Homework 3

PSTAT 131/231

Aarti Garaye

Homework 5

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: Vulpix, a Fire-type fox Pokémon from Generation 1 (also my favorite Pokémon!)

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. *This is an example of a* ***classification problem****, but these models can also be used for* ***regression problems****.*

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

```
# Setting seed
set.seed(05312025)

# Loading Necessary Libraries
library(tidymodels)
library(tidyverse)
library(ggplot2)
library(dplyr)
```

```r
library(readr)
library(knitr)
library(kableExtra)
library(corrplot)
library(corrr)
library(parsnip)
library(workflows)
library(discrim)
library(kknn)
library(yardstick)
library(glmnet)
library(janitor)
library(naniar)
library(corrplot)
library(themis)
library(forcats)
library(fastDummies)
library(ranger)
library(vip)

# Loading 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?

Solution: I have installed and loaded the `janitor` package in the introductory code chunk, please refer to that.

```
clean_pokemon <- clean_names(Pokemon)

save(clean_pokemon, file = "clean_pokemon.Rds")
```

The data is now saved as an R file in the homework 4 folder. I don't have to run the code again and again, I can just comment it out and load it whenever I want to use it. This reduces the computation time every time we knit the file.

`clean_names()` is a useful tool as it takes the data and returns a data frame which is easier to pipe. As the name suggests it cleans the name, the resulting names are unique and only consist of the `_` character, numbers, and letters. The default of this function returns a data frame that is not case sensitive meaning that it while cleaning the names it will treat "a" same as "A". Accented letters are translated and overall we just get a data frame of unique names which avoids having multiple observations that are the same.

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 grouping them, or "lumping them," together into an 'other' category. Using the `forcats` package, determine how to do this, and **lump all the other levels together except for the top 6 most frequent** (which are Bug, Fire, Grass, Normal, Water, and Psychic).

Convert `type_1` and `legendary` to factors.

Solution: First, let's create a bar chart of the outcome varaible using all of the data set.

```
ggplot(clean_pokemon, aes(x=type_1)) +
  geom_bar() +
  theme_bw() +
  ggtitle("Bar chart of types of Pokémon") +
  labs(x="Types of Pokémon", y = "Frequency") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

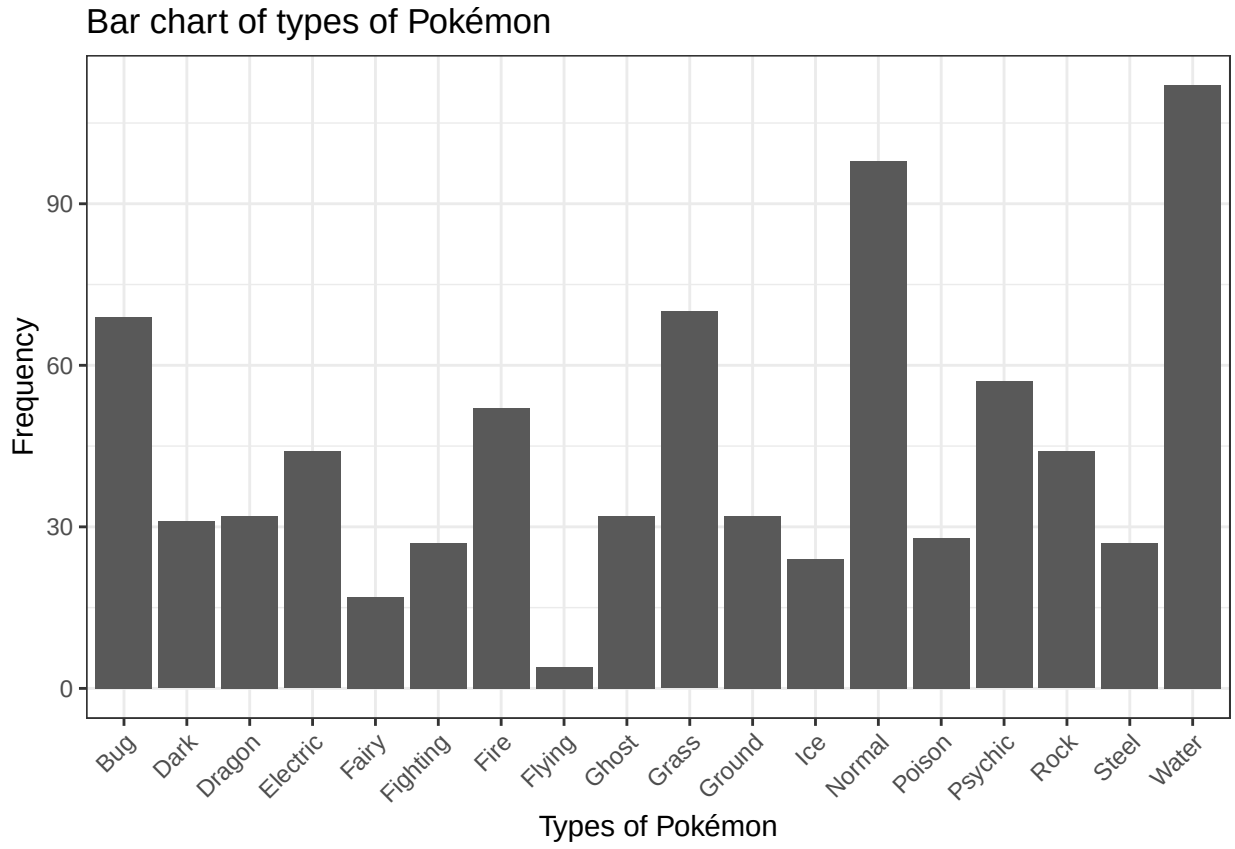

Figure 2: Bar chart of the outcome variable that is the type of Pokémon. It looks like there are a lot of different types and the maximum type is water Pokémon (number of observations in this data set) and the least is flying Pokémon.

There are a lot of classes, if we count them, there are 18 different types of Pokémon. There are significantly more Water Pokémon than Flying Pokémon. Other less popular ones are Fairy, and Ice Pokémon.

