Homework 6

Code

PSTAT 131/231

Hide

```r
library(tidymodels)
library(tidyverse)
library(ISLR) # For the Smarket data set
library(ISLR2) # For the Bikeshare data set
library(discrim)
library(poissonreg)
library(corrr)
library(klaR) # for naive bayes
library(forcats)
library(corrplot)
library(pROC)
library(recipes)
library(rsample)
library(parsnip)
library(workflows)
library(janitor)
library(glmnet)
library(rpart.plot)
library(vip)
library(janitor)
library(randomForest)
library(xgboost)
tidymodels_prefer()
```

Tree-Based Models

For this assignment, we will continue working with the file `"pokemon.csv"`, found in `/data`. The file is from Kaggle: https://www.kaggle.com/abcsds/pokemon (https://www.kaggle.com/abcsds/pokemon).

The Pokémon (https://www.pokemon.com/us/) franchise encompasses video games, TV shows, movies, books, and a card game. This data set was drawn from the video game series and contains statistics about 721 Pokémon, or “pocket monsters.” In Pokémon games, the user plays as a trainer who collects, trades, and battles Pokémon to (a) collect all the Pokémon and (b) become the champion Pokémon trainer.

Each Pokémon has a primary type (https://bulbapedia.bulbagarden.net/wiki/Type) (some even have secondary types). Based on their type, a Pokémon is strong against some types, and vulnerable to others. (Think rock, paper, scissors.) A Fire-type Pokémon, for example, is vulnerable to Water-type Pokémon, but strong against Grass-type.

Fig 1. Houndoom, a Dark/Fire-type canine Pokémon from Generation II.

The goal of this assignment is to build a statistical learning model that can predict the **primary type** of a Pokémon based on its generation, legendary status, and six battle statistics.

Note: Fitting ensemble tree-based models can take a little while to run. Consider running your models outside of the .Rmd, storing the results, and loading them in your .Rmd to minimize time to knit.

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.

Hide

```r
# read dataset
pokemon_raw <- read.csv("Pokemon.csv")

# clean_name()
pokemon1 <- clean_names(pokemon_raw)

# filter out rarer
pokemon <- pokemon1[which(pokemon1$type_1 == "Bug"| pokemon1$type_1 =="Fire"|
                     pokemon1$type_1 =="Grass"| pokemon1$type_1 =="Normal"|
                     pokemon1$type_1 =="Water"| pokemon1$type_1 =="Psychic"), ]

# convert factors
pokemon <- pokemon %>%
              mutate(type_1 = factor(type_1),
                  legendary =factor(legendary))

# initial split
set.seed(2022)
pokemon_split <- pokemon %>%
  initial_split(strata = type_1, prop = 0.7)
pokemon_train <- training(pokemon_split)
pokemon_test <- testing(pokemon_split)

# set up cross-validation
pokemon_fold <- vfold_cv(pokemon_train, v = 5, strata = type_1)

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

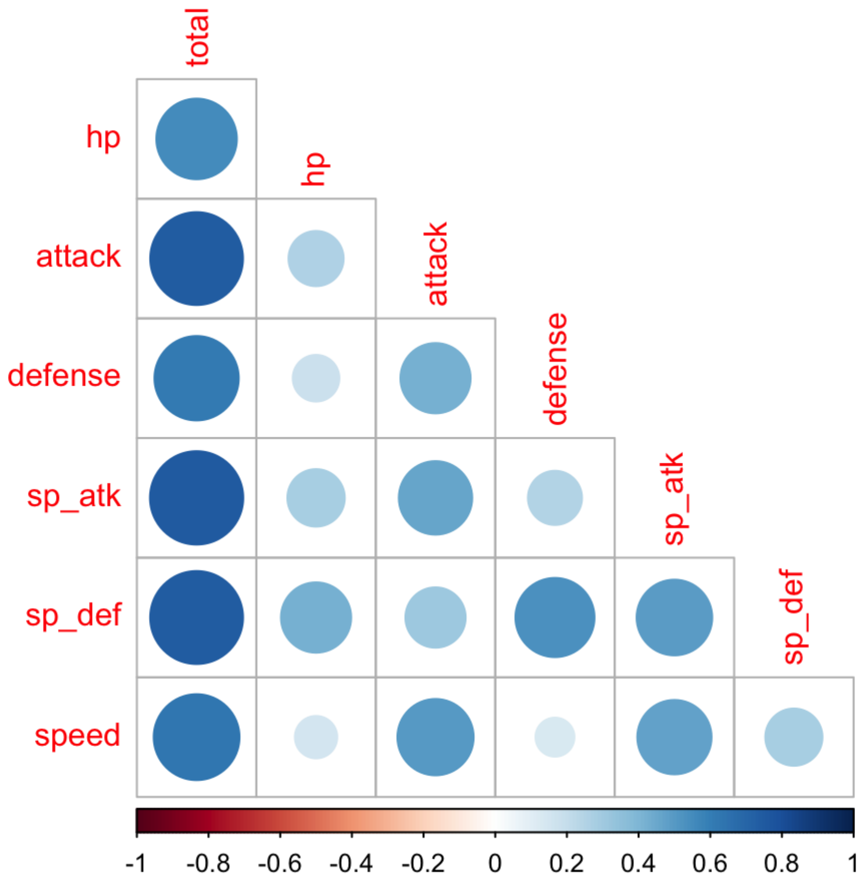
Exercise 2

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?

Hide

