Final Project

Yahir B

2024-09-05

Contents

Predicting Future Pokémon Team

PSTAT 131 - Final Project

Yahir Baltazar

Preface

Before we start, allow me to set the scene. t's a nostalgic Sunday night; you have school the next day, your mom has just tucked you into bed and flipped the lights off. "Sweet dreams," she says as she turns off the lights and closes the door. Little does she know, we have different plans—the night is still young, after all. Somehow we managed to sneak our Nintendo 3DS under our pillow. Hiding under the covers you open your 3DS and pop in your Pokemon Y game you got for Christmas. Your having so much fun exploring the Kalos region and learning all the Pokémon until you realize there's a problem. There are 721 unique Pokémon in the game! Of those 721 pokemon, you can only choose 6 to be on your team! Holy Arceus, that's a lot of options. Struggling to make a decision, you subdue to the heavy weight of your eyelids, knocking out under the covers (of course you do, what 9 year old can stay up after midnight?. You wake up the next morning, your body aching from sleeping like a pretzel, and your DS plastered on your face covered in drool (gross I know). And there she stands, arms crossed, towering in the door way with a disappointed look on her face. And that's how you got your 3DS taken away for the rest of the week.

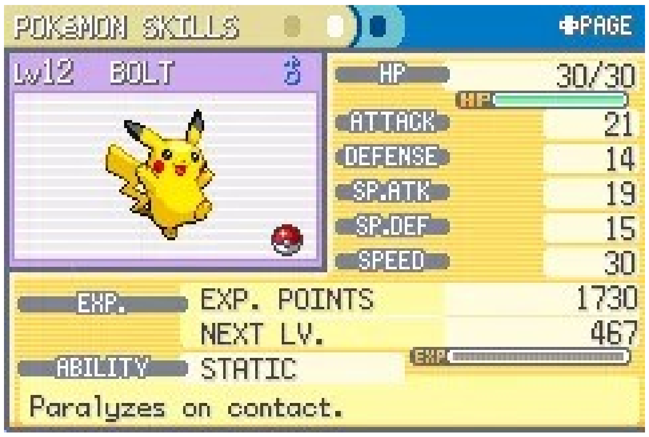
Introduction

No child should ever have to undergo such stressful and quite frankly traumatic situation. That is why the goal of this project will be to construct a statistical learning model to predict if a Pokemon will be on my very own team. In all seriousness though, I want to use this model to help the future generation of Pokemon trainers be 'the best that no one ever was' by making the overwhelming process of choosing a Pokemon team a little simpler. Pokémon holds a special place in my heart from childhood, and I hope to share the joy and excitement it brought me with others.

What is a Pokemon?

If you are reading this right now, and you're thinking to yourself, "Oh no, I don't know anything about pokemon!" Don't worry, I got you!
First of all, you know Pokemon, you definitely know this little guy.

Here's a little more background for our new trainers. Pokemon, in very simple terms, are cute little creatures with very strong and unique powers/abilities. As we mentioned previously, there are over 700 different types of pokemon! In this fictional universe, Pokemon trainers capture and level up their Pokemon by battling other Pokemon. Each Pokemon has a Type characteristic (ie. Water, Fire, Electric, Grass, etc) as well as 'stats' (ie. HP, Attack, Defense, Speed, etc) that determine its strengths and weaknesses. We'll explain these a little more when we analyze our dataset's variables. Here is an example of the stats on the right side of the picture.



Throughout this project, we will be using the following R packages to make our life a little easier.

```r
library(tidymodels)
library(tidyverse)
library(ggplot2)
library(corrplot)
library(knitr)
library(discrim)
library(rsample)
library(yardstick)
library(scales)
library(rpart.plot)
tidymodels_prefer()
```

Exploring Data Analysis

The Data

And now for the 'scary' part (It's really not). Let's bring in the data! For this project we'll be using an excel file containing the information and statistics of all 721 Pokemon. This excel file was derived from Kaggle titled '**Pokemon with Stats'** by *Alberto Barradas.* After downloading the kaggle dataset I slightly modified some of the observations and variables to better fit our situation (we'll explore these modifications more in this section. Below is the updated csv file with modified variables and observations.

Manual Modifications

After downloading the csv file, I manually added a column called `Team` of True/False values, signifying whether I would consider adding the Pokemon to my team.

Fortunately for us there isn't any missing observations! There is, however, a handful of duplicate pokemon entries which required me to manually tidying up. For example some Pokemon happen to have different forms, many due to gender/item characteristics. Pokemon #479 Rotom, for example, has 6 different forms all with different `Types` and stats; therefore I have decided to only use it's base form rather than considering each form. I applied this method, somewhat arbitrarily, to any Pokemon with different forms.

Likewise, some Pokemon, those who have the 'Mega Evolution' characteristic, had duplicate entries too. In order to preserve this characteristic I went ahead and added another variable called `Mega` describing if a Pokemon was capable of 'Mega Evolution' with True/False values.

Finally, I manually switched some of the Pokemon's `TYPE 1` and `TYPE 2` to better balance the data. I did this because as you will see later, we will not be considering `Type 2` as a variable.

The following is a link to the updated data set with my manual modifications: Modified_Data

Loading and Tidying Data

Lets load the data and see what we're working with!

```r
raw_pokemon_data <- read_csv('/Users/yahir/PokemonUpdated.csv')
head(raw_pokemon_data)
```

```
## # A tibble: 6 x 15
##     `#` 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     4 Charma~ Fire     <NA>       309    39     52      43        60        50
## 5     5 Charme~ Fire     <NA>       405    58     64      58        80        65
## 6     6 Chariz~ Fire     Flying     534    78     84      78       109        85
## # i 5 more variables: Speed <dbl>, Generation <dbl>, Legendary <lgl>,
## #   Team <lgl>, Mega <lgl>
```

Analyzing Variables

Let's see what kind of variables we are working with! Below is a general description of what each variable in the data set:

- **Num**: ID for each Pokemon
- **Name**: Name of each Pokemon
- **Type 1**: Each Pokemon has a type, this determines weakness/resistance to attacks
- **Type 2**: Some Pokemon are dual type and have 2
- **Total**: sum of all stats that come after this, a general guide to how strong a pokemon is
- **HP**: hit points, or health, defines how much damage a pokemon can withstand before fainting
- **Attack**: the base modifier for normal attacks (eg. Scratch, Punch)
- **Defense**: the base damage resistance against normal attacks
- **SP Atk**: special attack, the base modifier for special attacks (e.g. fire blast, bubble beam)
- **SP Def**: the base damage resistance against special attacks
- **Speed**: determines which Pokemon attacks first each round
- **Generation**: provides number of the generation the Pokemon belongs to
- **Legendary**: Indicates whether the Pokemon is considered Legendary
- **Team**: indicates whether I would add the Pokemon to my team if I were to run into them in game

This variable description is from the kaggle data set '**Pokemon with Stats'** by *Alberto Barradas.* I did, however,add and define my own variables to the codebook.

I have also attached a text file containing the variable names and description as well.

Dropping Variables

As interesting as all these variables can be, some aren't very useful. One could say they're not very effective :D

Lets see which ones we're dropping and why:

1. `Type 2` - Taking a look at this variable, notice that not every Pokemon has a second type. Because of this, I chose to not include it. Instead we'll only focus only consider `Type 1` . While this may cause some unforeseen bias in the model, I promise you, as someone who's been playing for 10 years, I forget some of these Pokemon have a second type, haha!
2. `Num` - Its not a very useful variable since Pokemon names happen to be unique, so just like that lvl 5 Bidoof I accidentally caught, we'll release it into the wild (we will drop it)
3. `HP`, `Attack`, `Defense`, `Sp. Atk`, `Sp. Def`, `Speed` - In order to reduce the chance of over-fitting, we'll be using `Total` which is just the sum of all these stats

poor bidoof :(

Lets drop them!

```
pokemon_data <- raw_pokemon_data %>% select(-`Type 2`, -`#`, -`HP`, -`Attack`,-`Defense`,-`Sp. Atk`,-`Sp
head(pokemon_data)
```

```
## # A tibble: 6 x 7
##   Name       ‘Type 1‘ Total Generation Legendary Team  Mega
##   <chr>      <chr>    <dbl>      <dbl> <lgl>     <lgl> <lgl>
## 1 Bulbasaur  Grass      318          1 FALSE     FALSE FALSE
## 2 Ivysaur    Grass      405          1 FALSE     FALSE FALSE
## 3 Venusaur   Grass      525          1 FALSE     FALSE TRUE
## 4 Charmander Fire       309          1 FALSE     TRUE  FALSE
## 5 Charmeleon Fire       405          1 FALSE     FALSE FALSE
## 6 Charizard  Fire       534          1 FALSE     FALSE TRUE
```

Renaming Our Variables

We will also rename our variables to make our syntax more consistent and our code easier to replicate. Below I have simply removed the spaces and capitalized each variable accordingly, this will make our jobs a lot easier, especially in the long run!

```
pokemon_data <- pokemon_data %>%
  rename(NAME = Name,
         TYPE = `Type 1`,
         GENERATION = Generation,
         LEGENDARY = Legendary,
         TEAM = Team,
         MEGA = Mega,
         TOTAL = Total
         )
```

