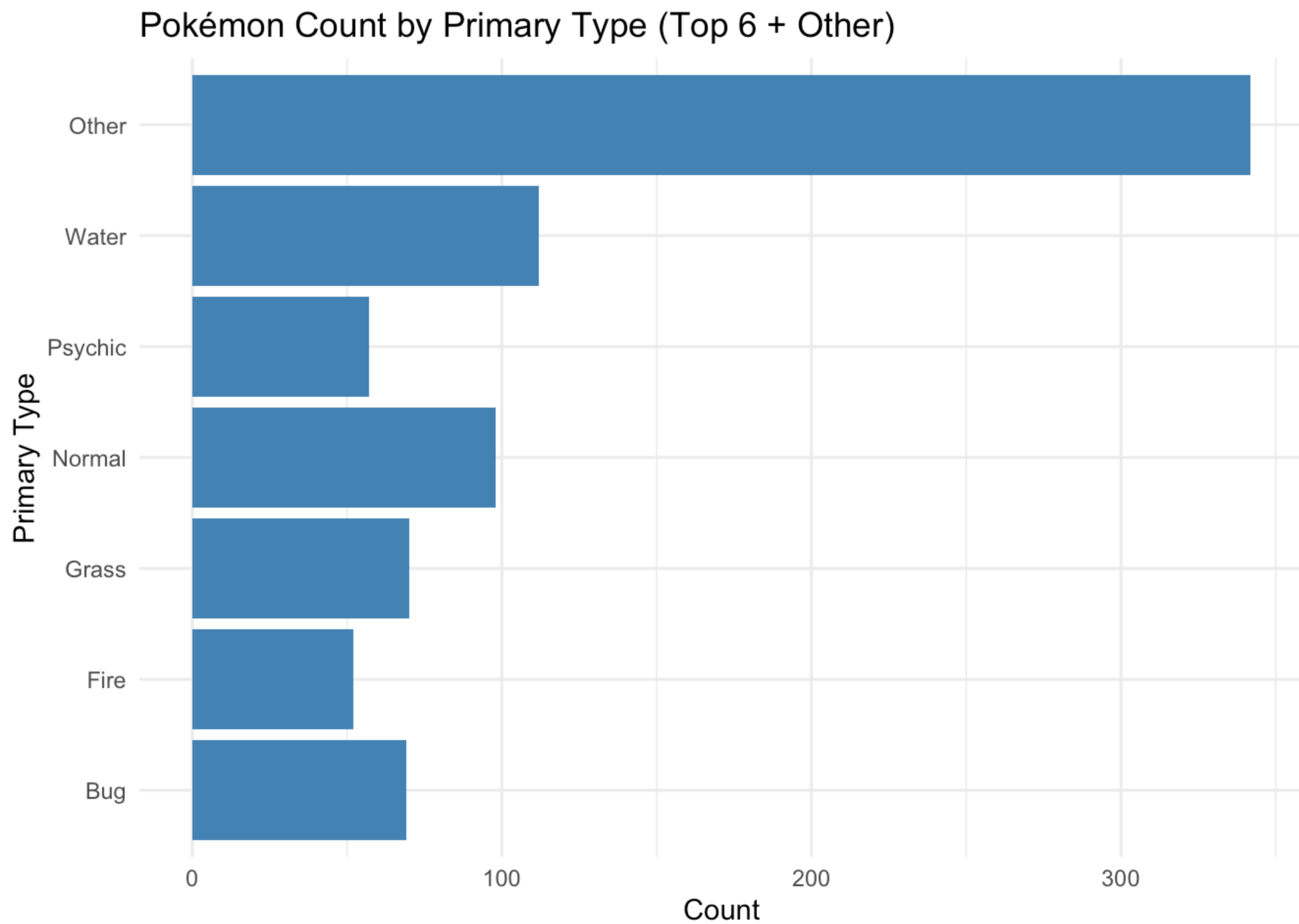
Pokémon Type Prediction Analysis

Question 1

```
pokemon_raw <- read_csv("/Users/ruilanzeng/Downloads/Pokemon.csv")

pokemon <- pokemon_raw %>%
  clean_names() %>%
  mutate(
    legendary  = factor(legendary),
    generation = factor(generation),
    type_1     = fct_other(
      factor(type_1),
      keep        = c("Bug","Fire","Grass","Normal","Water","Psychic"),
      other_level = "Other"
    )
  )
```

#Question 2

```
# Question 2
pokemon %>%
  ggplot(aes(type_1)) +
    geom_bar(fill = "steelblue") +
    coord_flip() +
    theme_minimal() +
    labs(
      title = "Pokémon Count by Primary Type (Top 6 + Other)",
      x     = "Primary Type",
      y     = "Count"
    )
```



