

# PSTAT131HW06

Yifei Zhang

2022-05-24

```
library(ggplot2)
library(tidyverse)
library(tidymodels)
library(corrplot)
library(klaR)
library(glmnet)
tidymodels_prefer()
Pokemon <- read_csv("Pokemon.csv")
library(janitor)
library(janitor)
library(randomForest)
library(xgboost)
library(rpart.plot)
```

## Exercise 1

Read in the data and set things up as in Homework 5:

- Use `clean_names()`
- Filter out the rarer Pokémon types
- Convert `type_1` and `legendary` to factors

```
cleaned <- clean_names(Pokemon)
cleaned
```

```
## # A tibble: 800 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      4 Venusaur~ Grass Poison  625   80  100  123  122  120   80
## 5      5 Charmand~ Fire  <NA>   309   39   52   43    60    50   65
## 6      6 Charmele~ Fire  <NA>   405   58   64   58    80    65   80
## 7      7 Charizard Fire  Flying  534   78   84   78   109    85  100
## 8      8 Charizar~ Fire  Dragon  634   78  130  111   130    85  100
## 9      9 Charizar~ Fire  Flying  634   78  104   78   159   115  100
## 10     7 Squirtle Water <NA>   314   44   48   65    50    64   43
## # ... with 790 more rows, and 2 more variables: generation <dbl>,
## #   legendary <lgl>
```

```

filtered <- cleaned %>% filter(
  type_1 == "Bug" | type_1 == "Fire" | type_1 == "Grass"
  | type_1 == "Normal" | type_1 == "Water" | type_1 == "Psychic"
)
filtered

```

```

## # A tibble: 458 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 Venusaur~ Grass  Poison  625    80   100   123   122   120    80
## 5      4 Charmand~ Fire   <NA>   309    39    52    43    60    50    65
## 6      5 Charmele~ Fire   <NA>   405    58    64    58    80    65    80
## 7      6 Charizard Fire   Flying  534    78    84    78   109    85   100
## 8      6 Charizar~ Fire   Dragon  634    78   130   111   130    85   100
## 9      6 Charizar~ Fire   Flying  634    78   104    78   159   115   100
## 10     7 Squirtle  Water  <NA>   314    44    48    65    50    64    43
## # ... with 448 more rows, and 2 more variables: generation <dbl>,
## #   legendary <lgl>

```

```

data <- filtered %>%
  mutate(type_1 = factor(type_1),
         legendary = factor(legendary),
         generation = factor(generation)
  )
data

```

```

## # A tibble: 458 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 Venusaur~ Grass  Poison  625    80   100   123   122   120    80
## 5      4 Charmand~ Fire   <NA>   309    39    52    43    60    50    65
## 6      5 Charmele~ Fire   <NA>   405    58    64    58    80    65    80
## 7      6 Charizard Fire   Flying  534    78    84    78   109    85   100
## 8      6 Charizar~ Fire   Dragon  634    78   130   111   130    85   100
## 9      6 Charizar~ Fire   Flying  634    78   104    78   159   115   100
## 10     7 Squirtle  Water  <NA>   314    44    48    65    50    64    43
## # ... with 448 more rows, and 2 more variables: generation <fct>,
## #   legendary <fct>

```

Do an initial split of the data; you can choose the percentage for splitting. Stratify on the outcome variable.

```

set.seed(2022)
pokemon_split <- data %>%
  initial_split(strata = type_1, prop = 0.75)
pokemon_train <- training(pokemon_split)
pokemon_test <- testing(pokemon_split)
dim(pokemon_train)

```

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

Fold the training set using  $v$ -fold cross-validation, with  $v = 5$ . Stratify on the outcome variable.

```
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 [270/71]> Fold1
## 2 <split [271/70]> Fold2
## 3 <split [273/68]> Fold3
## 4 <split [274/67]> Fold4
## 5 <split [276/65]> Fold5
```

