PSTAT 131 Homework 5

Luke Fields (8385924)

May 11, 2022

Below are the packages and libraries we are using in this assignment.

```
library(corrplot)
library(discrim)
library(corrr)
library(knitr)
library(MASS)
library(tidyverse)
library(tidymodels)
library(ggplot2)
library(glmnet)
library("dplyr")
library("yardstick")
tidymodels_prefer()
pokemon <- read_csv("Pokemon.csv")
# set global chunk options: images will be 7x5 inches
knitr::opts_chunk$set(
    echo = TRUE,
    fig.height = 5,
    fig.width = 7,
    tidy = TRUE,
    tidy.opts = list(width.cutoff = 60)
)
opts_chunk$set(tidy.opts=list(width.cutoff=60),tidy=TRUE)
options(digits = 4)



## indents are for indenting r code as formatted text
## They may need to be adjusted depending on your OS
# if your output looks odd, increase or decrease indent
indent1 = '    '
indent2 = '        '
indent3 = '            '
```

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?

```
pokemon
```

```
## # A tibble: 800 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 Bulba~ Grass    Poison     318    45     49      49        65        65
##  2     2 Ivysa~ Grass    Poison     405    60     62      63        80        80
##  3     3 Venus~ Grass    Poison     525    80     82      83       100       100
##  4     3 Venus~ Grass    Poison     625    80    100     123       122       120
##  5     4 Charm~ Fire     <NA>       309    39     52      43        60        50
##  6     5 Charm~ Fire     <NA>       405    58     64      58        80        65
##  7     6 Chari~ Fire     Flying     534    78     84      78       109        85
##  8     6 Chari~ Fire     Dragon     634    78    130     111       130        85
##  9     6 Chari~ Fire     Flying     634    78    104      78       159       115
## 10     7 Squir~ Water    <NA>       314    44     48      65        50        64
## # ... with 790 more rows, and 3 more variables: Speed <dbl>, Generation <dbl>,
## #   Legendary <lgl>
```

```
library(janitor)
pokemon <- pokemon %>%
  clean_names
pokemon
```