```
pokemon_train %>%
  select(where(is.numeric), -x,-generation) %>%
  cor(use = "complete.obs") %>%
  corrplot(type = "lower", diag = FALSE)
```



I remove variable x and generation since x is index and generation is not continuous.
Almost all the variables have positive relationship to other, especially, all variables are high positive relate to total which make sense to me that the higher variable a pokemon has the more stronger pokemon is.

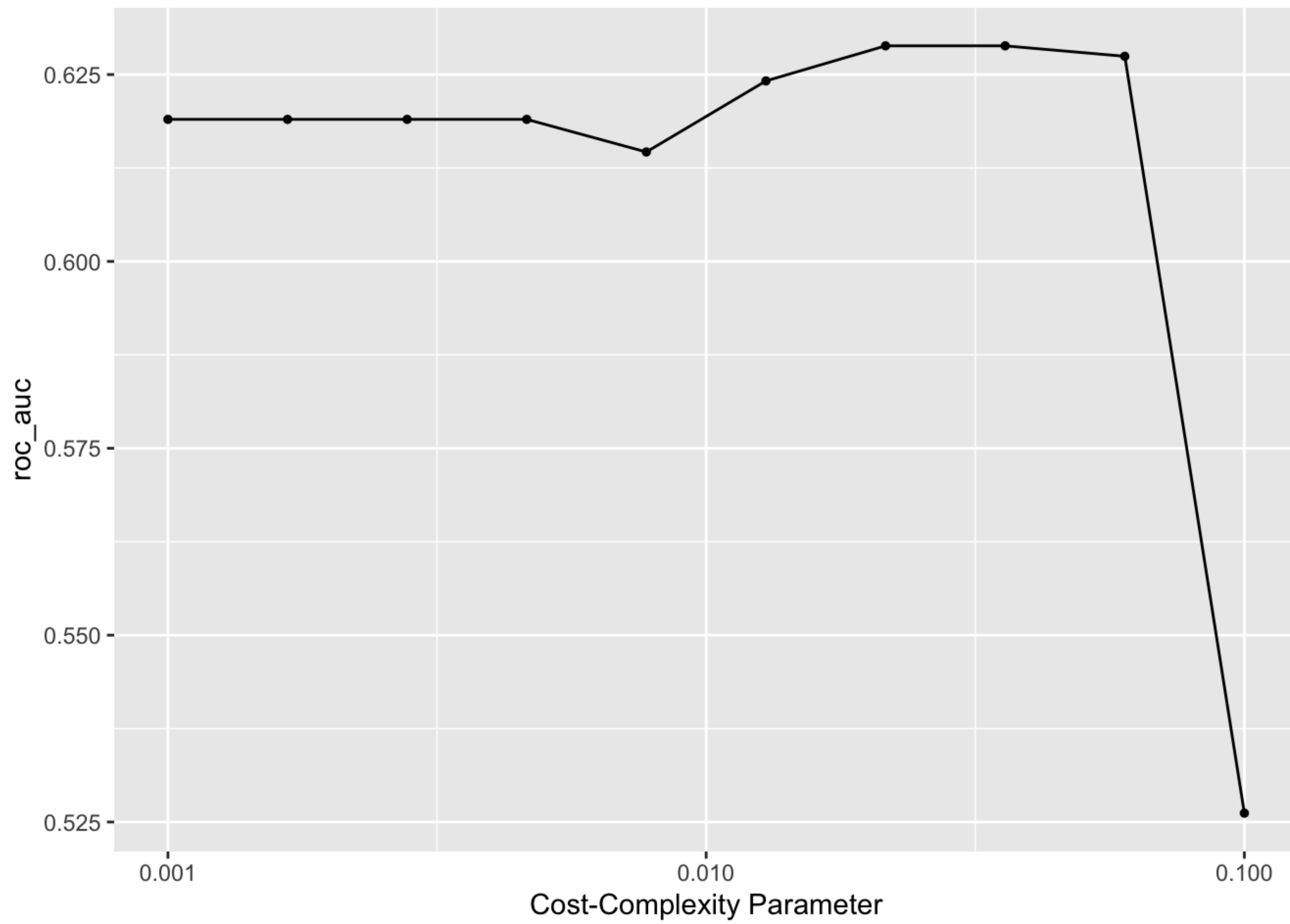
Exercise 3

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?

Hide

```
# set up model and workflow
tree_spec <- decision_tree() %>%
  set_engine("rpart")

class_tree_spec <- tree_spec %>%
  set_mode("classification")

class_tree_wf <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(class_tree_spec %>%
  set_args(cost_complexity = tune()))


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

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

# print results
autoplot(tune_res)
```



The roc_auc stay 0.615 and doesn't change a lot with the parameter increasing, and reach the peak when parameter around 0.02. The roc_auc start dropping after parameter around 0.075.
We can see a single decision tree perform better with a larger 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()`.*

Hide

