PSTAT 131 HW5

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`.

```
# Load in the data
load_in_pokemon <- read.csv(file = "pokemon.csv")
# Look at a few rows of the data
head(load_in_pokemon)
```

```
##   X.                  Name Type.1 Type.2 Total HP Attack Defense Sp..Atk
## 1  1             Bulbasaur  Grass Poison   318 45     49      49      65
## 2  2               Ivysaur  Grass Poison   405 60     62      63      80
## 3  3              Venusaur  Grass Poison   525 80     82      83     100
## 4  3 VenusaurMega Venusaur  Grass Poison   625 80    100     123     122
## 5  4            Charmander   Fire          309 39     52      43      60
## 6  5            Charmeleon   Fire          405 58     64      58      80
##   Sp..Def Speed Generation Legendary
## 1      65    45          1     False
## 2      80    60          1     False
## 3     100    80          1     False
## 4     120    80          1     False
## 5      50    65          1     False
## 6      65    80          1     False
```

```
# Load in needed packages
library(tidymodels)
```

```
## Warning: package 'tidymodels' was built under R version 4.0.5
```

```
## -- Attaching packages --------------------------------------- tidymodels 0.2.0 --
```

```
## v broom        0.7.12     v recipes      0.2.0
## v dials        0.1.0      v rsample      0.1.1
## v dplyr        1.0.8      v tibble       3.1.6
## v ggplot2      3.3.5      v tidyr        1.2.0
## v infer        1.0.0      v tune         0.2.0
## v modeldata    0.1.1      v workflows    0.2.6
## v parsnip      0.2.1      v workflowsets 0.2.1
## v purrr        0.3.4      v yardstick    0.0.9

## Warning: package 'broom' was built under R version 4.0.5

## Warning: package 'dials' was built under R version 4.0.5

## Warning: package 'dplyr' was built under R version 4.0.5

## Warning: package 'parsnip' was built under R version 4.0.5

## Warning: package 'recipes' was built under R version 4.0.5

## Warning: package 'tidyr' was built under R version 4.0.5

## Warning: package 'tune' was built under R version 4.0.5

## Warning: package 'workflows' was built under R version 4.0.5

## Warning: package 'workflowsets' was built under R version 4.0.5

## -- Conflicts ---------------------------------------- tidymodels_conflicts() --
## x purrr::discard() masks scales::discard()
## x dplyr::filter()  masks stats::filter()
## x dplyr::lag()     masks stats::lag()
## x recipes::step()  masks stats::step()
## * Use suppressPackageStartupMessages() to eliminate package startup messages
```

```
library(ISLR)
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v readr   2.1.2     v forcats 0.5.1
## v stringr 1.4.0

## Warning: package 'readr' was built under R version 4.0.5

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x readr::col_factor() masks scales::col_factor()
## x purrr::discard()    masks scales::discard()
## x dplyr::filter()     masks stats::filter()
## x stringr::fixed()    masks recipes::fixed()
## x dplyr::lag()        masks stats::lag()
## x readr::spec()       masks yardstick::spec()
```

```
tidymodels_prefer()

library(ggplot2)
library(dplyr)
library(glmnet)
```

```
## Loading required package: Matrix
```

```
##
## Attaching package: 'Matrix'
```

```
## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
```

```
## Loaded glmnet 4.1-3
```

```
library(pROC)
```

```
## Type 'citation("pROC")' for a citation.
```

```
# set seed
set.seed(27)
```

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?

```
# Install and load the 'janitor' package
library(janitor)

# Use clean_names() function on the pokemon data
# save results to work with for rest of assignment
pokemon <- clean_names(load_in_pokemon)
head(pokemon)
```

```
##   x                  name type_1 type_2 total hp attack defense sp_atk sp_def
## 1 1             Bulbasaur  Grass Poison   318 45     49      49     65     65
## 2 2               Ivysaur  Grass Poison   405 60     62      63     80     80
## 3 3              Venusaur  Grass Poison   525 80     82      83    100    100
## 4 3 VenusaurMega Venusaur  Grass Poison   625 80    100     123    122    120
## 5 4            Charmander   Fire          309 39     52      43     60     50
## 6 5            Charmeleon   Fire          405 58     64      58     80     65
##   speed generation legendary
## 1    45          1     False
## 2    60          1     False
## 3    80          1     False
## 4    80          1     False
## 5    65          1     False
## 6    80          1     False
```

Answer(Exercise 1): After installing and loading the 'janitor' package and applying the function clean_names() on the Pokemon data, the variable names have changed (ie. Type.1 changed to type_1). The resulting names are now unique, all lowercase and consist only of the '_' (underscore) character, numbers and letters.

