Part 5 - Imbalanced Data

Ibrahim Yazici

Load packages

We use the `tidyverse` suite of packages.

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.2     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.3     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

Read data

The code chunk below reads in the final project bonus data.

```
dfb <- readr::read_csv("paint_project_bonus_data.csv", col_names = TRUE)
```

```
## Rows: 1764 Columns: 9
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): Lightness, Saturation
## dbl (7): R, G, B, Hue, response, outcome, challenge_outcome
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

The `readr::read_csv()` function displays the data types and column names associated with the data. However, a glimpse is shown below that reveals the number of rows and also shows some of the representative values for the columns.

```
dfb %>% glimpse()
```

The data consist of continuous and categorical inputs. The `glimpse()` shown above reveals the data type for each variable which state to you whether the input is continuous or categorical. The RGB color model inputs, `R`, `G`, and `B` are continuous (dbl) inputs. The HSL color model inputs consist of 2 categorical inputs, `Lightness` and `Saturation`, and a continuous input, `Hue`.

Binary classification task

The Binary output variable, `outcome`, is a numeric variable.

```r
dfb %>% pull(outcome) %>% class()
```

```
## [1] "numeric"
```

However, there are **only** two unique values for `outcome`.

```r
dfb %>% count(outcome)
```

```
## # A tibble: 2 x 2
##   outcome     n
##     <dbl> <int>
## 1       0  1323
## 2       1   441
```

Below we create the dataset `dfbbb` for classification task with imbalanced data.

```r
dfbb <- dfb %>%
  select(-response) %>%
  mutate(challenge_outcome = ifelse(challenge_outcome == 1, 'event', 'non_event'),
         challenge_outcome = factor(challenge_outcome, levels = c('event', 'non_event')))
dfbbb <- dfbb %>%
  select(-outcome)
```

We observe counts of Lightness Categories below.

```r
ggplot(dfbbb, aes(x = Lightness)) +
  geom_bar() +
  theme_minimal() +
  labs(title = "Count of Lightness Categories", x = "Lightness", y = "Count")
```

We observe counts of Saturation Categories below.

```r
ggplot(dfbbb, aes(x = Saturation)) +
  geom_bar() +
  theme_minimal() +
  labs(title = "Count of Saturation Categories", x = "Saturation", y = "Count")
```

Below we create, prepare and apply a recipe to deal with low frequency categorical inputs and near zero variance features:

```r
library(caret)
```

```
## Loading required package: lattice

##
## Attaching package: 'caret'

## The following object is masked from 'package:purrr':
##
##     lift
```

```r
library(recipes)
```

```
##
## Attaching package: 'recipes'

## The following object is masked from 'package:stringr':
##
##     fixed

## The following object is masked from 'package:stats':
```

```
##
##     step
```

```
recipe_obj <- recipe(challenge_outcome ~ ., data = dfbbb) %>%
  step_other(all_nominal(), threshold = 0.05) %>%
  step_nzv(all_predictors())

prepped_recipe <- prep(recipe_obj, training = dfbbb)

