Pstat 131 Hw 5

Noe Arambula

2022-05-04

Contents

Load Libraries

```r
library(tidymodels)
library(ISLR)
library(ISLR2)
library(tidyverse)
library(discrim)
library(janitor)
library(caret)
library(glmnet)
library(yardstick)


tidymodels_prefer()
```

Set Data and Seed

```r
pokemon <- read_csv("Pokemon.csv")


set.seed(777)
```

Question 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
clean_pokemon <- clean_names(pokemon)
```

The column/variable names of the data were cleaned up so that they are easier to use now. # became number, everything became lowercase, and spaces were changed to . clean_names( ) is useful because it makes the data easier to work with by putting everything in a standard form we can reference each variable easier.

Question 2

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

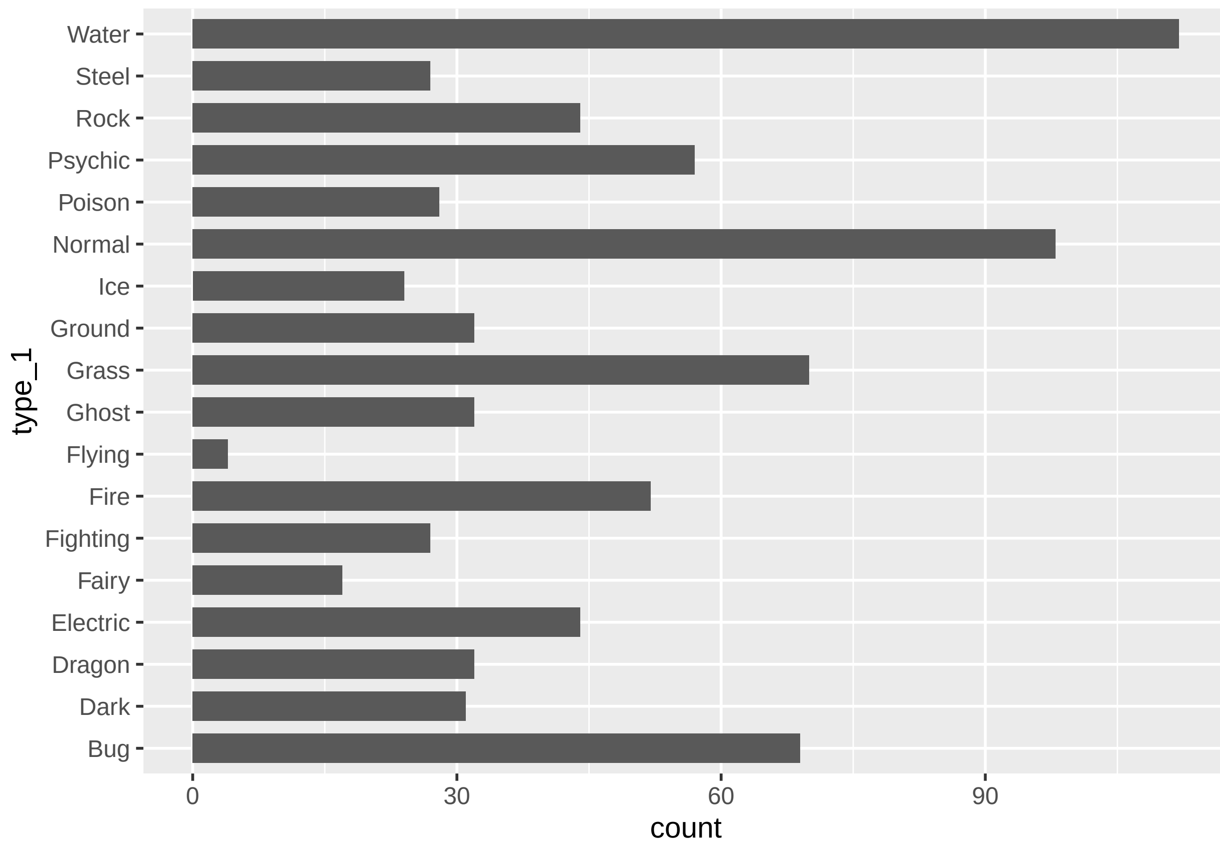
How many classes of the outcome are there? Are there any Pokémon types with very few Pokémon? If so, which ones?

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.

Entire Data set