Converting Factors

Now we'll go ahead and convert our categorical variables into factors. In our case, we will be converting the variable `TYPE` and our variable whose value are TRUE/FALSE: `GENERATION`, `LEGENDARY`, `TEAM`, `MEGA`

```
pokemon_data$TYPE <- as.factor(pokemon_data$`TYPE`)
pokemon_data$GENERATION <-as.factor(pokemon_data$GENERATION)
pokemon_data$LEGENDARY <-as.factor(pokemon_data$LEGENDARY)
pokemon_data$TEAM <-as.factor(pokemon_data$TEAM)
pokemon_data$MEGA <- as.factor(pokemon_data$MEGA)


head(pokemon_data)
```

```
## # A tibble: 6 x 7
##   NAME       TYPE  TOTAL GENERATION LEGENDARY TEAM  MEGA
##   <chr>      <fct> <dbl> <fct>      <fct>     <fct> <fct>
## 1 Bulbasaur  Grass   318 1          FALSE     FALSE FALSE
## 2 Ivysaur    Grass   405 1          FALSE     FALSE FALSE
## 3 Venusaur   Grass   525 1          FALSE     FALSE TRUE
## 4 Charmander Fire    309 1          FALSE     TRUE  FALSE
## 5 Charmeleon Fire    405 1          FALSE     FALSE FALSE
## 6 Charizard  Fire    534 1          FALSE     FALSE TRUE
```

Visual EDA

Let's take a moment to visually analyze our data and see if there are any interesting patterns. Not only is this fun to visualize, but it will be useful to better understand how our model works. By understanding the underlying patterns, we can better understand the outcome and make any changes, if needed.

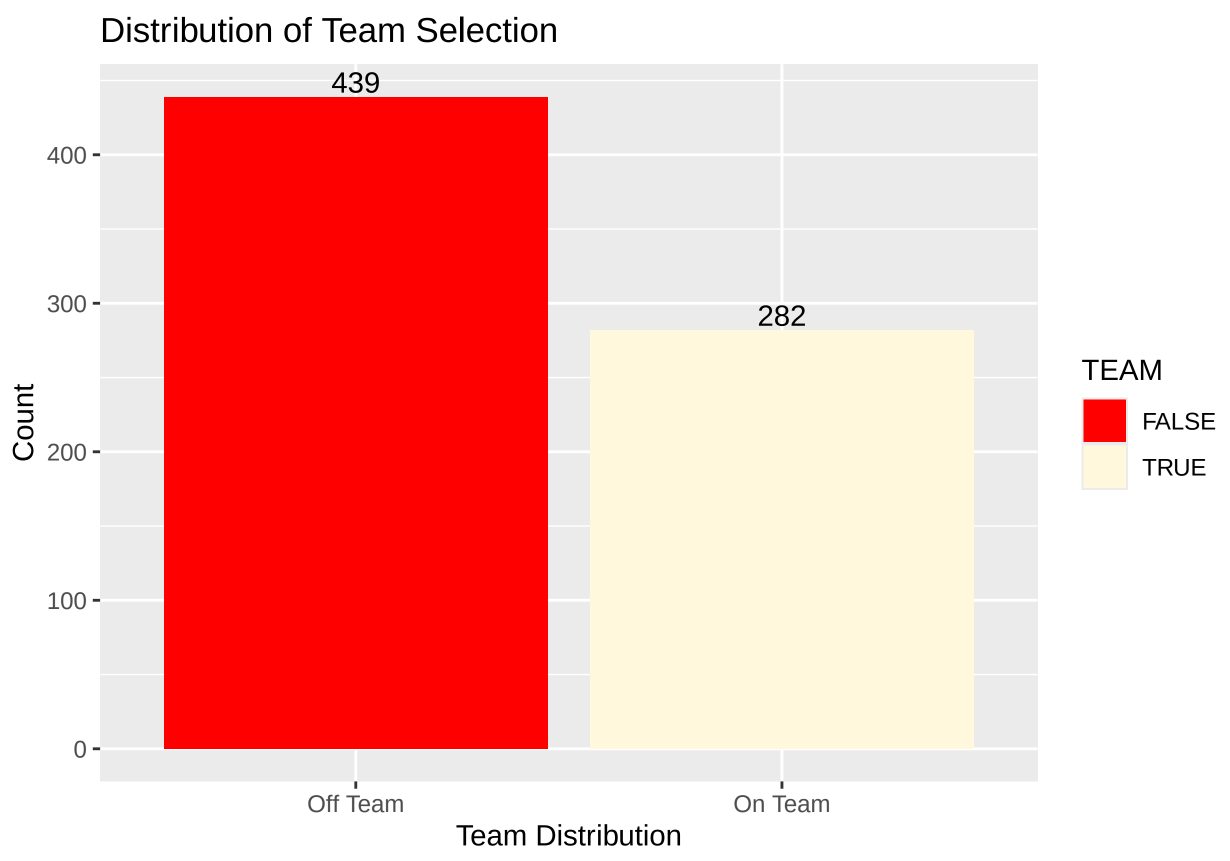
Outcome Distribution

Let's first explore our outcome variable `TEAM`. First, let's plot the distribution of our TRUE and FALSE values using a bar graph. We'll do this in order to assess how balanced the result of our outcome variable is. To make it fun, lets use Pokeball colors! (red and white)

```
team_count <- sum(pokemon_data$TEAM == 'TRUE')
no_team_count<- sum(pokemon_data$TEAM == 'FALSE')



pokemon_data %>%
  ggplot(aes(x = TEAM), fill = "Team") +
  geom_bar(aes(fill = TEAM)) +
  geom_text(stat = "count", aes(label = ..count..), vjust = -.2) +
  scale_x_discrete(labels = c("TRUE" = "On Team", "FALSE" = "Off Team")) +
  scale_fill_manual(values = c("TRUE" = "cornsilk", "FALSE" = "red")) +
  labs(title = "Distribution of Team Selection",
       x = "Team Distribution",
       y = "Count")
```



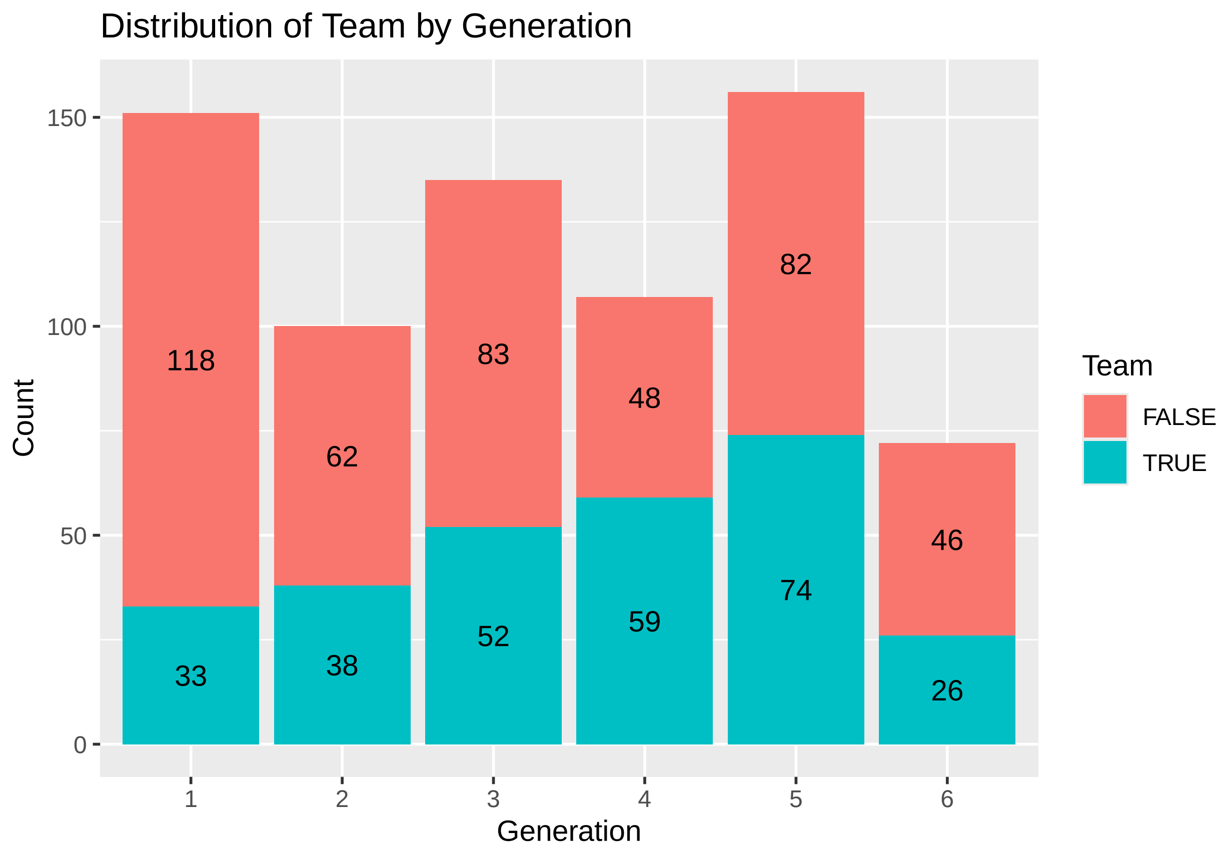
```r
#given the chance, of course I will use classic pokeball colors!
```

Let's Go! It looks pretty balanced to me. We definitely have more Pokemon with the FALSE value for `TEAM` compared to TRUE. This should not be alarming though. Remember this means have more Pokemon to reject and therefore less to consider for our team. Hopefully our model will be able to follow this pattern and will consider less pokemon to add to our team. This would make our life a lot easier especially since we only get to choose 6 Pokemon for our Pokemon team. Less is better! Since there is no extreme case of imbalanced data, we'll move forward.