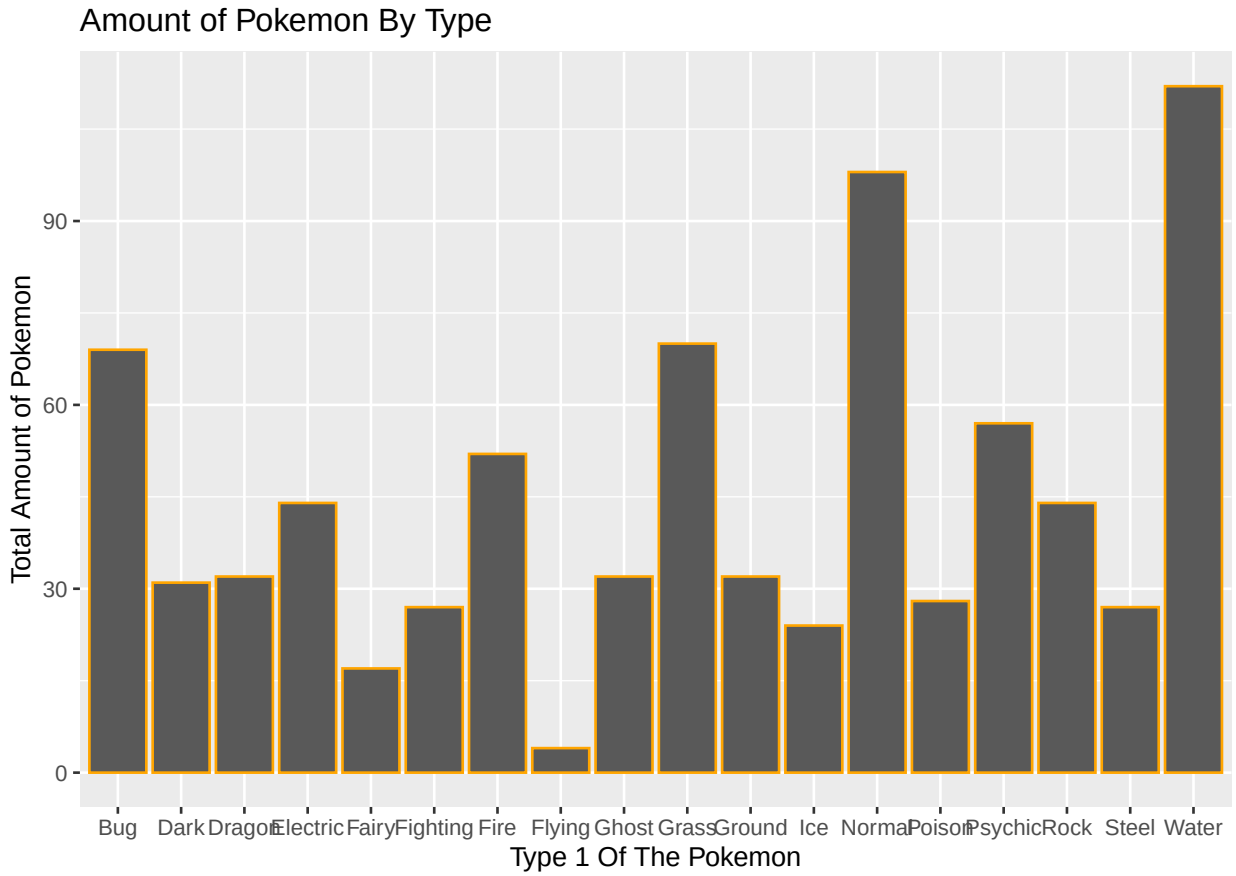
```
## # A tibble: 800 x 13
##    number name      type_1 type_2 total    hp attack defense sp_atk sp_def speed
##     <dbl> <chr>     <chr>  <chr>  <dbl> <dbl>  <dbl>   <dbl>  <dbl>  <dbl> <dbl>
##  1      1 Bulbasaur Grass  Poison   318    45     49      49     65     65    45
##  2      2 Ivysaur   Grass  Poison   405    60     62      63     80     80    60
##  3      3 Venusaur  Grass  Poison   525    80     82      83    100    100    80
##  4      3 Venusaur~ Grass  Poison   625    80    100     123    122    120    80
##  5      4 Charmand~ Fire   <NA>     309    39     52      43     60     50    65
##  6      5 Charmele~ Fire   <NA>     405    58     64      58     80     65    80
##  7      6 Charizard Fire   Flying   534    78     84      78    109     85   100
##  8      6 Charizar~ Fire   Dragon   634    78    130     111    130     85   100
##  9      6 Charizar~ Fire   Flying   634    78    104      78    159    115   100
## 10      7 Squirtle  Water  <NA>     314    44     48      65     50     64    43
## # ... with 790 more rows, and 2 more variables: generation <dbl>,
## #   legendary <lgl>
```

Clean_names is useful because it helps clean up the variable names that may have special characters that mean the same thing but are different literally, like the difference between "Charizard_X" and "Charizard X", but these are obviously the same Pokemon. For our data, the variables changed to lowercase and added underscores from something like "Type 1" to "type_1" to make everything more standard.

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 simply filtering them out. Filter the entire data set to contain only Pokémon whose type_1 is Bug, Fire, Grass, Normal, Water, or Psychic. After filtering, convert type_1 and legendary to factors.