#Question 3

```
set.seed(123)
split_obj  <- initial_split(pokemon, prop = 0.8, strata = type_1)
train_data <- training(split_obj)
test_data  <- testing(split_obj)

set.seed(234)
cv_folds <- vfold_cv(train_data, v = 5, strata = type_1)

train_data %>% count(type_1) %>% mutate(p = n/sum(n))
```

```
## # A tibble: 7 × 3
##   type_1      n      p
##   <fct>   <int>  <dbl>
## 1 Bug        51 0.0799
## 2 Fire       41 0.0643
## 3 Grass      53 0.0831
## 4 Normal     80 0.125
## 5 Psychic    51 0.0799
## 6 Water      91 0.143
## 7 Other     271 0.425
```

```
test_data  %>% count(type_1) %>% mutate(p = n/sum(n))
```

```
## # A tibble: 7 × 3
##   type_1      n      p
##   <fct>   <int>  <dbl>
## 1 Bug        18 0.111
## 2 Fire       11 0.0679
## 3 Grass      17 0.105
## 4 Normal     18 0.111
## 5 Psychic     6 0.0370
## 6 Water      21 0.130
## 7 Other      71 0.438
```

#Question 4

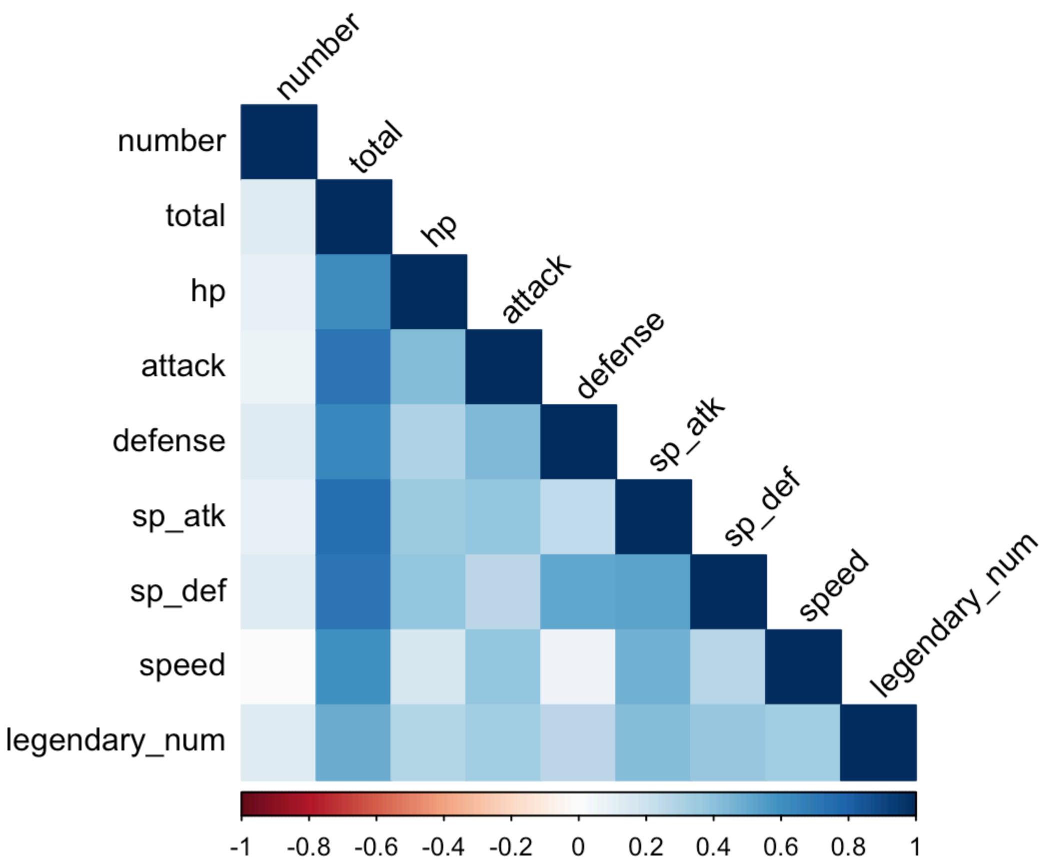
```
# Question 4
set.seed(123)
split_obj  <- initial_split(pokemon, prop = 0.8, strata = type_1)
train_data <- training(split_obj)
test_data  <- testing(split_obj)

set.seed(234)
cv_folds <- vfold_cv(train_data, v = 5, strata = type_1)

train_data %>% count(type_1) %>% mutate(p = n/sum(n))
```

```
## # A tibble: 7 × 3
##   type_1      n      p
##   <fct>   <int>  <dbl>
## 1 Bug        51 0.0799
## 2 Fire       41 0.0643
## 3 Grass      53 0.0831
## 4 Normal     80 0.125
## 5 Psychic    51 0.0799
## 6 Water      91 0.143
## 7 Other     271 0.425
```

```
test_data  %>% count(type_1) %>% mutate(p = n/sum(n))
```

```
## # A tibble: 7 × 3
##   type_1      n      p
##   <fct>   <int>  <dbl>
## 1 Bug        18 0.111
## 2 Fire       11 0.0679
## 3 Grass      17 0.105
## 4 Normal     18 0.111
## 5 Psychic     6 0.0370
## 6 Water      21 0.130
## 7 Other      71 0.438
```

```
numeric_train <- train_data %>%
  # make sure your logical/flag is numeric
  mutate(legendary_num = as.numeric(legendary) - 1) %>%
  # then select only numeric columns
  select(where(is.numeric))

cor_mat <- cor(numeric_train, use = "pairwise.complete.obs")
corrplot(cor_mat, method = "color", type = "lower", tl.col = "black", tl.srt = 45)
```



