HW6

Ben Hertzberg

2022-11-24

```r
library(tidyverse)
library(tidymodels)
library(corrplot)
library(rpart.plot)
library(vip)
library(janitor)
library(randomForest)
library(xgboost)
```

Exercise 1

```r
pok <- read.csv('./data/Pokemon.csv')
pok <- clean_names(pok)
pok2 <- subset(pok, type_1 == c('Bug','Fire','Grass',"Normal",'Water','Psychic'))
```

```
## Warning in type_1 == c("Bug", "Fire", "Grass", "Normal", "Water", "Psychic"):
## longer object length is not a multiple of shorter object length
```

```r
pok2$type_1 <- as.factor(pok2$type_1)
pok2$legendary <- as.factor(pok2$legendary)
```

```r
set.seed(619)

pok_split <- initial_split(pok2, prop = 0.80, strata = type_1)
pok_train <- training(pok_split)
pok_test <- testing(pok_split)
dim(pok_train)
```

```
## [1] 63 13
```

```r
dim(pok_test)
```

```
## [1] 19 13
```

```r
pok_folds <- vfold_cv(pok_train, v = 5, strata = type_1)
pok_folds
```

```
## #  5-fold cross-validation using stratification
## # A tibble: 5 x 2
##   splits          id
##   <list>          <chr>
## 1 <split [48/15]> Fold1
## 2 <split [49/14]> Fold2
## 3 <split [50/13]> Fold3
## 4 <split [52/11]> Fold4
## 5 <split [53/10]> Fold5
```

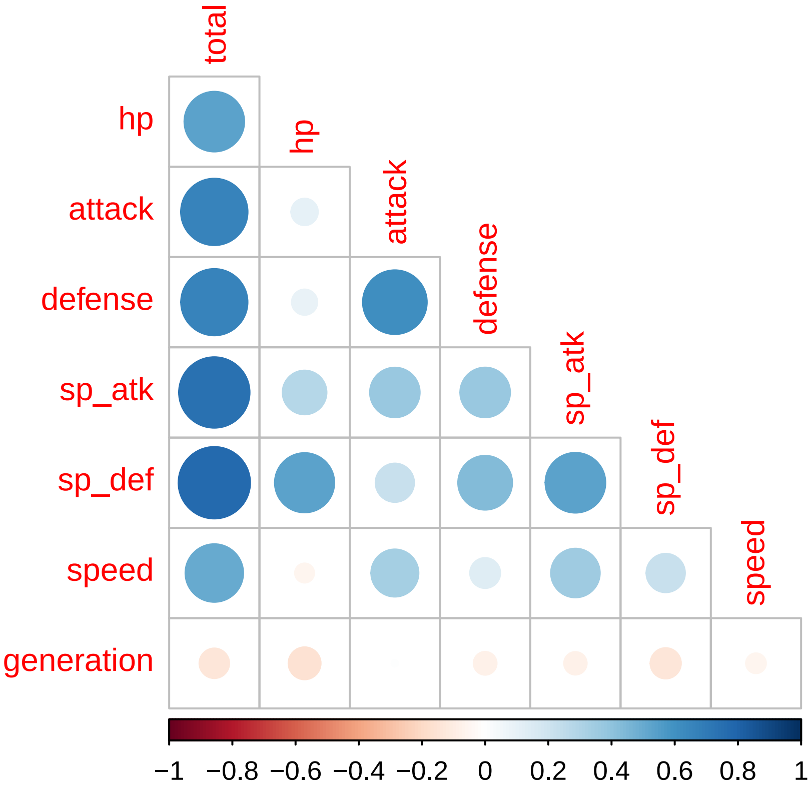
```
pok_rec <- recipe(type_1 ~ legendary + generation + sp_atk + attack + speed + defense + hp + sp_def, da

  step_dummy(legendary, generation) %>%
  step_normalize(all_numeric_predictors())
```

Exercise 2

```
pok_train %>%
  select(is.numeric, -x) %>%
  cor(use = 'complete.obs') %>%
  corrplot(type = 'lower', diag = FALSE)
```

```
## 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.
```



I removed the variable called 'x' from the correlation matrix. This variable contains an ID number for each pokemon, and it is not relevant in predictions.

