hw6

2022-11-18

Set up for hw 6

Exercise 1

```
library(janitor)
pokemon <- pokemon %>%
  clean_names()
#filter out rarer classes
pokemon_filter <- pokemon %>%
  filter(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
```

```
#convert type_1 and legendary to factors plus generation
pokemon_factor <- pokemon_filter %>%
  mutate(type_1 = factor(type_1),
         legendary = factor(legendary),
         generation = factor(generation))
#initial_split with percentage; stratify type_1
pokemon_split <- initial_split(pokemon_factor,
                               prop = 0.7,
                               strata = type_1)
pokemon_split
```

```
## <Training/Testing/Total>
## <54/28/82>
```

```
pokemon_train <- training(pokemon_split)
pokemon_test <- testing(pokemon_split)
#v-fold with v=5; strata = type_1
pokemon_folds <- vfold_cv(pokemon_train, v = 5,
                          strata = type_1)
pokemon_folds
```

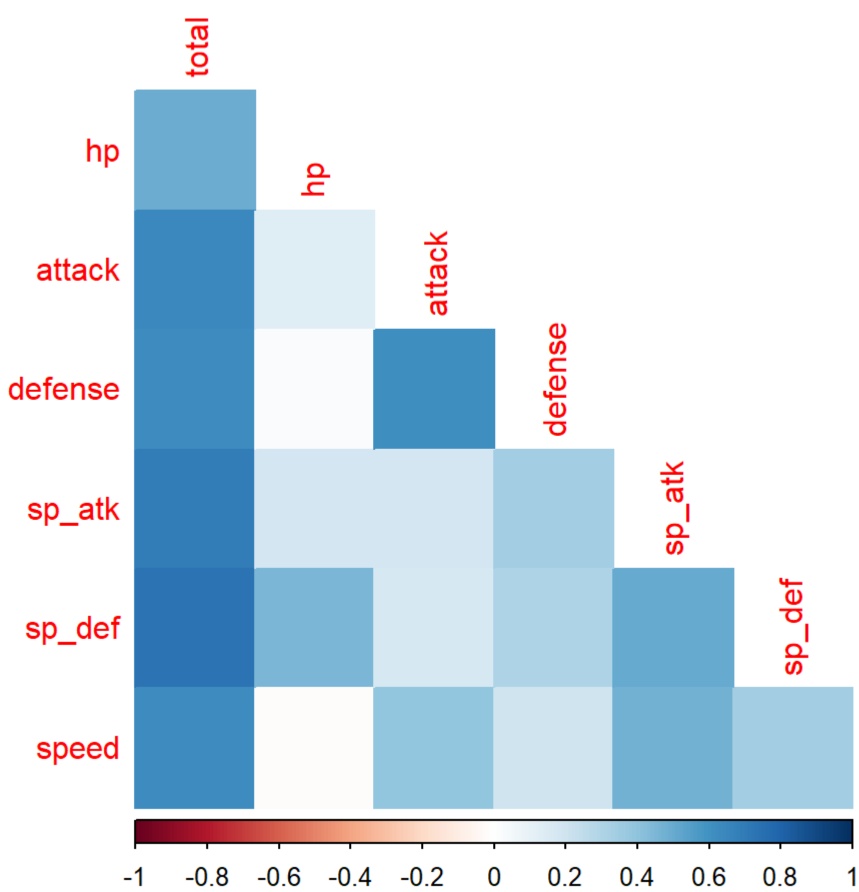
```
## #  5-fold cross-validation using stratification
## # A tibble: 5 × 2
##   splits          id
##   <list>          <chr>
## 1 <split [40/14]> Fold1
## 2 <split [42/12]> Fold2
## 3 <split [43/11]> Fold3
## 4 <split [45/9]>  Fold4
## 5 <split [46/8]>  Fold5
```

```
#recipe
pokemon_recipe <-
  recipe(type_1 ~ legendary + generation + sp_atk + attack + speed + defense +
           hp + sp_def, data = pokemon_train) %>%
  step_dummy(legendary, generation) %>%
  #step_normalize(all_predictors())
  step_center(all_predictors()) %>%
  step_scale(all_predictors())
pokemon_recipe
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          8
##
## Operations:
##
## Dummy variables from legendary, generation
## Centering for all_predictors()
## Scaling for all_predictors()
```

