Homework 5

PSTAT 131/231

Contents

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 <- read.csv("C:/Users/rocke/Downloads/homework-5/homework-5/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?

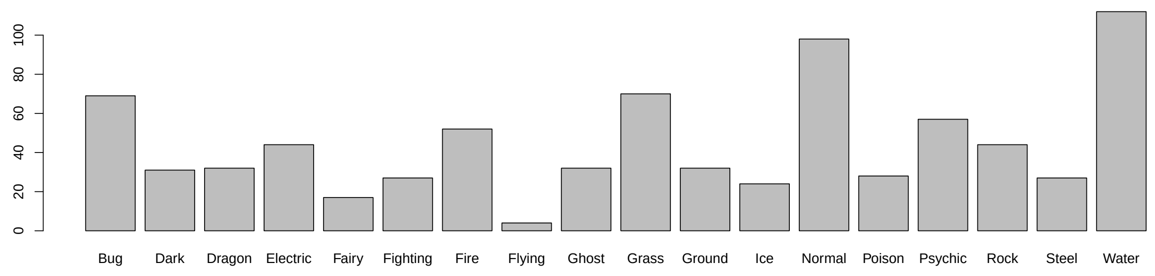
```r
library(janitor)
pokemon <- clean_names(pokemon)
```

Using clean_names(), the column names change so that only integers, characters, and "_" are used. I think clean_names is useful because it allows for a systematic naming convention, makes the calling columns easier if there are weird symbols, and allows for columns to be called if there is a space in the column name.

Exercise 2

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

```r
types <- table(pokemon$type_1)  # need to create a table for the barplot function
barplot(types)
```



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

There are 18 classes of the outcome. Yes, there are a few pokemon types with very few Pokemon, such as: Dark, Dragon, Fairy, Fighting, Flying, Ghost, Ground, Ice, Poison, and Steel.

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.

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

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

```r
pokemon$type_1 <- factor(pokemon$type_1)
pokemon$legendary <- factor(pokemon$legendary)
pokemon$generation <- factor(pokemon$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.

```r
set.seed(1234)
pokemon_split <- initial_split(pokemon, prop = 0.7, strata = type_1)
pokemon_train <- training(pokemon_split)
pokemon_test <- testing(pokemon_split)
dim(pokemon_split)  # verifying training and testing sets have the desired number of observations
```

```
##   analysis assessment          n          p
##        318        140        458         13
```

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?

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

Stratifying the folds may be useful because if we do not, there can be an imbalanced number of types in each fold, leading to poor performance. With an equal distribution of the types in each fold, the metrics will be more accurate.

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.

```
pokemon_recipe <- recipe(type_1 ~ legendary + generation + sp_atk + attack + speed + defense + hp + sp_
                         data = pokemon_train) %>%
  step_dummy(legendary) %>%
  step_dummy(generation) %>%
  step_normalize(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).

```
multi_reg <- multinom_reg(penalty = tune(), mixture = tune()) %>%
  set_mode("classification") %>%
  set_engine("glmnet")

pokemon_workflow <- workflow() %>%
  add_model(multi_reg) %>%
  add_recipe(pokemon_recipe)

params <- parameters(penalty(range = c(-5, 5)), mixture(range = c(0, 1)))

penalty_mixture <- grid_regular(params, levels = c(10, 10))

penalty_mixture
```

```
## # 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
```

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

I will be fitting 10 * 10 * 5 = 500 models.