After reading the documentation for `forcats` we can use `fct_lump()` to collapse the least frequent species other than the top 6 into others. Now we will have types of Pokémon as Bug, Fire, Grass, Normal, Water, Phychic, and other. Specifically, we will be using `fct_lump_n()` which lumps all levels except for the `n` most frequent species.

```
clean_pokemon$type_1 <- fct_lump_n(clean_pokemon$type_1, n = 6)
ggplot(clean_pokemon, aes(x=type_1)) +
  geom_bar() +
  theme_bw() +
  ggtitle("Bar chart of types of Pokémon") +
  labs(x="Types of Pokémon", y = "Frequency") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

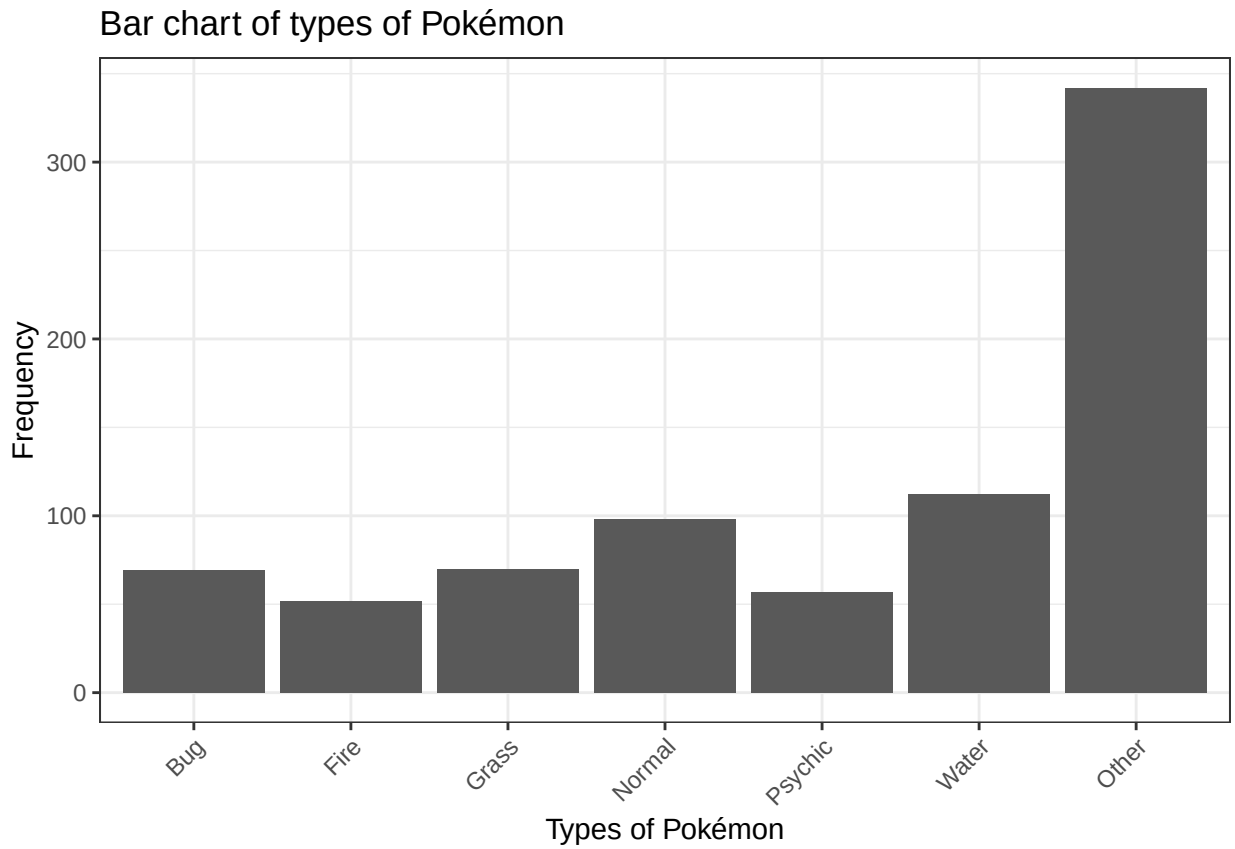

Figure 3: Now there are only six levels for our classification problem since we combined the infrequent species in other.

Lastly, we have to convert `type_1` and `legendary` to factors.

```
clean_pokemon <- clean_pokemon %>%
  mutate(
    type_1 = as.factor(type_1),
    legendary = as.factor(legendary),
  )

is.factor(clean_pokemon$type_1)
```

```
## [1] TRUE
```

```r
is.factor(clean_pokemon$legendary)
```

```
## [1] TRUE
```

We have successfully changed the `type_1` and `legendary` variables as factors and stored it in `clean_pokemon` variable.

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 do you think doing stratified sampling for cross-validation is useful?

Solution: Before splitting the data let's see the total number of observations in the data set.

```
kable(nrow(clean_pokemon), caption = "The total number of observations in the
      Pokémon dataset.")
```

Table 1: The total number of observations in the Pokémon dataset.

x
800

There aren't that many observations to work with so I would go with a 80-20 split. We can, thus, expect to have 640 observations in the training and 160 observations in the test dataset. It's a good thing that we are using a k fold cross validation to ensure that no data is going to waste. Furthermore, the seed it already set in the introductory chunk so we don't have to worry about the reproducibility problem.

```
pokemon_split <- initial_split(clean_pokemon, prop = 0.80, strata = "type_1")

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

kable(nrow(pokemon_train), caption = "As mentioned above the number of
      obersvations in the training data set is around640.")
```

Table 2: As mentioned above the number of obersvations in the training data set is around640.

x
639

```
kable(nrow(pokemon_test), caption = "As mentioned above the number of
      observations in the testing data set is around 160.")
```

Table 3: As mentioned above the number of observations in the testing data set is around 160.

x
161

Now we want to split it in 5 folds for performing cross validation. Stratifying on the outcome variable is very essential especially when we have a lot more Pokémons in the "other" category than in the top six. This

ensures that each fold is accurately representing the training set. The aspect of randomness is maintained because from each strata, it's choosing observations randomly. If we didn't stratify the data while making the folds each fold will have different representation of the population and then the average error wouldn't give us the best estimate of the test error.

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

Exercise 4

Create a correlation matrix of the training set, using the `corrplot` package. *Note: You can choose how to handle the categorical variables for this plot; justify your decision(s).*

What relationships, if any, do you notice?

Solution: One way of handling categorical variables is to dummy code them and include them in the correlation matrix.

```
pokemon_cormat <- pokemon_train %>%
  select(where(is.numeric), type_1, legendary) %>%
  select(-generation) %>%
  dummy_cols(select_columns = c("type_1", "legendary", "generation"),
             remove_selected_columns = TRUE,
             remove_first_dummy = TRUE)

pokemon_cor_matrix <- cor(pokemon_cormat, use = "pairwise.complete.obs")

corrplot(pokemon_cor_matrix, method = "color", type = "lower", diag = FALSE)
```

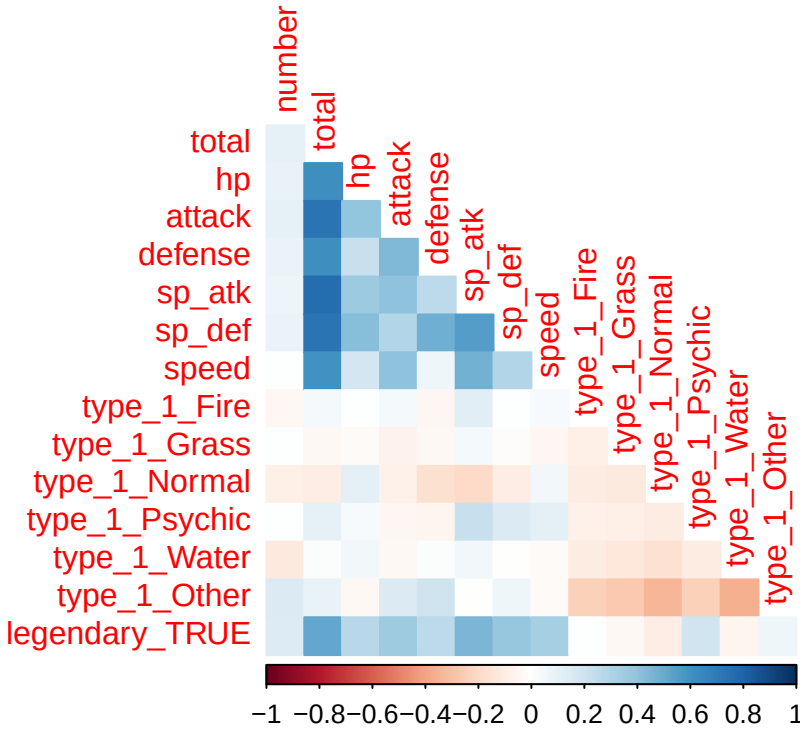

Figure 4: The correlation matrix that handles the categorical variables correctly.

Some moderately high positive correlation can be seen in the attack, special attack, defense, special defense, and hp (which is the hit points, or health i.e. it tells us how much damage can the Pokémon withstand before giving up) and the total. This is understandable because these are the main predictors/influences of the total which is a general guide to how strong a Pokémon is. Other correlation is very mild and nothing to worry about.

Note, we are not including generation because we don't really care about which generation the Pokémon belongs to, we want to explore how we can accurately predict the Pokémon type regardless of the generation

it belongs to. Generation looks like it's numerical but it's actually a categorical variable, so we deselected generation. We could dummy code that but as we said earlier we don't really care about which generation the Pokémon belongs to.

Exercise 5

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.

Solution: Follow the code below to see how the recipe is set.