Exercise 2

```
#EXERCISE 2: correlation matrix
#library(corrplot)
pokemon_train %>%
  select_if(is.numeric) %>%
  select(-x) %>%
  cor(use = "complete.obs") %>%
  corrplot(type = "lower", diag = FALSE, method = "color")
```



I decided to exclude x since it isn’t the main focus in analyzing the pokemon. The relationships I notice is that all variables have no relation to highly positively correlated relationship with each other. This makes sense to me as highly leveled pokemon individuals would have a higher attack, defense, and all the other varaibles (sp_atk, sp_def, hp) than lower level pokemon individuals, which will cause them to also have a higher total score.

Exercise 3

```
#EXERCISE 3: decision tree
tree_spec <- decision_tree() %>%
  set_engine("rpart")
class_tree_spec <- tree_spec %>%
  set_mode("classification") %>%
  set_args(cost_complexity = tune())

class_tree_wf <- workflow() %>%
  add_model(class_tree_spec) %>%
  add_recipe(pokemon_recipe) #recipe?

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

Exercise 3 con.

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

write_rds(tune_res, file = "decision-tree-res.rds")

decision_tree <- read_rds(file = "decision-tree-res.rds")
autoplot(decision_tree)
```

I see that the cost-complexity parameter always maintained below a 0.610 roc_auc, but once the cost-complexity parameter reached around 0.035 (between 0.01 and 0.065), it dropped down to below 0.59 roc_auc and rose slightly when the cost-complexity parameter was 0.1 to a value of almost 0.5975 roc_auc. Therefore a single decision tree performed better with a smaller complexity penalty as having a value too big may overprune the tree.

```
#EXERCISE 4: roc_auc
library(yardstick)
decision_tree <- read_rds(file = "decision-tree-res.rds")

decision_roc <- decision_tree %>%
  collect_metrics() %>%
  arrange(desc(mean)) %>%
  slice(1)
decision_roc
```

```
## # A tibble: 1 × 7
##   cost_complexity .metric .estimator  mean     n std_err .config
##             <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1           0.001 roc_auc hand_till  0.609     5  0.0503 Preprocessor1_Model01
```

The roc_auc of my best-performing pruned decision tree on the folds is 0.001 and estimates of the roc_auc curve are under 0.61.

Exercise 5

```
#EXERCISE 5 prt 1: rpart.plot
collect_metrics(decision_tree)
```

```
## # A tibble: 10 × 7
##    cost_complexity .metric .estimator  mean     n std_err .config
##              <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1         0.001   roc_auc hand_till  0.609     5  0.0503 Preprocessor1_Model01
##  2         0.00167 roc_auc hand_till  0.609     5  0.0503 Preprocessor1_Model02
##  3         0.00278 roc_auc hand_till  0.609     5  0.0503 Preprocessor1_Model03
##  4         0.00464 roc_auc hand_till  0.609     5  0.0503 Preprocessor1_Model04
##  5         0.00774 roc_auc hand_till  0.609     5  0.0503 Preprocessor1_Model05
##  6         0.0129  roc_auc hand_till  0.609     5  0.0503 Preprocessor1_Model06
##  7         0.0215  roc_auc hand_till  0.609     5  0.0503 Preprocessor1_Model07
##  8         0.0359  roc_auc hand_till  0.589     5  0.0368 Preprocessor1_Model08
##  9         0.0599  roc_auc hand_till  0.589     5  0.0368 Preprocessor1_Model09
## 10         0.1     roc_auc hand_till  0.597     5  0.0288 Preprocessor1_Model10
```

