Homework 5

Jules Merigot (8488256)

November 15, 2022

PSTAT 131/231 Statistical Machine Learning - Fall 2022

Elastic Net Tuning

Before we get started, let's load the Pokemon data in into our workspace.

```
pokemon_data <- read.csv(file = "C:/Users/jules/OneDrive/Desktop/homework-5/data/Pokemon.csv")
head(pokemon_data)
```

```
##   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

Let's load the janitor package, and use its `clean_names()` function on the Pokémon data. We'll save the results to work with for the rest of the assignment.

```
library(janitor)

Pokemon_data <- pokemon_data %>%
  clean_names()
head(Pokemon_data)
```

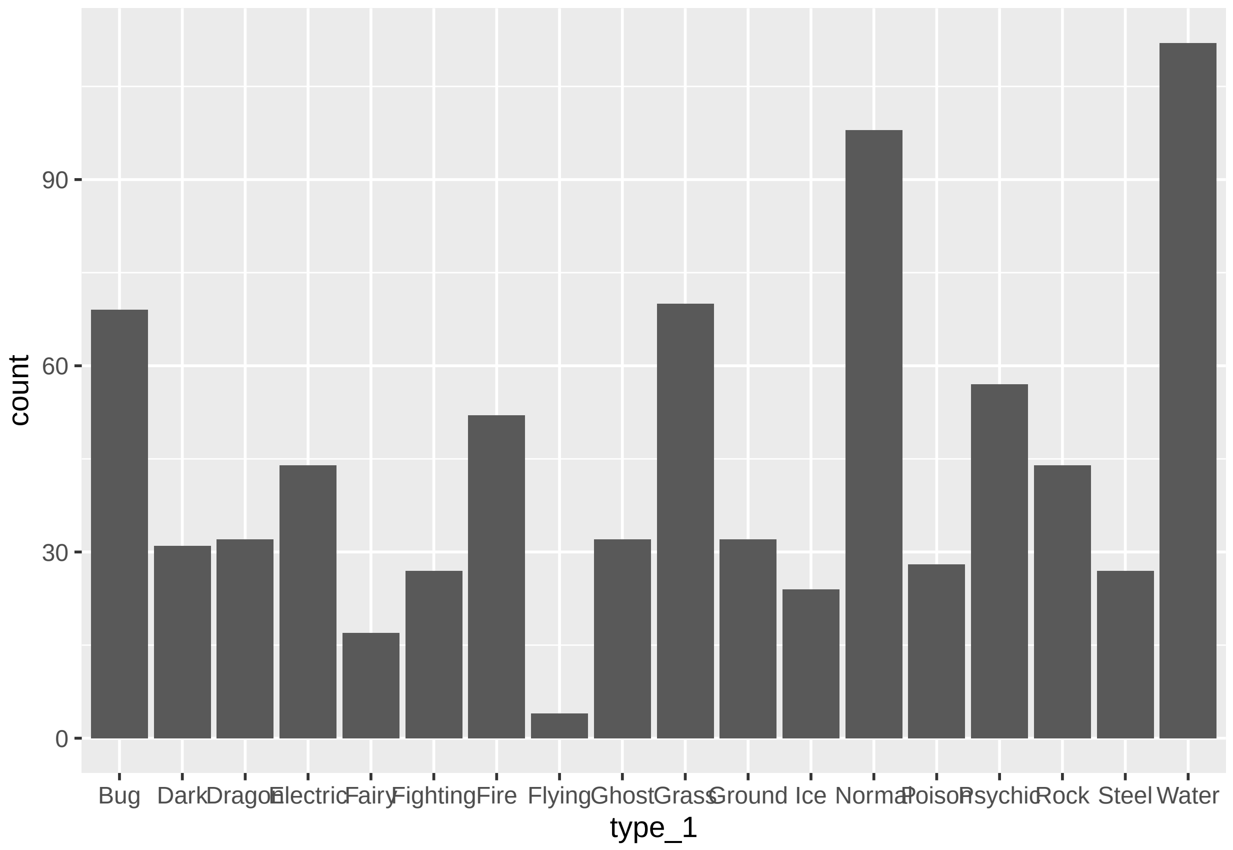
```
##   x                  name type_1 type_2 total hp attack defense sp_atk sp_def
## 1 1             Bulbasaur  Grass Poison   318 45     49      49     65     65
## 2 2               Ivysaur  Grass Poison   405 60     62      63     80     80
## 3 3              Venusaur  Grass Poison   525 80     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
```

As we can see in the data above, the names of each column have been changed to simpler, more efficient, and unique names using strictly the "_" character, numbers, and letters. This shows how useful `clean_names()` is, because it allows for a rapid change in the varaible and predictor names, thus allowing them to be referenced and used more efficiently in the rest of project or assignment being completed.

Exercise 2

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

