Homework 5

PSTAT 131

Contents

Homework 5

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

The Pokémon 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 (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.

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. *This is an example of a **classification problem**, but these models can also be used for **regression problems**.*

Read in the file and familiarize yourself with the variables using `pokemon_codebook.txt`.

Exercise 1

Install and load the `janitor` package. Use its `clean_names()` function on the Pokémon data, and save the results to work with for the rest of the assignment. What happened to the data? Why do you think `clean_names()` is useful?

```r
# Load the dataset
pokemon_data <- read_csv("Pokemon.csv")

# View the first few rows of the dataset
head(pokemon_data)
```

```
## # 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
## # i 3 more variables: Speed <dbl>, Generation <dbl>, Legendary <lgl>
```

```
# Clean the column names using clean_names() function
pokemon_data_clean <- pokemon_data %>%
  clean_names()

# View the cleaned dataset
head(pokemon_data_clean)
```

```
## # A tibble: 6 x 13
##   number name       type_1 type_2 total    hp attack defense sp_atk sp_def speed
##    <dbl> <chr>      <chr>  <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
## # i 2 more variables: generation <dbl>, legendary <lgl>
```

```
## So I noticed that "#" column is renamed as "number", and variables in original
#dataset such as 'Name' 'Type 1' is now 'type_1', also columns like 'Sp. Ask' is now
#'sp_atk'. The clean_names() function standardizes the column names in a dataset to make
#them easier to work with.
```

Exercise 2

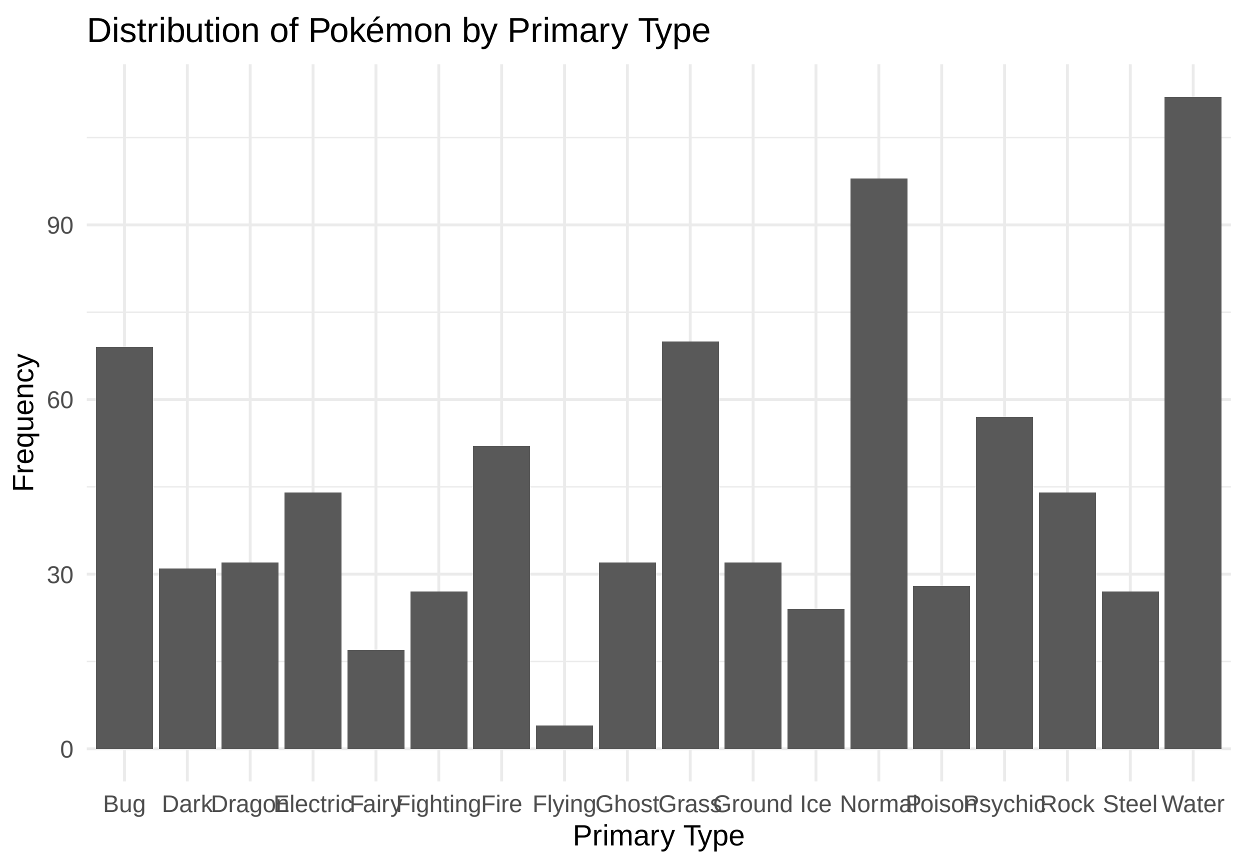
Using the entire data set, create a bar chart of the outcome variable, `type_1`.

