hw5

Yiting Zhang

2022-05-15

Elastic Net Tuning

For this assignment, we will be working with the file `"pokemon.csv"`, found in `/data`. The file is from Kaggle: https://www.kaggle.com/abcsds/pokemon.

The Pokémon franchise encompasses video games, TV shows, movies, books, and a card game. This data set was drawn from the video game series and contains statistics about 721 Pokémon, or "pocket monsters." In Pokémon games, the user plays as a trainer who collects, trades, and battles Pokémon to (a) collect all the Pokémon and (b) become the champion Pokémon trainer.

Each Pokémon has a primary type (some even have secondary types). Based on their type, a Pokémon is strong against some types, and vulnerable to others. (Think rock, paper, scissors.) A Fire-type Pokémon, for example, is vulnerable to Water-type Pokémon, but strong against Grass-type.

The goal of this assignment is to build a statistical learning model that can predict the **primary type** of a Pokémon based on its generation, legendary status, and six battle statistics.

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

```r
pokemon_data <- read.csv("Pokemon.csv",fileEncoding = "UTF8")
# view(pokemon_data)
```

Exercise 1

Install and load the `janitor` package. Use its `clean_names()` function on the Pokémon data, and save the results to work with for the rest of the assignment. What happened to the data? Why do you think `clean_names()` is useful?

```r
library(janitor)
pokemon <- pokemon_data %>%
  clean_names()
head(pokemon)
```

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

```
## 2    60          1      False
## 3    80          1      False
## 4    80          1      False
## 5    65          1      False
## 6    80          1      False
```

clean_names () is used on data.frame -like objects. We can see that the clean_names() function converts the some variable names according to certain conventions for names. Here, what it does are removing all uppercase in the variable names and making the resulting names consist only of the _character(instead of ..), numbers and letters.

This is very useful because we can easily identify a variable name as they follow the certain conventions and we won't be confused by the uppercase or lowercase.

Exercise 2

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

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

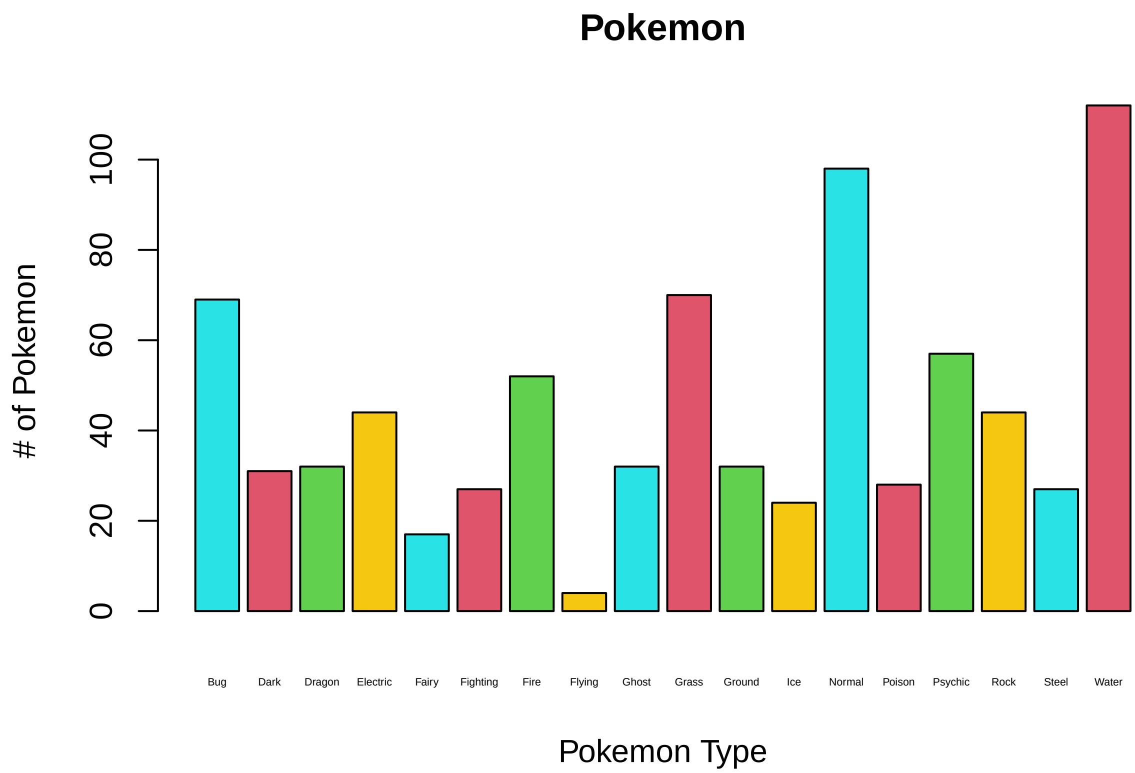
For this assignment, we'll handle the rarer classes by simply filtering them out. Filter the entire data set to contain only Pokémon whose `type_1` is Bug, Fire, Grass, Normal, Water, or Psychic.

