PSTAT 131 - Homework Assignment 5

Akshat Ataliwala (7924145)

May 12, 2022

Elastic Net Tuning

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.

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

```
library(tidymodels)
library(tidyverse)
library(ggplot2)
library(corrr)
library(klaR)
library(glmnet)
library(MASS)
library(discrim)
library(poissonreg)
tidymodels_prefer()
data <- read_csv("data/pokemon.csv")
data %>% head(5)
```

```
## # A tibble: 5 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
## # ... with 3 more variables: Speed <dbl>, Generation <dbl>, Legendary <lgl>
```

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?

```
library(janitor)
```

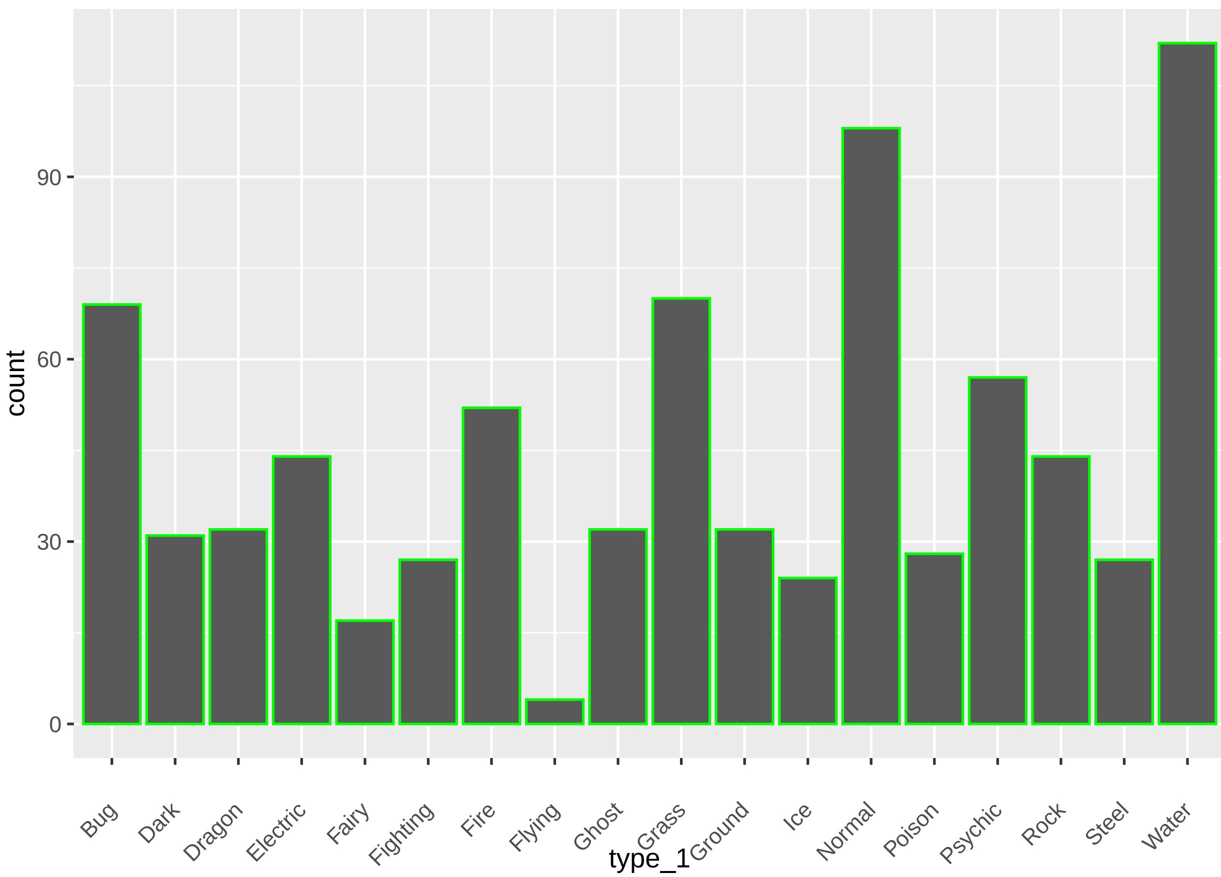
```
data <- data %>%
  clean_names()
data %>% head(5)
```

```
## # A tibble: 5 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
## # ... with 2 more variables: generation <dbl>, legendary <lgl>
```

We use clean_names() to handle problematic variable names with special characters, spaces, as well as makes everything unique to deal with repeat naming issues. In our case, we can see that the column names are now all lowercase and void of special characters and replaced with a more standard naming convention (Sp. Atk -> sp_atk, for example).