First, we create our bar plot.

```
bar_type1_pokemon <- ggplot(pokemon, aes(x = type_1)) +
  geom_bar(color = "orange")
bar_type1_pokemon + labs(title = "Amount of Pokemon By Type", x = "Type 1 Of The Pokemon", y = "Total An
```



Now, we get how many different types there are.

```
n_distinct(pokemon$type_1)
```

```
## [1] 18
```

From this bar plot, we can see that there 18 different classes of the outcome, so there are 18 different Type 1's for a Pokemon. There are very few fairy, fighting, ground, ice, poison, and steel Type 1 Pokemon, and only a handful of flying Type 1 Pokemon. So, we will filter our pokemon data into only pokemon that have type Bug, Fire, Grass, Normal, Water, or Psychic.

```
pokemon <- filter(pokemon, type_1 %in% c("Bug", "Fire", "Grass", "Normal", "Water", "Psychic"))
pokemon
```

```
## # A tibble: 458 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 Venusaur~ Grass  Poison   625    80    100     123    122    120    80
##  5      4 Charmand~ Fire   <NA>     309    39     52      43     60     50    65
##  6      5 Charmele~ Fire   <NA>     405    58     64      58     80     65    80
##  7      6 Charizard Fire   Flying   534    78     84      78    109     85   100
##  8      6 Charizar~ Fire   Dragon   634    78    130     111    130     85   100
##  9      6 Charizar~ Fire   Flying   634    78    104      78    159    115   100
## 10      7 Squirtle  Water  <NA>     314    44     48      65     50     64    43
## # ... with 448 more rows, and 2 more variables: generation <dbl>,
## #   legendary <lgl>
```

Lastly, we will convert type_1 and legendary to factors within our dataset.

```
pokemon$type_1 <- as.factor(pokemon$type_1)
pokemon$legendary <- as.factor(pokemon$legendary)
pokemon$generation <- as.factor(pokemon$generation)
pokemon
```

```
## # A tibble: 458 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 Venusaur~ Grass  Poison   625    80    100     123    122    120    80
##  5      4 Charmand~ Fire   <NA>     309    39     52      43     60     50    65
##  6      5 Charmele~ Fire   <NA>     405    58     64      58     80     65    80
##  7      6 Charizard Fire   Flying   534    78     84      78    109     85   100
##  8      6 Charizar~ Fire   Dragon   634    78    130     111    130     85   100
##  9      6 Charizar~ Fire   Flying   634    78    104      78    159    115   100
## 10      7 Squirtle  Water  <NA>     314    44     48      65     50     64    43
## # ... with 448 more rows, and 2 more variables: generation <fct>,
## #   legendary <fct>
```

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 might stratifying the folds be useful?

First, we will perform our intial split into our training and testing sets.

```
set.seed(912)
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
```

```r
dim(pokemon_test)
```

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

```r
0.7 * nrow(pokemon)
```

```
## [1] 320.6
```

```r
0.3 * nrow(pokemon)
```

```
## [1] 137.4
```

After performing a 70/30 train/test split, we see that there are 318 and 140 observations in the training data set and test data set, respectively, so it is verified that the training and testing sets have approximately the correct dimension, as 70% of the original data yields 320.6 observations, and 30% of the original data yields 137.4 observations.

```r
pokemon_folds <- vfold_cv(pokemon_train, v = 5, strata = type_1)
pokemon_folds
```

```
## #  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
```

The whole point of performing our fold cross validation is to try and continue to train our model even more. If our model is built upon the fact that type-1 is stratified, then it would be pointless for our folds to not take into account this decision.

Exercise 4: 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.

Below is our recipe.

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

pokemon_recipe
```

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

Exercise 5: 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, we'll let penalty range from -5 to 5 (it's log-scaled). How many total models will you be fitting when you fit these models to your folded data?

First, we will set up our model for multinomial regression with our Pokemon data.

```
pokemon_multreg <- multinom_reg(mixture = tune(), penalty = tune()) %>%
  set_mode("classification") %>%
  set_engine("glmnet")
```

Next, we will set up our workflow for multinomial regression with our Pokemon data model and recipe.

```
pokemon_wf <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(pokemon_multreg)
```

Lastly, we will set up our regular grid for penalty and mixture.