Generation Distribution

I, like many other trainers, am very biased when it comes to my favorite generation. From a Pokemon veteran's perspective, best believe not every generation is built the same. Certain generations released more Pokemon, many of which my 10 year old self would consider 'cool' and would then add them to my team . Let's take a look at the distribution our outcome variable `TEAM` separated by generation. While I naturally have my suspicions of my favorite Pokemon generation, lets visualize it with the data.

```r
#creating plot of selected pokemon by generation
pokemon_data %>%
  ggplot(aes(x = GENERATION, fill = TEAM)) +
  geom_bar(position = "stack") +
  geom_text(stat = "count", aes(label = ..count..), position = position_stack(vjust = 0.5)) +
  labs(title = "Distribution of Team by Generation",
       x = "Generation",
       y = "Count",
       fill = "Team")
```



Well, can you guess which generation is my favorite? Look at generation 5! Given the opportunity, I would be more than happy to have 74 of Generation 5's Pokemon on my team. Meanwhile poor little generation 6 only gets 26 Pokemon. In my defense, I grew up with Generation 5 - it was my first ever Pokemon game! The total number of TRUE/FALSE values by generation is interesting, however it is important to note the each generation has a different amount of Pokemon. For example, Generation 6 only has 72 new Pokemon, while Generation 1 has 151 new Pokemon.

To solve this issue, lets instead look at the percentage of of Pokemon who I would consider adding to my team, grouped by Generation. Below we have a table showing each Generation and the percentage of Pokemon who's `TEAM` value was TRUE.

```
gen_proportions <- pokemon_data %>%
  count(GENERATION, TEAM) %>%                 # Count occurrences of each TEAM within each
  group_by(GENERATION) %>%                    # Group by GENERATION
  mutate(total = sum(n)) %>%                  # Calculate total counts within each GENERATION
  mutate(Percentage = (n / total)*100) %>%          # Calculate proportion for each TEAM
  select(GENERATION, TEAM, Percentage)        # Select relevant columns


gen_proportions %>%
  filter(TEAM == TRUE) %>%
  select(GENERATION, Percentage) %>%
  kable(caption = "Percent of Pokémon On Team By Generations")
```

Table 1: Percent of Pokémon On Team By Generations

GENERATION	Percentage
1	21.85430
2	38.00000
3	38.51852
4	55.14019
5	47.43590
6	36.11111

```
#I used ChatGPT to help with the syntax of gen_proportions
```

WOAH! Lets take a look at each generation. Generations 4 and 5 have drastically higher number of worthy Pokemon (Pokemon whose `TEAM` value is true) closer to 50%. In other words, I naturally bias towards half of the Pokemon in Generations 4 and 5, since half of them would be considered on my team. Listen, y'all can call me a Pokemon bandwagon all you want, but Generations 4 and 5 are elite! These Generations arguably have the best Pokemon designs by far! It makes sense that these values have the highest percentage of Pokemon I'd allow on my team, especially since I grew up playing these generations.

Meanwhile Generations 2,3, and 6 have slightly lower percentages of about 38%. After some reflection, I would say that these generations are pretty decent, overall they have very interesting and strong Pokemon. I really only played these games when I was a bit older; it's possible that there is less of a nostalgia factor playing into my favoritism.

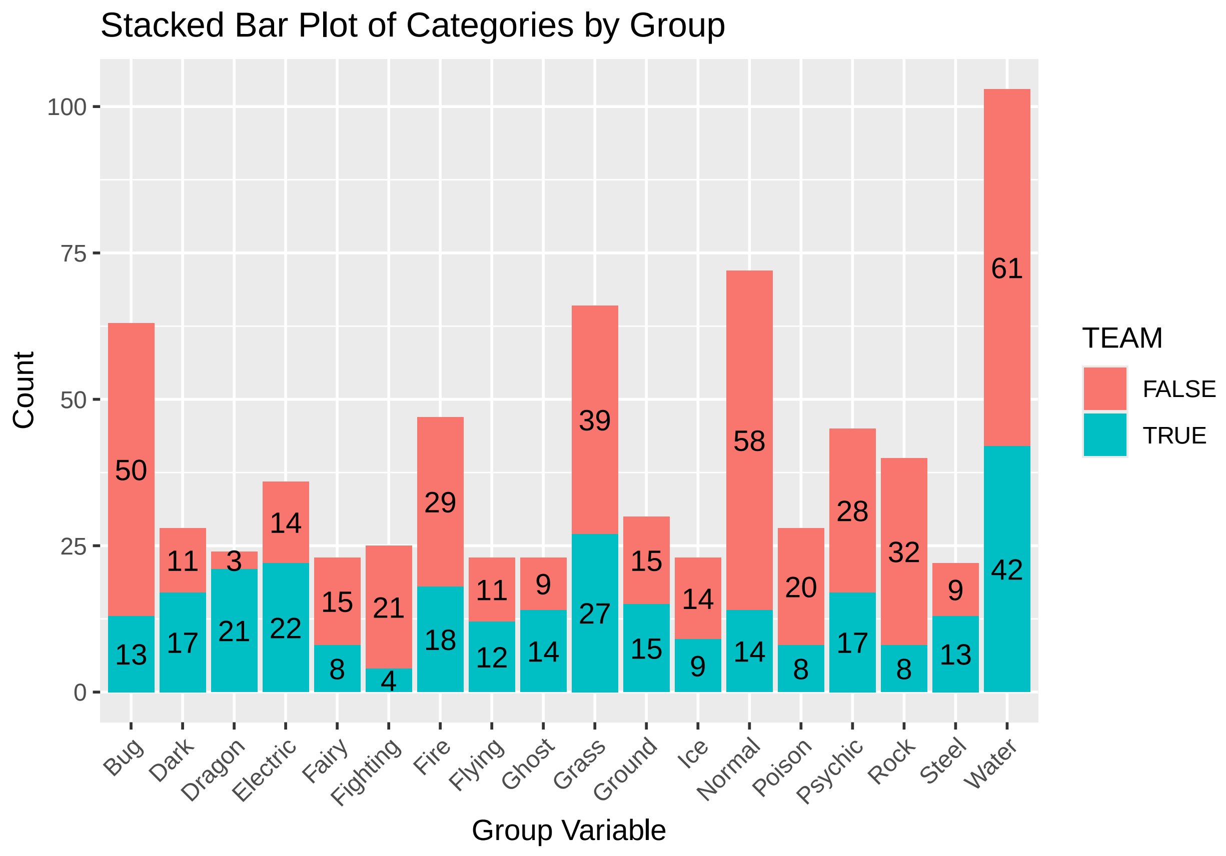
All you loyal Generation 1 lovers right now ^^^

I know a lot of the 'OG' Pokemon trainers are mad at me right now since Generation 1 has the lowest percentage of 21.8% . All I can say to that is, they were before my time (and they're kinda of basic). I mean look at this Generation 1 Pokemon, Pidgey. You can't sit here and tell me its not just a bird…

Pokemon Type Distribution

Once again, lets analyze the distribution of our outcome variable, but this time we'll visualize it by `TYPE`. By breaking down the data in this way, we can examine whether certain types of Pokemon are over represented in my selection, which might suggest natural biases or tendencies I have when building teams. Recognizing these patterns may also be useful for better understanding our predictive model. Let's run the code!

```
#This plots the proportion of pokemon on my team visualized by type
pokemon_data %>%
  ggplot(aes(x = TYPE, fill = TEAM)) +
  geom_bar(position = "stack") +
  geom_text(stat = "count", aes(label = ..count..), position = position_stack(vjust = 0.5)) +
  labs(title = "Stacked Bar Plot of Categories by Group",
       x = "Group Variable",
       y = "Count")+
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