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_def,
                          data = pokemon_train) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_normalize(all_predictors())
```

## Exercise 2

Create a correlation matrix of the training set, using the `corrplot` package. *Note: You can choose how to handle the continuous variables for this plot; justify your decision(s).*

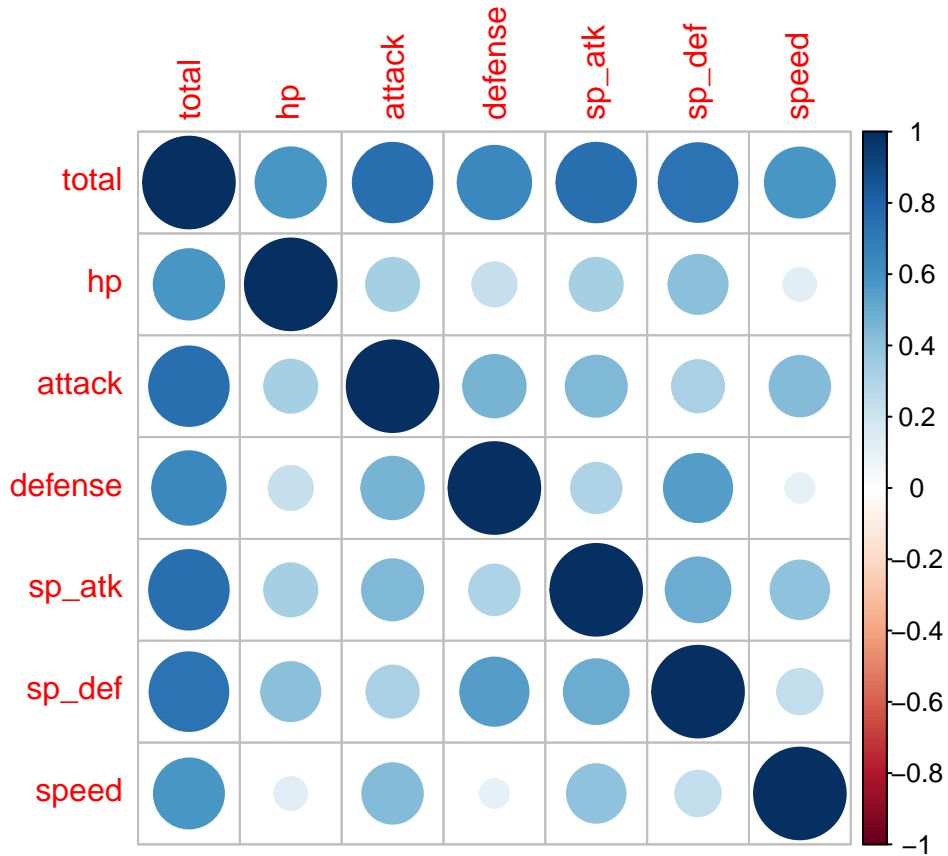
I dropped all non numeric variables, because they can not fit into the correlation function. I think the hw question is phrased wrong. We are eliminating the non continuous variables.

```
cordata <- subset (data, select = c( "total", "hp", "attack",
                                     "defense", "sp_atk", "sp_def",
                                     "speed"))
```

```
res <- cor(cordata)
res
```

```
##           total      hp  attack  defense  sp_atk  sp_def  speed
## total  1.0000000 0.5891580 0.7517256 0.6413857 0.7521862 0.7361142 0.5831479
## hp      0.5891580 1.0000000 0.3352739 0.2321308 0.3353892 0.4154919 0.1294510
## attack  0.7517256 0.3352739 1.0000000 0.4645177 0.4437926 0.3232667 0.4347855
## defense 0.6413857 0.2321308 0.4645177 1.0000000 0.3038973 0.5534584 0.1015110
## sp_atk  0.7521862 0.3353892 0.4437926 0.3038973 1.0000000 0.4992364 0.4077537
## sp_def  0.7361142 0.4154919 0.3232667 0.5534584 0.4992364 1.0000000 0.2471261
## speed   0.5831479 0.1294510 0.4347855 0.1015110 0.4077537 0.2471261 1.0000000
```