final_data <- bake(prepped_recipe, dfbbb)
```

We observe counts of Lightness Categories in the dataset `final_data` below.

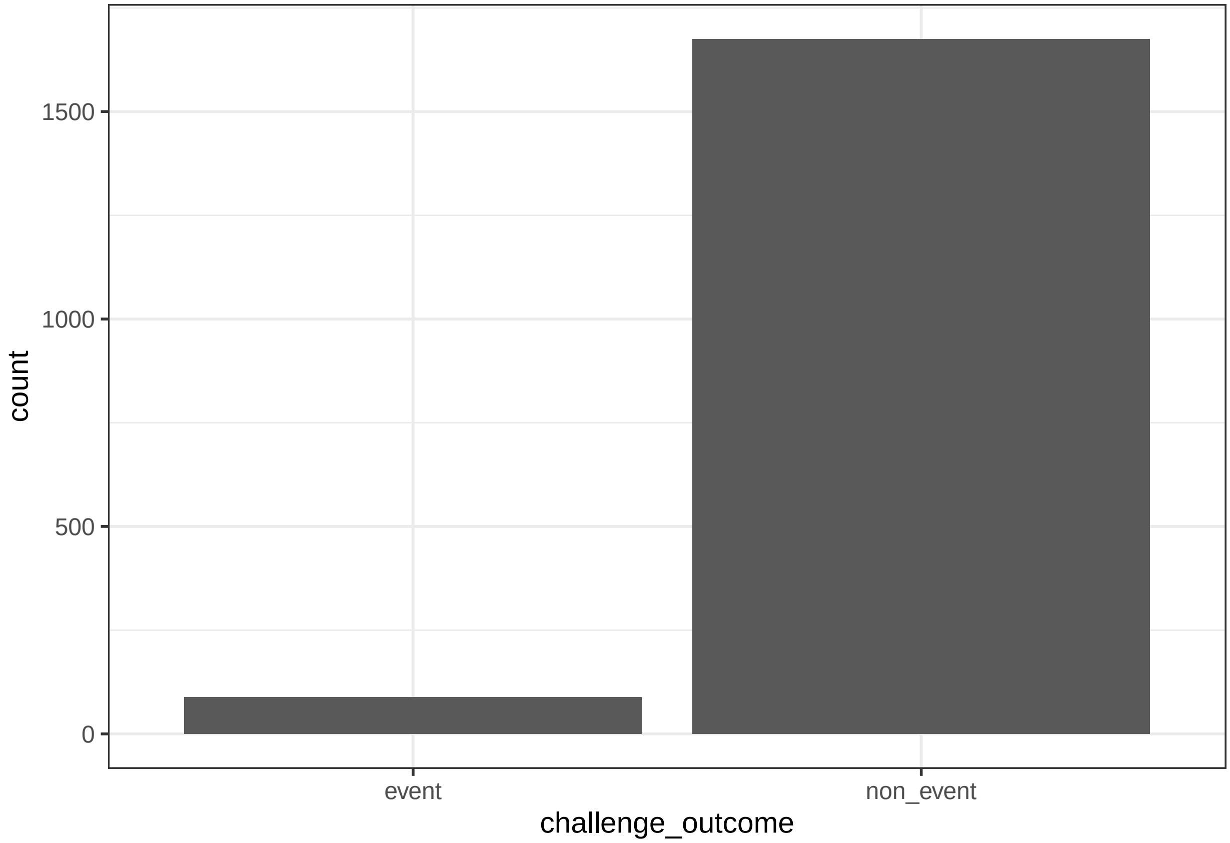
```
ggplot(final_data, aes(x = Lightness)) +
  geom_bar() +
  theme_minimal() +
  labs(title = "Count of Lightness Categories", x = "Lightness", y = "Count")
```

We observe counts of Lightness Saturation in the dataset `final_data` below.

```
ggplot(final_data, aes(x = Saturation)) +
  geom_bar() +
  theme_minimal() +
  labs(title = "Count of Saturation Categories", x = "Saturation", y = "Count")
```

We observe counts of challenge_outcome variable below.

```
final_data %>%
  ggplot(mapping = aes(x = challenge_outcome)) +
  geom_bar() +
  theme_bw()
```