```
pokemon_grid <- grid_regular(penalty(range = c(-5, 5)),
                             mixture(range = c(0, 1)),
                             levels = 10)
pokemon_grid
```

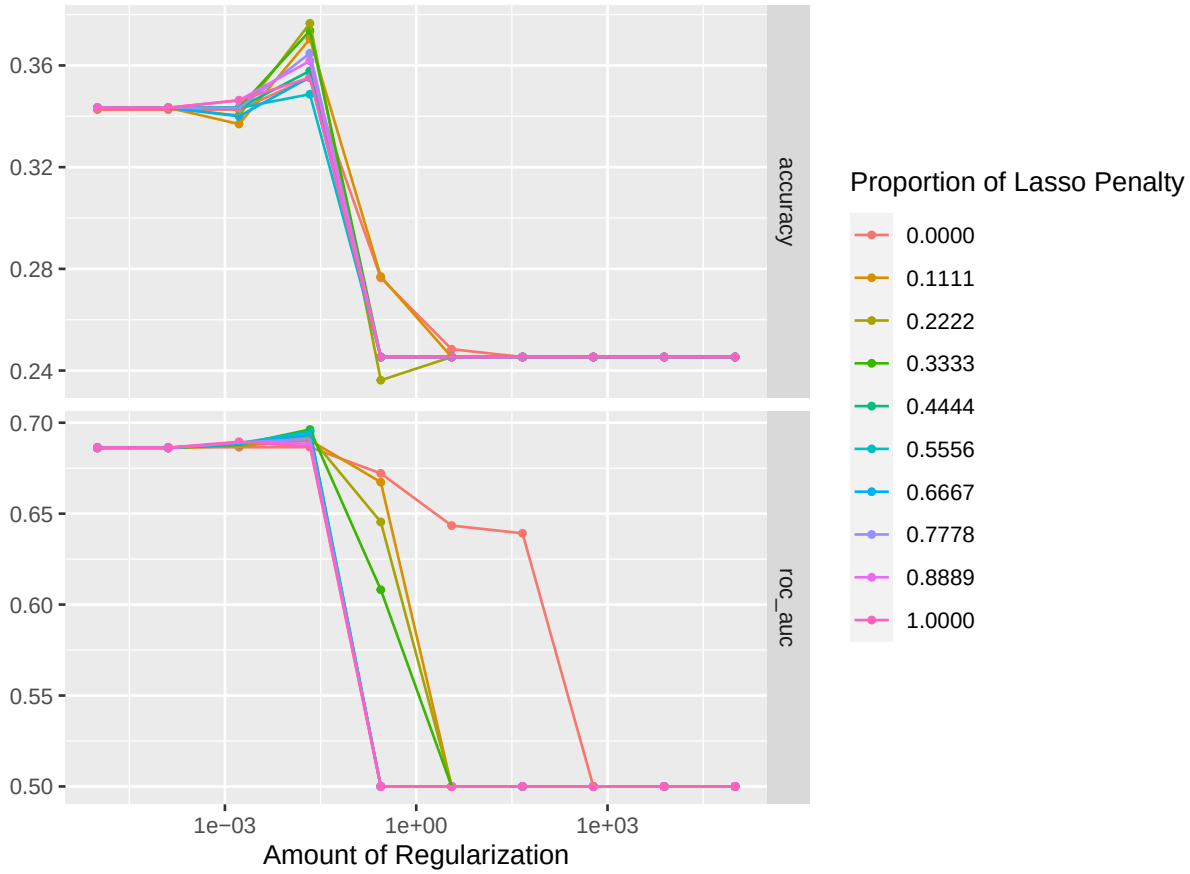
```
## # A tibble: 100 x 2
##          penalty mixture
##            <dbl>   <dbl>
##  1      0.00001        0
##  2      0.000129       0
##  3      0.00167        0
##  4      0.0215         0
##  5      0.278          0
##  6      3.59           0
##  7     46.4            0
##  8    599.             0
##  9   7743.             0
## 10 100000              0
## # ... with 90 more rows
```

We will be fitting 100 models, as there will be 10 different 10 models for each of the 10 different mixture levels.

Exercise 6: Fit the models to your folded data using tune_grid(). Use autoplot() on the results. What do you notice? Do larger or smaller values of penalty and mixture produce better accuracy and ROC AUC?