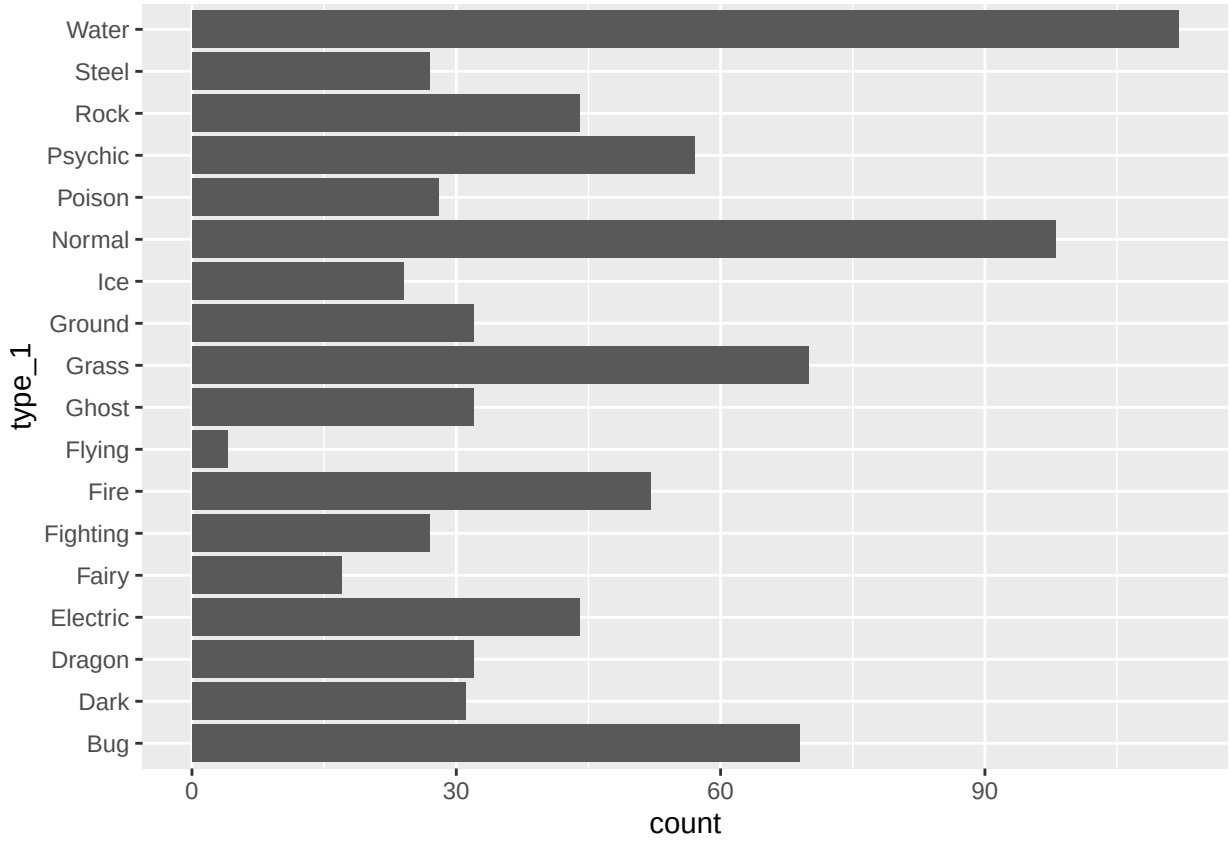
I think the function clean_names() is useful in terms of readability and reproducibility. For me (the original coder), it's easier for me to keep track and remember the variables I have. For others, they can read the code, and follow along without having to keep looking at the codebook. Overall, the function yields a big benefit with relatively low effort.

(Reference: https://www.rdocumentation.org/packages/janitor/versions/1.2.0/topics/clean_names)

Exercise 2

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

```
# use ggplot to create bar chart of outcome variable type_1
pokemon %>%
  ggplot(aes(y=type_1)) +
  geom_bar()
```



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

Answer(Exercise 2): There are 18 classes of the outcome variable type_1. There are very few of flying pokemon type; a small amount of fairy and ice pokemon types.