```
pokemon_recipe <- recipe(type_1 ~ legendary + generation + sp_atk + attack +
                           speed + defense + hp + sp_def,
                         data = pokemon_train) %>%
  step_dummy(legendary) %>%
  step_num2factor(generation, levels = as.character(1:6)) %>%
  step_dummy(generation) %>%
  step_center(all_predictors()) %>%
  step_scale(all_predictors())

prep(pokemon_recipe) %>%
  bake(new_data = pokemon_train) %>%
  head() %>%
  kable(caption = "Data after dummy coding legendary and generation and
        then centering and scaling all the predictors.",
        "latex", booktabs = T) %>%
  kable_styling(latex_options = c("scale_down", "hold_position"))
```

Table 4: Data after dummy coding legendary and generation and then centering and scaling all the predictors.

sp_atk	attack	speed	defense	hp	sp_def	type_1	legendary_TRUE.	generation_X2	generation_X3	generation_X4	generation_X5	generation_X6
0.2179544	-0.4564032	0.3771971	-0.5074669	-0.4433113	-0.2460428	Fire	-0.3005032	-0.3887345	-0.5074155	-0.4123871	-0.517144	-0.3333619
-1.4379055	-1.8356198	-1.3295652	-0.6067640	-0.7540913	-1.7099288	Bug	-0.3005032	-0.3887345	-0.5074155	-0.4123871	-0.517144	-0.3333619
-1.4379055	-1.6788906	-1.1588890	-0.7722592	-0.9483287	-1.7099288	Bug	-0.3005032	-0.3887345	-0.5074155	-0.4123871	-0.517144	-0.3333619
-0.8357746	0.3585885	0.2065209	-1.1032496	-0.1713789	0.3029144	Bug	-0.3005032	-0.3887345	-0.5074155	-0.4123871	-0.517144	-0.3333619
-1.7389710	2.2393384	2.5959882	-1.1032496	-0.1713789	0.3029144	Bug	-0.3005032	-0.3887345	-0.5074155	-0.4123871	-0.517144	-0.3333619
-1.1368401	-1.0519740	-0.4420488	-1.1032496	-1.1425662	-1.3439573	Normal	-0.3005032	-0.3887345	-0.5074155	-0.4123871	-0.517144	-0.3333619

The above table shows a preview of the variables in the recipe dummy coded and centered and scaled.

Exercise 6

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, let `penalty` range from 0.01 to 3 (this is on the `identity_trans()` scale; note that you'll need to specify these values in base 10 otherwise).

Solution: We will be fitting and tuning an elastic net. Let's see how we can set up this model and workflow.

```
# Setting the engine
pokemon_en <- multinom_reg(mode = "classification",
                           engine = "glmnet",
                           penalty = tune(),
                           mixture = tune())

# Setting the workflow
pokemon_en_workflow <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(pokemon_en)

# Setting the grid
pokemon_en_grid <- grid_regular(penalty(range = c(0.01, 3),
                                        trans = identity_trans()),
                                mixture(range = c(0, 1)),
                                levels = 10)
```

Exercise 7

Now set up a random forest model and workflow. Use the `ranger` engine and set `importance = "impurity"`; we'll be tuning `mtry`, `trees`, and `min_n`. Using the documentation for `rand_forest()`, explain in your own words what each of these hyperparameters represent.

Create a regular grid with 8 levels each. You can choose plausible ranges for each hyperparameter. Note that `mtry` should not be smaller than 1 or larger than 8. **Explain why neither of those values would make sense.**

What type of model does `mtry = 8` represent?

Solution: Setting up model for a random forest.

```
# Setting the model and engine
pokemon_rf <- rand_forest(mtry = tune(),
                          trees = tune(),
                          min_n = tune()) %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("classification")

# Setting the workflow
pokemon_rf_workflow <- workflow() %>%
  add_model(pokemon_rf) %>%
  add_recipe(pokemon_recipe)
```

Now that we have set our model with the engine and the workflow, we can discuss what the above code chunk is doing. The mtry parameter is the most influential hyperparameter. Each tree gets a subset of parameters to form the tree on because otherwise all of the trees will follow the greedy approach and split on the same parameters at every stage and that would not result in independent trees. So, `mtry` is a hyperparameter about how many predictors will be available to each tree at each split. Typically $m = \sqrt{p}$ where p is the number of parameters.

The next hyperparameter in the model is tress which is just the number of trees. As the number of trees increase the computation time increases. There's a tradeoff between computation time and test error. At some point if you increase the number of trees it would increase the time exponentially but the test error wouldn't see that significant of a difference.

The last hyperparameter we will be tuning is the `min_n` which is also known as the stopping time or the tree size. Without a min_n a tree can grow a lot and we have to prune it. The min_n tells when each tree must stop, this way we can ensure uniformity in tree size across our random forest.

Now we are to create a regular grid with 8 levels each.

```
pokemon_rf_grid <- grid_regular(mtry(range = c(2,7)),
                                trees(range = c(200,1000)),
                                min_n(range = c(5,20)),
                                levels = 8)
```

The range for the mtry is from 2 to 7. 1 and 8 wouldn't work because if $m = 1$ then every tree only has access to 1 predictor at each split. This would mean we are restricting every split for every tree to just one of the predictors and forcing it to choose that one predictor. This defies the greedy approach that we talked about. On the other hand, when $m = 8$ we are making all the predictors available for split. This results in all the trees making split on the predictor they think explains the most variability. As a result all trees make the first split on the same variable making the trees not independent. It represents a Bagging model when $m = 8$.

Exercise 8

Fit all models to your folded data using `tune_grid()`.

Note: Tuning your random forest model will take a few minutes to run, anywhere from 5 minutes to 15 minutes and up. Consider running your models outside of the .Rmd, storing the results, and loading them in your .Rmd to minimize time to knit. We'll go over how to do this in lecture.

Use `autoplot()` on the results. What do you notice? Do larger or smaller values of `penalty` and `mixture` produce better ROC AUC? What about values of `min_n`, `trees`, and `mtry`?

What elastic net model and what random forest model perform the best on your folded data? (What specific values of the hyperparameters resulted in the optimal ROC AUC?)

Solution: First we will fit the elastic net and tune it. Then we will move on to the random forest. We ran this once and then saved the results to save the rendering time.

```
#pokemon_tune_en <- tune_grid(
#  pokemon_en_workflow,
#  resamples = pokemon_folds,
#  grid = pokemon_en_grid
#)

#save(pokemon_tune_en, file = "pokemon_tune_en.rda")

load("pokemon_tune_en.rda")

autoplot(pokemon_tune_en) + theme_bw()
```

From the plot above we see that generally smaller penalty and mixture values give us a better result. We can use the `show_best()` or the `select_by_one_std_err()` to see the best models which match the computation skill.

```
best_en_pokemon <- select_by_one_std_err(pokemon_tune_en,
                                         penalty,
                                         mixture)
best_en_pokemon
```

```
## # A tibble: 1 x 3
##   penalty mixture .config
##     <dbl>   <dbl> <chr>
## 1    0.01       0 Preprocessor1_Model001
```

As we can see it's a very little penalty and 0 mixture for this model.