```
arrange(collect_metrics(tune_res),desc(mean))
```

```
## # A tibble: 10 × 7
##    cost_complexity .metric .estimator  mean     n std_err .config
##              <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1         0.0215  roc_auc hand_till  0.629     5 0.0110  Preprocessor1_Model07
##  2         0.0359  roc_auc hand_till  0.629     5 0.0110  Preprocessor1_Model08
##  3         0.0599  roc_auc hand_till  0.627     5 0.0131  Preprocessor1_Model09
##  4         0.0129  roc_auc hand_till  0.624     5 0.0153  Preprocessor1_Model06
##  5         0.001   roc_auc hand_till  0.619     5 0.00885 Preprocessor1_Model01
##  6         0.00167 roc_auc hand_till  0.619     5 0.00885 Preprocessor1_Model02
##  7         0.00278 roc_auc hand_till  0.619     5 0.00885 Preprocessor1_Model03
##  8         0.00464 roc_auc hand_till  0.619     5 0.00885 Preprocessor1_Model04
##  9         0.00774 roc_auc hand_till  0.615     5 0.0141  Preprocessor1_Model05
## 10         0.1     roc_auc hand_till  0.526     5 0.0262  Preprocessor1_Model10
```

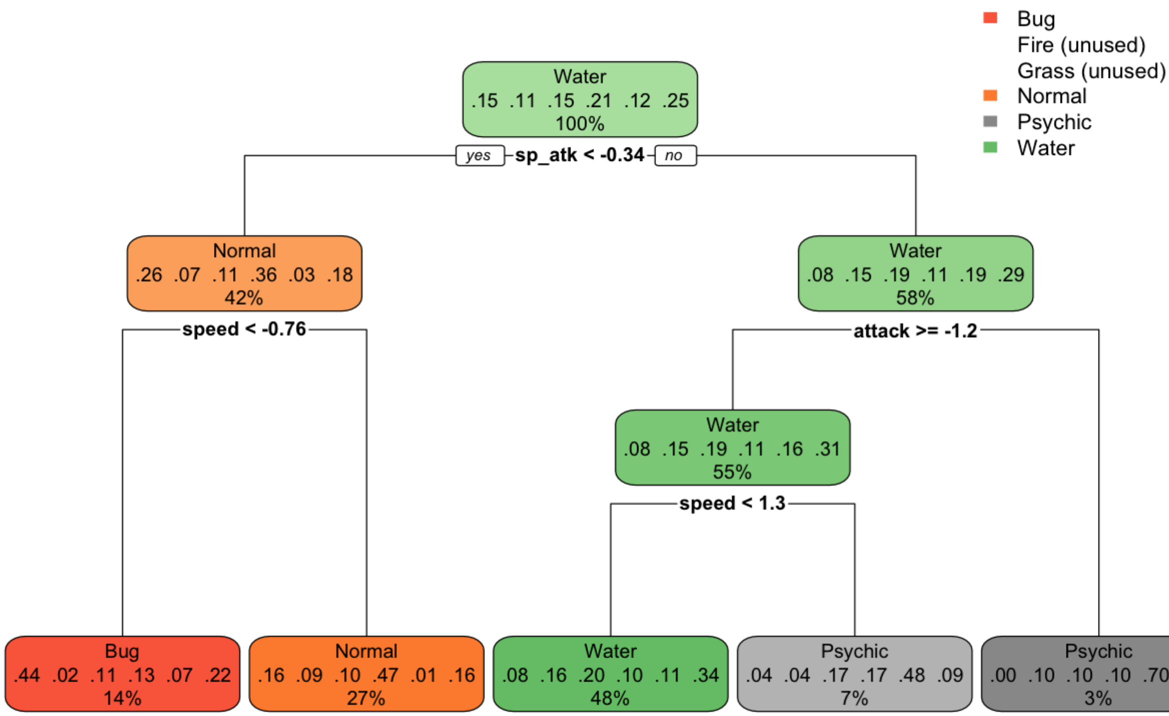
The best model is 0.6288 roc_auc along with 0.0215 parameter.

Exercise 5

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

Hide

```
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?**

Hide

```
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)

param_grid2 <- grid_regular(mtry(range = c(1, 8)),
                           trees(range = c(10, 1000)),
                           min_n(range = c(1, 8)),
                           levels = 8)
```