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

```r
# plot bar chart
type1 <- table(pokemon$type_1)
type1
```

```
##
##      Bug     Dark   Dragon Electric    Fairy Fighting     Fire   Flying
##       69       31       32       44       17       27       52        4
##    Ghost    Grass   Ground      Ice   Normal   Poison  Psychic     Rock
##       32       70       32       24       98       28       57       44
##    Steel    Water
##       27      112
```

```r
barplot(type1, xlab = "Pokemon Type", ylab = "# of Pokemon",
        main = "Pokemon",width = 0.1,
        cex.names = 0.3,col = c(5,2,3,7))
```




```
pokemon %>%
  group_by(type_1) %>%
  summarise(count = n()) %>%
  arrange(desc(count))
```

```
## # A tibble: 18 x 2
##    type_1   count
##    <chr>    <int>
##  1 Water      112
##  2 Normal      98
##  3 Grass       70
##  4 Bug         69
##  5 Psychic     57
##  6 Fire        52
##  7 Electric    44
##  8 Rock        44
##  9 Dragon      32
## 10 Ghost       32
## 11 Ground      32
## 12 Dark        31
## 13 Poison      28
## 14 Fighting    27
## 15 Steel       27
## 16 Ice         24
## 17 Fairy       17
## 18 Flying       4
```

From the plot and the table, there are 18 outcomes. Pokémon types 'flying' and 'fairy' have very few Pokémon.

```
# filter type_1
filtered_pokemon_types <- pokemon %>%
   filter(type_1 == "Bug" | type_1 == "Fire" |
            type_1 == "Grass" | type_1 == "Normal" |
            type_1 == "Water" | type_1 == "Psychic")
```

```
# check filtered pokemon types
filtered_pokemon_types %>%
  group_by(type_1) %>%
  summarise(count = n()) %>%
  arrange(desc(count))
```

```
## # A tibble: 6 x 2
##   type_1  count
##   <chr>   <int>
## 1 Water     112
## 2 Normal     98
## 3 Grass      70
## 4 Bug        69
## 5 Psychic    57
## 6 Fire       52
```

```
# convert `type_1` and `legendary` to factors
pokemon_factored <- filtered_pokemon_types %>%
  mutate(type_1 = factor(type_1)) %>%
  mutate(legendary = factor(legendary)) %>%
  mutate(generation = factor(generation))
```

Exercise 3

Perform an initial split of the data. Stratify by the outcome variable. You can choose a proportion to use. Verify that your training and test sets have the desired number of observations.

Next, use v-fold cross-validation on the training set. Use 5 folds. Stratify the folds by `type_1` as well. *Hint: Look for a `strata` argument.* Why might stratifying the folds be useful?

```
set.seed(100)
# initial split of the data
pokemon_split <- initial_split(pokemon_factored, strata = type_1, prop = 0.7)
pokemon_train <- training(pokemon_split)
pokemon_test <- testing(pokemon_split)

dim(pokemon_train) #318 observations
```

```
## [1] 318  13
```

```
dim(pokemon_test) #140 observations
```

```
## [1] 140  13
```

```r
# *v*-fold cross-validation
pokemon_fold <- vfold_cv(pokemon_train, strata = type_1, v = 5)
pokemon_fold
```

```
## #  5-fold cross-validation using stratification
## # A tibble: 5 x 2
##   splits           id
##   <list>           <chr>
## 1 <split [252/66]> Fold1
## 2 <split [253/65]> Fold2
## 3 <split [253/65]> Fold3
## 4 <split [256/62]> Fold4
## 5 <split [258/60]> Fold5
```

By using a proportion of 0.7, we can verify that there are 318 observations in the training set and 140 observations in the testing set. Stratifying on the folds is useful because it helps to make sure that in each fold the data is trained with the same distribution of the types of pokemon.Thus, stratifying on type_1 will help us to get a fair fold to train our model better for a better prediction for the future data.

Exercise 4

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

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

```r
# recipe
pokemon_recipe <- recipe(type_1 ~ legendary + generation +
                           sp_atk + attack + speed + defense +
                           hp + sp_def, data = pokemon_train) %>%
  step_dummy(legendary, generation) %>%
  step_center(all_predictors()) %>%
  step_scale(all_predictors())
pokemon_recipe
```

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

Exercise 5

We'll be fitting and tuning an elastic net, tuning `penalty` and `mixture` (use `multinom_reg` with the `glmnet` engine).