How many classes of the outcome are there? Are there any Pokémon types with very few Pokémon? If so, which ones?

For this assignment, we'll handle the rarer classes by grouping them, or "lumping them," together into an 'other' category. Using the `forcats` package, determine how to do this, and **lump all the other levels together except for the top 6 most frequent** (which are Bug, Fire, Grass, Normal, Water, and Psychic).

Convert `type_1`, `legendary`, and `generation` to factors.

```
# First create a bar chart for the 'type_1' variable
ggplot(pokemon_data_clean, aes(x = type_1)) +
  geom_bar() +
  theme_minimal() +
  labs(title = "Distribution of Pokémon by Primary Type",
       x = "Primary Type",
       y = "Frequency")
```



```r
# Count the number of Pokémon for each 'type_1'
type_1_count <- pokemon_data_clean %>%
  count(type_1) %>%
  arrange(desc(n))

# View the count
print(type_1_count)
```

```
## # A tibble: 18 x 2
##    type_1       n
##    <chr>    <int>
##  1 Water      112
##  2 Normal      98
##  3 Grass       70
##  4 Bug         69
##  5 Psychic     57
##  6 Fire        52
##  7 Electric    44
##  8 Rock        44
##  9 Dragon      32
## 10 Ghost       32
## 11 Ground      32
## 12 Dark        31
## 13 Poison      28
## 14 Fighting    27
## 15 Steel       27
## 16 Ice         24
## 17 Fairy       17
## 18 Flying       4
```

```
# Lump the rarer classes into an 'other' category, keeping the top 6 most frequent types
pokemon_data_clean$type_1 <- fct_lump_min(pokemon_data_clean$type_1,
                                          min = min(type_1_count$n[1:6]))

# Convert 'type_1', 'legendary', and 'generation' to factors
pokemon_data_clean$type_1 <- as.factor(pokemon_data_clean$type_1)
pokemon_data_clean$legendary <- as.factor(pokemon_data_clean$legendary)
pokemon_data_clean$generation <- as.factor(pokemon_data_clean$generation)

# Check the structure to confirm the changes
str(pokemon_data_clean)
```

```
## spc_tbl_ [800 x 13] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ number    : num [1:800] 1 2 3 3 4 5 6 6 6 7 ...
##  $ name      : chr [1:800] "Bulbasaur" "Ivysaur" "Venusaur" "VenusaurMega Venusaur" ...
##  $ type_1    : Factor w/ 7 levels "Bug","Fire","Grass",..: 3 3 3 3 2 2 2 2 2 6 ...
##  $ type_2    : chr [1:800] "Poison" "Poison" "Poison" "Poison" ...
##  $ total     : num [1:800] 318 405 525 625 309 405 534 634 634 314 ...
##  $ hp        : num [1:800] 45 60 80 80 39 58 78 78 78 44 ...
##  $ attack    : num [1:800] 49 62 82 100 52 64 84 130 104 48 ...
##  $ defense   : num [1:800] 49 63 83 123 43 58 78 111 78 65 ...
##  $ sp_atk    : num [1:800] 65 80 100 122 60 80 109 130 159 50 ...
##  $ sp_def    : num [1:800] 65 80 100 120 50 65 85 85 115 64 ...
##  $ speed     : num [1:800] 45 60 80 80 65 80 100 100 100 43 ...
##  $ generation: Factor w/ 6 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ legendary : Factor w/ 2 levels "FALSE","TRUE": 1 1 1 1 1 1 1 1 1 1 ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   `#` = col_double(),
##   ..   Name = col_character(),
##   ..   `Type 1` = col_character(),
##   ..   `Type 2` = col_character(),
##   ..   Total = col_double(),
##   ..   HP = col_double(),
##   ..   Attack = col_double(),
##   ..   Defense = col_double(),
##   ..   `Sp. Atk` = col_double(),
##   ..   `Sp. Def` = col_double(),
##   ..   Speed = col_double(),
##   ..   Generation = col_double(),
##   ..   Legendary = col_logical()
##   .. )
##  - attr(*, "problems")=<externalptr>
```

Exercise 3

Perform an initial split of the data. Stratify by the outcome variable. You can choose a proportion to use. Verify that your training and test sets have the desired number of observations.

Next, use *v*-fold cross-validation on the training set. Use 5 folds. Stratify the folds by `type_1` as well. *Hint: Look for a `strata` argument.*

Why do you think doing stratified sampling for cross-validation is useful?

```r
# Set seed for reproducibility
set.seed(123)

# Perform stratified sampling to split the data into training and testing sets
initial_split_obj <- initial_split(pokemon_data_clean, prop = 0.7,
                                   strata = type_1)
train_data <- training(initial_split_obj)
test_data <- testing(initial_split_obj)

# Verify the number of observations in each set
cat("Number of observations in the training set:", nrow(train_data), "\n")
```

```
## Number of observations in the training set: 558
```

```r
cat("Number of observations in the test set:", nrow(test_data), "\n")
```

```
## Number of observations in the test set: 242
```