```
type_prop <- pokemon_data %>%
  count(TYPE, TEAM) %>%
```

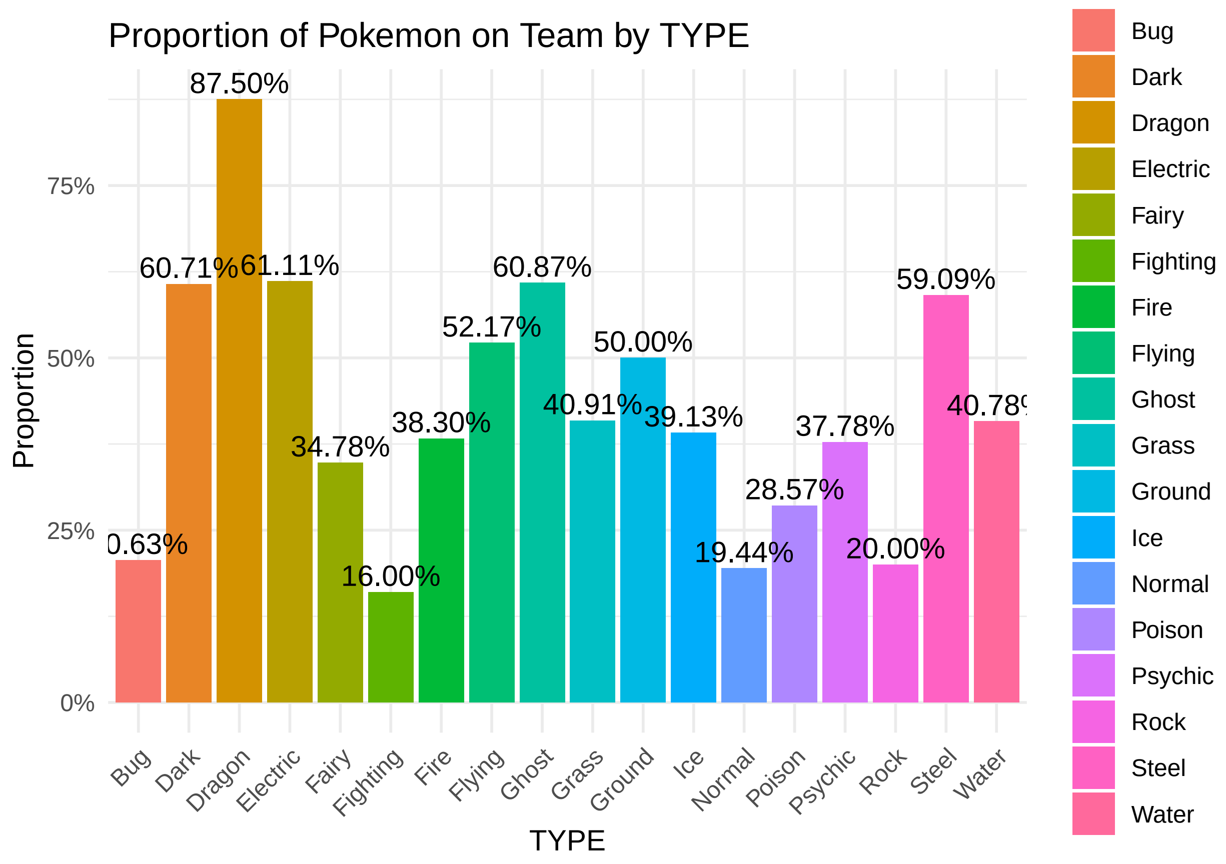
```r
  group_by(TYPE) %>%
  mutate(proportion = n / sum(n)) %>%
  filter(TEAM == "TRUE") %>%
  select(TYPE, proportion)
```

Holy Kyogre! That's a lot of Water type Pokemon on my team, 42 to be exact! Meanwhile, out of every Pokemon, there are only 4 Fighting-type Pokemon worth joining our team.

It is important to remember that we are only considering `TYPE 1` in this project, since we removed `TYPE 2` from the data. This might be where we run into some slight discrepancies; however, I did my best to minimize this misrepresentation during the 'Manual Data Modification' step. During this step, I manually swapped some of the Pokemon's `TYPE 1` and `TYPE 2` to create better representation. For example, many of the bird Pokemon such as Pidgeot, Unfezant, Fearow, Swellow, Braviary, and many more, have `TYPE 1 = 'Normal'` and `TYPE 2 = 'Flying'`. Since we're only considering TYPE 1, these Pokemon would be classified as Normal. However, looking at there designs, they are clearly birds and therefore should be considered as Flying yypes. That's the reason I manually changed their Types.

Similar to our previous steps, we'll analyze our proportions. Using a bar graph we'll plot each Generations proportion to better understand the distribution by `TYPE` .

```r
#We now plot the proportions of each
ggplot(type_prop, aes(x = TYPE, y = proportion, fill = TYPE)) +
  geom_bar(stat = "identity") +
  geom_text(aes(label = scales::percent(proportion)), vjust = -0.3) +
  labs(title = "Proportion of Pokemon on Team by TYPE",
       x = "TYPE",
       y = "Proportion") +
  scale_y_continuous(labels = scales::percent) +
  theme_minimal()+  theme(axis.text.x = element_text(angle = 45, hjust = 1)) #used ChatGPT to help plot
```

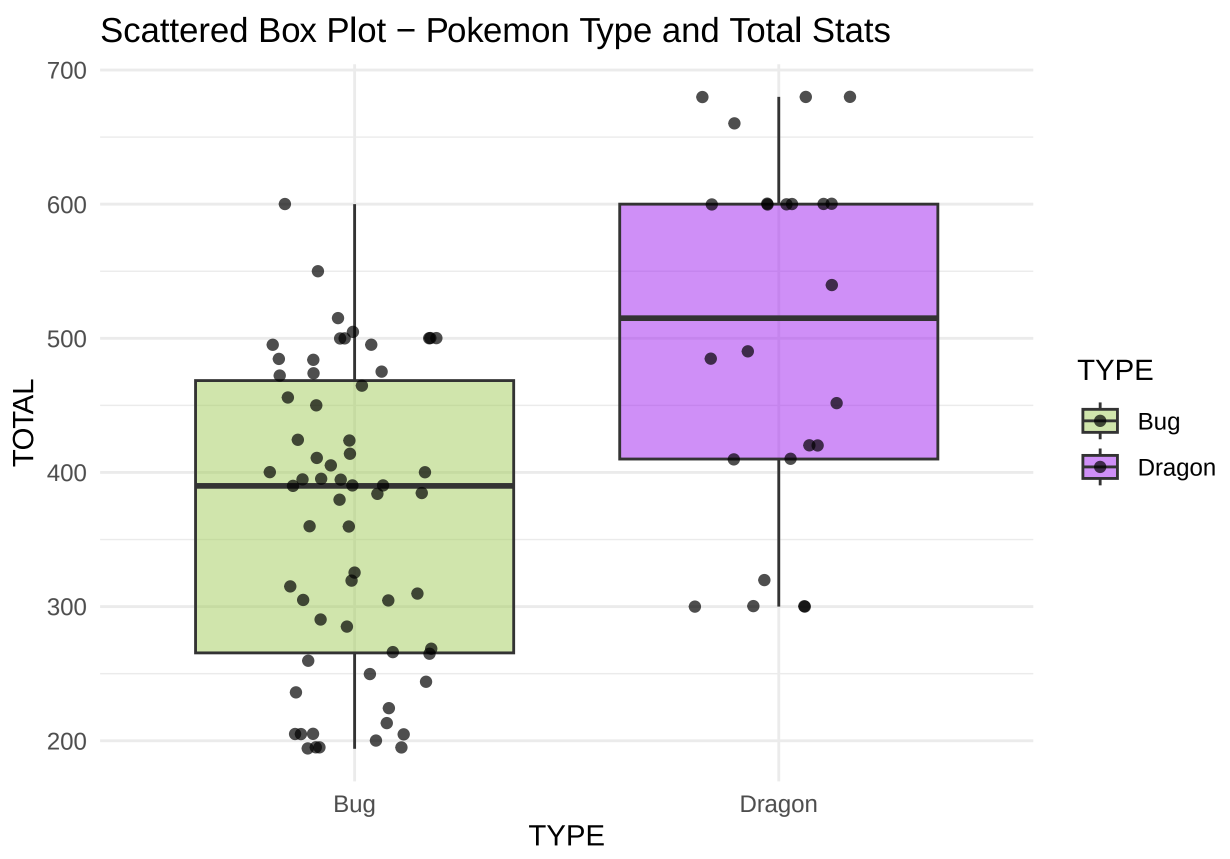


Can you guess what my favorite type is? Dragon type Pokemon have a significantly higher proportion of 87%. That is significantly higher than the rest of the Pokemon types. Even compared to the next highest value Dark type, there is a significant difference! Personally, this makes sense. My favorite type growing up was the Dragon type, both because the designs were usually cool!

Notice how small the proportion of Bug Type is! 0.63% is a bit suspicious. Lets compare our Dragon and Bug Types since they were our highest and lowest proportions of Pokemon on my team, respectively. We'll plot a box graph of each type, using the `TOTAl` variable. As a reminder, this variable is the sum of all of a Pokemon's stats (HP, Attack, Defense, Speed, Sp. Attack, Sp. Defense). We'll use it since its a good indicator of how strong each Pokemon is. Lets run the code!