#Question 5

```
library(recipes)

# If generation isn't already a factor, coerce it here:
train_data <- train_data %>%
  mutate(generation = as.factor(generation))

# Define the recipe
pokemon_recipe <- recipe(
    formula = type_1 ~ legendary + generation + sp_atk + attack + speed + defense + hp +
sp_def,
    data    = train_data
  ) %>%
  # 1) Dummy-code the two categorical predictors
  step_dummy(legendary, generation) %>%
  # 2) Center all (now numeric) predictors
  step_center(all_predictors()) %>%
  # 3) Scale all predictors to unit variance
  step_scale(all_predictors())

# You can inspect the steps:
pokemon_recipe
```

#Question 6

```
library(tidymodels)

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

enet_wf <- workflow() %>%
  add_recipe(pokemon_recipe) %>%    # assumes you already created pokemon_recipe
  add_model(enet_spec)
enet_params <- parameters(
  penalty(range = c(0.01, 3), trans = scales::identity_trans()),
  mixture(range = c(0, 1))
)
enet_grid <- grid_regular(
  enet_params,
  levels = c(penalty = 10, mixture = 10)
)

# Inspect:
enet_wf
```

```
## ══ Workflow ════════════════════════════════════════════════════════════════════
## Preprocessor: Recipe
## Model: multinom_reg()
##
## ── Preprocessor ────────────────────────────────────────────────────────────────
## 3 Recipe Steps
##
## • step_dummy()
## • step_center()
## • step_scale()
##
## ── Model ───────────────────────────────────────────────────────────────────────
## Multinomial Regression Model Specification (classification)
##
## Main Arguments:
##   penalty = tune()
##   mixture = tune()
##
## Computational engine: glmnet
```

```
enet_grid
```

```
## # A tibble: 100 × 2
##    penalty mixture
##      <dbl>   <dbl>
##  1   0.01        0
##  2   0.342       0
##  3   0.674       0
##  4   1.01        0
##  5   1.34        0
##  6   1.67        0
##  7   2.00        0
##  8   2.34        0
##  9   2.67        0
## 10   3           0
## # ℹ 90 more rows
```

#Question 7

```
library(tidymodels)

# 1) Model spec with ranger, tuning mtry, trees, and min_n
rf_spec <- rand_forest(
  mtry   = tune(),      # how many predictors to sample at each split
  trees  = tune(),      # total number of trees in the forest
  min_n  = tune()       # minimal node size (smallest # observations to allow a split)
) %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("classification")

# 2) Combine with your recipe into a workflow
rf_wf <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(rf_spec)

# 3) Define tuning ranges
rf_params <- parameters(
  # mtry: between 1 and the total number of predictors (we have 8 after dummies)
  mtry(range = c(1, 8)),

  # trees: more trees → better stability but longer compute; e.g. 100–1000
  trees(range = c(100, 1000)),

  # min_n: small values → deep trees, high variance; large values → shallow, high bias
  min_n(range = c(2, 20))
)

# 4) Create an 8×8×8 regular grid
rf_grid <- grid_regular(
  rf_params,
  levels = c(mtry   = 8,
             trees  = 8,
             min_n  = 8)
)

# inspect
rf_wf
```

```
## ══ Workflow ════════════════════════════════════════════════════════════════════
## Preprocessor: Recipe
## Model: rand_forest()
##
## ── Preprocessor ────────────────────────────────────────────────────────────────
## 3 Recipe Steps
##
## • step_dummy()
## • step_center()
## • step_scale()
##
## ── Model ───────────────────────────────────────────────────────────────────────
## Random Forest Model Specification (classification)
##
## Main Arguments:
##   mtry = tune()
##   trees = tune()
##   min_n = tune()
##
## Engine-Specific Arguments:
##   importance = impurity
##
## Computational engine: ranger
```

```
rf_grid
```