```r
ggplot(data = clean_pokemon) +
  geom_bar(mapping = aes(y = type_1), width = 0.7)
```



There are 18 different classes of the outcome. There are two types of Pokemon with very few pokemon which are flying type and fairy type with flying having very few ad fairy having a few more but still less than most other types.

Filtering Out The Data

```r
# Filter entire dataset
filtered_pokemon <- filter(clean_pokemon, type_1 == "Bug" | type_1 == "Fire"
                          | type_1 == "Grass" | type_1== "Normal" |
                          type_1 == "Water" | type_1 == "Psychic")

# Convert type_1 and legendary to factors
filtered_pokemon$type_1 <- factor(filtered_pokemon$type_1)
filtered_pokemon$legendary <- factor(filtered_pokemon$legendary)
```

Question 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?

Splitting The Data

```
set.seed(777)
pokemon_split <- initial_split(filtered_pokemon, strata = type_1, prop = 0.8)

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

dim(pokemon_train)
```

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

```
dim(pokemon_test)
```

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

Each data set has approximately the right number of observations, the training data has 364 obs. which is about 80% of the full data set, which contains 458 observations. This leaves the other 20% to the testing data set that has 94/458 observations

V-Fold Cross Validation

```
# Note K-fold and V-fold validation are the same thing

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

Stratifying the folds might be useful because we can get a proportionate amount of each type in the data into each fold. Otherwise, since some types have more pokemon than others, the folds could misrepresent the actual data population for each type_1.

Question 4

Set up a recipe to predict `type_1` with `legendary`, `generation`, `sp_atk`, `attack`, `speed`, `defense`, `hp`, and `sp_def`.

- Dummy-code `legendary` and `generation`;
- Center and scale all predictors.

```
pokemon_recipe <- recipe(type_1 ~ legendary + generation + sp_atk + attack
                         + speed + defense + hp + sp_def, data = pokemon_train) %>%
  step_dummy(legendary, generation) %>%   # creates dummy variables
  step_normalize(all_predictors())    # Centers and Scales all variables

pokemon_recipe
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          8
##
## Operations:
##
## Dummy variables from legendary, generation
## Centering and scaling for all_predictors()
```

Question 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?

```
# glmnet model
net_spec <- multinom_reg(penalty = tune(), mixture = tune()) %>%
  set_mode('classification') %>%  # note classification is the only option for mode
  set_engine('glmnet')

# workflow
net_workflow <- workflow( ) %>%
  add_recipe(pokemon_recipe) %>%
  add_model(net_spec)

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

```
## # A tibble: 100 x 2
##         penalty mixture
##           <dbl>   <dbl>
##  1      0.00001       0
##  2      0.000129      0
##  3      0.00167       0
##  4      0.0215        0
##  5      0.278         0
##  6      3.59          0
##  7     46.4           0
##  8    599.            0
##  9   7743.            0
## 10 100000             0
## # ... with 90 more rows
```

I will be fitting a total of 100 models when I fit these models to the folded data sets. I will fit 100 models to 5 different data subsets.

Question 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(
  net_workflow,
  resamples = pokemon_fold,
  grid = grid
)
```

```
## ! Fold1: preprocessor 1/1: The following variables are not factor vectors and wil...
```