```r
bug_dragon <- pokemon_data %>%
  filter(TYPE %in% c("Dragon", "Bug"))

# Lets create a box plots of Dragon and Bug Types!
ggplot(bug_dragon, aes(x = TYPE, y = TOTAL, fill = TYPE)) +
  geom_boxplot(alpha = 0.5) +  # Box plot with semi-transparent fill and no outliers
  geom_jitter(aes(color = TYPE), width = 0.2, alpha = 0.7) +  # Scatter plot with jitter (used CHAT_GPT
  labs(title = "Scattered Box Plot - Pokemon Type and Total Stats", x = "TYPE", y = "TOTAL") +
  scale_fill_manual(values = c("Dragon" = "purple", "Bug" = "darkolivegreen3")) +
  scale_color_manual(values = c("Dragon" = "black", "Bug" = "black")) +
  theme_minimal()
```



Let's interpret the results. First, we have Bug on the left with each individual Bug type Pokemon's `TOTAL` values plotted as scatter points. The green box illustrates the summary statistics of these scatter points. The Bug type box plot is centered around 350 (representing its mean), with an upper quartile of approximately 470 and a lower quartile of about 260. The scatter points points themselves range from a minimum of 200 to a maximum of 600.

Dragon type, on the other hand, is illustrated by the right box plot. It's summary statistics are shaded in green with scatter points representing each Dragon type Pokemon's `TOTAL` value. Notice the Dragon box is center around 510 (mean), with a lower quartile of 410 and a higher quartile of 600.

Oh my. Bug types aren't looking so hot right now. Look at how low the `TOTAL` stats are for bug type Pokemon! Also, notice the Bug Type Box Plot is significantly lower compared to the Dragon Type Box plot. Likewise, the Bug plot is negatively skewed since the median is towards the top of the box. This suggests that the mean of the bug totals is significantly lower than the median, implying that, overall, Bug type Pokemon typically have lower total stats. No wonder I don't want bugs on my team!

I'm sorry but Misty was right, "**Bugs are one of the 3 of the most disgusting things in the world**!" We'll also choose to stay away from them!

Something feels wrong though. Why are there so many outliers in the Dragon type box plot whose `TOTAL` is close to 700? Lets run some code to look at our Dragon type Pokemon. More specifically, lets look at the Dragon type Pokemon whose TOTAL is greater than or equal to 500. We'll choose 500 since it is very close to the mean of the Dragon Box Plot. Likewise, lets consider whether these Pokemon are Legendary or not.

```
high_dragons <- pokemon_data %>%
  filter(TYPE == "Dragon", TOTAL >= 500) %>%
  select(NAME,TOTAL, LEGENDARY, MEGA)%>%
  arrange(desc(LEGENDARY))
```

```r
kable(high_dragons, caption = "High TOTAL Dragon-type Pokémon")
```

Table 2: High TOTAL Dragon-type Pokémon

NAME	TOTAL	LEGENDARY	MEGA
Latias	600	TRUE	TRUE
Latios	600	TRUE	TRUE
Rayquaza	680	TRUE	TRUE
Reshiram	680	TRUE	FALSE
Zekrom	680	TRUE	FALSE
Kyurem	660	TRUE	FALSE
Zygarde	600	TRUE	FALSE
Dragonite	600	FALSE	FALSE
Salamence	600	FALSE	TRUE
Garchomp	600	FALSE	TRUE
Haxorus	540	FALSE	FALSE
Goodra	600	FALSE	FALSE

NO WONDER THE STATS ARE SO HIGH!! More than half (7/12) of the Pokemon are legendary! This doesn't sound very fair if you ask me. No wonder I naturally gravitated towards Dragon type Pokemon as a kid. It really did make the game easier! Normally this would be a problem, but I'm not too worried. The point of this project is to predict Pokemon to be on our team. Hence this Legendary dragon pattern should make it easier for our model to predict correctly.

Setting up Models

We've collected all our gym badges, and now its time to head through victory road! Lets start setting up our models!

Splitting the Data

Before we're able to get to the models, we need to split our data into training and testing. We'll use stratify in order to preserve the proportions of our outcome variable `TEAM`. We'll set a seed in order so that our random initial split can be reproduced in the future.

```r
set.seed(1025) # fun fact there are 1025 unique pokemon as of 2024! :D

poke_split <- initial_split(pokemon_data, prop =.7, strata =TEAM)

pokemon_train <- training(poke_split)
pokemon_test <- testing(poke_split)
```

Let's make sure our split works appropriately.

```r
dim(pokemon_train)
```

```
## [1] 504   7
```

```
dim(pokemon_test)
```

```
## [1] 217   7
```

Looks like it was super effective! We could always use more data to train our model but we are somewhat limited by the number of observations. If you notice, in the code, we chose to split our training data to have 70% of the observations. This will be enough data to train our model with.

Recipe

Its cooking time! Let's set up our recipes!

We'll be using recipes to make our data easier to work with and ready for modeling. In our case, our recipe will let us handle scaling numbers, converting categories into dummy variables, as well as helping us standardize our numerical variables. Using this `pokemon_recipe` will save us a lot of time and allow us to confidently produce consistent results throughout each data step.

```
pokemon_recipe <- recipe(TEAM ~ TYPE + TOTAL + GENERATION + LEGENDARY + MEGA, data = pokemon_train) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_center(TOTAL) %>%
  step_scale(TOTAL)
```

Now that our recipe is set up, it's always a good idea to double check that our recipe works. The last thing we want is for an error to hide in this recipe. Lets check using `prep` and `bake`

```
prep(pokemon_recipe) %>%
  bake(pokemon_train)
```

```
## # A tibble: 504 x 26
##      TOTAL TEAM  TYPE_Dark TYPE_Dragon TYPE_Electric TYPE_Fairy TYPE_Fighting
##      <dbl> <fct>     <dbl>       <dbl>         <dbl>      <dbl>         <dbl>
##  1  0.960  FALSE         0           0             0           0             0
##  2 -0.111  FALSE         0           0             0           0             0
##  3 -1.99   FALSE         0           0             0           0             0
##  4 -1.99   FALSE         0           0             0           0             0
##  5 -1.90   FALSE         0           0             0           0             0
##  6 -0.200  FALSE         0           0             0           0             0
##  7 -1.49   FALSE         0           0             0           0             0
##  8 -0.611  FALSE         0           0             0           0             0
##  9 -0.0398 FALSE         0           0             0           0             0
## 10 -1.39   FALSE         0           0             0           0             0
## # i 494 more rows
## # i 19 more variables: TYPE_Fire <dbl>, TYPE_Flying <dbl>, TYPE_Ghost <dbl>,
## #   TYPE_Grass <dbl>, TYPE_Ground <dbl>, TYPE_Ice <dbl>, TYPE_Normal <dbl>,
## #   TYPE_Poison <dbl>, TYPE_Psychic <dbl>, TYPE_Rock <dbl>, TYPE_Steel <dbl>,
## #   TYPE_Water <dbl>, GENERATION_X2 <dbl>, GENERATION_X3 <dbl>,
## #   GENERATION_X4 <dbl>, GENERATION_X5 <dbl>, GENERATION_X6 <dbl>,
## #   LEGENDARY_TRUE. <dbl>, MEGA_TRUE. <dbl>
```

Looks good! The recipe correctly turns each nominal predictor in to dummy variables and it scales and standardizes our numeric variable, `TOTAL`.

K-Fold Cross Validation

We'll use 10 folds to perform our cross validation, while stratifying on our `TEAM` variable to ensure that each fold maintains the same proportion of team categories. This will help preserve the balance of our target variable throughout the process. Rather than directly using the accuracy metric from fitting the models to our training data, we'll instead use cross validation with our 10 folds. Cross validation allows for a better estimation of accuracy since it averages the accuracy across each fold. In doing so, we get a more realistic estimate of how well the model will perform on new data. This way, we're not just relying on one split of the data, which can be a bit random, but instead looking at how the model performs across multiple subsets. It essentially gives us a better sense of the model's true accuracy.

```
pokemon_folds <- vfold_cv(pokemon_train, v=10, strata = TEAM)
```

Constructing Models

We've made it to the elite 4! The elite 4 models for prediction, if you will. Below I have chosen to work with my favorite classification models. We'll be exploring them individually.

For each model we'll follow a general outline:

1. Create model
2. Create workflow
3. Fit Model to Training Data
4. Fit Model to K Fold
5. Collect/Assess Accuracy and ROC AUC of step 4
6. Plot ROC Curve

Logistic Regression Model

We'll first create our logistic regression model, `pokemon_log_mod` . In the code below, we'll create the model, workflow, fit the model to our training data, and fit it with our k folds.

```r
#Logistic Regression
pokemon_log_mod <- logistic_reg() %>%
  set_mode('classification') %>%
  set_engine('glm')

#Logistic Regression Workflow
pokemon_log_wkf <- workflow() %>%
  add_model(pokemon_log_mod) %>%
  add_recipe(pokemon_recipe)

#Fitting Logistic Regression Model
pokemon_log_fit <- fit(pokemon_log_wkf, pokemon_train)
predict(pokemon_log_fit, new_data= pokemon_train, type = 'prob')
```

```
## # A tibble: 504 x 2
##    .pred_FALSE .pred_TRUE
##          <dbl>      <dbl>
##  1       0.481     0.519
##  2       0.810     0.190
##  3       0.941     0.0590
##  4       0.941     0.0590
##  5       0.940     0.0600
##  6       0.763     0.237
##  7       0.698     0.302
##  8       0.659     0.341
##  9       0.918     0.0816
## 10       0.693     0.307
## # i 494 more rows
```

```r
#fitting folds
pokemon_log_kfold <- fit_resamples(pokemon_log_wkf, pokemon_folds, control = control_resamples(save_pre
```

Let's see how our model did with the k fold data. While this model doesn't have any hyper parameters, it is still important to measure its accuracy and ROC_AUC.