```
Pokemon_data %>%
  ggplot(aes(x=type_1)) +
  geom_bar()
```



There are 18 classes of the outcome `type_1`, which means there are 18 different types of Pokemon. While

there are many Pokemon of the “Water” type, there are very few Pokemon of the “Flying” type.
For this assignment, we’ll handle the rarer classes by simply filtering them out. Let’s filter the entire data set to contain only Pokemon whose `type_1` is Bug, Fire, Grass, Normal, Water, or Psychic.

```
Pokemon_data <- subset(Pokemon_data, type_1 == c("Bug", "Fire", "Grass",
                                                 "Normal", "Water", "Psychic"))
```

Now that we’re done filtering, let’s convert `type_1`, `legendary`, and `generation` to factors.

```
Pokemon_data$type_1 <- factor(Pokemon_data$type_1)
Pokemon_data$legendary <- factor(Pokemon_data$legendary)
Pokemon_data$generation <- factor(Pokemon_data$generation)
```

Exercise 3

Let’s perform an initial split of the data, and stratify by the outcome variable.

```
set.seed(8488)

Pokemon_split <- initial_split(Pokemon_data, prop=0.70, strata=type_1)

Pokemon_train <- training(Pokemon_split)
Pokemon_test <- testing(Pokemon_split)
```

For splitting the data, I chose a proportion of 0.70 because it allows for more training data, while retaining enough data to be tested since there is a limited amount of observations. The training data has 559 observations while the testing data has 241 observations.

Next, let’s use v-fold cross-validation on the training set, using 5 folds. We’ll stratify the folds by `type_1` as well.

```
Pokemon_folds <- vfold_cv(Pokemon_train, v = 5, strata=type_1)
```

In this case, stratifying the folds is useful to ensure that each fold is representative of all strata of the data.

Exercise 4

Let’s set up a recipe to predict `type_1` with `legendary`, `generation`, `sp_atk`, `attack`, `speed`, `defense`, `hp`, and `sp_def`. We’ll also dummy-code `legendary` and `generation`, as well as 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(c(legendary, generation)) %>%
  step_normalize(all_predictors())

Pokemon_recipe %>% prep() %>% juice()
```

```
## # A tibble: 54 x 13
##    sp_atk  attack  speed defense      hp  sp_def type_1 legenda~1 gener~2 gener~3
##     <dbl>   <dbl>  <dbl>   <dbl>   <dbl>   <dbl> <fct>      <dbl>   <dbl>   <dbl>
##  1 -0.535  1.48    1.23    0.601  0.0983  0.333 Bug       -0.136  -0.443  -0.350
##  2 -0.535 -1.19    0.548  -0.613 -0.487   1.35  Bug       -0.136   2.22   -0.350
##  3 -0.535  2.20   -0.137   1.41   0.0983  0.333 Bug       -0.136   2.22   -0.350
##  4 -0.782  0.164   0.548  -0.411 -0.0968  0.163 Bug       -0.136  -0.443   2.80
##  5 -0.999 -0.549  -0.925   0.196 -0.877  -0.348 Bug       -0.136  -0.443  -0.350
##  6 -1.15  -0.121  -0.480   0.803 -0.682  -1.20  Bug       -0.136  -0.443  -0.350
##  7  0.857  0.0574  2.60   -1.02   0.488  -0.348 Bug       -0.136  -0.443  -0.350
##  8 -1.40  -1.19   -1.16   -1.02  -1.15   -1.54  Bug       -0.136  -0.443  -0.350
##  9  1.78   2.20    1.06    1.85   0.410   0.503 Fire      -0.136  -0.443  -0.350
## 10 -0.380 -0.584  -0.137  -0.896 -1.11   -0.689 Fire      -0.136   2.22   -0.350
## # ... with 44 more rows, 3 more variables: generation_X4 <dbl>,
## #   generation_X5 <dbl>, generation_X6 <dbl>, and abbreviated variable names
## #   1: legendary_True, 2: generation_X2, 3: generation_X3
```

