# Homework 5

PSTAT 131/231

## Contents



### 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.

Figure 1: Fig 1. Vulpix, a Fire-type fox Pokémon from Generation 1.

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`.

```
# packages
library(janitor)
library(tidymodels)
library(tidyverse)
library(data.table)
```

```r
# read in data
pokemon <- read.csv("data/Pokemon.csv")
```

### 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
head(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
```

```r
head(clean_names(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
```

```r
pokemon <- clean_names(pokemon)
```

`clean_names()` altered the ugly column names the `read.csv()` generated by default. It replaces all sequences of periods that separated alphanumeric characters with a single underscore, stripped any trailing periods, and it also made all uppercase letters lowercase. This makes it much easier to read and type the names while working throughout the project.

### Exercise 2

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

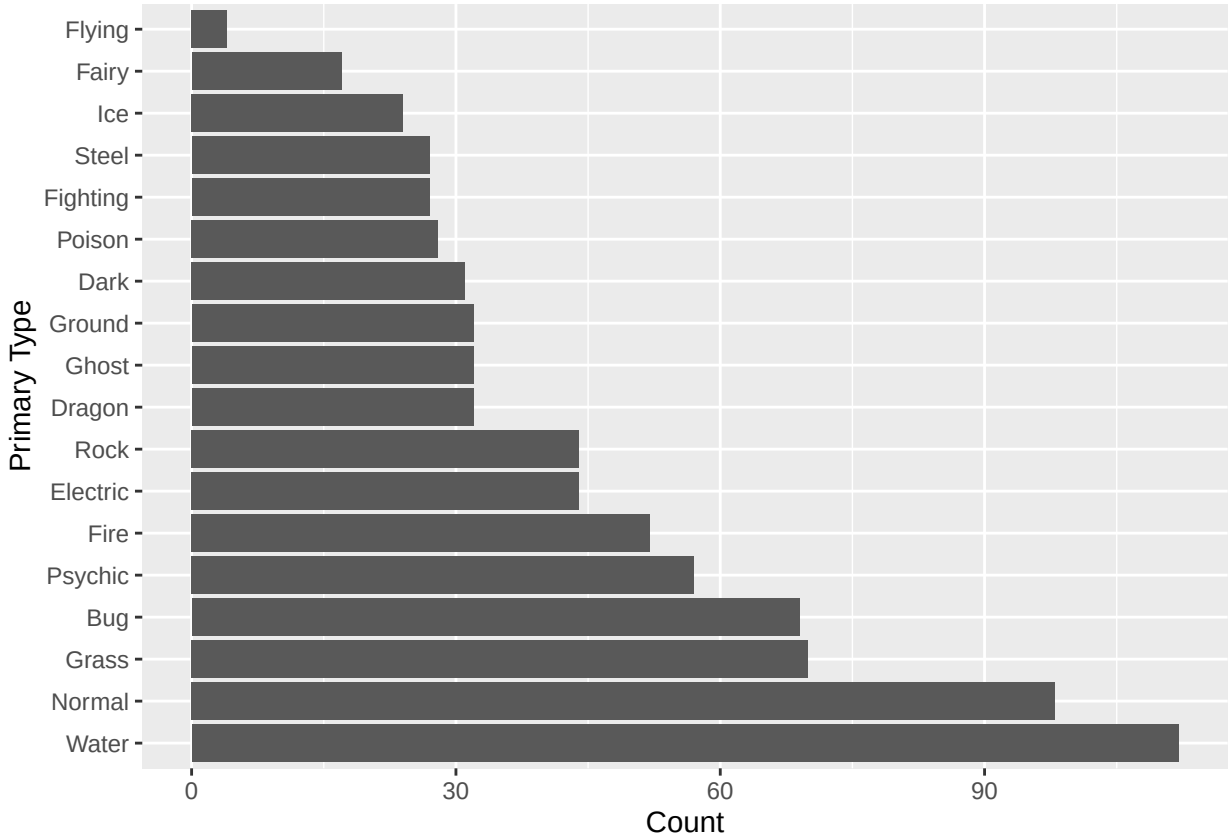
How many classes of the outcome are there? Are there any Pokémon types with very few Pokémon? If so, which ones?

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.