The variable 'total' has a strong positive correlation with every variable except generation. This makes sense because total is a sum of all these other values. Defense and attack are positively correletated, which migh mean that stronger pokemon perform better in both of these areas. Defense speed (sp_def) and attack speed (sp_atk) are also positively correlated; if a pokemon is fast in one area it makes sense they would be fast in the other.

Exercise 3

```
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(pok_rec)
```

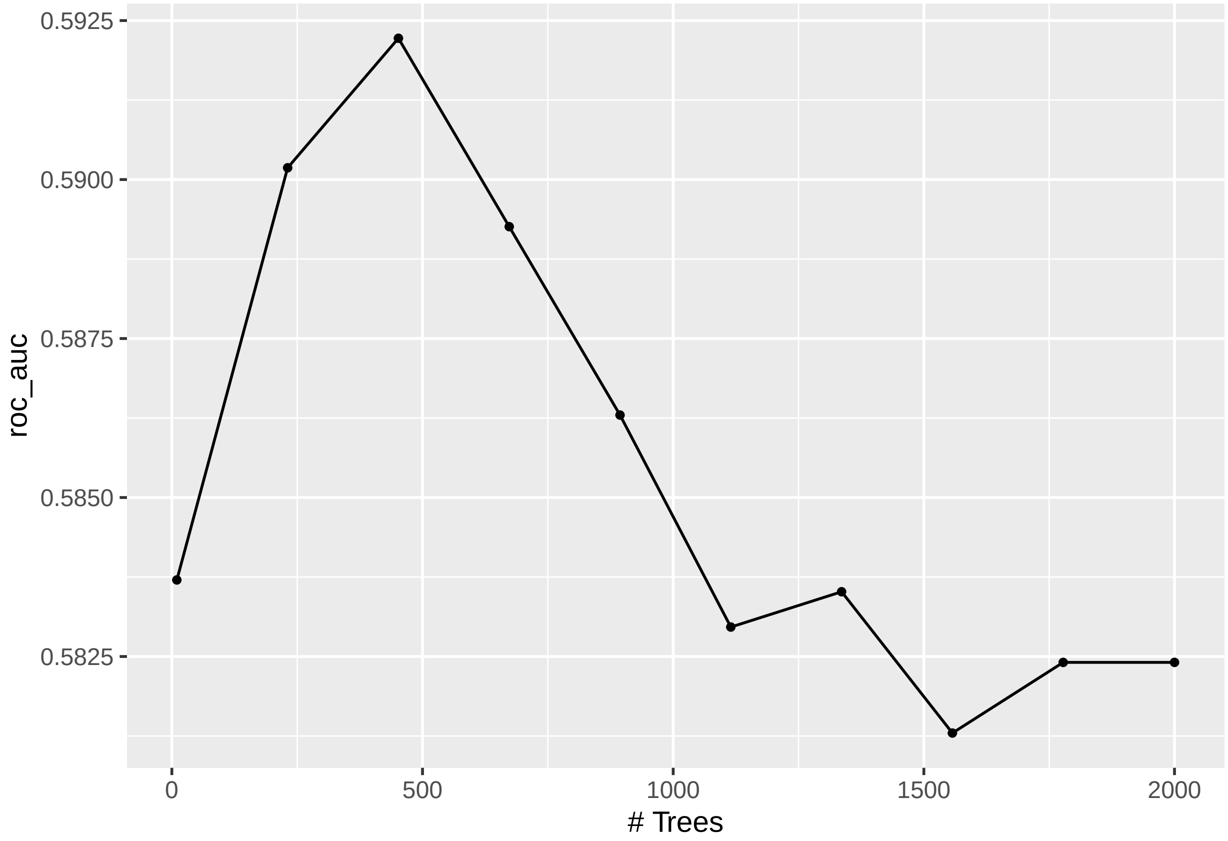
```
param_grid <- grid_regular(cost_complexity(range = c(-3, -1)), levels = 10)

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

write_rds(tune_res, file = 'SavedModels/poketree_tuned_res.rds')
```

```
poketree_tune_res <- read_rds(file = 'SavedModels/poketree_tuned_res.rds')

autoplot(poketree_tune_res)
```



The results for roc_auc are nearly constant until the cost complexity reached about 0.06, where the roc_auc slightly increased. As cost complexity increased beyond that value, roc_auc significantly decreased. In general, it seems like the single decision tree performed better with a smaller cost complexity penalty.

