PSTAT 131 : HW 5

Ephets Head

5/13/2022

We will be working with the file "pokemon.cvs" from Kaggle. 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.

```
pokemon <- read.csv("data_pokemon/Pokemon.csv")
head(pokemon,6)
```

```
##   X.                  Name Type.1 Type.2 Total HP Attack Defense Sp..Atk
## 1  1             Bulbasaur  Grass Poison   318 45     49      49      65
## 2  2               Ivysaur  Grass Poison   405 60     62      63      80
## 3  3              Venusaur  Grass Poison   525 80     82      83     100
## 4  3 VenusaurMega Venusaur  Grass Poison   625 80    100     123     122
## 5  4            Charmander   Fire          309 39     52      43      60
## 6  5            Charmeleon   Fire          405 58     64      58      80
##   Sp..Def Speed Generation Legendary
## 1      65    45          1     False
## 2      80    60          1     False
## 3     100    80          1     False
## 4     120    80          1     False
## 5      50    65          1     False
## 6      65    80          1     False
```

Exercise 1: Install and load the janitor package. Use its clean_names() function on the Pokémon data and save the results. What happened to the data? Why is this function useful?

```
library("janitor")
```

```
##
## Attaching package: 'janitor'
```

```
## The following objects are masked from 'package:stats':
##
##     chisq.test, fisher.test
```

```
Pokemon <- clean_names(pokemon)
names(Pokemon)
```

```
##  [1] "x"          "name"       "type_1"     "type_2"     "total"
##  [6] "hp"         "attack"     "defense"    "sp_atk"     "sp_def"
## [11] "speed"      "generation" "legendary"
```

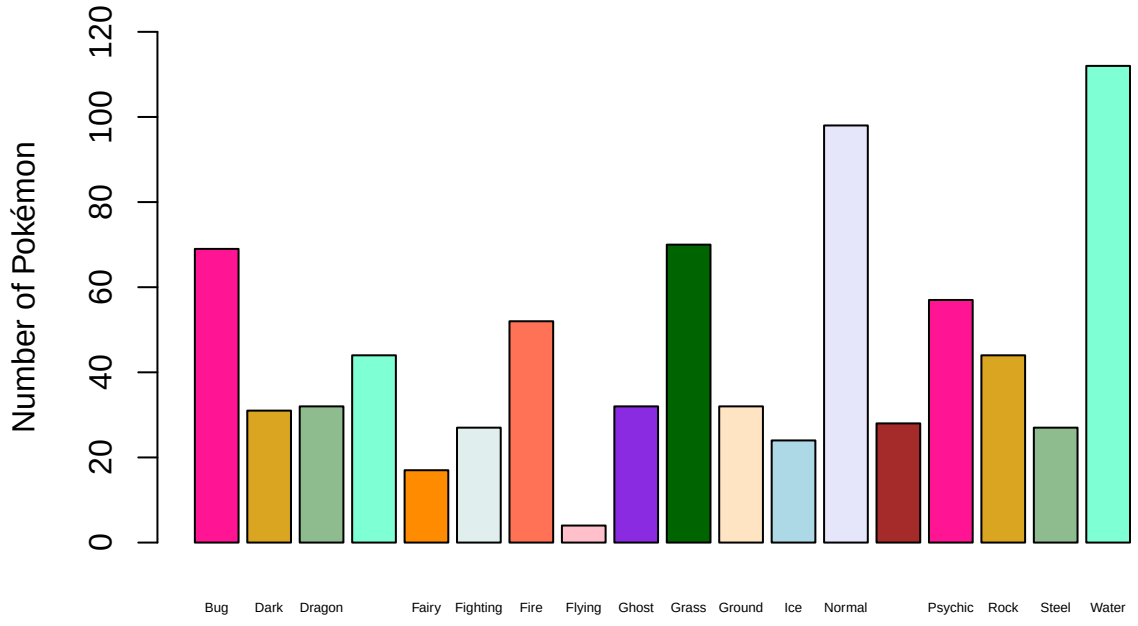
Comparing the column names outputted after using the clean_names() function with the ones we first downloaded from kaggle, we can see that the function replaced the placeholder character "." with "_" and made all column titles lowercase. This function is useful because it standardizes variable names into one consistent format. Some of the names before were irregular, with ".." in between words instead of a single ".". Now, the names are all readable, unique, and standardized.