```
p<-ggplot(data=pokemon, aes(x=fct_infreq(type_1))) +
  geom_bar(stat="count")

p + coord_flip() + labs(x = "Primary Type", y = "Count")
```




Pokemon have 18 different types. The most notable type associated with few Pokemon is Flying type. In reality, there are tons of Flying type Pokemon, but Tornadus is the only Pokemon whose primary type is Flying. Flying is the most common secondary type among Pokemon that have a secondary type.

```
# filter the pokemon types to make life easier
mons <- pokemon%>%
  filter(
      type_1=="Bug"|
      type_1=="Fire"|
      type_1=="Grass"|
      type_1=="Normal"|
      type_1=="Water"|
      type_1=="Psychic"
    )

# convert `type_1` and `legendary` to a factor
mons$type_1 <- as.factor(mons$type_1)
mons$legendary <- as.factor(mons$legendary)
```

### 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?

```
set.seed(42069) # for reproducibiility

# get training and testing data
pokemon_split <- initial_split(mons,prop=0.8,
                               strata=type_1)

pokemon_train <- training(pokemon_split)
pokemon_test <- testing(pokemon_split)

#verification
dim(pokemon_train)
```

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

```
dim(pokemon_test)
```

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

```
# cross-validation folds
pokemon_folds <- vfold_cv(pokemon_train, v=5, strata=type_1)
```

Stratifying the folds is a good idea for the same reason stratifying a training and test set is a good idea. We are using the folds to fit and evaluate the model, so it makes sense that the folds are stratified so that the outcome levels are represented properly when training the model.

### 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.

```
# create recipe to predict `type_1`
pokemon_recipe <- recipe(type_1 ~ ., data = mons%>%dplyr::select(type_1,hp:legendary))%>%
  step_dummy(c("legendary", "generation"))%>%
  step_center(all_predictors())%>%
  step_scale(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?

```
# the model
pokemon_spec <- multinom_reg(engine="glmnet",
                             penalty=tune(),
```

```
                              mixture=tune())

# set up workflow
pokemon_workflow <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(pokemon_spec)

# grid of parameter values to use to evaluate the model with
param_grid <- grid_regular(mixture(),penalty(range=c(-5,5)), levels=10)
param_grid
```

```
## # A tibble: 100 x 2
##    mixture penalty
##      <dbl>   <dbl>
##  1   0     0.00001
##  2   0.111 0.00001
##  3   0.222 0.00001
##  4   0.333 0.00001
##  5   0.444 0.00001
##  6   0.556 0.00001
##  7   0.667 0.00001
##  8   0.778 0.00001
##  9   0.889 0.00001
## 10   1     0.00001
## # ... with 90 more rows
```

In total, we will be fitting 500 models when we fit these models to our folded data. There are 10 levels to `penalty` and 10 levels to `mixture`, so there are 100 paramater pairs to fit the models with. Furthermore, in 5-fold cross validation, we will be training each model to each subset of 4 folds. Therefore, we are fitting each model with a given parameter pair ${}_5C_4 = 5$ times. Thus, 500 models.