```r
log_metrics <- collect_metrics(pokemon_log_kfold) %>%
  filter(.metric %in% c("accuracy", "roc_auc"))

log_metrics
```

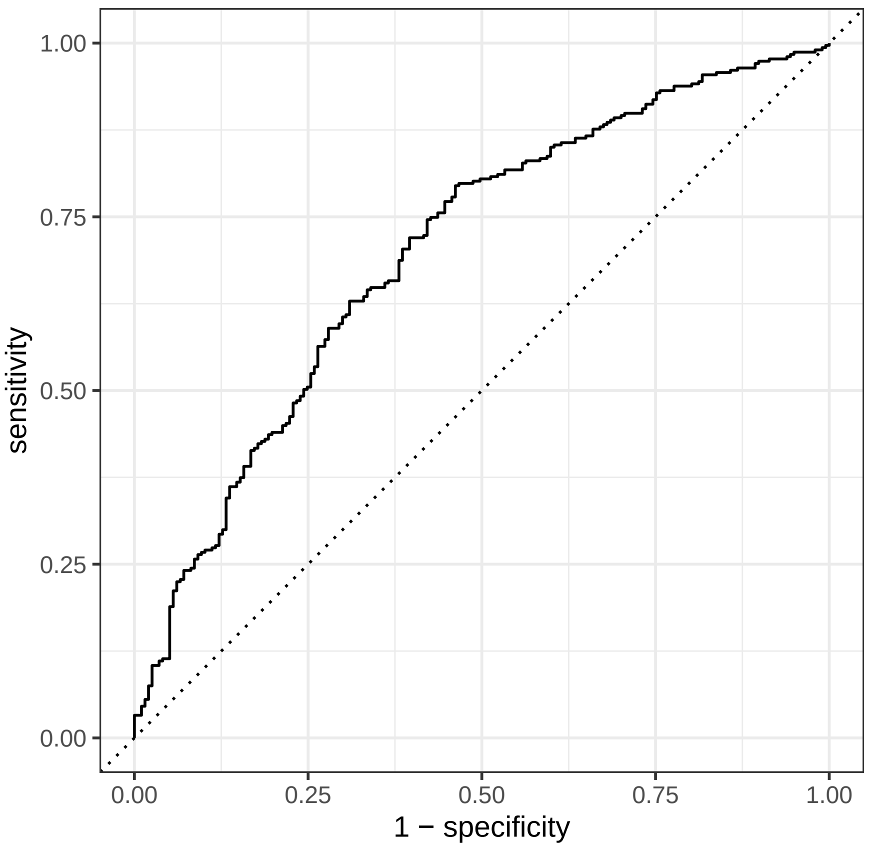
```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.683    10  0.0180 Preprocessor1_Model1
## 2 roc_auc  binary     0.696    10  0.0189 Preprocessor1_Model1
```

Not too bad! Ideally we want the mean closer of `accuracy` and `roc_auc` closer to 1. Lets plot the ROC curve just so we can visualize it a bit better!

```r
log_predictions <- collect_predictions(pokemon_log_kfold)
```

```r
#creating ROC Curve
roc_log <- roc_curve(log_predictions, truth = TEAM, .pred_FALSE) %>%
  autoplot()

roc_log
```



Linear Discriminant Model

Again, we'll follow the same outline. First creating the model, then workflow, fitting to the training data, and then to our k folds.

```r
#Linear Discriminant Model
pokemon_lda_mod <- discrim_linear() %>%
  set_mode('classification') %>%
  set_engine("MASS")

#Linear Disc Workfolow
pokemon_lda_wkf <- workflow() %>%
  add_model(pokemon_lda_mod) %>%
  add_recipe(pokemon_recipe)

#Linear Model Fit
```

```r
pokemon_lda_fit <- fit(pokemon_lda_wkf, pokemon_train)
predict(pokemon_lda_fit, new_data = pokemon_train, type = 'prob')
```

```
## # A tibble: 504 x 2
##    .pred_FALSE .pred_TRUE
##          <dbl>      <dbl>
##  1       0.472     0.528
##  2       0.815     0.185
##  3       0.941     0.0588
##  4       0.941     0.0588
##  5       0.940     0.0598
##  6       0.758     0.242
##  7       0.686     0.314
##  8       0.646     0.354
##  9       0.921     0.0790
## 10       0.681     0.319
## # i 494 more rows
```

```r
#fitting our folds
pokemon_lda_kfold <- fit_resamples(pokemon_lda_wkf, pokemon_folds, control = control_grid(save_pred = T
```

Lets see how the model did, again we'll look at the accuracy and ROC AUC.

```r
lda_metrics<- collect_metrics(pokemon_lda_kfold)%>%
  filter(.metric %in% c("accuracy", "roc_auc"))
lda_metrics
```

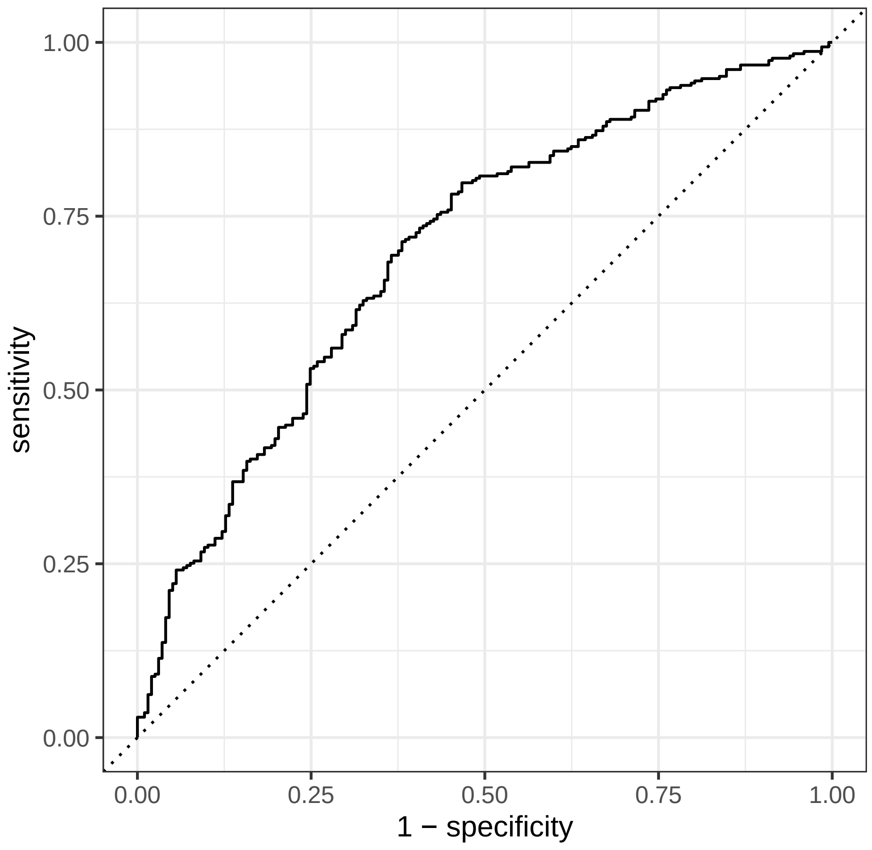
```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.683    10  0.0192 Preprocessor1_Model1
## 2 roc_auc  binary     0.698    10  0.0186 Preprocessor1_Model1
```

Not bad at all! Similar to the Logistic regression model, we're in the high 60s for both `accuracy` and `auc_roc`. Now lets plot the ROC curve for fun.

```r
lda_predictions <- collect_predictions(pokemon_lda_kfold)

#Creating ROC Curve
roc_lda <- roc_curve(lda_predictions, truth = TEAM, .pred_FALSE) %>%
  autoplot()

roc_lda
```



Quadratic Discriminant Model

We'll perform the same steps as our previous models.

```
#Quadratic Discriminant Model
pokemon_qda_mod <- discrim_quad() %>%
  set_mode('classification') %>%
  set_engine('MASS')

pokemon_qda_wkf <- workflow() %>%
  add_model(pokemon_qda_mod) %>%
  add_recipe(pokemon_recipe)

pokemon_qda_fit <- fit(pokemon_qda_wkf, pokemon_train)
predict(pokemon_qda_fit, new_data = pokemon_train, type = 'prob')
```

```
## # A tibble: 504 x 2
##    .pred_FALSE .pred_TRUE
##          <dbl>      <dbl>
##  1      0.0166 0.983
##  2      0.998  0.00170
##  3      1.00   0.00000722
##  4      1.00   0.00000722
##  5      1.00   0.00000788
##  6      0.916  0.0839
##  7      0.121  0.879
##  8      0.0426 0.957
##  9      1.00   0.000146
## 10      0.108  0.892
## # i 494 more rows
```

```r
pokemon_qda_kfold <- fit_resamples(pokemon_qda_wkf, pokemon_folds, control = control_grid(save_pred = T
```

Again, lets see how our quadratic discriminant model did! We'll go ahead and collect our `accuracy` and `roc_auc` metrics.