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.

```
type_1_bar <- ggplot(data, aes(x = type_1)) +
  geom_bar(color = "green") +
  theme(axis.text.x = element_text(angle = 45, vjust = 0.5, hjust=1))
type_1_bar
```



There are 18 different classes of outcomes (Pokemon types). There are some types like flying that contain very few Pokemon, and others like water and normal that make up a big proportion of the data. The rest of the types seem to fall into the 25-50 pokemon range, with some higher and some lower.

```r
# Filtering dataset to contain only pokemon whose type is Bug, Fire, Grass, Normal, Water, Psychic
data <- data %>% filter((type_1 == "Bug" | type_1 == "Fire" |
                          type_1 == "Grass" | type_1 == "Normal" |
                          type_1 == "Water" | type_1 == "Psychic"))
data %>% head(5)
```

```
## # A tibble: 5 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
## # ... with 2 more variables: generation <dbl>, legendary <lgl>
```

```r
# Converting type_1 and legendary to factors
data$type_1 <- as.factor(data$type_1)
data$generation <- as.factor(data$generation)
data$legendary <- as.factor(data$legendary)
data %>% head(5)
```

```
## # A tibble: 5 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
## # ... with 2 more variables: generation <fct>, legendary <fct>
```

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?

```r
# Setting seed
set.seed(3478)

# Stratified initial split
data_split <- initial_split(data,
                            prop = 0.8,
                            strata = type_1)

train <- training(data_split)
test <- testing(data_split)
```

```r
# Verifying that the training/test sets have the desired number of observations
dim(data)
```

```
## [1] 458  13
```

```r
0.8 * nrow(data)
```

```
## [1] 366.4
```

```r
dim(train)
```

```
## [1] 364  13
```

```r
dim(test)
```

```
## [1] 94 13
```

```r
# Stratified 5 Fold CV
folds <- vfold_cv(data = train,
                  v = 5,
                  strata = type_1)
folds
```

```
## #  5-fold cross-validation using stratification
## # A tibble: 5 x 2
##   splits           id
##   <list>           <chr>
## 1 <split [289/75]> Fold1
## 2 <split [291/73]> Fold2
## 3 <split [291/73]> Fold3
## 4 <split [292/72]> Fold4
## 5 <split [293/71]> Fold5
```

We want to stratify the folds as well as the training/testing data to ensure that the models we train on and fit to the data are representative of the true distribution. If we stratify the training set but not the cross fold validation, we essentially lose the effect of stratifying because each fold isn't taking into account the distribution of type_1, and we are using the folds to train models and find the best one.

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.

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

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?

```
# model w/ parameters to tune
pokemon_spec <- multinom_reg(penalty = tune(), mixture = tune()) %>%
  set_mode("classification") %>%
  set_engine("glmnet")
```

```
# workflow with recipe and model
pokemon_workflow <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(pokemon_spec)
```

```
# hyperparameter tuning grid
penalty_grid <- grid_regular(penalty(range = c(-5, 5)),
                             mixture(range = c(0,1)),
                             levels = 10)
penalty_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
```

Because we are tuning penalty and mixture with 10 levels each, we will be fitting 100 models in total.