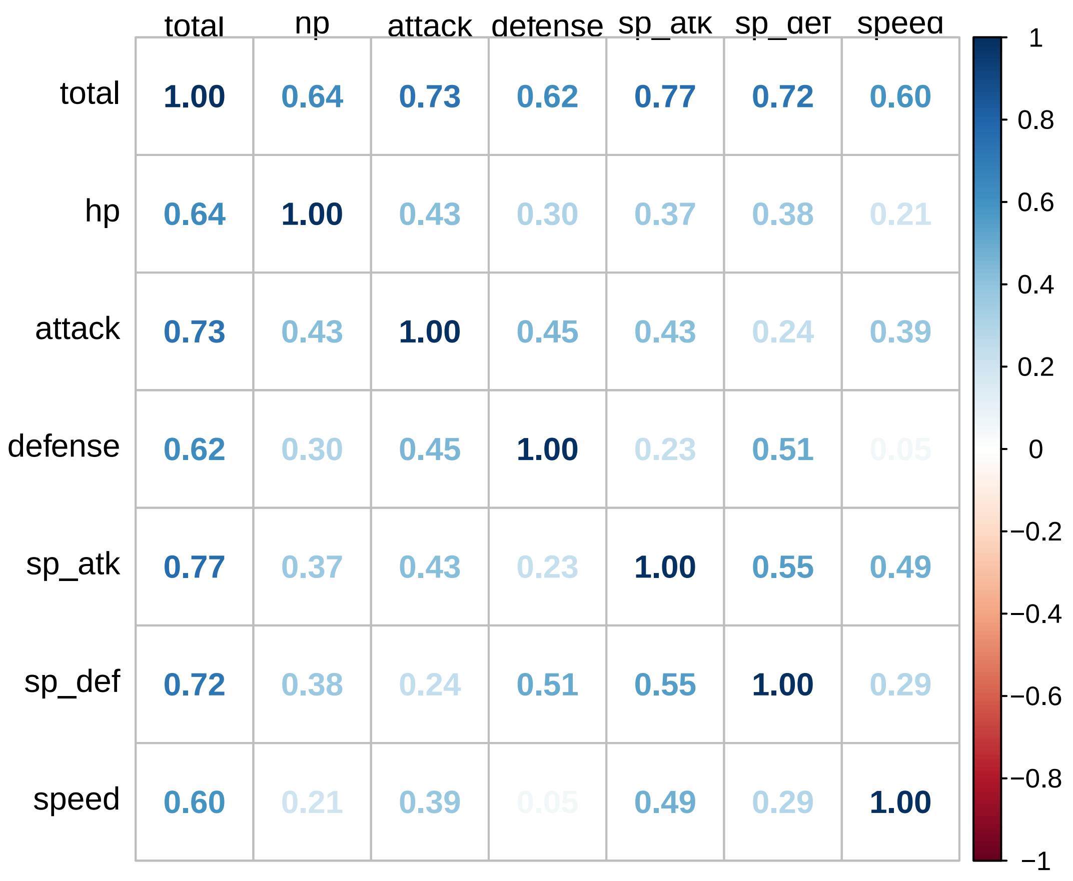
```r
# Perform 5-fold cross-validation on the training set, stratified by 'type_1'
cv_folds <- vfold_cv(train_data, v = 5, strata = type_1)

# It is useful when the outcome variable has imbalanced classes. Stratification ensures
#that each class is equally represented in each fold during cross-validation, providing
#a more robust measure of the model's performance.
```

Exercise 4

Create a correlation matrix of the training set, using the `corrplot` package. *Note: You can choose how to handle the categorical variables for this plot; justify your decision(s).*

What relationships, if any, do you notice?

```r
# Calculate the correlation matrix for numerical variables in the training set.
numerical_vars <- select(train_data, total, hp, attack, defense, sp_atk,
                         sp_def, speed)
cor_matrix <- cor(numerical_vars)

# I chose to exclude the categorical variables from the correlation matrix. The reason
#is that correlation matrices are typically most informative for continuous variable

# Create the correplot
corrplot(cor_matrix, method = "number", tl.col = "black", tl.srt = 0)
```



```
# As expected, the total stat has a moderate to strong positive correlation with all
#other stats, ranging from 0.60 to 0.77. This makes sense, as total is a composite
#measure that includes these other stats.

# Speed has the lowest correlation with defense (0.05), suggesting almost no linear
#relationship between the two.

# There's a strong correlation of 0.77 between these two, indicating that Pokémon with
#higher special attacks tend to have a higher total stat value.

# Stats related to offense (attack, sp_atk) and defense (defense, sp_def) are moderately
#correlated within their categories.
```

Exercise 5

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.

```
# Create the Pokémon recipe
pokemon_recipe <- recipe(type_1 ~ legendary + generation + sp_atk + attack +
                           speed + defense + hp + sp_def,
```

```r
                          data = train_data) %>%
  step_novel(all_nominal_predictors(), -all_outcomes()) %>%
  step_dummy(legendary, generation, one_hot = TRUE) %>%
  step_zv(all_predictors()) %>%
  step_center(all_predictors()) %>%
  step_scale(all_predictors())
```

Exercise 6

We'll be fitting and tuning an elastic net, tuning `penalty` and `mixture` (use `multinom_reg()` with the `glmnet` engine).