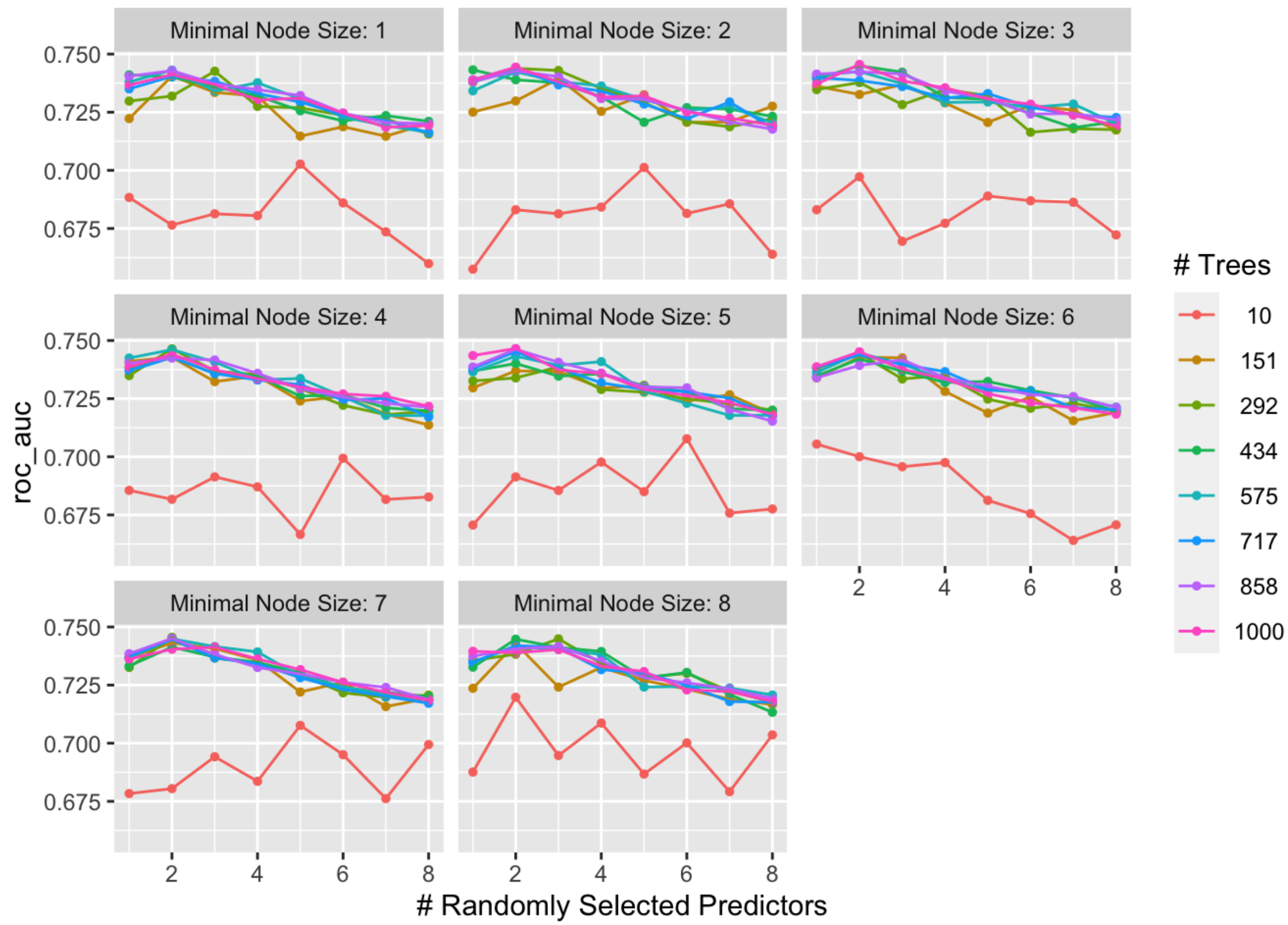
- mtry: The number of predictors that will be randomly sampled with each split of the tree models.
- trees: The number of trees in the ensemble tree models.
- min_n: minimum number of data points in a node required to make a split. *mtry should not be smaller than 1 or larger than 8, since we only have 8 predictors. If we set the it yo 8, it will represent the 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?

Hide

```
tune_res <- tune_grid(
  rf_wf,
  resamples = pokemon_fold,
  grid = param_grid2,
  metrics = metric_set(roc_auc)
)

# print results
autoplot(tune_res)
```



The best model's roc_auc is 0.7489 which contains 2 mtry, 574 trees,and 4 min_n.
In general, The more trees we add the better performance we have, and the roc_auc and the number of predictors are negative relative.

Exercise 7

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

Hide

```
arrange(collect_metrics(tune_res),desc(mean))
```

```
## # 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     2  1000     5 roc_auc hand_till  0.747     5  0.0128 Preprocessor1_Model…
##  2     2   292     4 roc_auc hand_till  0.746     5  0.0138 Preprocessor1_Model…
##  3     2   858     5 roc_auc hand_till  0.746     5  0.0125 Preprocessor1_Model…
##  4     2   575     4 roc_auc hand_till  0.746     5  0.0134 Preprocessor1_Model…
##  5     2  1000     3 roc_auc hand_till  0.746     5  0.0111 Preprocessor1_Model…
##  6     2   292     7 roc_auc hand_till  0.746     5  0.0131 Preprocessor1_Model…
##  7     2   717     5 roc_auc hand_till  0.745     5  0.0123 Preprocessor1_Model…
##  8     2  1000     6 roc_auc hand_till  0.745     5  0.0110 Preprocessor1_Model…
##  9     2   858     7 roc_auc hand_till  0.745     5  0.0125 Preprocessor1_Model…
## 10     2   434     3 roc_auc hand_till  0.745     5  0.0130 Preprocessor1_Model…
## # … with 502 more rows
```

The best model's roc_auc is 0.7489 which contains 2 mtry, 575 trees,and 4 min_n.