Exercise 4

```
best_mod <- collect_metrics(poketree_tune_res) %>%
  arrange(desc(mean))%>%
  dplyr::slice(1)
```

The `roc_auc` of my best-performing pruned decision tree on the folds is 0.560

Exercise 5

```r
pok_final <- finalize_workflow(class_tree_wf, best_mod)
pok_final_fit <- fit(pok_final, data = pok_train)
pok_final_fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```

The rplot above works perfect in my rmd file, but fails to knit.

Exercise 5

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


rf_wf <- workflow() %>%
  add_model(bagging_spec) %>%
  add_recipe(pok_rec)
```

mtry specifies how many randomly chosen variables are available to the model at each split, which guarantees that not all trees make the same splits. trees is a value that specifies the number of trees the entire ensemble can contain. min_n specifies a minimum number of data points in a node to allow another split to be made.

```r
reg_grid <- grid_regular(mtry(range = c(1,8)), trees(range = c(10, 300)), min_n(range = c(2,12)), levels
```

You can't choose a mtry value less than 1 because then the model would have no variables to choose from to make a split, so no splits would be made. In thise case, you can't have a mtry value more than 8, because there are only 8 predictor variables, so you can't provide the model with more options to make a split on. A mtry value of 8 is bagging.

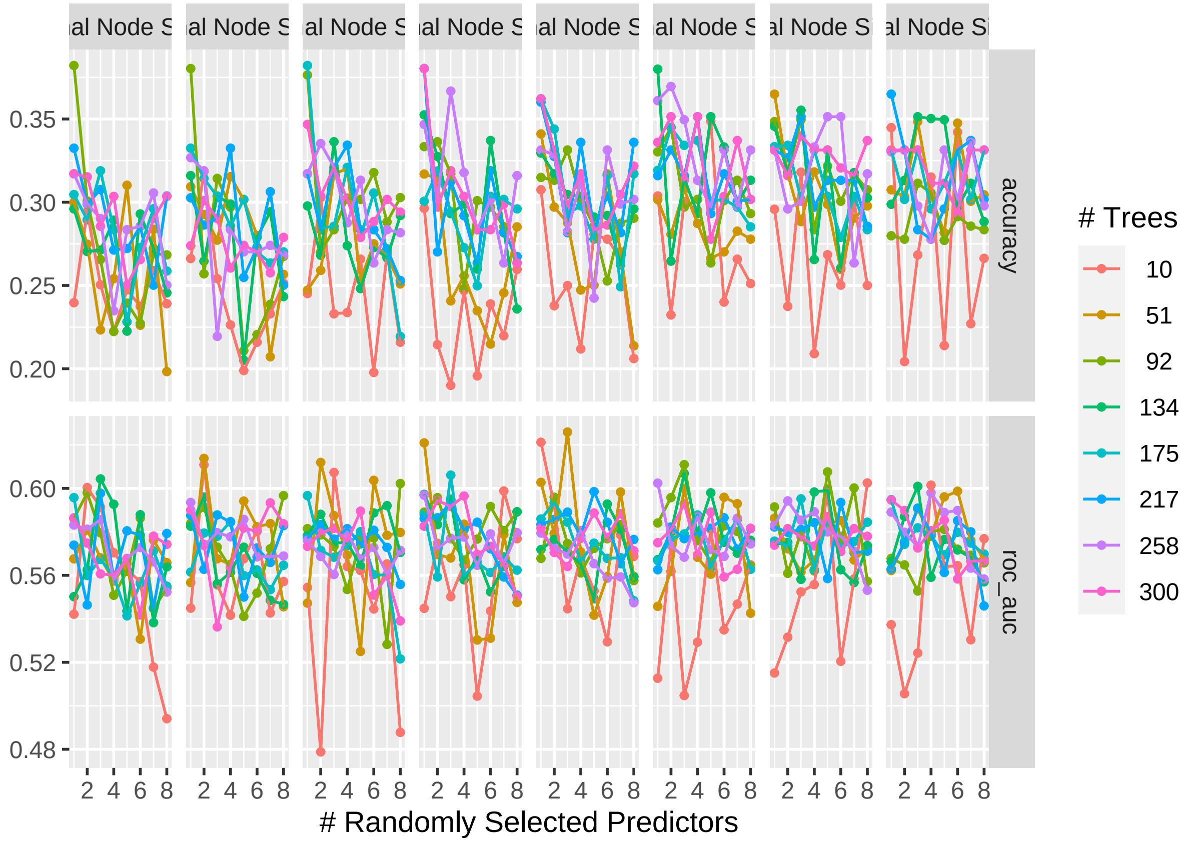
Exercise 6

```r
tune_res2 <- tune_grid(
  rf_wf,
  resamples = pok_folds,
  grid = reg_grid
)

write_rds(tune_res2, file = 'SavedModels/poke_rf_tuned_res.rds')
```

```r
poke_rf_res <- read_rds(file = 'SavedModels/poke_rf_tuned_res.rds')

autoplot(poke_rf_res)
```