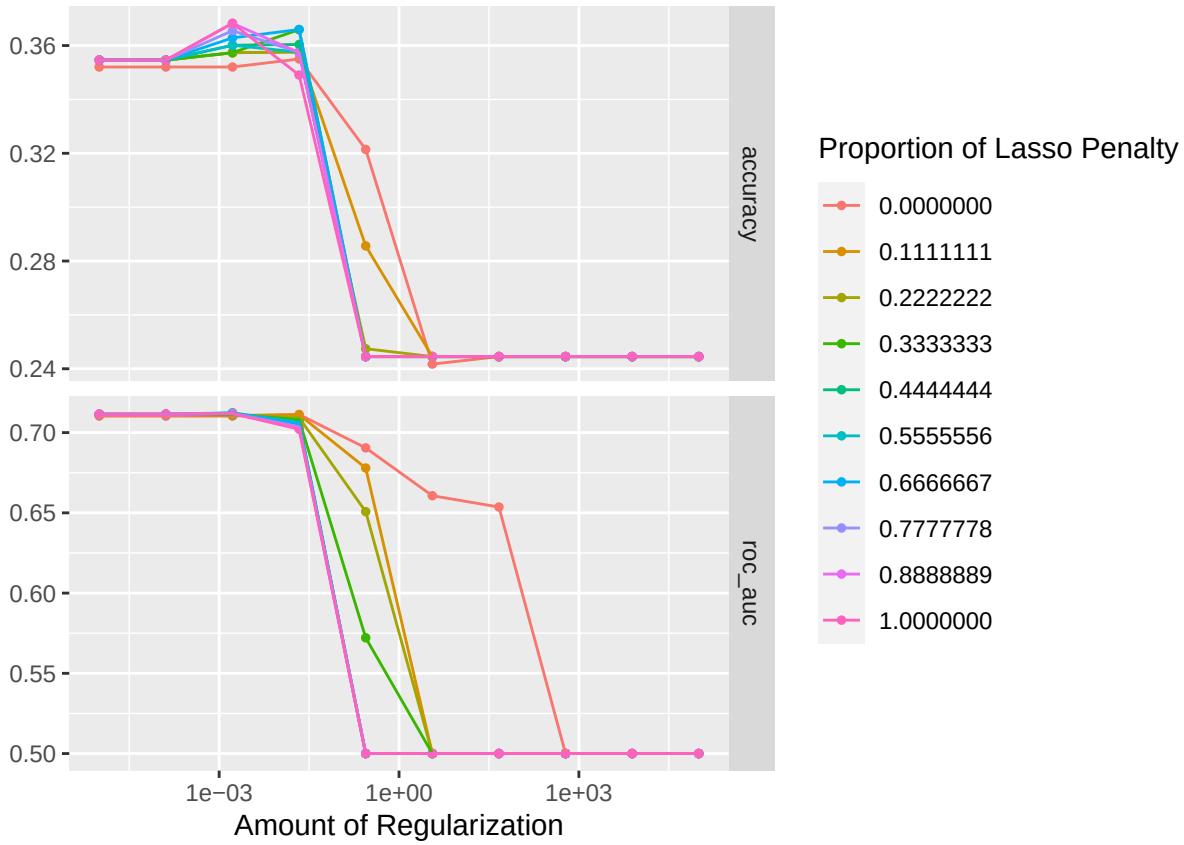
### Exercise 6

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?

```
tune_res <- tune_grid(
  pokemon_workflow,
  resamples = pokemon_folds,
  grid = param_grid
)

autoplot(tune_res)
```




It definitely seems like smaller `penalty` result in better accuracy and ROC AUC. For small and large values of `penalty`, the value of `mixture` is largely irrelevant. However, for midrange values of `penalty`, a small value of `mixture` has much better ROC AUC and significantly better accuracy.

### 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.

```
best_params <- select_best(tune_res, metric = "roc_auc")
best_params
```

```
## # A tibble: 1 x 3
##   penalty mixture .config
##     <dbl>   <dbl> <chr>
## 1 0.00167   0.556 Preprocessor1_Model053
```

```
pokemon_final <- finalize_workflow(pokemon_workflow, best_params)

pokemon_final_fit <- fit(pokemon_final, data = pokemon_train)

# quantified accuracy
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     0.479
```

**Exercise 8**

Calculate the overall ROC AUC on the testing set.