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.

```
#first we will create a df of each type_1 class and its accompanying count
Pokemon_sorted <- group_by(Pokemon, type_1)
type_counted <- count(Pokemon_sorted)
type_counted <- data.frame(type_counted)

color_list <- c("deeppink1","goldenrod","darkseagreen","aquamarine", "darkorange","azure2","coral1","pi

#now we will plot the number of Pokemon of each type in a bar chart
barplot(height=type_counted$n,xlab= "Type of Pokémon", ylab= "Number of Pokémon",names.arg=type_counted
```



The graph shows that there are 18 different classes of the outcome `type_1`. Some of these classes, particularly the type "Flying", contain almost no Pokémon. According to the graph, the other smallest classes (all containing less than 30 Pokémon) are "Fairy", "Ice", "Fighting", "Steel", and "Poison".

We will now filter the entire set to only include the top six most common types.

```
Pokemon <- Pokemon %>%
  filter(
    type_1 %in% c("Bug","Fire","Grass","Normal","Water","Psychic")
  )
```

Next, we convert variables `type_1` and `legendary` to factors.

```
Pokemon <- Pokemon %>%
  mutate(
```

```
    type_1 = factor(type_1),
    legendary = factor(legendary),
    generation = factor(generation)
  )
```

Exercise 3: Perform an initial split of the data. Stratify by the outcome variable. You can choose the proportion to use. Verify that both 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 (look for a `strata` argument.)

```
#first we will set a seed and split the data
set.seed(9876)
poke_split <- initial_split(Pokemon, prop=0.75, strata=type_1)
pokemon_train <- training (poke_split)
pokemon_test <- testing(poke_split)

#now we check that the dimensions of the training and testing sets seem correct
dim(pokemon_train)
```

```
## [1] 341  13
```

```
dim(pokemon_test)
```

```
## [1] 117  13
```

```
#now we use v-fold cross validation on the training set with 5 folds, stratified by type_1
pokemon_fold <- vfold_cv(pokemon_train, v=5, strata=type_1)
```

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`, and center and scale all predictors.

```
pokemon_recipe <- recipe(type_1 ~ legendary + generation + sp_atk + attack + speed + defense + hp + sp_
  step_dummy(legendary) %>%
  step_dummy(generation) %>%
  step_center(all_predictors()) %>%
  step_scale(all_predictors())
```

Exercise 5: We'll be fitting and tuning an elastic net, tuning `penalty` and `mixture` (use `multinom_reg` with the `glm` 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, and `penalty` should range from -5 to 5 (it's log-scaled.) How many total models will we be fitting across the folded data?