Set up this model and workflow. Create a regular grid for `penalty` and `mixture` with 10 levels each; `mixture` should range from 0 to 1. For this assignment, let `penalty` range from 0.01 to 3 (this is on the `identity_trans()` scale; note that you'll need to specify these values in base 10 otherwise).

```r
# Specify the Elastic Net model using multinom_reg() and the glmnet engine
elastic_net_spec <- multinom_reg(penalty = tune(),
                                 mixture = tune()) %>%
  set_engine("glmnet")

# Create a regular grid for 'penalty' and 'mixture'
tune_grid <- grid_regular(
  penalty(range = c(0.01, 3)),
  mixture(range = c(0, 1)),
  levels = 10
)

# Combine the recipe and model specification into a workflow
elastic_net_workflow <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(elastic_net_spec)

# Print the workflow to verify its structure
print(elastic_net_workflow)
```

```
## == Workflow ====================================================================
## Preprocessor: Recipe
## Model: multinom_reg()
##
## -- Preprocessor ----------------------------------------------------------------
## 5 Recipe Steps
##
## * step_novel()
## * step_dummy()
## * step_zv()
## * step_center()
## * step_scale()
##
## -- Model -----------------------------------------------------------------------
## Multinomial Regression Model Specification (classification)
##
## Main Arguments:
```

```
##   penalty = tune()
##   mixture = tune()
##
## Computational engine: glmnet
```

```r
# Print the tuning grid to verify its structure
print(tune_grid)
```

```
## # A tibble: 100 x 2
##    penalty mixture
##      <dbl>   <dbl>
##  1    1.02       0
##  2    2.20       0
##  3    4.73       0
##  4   10.2        0
##  5   21.8        0
##  6   46.9        0
##  7  101.         0
##  8  217.         0
##  9  465.         0
## 10 1000          0
## # i 90 more rows
```

Exercise 7

Now set up a random forest model and workflow. Use the `ranger` engine and set `importance = "impurity"`; we'll be tuning `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 neither of those values would make sense.**

What type of model does `mtry = 8` represent?

```r
# Specify the Random Forest model using rand_forest() and the ranger engine
random_forest_spec <- rand_forest(
  mtry = tune(),
  trees = tune(),
  min_n = tune(),
  mode = "classification") %>%
  set_engine("ranger", importance = "impurity")

# Create a regular grid for 'mtry', 'trees', and 'min_n'
rf_tune_grid <- grid_regular(
  mtry(range = c(1, 8)),
  trees(range = c(50, 500)),
  min_n(range = c(2, 20)),
  levels = 8
)

# Combine the recipe and model specification into a workflow
rf_workflow <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(random_forest_spec)

# Print the workflow to verify its structure
print(rf_workflow)
```

```
## == Workflow ====================================================================
## Preprocessor: Recipe
## Model: rand_forest()
##
## -- Preprocessor ----------------------------------------------------------------
## 5 Recipe Steps
##
## * step_novel()
## * step_dummy()
## * step_zv()
## * step_center()
## * step_scale()
##
## -- Model -----------------------------------------------------------------------
## Random Forest Model Specification (classification)
##
## Main Arguments:
##   mtry = tune()
##   trees = tune()
##   min_n = tune()
##
## Engine-Specific Arguments:
##   importance = impurity
##
## Computational engine: ranger
```

```
# Print the tuning grid to verify its structure
print(rf_tune_grid)
```

```
## # A tibble: 512 x 3
##     mtry trees min_n
##    <int> <int> <int>
##  1     1    50     2
##  2     2    50     2
##  3     3    50     2
##  4     4    50     2
##  5     5    50     2
##  6     6    50     2
##  7     7    50     2
##  8     8    50     2
##  9     1   114     2
## 10     2   114     2
## # i 502 more rows
```

Exercise 8

Fit all models to your folded data using `tune_grid()`.