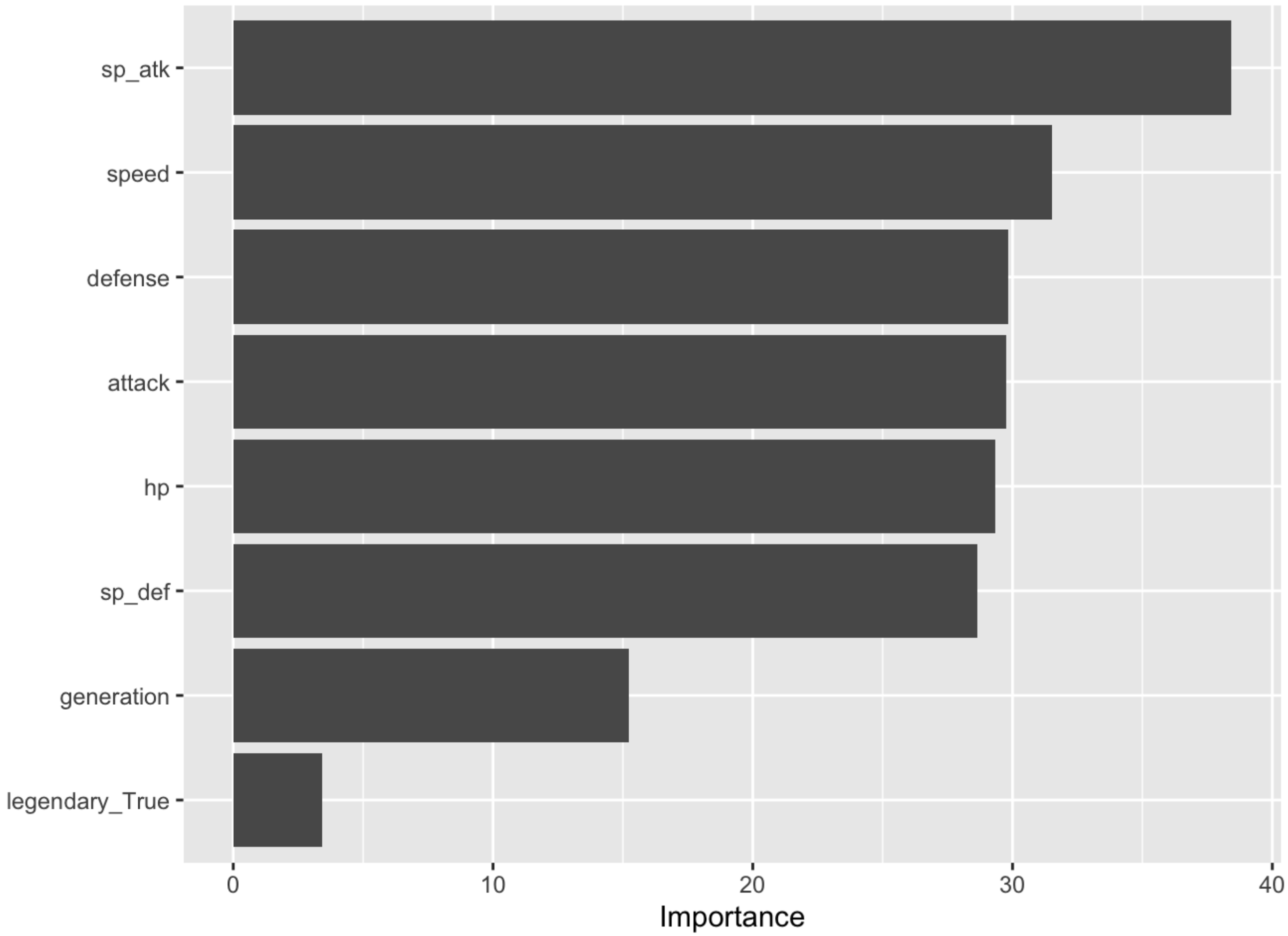
Exercise 8

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?

Hide

```
best_complexity <- select_best(tune_res, metric = "roc_auc")
pokemon_final <- finalize_workflow(rf_wf, best_complexity)
rf_fit <- fit(pokemon_final,data = pokemon_train)
rf_fit %>%
  extract_fit_engine() %>%
  vip()
```



The most useful variable is sp_atk and the least useful variable is legendary_True, I’m so surprise that the sp_atk is the most useful variable, and the other variables are under my consideration.