Set up this model and workflow. Create a regular grid for `penalty` and `mixture` with 10 levels each; `mixture` should range from 0 to 1. For this assignment, we'll let `penalty` range from -5 to 5 (it's log-scaled).

How many total models will you be fitting when you fit these models to your folded data?

```
# set up model
pokemon_net <- multinom_reg(penalty = tune(),
                            mixture = tune()) %>%
  set_engine("glmnet")

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

# regular grid 'penalty' and 'mixture'
pokemon_grid <- grid_regular(penalty(range = c(-5,5)),
                             mixture(range = c(0,1)),
                             levels = 10)
pokemon_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
```

500 total models will be fitting when fit these models to our folded data by fitting 100 different penalty and mixture combinations 5 times each fold.

Exercise 6

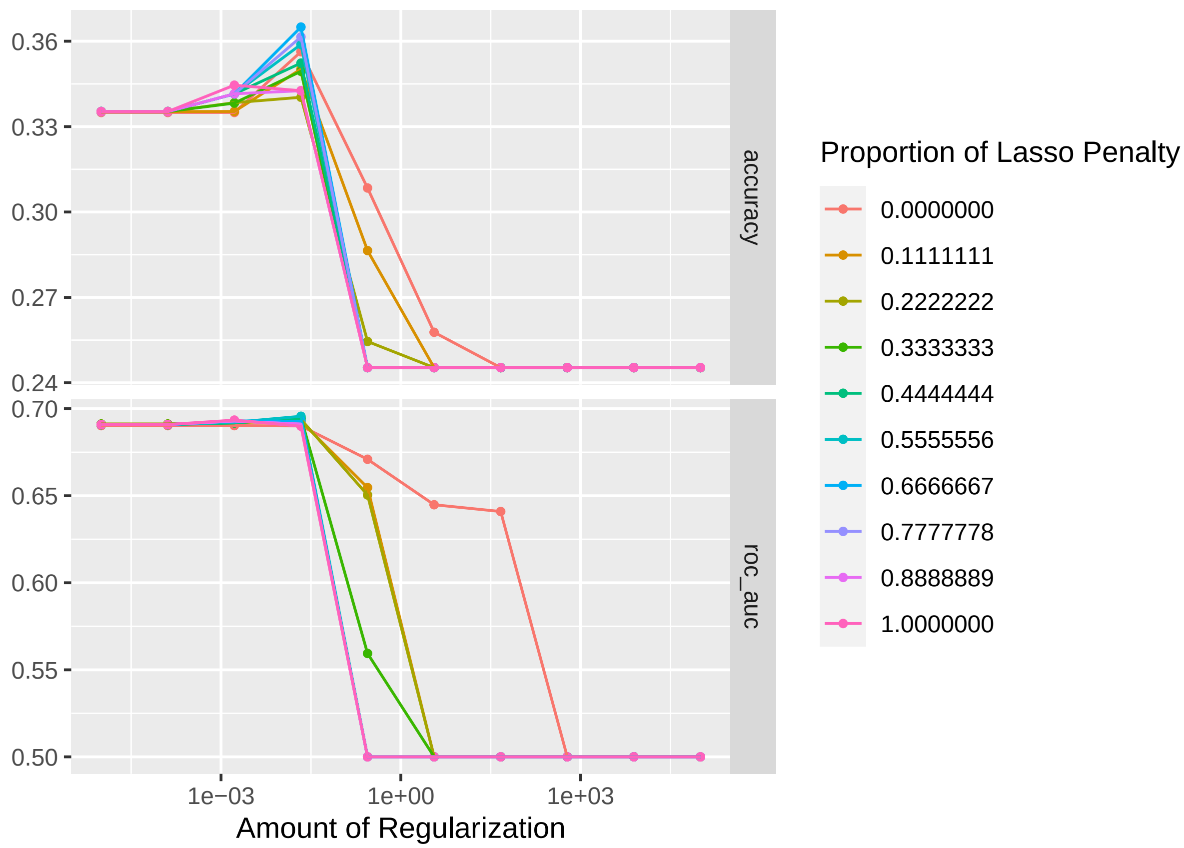
Fit the models to your folded data using `tune_grid()`.

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

```
# use 'tune_grid()' to fit the models to folded data
pokemon_tune_grid <- tune_grid(pokemon_workflow,
                               resamples = pokemon_fold,
```

```r
                                  grid = pokemon_grid)

# autoplot the results
autoplot(pokemon_tune_grid)
```



I noticed that smaller values of accuracy and mixture produce better accuracy. Smaller values of `penalty` and `mixture` produce better accuracy and ROC AUC.