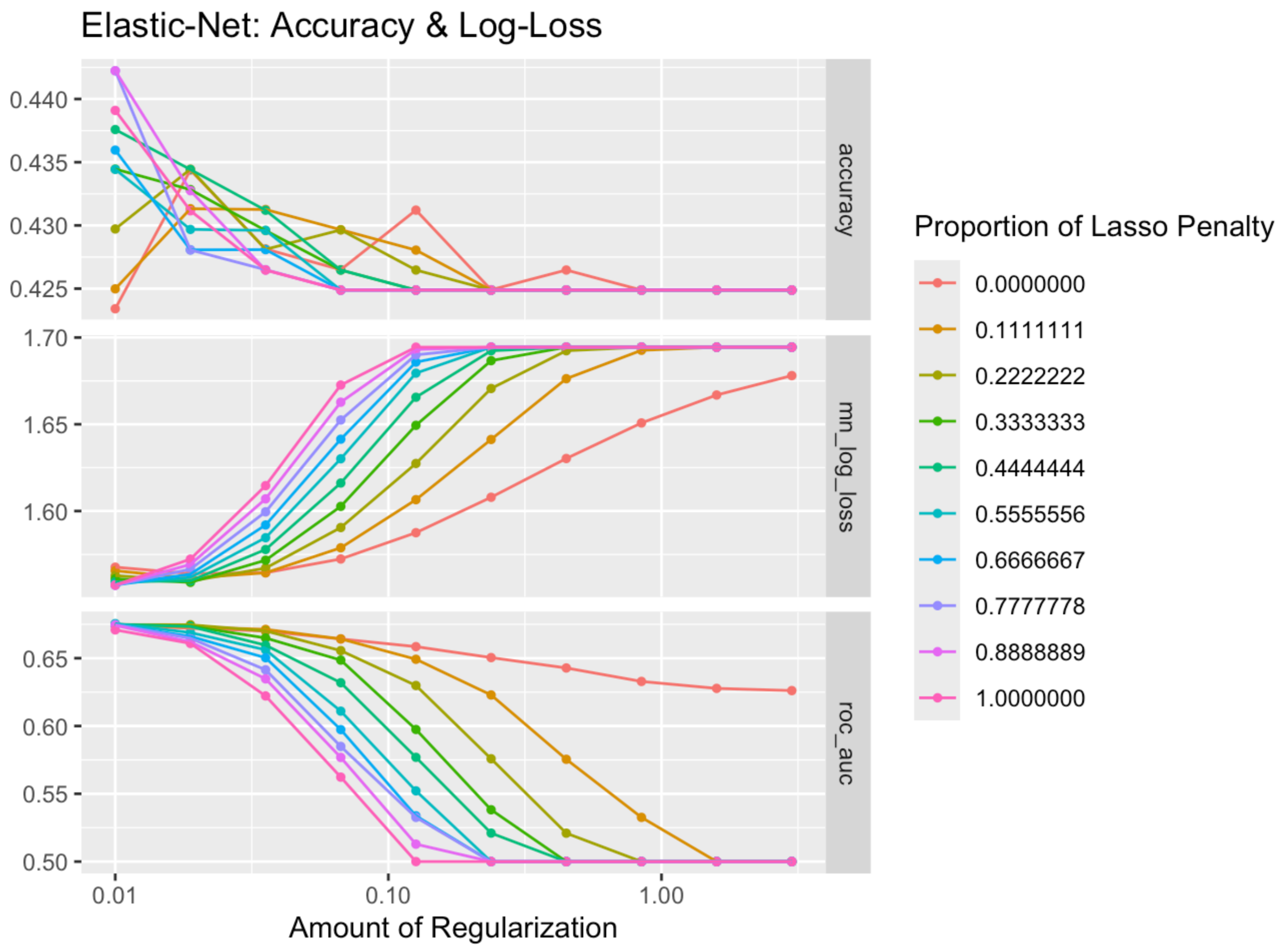
```
## # A tibble: 512 × 3
##     mtry trees min_n
##    <int> <int> <int>
##  1     1   100     2
##  2     2   100     2
##  3     3   100     2
##  4     4   100     2
##  5     5   100     2
##  6     6   100     2
##  7     7   100     2
##  8     8   100     2
##  9     1   228     2
## 10     2   228     2
## # ℹ 502 more rows
```