Exercise 9

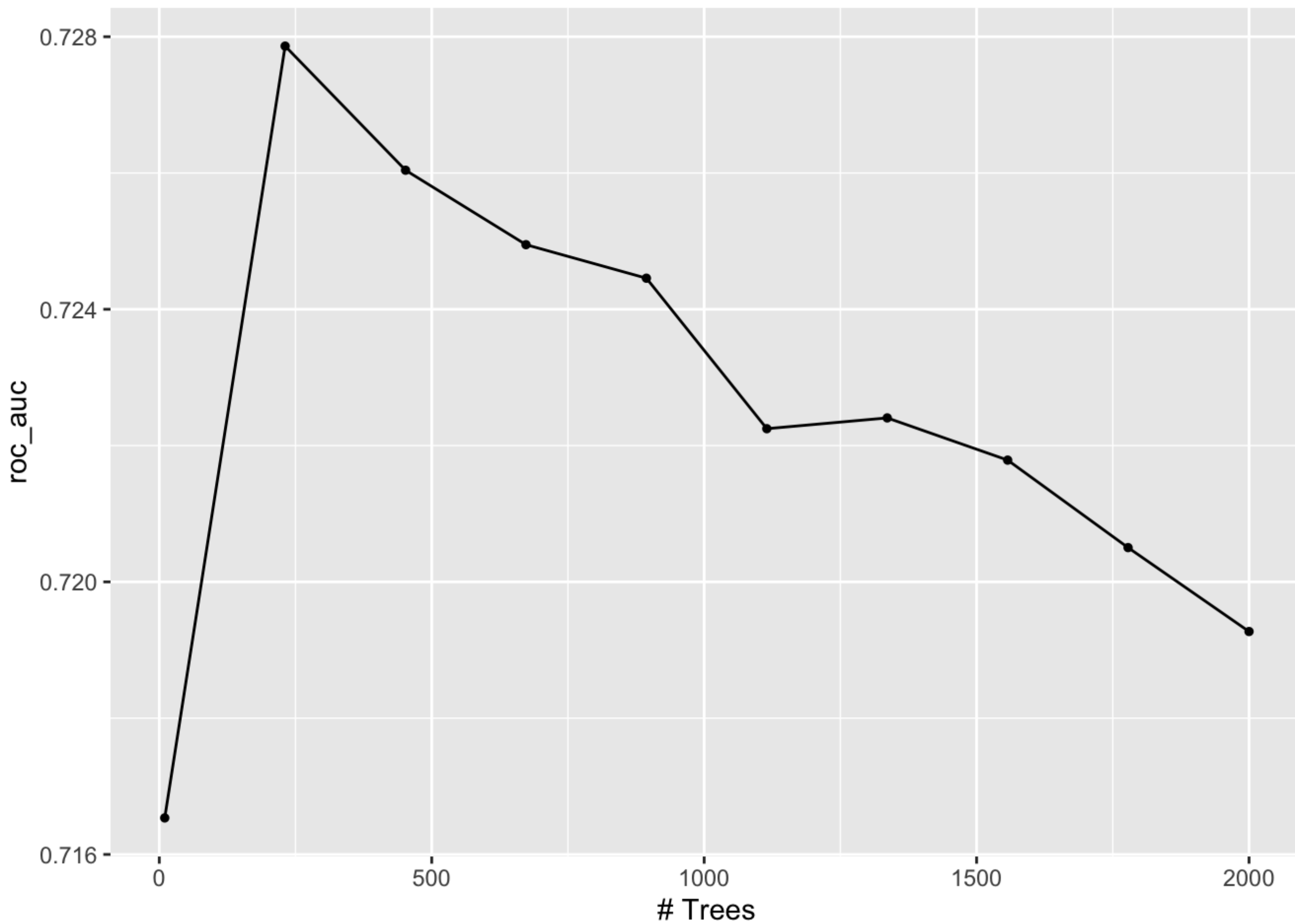
Finally, set up a boosted tree model and workflow. Use the `xgboost` engine. Tune `trees` . Create a regular grid with 10 levels; let `trees` range from 10 to 2000. Specify `roc_auc` and again print an `autoplot()` of the results.

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()`.*

Hide

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

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

boost_tree_wf <- workflow() %>%
  add_model(boost_tree_spec) %>%
  add_formula(type_1 ~ sp_atk + attack + speed +
                defense + hp + sp_def + legendary + generation)

boost_tune_res <- tune_grid(
  boost_tree_wf,
  resamples = pokemon_fold,
  grid = boost_tree_grid,
  metrics = metric_set(roc_auc),
)

autoplot(boost_tune_res)
```



The rou_auc keep increasing until around 250 reach the peak and decrease with the tree number increasing.

Hide

```
arrange(collect_metrics(boost_tune_res),desc(mean))
```

```
## # A tibble: 10 × 7
##    trees .metric .estimator  mean     n std_err .config
##    <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1   231 roc_auc hand_till  0.728     5  0.0179 Preprocessor1_Model02
##  2   452 roc_auc hand_till  0.726     5  0.0178 Preprocessor1_Model03
##  3   673 roc_auc hand_till  0.725     5  0.0171 Preprocessor1_Model04
##  4   894 roc_auc hand_till  0.724     5  0.0174 Preprocessor1_Model05
##  5  1336 roc_auc hand_till  0.722     5  0.0187 Preprocessor1_Model07
##  6  1115 roc_auc hand_till  0.722     5  0.0181 Preprocessor1_Model06
##  7  1557 roc_auc hand_till  0.722     5  0.0190 Preprocessor1_Model08
##  8  1778 roc_auc hand_till  0.721     5  0.0199 Preprocessor1_Model09
##  9  2000 roc_auc hand_till  0.719     5  0.0201 Preprocessor1_Model10
## 10    10 roc_auc hand_till  0.717     5  0.0167 Preprocessor1_Model01
```

The best model has 0.727 roc_auc and 231 trees.

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.

Hide