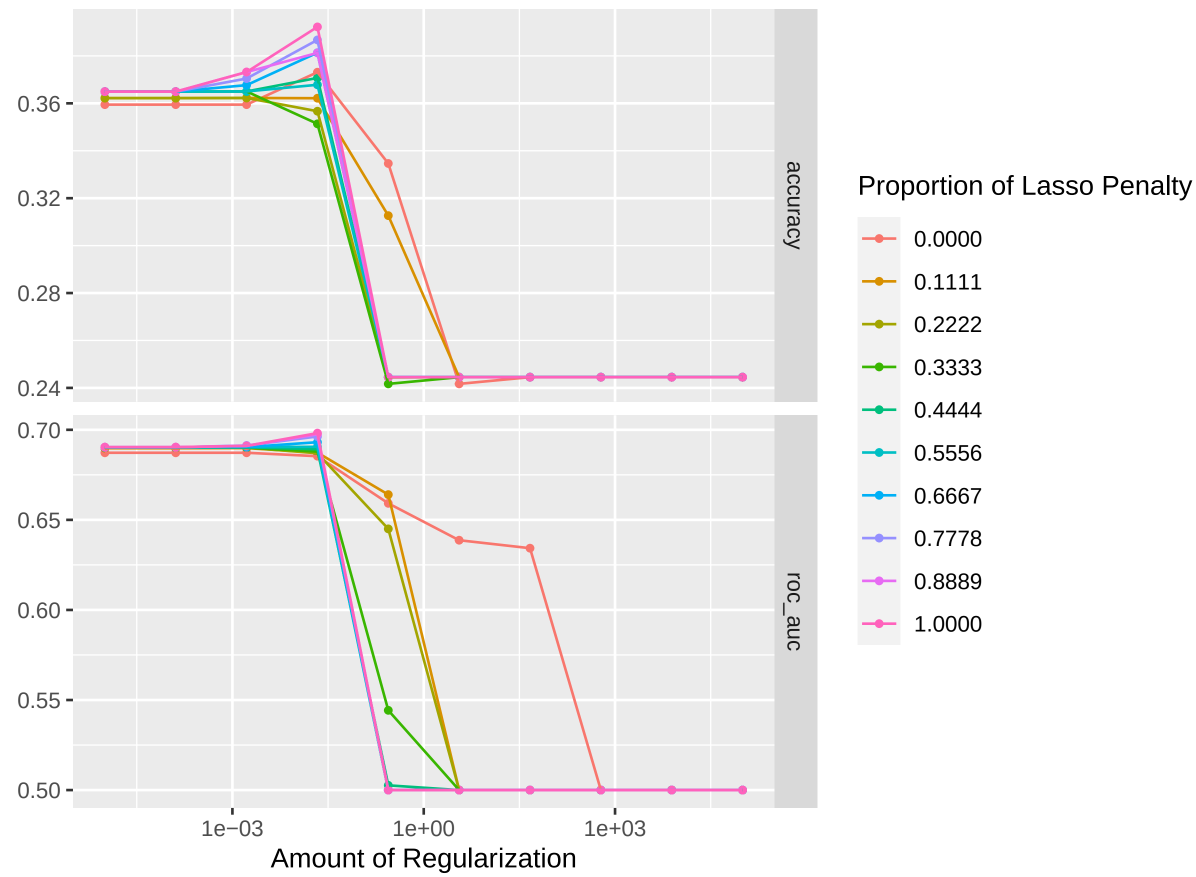
6. Fit the models to your folded data using tune_grid(). Use autoplot() on the results. What do you notice? Do larger or smaller values of penalty and mixture produce better accuracy and ROC AUC?

```
tune_res <- tune_grid(pokemon_workflow,
                      resamples = folds,
                      grid = penalty_grid)

tune_res
```

```
## # Tuning results
## # 5-fold cross-validation using stratification
## # A tibble: 5 x 4
##   splits           id    .metrics           .notes
##   <list>           <chr> <list>             <list>
## 1 <split [289/75]> Fold1 <tibble [200 x 6]> <tibble [0 x 3]>
## 2 <split [291/73]> Fold2 <tibble [200 x 6]> <tibble [0 x 3]>
## 3 <split [291/73]> Fold3 <tibble [200 x 6]> <tibble [0 x 3]>
## 4 <split [292/72]> Fold4 <tibble [200 x 6]> <tibble [0 x 3]>
## 5 <split [293/71]> Fold5 <tibble [200 x 6]> <tibble [0 x 3]>
```

```
autoplot(tune_res)
```




```
collect_metrics(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 0.359     5  0.0352 Preprocessor1_Model~
##  2 0.00001        0 roc_auc  hand_till  0.687     5  0.0230 Preprocessor1_Model~
##  3 0.000129       0 accuracy multiclass 0.359     5  0.0352 Preprocessor1_Model~
##  4 0.000129       0 roc_auc  hand_till  0.687     5  0.0230 Preprocessor1_Model~
##  5 0.00167        0 accuracy multiclass 0.359     5  0.0352 Preprocessor1_Model~
##  6 0.00167        0 roc_auc  hand_till  0.687     5  0.0230 Preprocessor1_Model~
##  7 0.0215         0 accuracy multiclass 0.373     5  0.0325 Preprocessor1_Model~
##  8 0.0215         0 roc_auc  hand_till  0.685     5  0.0239 Preprocessor1_Model~
##  9 0.278          0 accuracy multiclass 0.335     5  0.0319 Preprocessor1_Model~
## 10 0.278          0 roc_auc  hand_till  0.659     5  0.0289 Preprocessor1_Model~
## # ... with 190 more rows
```

Based on the plot, it seems that both roc_auc and accuracy start off high and slowly increase and peak at just under 1e+00 regularization, where both metrics have a steep fall off. All Lasso Penalty values tend to provide relatively similar results with perhaps values closer to 1 for lasso penalty being slightly better.