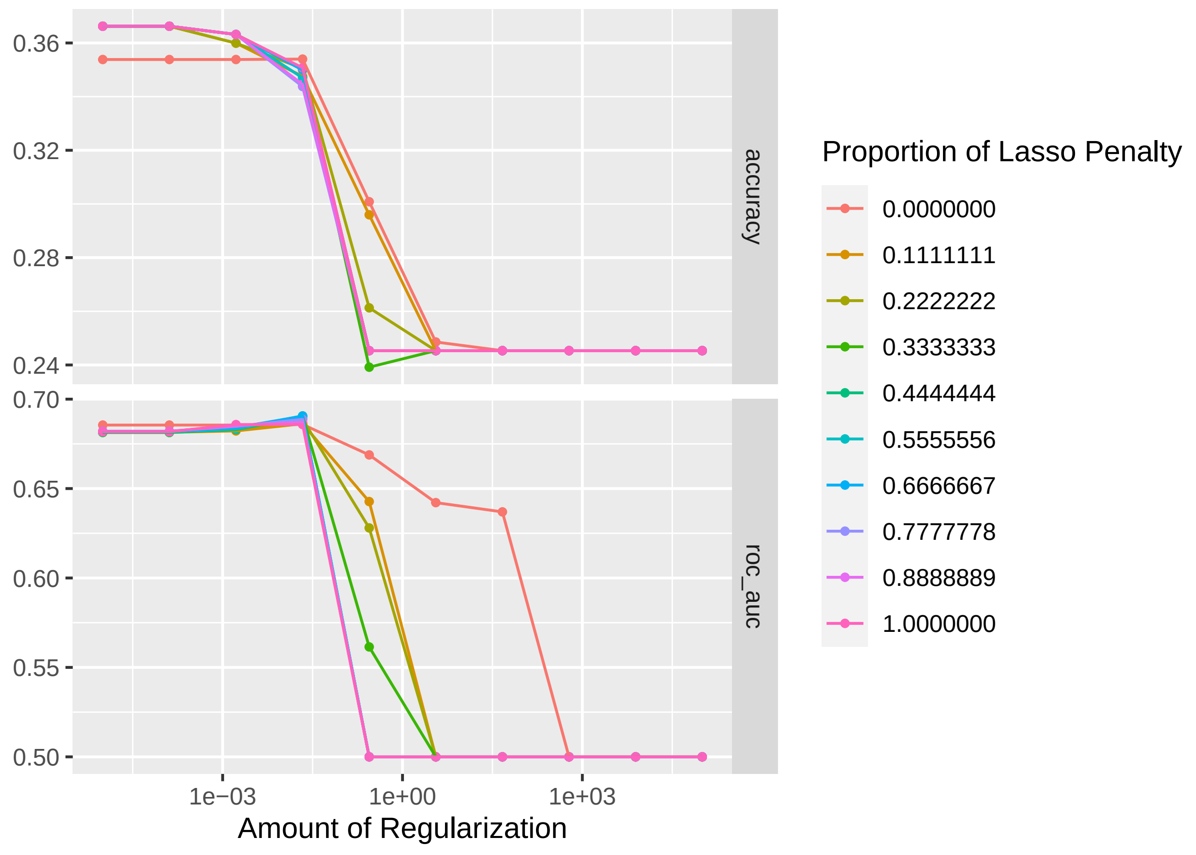
Exercise 6

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

```r
tune_res <- tune_grid(
  object = pokemon_workflow,
  resamples = pokemon_folds,
  grid = penalty_mixture
)
```

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

```r
autoplot(tune_res)
```



I notice that each of the lines tends to get better at a certain amount of regularization, and then drop in performance(accuracy & ROC AUC) drastically. Smaller values of penalty and larger values of mixture tend to 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
best_penalty_mixture <- select_best(tune_res, metric = "roc_auc")

pokemon_final <- finalize_workflow(pokemon_workflow, best_penalty_mixture)
```

```r
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     0.364
```

Exercise 8

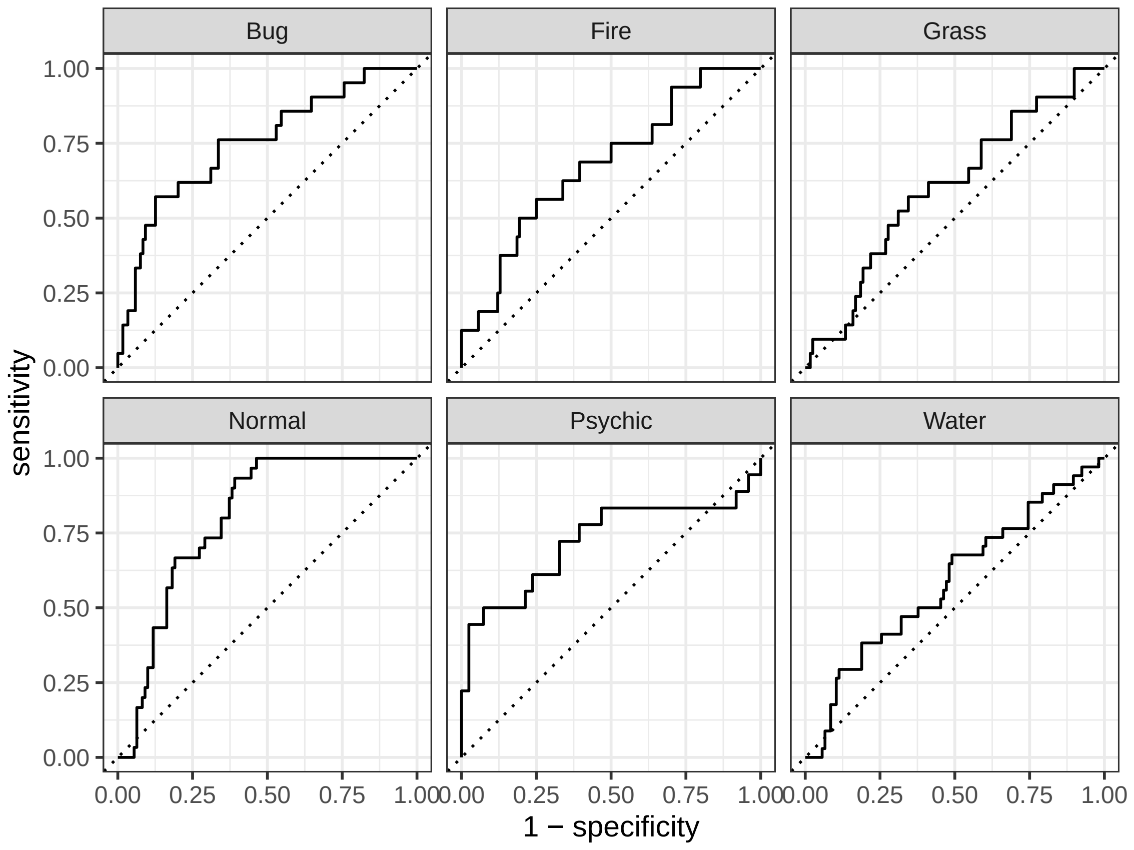
Calculate the overall ROC AUC on the testing set.

