Pstat 131 Homework 5

Yu Tian

Spring 2022-05-15

Elastic Net Tuning

```r
# Read the Pokemon data set into R using read_csv()
Pokemon <- read_csv(file = "Pokemon.csv")
Pokemon %>% head()
```

View Pokemon Date

```
## # A tibble: 6 x 13
##     `#` 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     3 Venusa~ Grass    Poison     625    80    100     123       122       120
## 5     4 Charma~ Fire     <NA>       309    39     52      43        60        50
## 6     5 Charme~ Fire     <NA>       405    58     64      58        80        65
## # ... with 3 more variables: Speed <dbl>, Generation <dbl>, Legendary <lgl>
```

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 %>%
  clean_names()

head(pokemon)
```

Answer

```
## # A tibble: 6 x 13
##   number name       type_1 type_2 total    hp attack defense sp_atk sp_def speed
##    <dbl> <chr>      <chr>  <chr>  <dbl> <dbl>  <dbl>   <dbl>  <dbl>  <dbl> <dbl>
## 1      1 Bulbasaur  Grass  Poison   318    45     49      49     65     65    45
## 2      2 Ivysaur    Grass  Poison   405    60     62      63     80     80    60
## 3      3 Venusaur   Grass  Poison   525    80     82      83    100    100    80
## 4      3 VenusaurM~ Grass  Poison   625    80    100     123    122    120    80
## 5      4 Charmander Fire   <NA>     309    39     52      43     60     50    65
```

```
## 6      5 Charmeleon Fire   <NA>     405    58     64      58      80      65    80
## # ... with 2 more variables: generation <dbl>, legendary <lgl>
```

Compared with the two tables above, we can find the cleam_names() function make the variables name changed with lower-case letter. Besides, the space and dots which separate the words becomes the underscore "_". Since this function standardizes all the variable names into the same clear format, it will be more convenient and easier for the latter understanding and data processing.

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.

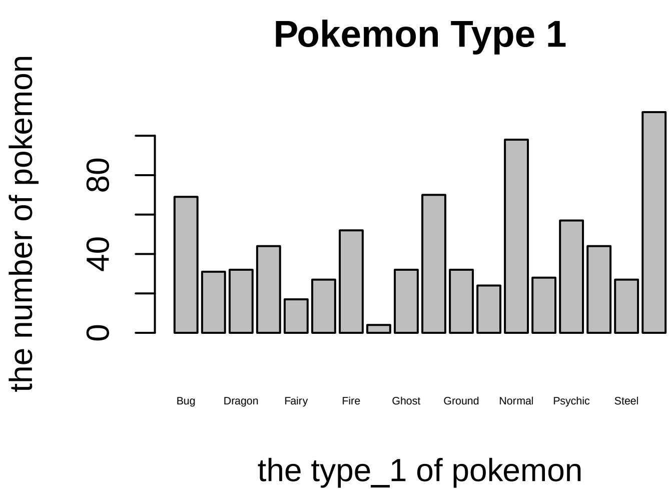
After filtering, convert type_1 and legendary to factors.

```
# Using the entire data set, create a bar chart of the outcome variable, type_1.
count = table(pokemon$type_1)
count
```

Answer

```
##
##      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
```

```
barplot(count, main="Pokemon Type 1",
        xlab = "the type_1 of pokemon", ylab="the number of pokemon",
        width = 1, cex.names = 0.3)
```




```
# How many classes of the outcome are there? Are there any Pokémon types with very few Pokémon? If so, 
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 table above, we can find that there are 18 classes of the outcome. Flying is the Pokémon types with very few Pokémon whose count is only 4.

```
# Filter the entire data set to contain only Pokémon whose type_1 is Bug, Fire, Grass, Normal, Water, o
pokemon_filter <- pokemon %>% filter(type_1 == "Bug" |
                                       type_1 == "Fire" |
                                       type_1 == "Grass" |
                                       type_1 == "Normal" |
                                       type_1 == "Water" |
                                       type_1 == "Psychic")
pokemon_filter %>%
  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_filter_factor <- pokemon_filter %>%
  mutate(type_1 = factor(type_1)) %>%
  mutate(legendary = factor(legendary)) %>%
  mutate(generation = factor(generation))
```

```
head(pokemon_filter_factor)
```

```
## # A tibble: 6 x 13
##   number name       type_1 type_2 total    hp attack defense sp_atk sp_def speed
##    <dbl> <chr>      <fct>  <chr>  <dbl> <dbl>  <dbl>   <dbl>  <dbl>  <dbl> <dbl>
## 1      1 Bulbasaur  Grass  Poison   318    45     49      49     65     65    45
## 2      2 Ivysaur    Grass  Poison   405    60     62      63     80     80    60
## 3      3 Venusaur   Grass  Poison   525    80     82      83    100    100    80
## 4      3 VenusaurM~ Grass  Poison   625    80    100     123    122    120    80
## 5      4 Charmander Fire   <NA>     309    39     52      43     60     50    65
## 6      5 Charmeleon Fire   <NA>     405    58     64      58     80     65    80
## # ... with 2 more variables: generation <fct>, legendary <fct>
```