```
corrplot(res, method = "circle")
```



What relationships, if any, do you notice? Do these relationships make sense to you?

All the variables have positive correlation with each other to some degree. All the variables have a positive correlation with total which makes sense. Other than that, speed is pretty positively correlated with attack and defense which also makes sense. Attack is positively correlated with sp attack and defense. Defense is also positively correlated with sp defense. They all make sense.

### Exercise 3

First, set up a decision tree model and workflow. Tune the `cost_complexity` hyperparameter. Use the same levels we used in Lab 7 – that is, `range = c(-3, -1)`. Specify that the metric we want to optimize is `roc_auc`.

Print an `autoplot()` of the results. What do you observe? Does a single decision tree perform better with a smaller or larger complexity penalty?

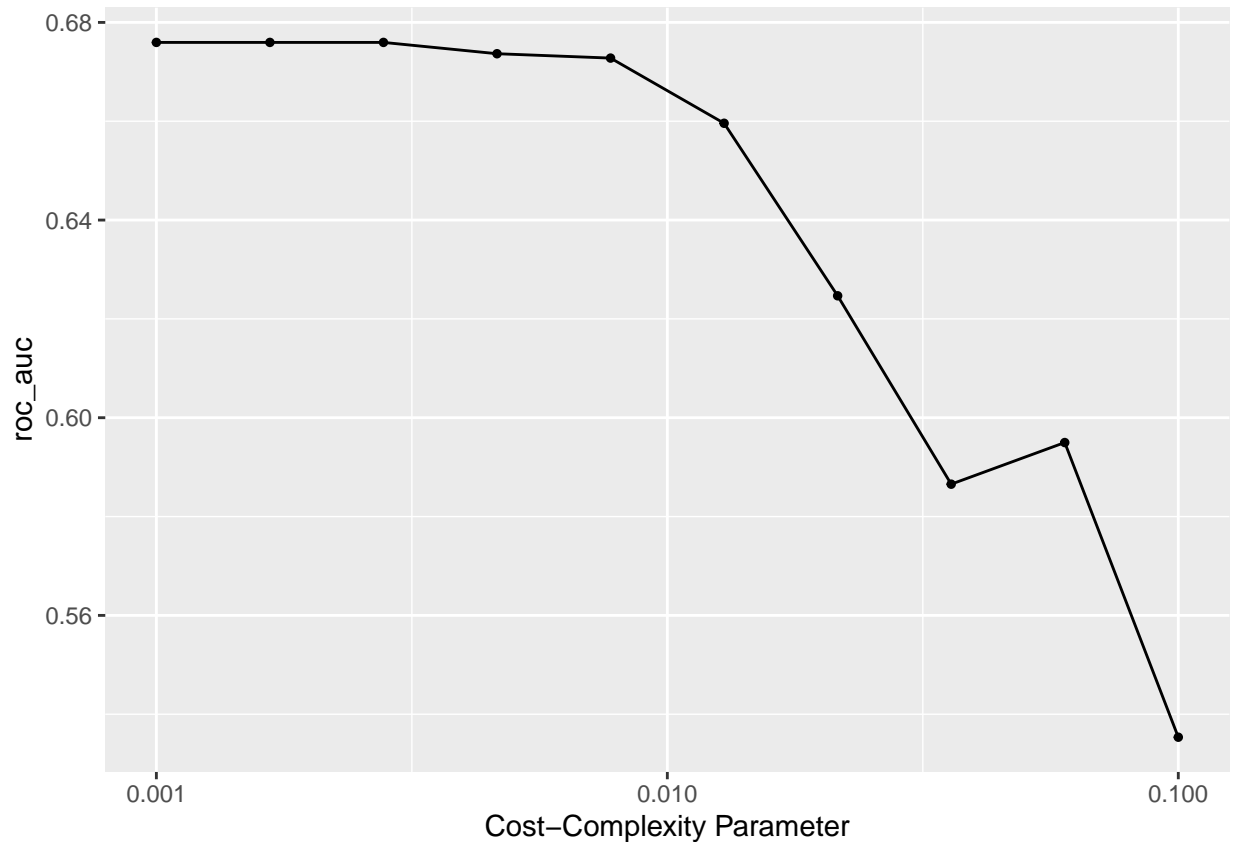
```
tree_spec <- decision_tree() %>%
  set_engine("rpart")