```
show_best(pokemon_tune_en)
```

```
## # A tibble: 5 x 8
##   penalty mixture .metric .estimator  mean     n std_err .config
##     <dbl>   <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1    0.01   1     roc_auc hand_till  0.671     5  0.0230 Preprocessor1_Model091
## 2    0.01   0.889 roc_auc hand_till  0.669     5  0.0236 Preprocessor1_Model081
## 3    0.01   0.556 roc_auc hand_till  0.667     5  0.0226 Preprocessor1_Model051
## 4    0.01   0.444 roc_auc hand_till  0.667     5  0.0229 Preprocessor1_Model041
## 5    0.01   0.667 roc_auc hand_till  0.667     5  0.0231 Preprocessor1_Model061
```

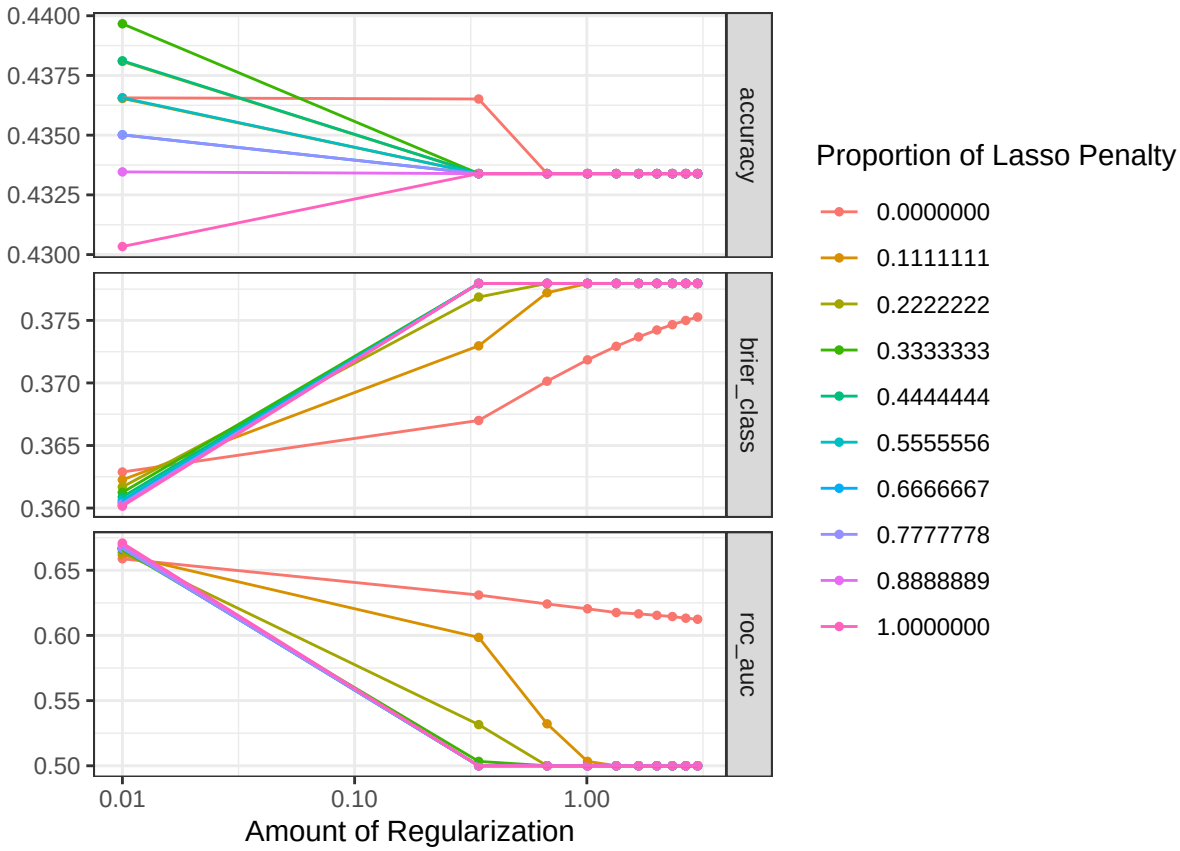

Figure 5: The autoplot shows that all models start with a pretty high area under the roc curve with 0.01 amount of regularization. It might be best to look at accuracy since that's where we see more differences between the models. It looks like the most consistent accuracy throughout the amount of regularization in the red model which has 0 proportion of the lasso penalty, this also has a consistent area under the roc curve.

This shows mix results because the `show_best()` gives same value for the penalty but the mixture it pretty high for these models. However, I think it's in our best interest to go with the model that is simple, gives good results and is computationally simpler which would be the one with lower mixture values.

I ran the `tune_grid()` outside the .rmd file and save it I have added the code snippet but commented it out so it doesn't run every time I render it. Now we move on to the random forest.

```
#pokemon_tune <- tune_grid(
#  pokemon_rf_workflow,
#  resamples = pokemon_folds,
#  grid = pokemon_rf_grid
#)

#save(pokemon_tune, file = "pokemon_rf_tune.rda")

load("pokemon_rf_tune.rda")

autoplot(pokemon_tune) + theme_bw()
```

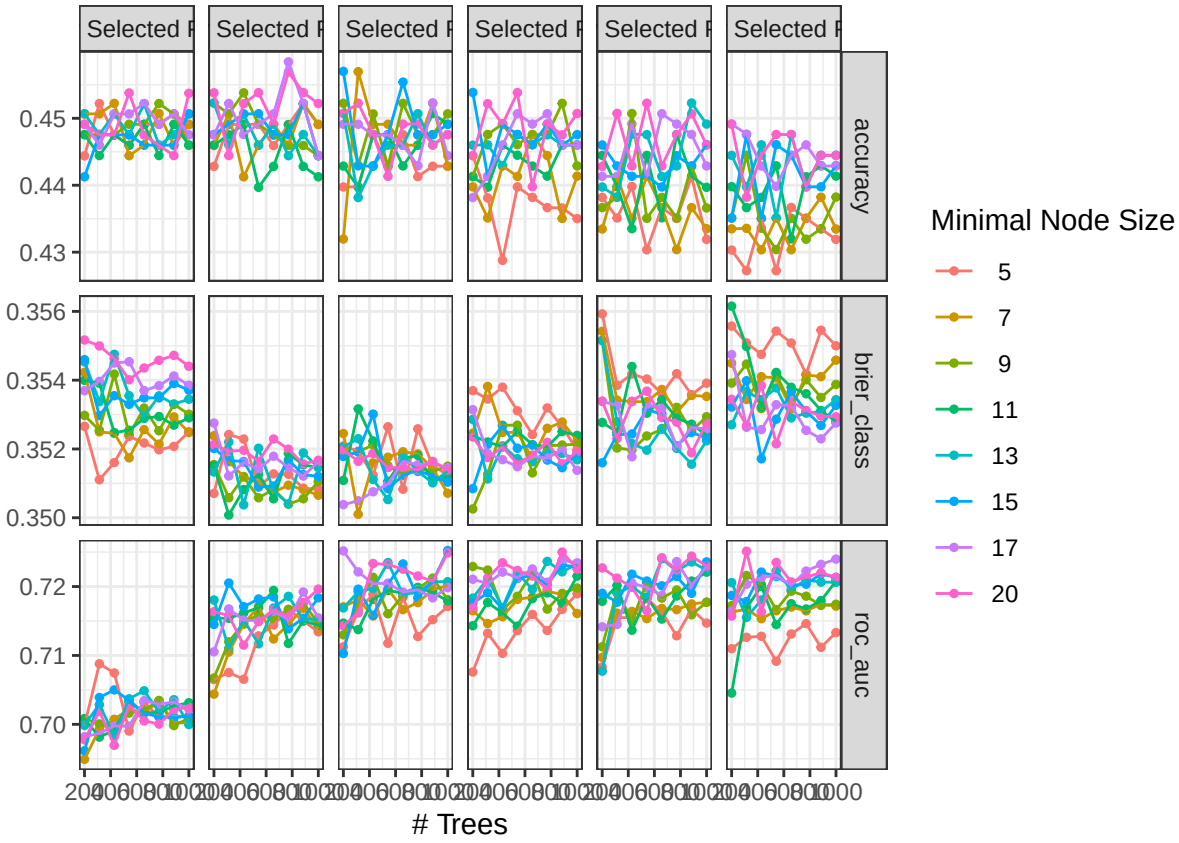

Figure 6: The autoplot is a little harder to see like this, so we change it to be more readable.

