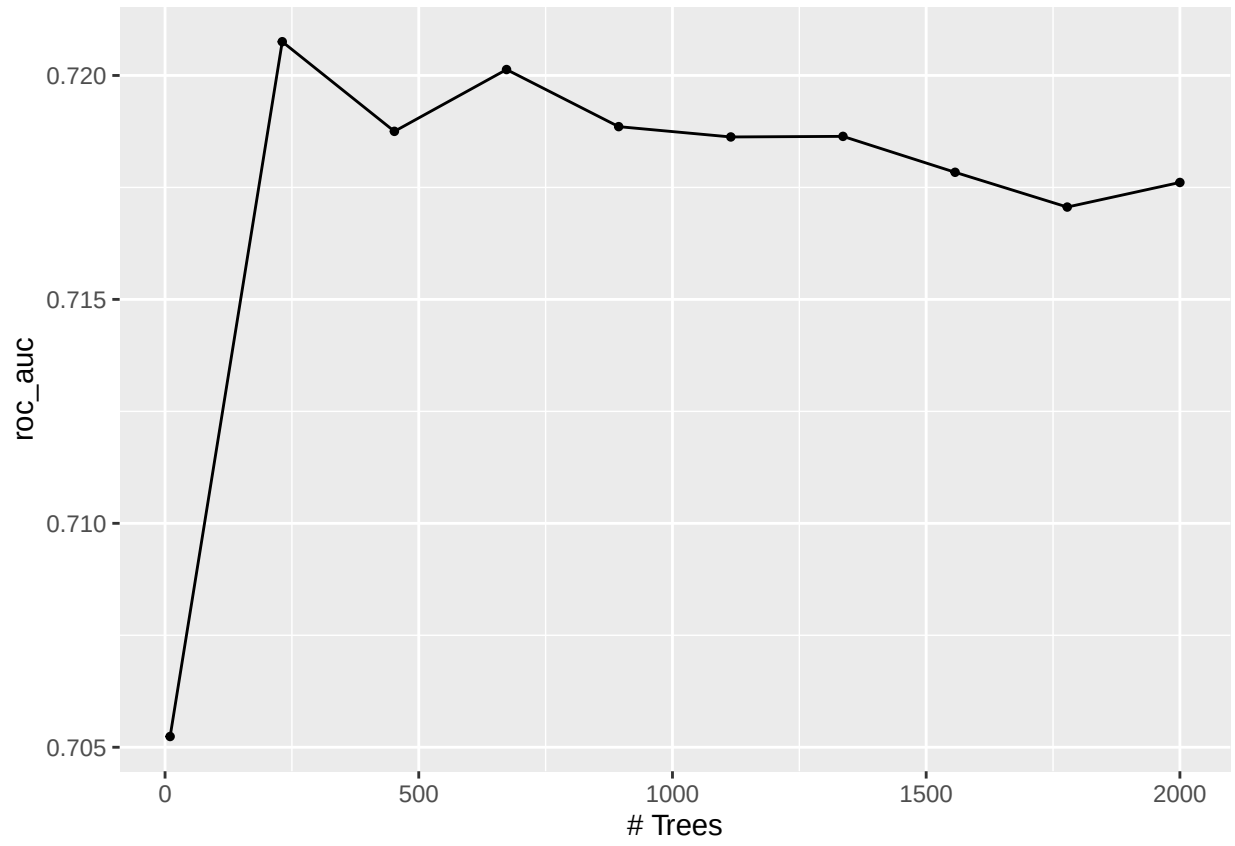
Exercise 9

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.

```
boost_spec <- boost_tree(trees = tune()) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

boost_wf <- workflow() %>%
  add_model(boost_spec) %>%
  add_recipe(pokemon_recipe)

tree_grid <- grid_regular(trees(range = c(10, 2000)), levels = 10)
```

```
tuning_res <- tune_grid(
  boost_wf,
  resamples = pokemon_folds,
  grid = tree_grid,
  metrics = metric_set(roc_auc)
)
```

```
autoplot(tuning_res)
```



What do you observe?

I observe that a higher number of trees was greatly improving the model until around 250, then gradually performed worse.