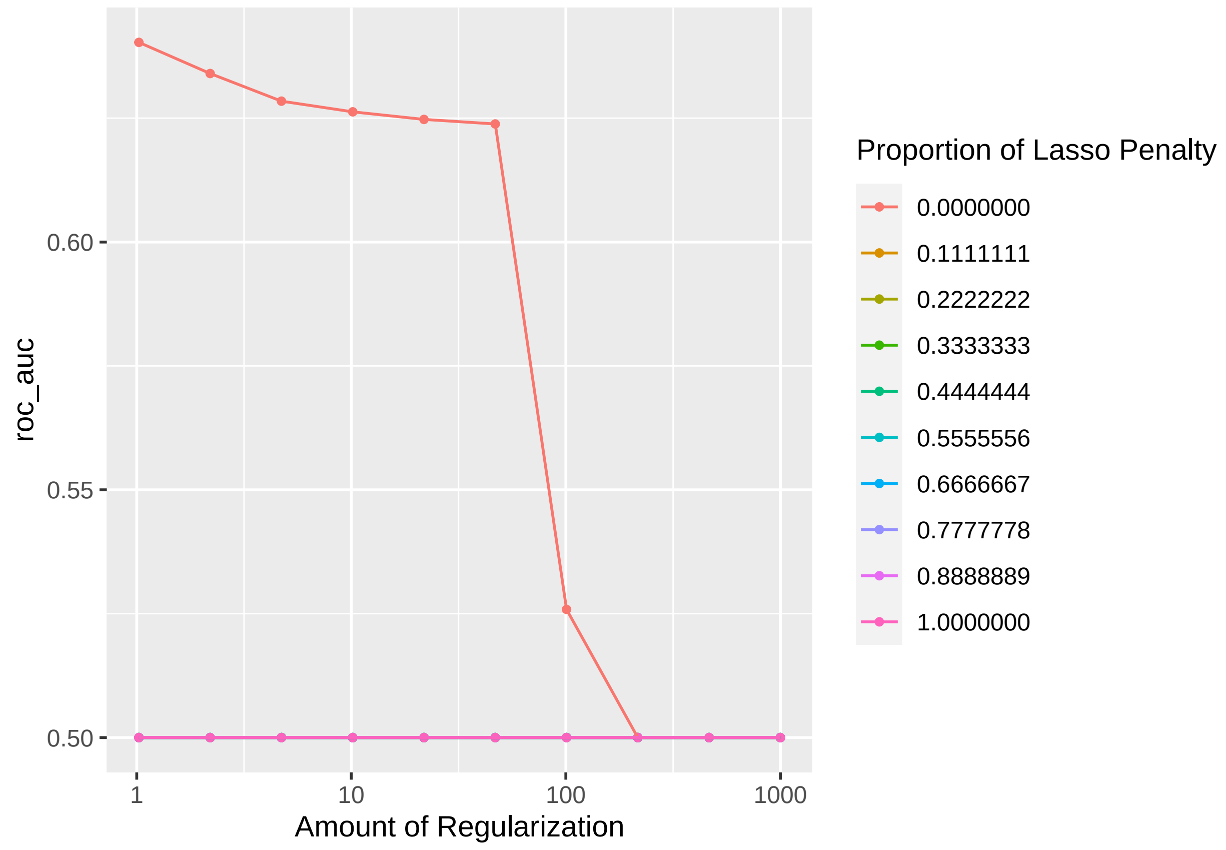
Note: Tuning your random forest model will take a few minutes to run, anywhere from 5 minutes to 15 minutes and up. Consider running your models outside of the .Rmd, storing the results, and loading them in your .Rmd to minimize time to knit. We'll go over how to do this in lecture.

Use `autoplot()` on the results. What do you notice? Do larger or smaller values of `penalty` and `mixture` produce better ROC AUC? What about values of `min_n`, `trees`, and `mtry`?

What elastic net model and what random forest model perform the best on your folded data? (What specific values of the hyperparameters resulted in the optimal ROC AUC?)

```r
# Perform grid search for Elastic Net
enet_results <- tune_grid(
  elastic_net_workflow,
  resamples = cv_folds,
  grid = tune_grid
)

# Perform grid search for Random Forest
rf_results <- tune_grid(
  rf_workflow,
  resamples = cv_folds,
  grid = rf_tune_grid
)

# Save the results object to a .rds file
saveRDS(enet_results, "enet_results.rds")
saveRDS(rf_results, "rf_results.rds")
```