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.

```
# Use the filter function and grepl() function to filter rows that contains a certain string
pokemon <- pokemon %>% filter(grepl("Bug|Fire|Grass|Normal|Water|Psychic", type_1))

# Verify that we have filtered the entire data set to
# contain only Pokémon whose `type_1` is Bug, Fire,
# Grass, Normal, Water, or Psychic.

# use table function
table(pokemon$type_1)
```

```
##
##     Bug    Fire   Grass  Normal Psychic   Water
##      69      52      70      98      57     112
```

After filtering, convert `type_1` and `legendary` to factors.

```
# use function factor() to convert `type_1` and `legendary` to factors
pokemon$type_1 <- factor(pokemon$type_1)
# verify that type_1 is now a factor
class(pokemon$type_1)
```

```
## [1] "factor"
```

```
pokemon$legendary <- factor(pokemon$legendary)
# verify that type_1 is now a factor
class(pokemon$legendary)
```

```
## [1] "factor"
```

```
# (Per TA's recommendation) Here we also want to make "generation" a factor -
# This is so that #4 does not throw an error
pokemon$generation <- factor(pokemon$generation)
```

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.

```
# Initial split, stratifying on outcome variable type_1
pokemon_split <- initial_split(pokemon, prop = 0.80,
                               strata="type_1")
pokemon_split
```

```
## <Analysis/Assess/Total>
## <364/94/458>
```

```r
# training
pokemon_train <- training(pokemon_split)
# testing
pokemon_test <- testing(pokemon_split)

# Verifying that training and test set shave the desired number of obs
dim(pokemon_train);dim(pokemon_test)
```

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

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

Answer(Exercise 3): Note for splitting the data we will be using a 80/20 percentage split (80% for training, 20% for test). In addition, when we use the dim() function, we can see that the 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
# use the vfold_cv() function, k/v = 5
# use the strata = type_1 argument
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 [289/75]> Fold1
## 2 <split [291/73]> Fold2
## 3 <split [291/73]> Fold3
## 4 <split [292/72]> Fold4
## 5 <split [293/71]> Fold5
```

Answer(Exercise 3): Stratifying the folds might be useful because it allows for the same class ratio seen in the original dataset (that also has been stratified) to be preserved throughout the k folds.

(ie. Allows us to rearrange the data in a way where each fold has a good representation of the whole dataset)

This method is also useful if the data is imbalanced or if the dataset size is smaller.

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.

```r
# set up recipe using recipe() function, with `legendary`,
#`generation`, `sp_atk`, `attack`, `speed`, `defense`, `hp`, and `sp_def
pokemon_recipe <- recipe(type_1 ~ legendary + generation + sp_atk +
                           attack + speed + defense + hp
                         + sp_def, data = pokemon_train) %>%
  # step_dummy to dummy code legendary and generation
  step_dummy(all_nominal_predictors())%>%
  # center and scale all predictors
  step_normalize(all_predictors())
pokemon_recipe
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          8
##
## Operations:
##
## Dummy variables from all_nominal_predictors()
## 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?

Answer(Exercise 5): We will be fitting a total of 500 models when we fit these models to the folded data (100 models folded 5 times).