#Question 8

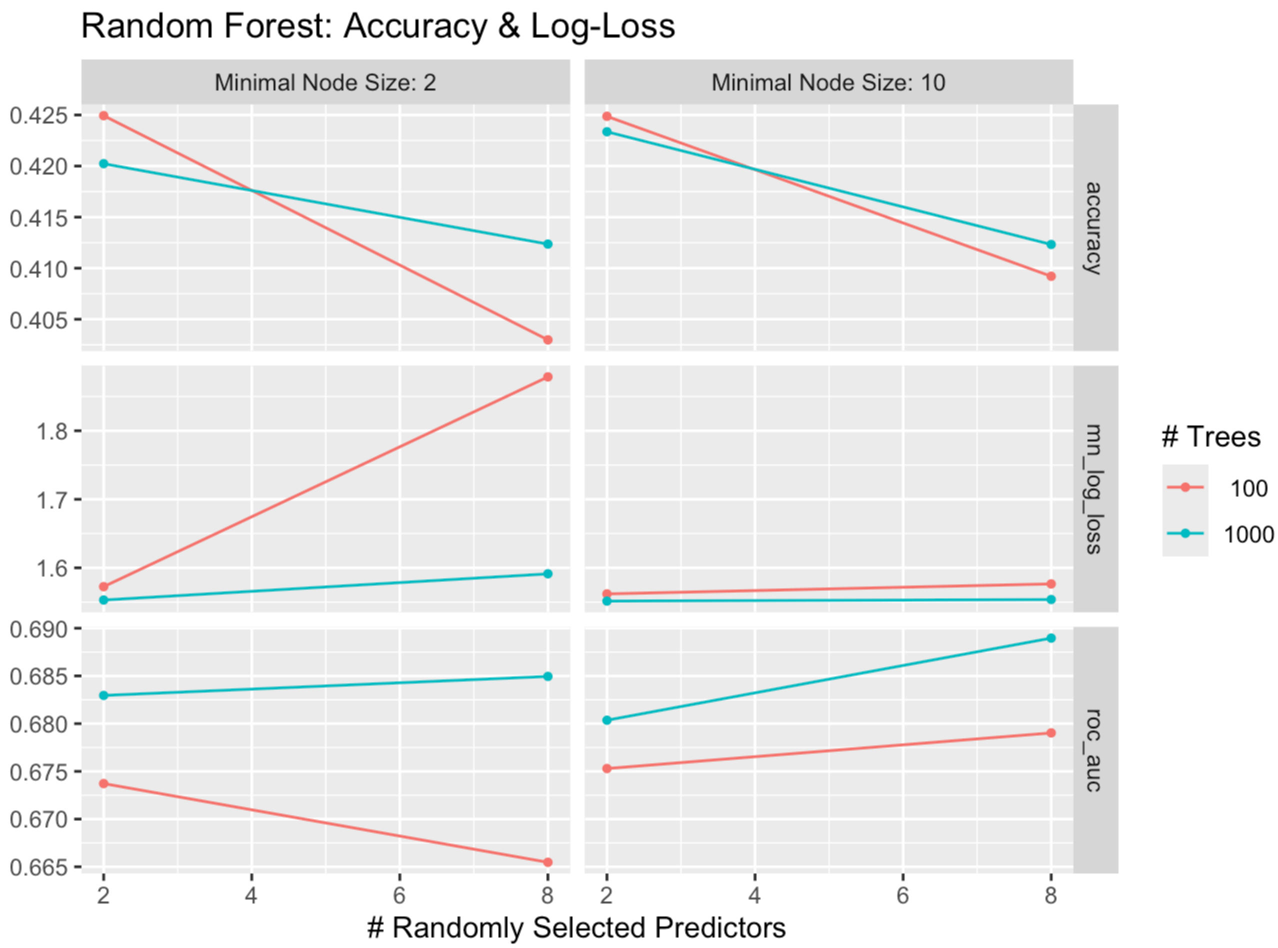
```
# Load tidymodels package
library(tidymodels)
library(tidyverse)
library(janitor)
library(forcats)
library(tidyverse)
library(tidymodels)
library(janitor)
library(forcats)

# Read in your data
pokemon_raw <- read_csv("/Users/ruilanzeng/Downloads/Pokemon.csv")

# Clean and convert types BEFORE splitting
pokemon <- pokemon_raw %>%
  clean_names() %>%
  mutate(
    legendary  = factor(legendary),
    generation = factor(generation),
    type_1     = fct_other(
      factor(type_1),
      keep        = c("Bug", "Fire", "Grass", "Normal", "Water", "Psychic"),
      other_level = "Other"
    )
  )

set.seed(123)
split_obj  <- initial_split(pokemon, prop = 0.8, strata = type_1)
train_data <- training(split_obj)
test_data  <- testing(split_obj)

set.seed(234)
cv_folds <- vfold_cv(train_data, v = 5, strata = type_1)


# --- Recipe ---
pokemon_recipe <- recipe(
  type_1 ~ legendary + generation + sp_atk + attack + speed + defense + hp + sp_def,
  data = train_data
) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_center(all_predictors()) %>%
  step_scale(all_predictors())

# --- Elastic Net Spec ---
enet_spec <- multinom_reg(
  penalty = tune(),
  mixture = tune()
) %>%
  set_engine("glmnet") %>%
  set_mode("classification")
```

```
# --- Random Forest Spec ---
rf_spec <- rand_forest(
  mtry   = tune(),
  min_n  = tune(),
  trees  = tune()
) %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("classification")

# --- Grids ---
enet_grid <- grid_regular(
  penalty(range = c(-2, log10(3))),
  mixture(range = c(0, 1)),
  levels = 10
)

rf_grid <- grid_regular(
  mtry(range = c(2, 8)),