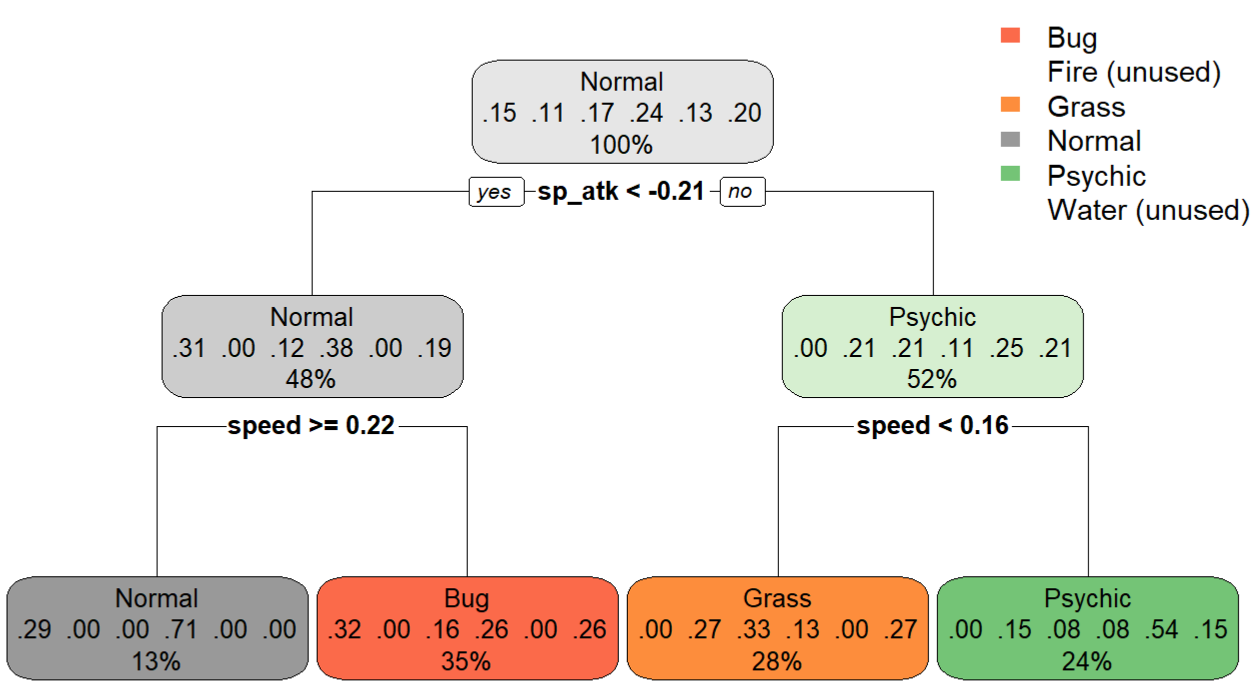
```
best_penalty <- select_best(decision_tree, metric = "roc_auc")

tree_final <- finalize_workflow(class_tree_wf, best_penalty)

tree_final_fit <- fit(tree_final, data = pokemon_train)

tree_final_fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```

```
## Warning: Cannot retrieve the data used to build the model (so cannot determine roundint and i
s.binary for the variables).
## To silence this warning:
##     Call rpart.plot with roundint=FALSE,
##     or rebuild the rpart model with model=TRUE.
```



Exercise 5 prt 6 (“Exercise 6”)

```
#EXERCISE 5: random forest model and wrkflow
bagging_spec <- rand_forest() %>%
  set_engine("ranger", importance = 'impurity') %>%
  set_mode("classification") %>%
  set_args(mtry = tune(),
           trees = tune(),
           min_n = tune())

class_tree_wf2 <- workflow() %>%
  add_model(bagging_spec) %>%
  add_recipe(pokemon_recipe)
param_grid2 <- grid_regular(mtry(range = c(1, 8)), trees(range = c(200, 1000)),
                            min_n(range = c(5, 20)), levels = 8)
```

The mtry is the number of randomly sampled variables for each split. Trees are the number of trees per forest. Min_n is the minimum number of predictors at each split. Mtry should not be smaller than 1 or larger than 8 as you would be using more predictors than provided. Mtry = 8 means that each decision tree has all the available predictors for each split.

Exercise 6 or "Exercise 7"