```r
augment(pokemon_final_fit, new_data = pokemon_test) %>%
  roc_auc(truth = type_1, .pred_Bug:.pred_Water)
```

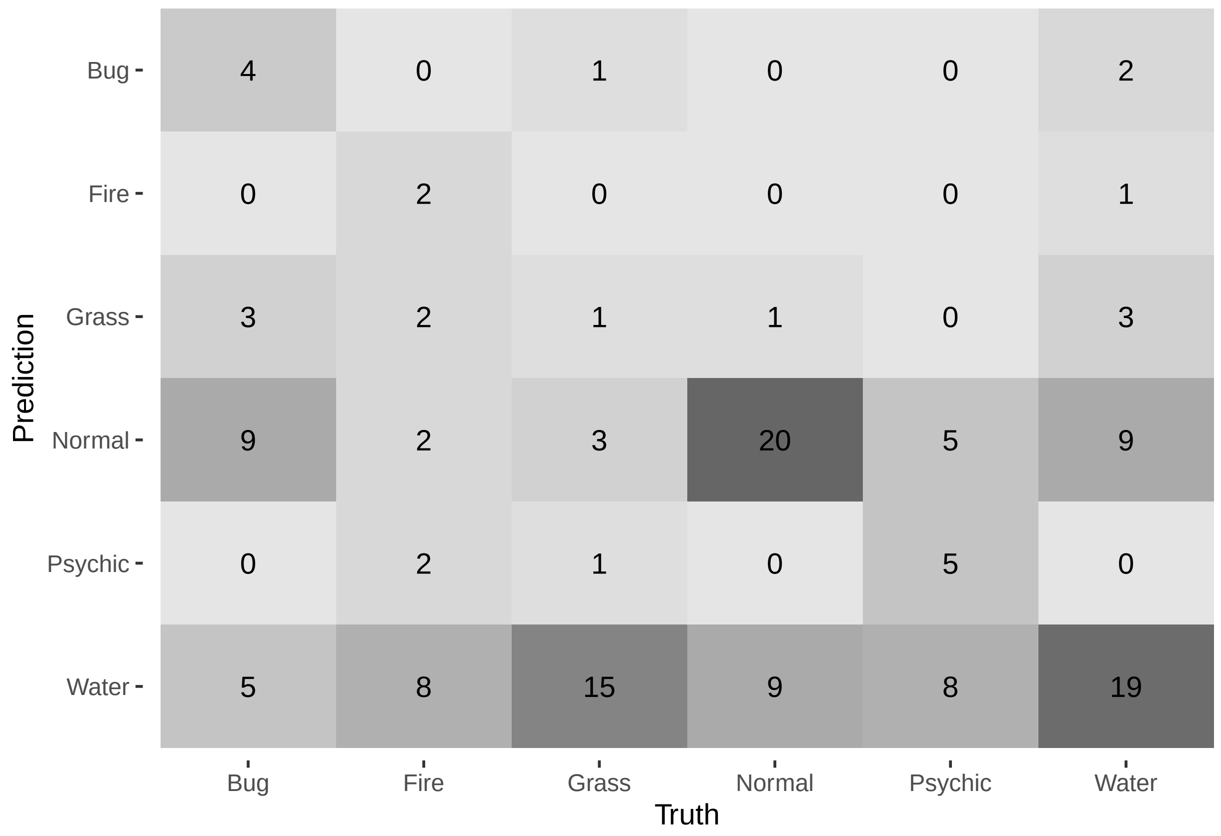
```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc hand_till      0.688
```

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

```r
augment(pokemon_final_fit, new_data = pokemon_test) %>%
  roc_curve(truth = type_1, .pred_Bug:.pred_Water) %>%
  autoplot()
```



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



Prediction \ Truth	Bug	Fire	Grass	Normal	Psychic	Water
Bug	4	0	1	0	0	2
Fire	0	2	0	0	0	1
Grass	3	2	1	1	0	3
Normal	9	2	3	20	5	9
Psychic	0	2	1	0	5	0
Water	5	8	15	9	8	19

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?

I notice that my model has varying results for each pokemon type. My model had an overall accuracy of 36.43% and an overall ROC AUC of .6875. It does the best at predicting Normal types, while it does the worst at predicting Water Types. I think it does well at predicting Normal types because it has some characteristics that distinguish it from other types (and because it has the 2nd most amount of data). I think it does the worst at predicting Water types because the model cannot pick up on characteristics that distinguish it from others; thus, it predicts Water type for many of the observations (note that Water has the most amount of data, may increase the probability that the model classifies a type as Water).