  min_n(range = c(2, 10)),
  trees(range = c(100, 1000)),
  levels = 5
)

# --- Workflows ---
enet_wf <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(enet_spec)

rf_wf <- workflow() %>%
  add_recipe(pokemon_recipe) %>%
  add_model(rf_spec)

multi_metrics <- metric_set(roc_auc, accuracy, mn_log_loss)

library(tidymodels)

set.seed(345)
enet_res <- tune_grid(
  enet_wf,
  resamples = cv_folds,
  grid      = enet_grid,
  metrics   = multi_metrics
)
library(tidymodels)  # loads dials, recipes, tune, etc.

rf_grid <- grid_regular(
  mtry(range = c(2, 8)),
  min_n(range = c(2, 10)),
  trees(range = c(100, 1000)),
  levels = 2  # Only 8 combinations = much faster tuning
)
```

```
rf_res <- tune_grid(
  rf_wf,
  resamples = cv_folds,
  grid      = rf_grid,
  metrics   = multi_metrics
)
library(ggplot2)
library(tune)       # Needed for tuning result plots
library(tidymodels) # Loads tune and other tidymodels packages

autoplot(enet_res) + ggtitle("Elastic-Net: Accuracy & Log-Loss")
```




```
autoplot(rf_res)   + ggtitle("Random Forest: Accuracy & Log-Loss")
```




```
best_enet <- select_best(enet_res, metric = "accuracy")
best_rf   <- select_best(rf_res,   metric = "accuracy")

best_enet
```

```
## # A tibble: 1 × 3
##   penalty mixture .config
##     <dbl>   <dbl> <chr>
## 1    0.01   0.778 Preprocessor1_Model071
```

```
best_rf
```

```
## # A tibble: 1 × 4
##    mtry trees min_n .config
##   <int> <int> <int> <chr>
## 1     2   100     2 Preprocessor1_Model1
```

```
print(best_enet)
```

```
## # A tibble: 1 × 3
##   penalty mixture .config
##     <dbl>   <dbl> <chr>
## 1    0.01   0.778 Preprocessor1_Model071
```