class_tree_spec <- tree_spec %>%
  set_mode("classification")

class_tree_fit <- class_tree_spec %>%
  fit(type_1 ~ legendary + generation + sp_atk +
```



```
autoplot(tune_res)
```



It is pretty steady at the beginning, and then it drastically dropped. It performs better with a lower complexity penalty, it will plumb if it is too large.

#### Exercise 4

What is the `roc_auc` of your best-performing pruned decision tree on the folds? *Hint: Use `collect_metrics()` and `arrange()`.*

The `roc_auc` of your best-performing pruned decision tree on the folds is 0.6759523

```
collect_metrics(tune_res) %>% arrange()
```

```
## # A tibble: 10 x 7
##   cost_complexity .metric .estimator mean      n std_err .config
##           <dbl> <chr>   <chr>    <dbl> <int>   <dbl> <chr>
## 1      0.001   roc_auc hand_till 0.676     5 0.0188 Preprocessor1_Model01
## 2     0.00167 roc_auc hand_till 0.676     5 0.0188 Preprocessor1_Model02
## 3     0.00278 roc_auc hand_till 0.676     5 0.0188 Preprocessor1_Model03
## 4     0.00464 roc_auc hand_till 0.674     5 0.0168 Preprocessor1_Model04
## 5     0.00774 roc_auc hand_till 0.673     5 0.0189 Preprocessor1_Model05
## 6     0.0129  roc_auc hand_till 0.660     5 0.0204 Preprocessor1_Model06
## 7     0.0215  roc_auc hand_till 0.625     5 0.0122 Preprocessor1_Model07
## 8     0.0359  roc_auc hand_till 0.587     5 0.00577 Preprocessor1_Model08
```

```
## 9      0.0599 roc_auc hand_till 0.595      5 0.0140 Preprocessor1_Model09
## 10     0.1      roc_auc hand_till 0.535      5 0.0218 Preprocessor1_Model10
```

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

