Pstat 131 Homework 6

Yu Tian

Spring 2022-05-24

Tree-Based Models

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.

Answer Q1

```
# Read the Pokemon data set into R
Pokemon <- read_csv(file = "Pokemon.csv")
head(Pokemon)
```

```
## # A tibble: 6 x 13
##     `#` Name    `Type 1` `Type 2` Total    HP Attack Defense `Sp. Atk` `Sp. Def`
##   <dbl> <chr>   <chr>    <chr>    <dbl> <dbl>  <dbl>   <dbl>     <dbl>     <dbl>
## 1     1 Bulbas~ Grass    Poison     318    45     49      49        65        65
## 2     2 Ivysaur Grass    Poison     405    60     62      63        80        80
## 3     3 Venusa~ Grass    Poison     525    80     82      83       100       100
## 4     3 Venusa~ Grass    Poison     625    80    100     123       122       120
## 5     4 Charma~ Fire     <NA>       309    39     52      43        60        50
## 6     5 Charme~ Fire     <NA>       405    58     64      58        80        65
## # ... with 3 more variables: Speed <dbl>, Generation <dbl>, Legendary <lgl>
```

```
# Set things up
pokemon <- Pokemon %>%
  # Use clean_names() for lowercase of variable names
  clean_names() %>%
  # Filter out the rarer Pokémon types
  filter(type_1 == "Bug" |
           type_1 == "Fire" |
           type_1 == "Grass" |
           type_1 == "Normal" |
           type_1 == "Water" |
           type_1 == "Psychic") %>%
  # Convert type_1 and legendary to factors
  mutate(type_1 = as.factor(type_1)) %>%
  mutate(legendary = as.factor(legendary)) %>%
```

```
  mutate(generation = as.factor(generation))

head(pokemon)
```

```
## # A tibble: 6 x 13
##   number name       type_1 type_2 total    hp attack defense sp_atk sp_def speed
##    <dbl> <chr>      <fct>  <chr>  <dbl> <dbl>  <dbl>   <dbl>  <dbl>  <dbl> <dbl>
## 1      1 Bulbasaur  Grass  Poison   318    45     49      49     65     65    45
## 2      2 Ivysaur    Grass  Poison   405    60     62      63     80     80    60
## 3      3 Venusaur   Grass  Poison   525    80     82      83    100    100    80
## 4      3 VenusaurM~ Grass  Poison   625    80    100     123    122    120    80
## 5      4 Charmander Fire   <NA>     309    39     52      43     60     50    65
## 6      5 Charmeleon Fire   <NA>     405    58     64      58     80     65    80
## # ... with 2 more variables: generation <fct>, legendary <fct>
```

```
# Do an initial split of the data
set.seed(0623)
pokemon_split <- initial_split(pokemon, prop = 0.7, strata = type_1)
pokemon_train <- training(pokemon_split)
pokemon_test <- testing(pokemon_split)

dim(pokemon_train)
```

```
## [1] 318  13
```

```
dim(pokemon_test)
```

```
## [1] 140  13
```

```
# Fold the training set using v-fold cross-validation, with v = 5. Stratify on the outcome variable
pokemon_fold <- vfold_cv(pokemon_train, v = 5, strata = type_1)
pokemon_fold
```

```
## #  5-fold cross-validation using stratification
## # A tibble: 5 x 2
##   splits           id
##   <list>           <chr>
## 1 <split [252/66]> Fold1
## 2 <split [253/65]> Fold2
## 3 <split [253/65]> Fold3
## 4 <split [256/62]> Fold4
## 5 <split [258/60]> Fold5
```

```
# Set up a recipe to predict type_1
pokemon_recipe <- recipe(type_1 ~ legendary + generation + sp_atk + attack +
                           speed + defense + hp + sp_def, data = pokemon_train) %>%
  # Dummy-code legendary and generation
  step_dummy(legendary) %>%
  step_dummy(generation) %>%
  # Center and scale 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
## Dummy variables from generation
## Centering for all_predictors()
## Scaling for 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).

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

Answer Q2