Models with fewer trees (less than 100) seemed to have a lot more variability. For every model, the number of randomly selected predictors seemed to have the largest effect on roc_auc.

Models with 2 to 4 randomly selected predictors performed the best. Of these, models with less than 100 trees had the highest peaks in roc_auc, but have far more variability than those with more trees.

Exercise 7

```
best_rf_mod <- collect_metrics(poke_rf_res) %>%
  arrange(desc(mean))%>%
  dplyr::slice(1)
```

The `roc_auc` of the best-performing random forest model is 0.6259

Exercise 8

```
pok_rf_final <- finalize_workflow(rf_wf, best_rf_mod)
rf_fit <- fit(pok_rf_final, data = pok_train)
```

```
## Warning: The following variables are not factor vectors and will be ignored:
## ‘generation‘
```

```
rf_fit
```

```
## == Workflow [trained] ==========================================================
## Preprocessor: Recipe
## Model: rand_forest()
##
## -- Preprocessor ----------------------------------------------------------------
## 2 Recipe Steps
##
## * step_dummy()
## * step_normalize()
##
## -- Model -----------------------------------------------------------------------
## Ranger result
##
## Call:
##  ranger::ranger(x = maybe_data_frame(x), y = y, mtry = min_cols(~3L,      x), num.trees = ~51L, min.
##
## Type:                             Probability estimation
## Number of trees:                  51
## Sample size:                      63
## Number of independent variables:  8
## Mtry:                             3
## Target node size:                 7
## Variable importance mode:         impurity
## Splitrule:                        gini
## OOB prediction error (Brier s.):  0.6527212
```

```
vip(rf_fit)
```

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

I can't get vip() to work. I receive the following message: 'Error: Model-specific variable importance scores are currently not available for this type of model.' I have looked into documentation and online forums with no luck. I saw that in the lab, vip runs when the workflow uses the 'randomForest' engine, and in this problem we are told to use 'ranger'. I wanted to explore this further, but doing so would require retuning the random forest model, and I did not have the patience for that (it takes about an hour on my laptop).