```
best_boost_tree <- select_best(boost_tune_res)
boost_tree_final <- finalize_workflow(boost_tree_wf, best_boost_tree)
boost_tree_final_fit <- fit(boost_tree_final, data = pokemon_train)
```

Hide

```
final_class_model = augment(class_tree_final_fit, new_data = pokemon_train)
final_rand_model = augment(rf_fit, new_data = pokemon_train)
final_boost_model = augment(boost_tree_final_fit, new_data = pokemon_train)
results<- bind_rows(
  roc_auc(final_class_model, truth = type_1, .pred_Bug, .pred_Fire, .pred_Grass,
          .pred_Normal, .pred_Water, .pred_Psychic),
  roc_auc(final_rand_model, truth = type_1, .pred_Bug, .pred_Fire, .pred_Grass,
          .pred_Normal, .pred_Water, .pred_Psychic),
  roc_auc(final_boost_model, truth = type_1, .pred_Bug, .pred_Fire, .pred_Grass,
          .pred_Normal, .pred_Water, .pred_Psychic)
)
results
```

```
## # A tibble: 3 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.598
## 2 roc_auc hand_till      0.797
## 3 roc_auc hand_till      0.795
```

So we see that the best model is random forest which has roc_auc is 0.79566.

Hide

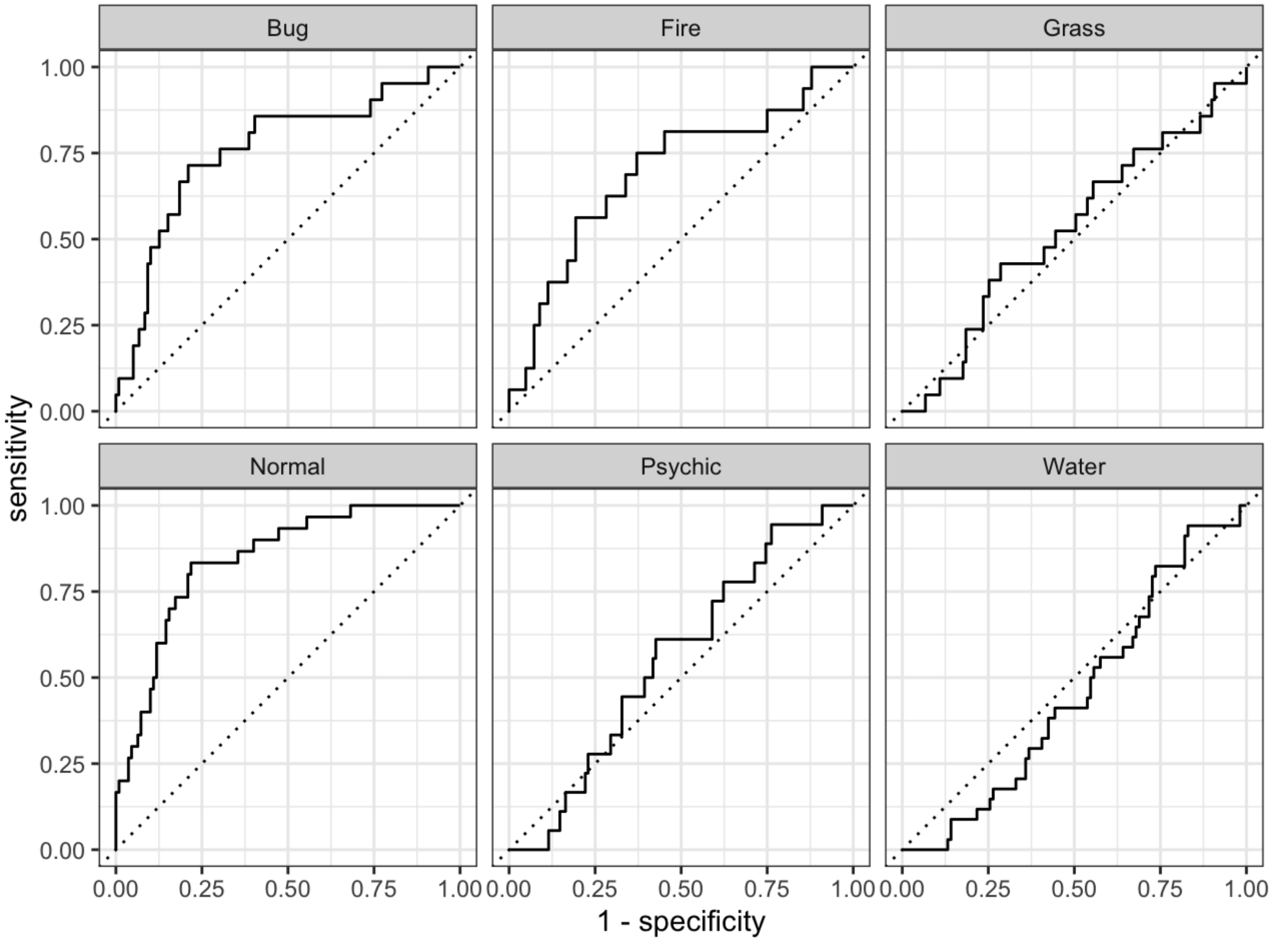
```
final_rand_model_test = augment(rf_fit, new_data = pokemon_test)
roc_auc(final_rand_model_test, truth = type_1, .pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal, .pred_Water, .pre
d_Psychic)
```

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.635
```

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.