```r
# Create a correlation matrix of the training set
pokemon_train%>%
  select(where(is.numeric)) %>%
  select(-number) %>%
  cor() %>%
  corrplot(method = 'number')
```

I decided to only use the numeric variable in the plot since their value can be calculated to plot. Besides, I decided to remove the numeric variable, which is "number", since it just represent the ID number of each pokemon and has no relationship with other variables.

From the matrix above, we can find there is no negative correlation among all these variables. Total (sum of all stats) has a strong positive correlation with alomost all other variables. Besides, (1) Defense and sp_def (special defense against special attacks), (2) attack and sp_atk (special attack), (3) sp_def and sp_attack, (4) attack and defense also have strong positive correlation.

I think these relationships make sense to me. Since the total represents how strong a pokemon is, it is reasonable that with higher total, the value of all other variables (skills) will be higher. Both defense and sp_def are the skills against attack, so they have positive relationship. Both attack and sp_atk are the skills against attack, so they have positive relationship.

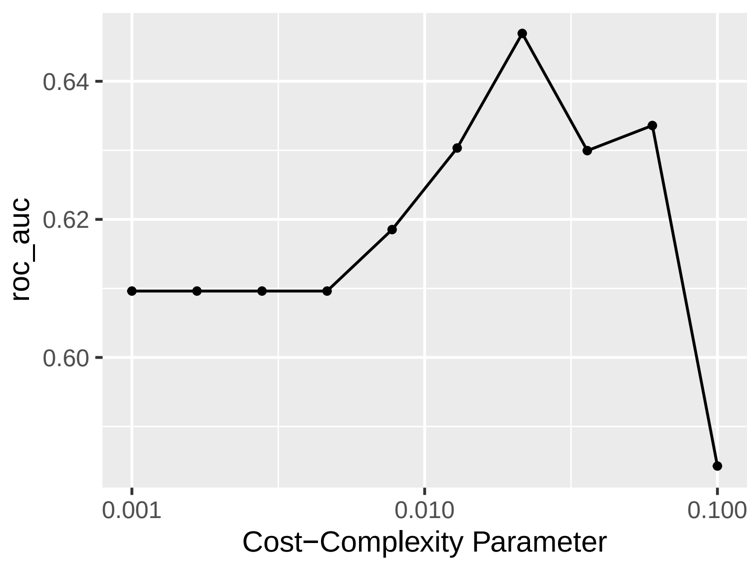
Exercise 3

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?

Answer Q3

```r
# set up a decision tree model and workflow
pokemon_tree_model <- decision_tree() %>%
  set_engine("rpart") %>%
  set_mode("classification") %>%
  set_args(cost_complexity = tune())

pokemon_wf <- workflow() %>%
  add_model(pokemon_tree_model) %>%
  add_recipe(pokemon_recipe)

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

pokemon_tune_res <- tune_grid(pokemon_wf,
                              resamples = pokemon_fold,
                              grid = pokemon_grid,
                              metrics = metric_set(roc_auc))

autoplot(pokemon_tune_res)
```



From the graph above, we can observe that in the beginning, with the cost-complexity parameter increase, the roc_auc also increase. However, after the value of roc_auc get the peak value, the roc_auc significantly decrease as the cost-complexity parameter increase. Thus, we can conclude that a single decision tree perform better with 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().

Answer Q4

```
best_auc <- collect_metrics(pokemon_tune_res) %>%
  arrange(desc(mean))
best_auc
```

```
## # A tibble: 10 x 7
##    cost_complexity .metric .estimator  mean     n std_err .config
##              <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1         0.0215  roc_auc hand_till  0.647     5 0.0124  Preprocessor1_Model07
##  2         0.0599  roc_auc hand_till  0.634     5 0.00693 Preprocessor1_Model09
##  3         0.0129  roc_auc hand_till  0.630     5 0.0199  Preprocessor1_Model06
##  4         0.0359  roc_auc hand_till  0.630     5 0.00924 Preprocessor1_Model08
##  5         0.00774 roc_auc hand_till  0.619     5 0.0213  Preprocessor1_Model05
##  6         0.001   roc_auc hand_till  0.610     5 0.0286  Preprocessor1_Model01
##  7         0.00167 roc_auc hand_till  0.610     5 0.0286  Preprocessor1_Model02
##  8         0.00278 roc_auc hand_till  0.610     5 0.0286  Preprocessor1_Model03
##  9         0.00464 roc_auc hand_till  0.610     5 0.0286  Preprocessor1_Model04
## 10         0.1     roc_auc hand_till  0.584     5 0.0346  Preprocessor1_Model10
```

The roc_auc of my best-performing pruned decision tree on the folds is 0.6469358

Exercise 5

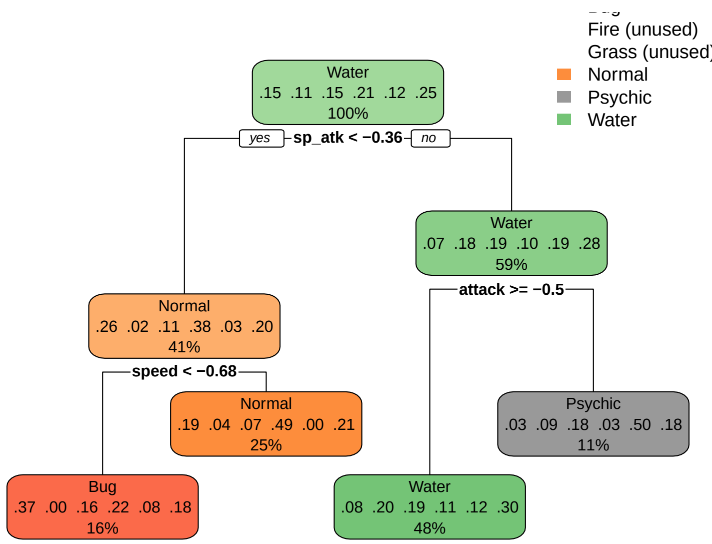
Using rpart.plot, fit and visualize your best-performing pruned decision tree with the training set.

Answer Q5