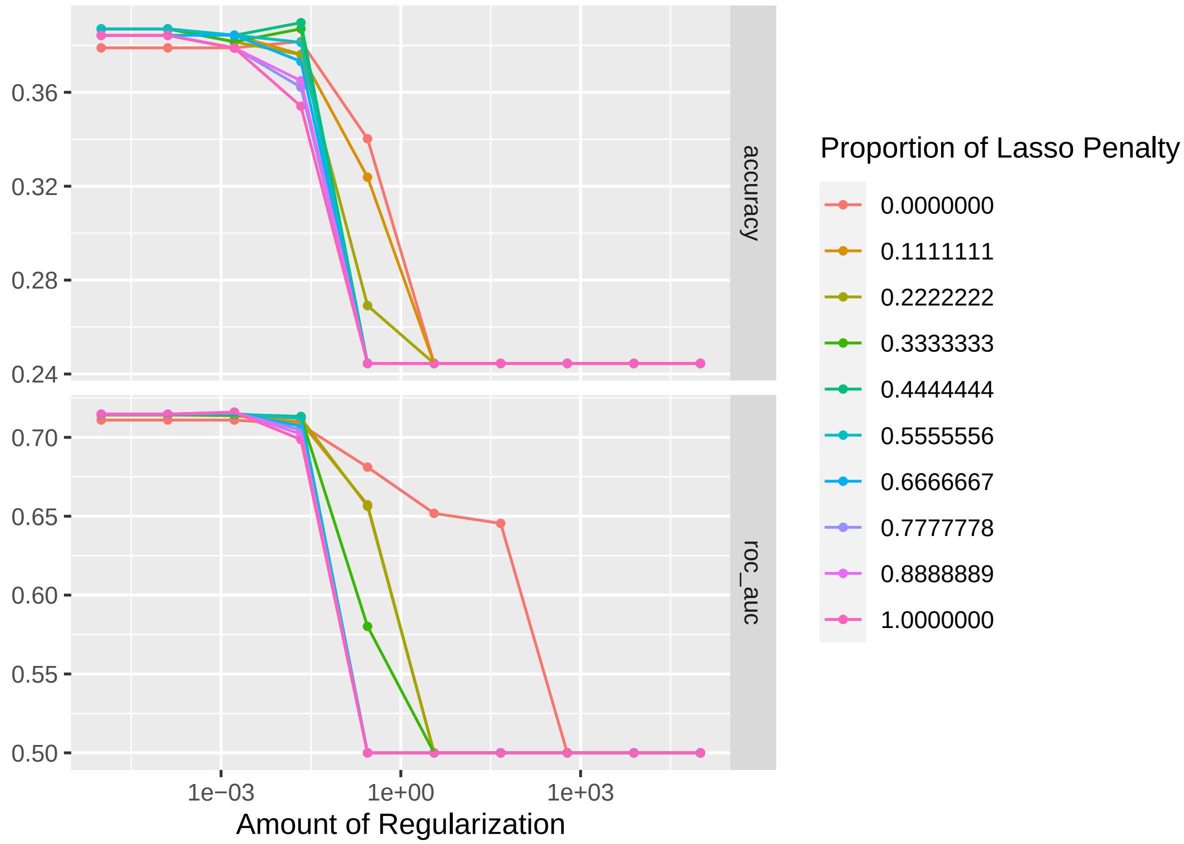
```
## ! Fold2: preprocessor 1/1: The following variables are not factor vectors and wil...
```

```
## ! Fold3: preprocessor 1/1: The following variables are not factor vectors and wil...
```

```
## ! Fold4: preprocessor 1/1: The following variables are not factor vectors and wil...
```

```
## ! Fold5: preprocessor 1/1: The following variables are not factor vectors and wil...
```

```
autoplot(tune_res)
```



I Notice that the accuracy and ROC AUC drop significantly after about 0.01. So, smaller values of penalty and mixture produce better accuracy and ROC AUC

Question 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.

```
# pick best model
best_model <- select_best(tune_res, metric = "roc_auc")
best_model
```

```
## # A tibble: 1 x 3
##   penalty mixture .config
##     <dbl>   <dbl> <chr>
## 1 0.00167   0.889 Preprocessor1_Model083
```

```
# Fit model to training set and evaluate performance
net_final <- finalize_workflow(net_workflow, best_model)

net_final_fit <- fit(net_final, data = pokemon_train)
```

```
## Warning: The following variables are not factor vectors and will be ignored:
## ‘generation‘
```

```
augment(net_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     0.351
```

Question 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.