Exercise 5

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

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

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

Pokemon_workflow <- workflow() %>%
  add_recipe(Pokemon_recipe) %>%
  add_model(Pokemon_spec)

pen_mix_grid <- grid_regular(penalty(range = c(-5, 5)), mixture(range = c(0,1)), levels = 10)
pen_mix_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
```

Since we have 10 levels for penalty and 10 levels mixture as well as 5 folds for the training data, we will be fitting a total of 500 models when fitting these models to our folded data.

Exercise 6

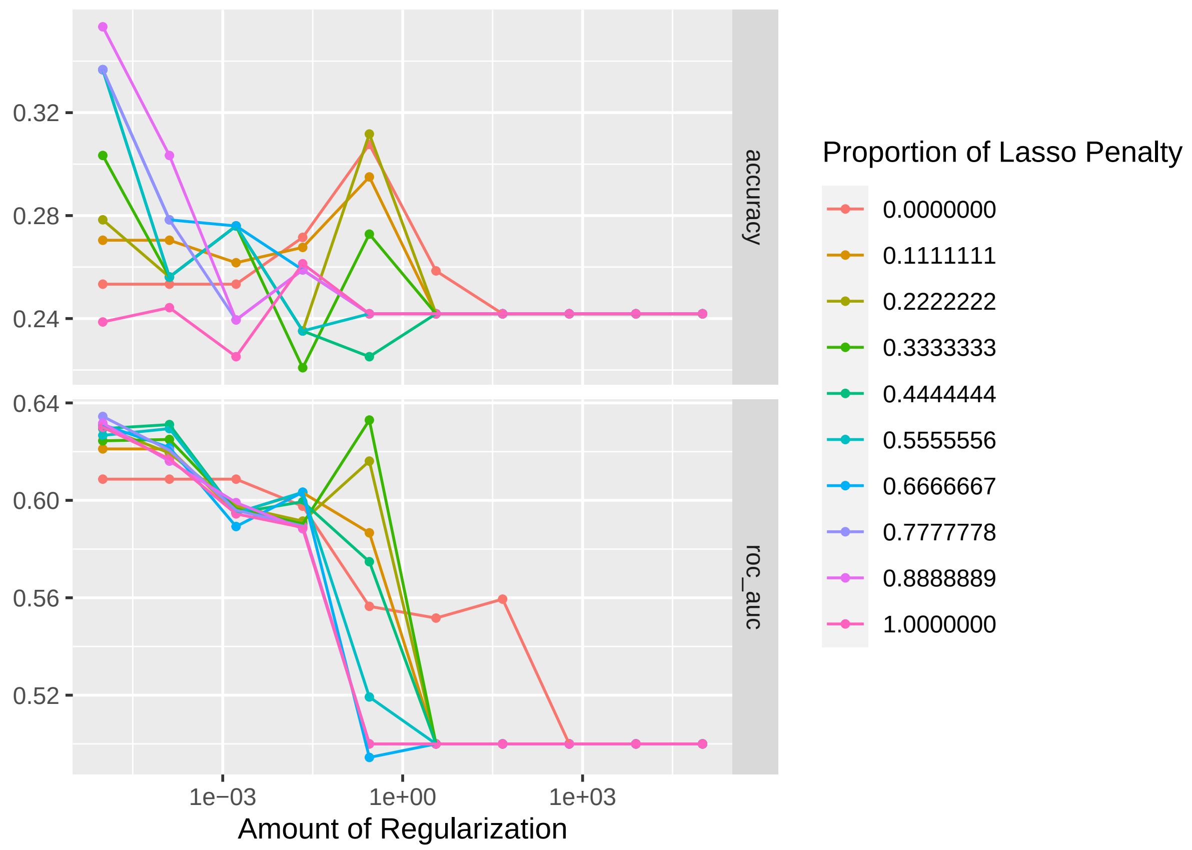
Let's fit the models to our folded data using `tune_grid()`.