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

```
boosted_tree <- collect_metrics(tuning_res) %>%
  arrange(desc(mean))
boosted_tree
```

```
## # 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  0.721     5 0.0105  Preprocessor1_Model02
##  2   673 roc_auc hand_till  0.720     5 0.00939 Preprocessor1_Model04
##  3   894 roc_auc hand_till  0.719     5 0.00971 Preprocessor1_Model05
##  4   452 roc_auc hand_till  0.719     5 0.00940 Preprocessor1_Model03
##  5  1336 roc_auc hand_till  0.719     5 0.00961 Preprocessor1_Model07
##  6  1115 roc_auc hand_till  0.719     5 0.00991 Preprocessor1_Model06
##  7  1557 roc_auc hand_till  0.718     5 0.00942 Preprocessor1_Model08
##  8  2000 roc_auc hand_till  0.718     5 0.00929 Preprocessor1_Model10
##  9  1778 roc_auc hand_till  0.717     5 0.00934 Preprocessor1_Model09
## 10    10 roc_auc hand_till  0.705     5 0.0137  Preprocessor1_Model01
```

The roc_auc of the best performing boosted tree is 0.7207540.

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.

```
models <- data.frame(decision_tree[1,4])
models[2,] <- random_forest[1,6]
models[3,] <- boosted_tree[1,4]

rownames(models) <- c("Decision Tree", "Random Forest", "Boosted Tree")
models %>%
  arrange(desc(mean))
```

```
##                    mean
## Random Forest 0.7426766
## Boosted Tree  0.7207540
## Decision Tree 0.6603222
```

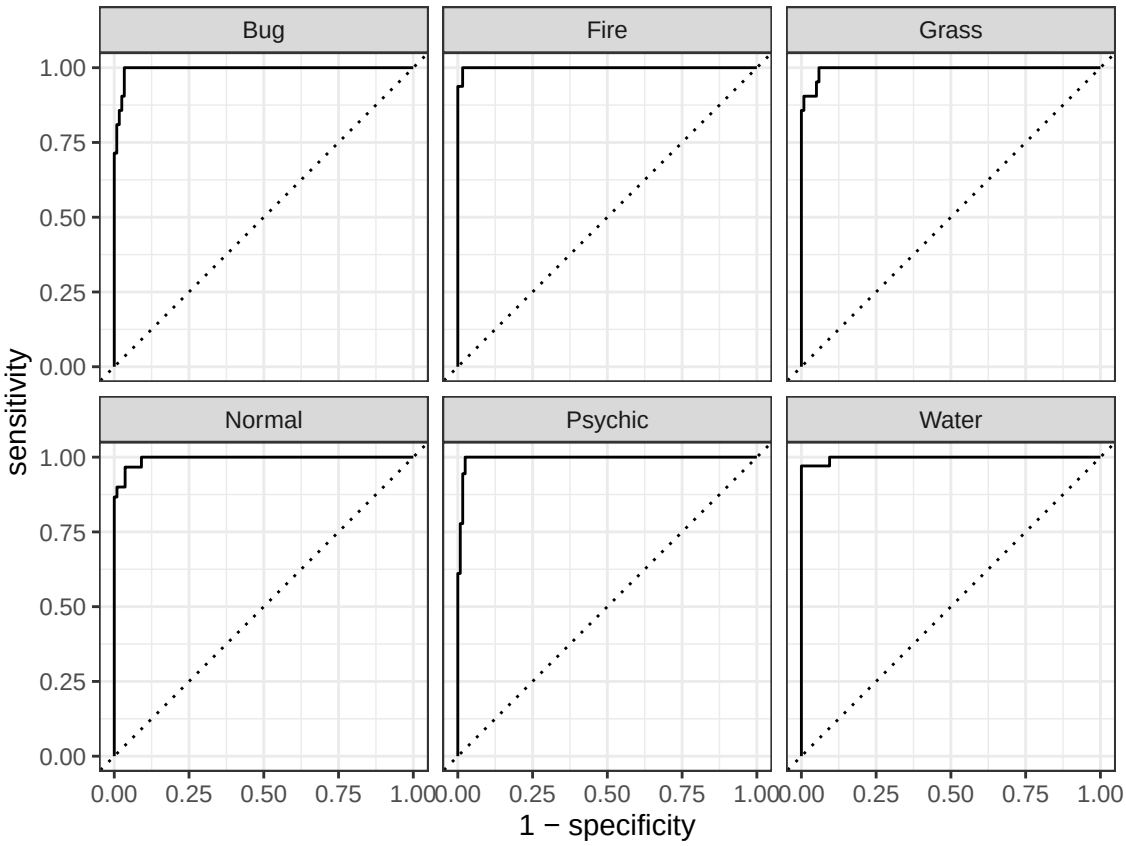
```
best_metrics <- select_best(tune_res)
rf_final <- finalize_workflow(rf_wf, best_metrics)
rf_final_fit_testdata <- fit(rf_final, data = pokemon_test)  # use it on the testing or training data?
```

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.