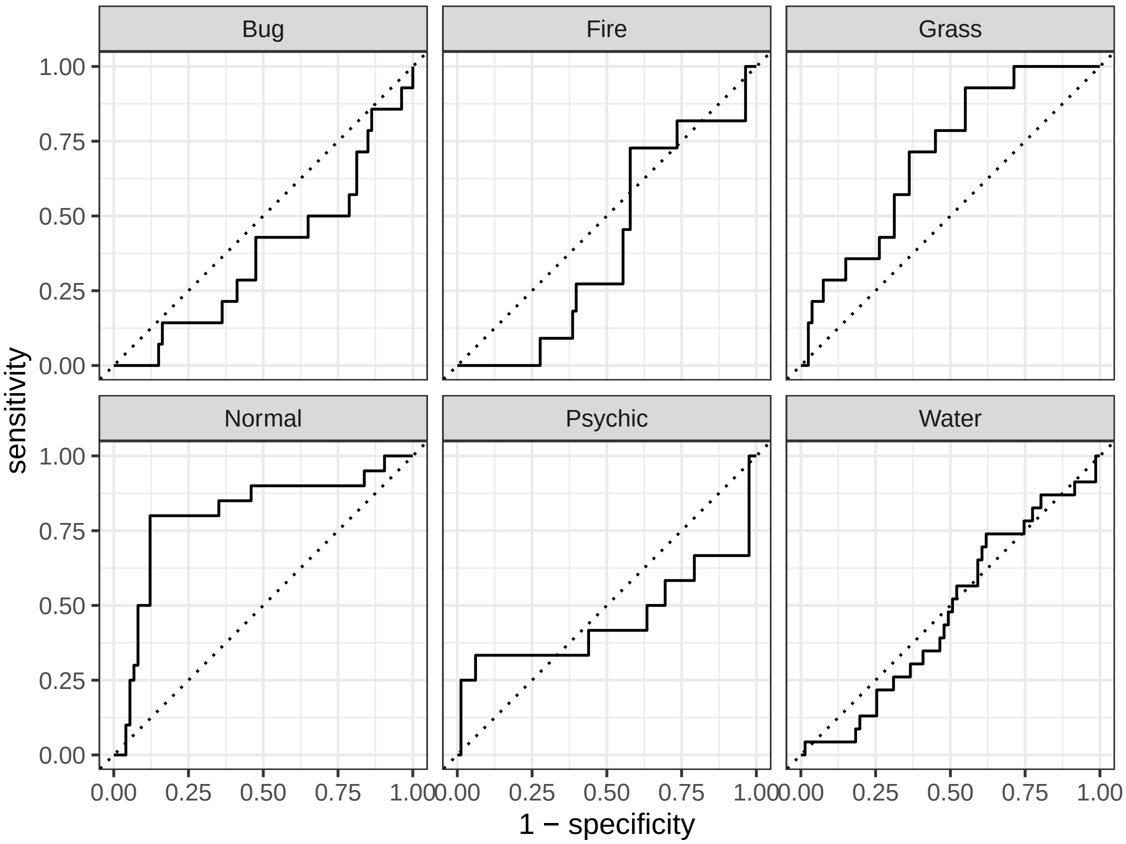
What do you notice? How did your model do? Which Pokemon types is the model best at predicting, and which is it worst at? Do you have any ideas why this might be?

ROC and AUC

```
# Calculate overall ROC AUC
augment(net_final_fit, new_data = pokemon_test) %>%
  roc_auc(type_1, .pred_Fire, .pred_Bug, .pred_Grass, .pred_Normal,
            .pred_Water, .pred_Psychic)
```