```
#EXERCISE 6: roc_auc as metric -- takes a few minutes to run
#tune_res2 <- tune_grid(
  #class_tree_wf2,
  #resamples = pokemon_folds,
  #grid = param_grid2,
  #metrics = metric_set(roc_auc)
#)

write_rds(tune_res2, file = "rand-forest-res.rds")
rand_tree <- read_rds(file = "rand-forest-res.rds")
autoplot(rand_tree)
```

I observe that the roc_auc would increase as the minimal node size increases. The values of hyperparameters seem to yield the best performance were 15, 17, and 20.

Exercise 7 ("Exercise 8")

```
#roc_auc of random forest model on folds
rand_tree <- read_rds(file = "rand-forest-res.rds")
rand_roc <- rand_tree %>%
  collect_metrics() %>%
  arrange(desc(mean)) %>%
  slice(1)
```

The roc_auc of my best-performing random forest model on the folds is 0.702.

Exercise 8 ("Exercise 9")

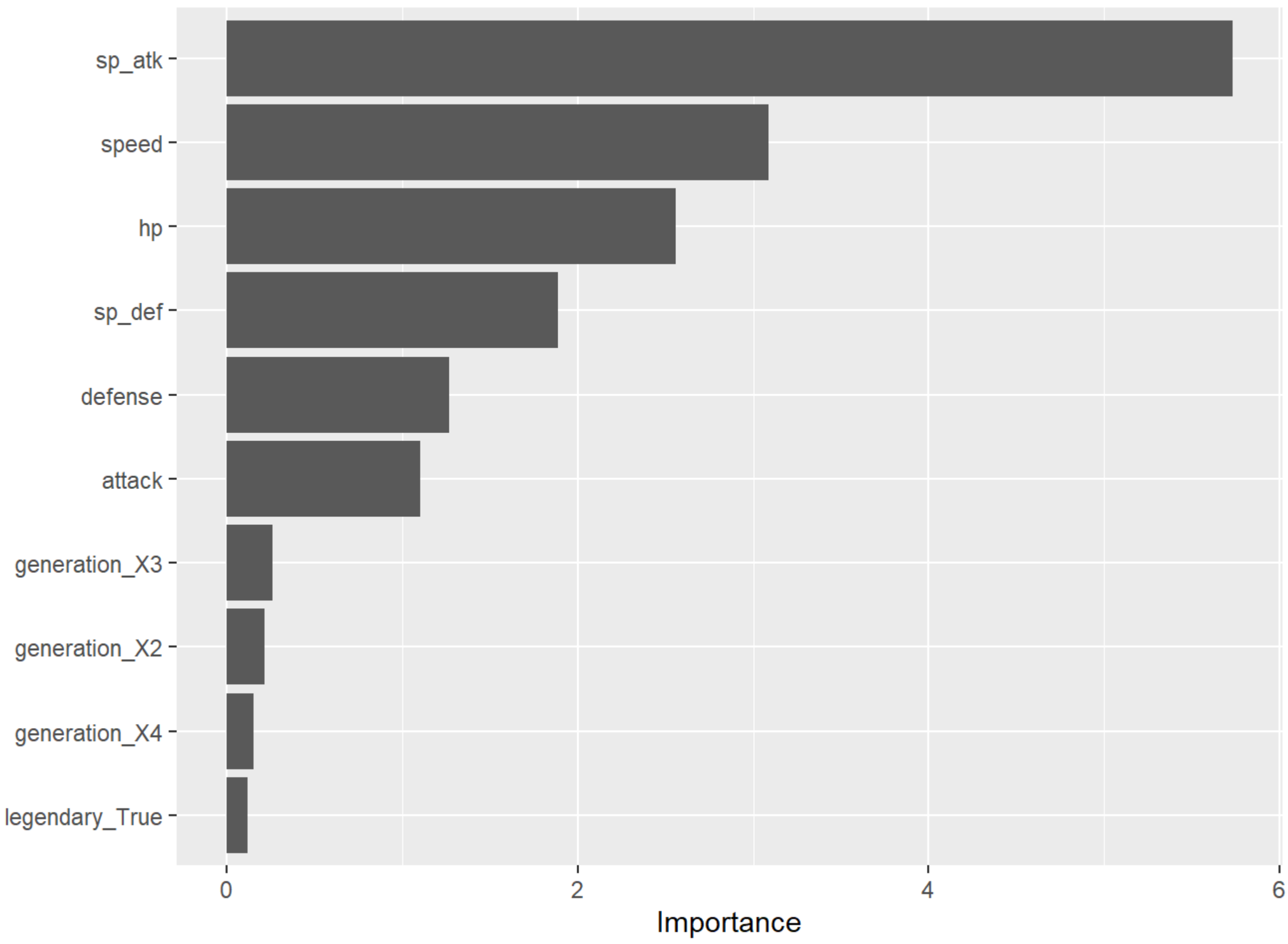
```
#EXERCISE 8: vip()
collect_metrics(rand_tree)
```