```
pokemon_tune %>%
  collect_metrics() %>%
  ggplot(aes(x = mtry, y = mean, color = factor(trees))) +
  geom_line(aes(group = trees)) +
  geom_point() +
```

```
facet_grid(.metric ~ min_n,
           scales = "free_y",
           labeller = labeller(min_n = function(x) paste("Min Node:", x))) +
labs(
  x = "# Randomly Selected Predictors",
  y = "Value",
  color = "# Trees"
) +
theme_bw() +
theme(
  strip.text = element_text(size = 5),
  strip.text.x = element_text(margin = margin(2, 2, 2, 2)),
  legend.position = "right"
)
```

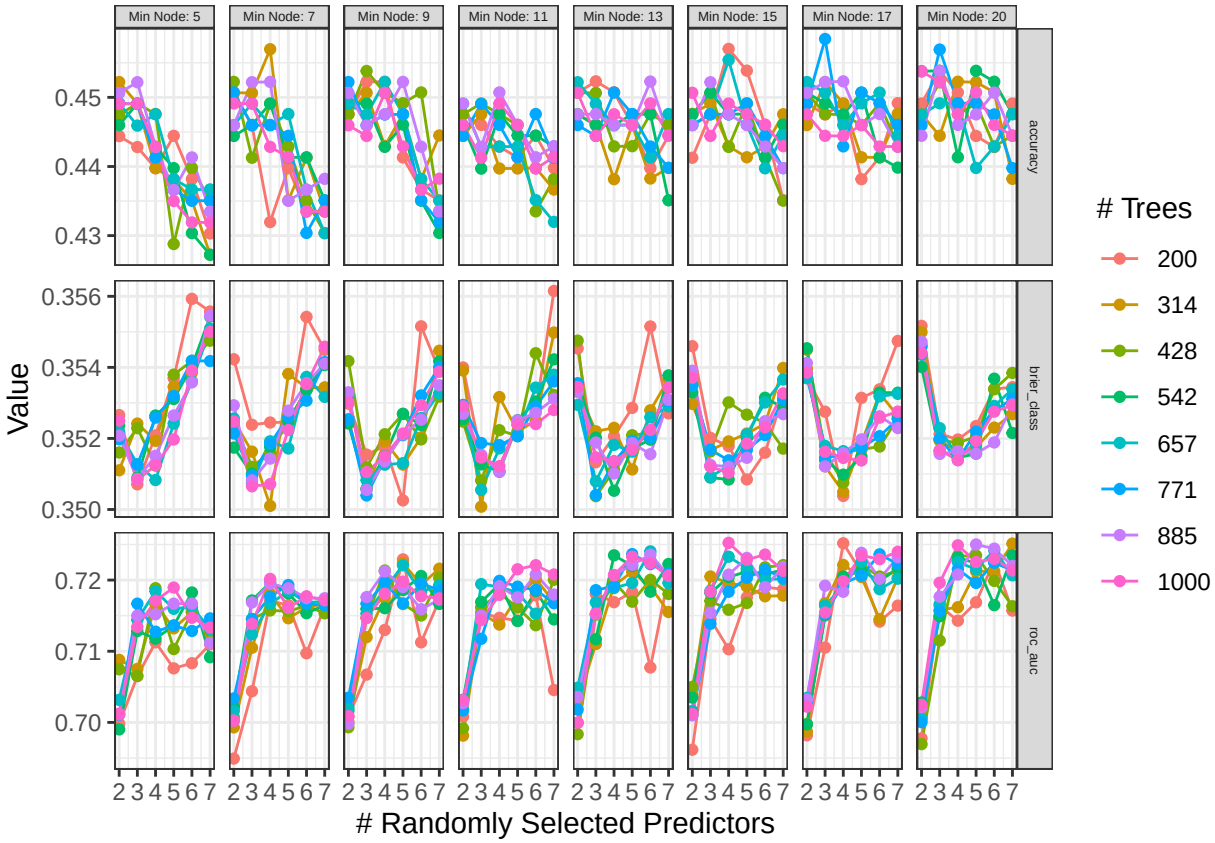

Figure 7: This cleaned up version of autoplot is much easier to read. Just focusing on the area under the roc curve it looks like the best results can be achieved by either 1000 trees adn 15 minimum nodes. Other that comes close is the 200 trees with randomly selected predictors to be 4 and min node is 17.

As the caption says above if we focus on the area under the roc curve, the best mode is the one which uses 200 trees, with minimum node of 17, and 4 randomly selected predictors. This is close to $\sqrt{8} = 2.828$ which is described as generally the better m. We can use the same `select_by_one_std_err()` or `show_best()` to see which one is the best model

```
best_rf_pokemon <- select_by_one_std_err(pokemon_tune,
                                         metric = "roc_auc",
                                         mtry,
                                         trees,
                                         min_n)
best_rf_pokemon
```

```
## # A tibble: 1 x 4
##    mtry trees min_n .config
##   <int> <int> <int> <chr>
## 1     2   314     5 Preprocessor1_Model007
```

According to the select by one standard error the best model is one where $m = 2$ trees are 314 and min_n is 5. Let's see if that's the same for show best.

```
show_best(pokemon_tune)
```

```
## # A tibble: 5 x 9
##    mtry trees min_n .metric .estimator  mean     n std_err .config
##   <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1     4  1000    15 roc_auc hand_till  0.725     5  0.0167 Preprocessor1_Model2~
## 2     4   200    17 roc_auc hand_till  0.725     5  0.0167 Preprocessor1_Model2~
## 3     7   314    20 roc_auc hand_till  0.725     5  0.0171 Preprocessor1_Model3~
## 4     5   885    20 roc_auc hand_till  0.725     5  0.0176 Preprocessor1_Model3~
## 5     4  1000    20 roc_auc hand_till  0.725     5  0.0166 Preprocessor1_Model3~
```

The best one is with $m = 4$ 1000 trees and 15 min_n, this model can be seen in the autoplot. The one we predicted in the autoplot is the second best in this result. However, if the area under the roc curve is not increasing by that much and a lot of computational time is being saved, I would go with the model that select by one standard error gave me.

Exercise 9

Select your optimal **random forest model** in terms of `roc_auc`. Then fit that model to your training set and evaluate its performance on the testing set.

Using the **training** set:

- Create a variable importance plot, using `vip()`. *Note that you'll still need to have set `importance = "impurity"` when fitting the model to your entire training set in order for this to work.*
  - What variables were most useful? Which were least useful? Are these results what you expected, or not?

Using the testing set:

- Create plots of the different ROC curves, one per level of the outcome variable;
- Make a heat map of the confusion matrix.

Solution: As discussed in the above question, the best random forest model is the one with 314 trees, m of 2, and 5 as the min_n. Let's fit that to the traning set and evaluate it's performance on the testing set.

```
# Setting the engine and making the model
final_rf_model <- rand_forest(
  mode = "classification",
  mtry = best_rf_pokemon$mtry,
  trees = best_rf_pokemon$trees,
  min_n = best_rf_pokemon$min_n
) %>%
  set_engine("ranger", importance = "impurity")

# Setting the workflow
final_rf_workflow <- workflow() %>%
  add_model(final_rf_model) %>%
  add_recipe(pokemon_recipe)

# Fitting on the training set
fitted_pokemon_rf <- fit(final_rf_workflow, data = pokemon_train)

# Evaluating on the testing set
pokemon_preds <- augment(fitted_pokemon_rf, new_data = pokemon_test)
pokemon_preds
```

```
## # A tibble: 161 x 21
##    .pred_class .pred_Bug .pred_Fire .pred_Grass .pred_Normal .pred_Psychic
##    <fct>           <dbl>      <dbl>       <dbl>        <dbl>         <dbl>
##  1 Other          0.0325     0.0855      0.129        0.0716        0.0710
##  2 Other          0.123      0.132       0.142        0.0918        0.0484
##  3 Other          0.0261     0.202       0.0857       0.0768        0.197
##  4 Other          0.133      0.0446      0.219        0.0663        0.0273
##  5 Other          0.0680     0.0876      0.0987       0.0531        0.166
##  6 Other          0.0795     0.0745      0.0480       0.255         0.0311
##  7 Other          0.180      0.0415      0.0268       0.311         0.00789
```

```
##  8 Other          0.133      0.102       0.0492       0.185        0.0199
##  9 Other          0.0982     0.0834      0.0670       0.129        0.0368
## 10 Other          0.0595     0.183       0.0333       0.167        0.0316
## # i 151 more rows
## # i 15 more variables: .pred_Water <dbl>, .pred_Other <dbl>, number <dbl>,
## #   name <chr>, type_1 <fct>, type_2 <chr>, total <dbl>, hp <dbl>,
## #   attack <dbl>, defense <dbl>, sp_atk <dbl>, sp_def <dbl>, speed <dbl>,
## #   generation <dbl>, legendary <fct>
```

The prediction seems pretty okay, moving on to making the variable plot we can better evaluate the model.