```r
qda_metrics <- collect_metrics(pokemon_qda_kfold)%>%
  filter(.metric %in% c("accuracy", "roc_auc"))

qda_metrics
```

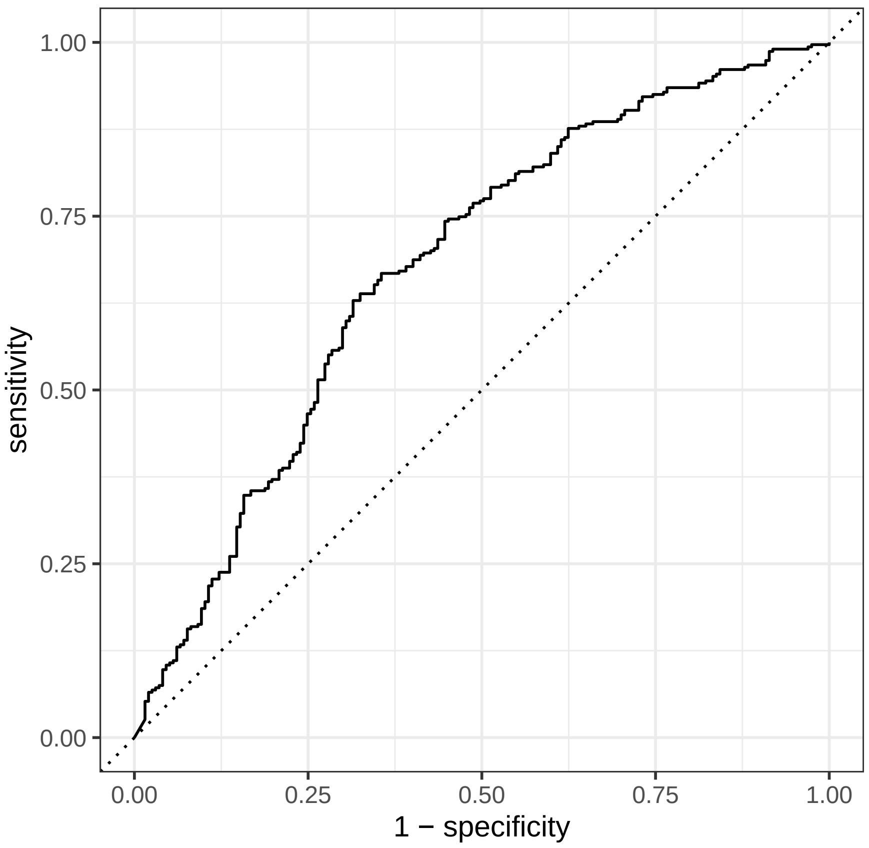
```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.669    10  0.0168 Preprocessor1_Model1
## 2 roc_auc  binary     0.681    10  0.0211 Preprocessor1_Model1
```

Wow, that a little shocking, our quadratic discriminant model also did very similar to our previous two models scoring in the high 60s. Lets plot the ROC curve.

```r
qda_predictions <- collect_predictions(pokemon_qda_kfold)

roc_qda <- roc_curve(qda_predictions, truth = TEAM, .pred_FALSE) %>%
  autoplot()

roc_qda
```



Pruned Decision Tree Model

Our final classification model is our Pruned Decision Tree. This time, we'll fist create our model and workflow

```
pokemon_tree_spec <-decision_tree(cost_complexity = tune()) %>%
  set_engine("rpart") %>%
  set_mode("classification")

pokemon_tree_wkf <- workflow() %>%
  add_model(pokemon_tree_spec) %>%
  add_recipe(pokemon_recipe)
```

Now is where things change from our usual outline. With our Pruned Decision Tree Model we are able to evaluate different hyperparameters. We'll consider every hyper-parameter and choose the best one according to its `ROC_AUC`. Once the best hyperparameter is selected, we'll go ahead and add it into our workflow and fit the model to our training data.

```
pokemon_param_grid <- grid_regular(cost_complexity(range = c(-3, -1)), levels = 10)

#tuning hyper-parameter (considering each hyper parameter)
pokemon_tune_tree <- tune_grid(
  pokemon_tree_wkf,
  resamples = pokemon_folds,
  grid = pokemon_param_grid,
  control = control_grid(save_pred = TRUE)
)

#choosing the best hyperparameter
best_roc_auc <- select_best(pokemon_tune_tree, metric = "roc_auc")

print(best_roc_auc)
```

```
## # A tibble: 1 x 2
##   cost_complexity .config
##             <dbl> <chr>
## 1           0.001 Preprocessor1_Model01
```

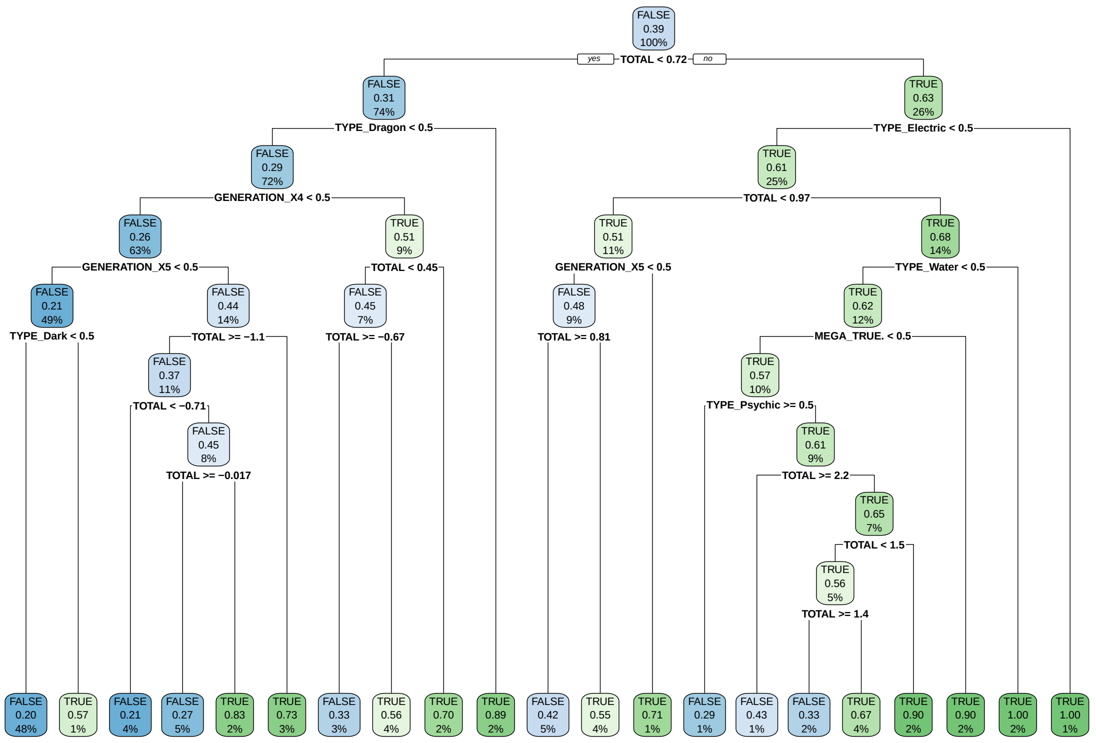
It looks like `Preprocessor1_Model01` performed best overall! We'll finalize our workflow using this model!

```
# finalizing workflow with the best hyperparameter
pokemon_tree_final <- finalize_workflow(pokemon_tree_wkf, best_roc_auc)

# fitting the finalized model on the training data
pokemon_tree_final_fit <- fit(pokemon_tree_final, data = pokemon_train)
```

Before we collect our metrics, lets plot of visual of our tree!