```
## # A tibble: 1 x 2
##   cost_complexity .config
##         <dbl> <chr>
## 1      0.001 Preprocessor1_Model01
```

## Exercise 5

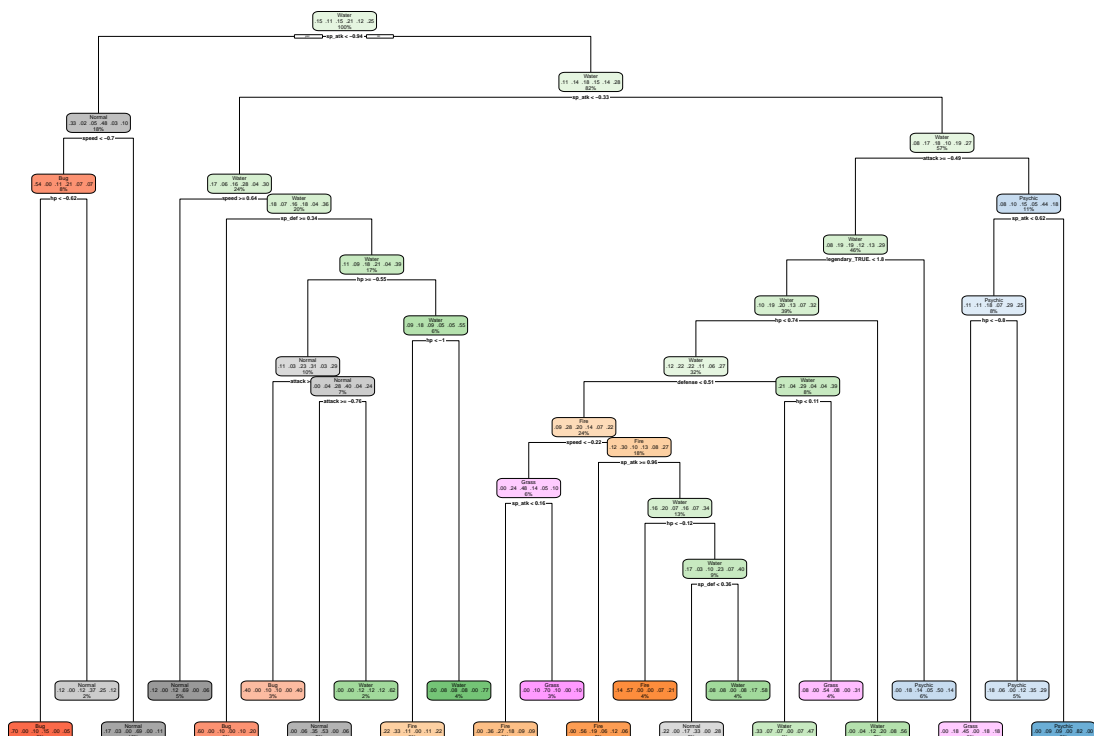
Using `rpart.plot`, fit and visualize your best-performing pruned decision tree with the *training* set.

```
best_complexity <- select_best(tune_res)
```

```
class_tree_final <- finalize_workflow(class_tree_wf, best_complexity)
```

```
class_tree_final_fit <- fit(class_tree_final, data = pokemon_train)
```

```
class_tree_final_fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```



### Exercise 5

Now set up a random forest model and workflow. Use the `ranger` engine and set `importance = "impurity"`. Tune `mtry`, `trees`, and `min_n`. Using the documentation for `rand_forest()`, explain in your own words what each of these hyperparameters represent.

Create a regular grid with 8 levels each. You can choose plausible ranges for each hyperparameter. Note that `mtry` should not be smaller than 1 or larger than 8. **Explain why not. What type of model would `mtry = 8` represent?**

### Exercise 6

Specify `roc_auc` as a metric. Tune the model and print an `autoplot()` of the results. What do you observe? What values of the hyperparameters seem to yield the best performance?

### Exercise 7

What is the `roc_auc` of your best-performing random forest model on the folds? *Hint: Use `collect_metrics()` and `arrange()`.*

### Exercise 8

Create a variable importance plot, using `vip()`, with your best-performing random forest model fit on the *training* set.

Which variables were most useful? Which were least useful? Are these results what you expected, or not?

### Exercise 9

Finally, set up a boosted tree model and workflow. Use the `xgboost` engine. Tune `trees`. Create a regular grid with 10 levels; let `trees` range from 10 to 2000. Specify `roc_auc` and again print an `autoplot()` of the results.

What do you observe?

What is the `roc_auc` of your best-performing boosted tree model on the folds? *Hint: Use `collect_metrics()` and `arrange()`.*

### Exercise 10

Display a table of the three ROC AUC values for your best-performing pruned tree, random forest, and boosted tree models. Which performed best on the folds? Select the best of the three and use `select_best()`, `finalize_workflow()`, and `fit()` to fit it to the *testing* set.

Print the AUC value of your best-performing model on the testing set. Print the ROC curves. Finally, create and visualize a confusion matrix heat map.

Which classes was your model most accurate at predicting? Which was it worst at?