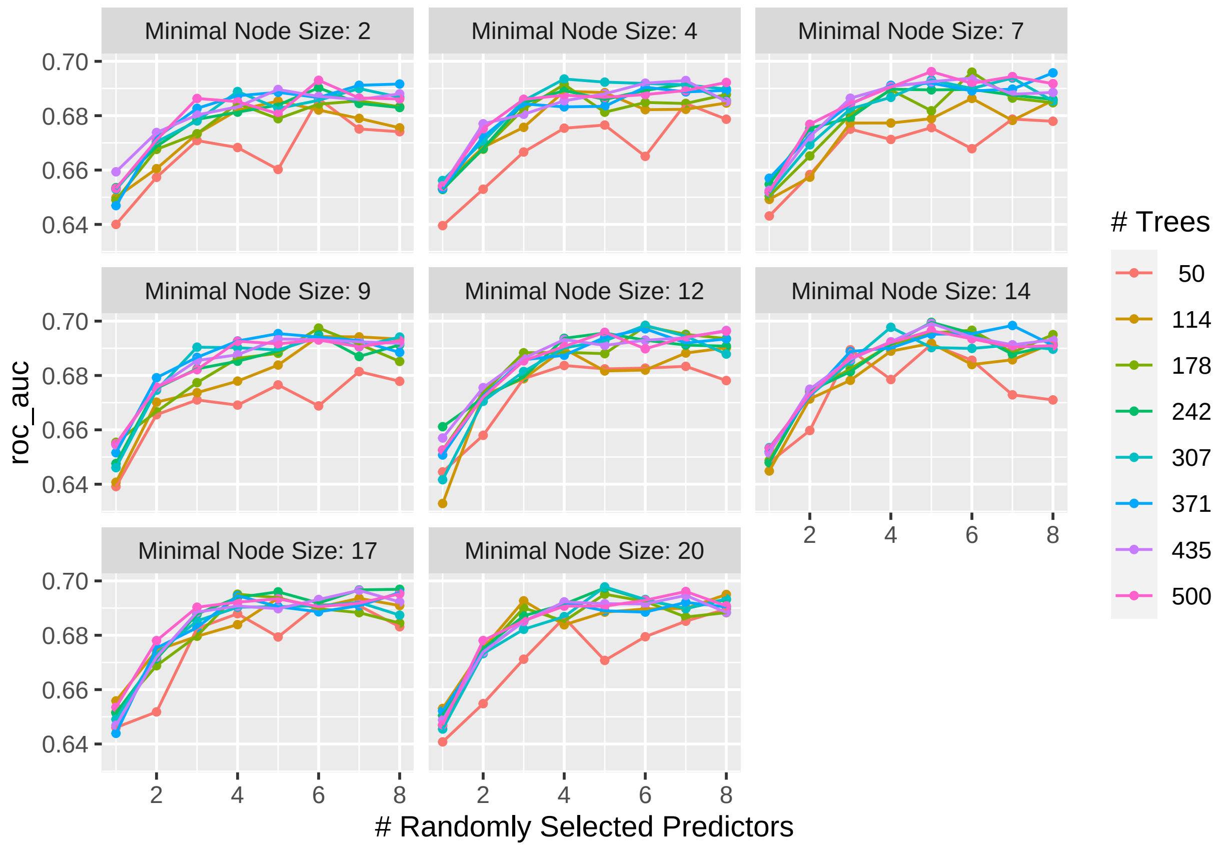
```r
# Load Pre-trained Model.
# Load the enet_results object from the .rds file
enet_results <- readRDS("enet_results.rds")

# Load the rf_results object from the .rds file
rf_results <- readRDS("rf_results.rds")

# Autoplot for Elastic Net
autoplot(enet_results, metric = "roc_auc")
```



```r
show_notes(enet_results)
```

```
## Great job! No notes to show.
```

```r
# Autoplot for Random Forest
autoplot(rf_results, metric = "roc_auc")
```



```r
show_notes(rf_results)
```

```
## Great job! No notes to show.
```

```r
## From the plot of ENET model, we can conclude that as the amount of
#regularization
#increases, the ROC_AUC goes down, which suggests that smaller values of
#'penalty' might
#be better. Also, the highest ROC AUC occurred when the proportion of Lasso
#penalty was
#zero, indicating that Ridge Regression (mixture = 0) is favored.

## By observing the plot of RF model, a larger ensemble of trees (closer to
#500) is generally beneficial for this dataset. And the model seems to perform
#best when using a subset of 4-7 predictors, suggesting that some features are
#more informative than others.
```

```r
# Identifying the Best Model:
best_enet <- enet_results %>% show_best("roc_auc", n = 1)
best_rf <- rf_results %>% show_best("roc_auc", n = 1)

# Show the best models
print(best_enet)
```

```
## # A tibble: 1 x 8
##   penalty mixture .metric .estimator  mean     n std_err .config
##     <dbl>   <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1    1.02       0 roc_auc hand_till  0.640     5 0.00531 Preprocessor1_Model001
```

```
print(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     5   242    14 roc_auc hand_till  0.700     5 0.00927 Preprocessor1_Model3~
```

```
# From the outputs above, for best Elastic Net Model, we have: Penalty:
#1.023293,
#Mixture: 0 (This indicates that the model favored Ridge Regression over
#Lasso), Optimal
#ROC AUC: 0.640295, Number of Folds: 5, and Standard Error: 0.005308037
# And for best Random Forest model, we have Number of Randomly Selected
#Predictors: 5, Number of Trees: 242, Minimal Node Size: 14, Optimal ROC AUC:
#0.6995358, Number of Folds: 5, Standard Error: 0.009270437
```

Exercise 9

Select your optimal **random forest model**in terms of `roc_auc`. Then fit that model to your training set and evaluate its performance on the testing set.