Exercise 9

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

boost_wf <- workflow() %>%
  add_model(boost_spec) %>%
  add_recipe(pok_rec)
```

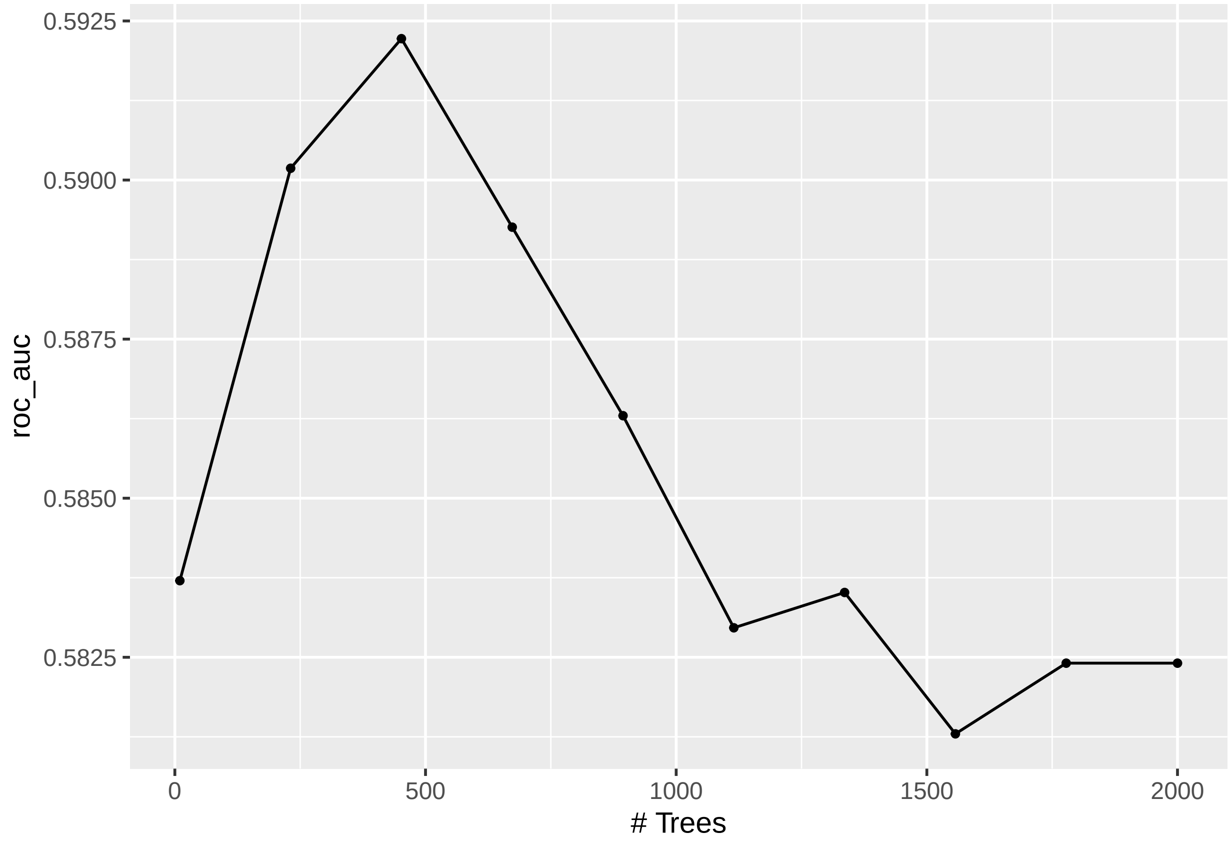
```
boost_grid <- grid_regular(trees(range = c(10, 2000)), levels = 10)

tune_res <- tune_grid(
  boost_wf,
  resamples = pok_folds,
  grid = boost_grid,
  metrics = metric_set(roc_auc)
)

write_rds(tune_res, file = 'SavedModels/poketree_boost_res.rds')
```

```
poke_boost_res <- read_rds(file = 'SavedModels/poketree_boost_res.rds')

autoplot(poke_boost_res)
```



The roc_auc peaks around 500 trees at 0.5925. Increasing or deacreasing the trees leads to a large drop in roc_auc.

```
best_boost_mod <- collect_metrics(poke_boost_res) %>%
  arrange(desc(mean)) %>%
  dplyr::slice(1)
```

The `roc_auc` of the best-performing boosted tree model is 0.5922

Exercise 10

```
best_mod$mean
```

```
## [1] 0.5922222
```

```
best_rf_mod$mean
```

```
## [1] 0.6259259
```

```
best_boost_mod$mean
```

```
## [1] 0.5922222
```

The random forest model performed the best on the folds.