```
# fit and visualize the best-performing pruned decision tree with the training set
best_complexity <- select_best(pokemon_tune_res)
best_complexity
```

```
## # A tibble: 1 x 2
##   cost_complexity .config
##             <dbl> <chr>
## 1          0.0215 Preprocessor1_Model07
```

```
pokemon_tree_final <- finalize_workflow(pokemon_wf, best_complexity)

pokemon_tree_final_fit <- fit(pokemon_tree_final, data = pokemon_train)

pokemon_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?

Answer Q5

```r
# set up a random forest model
pokemon_rf_model <- rand_forest() %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("classification") %>%
  set_args(mtry = tune(), trees = tune(), min_n = tune())

# set up a random forest workflow
pokemon_rf_wf <- workflow() %>%
  add_model(pokemon_rf_model) %>%
  add_recipe(pokemon_recipe)

# create a regular grid
pokemon_rf_grid <- grid_regular(mtry(range = c(2,7)),
                                trees(range = c(10, 800)),
                                min_n(range = c(2, 20)),
                                levels = 8)
```

mtry means the number of predictors that will be randomly sampled for creating each split of the tree models.

trees means the number of individual trees in the tree models

min_n means the minimum number of data points in a nod required for the node to continue split

mtry should not be smaller than 1 or larger than 8 because we only have 8 predictors in our specified model and we can not have 0 predictor in the model.

mtry = 8 represent the random forest model use all predictors and all the trees will be identical.

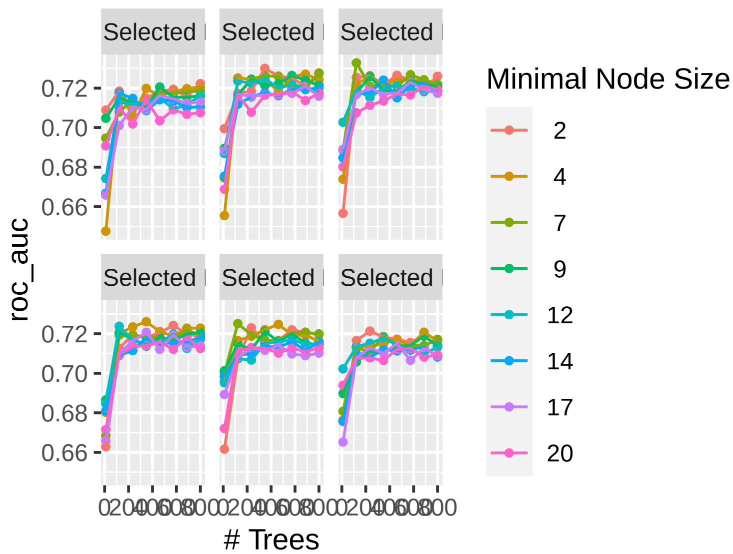
Exercise 6

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?

Answer Q6

```
pokemon_turn_rf <- tune_grid(pokemon_rf_wf,
                             resamples = pokemon_fold,
                             grid = pokemon_rf_grid,
                             metrics = metric_set(roc_auc))
```

```
autoplot(pokemon_turn_rf)
```



From the graph above, we can observe that with the number of trees increase, the value of roc_auc also increase significantly. However, when the number of trees is larger than about 100, the value of roc_auc will slow increasing and trend to fluctuate and be stable. Besides, the difference among the minimum nod sizes is not very large and obvious, but we can still find that less minimum nod sizes with larger roc_auc, which means perform slightly better. Also, the difference among the the number of randomly selected predictors is not very large and obvious, but we can still find that the number 3,4,5 of randomly selected predictors with larger roc_auc, which means perform slightly better.