```
vip(fitted_pokemon_rf$fit$fit) + theme_bw()
```

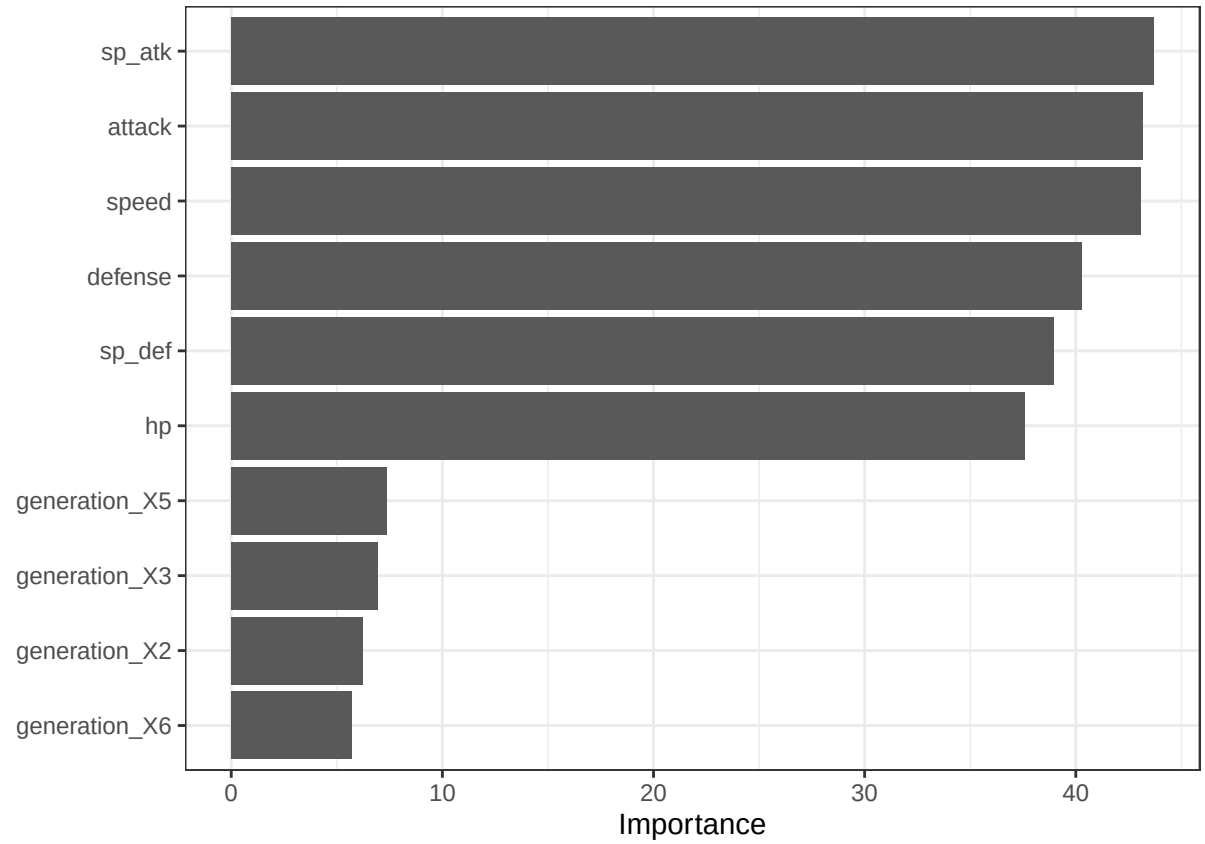

Figure 8: The most important variable in predicting the type of Pokemon or making the splits in the trees is attack and a close second is the special attack variable.

Some of the most useful variables were attack, special attack, speed, defense, special defense, and hp. Some of the least useful were the different generations. This was a very expected outcome because as mentioned in the previous questions, generation is when the games came out it doesn't affect the type of pokemons whereas the attack, speed and defense does.

Let's see us fitting the different roc curves, one per level of the outcome variable and the heatmap and confusion matrix

```
pokemon_preds_fixed <- pokemon_preds %>%
  rowwise() %>%
```

```
  mutate(
    .pred_class = c("Bug", "Fire", "Grass", "Normal", "Psychic",
                    "Water", "Other")[
      which.max(c(.pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal,
                  .pred_Psychic, .pred_Water, .pred_Other))
    ]
  ) %>%
  mutate(.pred_class = factor(.pred_class, levels = levels(type_1)))


pokemon_preds %>%
  roc_curve(truth = type_1, .pred_Bug, .pred_Fire, .pred_Grass,
            .pred_Normal, .pred_Other, .pred_Water, .pred_Psychic) %>%
  autoplot()
```

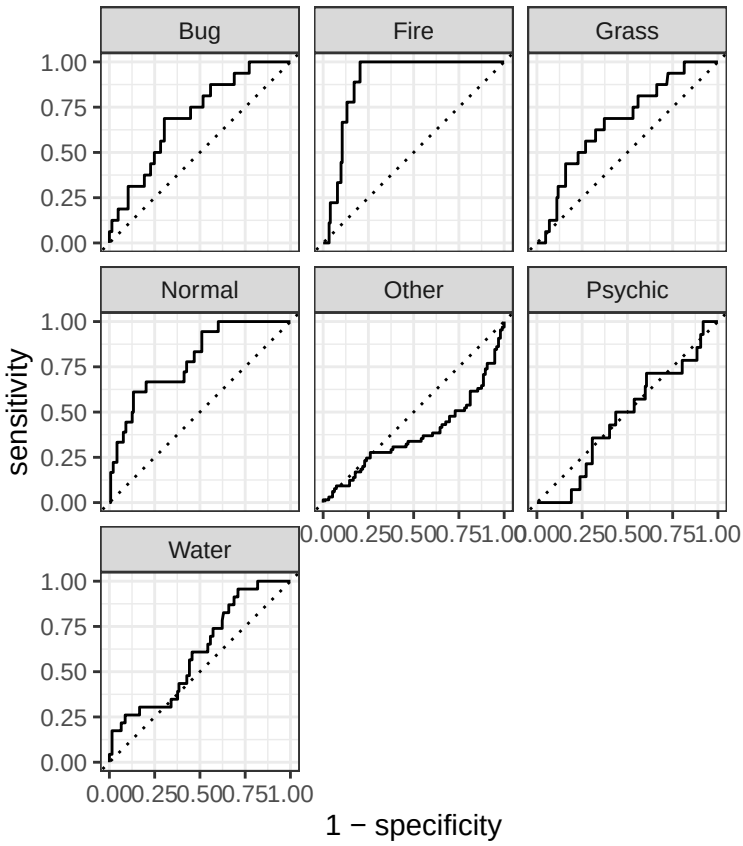

Figure 9: The plots show the area under the curve for different predicted class in the model. As we can see for water, psychic, and other is not doing very well with the model. The model is doing very well with predicting fire type pokemons.

It seems like the mode is doing really well with normal and fire type pokemon. Other types, not so much. Let's see that this is confirmed with the heatmap and the confusion matrix