```
final_mod <- best_rf_mod

pok_final <- finalize_workflow(rf_wf, final_mod)
test_fit <- fit(pok_final, data = pok_test)
```

```
## Warning: The following variables are not factor vectors and will be ignored:
## ‘generation‘
```

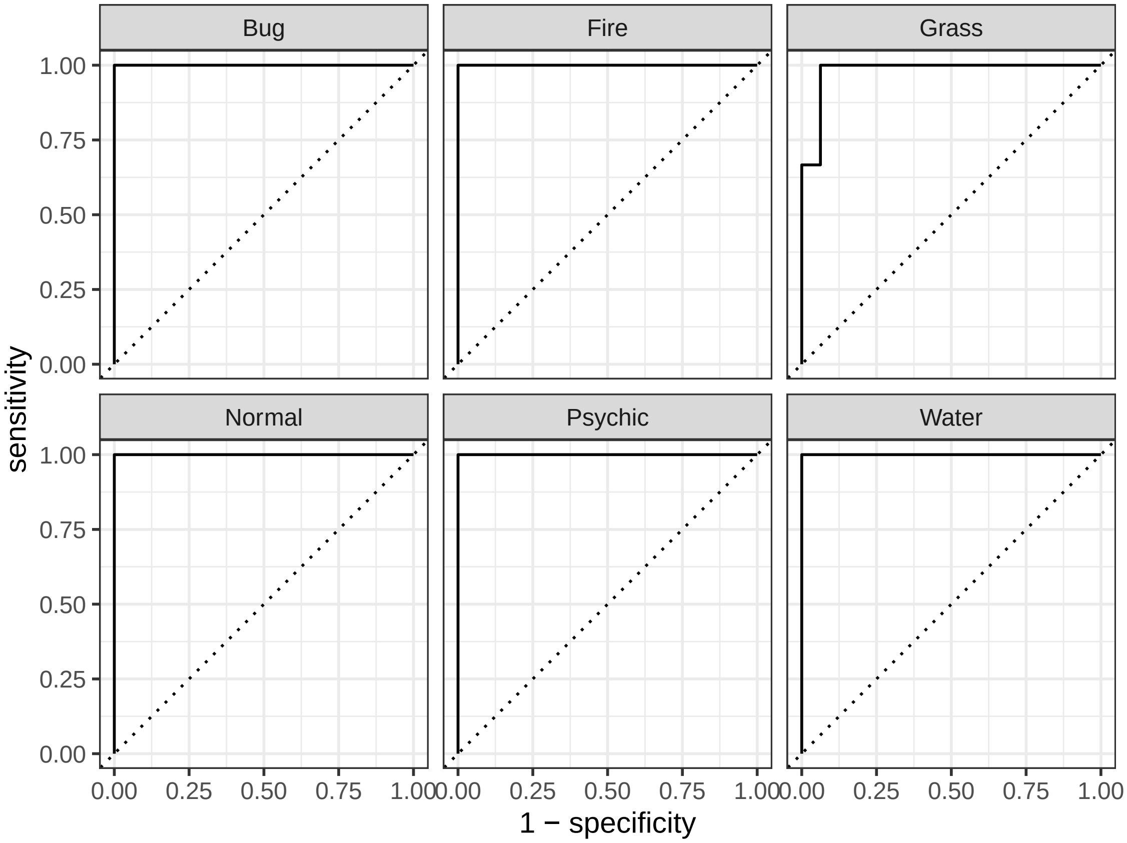
```
test_fit
```