```
tune_res <- tune_grid(
  Pokemon_workflow,
  resamples = Pokemon_folds,
  grid = pen_mix_grid
)
```

```
## ! Fold1: preprocessor 1/1, model 1/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold1: preprocessor 1/1, model 2/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold1: preprocessor 1/1, model 3/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold1: preprocessor 1/1, model 4/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold1: preprocessor 1/1, model 5/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold1: preprocessor 1/1, model 6/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold1: preprocessor 1/1, model 7/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold1: preprocessor 1/1, model 8/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold1: preprocessor 1/1, model 9/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold1: preprocessor 1/1, model 10/10: one multinomial or binomial class has fewer than 8  observat

## ! Fold2: preprocessor 1/1, model 1/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold2: preprocessor 1/1, model 2/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold2: preprocessor 1/1, model 3/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold2: preprocessor 1/1, model 4/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold2: preprocessor 1/1, model 5/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold2: preprocessor 1/1, model 6/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold2: preprocessor 1/1, model 7/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold2: preprocessor 1/1, model 8/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold2: preprocessor 1/1, model 9/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold2: preprocessor 1/1, model 10/10: one multinomial or binomial class has fewer than 8  observat
## ! Fold3: preprocessor 1/1, model 1/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold3: preprocessor 1/1, model 2/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold3: preprocessor 1/1, model 3/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold3: preprocessor 1/1, model 4/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold3: preprocessor 1/1, model 5/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold3: preprocessor 1/1, model 6/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold3: preprocessor 1/1, model 7/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold3: preprocessor 1/1, model 8/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold3: preprocessor 1/1, model 9/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold3: preprocessor 1/1, model 10/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold4: preprocessor 1/1, model 1/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold4: preprocessor 1/1, model 2/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold4: preprocessor 1/1, model 3/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold4: preprocessor 1/1, model 4/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold4: preprocessor 1/1, model 5/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold4: preprocessor 1/1, model 6/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold4: preprocessor 1/1, model 7/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold4: preprocessor 1/1, model 8/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold4: preprocessor 1/1, model 9/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold4: preprocessor 1/1, model 10/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold5: preprocessor 1/1: Column(s) have zero variance so scaling cannot be used: ‘legendary_True‘.

## ! Fold5: preprocessor 1/1, model 1/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold5: preprocessor 1/1, model 2/10: one multinomial or binomial class has fewer than 8  observatio

## ! Fold5: preprocessor 1/1, model 3/10: one multinomial or binomial class has fewer than 8  observatio
## ! Fold5: preprocessor 1/1, model 4/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold5: preprocessor 1/1, model 5/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold5: preprocessor 1/1, model 6/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold5: preprocessor 1/1, model 7/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold5: preprocessor 1/1, model 8/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold5: preprocessor 1/1, model 9/10: one multinomial or binomial class has fewer than 8  observati

## ! Fold5: preprocessor 1/1, model 10/10: one multinomial or binomial class has fewer than 8  observat
```

We now use `autoplot()` on the results.

```
autoplot(tune_res)
```



As we can see in the plots above, larger values of penalty tend to produce lower accuracy values and lower ROC AUC values for each mixture level, while smaller values of penalty tend to produce higher accuracy and ROC AUC values. Additionally, larger values of mixture tend to produce more consistent accuracy and ROC AUC across all penalty levels.

Exercise 7

Let's use `select_best()` to choose the model that has the optimal `roc_auc`.

```
collect_metrics(tune_res)
```

```
## # A tibble: 200 x 8
##     penalty mixture .metric  .estimator  mean     n std_err .config
##       <dbl>   <dbl> <chr>    <chr>      <dbl> <int>   <dbl> <chr>
##  1 0.00001        0 accuracy multiclass 0.253     5  0.0349 Preprocessor1_Model~
##  2 0.00001        0 roc_auc  hand_till  0.609     5  0.0138 Preprocessor1_Model~
##  3 0.000129       0 accuracy multiclass 0.253     5  0.0349 Preprocessor1_Model~
##  4 0.000129       0 roc_auc  hand_till  0.609     5  0.0138 Preprocessor1_Model~
##  5 0.00167        0 accuracy multiclass 0.253     5  0.0349 Preprocessor1_Model~
##  6 0.00167        0 roc_auc  hand_till  0.609     5  0.0138 Preprocessor1_Model~
##  7 0.0215         0 accuracy multiclass 0.272     5  0.0415 Preprocessor1_Model~
##  8 0.0215         0 roc_auc  hand_till  0.598     5  0.0192 Preprocessor1_Model~
##  9 0.278          0 accuracy multiclass 0.308     5  0.0361 Preprocessor1_Model~
## 10 0.278          0 roc_auc  hand_till  0.556     5  0.0116 Preprocessor1_Model~
## # ... with 190 more rows
```