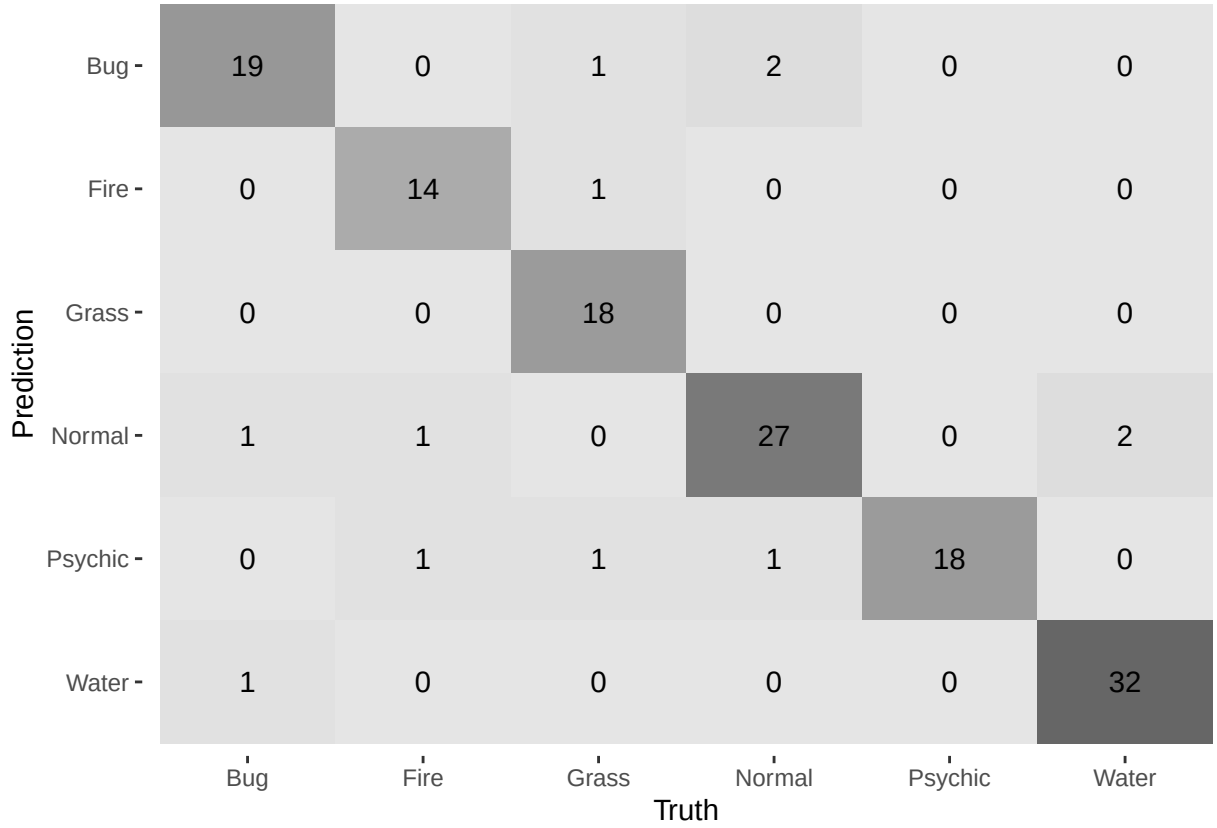
```
augment(rf_final_fit_testdata, new_data = pokemon_test) %>%
  roc_auc(truth = type_1, .pred_Bug:.pred_Water)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.996
```

```
augment(rf_final_fit_testdata, new_data = pokemon_test) %>%
  roc_curve(truth = type_1, .pred_Bug:.pred_Water) %>%
  autoplot()
```




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



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

My model was most accurate at predicting Water, Fire, and Grass. It was the worst at predicting Psychic, Normal, and Bug.