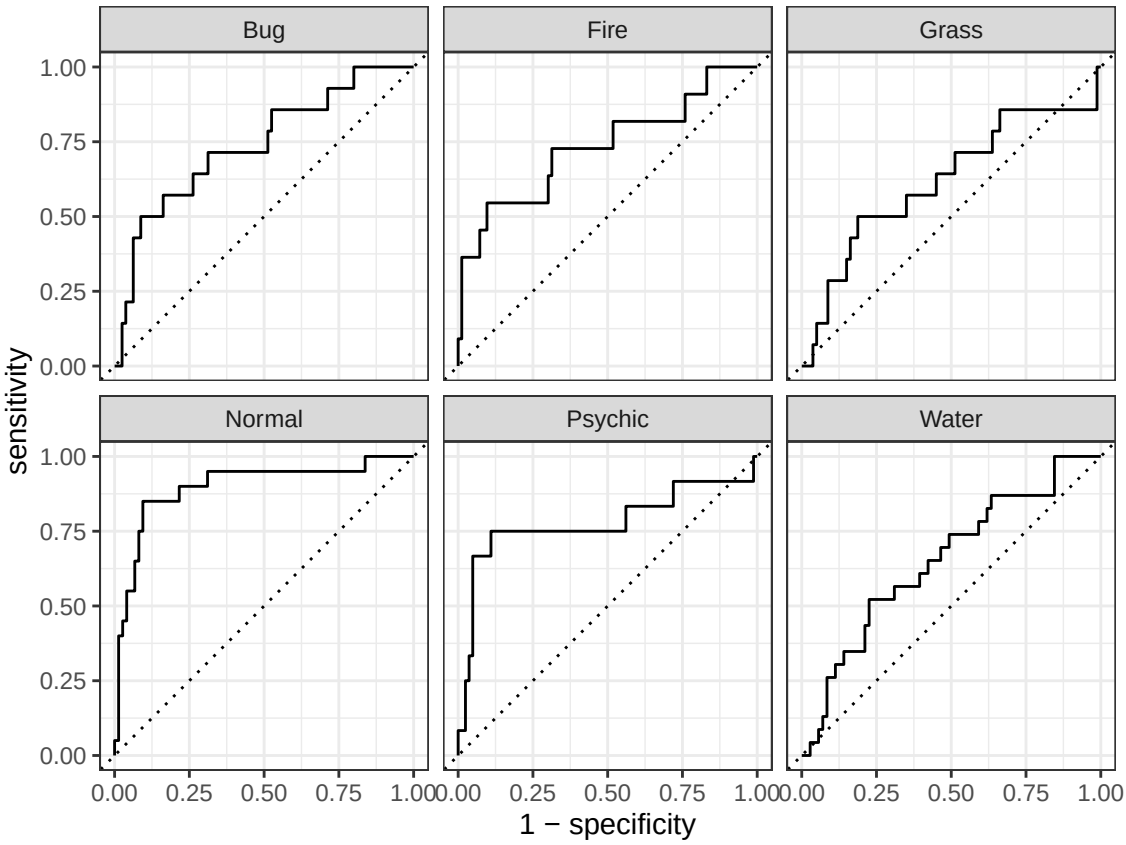
Then create plots of the different ROC curves, one per level of the outcome. Also make a heat map of the confusion matrix.

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?