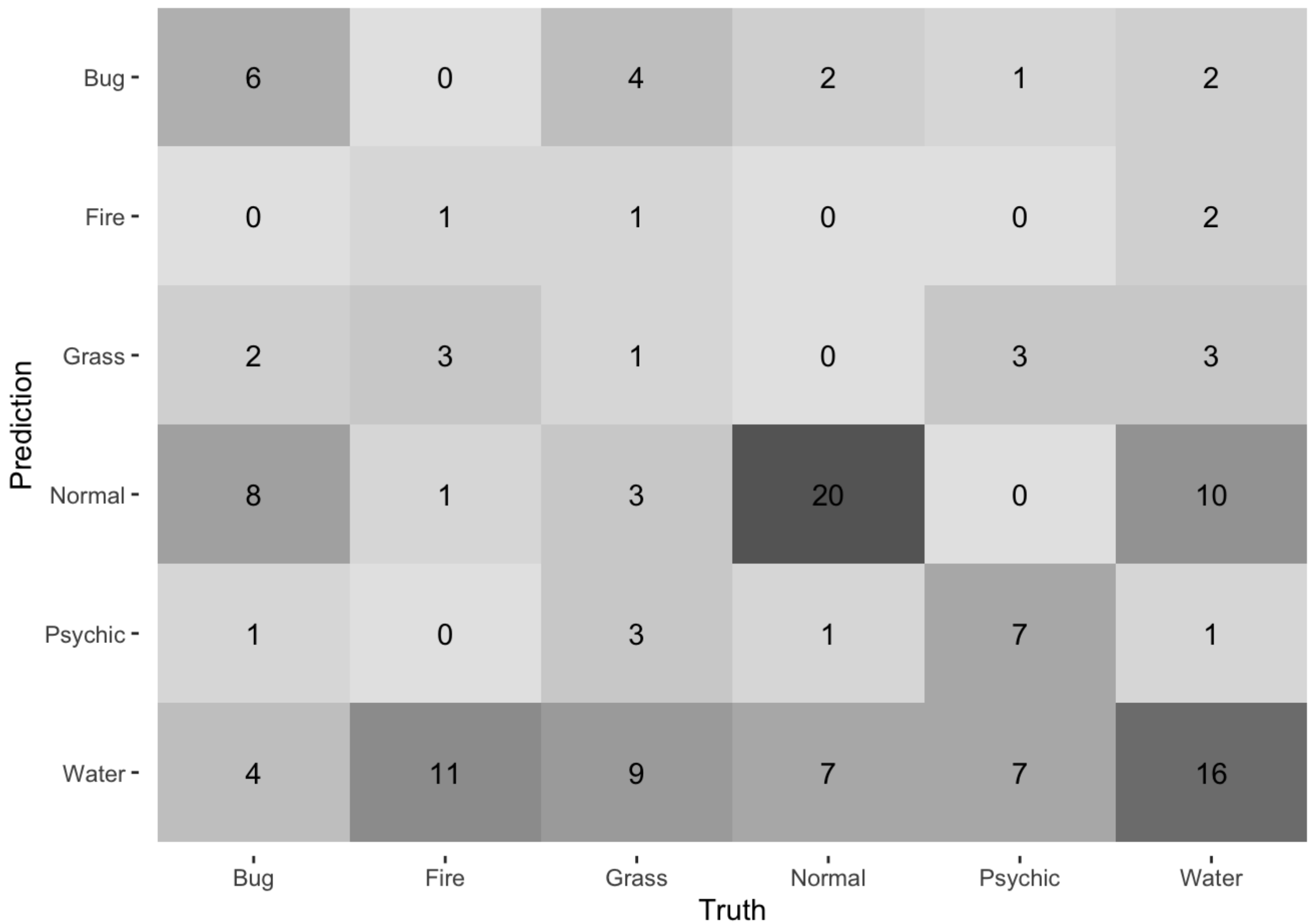
Hide

```
autoplot(roc_curve(final_rand_model_test, truth = type_1, .pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal, .pred
_Water, .pred_Psychic))
```



Hide

```
conf_mat(final_rand_model_test, truth = type_1, estimate = .pred_class) %>% #calclate confusion matri
  autoplot(type = "heatmap")
```



Which classes was your model most accurate at predicting? Which was it worst at?
The model is most accurate at Bug and Normal, and worst at Grass, Psychic, Water.