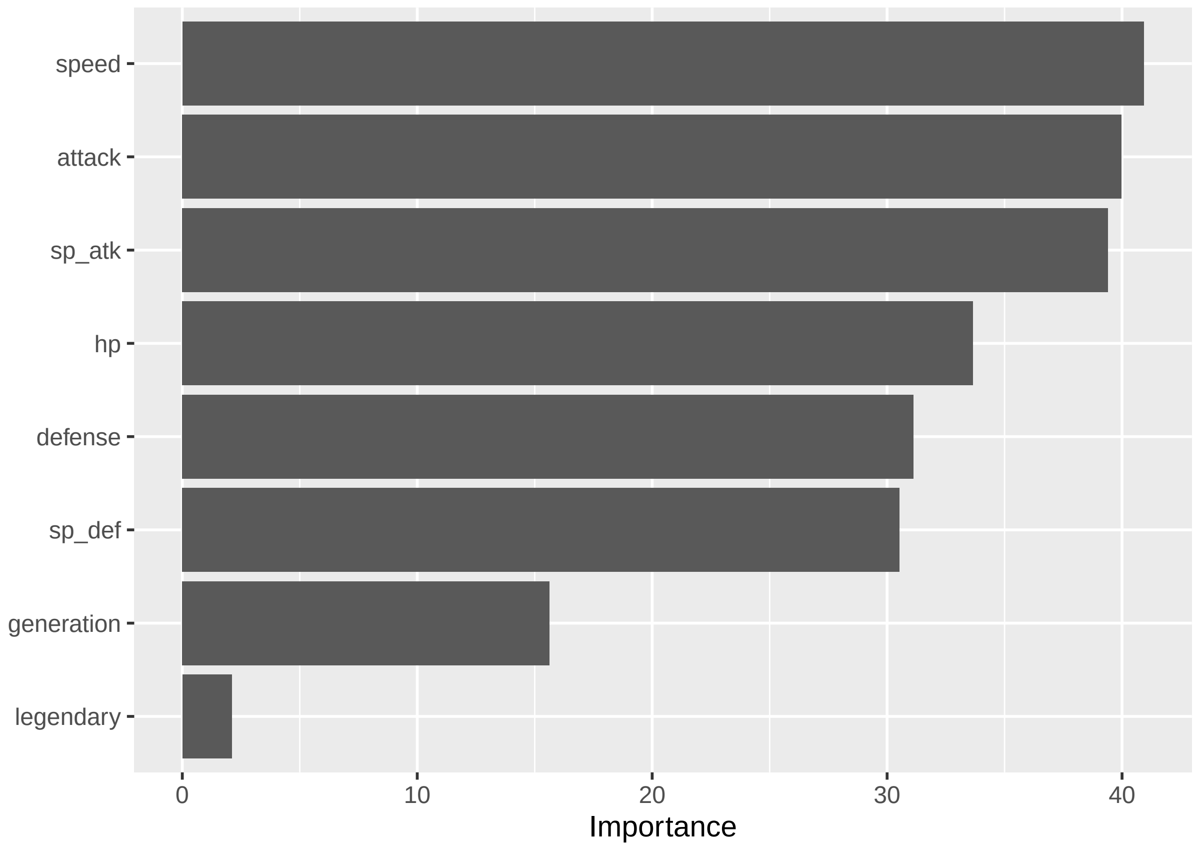
Using the **training** set:

- Create a variable importance plot, using `vip()`. *Note that you'll still need to have set* `importance = "impurity"` *when fitting the model to your entire training set in order for this to work.*
  - What variables were most useful? Which were least useful? Are these results what you expected, or not?

Using the testing set:

- Create plots of the different ROC curves, one per level of the outcome variable;
- Make a heat map of the confusion matrix.

```
# Fit the optimal Random Forest model to the training set
best_rf_model <- rand_forest(
  mtry = 5,
  trees = 242,
  min_n = 14,
  mode = "classification") %>%
  set_engine("ranger", importance = "impurity") %>%
  fit(type_1 ~ legendary + generation + sp_atk + attack + speed +
        defense + hp + sp_def,
      data = train_data)

# Create the variable importance plot
vip(best_rf_model)
```

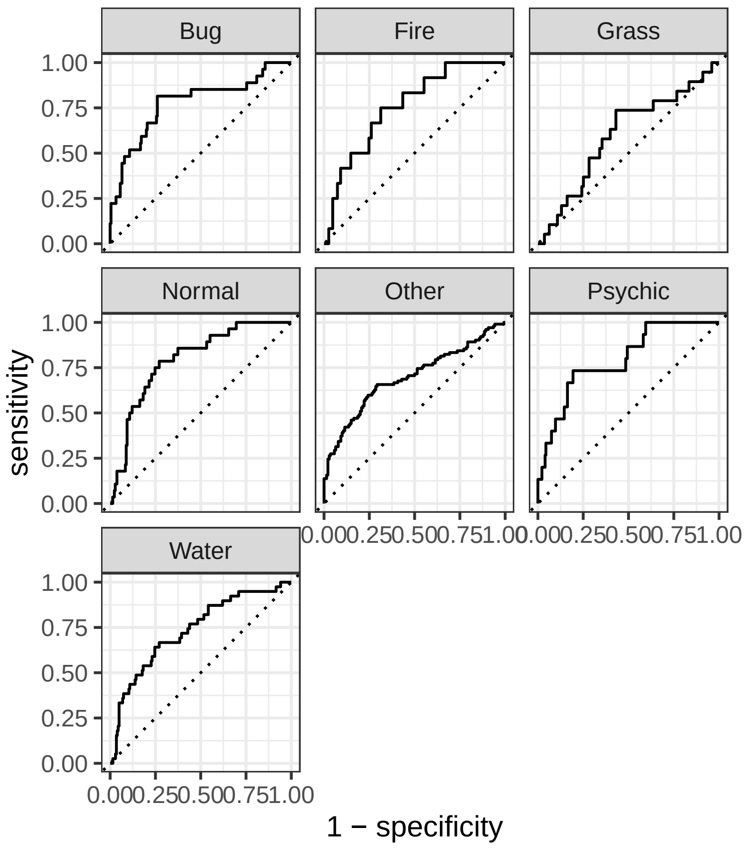


```
# By observing the plot, 'Speed' seems to be the most useful, while legendary seems to be the least use
```