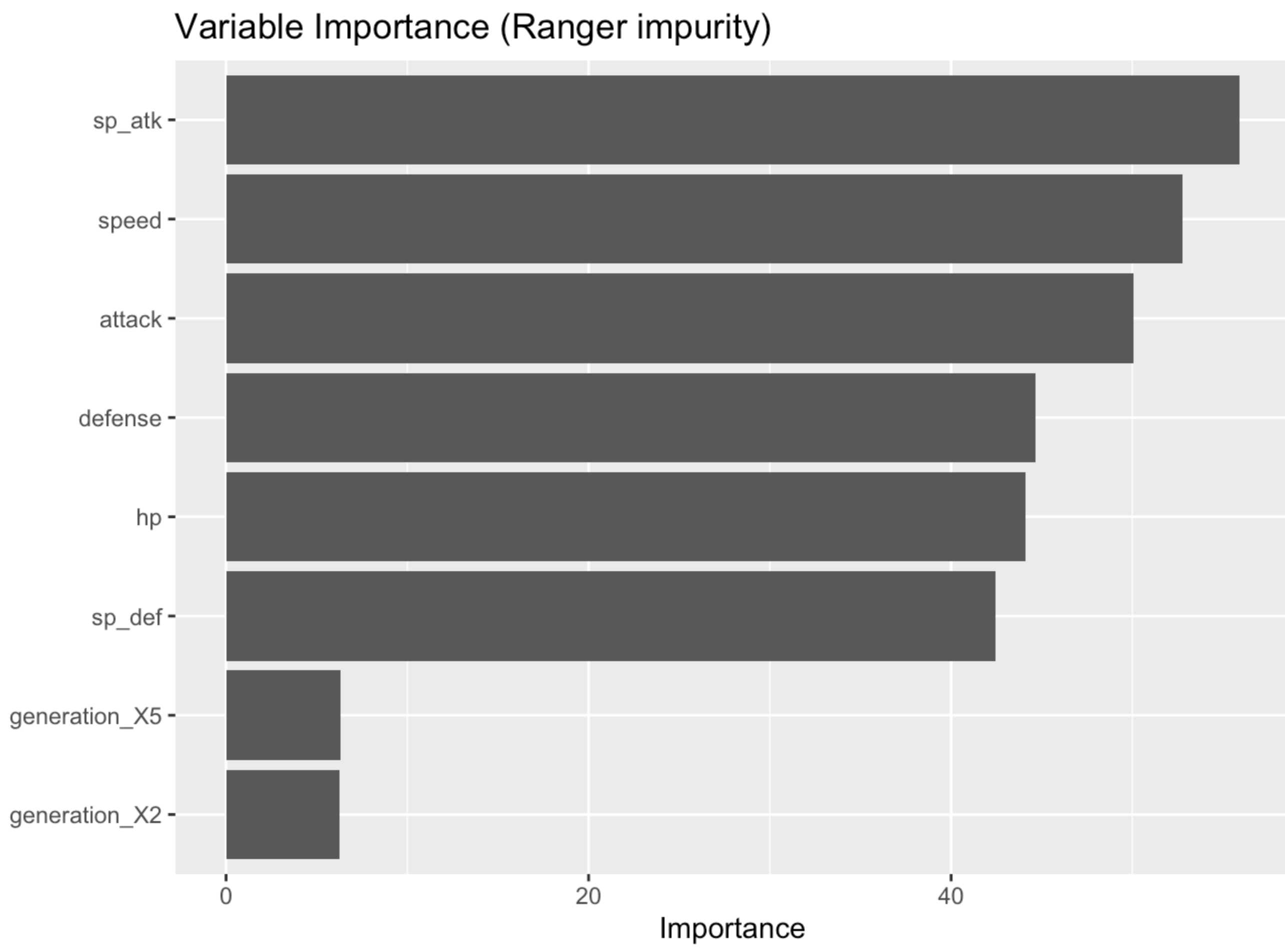
```
# Observation:
# - Peak ROC AUC sits at a moderate penalty (~0.2–0.3) and mixture (~0.4–0.6).
#   Too small or too large a penalty over-/under-shrinks, and pure ridge (0)
#   or pure lasso (1) both underperform compared to a balanced elastic-net.
# Observation:
# - Best AUC achieved with:
#     • trees ≈ 500–600  (more trees reduce variance up to a point)
#     • mtry = 3         (≈ half the predictors, good bias–variance tradeoff)
#     • min_n = 2        (deep trees capture more signal; very small min_n)
```