```
pokemon_tune_res <- tune_grid(pokemon_wf,
                              resamples = pokemon_folds,
                              grid = pokemon_grid)
```

```
autoplot(pokemon_tune_res)
```



Do larger or smaller values of penalty and mixture produce better accuracy and ROC AUC?

For ROC AUC, the smaller mixtures produced significantly better ROC AUC results

For accuracy, the mixture values were mostly the same at producing better accuracy, with the

```
collect_metrics(pokemon_tune_res)
```

```
## # A tibble: 200 x 8
##     penalty mixture .metric  .estimator  mean     n std_err .config
```

```
##       <dbl>   <dbl> <chr>    <chr>      <dbl> <int>   <dbl> <chr>
##  1 0.00001        0 accuracy multiclass 0.343     5 0.0293  Preprocessor1_Model~
##  2 0.00001        0 roc_auc  hand_till  0.687     5 0.0228  Preprocessor1_Model~
##  3 0.000129       0 accuracy multiclass 0.343     5 0.0293  Preprocessor1_Model~
##  4 0.000129       0 roc_auc  hand_till  0.687     5 0.0228  Preprocessor1_Model~
##  5 0.00167        0 accuracy multiclass 0.343     5 0.0293  Preprocessor1_Model~
##  6 0.00167        0 roc_auc  hand_till  0.687     5 0.0228  Preprocessor1_Model~
##  7 0.0215         0 accuracy multiclass 0.355     5 0.0307  Preprocessor1_Model~
##  8 0.0215         0 roc_auc  hand_till  0.687     5 0.0246  Preprocessor1_Model~
##  9 0.278          0 accuracy multiclass 0.276     5 0.00994 Preprocessor1_Model~
## 10 0.278          0 roc_auc  hand_till  0.672     5 0.0238  Preprocessor1_Model~
## # ... with 190 more rows
```

Exercise 7: Use select_best() to choose the model that has the optimal roc_auc. Then use finalize_workflow(), fit(), and augment() to fit the model to the training set and evaluate its performance on the testing set.

We will select our best model, which is the 34th model our of the 100 we have, and then fit this model to the training set before evaluating its performance on the testing set.

```
pokemon_best_pen <- select_best(pokemon_tune_res, metric = "roc_auc")

pokemon_multreg_final <- finalize_workflow(pokemon_wf, pokemon_best_pen)

pokemon_final_fit <- fit(pokemon_multreg_final, data = pokemon_train)