Exercise 7

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

Answer Q7

```
rf_best_auc <- collect_metrics(pokemon_turn_rf) %>%
  arrange(desc(mean))
rf_best_auc
```

```
## # A tibble: 384 x 9
##     mtry trees min_n .metric .estimator  mean     n std_err .config
##    <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1     4   122     7 roc_auc hand_till  0.733     5 0.0156  Preprocessor1_Model~
##  2     3   348     2 roc_auc hand_till  0.730     5 0.0132  Preprocessor1_Model~
```

```
## 3      3   800      2 roc_auc hand_till  0.728     5 0.0132  Preprocessor1_Model~
## 4      3   800      7 roc_auc hand_till  0.727     5 0.00997 Preprocessor1_Model~
## 5      3   687      4 roc_auc hand_till  0.727     5 0.0115  Preprocessor1_Model~
## 6      4   574      7 roc_auc hand_till  0.727     5 0.0104  Preprocessor1_Model~
## 7      4   461      2 roc_auc hand_till  0.726     5 0.0130  Preprocessor1_Model~
## 8      3   574      9 roc_auc hand_till  0.726     5 0.00882 Preprocessor1_Model~
## 9      3   348      7 roc_auc hand_till  0.726     5 0.0101  Preprocessor1_Model~
## 10     4   235      2 roc_auc hand_till  0.726     5 0.0128  Preprocessor1_Model~
## # ... with 374 more rows
```

The roc_auc of my best-performing random forest model on the folds is 0.733 with 4 mtry, 122 trees, and 7 min_n.

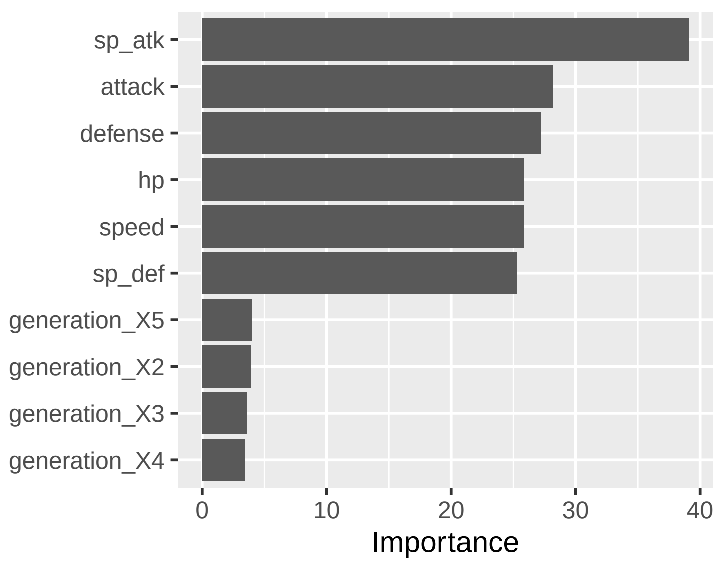
Exercise 8

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?

Answer Q8

```r
best_rf <- select_best(pokemon_turn_rf, metric='roc_auc')
rf_final <- finalize_workflow(pokemon_rf_wf, best_rf)
rf_final_fit <- fit(rf_final, data=pokemon_train)

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



sp_atk is most useful. Speed, attack, hp, defense, sp_def were also more useful. generation and legendary were least useful.

These results is consistent with my expected, since the generation and legendary has less relevance with the type_1 to determine the strongness of pokemon. Other variables is more relevant since they are all the stats of pokemon.

Exercise 9

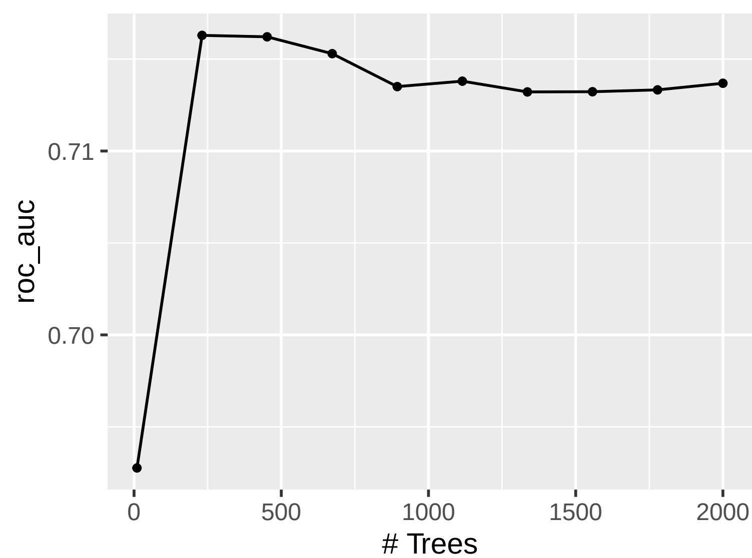
Finally, set up a boosted tree model and workflow. Use the xgboost engine. Tune trees. Create a regular grid with 10 levels; let trees range from 10 to 2000. Specify roc_auc and again print an autoplot() of the results.

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().

Answer Q9