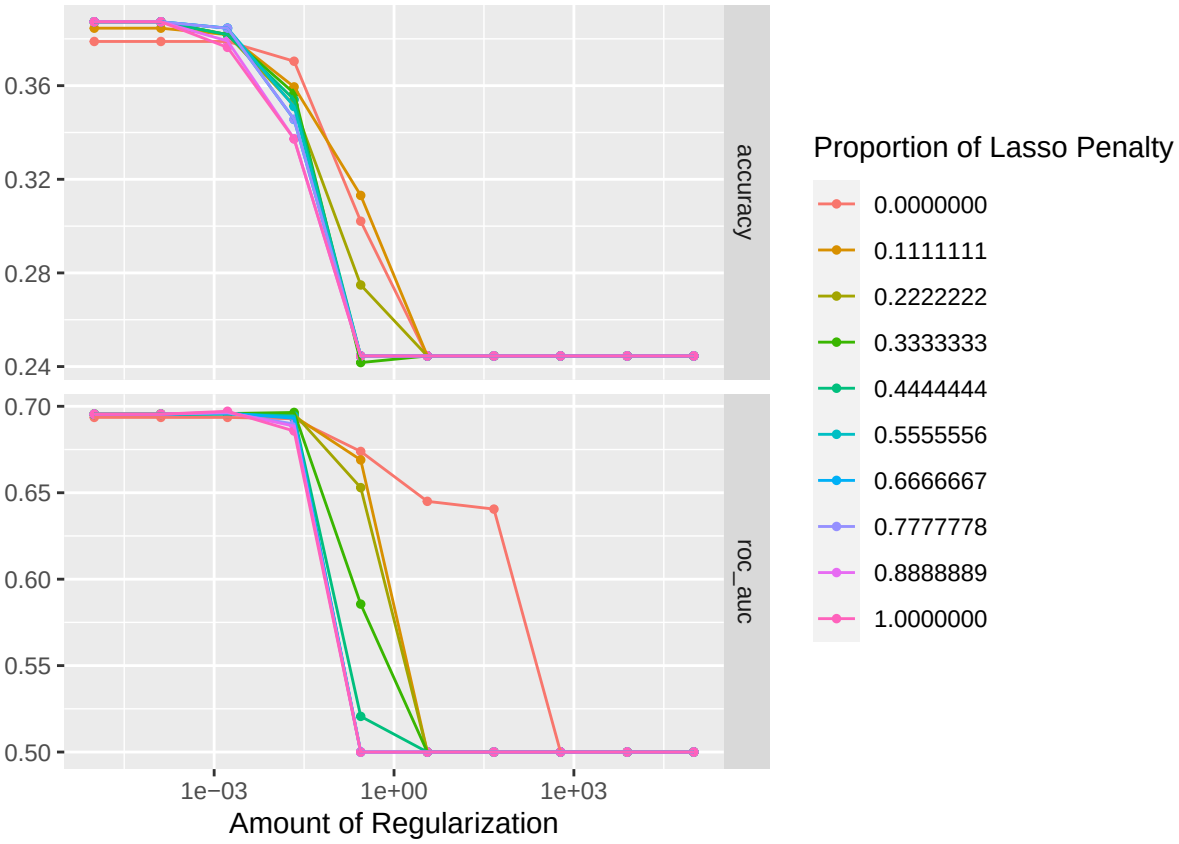
```r
# regular grid for penalty and mixture
# mixture range from 0 to 1
# penalty range from -5 to 5
grid <- grid_regular(mixture(range = c(0, 1)), penalty(range = c(-5, 5)), levels = 10)

# set up the model for the elastic net
multinom_reg_spec <-
  # set penalty and mixture to tune()
  multinom_reg(penalty = tune(), mixture = tune()) %>%
  set_mode("classification") %>%
  set_engine("glmnet")

# set up the work flow for the elastic net s
multinom_reg_workflow <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(multinom_reg_spec)
```

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?

```
# fit the models to folded data using function tune_grid()
tune_res <- tune_grid(
  multinom_reg_workflow,
  resamples = pokemon_folds,
  grid =  grid
)

# use function autoplot() on the results
autoplot(tune_res)
```



Answer(Exercise 6): After using the function autoplot() on the results, we noticed that in both plots, a mixture of 0.000000 tends to drop slower compared to the other values. In addition as the penalty nears 1e+00 the model starts to do very poorly. This leads us to conclude that smaller values of penalty produces better accuracy and ROC AUC. Though it is hard to tell, it seems like larger values of mixture produces better accuracy and ROC AUC (pink denoting 1.00000 seems to do the best while the red denoting 0.00000 seems to do the worst).

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.

```
# Use function select_best() to choose the model that has the optimal roc_auc
best_model <- select_best(tune_res, metric = "roc_auc")
best_model
```

```
## # A tibble: 1 x 3
##   penalty mixture .config
##     <dbl>   <dbl> <chr>
## 1 0.00167       1 Preprocessor1_Model093
```

```
# use `finalize_workflow()`, `fit()`, and `augment()` to fit the model to
# the training set and evaluate its performance on the testing set
final_wf <- finalize_workflow(multinom_reg_workflow, best_model)
final_fit <- fit(final_wf, pokemon_train)
# predictions for each observation
predicted_data <- augment(final_fit, new_data = pokemon_test)
```

Answer(Exercise 7): Using the select_best() function, the model that has the optimal roc_auc has a penalty of 0.001668101 and a mixture of 1.