```
best_penalty <- select_best(tune_res, metric = "roc_auc")
best_penalty
```

```
## # A tibble: 1 x 3
##   penalty mixture .config
##     <dbl>   <dbl> <chr>
## 1 0.00001   0.778 Preprocessor1_Model071
```

We can see above the model that has the optimal `roc_auc`.

Then we'll use `finalize_workflow()`, `fit()`, and `augment()` to fit the model to the training set and evaluate its performance on the testing set.

```
lasso_final <- finalize_workflow(Pokemon_workflow, best_penalty)

lasso_final_fit <- fit(lasso_final, data = Pokemon_train)

augment(lasso_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     0.214
```

Exercise 8

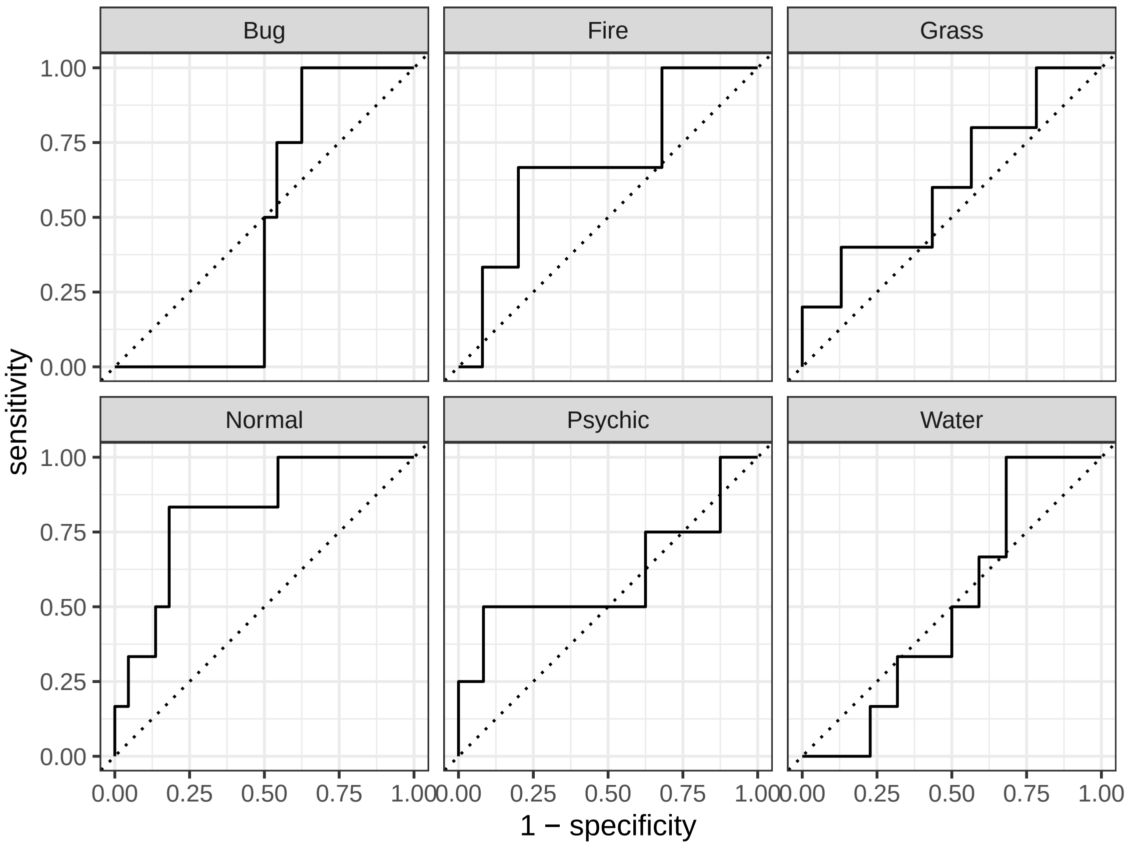
Almost there! Now let's calculate the overall ROC AUC on the testing set.