```
# set up a boosted tree model
boost_spec <- boost_tree() %>%
  set_engine("xgboost") %>%
  set_args(trees = tune()) %>%
  set_mode("classification")

# set up a boosted tree workflow
boost_wf <- workflow() %>%
  add_model(boost_spec) %>%
  add_recipe(pokemon_recipe)

# Create a regular grid
boost_grid <- grid_regular(trees(range = c(10, 2000)), levels = 10)

tune_boost <- tune_grid(
  boost_wf,
  resamples = pokemon_fold,
  grid = boost_grid,
  metrics = metric_set(roc_auc))

autoplot(tune_boost)
```



From the graph above, we can observe that with the number of trees increasing, the value of roc_auc significantly increase in the begging. However, after the value of roc_auc achieving the peak, (the number of trees is around 250), the value of roc_vac starts to slowly decrease with the increased number of trees.

```
best_boost <- collect_metrics(tune_boost) %>%
  arrange(desc(mean))
best_boost
```

```
## # 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  0.716     5  0.0260 Preprocessor1_Model02
##  2   452 roc_auc hand_till  0.716     5  0.0267 Preprocessor1_Model03
##  3   673 roc_auc hand_till  0.715     5  0.0270 Preprocessor1_Model04
##  4  1115 roc_auc hand_till  0.714     5  0.0275 Preprocessor1_Model06
##  5  2000 roc_auc hand_till  0.714     5  0.0279 Preprocessor1_Model10
##  6   894 roc_auc hand_till  0.714     5  0.0280 Preprocessor1_Model05
##  7  1778 roc_auc hand_till  0.713     5  0.0277 Preprocessor1_Model09
##  8  1557 roc_auc hand_till  0.713     5  0.0278 Preprocessor1_Model08
##  9  1336 roc_auc hand_till  0.713     5  0.0278 Preprocessor1_Model07
## 10    10 roc_auc hand_till  0.693     5  0.0179 Preprocessor1_Model01
```

From the graph above, we can find that the roc_auc of my best-performing boosted tree model on the folds is 0.7162906.

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?

Answer Q10

```
# Display a table
best_tree <- collect_metrics(pokemon_tune_res) %>%
  arrange(desc(mean)) %>%
  filter(row_number() == 1)
best_tree
```

```
## # A tibble: 1 x 7
##   cost_complexity .metric .estimator  mean     n std_err .config
##             <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1          0.0215 roc_auc hand_till  0.647     5  0.0124 Preprocessor1_Model07
```

```
best_tree_auc <- best_tree['mean']
best_tree_auc
```

```
## # A tibble: 1 x 1
##    mean
##   <dbl>
## 1 0.647
```

```
best_rf <- collect_metrics(pokemon_turn_rf) %>%
  arrange(desc(mean)) %>%
  filter(row_number() == 1)
best_rf
```

```
## # A tibble: 1 x 9
##    mtry trees min_n .metric .estimator  mean     n std_err .config
##   <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1     4   122     7 roc_auc hand_till  0.733     5  0.0156 Preprocessor1_Model1~
```

```
best_rf_auc <- best_rf['mean']
best_rf_auc
```

```
## # A tibble: 1 x 1
##    mean
##   <dbl>
## 1 0.733
```

```
best_boost <- collect_metrics(tune_boost) %>%
  arrange(desc(mean)) %>%
  filter(row_number() == 1)
best_boost
```

```
## # 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  0.716     5  0.0260 Preprocessor1_Model02
```