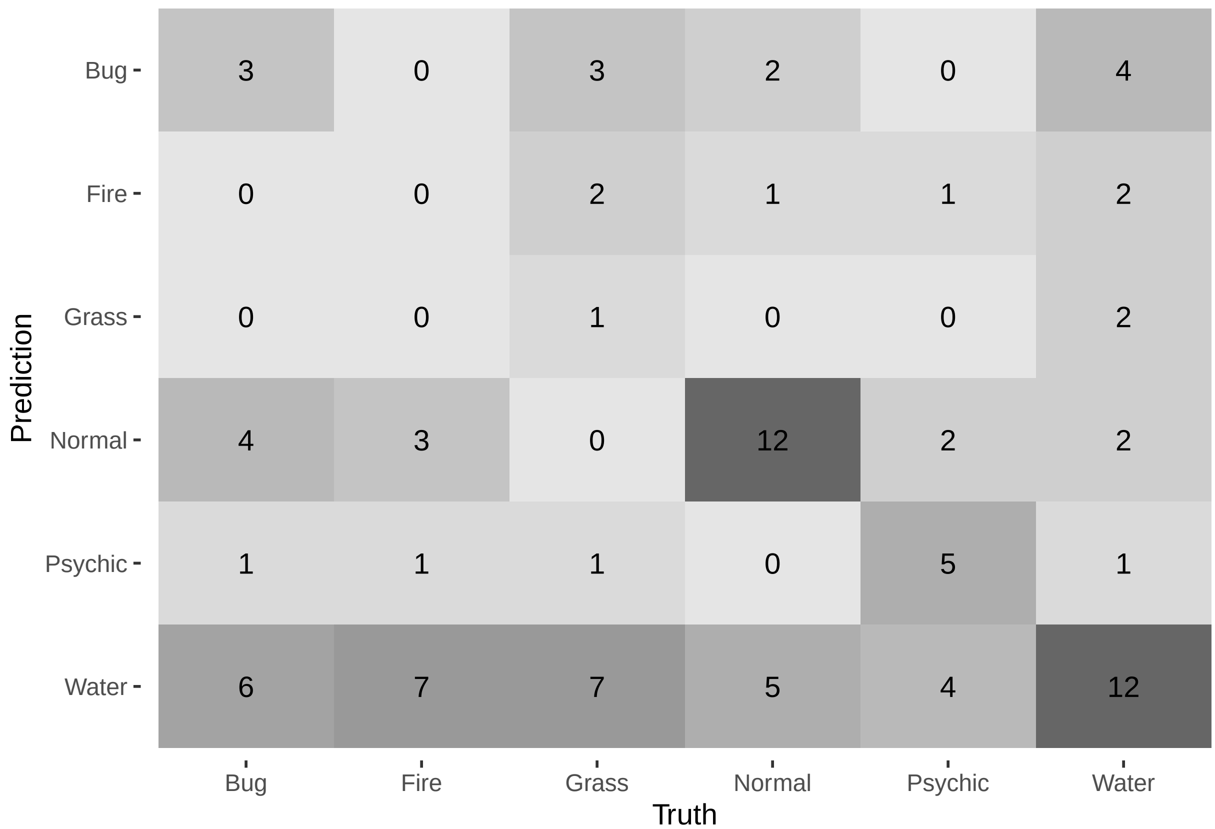
```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.533
```

```
# PLot ROC curve
augment(net_final_fit, new_data = pokemon_test) %>%
  roc_curve(type_1, .pred_Fire, .pred_Bug, .pred_Grass, .pred_Normal,
            .pred_Water, .pred_Psychic) %>%
  autoplot()
```



Confusion Matrix

```
augment(net_final_fit, new_data = pokemon_test) %>%
  conf_mat(truth = type_1, estimate = .pred_class) %>%
  autoplot(type = "heatmap")
```



I noticed that the ROC curves and confusion matrix varies vastly from type to type. Overall, the model was not great at predicting types. The model did not perform very well.

The model best predicts normal type pokemon the best as is shown in the ROC curve and since it has the best truths predicted. Water types are also predicted well according to the confusion matrix however, the ROC curve is just about as good as chance. I believe the model might be best at predicting normal type pokemon because that was the type with the most pokemon in it in the data. The model was worst at predicting fire and grass types looking at the confusion matrix. Although, the ROC curve for grass was good. I do not have any good explanation as to why this might be other than that there were less observations for fire and grass.