```
# Now create the confusion matrix
pokemon_preds_fixed %>%
  conf_mat(truth = type_1, estimate = .pred_class) %>%
  autoplot(type = "heatmap")
```

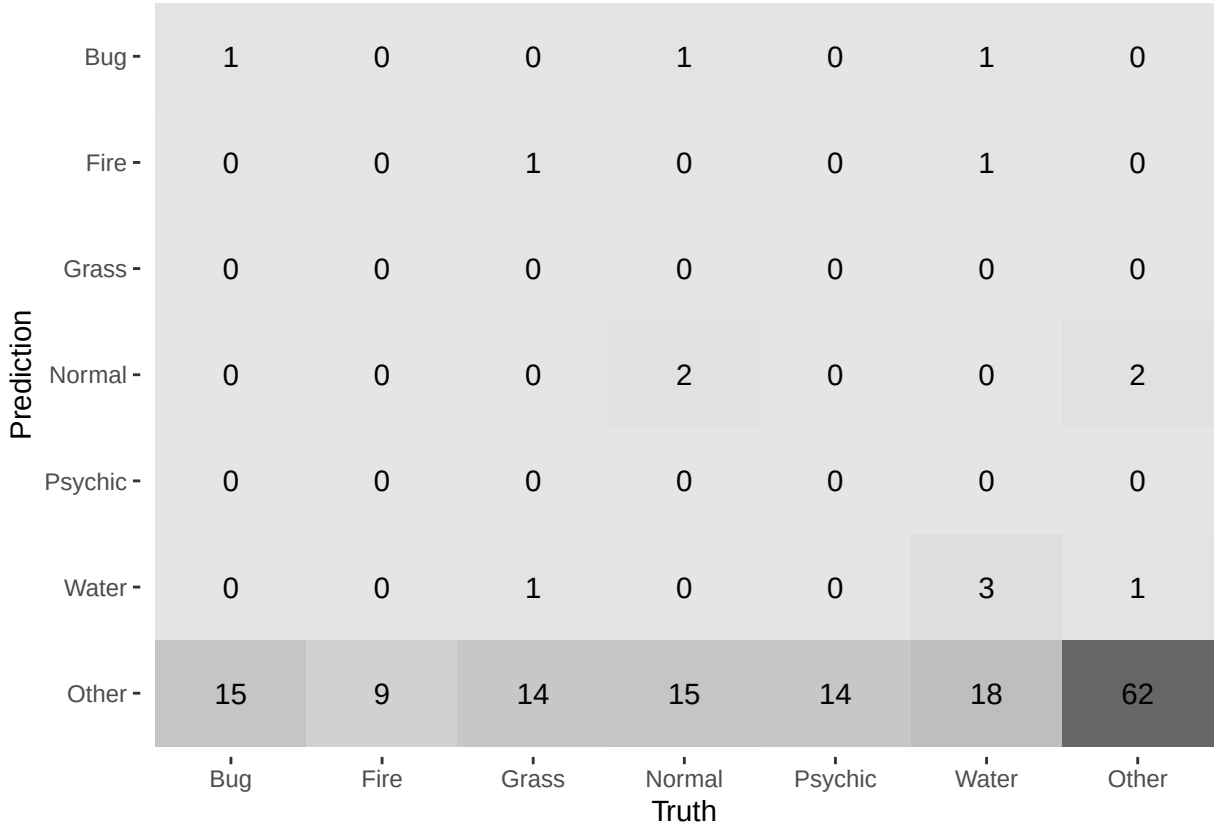

Figure 10: The heatmap of the confusion matrix. It might be confusing why there are so many 0s and 1s but this is not to be confused with how we interpret the matrix. These numbers are the counts and they mean how many it misclassifies so if it was a bug type it misclassified it 15 times as other type.

Exercise 10

How did your best random forest model do on the testing set?

Which Pokemon types is the model best at predicting, and which is it worst at? (Do you have any ideas why this might be?)

Solution: The best model did fairly well on the testing set. Obviously there are a lot of areas where the model could increase. It was really good at predicting fire and normal type of pokemon I think that's because there are a lot of fire and normal type. Obviously other type is really big but recall that it is a collection of all other types. I think it's worst at predicting others, and water type. Let's see how many observations in the testing set for these types

```
pokemon_train %>%
  count(type_1) %>%
  kable(caption = "Number of pokemon in each type in the training set.")
```

Table 5: Number of pokemon in each type in the training set.

type_1	n
Bug	53
Fire	43
Grass	54
Normal	80
Psychic	43
Water	89
Other	277

It looks like the model did fairly well when there were less observations. When there were a lot of observations it didn't perform as well which is really surprising. I would've expected the more observations there are in the training set, the more trained a model could get improving it's area under the roc curve for that particular type.

It might be because the minority class might have some distinct features that the other types don't. There might be some special attacks so special to the fire type that the model quickly picked up how to distinguish these types. It would be really hard to guess others since it is a mix of different types and they all would have variety of special features blended together making it harder to predict.

For 231 Students

Exercise 11

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.

Solution: First we would just make our initial data as 337 makes and 464 misses.

```
makes <- rep(1, 337)
misses <- rep(0, 464)
shots <- c(makes, misses)
```

Now we want to perform 1000 bootstrap resamples and calculating the mean of these

```
n <- 1000
bootstrap_mean <- numeric(n)

for (i in 1:n) {
  sample_i <- sample(shots, size = length(shots), replace = TRUE)
  bootstrap_mean[i] <- mean(sample_i)
}
```

Now we want to plot the histogram of these values

```
hist(bootstrap_mean,
     main = "Bootstrap Distribution of 3PT FG%",
     xlab = "FG%",
     breaks = 15)
```

Now we want to compute the 99% confidence interval using the quantile function.

```
CI <- quantile(bootstrap_mean, probs = c(0.005, 0.995))
CI
```

```
##      0.5%     99.5%
## 0.3757803 0.4669164
```

When we compute a 99% confidence interval, we want to capture the middle 99% of the bootstrap distribution, leaving 0.5% in each tail (totaling 1% outside the interval). So we cut off the lowest 0.5% and the highest 0.5%.

Based on the 1000 bootstrap resamples, the 99% confidence interval for Stephen Curry's three-point shooting in the 2020-21 season is approximately $[0.3782772, 0.4669164]$, meaning we are 99% confident his true FG% lies within that interval, accounting for sampling variability.

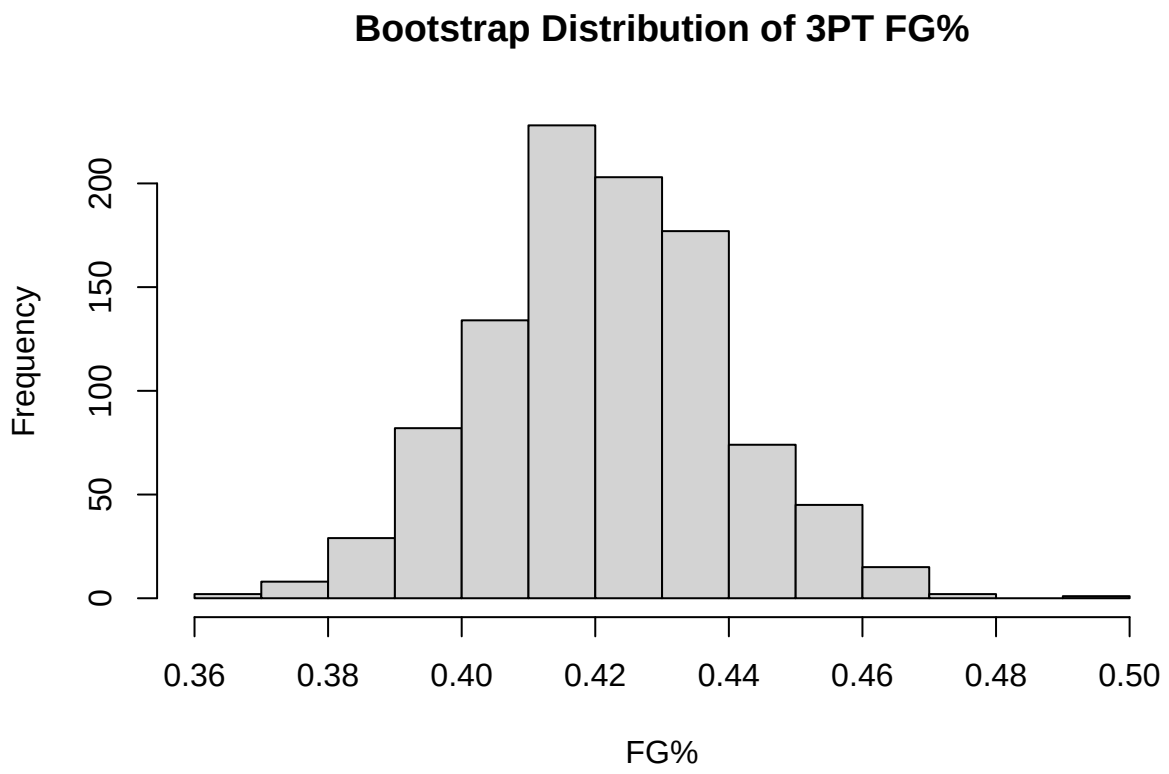

Figure 11: This is the histogram of 1000 brootstrop resamples means of Stephen Currys shots. It looks somewhat like a normal distribution.

Exercise 12

Using the `abalone.txt` data from previous assignments, fit and tune a **random forest** model to predict `age`. Use stratified cross-validation and select ranges for `mtry`, `min_n`, and `trees`. Present your results. What was your final chosen model's **RMSE** on your testing set?

Solution: We have to read the abalone data set first