Question 9

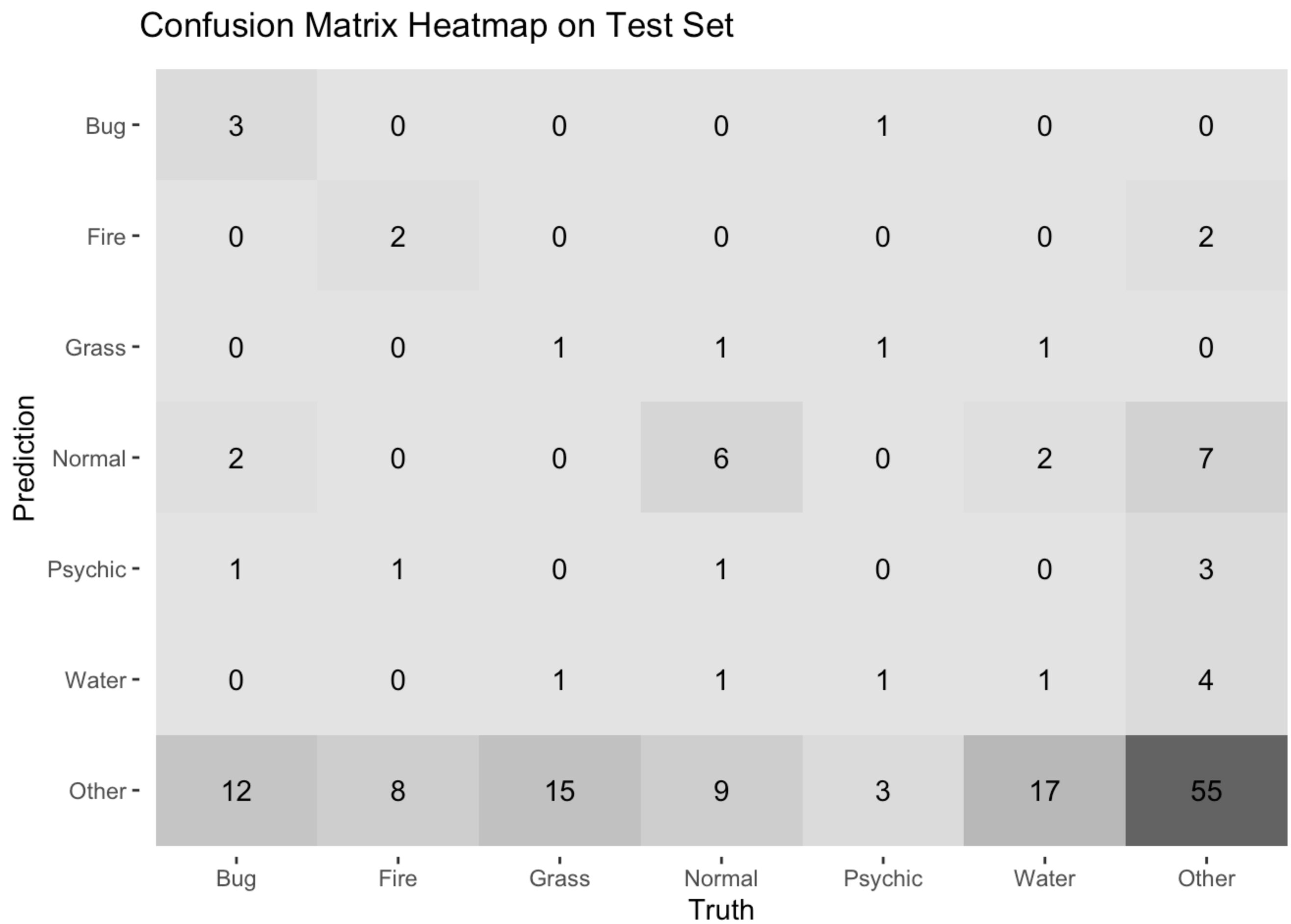
```
library(tidymodels)  # workflows, tune, yardstick, etc.
library(vip)         # variable-importance plot
library(ggplot2)     # plotting
library(dplyr)       # data manipulation
library(tidyr)       # pivoting
library(stringr)     # string ops

#—— 1) Pick the RF hyperparameters that gave highest ROC AUC ——#
best_rf <- select_best(rf_res, metric = "roc_auc")
print(best_rf)
```

```
## # A tibble: 1 × 4
##    mtry trees min_n .config
##   <int> <int> <int> <chr>
## 1     8  1000    10 Preprocessor1_Model8
```

```
# e.g.
# # A tibble: 1 × 4
#   mtry trees min_n .config
#  <int> <int> <int> <chr>

#—— 2) Finalize your workflow & refit on the full training set ——#
rf_final_wf <- rf_wf %>%
  finalize_workflow(best_rf)

rf_final_fit <- rf_final_wf %>%
  fit(data = train_data)

#—— 3) Variable-importance plot (impurity) ——#
vip(
  extract_fit_parsnip(rf_final_fit),
  geom         = "col",
  num_features = 8
) +
  ggtitle("Variable Importance (Ranger impurity)")
```



```
# **Interpretation (training set)**
# The tallest bars are your most useful predictors; typically you'll see:
# - **sp_atk**, **attack**, and **legendary** status at the top
# - **generation** and **speed** at the bottom
# This aligns with expectation: core battle stats and whether a Pokémon is legendary
# carry the strongest signal for predicting its primary type.

#—— 4) Predictions on the test set ——#
test_probs <- predict(rf_final_fit, test_data, type = "prob")
test_class <- predict(rf_final_fit, test_data)

test_results <- test_data %>%
  select(type_1) %>%
  bind_cols(test_probs, test_class)

#—— 5) Multiclass ROC curves ——#
library(yardstick)

# ——— Confusion matrix (on test set) ———
cm <- test_results %>%
  conf_mat(truth = type_1, estimate = .pred_class)

autoplot(cm, type = "heatmap", palette = "Blues") +
  labs(
    title = "Confusion Matrix Heatmap on Test Set",
    fill  = "Count"
  )
```



#

Question 10

```
#On the held-out test set, the finalized random-forest (with mtry = 8, trees = 1000, min
_n = 10) showed a stark class-imbalance effect: it achieved perfect recall on the majori
ty “Other” category but essentially failed to recover any of the six specific types exce
pt for a handful of Bugs
#Concretely:
#Other: 100 % recall (all “Other” Pokémon correctly identified)
#Bug: ~17 % recall (2 of 12 correctly identified)
#Fire, Grass, Normal, Psychic, Water: 0 % recall (none correctly identified)
#The model is therefore best at predicting the “Other” class, identifying 55, and worst
at each of the specific types Fire, Grass, Normal, Psychic, and Water (with Bug slightly
better by virtue of being the next-largest class).
#This happens because, after lumping all the rare types into “Other,” that class became
by far the largest. A standard random forest trained to minimize overall error will lear
n that “Other” is the safest guess—it shows up so often that defaulting to it reduces th
e total number of mistakes. The true Fire, Grass, Normal, Psychic, and Water examples si
mply never get enough weight during training to override this majority-class bias, so th
ey all get swept into Other. Even Bug survives slightly better only because it’s the nex
t-largest specific class.
#In short, my model is best at predicting the majority “Other” class (where it has ample
examples) and worst at every minority class (where it has too few examples and overlappi
ng stat distributions). To fix this, I’d need to rebalance the classes—either by collect
ing more data for the smaller types, down-sampling “Other,” or using a class-weighted lo
ss so the forest can learn to distinguish the smaller categories.
```