```
summary(final_data$challenge_outcome)
```

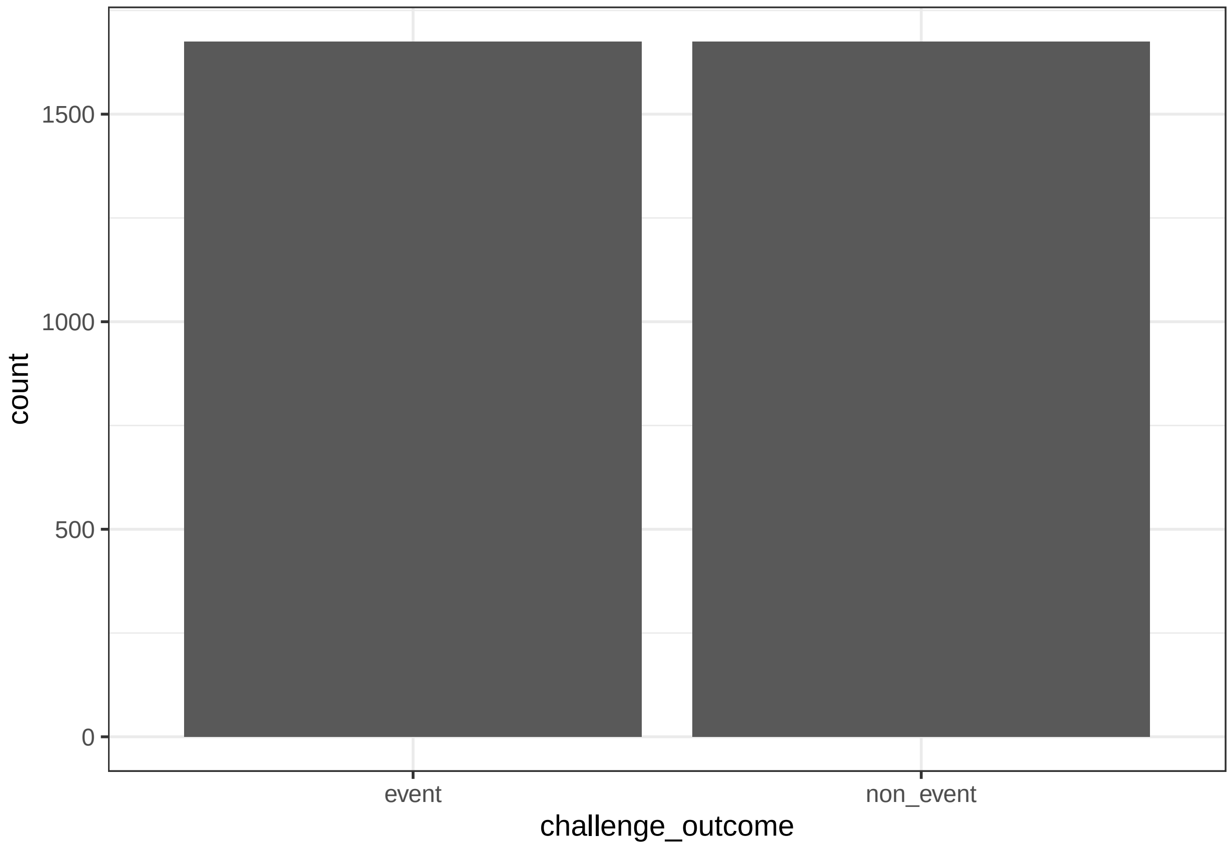
```
##     event non_event
##        89      1675
```

Below we use `upSample` method to deal with output class imbalance:

```
balanced_data <- upSample(x = final_data[, names(final_data) != "challenge_outcome"],
                          y = final_data$challenge_outcome)

final_data_balanced <- cbind(balanced_data[, -ncol(balanced_data)], challenge_outcome = balanced_data$Cl

final_data_balanced$Class <- NULL
```

We observe counts of challenge_outcome variable in the dataset `final_data_balanced` below.

```
final_data_balanced %>%
  ggplot(mapping = aes(x = challenge_outcome)) +
  geom_bar() +
  theme_bw()
```




```
summary(final_data_balanced$challenge_outcome)
```

```
##     event non_event
##      1675      1675
```

We must specify a resampling scheme and a primary performance metric. Let's use 5-fold cross-validation with 3-repeats. Our primary performance metric will be Accuracy.

```
my_ctrl <- trainControl(method = "repeatedcv", number = 5, repeats = 3)

my_metric <- "Accuracy"
```

Below we train and tune 3 models, Gradient boosted tree, Random forest, and Neural network.