```
#plotting tree
pokemon_tree_final_fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```



Finally, lets collect the ROC_AUC of our best tree model!

```
pokemon_tune_metrics <- collect_metrics(pokemon_tune_tree)

poke_tree_metrics <- pokemon_tune_metrics %>%
  filter(.metric == "roc_auc") %>%
  arrange(desc(mean)) %>%
  slice(1)

# Print the best ROC AUC metric
print(poke_tree_metrics)
```

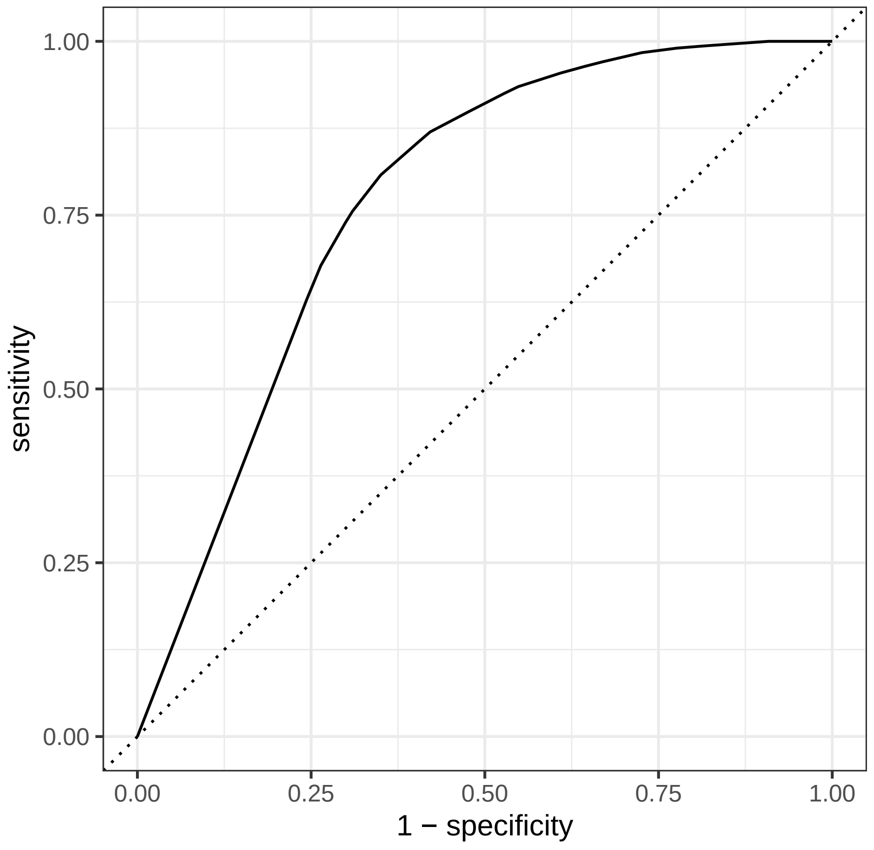
```
## # A tibble: 1 x 7
##   cost_complexity .metric .estimator  mean     n std_err .config
##             <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1           0.001 roc_auc binary     0.632    10  0.0193 Preprocessor1_Model01
```

If we look at out mean it turns out to be about 63.19% which, in my opinion, isn't too bad! Lets go ahead and plot the ROC Curve just like the previous steps.

```
#plotting ROC curve
#confusion matrix for accuracy/roc
pokemon_tree_roc <- augment(pokemon_tree_final_fit, new_data = pokemon_train)

pokemon_tree_roc %>%
```

```r
  roc_curve(TEAM, .pred_FALSE) %>%
  autoplot()
```



Now that the model have competed, lets look at the results!

Comparing our Model Results

Lets compare our model results, more specifically each model's `ROC_AUC`. We will display it using a table.

```r
log_metrics
```

```
## # A tibble: 2 x 6
##   .metric  .estimator  mean     n std_err .config
##   <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1 accuracy binary     0.683    10  0.0180 Preprocessor1_Model1
## 2 roc_auc  binary     0.696    10  0.0189 Preprocessor1_Model1
```

```r
final_log_roc <- log_metrics %>%
  filter(.metric == "roc_auc") %>%
  pull(mean)

final_lda_roc <- lda_metrics %>%
  filter(.metric == "roc_auc") %>%
  pull(mean)
```

```r
final_qda_roc <- qda_metrics %>%
  filter(.metric == "roc_auc") %>%
  pull(mean)

final_tree_roc <- poke_tree_metrics %>%
  pull(mean)

roc_auc_summary <- data.frame(
  Model = c("Logistic Regression", "LDA", "QDA", "Decision Tree"),
  ROC_AUC = c(
    final_log_roc,
    final_lda_roc,
    final_qda_roc,
    final_tree_roc
  )
)

kable(roc_auc_summary, caption = "ROC AUC Scores for Each Model")
```

Table 3: ROC AUC Scores for Each Model

Model	ROC_AUC
Logistic Regression	0.6960413
LDA	0.6975354
QDA	0.6808772
Decision Tree	0.6319369

As you can see, out highest `ROC_AUC` value is held by our Linear Discriminant Model! Between you and me, I am a little bummed, I was really hoping to use the pruned tree model :(. No worries though, it'll still be exciting to see how our model does on our testing data!

Queue Champion Cynthia's Battle theme because the moment has arrived…

Linear Discriminant Model I Choose You!

Since our Linear Discriminant Model did the best according to ROC_AUC, we will be using it to predict our testing data. Let's see how our model does on our testing data set.

```r
#fitting lda model to testing data
pokemon_test_augmented <- augment(pokemon_lda_fit, new_data = pokemon_test)

# Compute accuracy
accuracy <- pokemon_test_augmented %>%
  accuracy(truth = TEAM, estimate = .pred_class)

# Compute AUC ROC
auc_roc <- pokemon_test_augmented %>%
  roc_auc(truth = TEAM, .pred_FALSE)
```

```r
results_table <- tibble(
  Metric = c("Accuracy", "AUC ROC"),
  Value = c(accuracy$.estimate, auc_roc$.estimate)
)

#creating table of results
results_table %>%
  kable()
```

Metric	Value
Accuracy	0.6221198
AUC ROC	0.6786096

What a surprise, our model is pretty decent at predicting the testing data. If we recall, our `ROC_AUC` from our k-fold training data was approximately 0.697. Compared to the testing data, whose ROC_AUC value is 0.6786, we see that its not too far off! It looks like our k-fold cross validation was very useful in predicting how accurate our model would perform on the testing data. Hmm, I'm not entirely satisfied though.

Okay, just for funsies lets see how our other model, the Pruned Decision Tree, would have done! (Don't call the statistical ethics police, they might get mad a me). Obviously, in the real world, we shouldn't apply multiple models to our testing data, since it can lead to overfitting and inaccuracies when making future predictions. Just this once, lets see how our Pruned Decision Tree Model would have 'hypothetically' done!

```r
#fitting Best Pruned Tree Model to testing data
pokemon_test_augmented2 <- augment(pokemon_tree_final_fit, new_data = pokemon_test)


# Compute accuracy again
accuracy2 <- pokemon_test_augmented2 %>%
  accuracy(truth = TEAM, estimate = .pred_class)

# Compute AUC ROC again
auc_roc2 <- pokemon_test_augmented2 %>%
  roc_auc(truth = TEAM, .pred_FALSE)


results_table2 <- tibble(
  Metric = c("Accuracy", "AUC ROC"),
  Value = c(accuracy2$.estimate, auc_roc2$.estimate)
)

#craeting table of results
results_table2 %>%
  kable()
```

Metric	Value
Accuracy	0.6405530
AUC ROC	0.6707219

Not too bad! While this model performed similarly to our LDA model, it's important to remember that in the long run, our LDA model would likely perform better in predicting accurate results. This is largely due to the k-fold resampling we did earlier. The LDA model consistently showed a higher average of correct predictions compared to the pruned decision tree model.

Conclusions

In this analysis, we compared multiple classification models including: Logistic Regression, Linear Discriminant, Quadratic Discriminant, and Pruned Decision Tree models. Using the Receiver Operating Characteristic Area Under (ROC_AUC) metric to evaluate each model's performance, we found that our Linear Discriminant model performed the best.

To ensure the reliability of our results, we used 10-fold cross-validation. Using this method, we split the data into ten parts, training and assessing the models across different splits. After choosing our model, we applied it to our testing data, and as shown above, its performance was consistent with our k-fold cross-validation results.

If there's any wisdom to learn from this experience, it's that sometimes the simpler model is better! Sometimes a regular old Pokeball will do the job, there's really no point to wasting your masterball on a lvl 5 Magicarp.

In all seriousness though, I was quite surprised to see that out best model turned out to be our Linear Discriminant Model. Until now, I really did undermine the capabilities of this model. While this model did pretty well in my opinion, I wonder how other models, that were not included in this project, would have done.

Its interesting, while I was starting this project, even while finding the data set to work with, I felt very confident to the point where I thought that my model's were going to do very well. It wasn't until I started tidying the data and considering my model parameters, that I realized how much more complicated this process was. Due to this, I believe there may be many ways to further improve this model. For starters, I am very confident that the models would have better predictions had I considered both Pokemon types (`Type 1` and `Type 2`). Likewise, as we noted in our EDA section, a lot of the Pokemon on my team happened to be Legendarys. It would be interesting to see how different these models would work if I were to remove all Legendary Pokemon out of the question.

As much as I hate to admit it, this was honestly a really fun project to tackle (pun very much intended). I enjoyed analyzing my biases when it came to choosing Pokemon to be on my team. While somewhat overwhelming at first, this project was very beneficial to my growth as programmer and data analyst. Throughout this project, I have found myself getting more and more confident using R and all the unique packages with it. Even in moment of struggle and frustration, mainly due to the debugging process, I was able to keep calm and move forward.

I hope this project was helpful to others as much as it is for me; especially to all the new Pokemon trainers playing their first ever Pokemon game hiding under their covers.

Sources

This data was originally inspired and derived from Kaggle called '**Pokemon with Stats'** by *Alberto Barradas.*

ChatGPT for specific lines of code in EDA section