```
best_boost_auc <- best_boost['mean']
best_boost_auc
```

```
## # A tibble: 1 x 1
##    mean
##   <dbl>
## 1 0.716
```

```
table <- rbind(best_tree_auc, best_rf_auc, best_boost_auc)%>%
  mutate(model = c("pruned tree", "random forest", "boosted tree")) %>%
  arrange(desc(mean))
table
```

```
## # A tibble: 3 x 2
##    mean model
##   <dbl> <chr>
## 1 0.733 random forest
## 2 0.716 boosted tree
## 3 0.647 pruned tree
```

From the table above, random forest performed best on the folds.

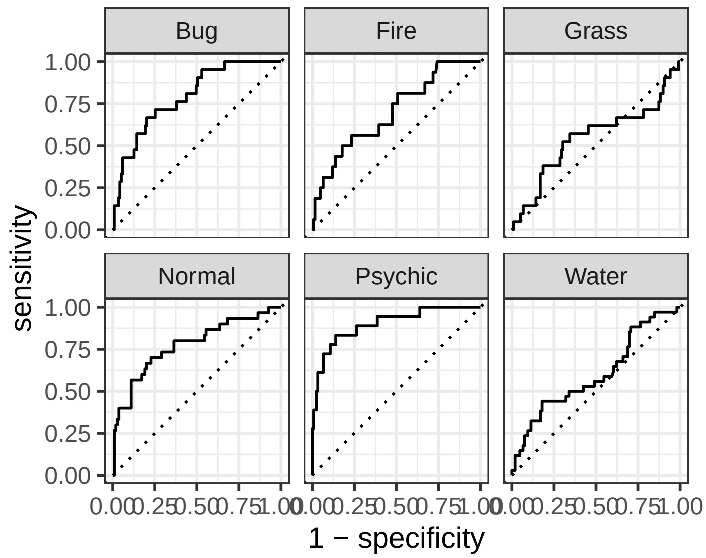
```
# fit to the test
best <- select_best(pokemon_turn_rf, metric = 'roc_auc')

pokemon_final_test <- finalize_workflow(pokemon_rf_wf, best)

pokemon_final_fit <- fit(pokemon_final_test, data = pokemon_train)
```

```
# Print the AUC value of the best-performing model on the testing set.
roc_auc(augment(pokemon_final_fit, new_data = pokemon_test), type_1, .pred_Bug, .pred_Fire,
                .pred_Grass, .pred_Normal, .pred_Psychic, .pred_Water)
```

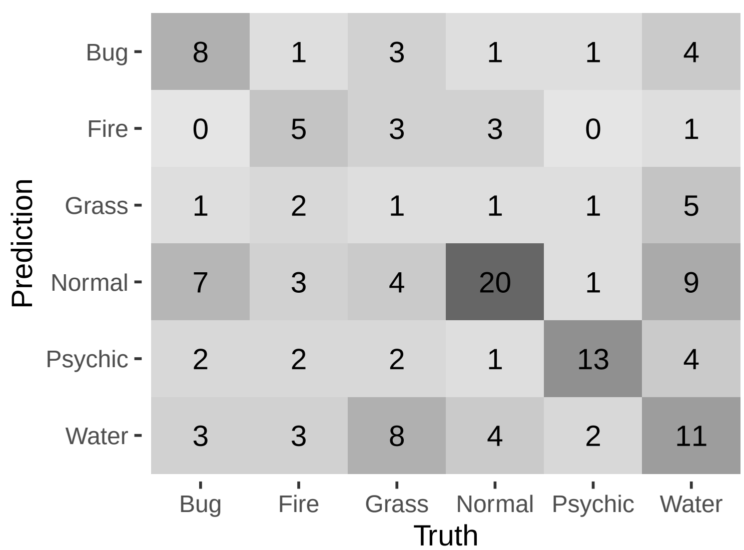
```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
```

```
## 1 roc_auc hand_till      0.718
```

```r
# Print the ROC curves.
augment(pokemon_final_fit, new_data = pokemon_test) %>%
  roc_curve(type_1, .pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal,
            .pred_Psychic,.pred_Water) %>%
  autoplot()
```




```r
# create and visualize a confusion matrix heat map.
augment(pokemon_final_fit, new_data = pokemon_test) %>%
  conf_mat(truth = type_1, estimate = .pred_class)  %>%
  autoplot(type = "heatmap")
```



Normal class was my model most accurate at predicting, and water class was the worst.