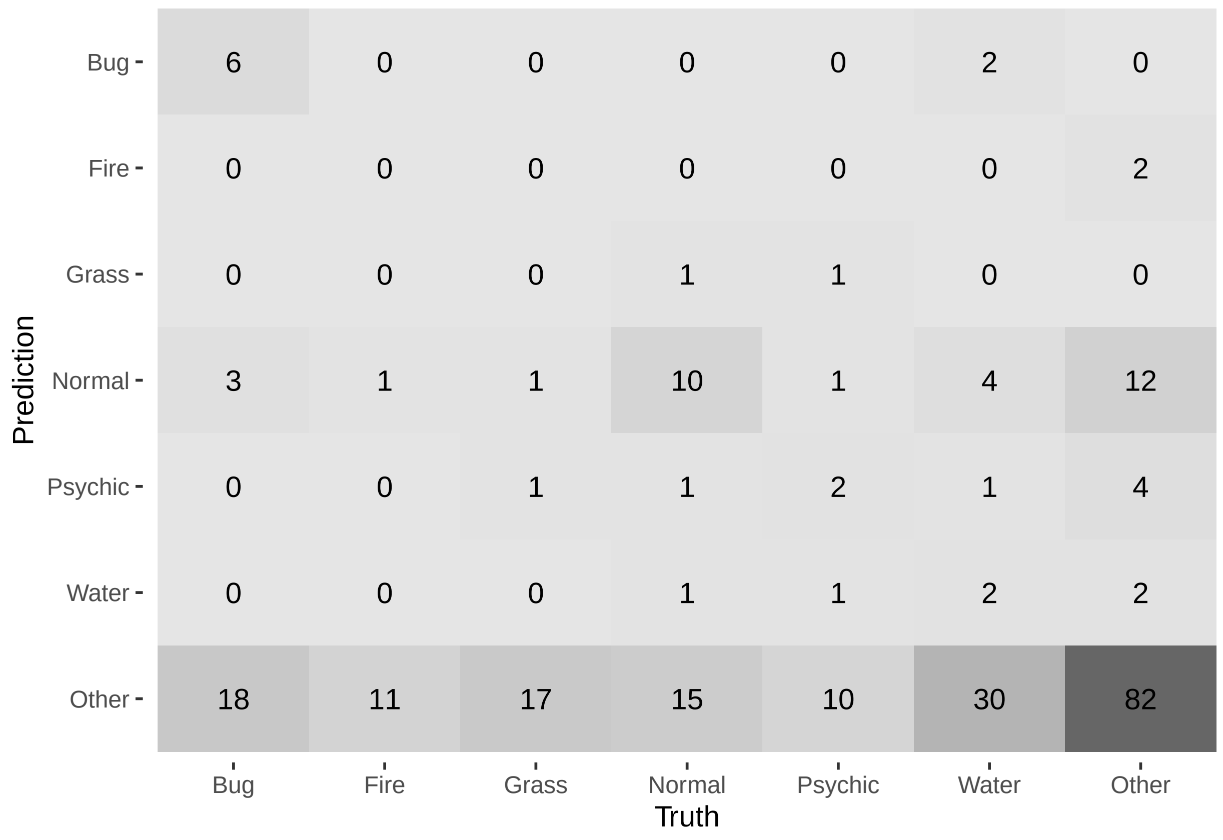
```
# Augment the model to get the predictions
best_rf_model_test <- augment(best_rf_model, test_data) %>%
  select(type_1, starts_with(".pred"))

# Calculate the ROC AUC for each class
roc_auc(best_rf_model_test, truth = type_1, .pred_Bug:.pred_Other)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.726
```

```
# Generate and plot the ROC curves for each class
roc_curve(best_rf_model_test, truth = type_1, .pred_Bug:.pred_Other) %>%
  autoplot()
```



```
conf_mat(best_rf_model_test, truth = type_1,
         .pred_class) %>%
  autoplot(type = "heatmap")
```



Exercise 10

How did your best random forest model do on the testing set?

Which Pokemon types is the model best at predicting, and which is it worst at? (Do you have any ideas why this might be?)

```
# From the output in Q9, we have the ROC AUC value for the best random forest
#model on the testing set is approximately 0.727. This suggests that the model
#has a reasonably good ability to distinguish between the different classes.

# The model is best at predicting 'Psychic', and worst at predicting 'Other'.
# Reasons: 1. Probably because the data used for training are biased, given
#that we randomly chose 70% from the original dataset to be the tranning data.
# 2. Some types may have similar feature values, making them difficult to
#distinguish. For example, 'Others' type may contain less unique features.
```