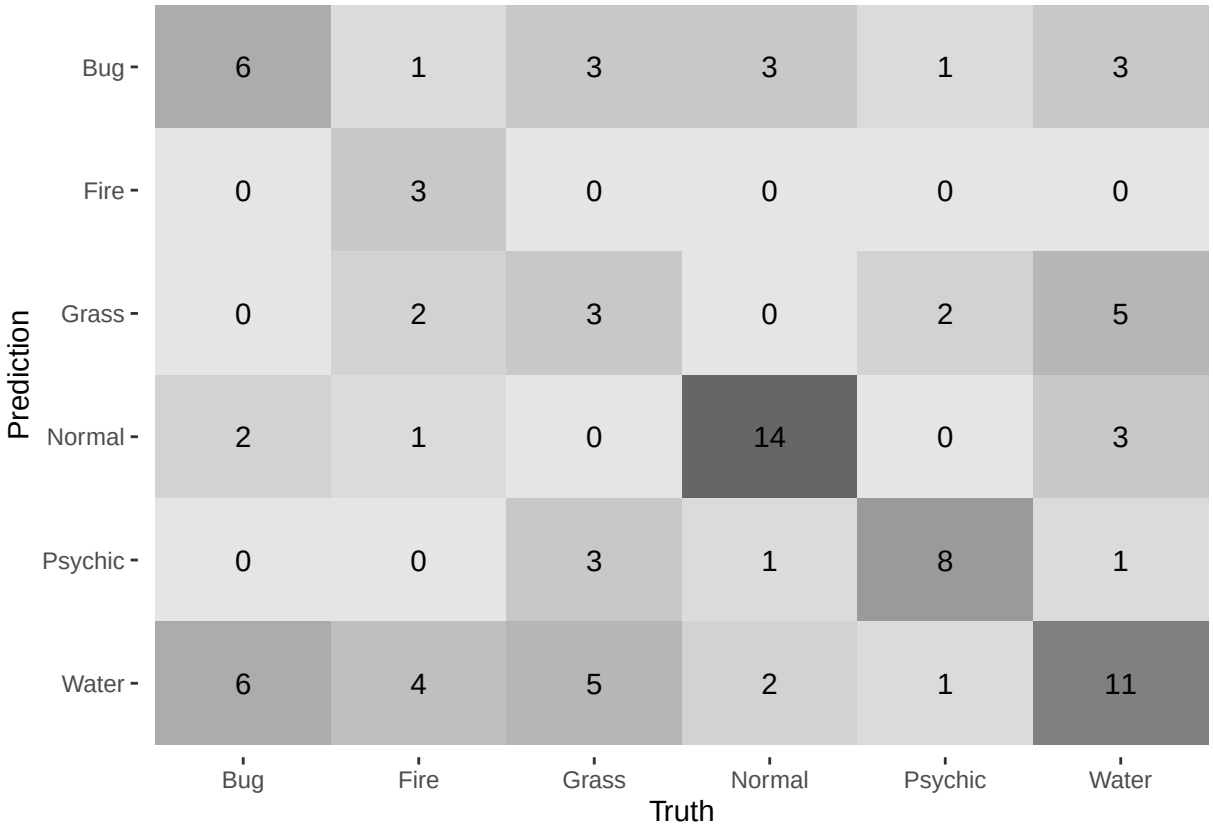
```
# calculate the overall roc auc
augment(pokemon_final_fit, new_data = pokemon_test) %>%
  roc_auc(truth = type_1, estimate = .pred_Bug:.pred_Water)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.734
```

```
# roc curves for the different factor levels
augment(pokemon_final_fit, new_data = pokemon_test) %>%
  roc_curve(truth = type_1, estimate = .pred_Bug:.pred_Water) %>%
  autoplot()
```




```
# visualization of the accuracy
augment(pokemon_final_fit, new_data = pokemon_test) %>%
  conf_mat(truth = type_1, estimate = .pred_class) %>%
  autoplot(type="heatmap")
```




```
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     0.479
```

We can see from the confusion matrix that Normal types stand out as the "easiest" type to classify (classified correctly: 75%), while Grass type seem to be the hardest (classified correctly: 21.4%). The first hypothesis may be that there are simply more Normal types than any other type.

```
# counts for the the 6 types in our data set
table(mons$type_1)
```