```
## # A tibble: 512 × 9
##     mtry trees min_n .metric .estimator  mean     n std_err .config
##    <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1     1   200     5 roc_auc hand_till  0.580     5  0.0377 Preprocessor1_Model…
##  2     2   200     5 roc_auc hand_till  0.586     5  0.0446 Preprocessor1_Model…
##  3     3   200     5 roc_auc hand_till  0.615     5  0.0497 Preprocessor1_Model…
##  4     4   200     5 roc_auc hand_till  0.606     5  0.0426 Preprocessor1_Model…
##  5     5   200     5 roc_auc hand_till  0.649     5  0.0431 Preprocessor1_Model…
##  6     6   200     5 roc_auc hand_till  0.634     5  0.0450 Preprocessor1_Model…
##  7     7   200     5 roc_auc hand_till  0.638     5  0.0509 Preprocessor1_Model…
##  8     8   200     5 roc_auc hand_till  0.651     5  0.0510 Preprocessor1_Model…
##  9     1   314     5 roc_auc hand_till  0.586     5  0.0384 Preprocessor1_Model…
## 10     2   314     5 roc_auc hand_till  0.590     5  0.0384 Preprocessor1_Model…
## # … with 502 more rows
```

```
best_penalty2 <- select_best(rand_tree, metric = "roc_auc")

tree_final2 <- finalize_workflow(class_tree_wf2, best_penalty2)

tree_final_fit2 <- fit(tree_final2, data = pokemon_train)
vip(extract_fit_engine(tree_final_fit2))
```



The variables that were the most useful were sp_atk, speed, and hp. The variables that were the least useful were generation_X3, generation_X2, legendary_True, and generation_X6. These are the results I expected.

Exercise 9 (“Exercise 10”)

```
#EXERCISE 9: boosted tree model
boost_spec <- boost_tree() %>%
  set_engine("xgboost") %>%
  set_mode("classification") %>%
  set_args(trees = tune())

class_tree_wf3 <- workflow() %>%
  add_model(boost_spec) %>%
  add_recipe(pokemon_recipe)

param_grid3 <- grid_regular(trees(range = c(10, 2000)),levels = 10)

#tune_res3 <- tune_grid(
  #class_tree_wf3,
  #resamples = pokemon_folds,
  #grid = param_grid3,
  #metrics = metric_set(roc_auc)
#)

write_rds(tune_res3, file = "boosted-forest-res.rds")
boost_tree <- read_rds(file = "boosted-forest-res.rds")
autoplot(boost_tree)
```

I observe that the highest roc_auc was when the number of trees was around 894 trees with almost 0.666 roc_auc.

Exercise 9 (“Exercise 10”) con.