augment(pokemon_final_fit, new_data = pokemon_test) %>%
  accuracy(truth = type_1, estimate = .pred_class)
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.371
```

We did not have the most accurate estimator, unfortunately.

Exercise 8: Calculate the overall ROC AUC on the testing set. Then create plots of the different ROC curves, one per level of the outcome. Also make a heat map of the confusion matrix. What do you notice? How did your model do? Which Pokemon types is the model best at predicting, and which is it worst at? Do you have any ideas why this might be?

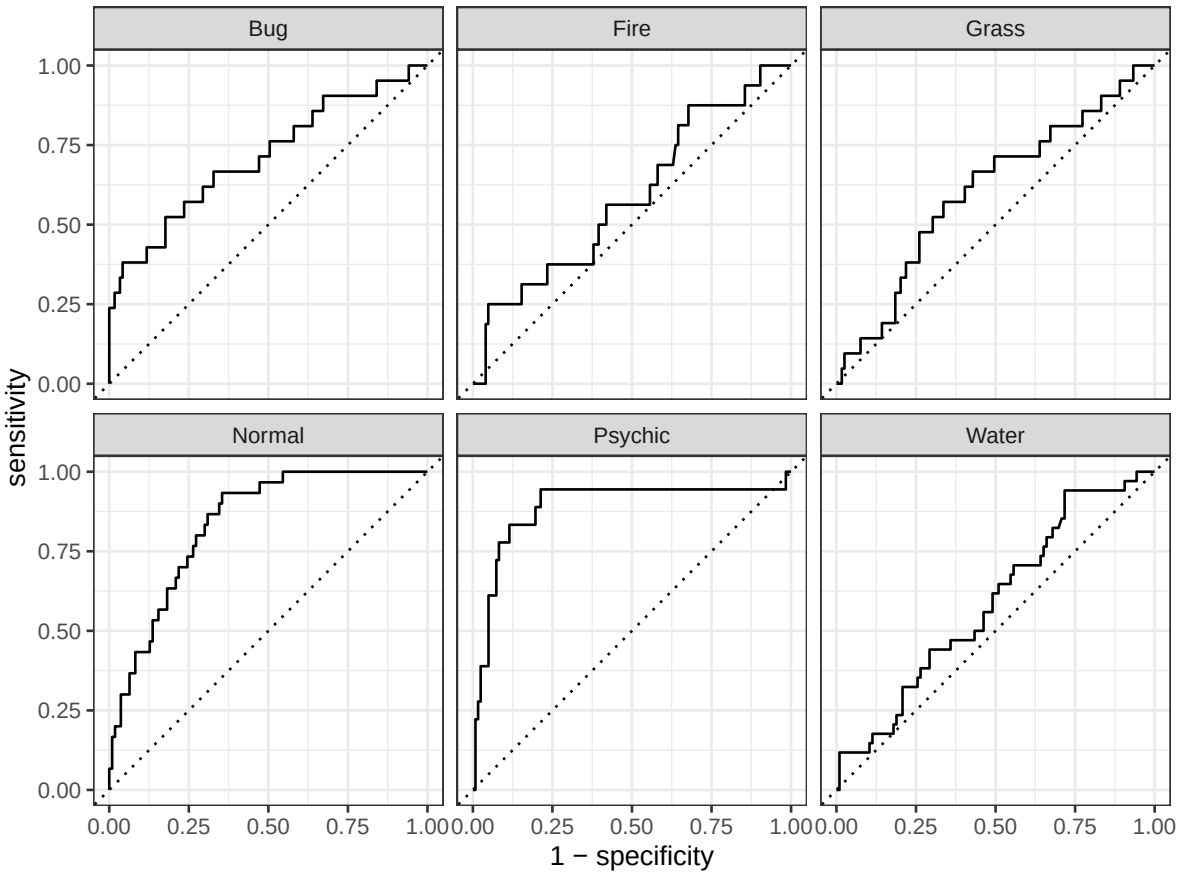
First, we will calculate the overall ROC AUC on the testing set.