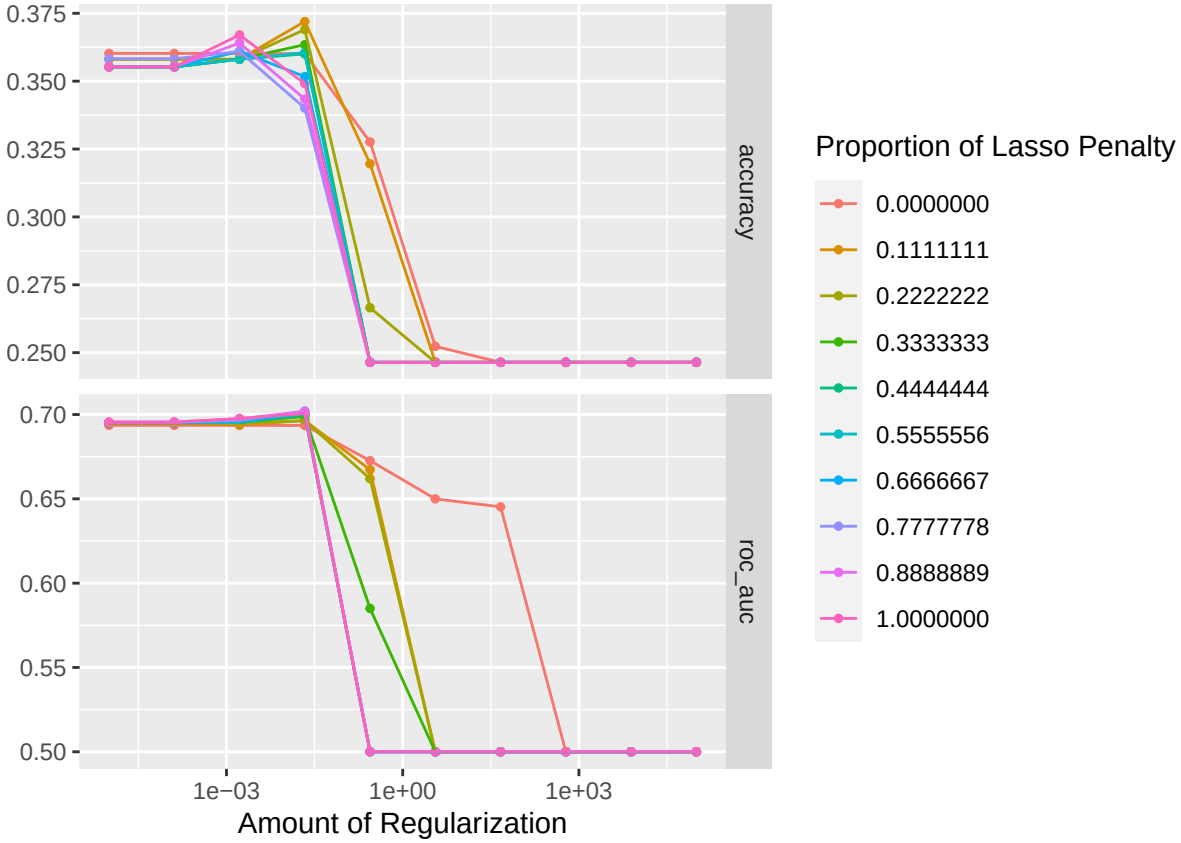
```
#first we specify a multinom_reg() model with glmnet engine
mr_spec <- multinom_reg(penalty=tune(),mixture=tune()) %>%
  set_mode("classification") %>%
  set_engine("glmnet")

#using our model engine and our recipe, we set up a workflow (with pen. and mix. tuned)
mr_workflow <- workflow() %>%
  add_model(mr_spec) %>%
  add_recipe(pokemon_recipe)

#now we create a regular grid for penalty and mixture with 10 levels each

pen_grid <- grid_regular(penalty(range=c(-5,5)),
```

```
                        mixture(range=c(0,1)),
                        levels=10)
```

Since we have 5 folds and 10 levels to our elastic net grid, we will be fitting a total of 50 models.

Exercise 6: Fit the models to your 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_pokemon <- tune_grid(mr_workflow, grid=pen_grid, resamples=pokemon_fold)
autoplot(fit_pokemon)
```



From the graph above, we can infer that larger values of penalty produce smaller values of ROC AUC and accuracy.

Exercise 7: Use `select_best()` to choose the model that has an optimal roc_auc. Then, use `finalize_workflow()`, `fit()`, and `augment()` to fit the data to the training set and evaluate its performance on the testing set.

```
#first we will select the model with values of penalty and mixture that optimize ROC AUC
best_model <- select_best(fit_pokemon, metric="roc_auc")
best_model
```

```
## # A tibble: 1 x 3
##   penalty mixture .config
##     <dbl>   <dbl> <chr>
## 1  0.0215   0.778 Preprocessor1_Model074
```

```
#next we will update/finalize the workflow using the best model
final_workflow <- finalize_workflow(mr_workflow, best_model)
```

```
#now we fit the final optimized workflow to the training data set
fitted_final_wf <- fit(final_workflow, data=pokemon_train)