We then used the functions finalize_workflow(), fit(), and augment() to fit the model to the training set and evaluate its performance on the testing set. This will be further evaluated in Exercise 8.

Exercise 8

Calculate the overall ROC AUC on the testing set.

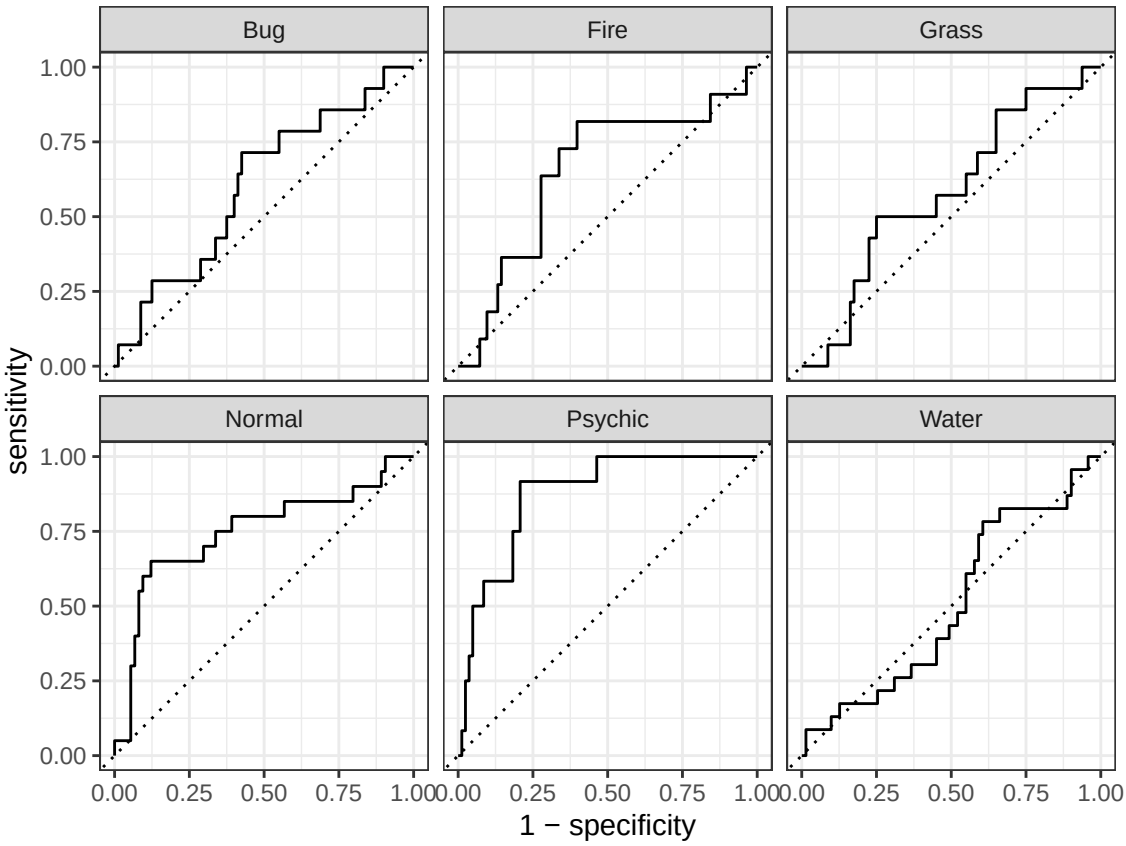
```
# use the functions augment() and roc_auc() to calculate the overall ROC AUC
# on the testing set
predicted_data <- augment(final_fit, new_data = pokemon_test) %>%
  select(type_1, starts_with(".pred"))

predicted_data%>%roc_auc(type_1,.pred_Bug:.pred_Water)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.657
```