```
augment(pokemon_final_fit, new_data = pokemon_test) %>%
  roc_auc(truth = type_1, estimate = c(".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      0.703
```

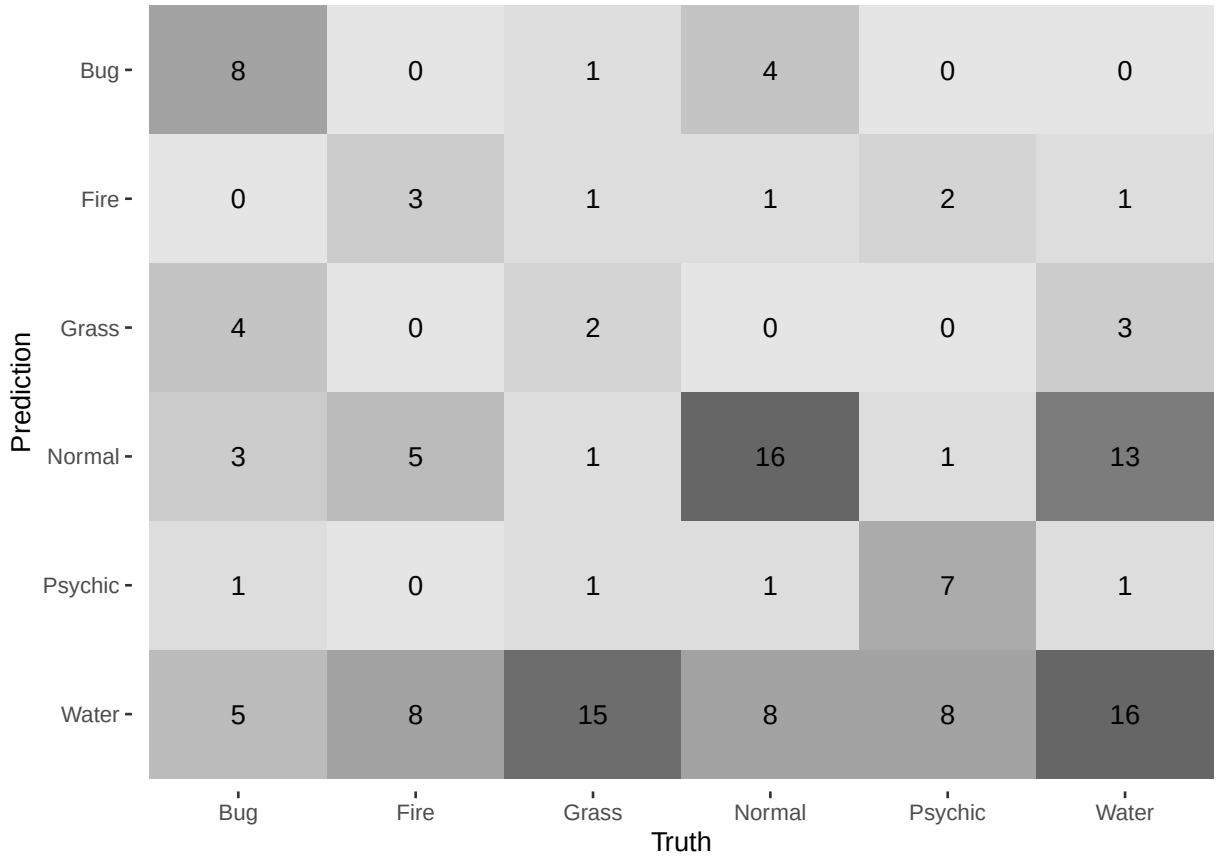
Next, we want to create 6 different ROC curves, each for the different types of Pokemon.

```
pokemon_roc_pertype <- augment(pokemon_final_fit, new_data = pokemon_test) %>%
  roc_curve(truth = type_1, estimate = c(".pred_Bug", ".pred_Fire", ".pred_Grass",
                                         ".pred_Normal", ".pred_Psychic", ".pred_Water"))

autoplot(pokemon_roc_pertype)
```



We also will make a confusion matrix of our predicted class, then do a heatmap visualization of it.

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

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



Analyzing our results, we can see that our model did not perform well, esepcially when we factor in the accuracy estimate for question 7.

Looking at the ROC curves, Normal and Psychic appear to perform a lot better than the other type predictions, with Water, Fire, and Grass all struggling according to the ROC curve.

These results are backed up by the confusion matrix, as Normal type is correctly predicted 16 times, significantly more often than it is incorrectly predicted. The opposite can be seen for the Water type predictions, as even though Water type Pokemon were correctly predicted to be Water type 16 times, their type was also incorrectly predicted 18 times.

The reason our model performed so poorly is because Water type pokemon, and even Fire and Grass type, make up a significant amount of our dataset. So, if the model is correctly predicting a significant amount of Pokemon types incorrectly majority of the time, then our model is obviously going to suffer, which it unfortunately did.

END OF HOMEWORK 5