```
set.seed(2023)

fit_xgb <- train(challenge_outcome ~ .,
                    data = final_data_balanced,
                    method = "xgbTree",
                    metric = my_metric,
                    trControl = my_ctrl,
                    verbosity = 0,
                    nthread = 1)
```

```
set.seed(2023)

fit_rf <- train(challenge_outcome ~ .,
                 data = final_data_balanced,
                 method = "rf",
```

```
                 metric = my_metric,
                 trControl = my_ctrl,
                 importance = TRUE)
```

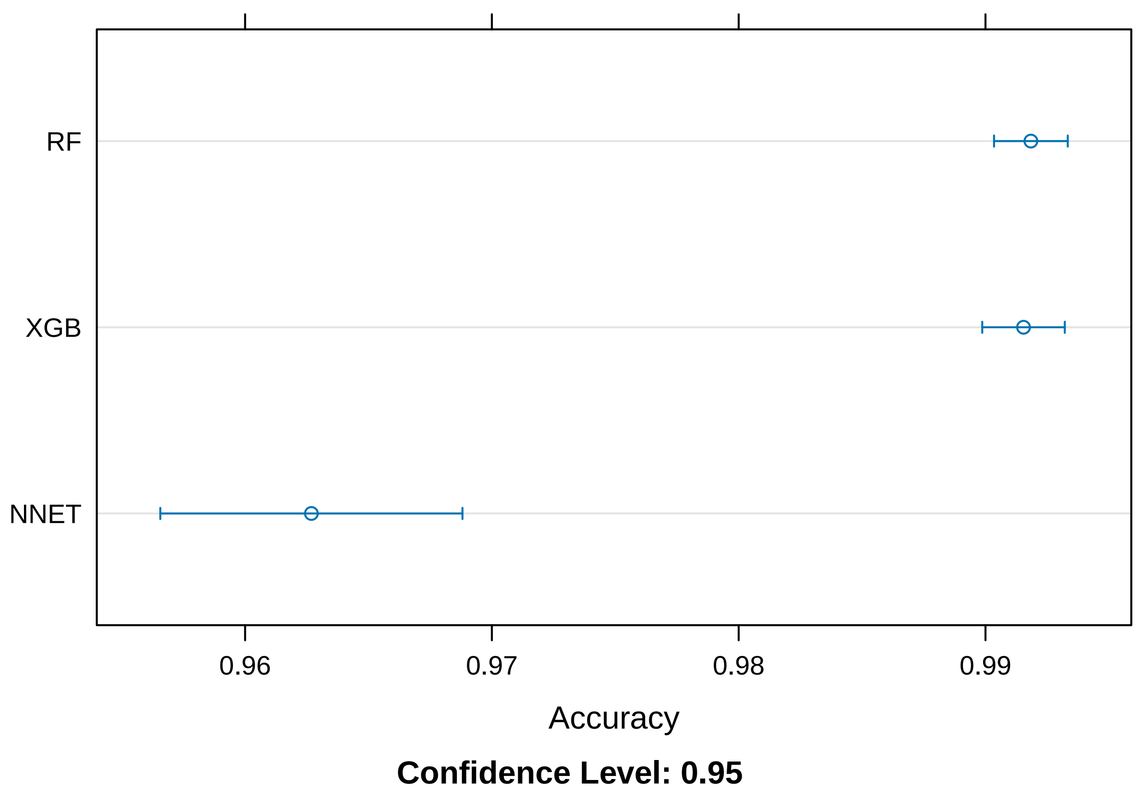
```
set.seed(2023)

fit_nnet <- train(challenge_outcome ~ .,
                     data = final_data_balanced,
                     method = "nnet",
                     metric = my_metric,
                     preProcess = c("center", "scale"),
                     trControl = my_ctrl,
                     trace = FALSE)
```

Let's compare the models. We compile the resampling results together.

```
my_results <- resamples(list(NNET = fit_nnet,
                              RF = fit_rf,
                              XGB = fit_xgb))
```

Compare models based on Accuracy.

```
dotplot(my_results, metric = "Accuracy")
```



Based on the results above, we observe that the Random forest method performs better than the other two methods.