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

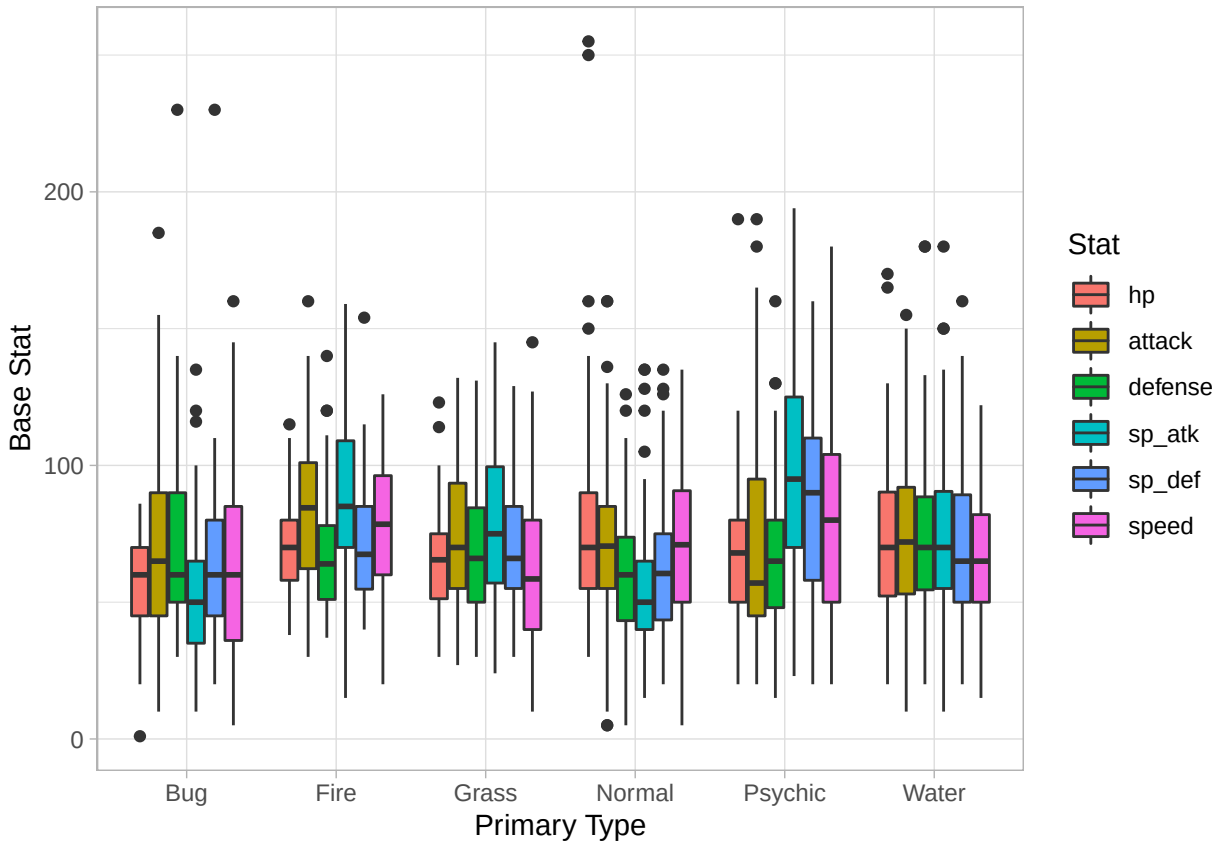
However, a Pokemon enthusiat such as myself doesn't need the line of code above to recite the fact that Water Pokemon are the most common type in the franchise, and our model didn't do so well at classifying Water types (47.8%). Furthermore, there are relatively few Psychic type Pokemon in our training set, but Psychic was the next best type for classification (66.7%).

The next idea to explore is the breakdown of stats between the different types. The bulk of our predictors were just the base stats of the Pokemon used in the main-line video game series. We can construct a boxplot of the stats as follows

```
# create box plot summarizing base stats for different types
mons_dt <- data.table::setDT(copy(mons))
mons_plot <- melt(mons_dt,
```

```
                  measure.vars =
                    c("hp","attack","defense","sp_atk","sp_def","speed"),
                  value.name = "Base.Stat",
                  variable.name = "Stat")

ggplot(mons_plot, aes(x = type_1,y = Base.Stat,fill = Stat))+
  geom_boxplot()+
  labs(x="Primary Type", y="Base Stat")+
  theme_light()
```




This plot, along with the confusion matrix, gives us a little bit of insight. There's a few noticable patterns in the base stats of Pokemon given their type. The most obvious is how well-rounded Water type Pokemon are. They seem to have pretty equal stats across the board for the most part.

Arguably the next most "well-rounded" type is Grass; the type that was misclassified most frequently. On top of that, from our confusion matrix, we see that Grass Pokemon were misclassified as Water types most frequently. This could be because Water types are the most abundant in the data set, so well-rounded Grass types get classified as Water types (note that this holds vice versa as Grass type is the most common misclassification for Water types as well).

Notice that Psychic types also have a pretty characteristic stat distribution: very low `hp`, `attack`, and `defense` with very high `sp_atk` and decently high `sp_def` and `speed`. Furthermore, there is this cascading waterfall created by `sp_atk`, `sp_def`, and `speed`, and we see this same pattern in Grass types. Grass type's second most misclassified type is Psychic and vice versa.

I am not too sure why Normal types get classified so accurately; perhaps it is because of their consistently low `sp_atk` which most types expect Bug don't share. It is noteworthy that Bug types have a roughly well-rounded stat spread (like Water), but the also tend to have quite low `sp_atk`(like Normal), and Bugs

were only missclassified as Water and Normal types.