Exercise 7

Use `select_best()` to choose the model that has the optimal `roc_auc`. Then use `finalize_workflow()`, `fit()`, and `augment()` to fit the model to the training set and evaluate its performance on the testing set.

```r
# choose the model
best_model <- select_best(pokemon_tune_grid,
                          metric = "roc_auc")
best_model
```

```
## # A tibble: 1 x 3
##   penalty mixture .config
##     <dbl>   <dbl> <chr>
## 1  0.0215   0.556 Preprocessor1_Model054
```

```r
# fit the model to the training set
pokemon_finalized_model <- finalize_workflow(pokemon_workflow,
                                             best_model)

pokemon_final_fit <- fit(pokemon_finalized_model,
                         data = pokemon_train)

prediction <- augment(pokemon_final_fit,
                      new_data = pokemon_test) %>%
  select(type_1, .pred_class, .pred_Bug, .pred_Fire, .pred_Grass,
         .pred_Normal, .pred_Psychic, .pred_Water)

# evaluate the performance
accuracy(prediction, type_1,.pred_class)
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.357
```

Performs not very good.

Exercise 8

Calculate the overall ROC AUC on the testing set.

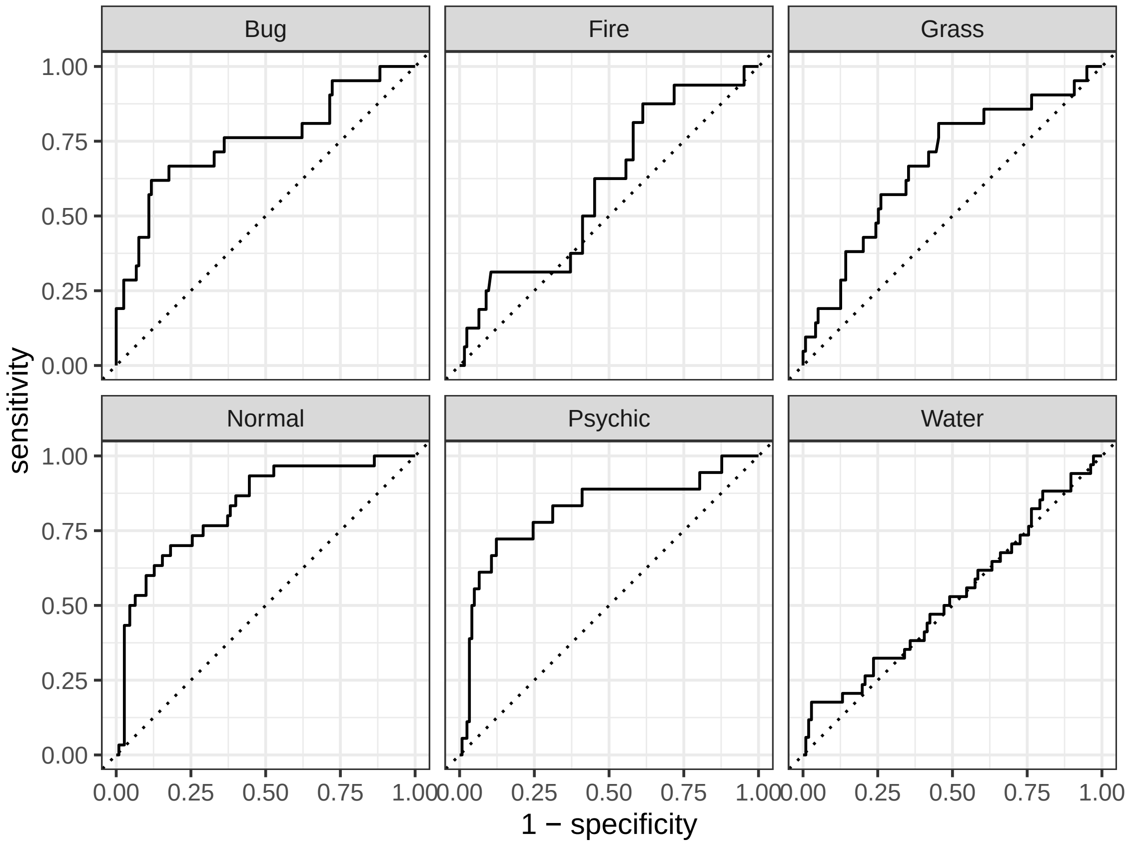
Then create plots of the different ROC curves, one per level of the outcome. Also make a heat map of the confusion matrix.