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(0623)
pokemon_split <- initial_split(pokemon_filter_factor, prop = 0.7, strata = type_1)
pokemon_train <- training(pokemon_split)
pokemon_test <- testing(pokemon_split)
dim(pokemon_filter_factor)
```

Answer

```
## [1] 458  13
```

```
dim(pokemon_train)
```

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

```
dim(pokemon_test)
```

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

```
# Verify the training and testing data sets have the appropriate number of observations
# the number of observations for all data
a <- nrow(pokemon_filter_factor)
a
```

```
## [1] 458
```

```
# the number of observations for training data
b <- nrow(pokemon_train)
b
```

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

```
# the number of observations for test data
c <- nrow(pokemon_test)
c
```

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

```r
# the percentage of observations for training data
b/a
```

```
## [1] 0.6943231
```

```r
# the percentage of observations for test data
c/a
```

```
## [1] 0.3056769
```

The probability of training data observations is 0.6943231, which is almost equal to prob=0.70, so the training and testing data sets have the desired number of observations.

```r
# use v-fold cross-validation on the training set. Use 5 folds. Stratify the folds by type_1 as well.
pokemon_folds <- vfold_cv(pokemon_train, v = 5, strata = type_1)
pokemon_folds
```

```
## #  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
```

We are trying to use v-fold cross-validation and divide the testing data into 5 groups of roughly equal size to prepare for the later fitting and prediction process.

v-fold cross-validation is one kind of resampling method. For each model, this method will randomly divide the observation data into v groups of roughly equal sizes, which are folds. This method will hold out the 1st fold as the validation set to be evaluated. Then the remaining v-1 folds will be analyzed to fit the model. The final estimate of model will get by the average of v results.

Thus, it is useful to make sure the distribution of types in each fold is balanced with the entire data set for the later better prediction.

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

pokemon_recipe
```

Answer

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          8
##
## Operations:
##
## Dummy variables from legendary
## Dummy variables from 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?

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

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

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

Answer

```
## # 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 models in total will be fitting to the data. There are 5 folds and 100 models I will fit to each fold, so total number is 5*100=500.

Exercise 6

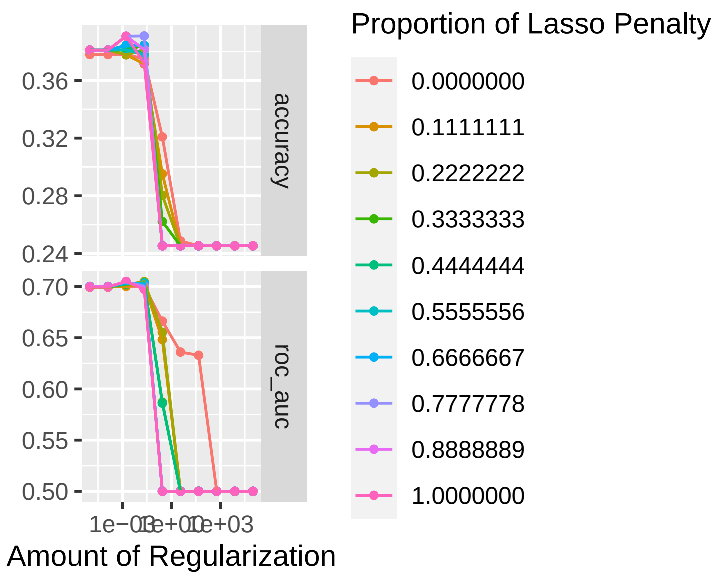
Fit the models to your folded data using tune_grid().

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

```
# Fit the models to your folded data using tune_grid().
pokemon_tune_res <- tune_grid(pokemon_workflow,
                              resamples = pokemon_folds,
                              grid = pokemon_grid)

# Use autoplot() on the results.
autoplot(pokemon_tune_res)
```

Answer



What do you notice? Do larger or smaller values of penalty and mixture produce better accuracy and ROC AUC?

From the graph above, we can find that with the value of penalty and mixture get larger, the value of accuracy and roc_auc become smaller. Thus, The 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.