```
## == Workflow [trained] ==========================================================
## Preprocessor: Recipe
## Model: rand_forest()
##
## -- Preprocessor ----------------------------------------------------------------
## 2 Recipe Steps
##
## * step_dummy()
## * step_normalize()
##
## -- Model -----------------------------------------------------------------------
## Ranger result
##
## Call:
##  ranger::ranger(x = maybe_data_frame(x), y = y, mtry = min_cols(~3L,      x), num.trees = ~51L, min.
##
## Type:                             Probability estimation
## Number of trees:                  51
## Sample size:                      19
## Number of independent variables:  8
## Mtry:                             3
## Target node size:                 7
## Variable importance mode:         impurity
## Splitrule:                        gini
## OOB prediction error (Brier s.):  0.7765371
```

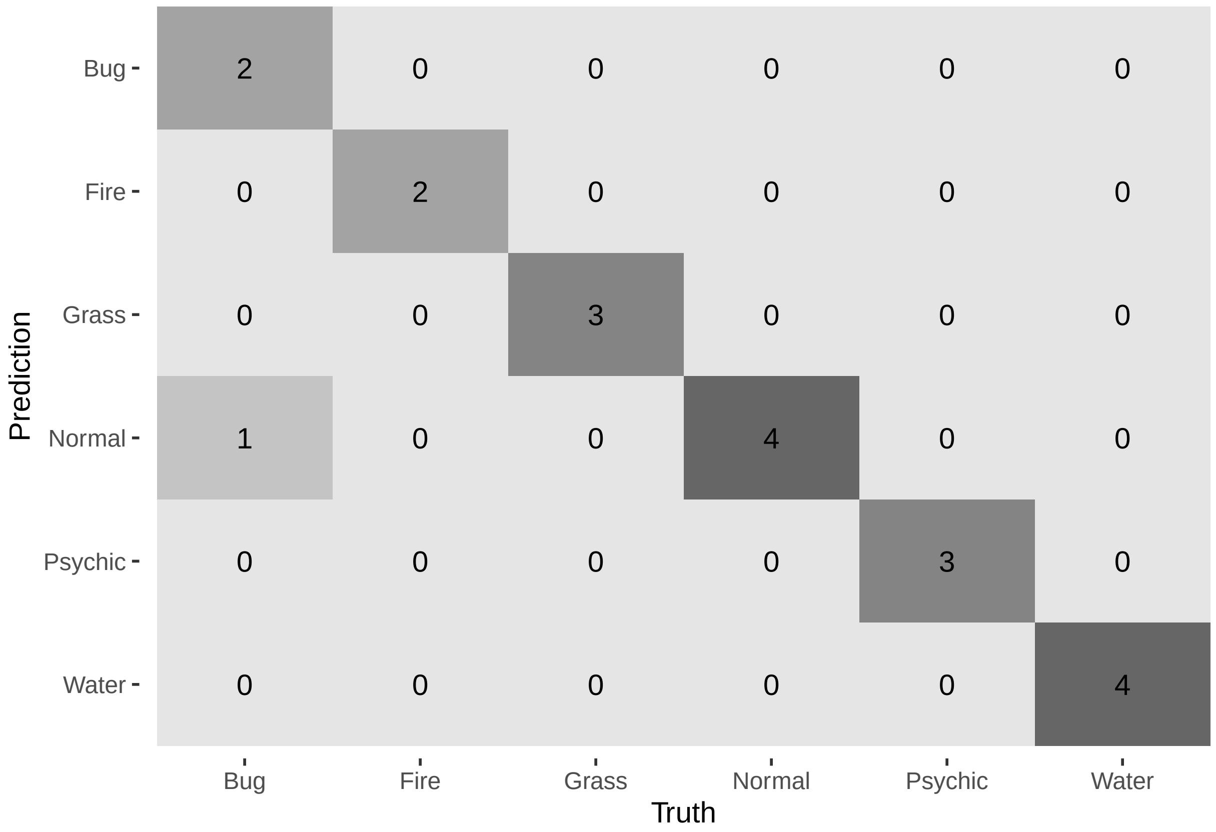
```
augment(test_fit, new_data = pok_test) %>%
  roc_auc(type_1, .pred_Bug:.pred_Water)
```

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

```
augment(test_fit, new_data = pok_test) %>%
  roc_curve(type_1, .pred_Bug:.pred_Water) %>%
  autoplot()
```



```
augment(test_fit, new_data = pok_test) %>%
  conf_mat(truth = type_1, estimate = .pred_class) %>%
  autoplot(type = 'heatmap')
```



	Bug	Fire	Grass	Normal	Psychic	Water
Bug	2	0	0	0	0	0
Fire	0	2	0	0	0	0
Grass	0	0	3	0	0	0
Normal	1	0	0	4	0	0
Psychic	0	0	0	0	3	0
Water	0	0	0	0	0	4

I am getting an roc_auc of 1, which suggests perfect prediction. This might be an error, but I also noticed that in this homework, unlike other weeks, we fit the final model to the test data, and then checked performance using the same test data. It makes sense that the model would be perfect in making predictions for a data set that it was trained on. However, the heatmap actually reveals that the model incorrectly labeled two bug types - one it labeled grass, another normal. This could arise from overlap of predictor variable values in the test set, which could confuse the model.