```
#boosted tree model and workflow with roc_auc
boost_tree <- read_rds(file = "boosted-forest-res.rds")
boost_roc <- boost_tree %>%
  collect_metrics() %>%
  arrange(desc(mean)) %>%
  slice(1)
```

The roc_auc of my best-performing boosted tree model on the folds was 0.666.

Exercise 10 (“Exercise 11”)

```
set.seed(1234)
result <- bind_rows(decision_roc, rand_roc, boost_roc) %>%
  tibble() %>%
  mutate(model = c('pruned tree model', 'random forest model',
                   'boost tree model'),
         .before = .metric)
result
```

```
## # A tibble: 3 × 11
##   cost_com…¹ model .metric .esti…²  mean     n std_err .config  mtry trees min_n
##        <dbl> <chr> <chr>   <chr>   <dbl> <int>   <dbl> <chr>   <int> <int> <int>
## 1      0.001 prun… roc_auc hand_t… 0.609     5  0.0503 Prepro…    NA    NA    NA
## 2     NA     rand… roc_auc hand_t… 0.702     5  0.0542 Prepro…     7   428    20
## 3     NA     boos… roc_auc hand_t… 0.666     5  0.0506 Prepro…    NA   894    NA
## # … with abbreviated variable names ¹cost_complexity, ².estimator
```

```
#random forest did best:

collect_metrics(rand_tree)
```

```
## # A tibble: 512 × 9
##     mtry trees min_n .metric .estimator  mean     n std_err .config
##    <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1     1   200     5 roc_auc hand_till  0.580     5  0.0377 Preprocessor1_Model…
##  2     2   200     5 roc_auc hand_till  0.586     5  0.0446 Preprocessor1_Model…
##  3     3   200     5 roc_auc hand_till  0.615     5  0.0497 Preprocessor1_Model…
##  4     4   200     5 roc_auc hand_till  0.606     5  0.0426 Preprocessor1_Model…
##  5     5   200     5 roc_auc hand_till  0.649     5  0.0431 Preprocessor1_Model…
##  6     6   200     5 roc_auc hand_till  0.634     5  0.0450 Preprocessor1_Model…
##  7     7   200     5 roc_auc hand_till  0.638     5  0.0509 Preprocessor1_Model…
##  8     8   200     5 roc_auc hand_till  0.651     5  0.0510 Preprocessor1_Model…
##  9     1   314     5 roc_auc hand_till  0.586     5  0.0384 Preprocessor1_Model…
## 10     2   314     5 roc_auc hand_till  0.590     5  0.0384 Preprocessor1_Model…
## # … with 502 more rows
```

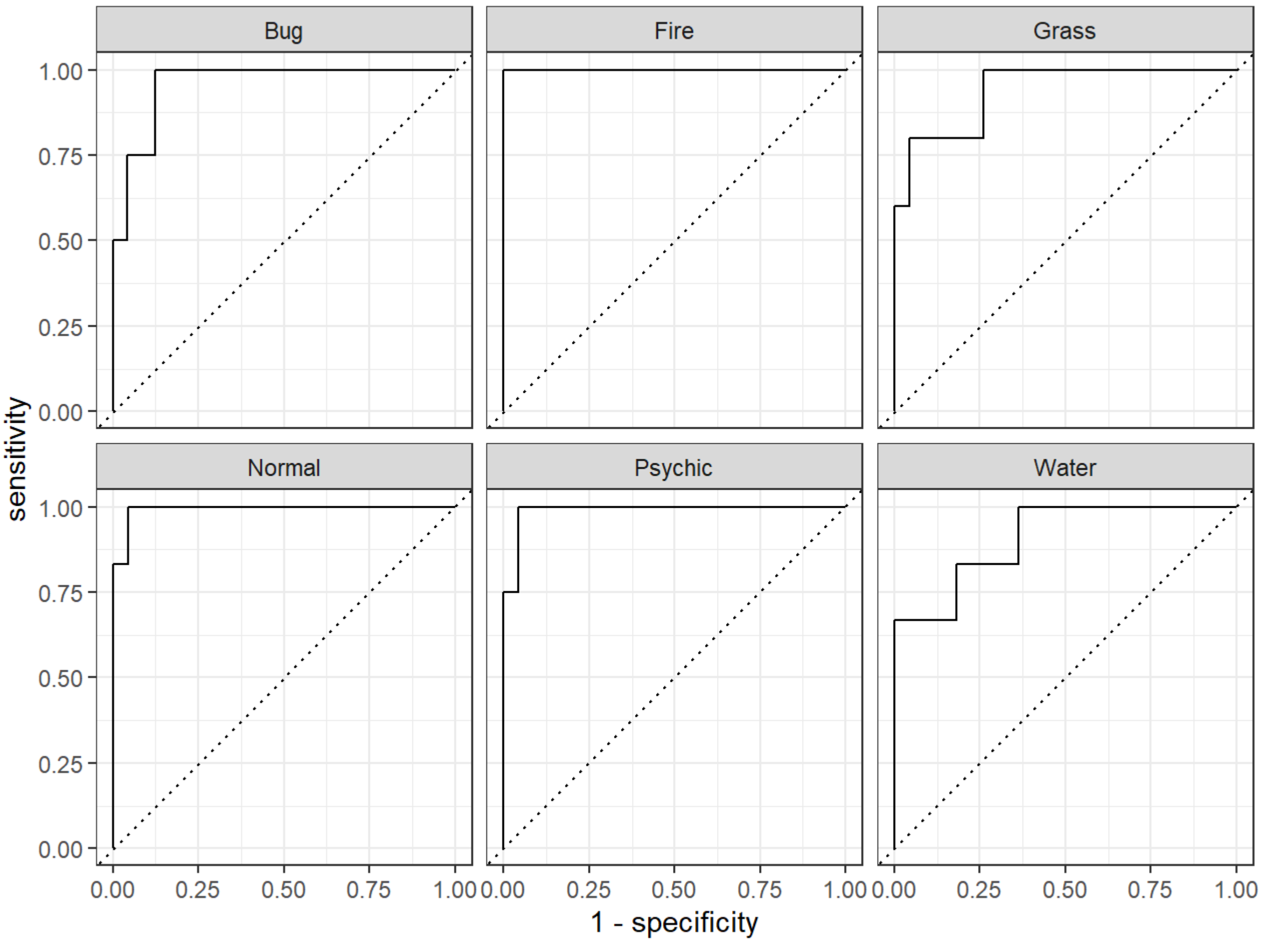
```
final_penalty <- select_best(rand_tree)

final_tree <- finalize_workflow(class_tree_wf2, final_penalty)