```
abalone <- read_csv("data/abalone.csv")
abalone <- abalone %>%
  mutate(age = rings + 1.5) %>%
  select(-rings) # we are dropping rings because age is made from rings
```

Splitting the data into an 80-20 splits

```
abalone_splits <- initial_split(abalone, prop = 0.80,
                                strata = age)
abalone_train <- training(abalone_splits)
abalone_test <- testing(abalone_splits)

abalone_folds <- vfold_cv(abalone_train, v = 5)
```

Now we use the same recipe we have been using in the past homeworks.

```
abalone_recipe <- recipe(age ~ ., data = abalone_train) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_interact(terms = ~ starts_with("type_"):shucked_weight) %>%
  step_interact(terms = ~ longest_shell:diameter) %>%
  step_interact(terms = ~ shucked_weight:shell_weight) %>%
  step_center(all_predictors()) %>%
  step_scale(all_predictors())

prep(abalone_recipe) %>%
  bake(new_data = abalone_train) %>%
  head() %>%
  kable(caption =
          "Data with dummy coding, interaction terms,
        centering all predictors, and scaling all predictors.",
        "latex", booktabs = T) %>%
  kable_styling(latex_options = c("scale_down", "hold_position"))
```

Table 6: Data with dummy coding, interaction terms, centering all predictors, and scaling all predictors.

longest_shell	diameter	height	whole_weight	shucked_weight	viscera_weight	shell_weight	age	type_I	type_M	type_I_x_shucked_weight	type_M_x_shucked_weight	longest_shell_x_diameter	shucked_weight_x_shell_weight
-1.4511513	-1.4407632	-1.1676121	-1.2259227	-1.1670877	-1.2033992	-1.2099564	8.5	-0.6916553	1.3112417	-0.5351231	-0.2419070	-1.406696	-0.8856596
-1.6175009	-1.5413905	-1.4023150	-1.2673629	-1.2116960	-1.2848669	-1.3171098	8.5	1.4453740	-0.7624076	0.2286359	-0.6380710	-1.497576	-0.9025407
-0.8273403	-1.0885676	-1.0502606	-0.9712173	-0.9819633	-0.9408925	-0.8527785	9.5	1.4453740	-0.7624076	0.6681173	-0.6380710	-1.039479	-0.8033824
-1.4095639	-1.2898222	-1.2849635	-1.0945271	-1.1871614	-1.2848669	-0.8884962	8.5	1.4453740	-0.7624076	0.2755708	-0.6380710	-1.336423	-0.8529306
-0.4946410	-0.5351174	-0.8155576	-0.7124690	-0.5983320	-0.5199766	-0.8170607	9.5	-0.6916553	1.3112417	-0.5351231	0.2657403	-0.642408	-0.7087077
-2.3660741	-2.3464090	-2.2237754	-1.5402615	-1.4704241	-1.4296981	-1.5671344	6.5	1.4453740	-0.7624076	-0.2663141	-0.6380710	-1.942993	-0.9380179

Now we want to set up the random forest model. Before we would want to set engines, workflow, and the grids.

```
abalone_rf_model <- rand_forest(
  mtry = tune(),
  trees = tune(),
  min_n = tune()
) %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("regression")

abalone_rf_workflow <- workflow() %>%
  add_model(abalone_rf_model) %>%
  add_recipe(abalone_recipe)

abalone_rf_grid <- grid_regular(mtry(range = c(2, 7)), # there are 8 predictors
                                trees(range = c(200,600)),
                                min_n(range = c(10,20)),
                                levels = 5)
```

We tuned the random forset and saved it but now we should load it in

```
#abalone_tune_rf <- tune_grid(
#  abalone_rf_workflow,
#  resamples = abalone_folds,
#  grid = abalone_rf_grid
#)

#save(abalone_tune_rf, file = "abalone_tune_rf.rda")
```

```
load("abalone_tune_rf.rda")
```

Now we want to select the best model and see which parameters work well for predicting age

```
best_rf_abalone <- abalone_tune_rf %>%
  select_best(metric = "rmse")
best_rf_abalone
```

```
## # A tibble: 1 x 4
##    mtry trees min_n .config
##   <int> <int> <int> <chr>
## 1     7   500    17 Preprocessor1_Model095
```

The best model is with m of 7, 500 trees and min_n of 17. Now we are to fit the data.

```
final_rf_abalone_model <- rand_forest(
  mode = "regression",
  mtry = best_rf_abalone$mtry,
  trees = best_rf_abalone$trees,
  min_n = best_rf_abalone$min_n
) %>%
  set_engine("ranger", importance = "impurity")

final_rf_abalone_wf <- workflow() %>%
  add_model(final_rf_abalone_model) %>%
  add_recipe(abalone_recipe)

fitted_abalone_rf <- fit(final_rf_abalone_wf, data=abalone_train)

abalone_pred <- augment(fitted_abalone_rf, new_data=abalone_test)
abalone_pred
```

```
## # A tibble: 837 x 10
##    .pred type  longest_shell diameter height whole_weight shucked_weight
##    <dbl> <chr>         <dbl>    <dbl>  <dbl>        <dbl>          <dbl>
##  1 10.7  M             0.475    0.37   0.125        0.509          0.216
##  2 10.4  M             0.365    0.295  0.08         0.256          0.097
##  3 11.7  F             0.565    0.44   0.155        0.940          0.428
##  4 12.9  F             0.56     0.44   0.14         0.928          0.382
##  5 10.2  F             0.45     0.355  0.105        0.522          0.237
##  6 12.5  F             0.55     0.425  0.135        0.852          0.362
##  7  5.99 I             0.205    0.15   0.055        0.042          0.0255
##  8 12.8  F             0.525    0.425  0.16         0.836          0.354
##  9 10.5  I             0.52     0.41   0.12         0.595          0.238
## 10  6.45 I             0.245    0.19   0.06         0.086          0.042
## # i 827 more rows
## # i 3 more variables: viscera_weight <dbl>, shell_weight <dbl>, age <dbl>
```

So to see what the final chosen model's RMSE was on the testing data we can use the collect_metric

```r
abalone_rmse <- rmse(abalone_pred, truth = age, estimate = .pred)
abalone_rmse
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        2.04
```

So the final RMSE on the test set is 2.04 which I think it pretty low and good for the abalone dataset. In previous homeworks we 2.346574 when we tuned the knn so a random forest did a little better then the knn model.