## For 231 Students

### Exercise 9

In the 2020-2021 season, Stephen Curry, an NBA basketball player, made 337 out of 801 three point shot attempts (42.1%). Use bootstrap resampling on a sequence of 337 1's (makes) and 464 0's (misses). For each bootstrap sample, compute and save the sample mean (e.g. bootstrap FG% for the player). Use 1000 bootstrap samples to plot a histogram of those values. Compute the 99% bootstrap confidence interval for Stephen Curry's "true" end-of-season FG% using the quantile function in R. Print the endpoints of this interval.

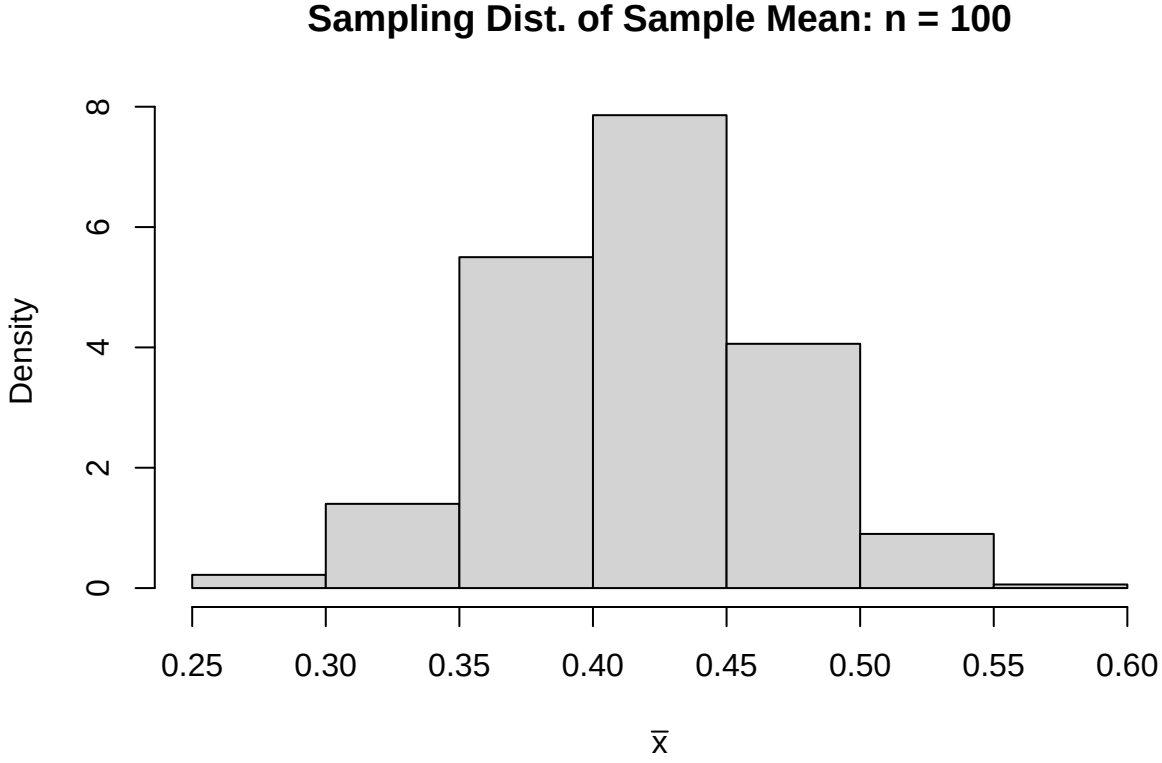
```
set.seed(1234) # for reproducibility

shots <- c(rep(1,337), rep(0, 464)) # data

sample_means <- rep(0,1000) # initialize sample means

# bootstrap sample means
for (i in 1:length(sample_means)){
  sample_means[i] <- mean(sample(shots, size = 100, replace = TRUE))
}

# plot
hist(sample_means, freq = FALSE, main="Sampling Dist. of Sample Mean: n = 100", xlab = expression(bar("
```




```
# 99% confidence interval for true proportion of shots made
quantile(sample_means, probs = c(0.005,0.995))
```

```
##  0.5% 99.5%
```

```
##  0.27  0.55
```