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

```r
# get overall roc_auc value
augment(pokemon_final_fit, new_data = pokemon_test) %>%
  roc_auc(truth = type_1, estimate = c(.pred_Bug,
                                       .pred_Fire, .pred_Grass, .pred_Normal,
                                       .pred_Psychic, .pred_Water))
```

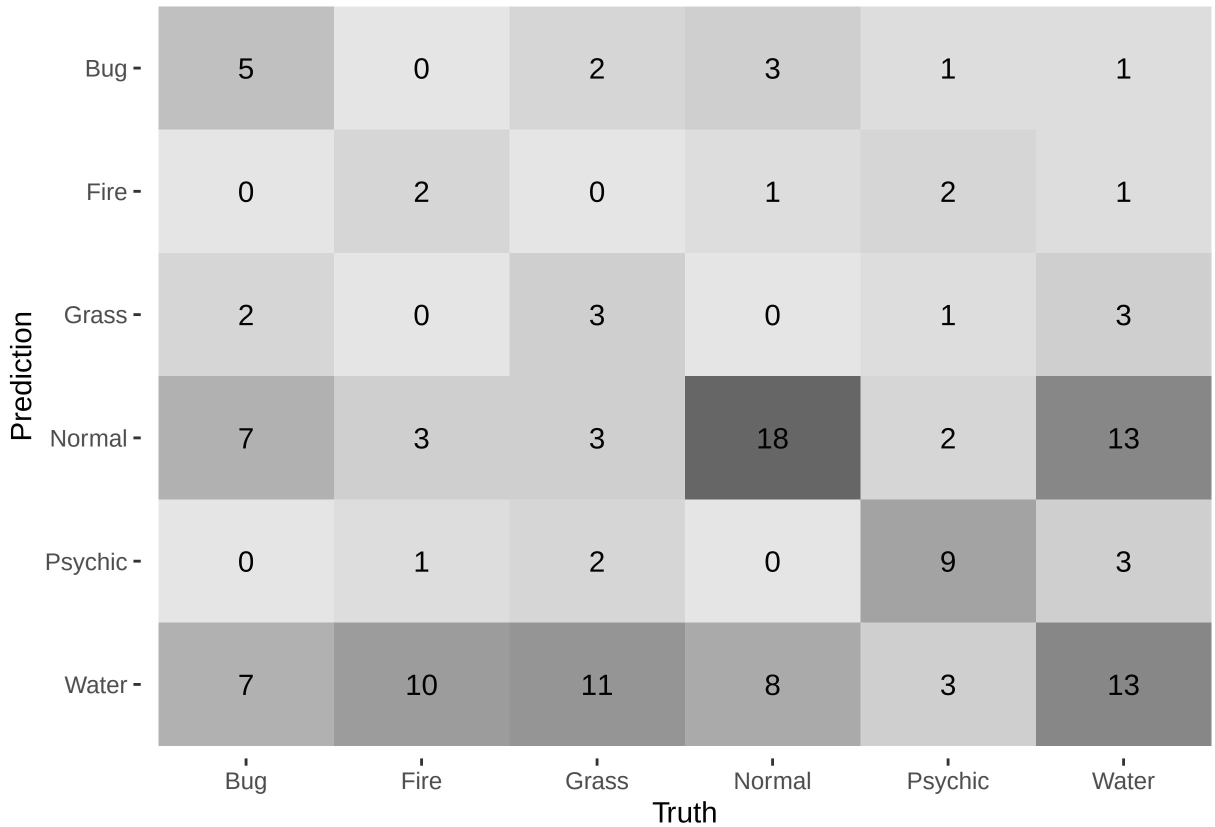
```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.700
```

```r
#plots of different roc curves
prediction %>%
  roc_curve(type_1, .pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal, .pred_Psychic, .pred_Water) %>%
  autoplot()
```



- The Pokemon type that it is best at predicting is Normal, and the Pokemon type that it is worst at predicting is Water.
- Probably because that we can see from the confusion matrix below, there are only few predictions that are normal with the true values being not normal while there are high number of prediction of Normal where Normal is the true value.
- And when it comes to Water, it is the worst at predicting because it tends to predict high numbers of other types that are not water as being water.

```
prediction %>%
  conf_mat(type_1, .pred_class) %>%
  autoplot(type = "heatmap")
```


Prediction
Bug
Fire
Grass
Normal
Psychic
Water
5 0 2 3 1 1
0 2 0 1 2 1
2 0 3 0 1 3
7 3 3 18 2 13
0 1 2 0 9 3
7 10 11 8 3 13
Bug
Fire
Grass
Normal
Psychic
Water
Truth