#finally, we use augment() to try our model on our testing set and validate its performance
fit_on_test <- augment(fitted_final_wf, new_data=pokemon_test)

fit_on_test %>%
  accuracy(truth=type_1, estimate=.pred_class)
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.359
```

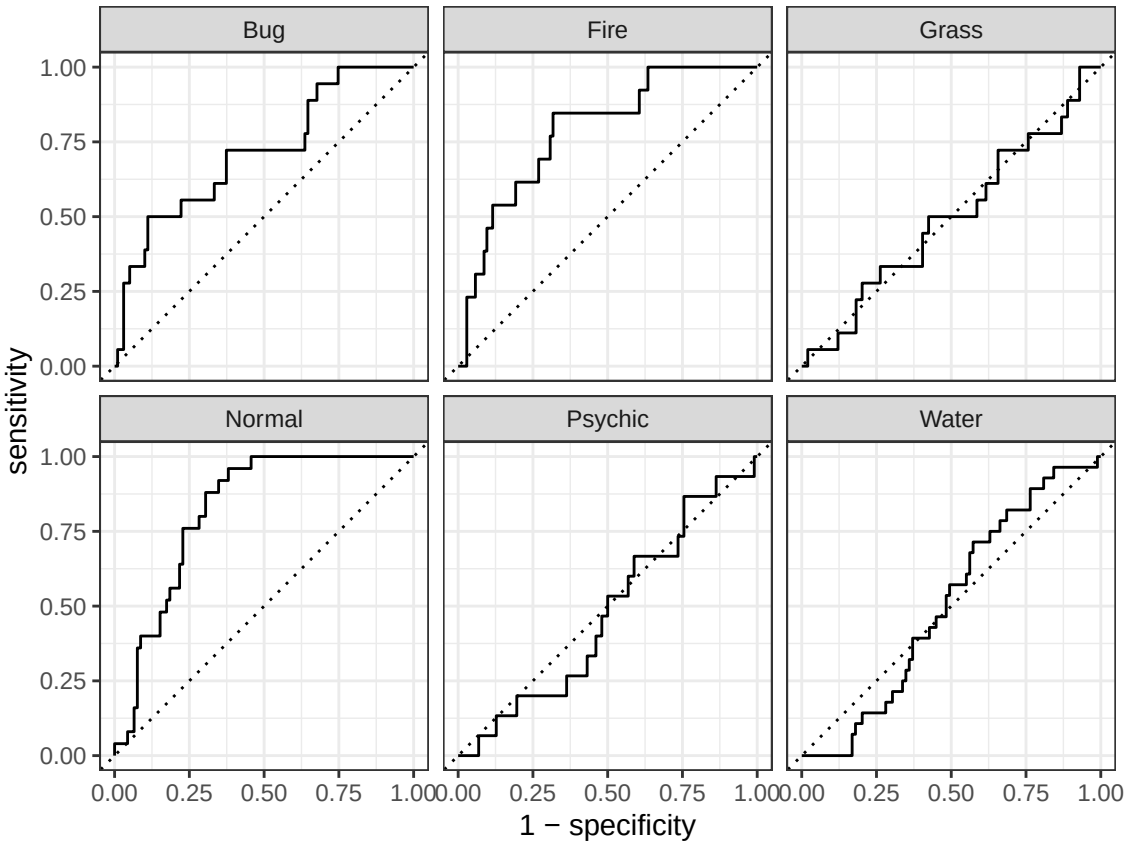
Exercise 8: Calculate the overall ROC AUC on the testing set. Then create plots of the different ROC curves, one per level of the outcome. Also make a heat map of the confusion matrix.

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

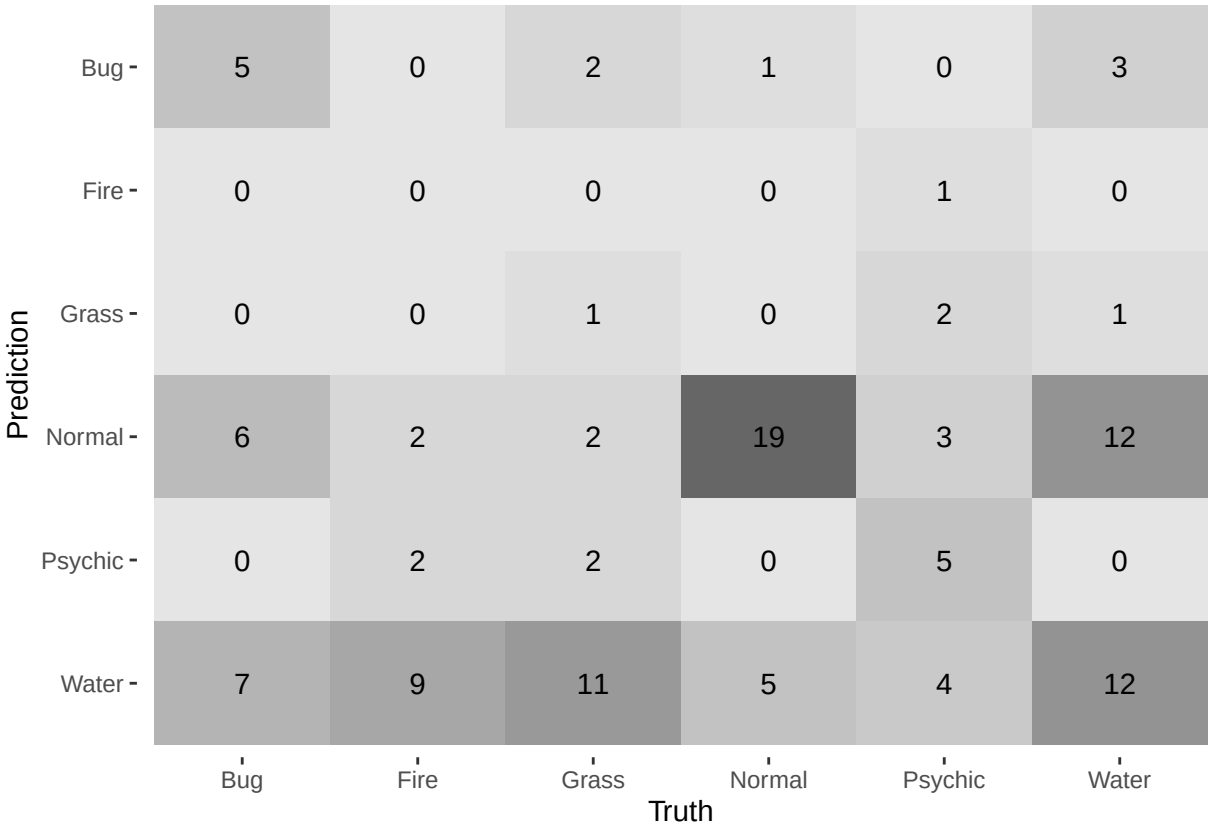
```
#first we calculate the ROC AUC of our final fitted workflow on the testing data
fit_on_test %>%
  roc_auc(truth=type_1,
          estimate=c(.pred_Bug,.pred_Fire,.pred_Grass,.pred_Normal,.pred_Water,.pred_Psychic))
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.625
```

```
#next we plot the ROC curve of each level of the outcome
fit_on_test %>%
  roc_curve(type_1,
            c(.pred_Bug,.pred_Fire,.pred_Grass,.pred_Normal,.pred_Water,.pred_Psychic))%>%
  autoplot()
```




```
#to create a heatmap of the confusion matrix, we use conf_mat
fit_on_test %>%
  conf_mat(truth=type_1, estimate= .pred_class) %>%
  autoplot(type="heatmap")
```



As shown in the heatmap, the model did a very good job in correctly predicting the `type_1` of Normal Pokémon, but very badly in predicting Fire and Grass Pokémon. Water types were successfully predicted about $\frac{1}{2}$ of the time, and Bug and Psychic types were each predicted correctly about $\frac{1}{3}$ of the time (pretty low accuracy).

Since data such as `speed`, `defense`, and `generation` all have a similar range of possible values regardless of the `type_1` classification, it seems likely that the model has identified certain values as characteristic of one type that, in reality, could be exhibited by any of the six types. This might explain why a substantial amount of every type of Pokémon was classified incorrectly as a Water or Normal types.The types that performed the best were also the ones with the largest numbers of observations.