```
roc <- augment(lasso_final_fit, Pokemon_test)

roc %>%
  roc_auc(type_1, 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      0.588
```

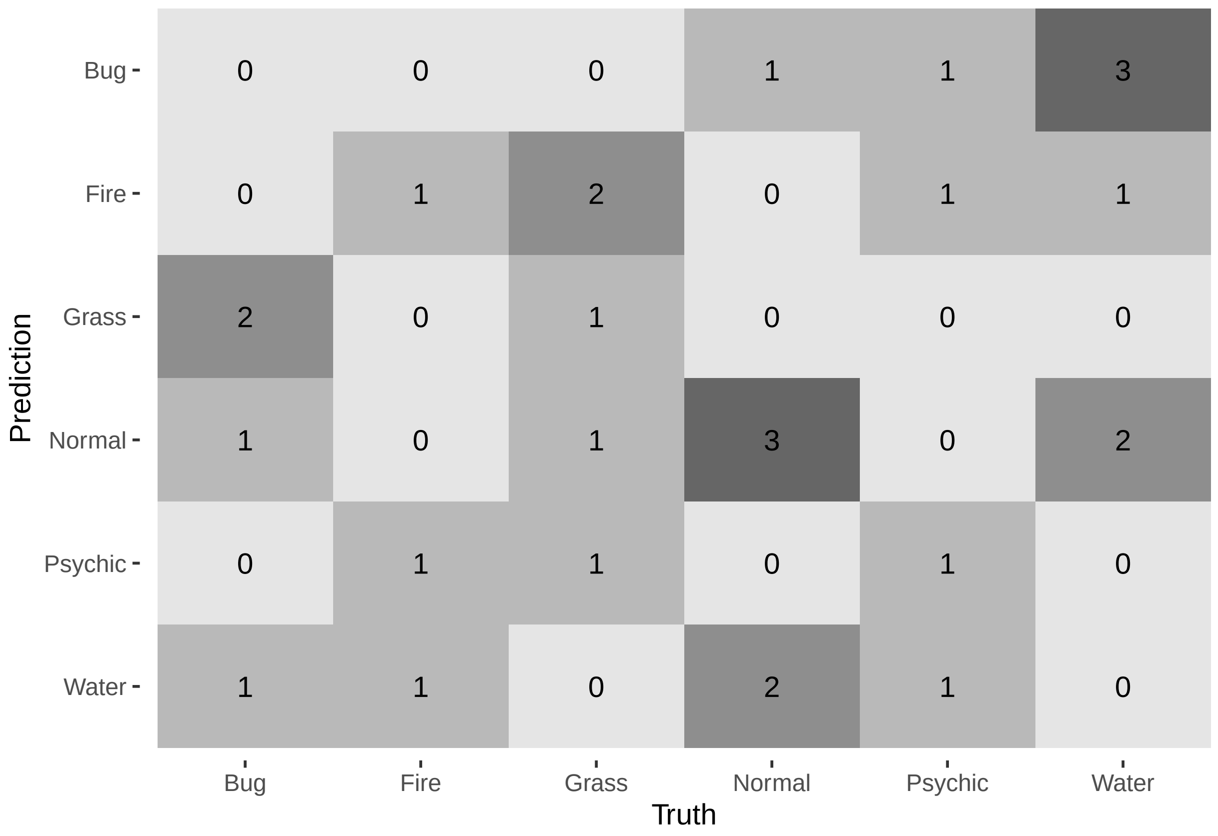
Then we'll create plots of the different ROC curves, one per level of the outcome.

```
roc %>%
  roc_curve(type_1, c(.pred_Bug, .pred_Fire, .pred_Grass, .pred_Normal,
                      .pred_Psychic, .pred_Water)) %>%
  autoplot()
```



Finally, we'll also make a heat map of the confusion matrix.

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



As we can see from our overall ROC AUC value of 0.5879 on the testing dataset, our model did not do too great. Generally, AUC values between 0.6 and 0.5 are considered to be poor results. However, our model did a surprisingly good job at predicting Pokemon of types Normal and Fire, while doing a worse job of predicting Pokemon of types Bug and Water. Since Normal type is the second most common Pokemon type in our dataset, it makes sense that our modle could predict it better since it has more training data to work with in that category. However, this contradicts the fact that Water type is the most common but has one of the worst ROC AUC values. The most likely reason for this, is that Water types have a large variety of possible secondary types (`type_2`), which is most likely interfering with the prediction quality of our model. This is confirmed when we look at Fire type, which has a smaller variety of second types.