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.

```
#model selection
best_penalty <- select_best(tune_res, metric = "roc_auc")
best_penalty
```

```
## # A tibble: 1 x 3
##   penalty mixture .config
##     <dbl>   <dbl> <chr>
## 1  0.0215       1 Preprocessor1_Model094
```

```
# finalizing workflow and fitting best model on the training set
pokemon_final <- finalize_workflow(pokemon_workflow, best_penalty)

pokemon_final_fit <- fit(pokemon_final, data = train)
```

```
# evaluating best model on the test set
final_model_acc <- augment(pokemon_final_fit, new_data = test) %>%
  accuracy(truth = type_1, estimate = .pred_class)
final_model_acc
```

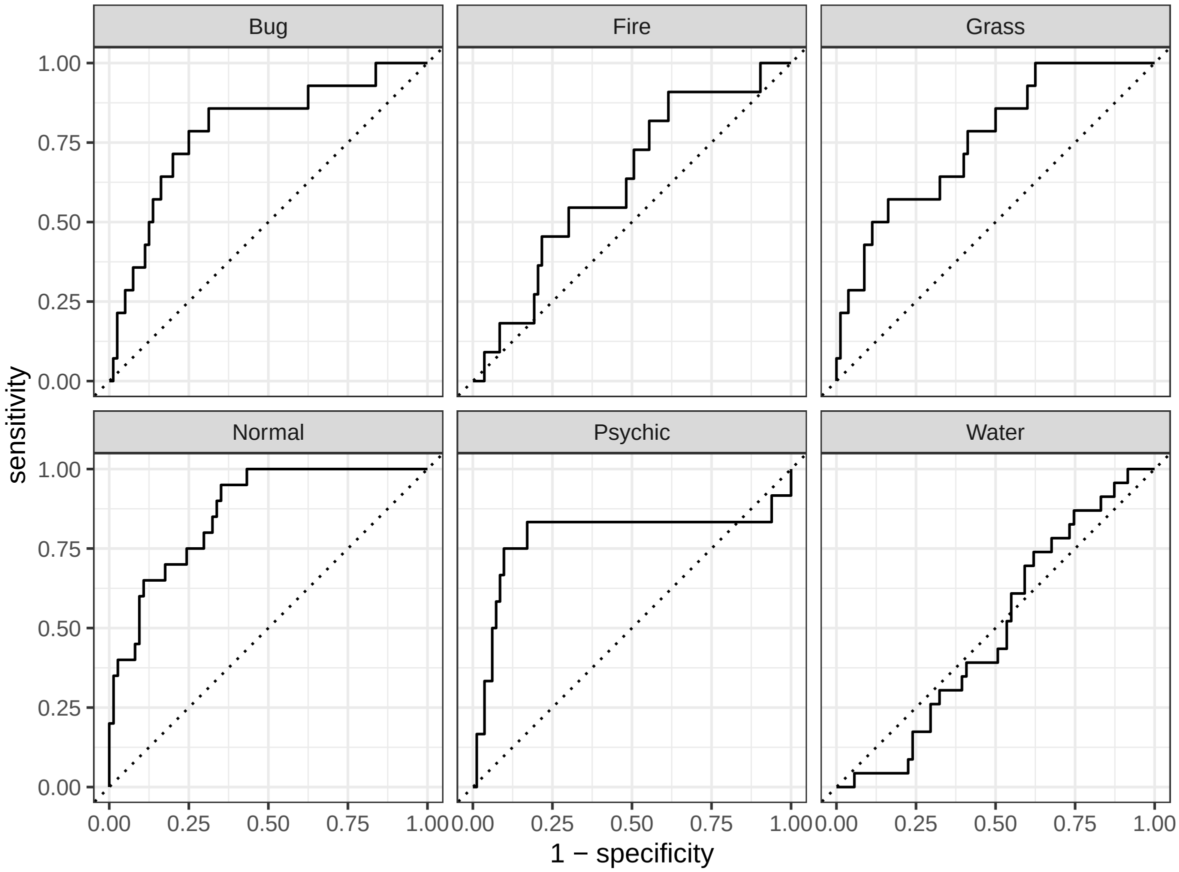
```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.330
```

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?