final_fit <- fit(final_tree, data = pokemon_test)
final_auc_roc <- augment(final_fit, new_data = pokemon_test) %>%
  select(type_1, starts_with(".pred")) %>%
  roc_auc(type_1, .pred_Bug:.pred_Water)
final_auc_roc #.964
```

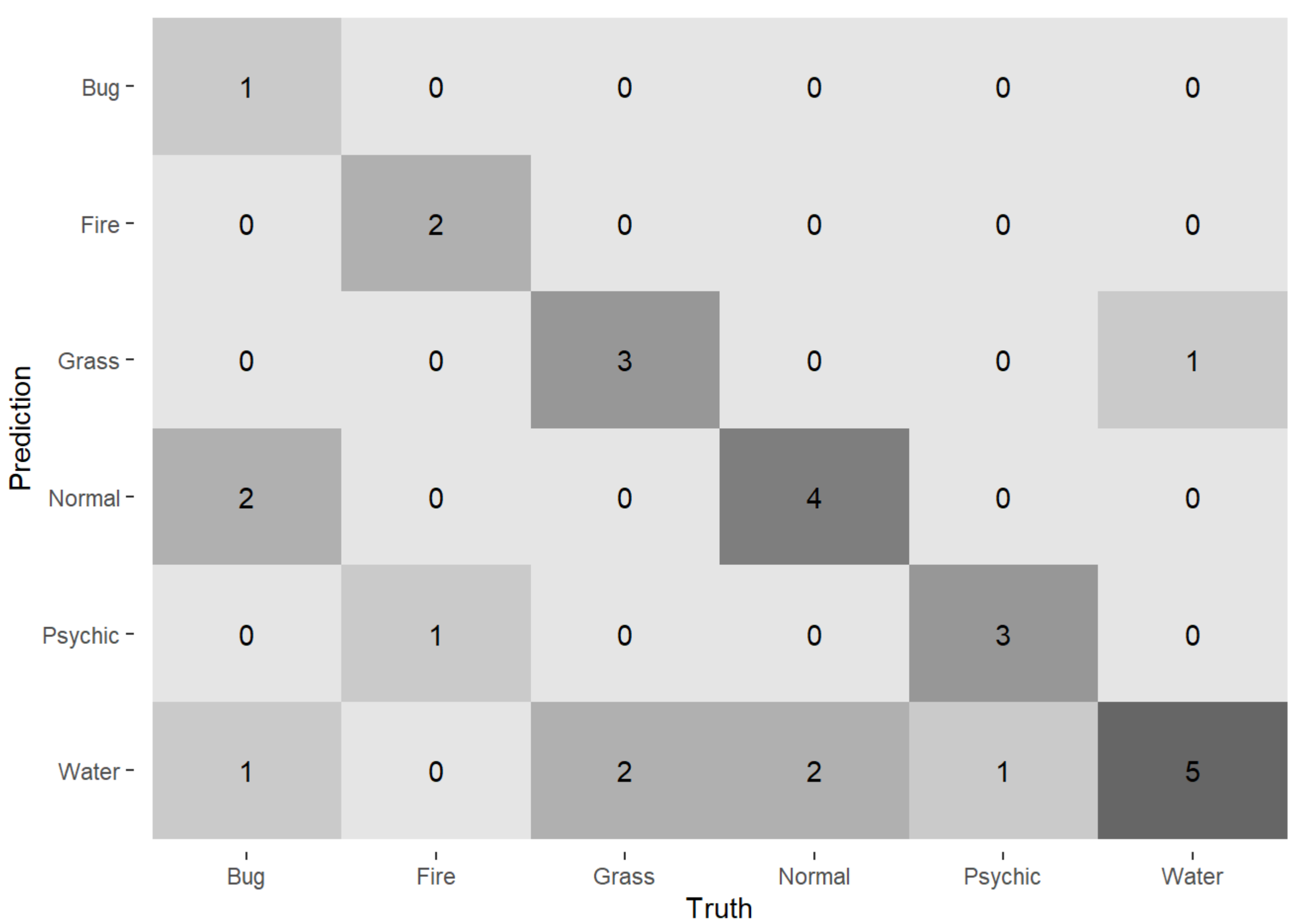
```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.964
```

```
roc_auc2 <- augment(final_fit, new_data = pokemon_test) %>%
  select(type_1, starts_with(".pred"))
final_fit2 <- roc_auc2 %>%
  roc_curve(type_1, .pred_Bug:.pred_Water)
ggplot2::autoplot(final_fit2)
```




```
fit_3 <- augment(final_fit, new_data = pokemon_test) %>%
  conf_mat(truth = type_1, estimate = .pred_class)

autoplot(fit_3, type = "heatmap")
```



Prediction \ Truth	Bug	Fire	Grass	Normal	Psychic	Water
Bug	1	0	0	0	0	0
Fire	0	2	0	0	0	0
Grass	0	0	3	0	0	1
Normal	2	0	0	4	0	0
Psychic	0	1	0	0	3	0
Water	1	0	2	2	1	5

My model was good at predicting Bug, Psychic, Fire, Normal, and Grass. However, it was the worst at predicting Water.