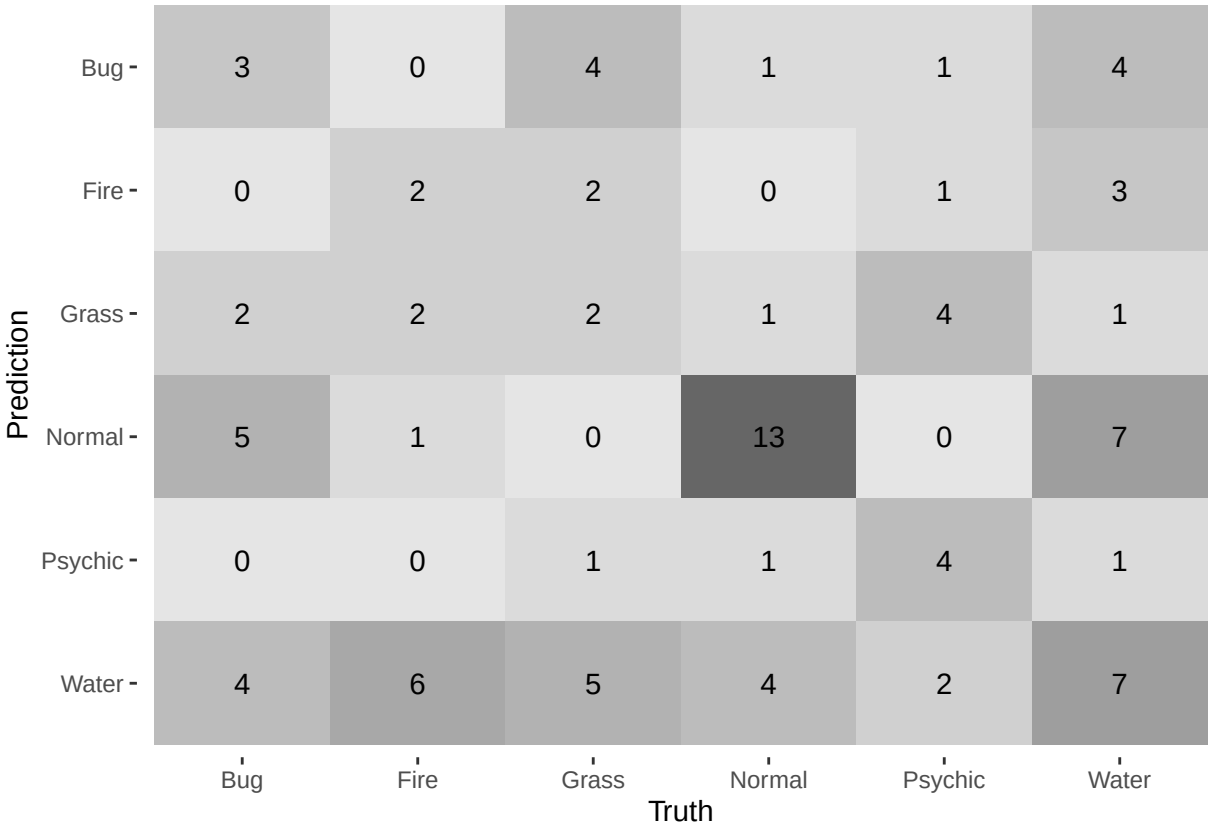
Answer(Exercise 8): After using the functions augment() and roc_auc(), we find that the overall ROC AUC on the testing set yields an estimate of 0.6569175.

Then create plots of the different ROC curves, one per level of the outcome. Also make a heat map of the confusion matrix.

```
# use function roc_curve() function to create different ROC curves,
# one per level of the outcome
predicted_data%>%roc_curve(type_1,.pred_Bug:.pred_Water)%>%
  autoplot()
```




```
# use conf_map() and autoplot() to make a heat map of the confusion matrix
predicted_data%>%
  conf_mat(truth=type_1,estimate=.pred_class)%>%
  autoplot(type="heatmap")
```



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?

Answer(Exercise 8): There are a lot of interesting observations we can make on the model. Overall, I don't think my model did very well. This was slightly hinted in Exercise 8; we got an overall ROC AUC estimate of 0.6569175 (though there is no specific threshold, the range 0.5-0.7 usually equates to poor discrimination).

In the plots of the different ROC curves, the model seems to be the best at predicting psychic and normal. The model seems to be worst at predicting water, grass, bug and fire.

To see what might be going on, we can take a look at the heat map of the confusion matrix. We can see that normal and water types are predicted the most while fire and grass don't get predicted at all (the other types also have small amounts). It seems like the error is that regardless of the truth, if the model doesn't know, it will just classify the type as water or normal.