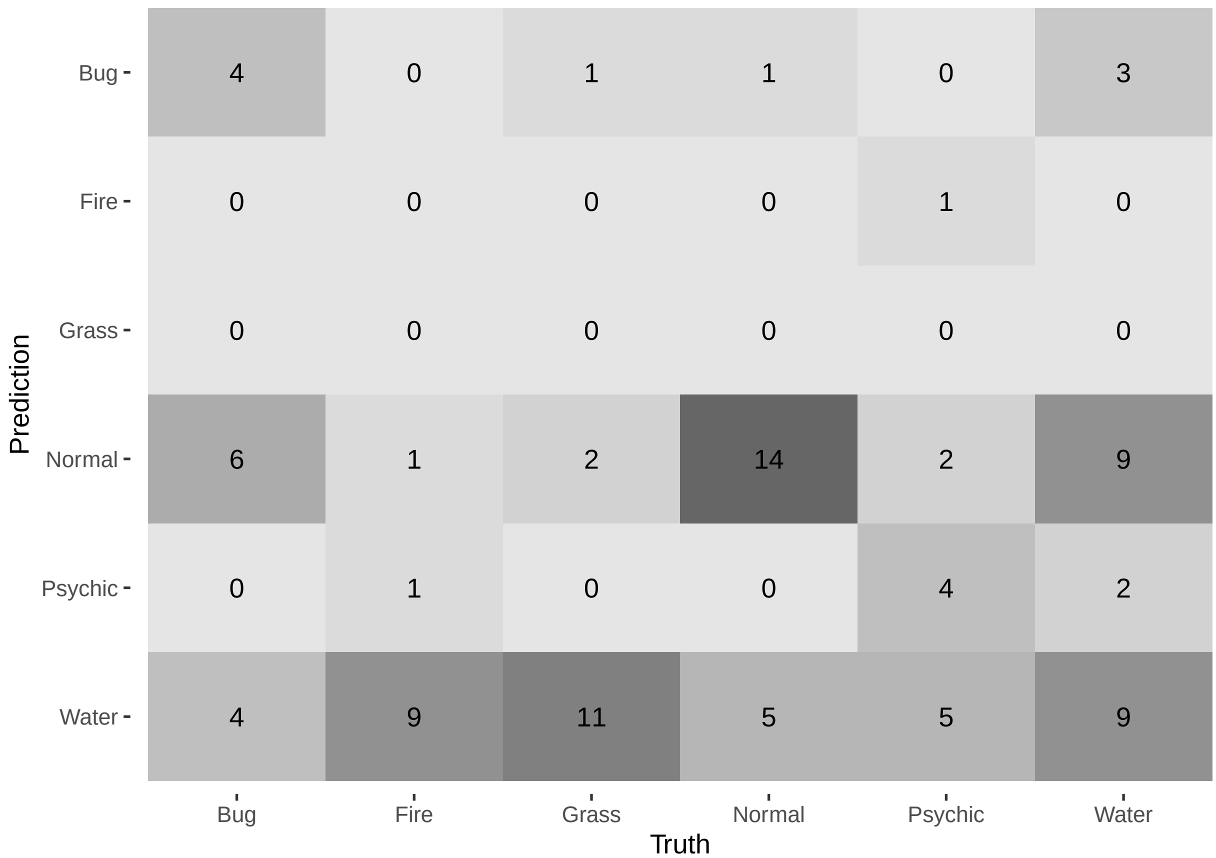
```
#overall ROC_AUC
total_roc_auc <- augment(pokemon_final_fit, new_data = test) %>%
  roc_auc(truth = type_1, estimate = c(.pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal, .pred_Psychic,
total_roc_auc
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.715
```

```
roc_curves <- augment(pokemon_final_fit, new_data = test) %>%
  roc_curve(truth = type_1, estimate = c(.pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal, .pred_Psychi
  autoplot()
roc_curves
```




```
final_model_conf <- augment(pokemon_final_fit, new_data = test) %>%
  conf_mat(truth = type_1, estimate = .pred_class) %>%
  autoplot(type = "heatmap")
final_model_conf
```



Overall, my model did pretty poorly, as it had an overall accuracy of 32% and ROC_AUC of 72%. The model seemed to be the worst at predicting water types, followed by mediocre performance on fire and grass. Depending on the specificity the model was either good at predicting psychic or extremely bad, and overall the model was the best at predicting bug and normal types.

According to the heatmap, water types were equally classified as water as they were normal, grass was overwhelmingly classified as water, as was fire. Bug was overly misclassified as normal. I think the very poor result we got can be due to the variability in the water Pokemon, as they are in the greatest abundance in the data and seem to have the most attribute variance resulting in the misclassification they are experiencing.