```
# Use select_best() to choose the model that has the optimal roc_auc.
optimal_auc <- select_best(pokemon_tune_res, metric = "roc_auc")
optimal_auc
```

Answer

```
## # A tibble: 1 x 3
##   penalty mixture .config
##     <dbl>   <dbl> <chr>
## 1 0.00167       1 Preprocessor1_Model093
```

```
# use finalize_workflow(), fit(), and augment() to fit the model to the training set and evaluate its p
pokemon_final <- finalize_workflow(pokemon_workflow, optimal_auc)

pokemon_final_fit <- fit(pokemon_final, data = pokemon_train)

augment(pokemon_final_fit, new_data = pokemon_test) %>%
  accuracy(truth = type_1, estimate = .pred_class)
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.329
```

The accuracy of testing set is 0.3142857, which performs not well.

Exercise 8

Calculate the overall ROC AUC on the testing set.

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

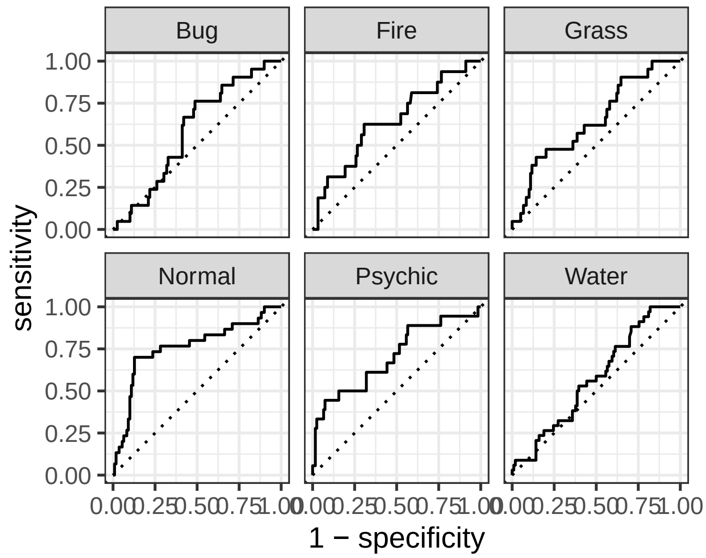
What do you notice? How did your model do? Which Pokemon types is the model best at predicting, and which is it worst at? Do you have any ideas why this might be?

```
# Calculate the overall ROC AUC on the testing set.
roc_auc(augment(pokemon_final_fit, new_data = pokemon_test), type_1, .pred_Bug, .pred_Fire,
                .pred_Grass, .pred_Normal, .pred_Psychic, .pred_Water)
```

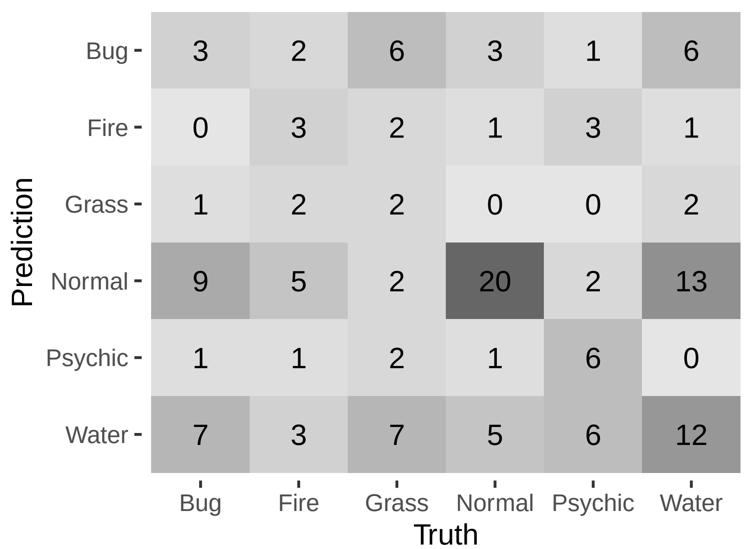
Answer

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.651
```

```
#  create plots of the different ROC curves, one per level of the outcome.
augment(pokemon_final_fit, new_data = pokemon_test) %>%
  roc_curve(type_1, .pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal,
            .pred_Psychic,.pred_Water) %>%
  autoplot()
```




```r
# make a heat map of the confusion matrix.
augment(pokemon_final_fit, new_data = pokemon_test) %>%
  conf_mat(truth = type_1, estimate = .pred_class)  %>%
  autoplot(type = "heatmap")
```



From the graphs and calculation results above, we can find that the value of accuracy and roc_auc are not very high. Thus, the model performs not very well. Also, since the prediction accuracy of the six types is different, we can find that Normal Pokemon type is the model best at predicting, the Water Pokemon type is the second model best at predicting, the Grass Pokemon type is the model worst at predicting, the Fire Pokemon type is the second model worst at predicting. I think we get this result because the Normal and Water Pokemon Type has more observations in the data set, and the Grass and Fire Pokemon Type has less observations to predict in the data set.