HarvardX: PH125.9x Data Science: Second Capstone Project
Predicting Age Group of Internet application

Edward Ho

15 Oct 2021

Introduction

This is a second project of PH125.9x Data Science: Capstone course. We are encourage to take what we learnt from the course to the next level and solidify our knowledge with real life situation. This project does not provide us with any cleaned data set like the first project. We are free to select any data set publicly available for this project. Kaggle is a very popular Internet site for data scientists and machine learning professionals. In this project, we select one of the data set available in Kaggle to train our algorithm.

Overview

Trell is India's largest lifestyle videos and shopping app where you can discover latest fashion trends, makeup tutorials, fitness routines and etc. This application is very similar to another famous Chinese app named TikTok.

Trell has over 10 million downloads in Google play store. Predicting age group of users will help to identify appropriate content delivery to users.

We are trying to use machine learning method to predict the age group of users of the application.

Data wrangling

The data set is provided in this Kaggle website.

https://www.kaggle.com/adityak80/trell-social-media-usage-data

We will download and identify any missing data in our data set. After an inspection of the training set and test set provided in the Kaggle web site. One major problem in the test set given in the website is that the outcome column "age_group" is missing. We have to use the training set as our major source data. The original training data set will split into two. A training set and a test set.

One common problem amount public data set is that data collection process may be not complete. Some missing value or NA will be introduced into data set. We need to make sure that it is not happening to our data set and make sure it is clean and well define.

```
##########################################################
# Create train and test set (final hold-out test set)
##########################################################

# Note: this process could take a couple of minutes
```

```r
if(!require(tidyverse)) install.packages("tidyverse", repos = "http://cran.us.r-project.org")
if(!require(caret)) install.packages("caret", repos = "http://cran.us.r-project.org")
if(!require(data.table)) install.packages("data.table", repos = "http://cran.us.r-project.org")
if(!require(ggplot2)) install.packages("ggplot2", repos = "http://cran.us.r-project.org")
if(!require(lubridate)) install.packages("lubridate", repos = "http://cran.us.r-project.org")
if(!require(corrr)) install.packages("corrr", repos = "http://cran.us.r-project.org")
if(!require(ggcorrplot)) install.packages("ggcorrplot", repos = "http://cran.us.r-project.org")
if(!require(xgboost)) install.packages("xgboost", repos = "http://cran.us.r-project.org")
if(!require(reshape2)) install.packages("reshape2", repos = "http://cran.us.r-project.org")
if(!require(dplyr)) install.packages("dplyr", repos = "http://cran.us.r-project.org")
if(!require(gridExtra)) install.packages("dplyr", repos = "http://cran.us.r-project.org")
if(!require(randomForest)) install.packages("randomForest", repos = "http://cran.us.r-project.org")
if(!require(e1071)) install.packages("e1071", repos = "http://cran.us.r-project.org")

# Loading
library(tidyverse)
library(caret)
library(data.table)
library(ggplot2)
library(lubridate)
library(corrr)
library(ggcorrplot)
library(xgboost)
library(reshape2)
library(dplyr)
library(gridExtra)
library(randomForest)
library(e1071)

dl <- tempfile()

## Read csv file
download.file("https://dphi.s3.ap-south-1.amazonaws.com/dataset/train_age_dataset.csv", dl)

## Store data into variable
my_data_org <- read.csv(dl, stringsAsFactors=FALSE)
glimpse(my_data_org)

# Remove temp file as placeholder for the zip file
rm(dl)
```

We exam the structure and data type of our data set. Information such as NA, data type, number of columns, row names, features and outcome have to be identify before we move one to visualization and algorithm process.

```r
# Test NA of data
anyNA(my_data_org)
```

```
## [1] FALSE
```

```r
# Display general structure of our data
str(my_data_org)
```

```
## 'data.frame':    488877 obs. of  27 variables:
##  $ Unnamed..0                     : int  265153 405231 57867 272618 251123 229892 18167 18705 498266
##  $ userId                         : int  48958844 51100441 6887426 50742404 45589200 41104551 3282424
##  $ tier                           : int  2 2 2 2 2 2 1 1 2 2 ...
##  $ gender                         : int  1 2 1 1 2 1 1 2 1 1 ...
##  $ following_rate                 : num  0 0 0 0 0 ...
##  $ followers_avg_age              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ following_avg_age              : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ max_repetitive_punc            : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ num_of_hashtags_per_action     : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ emoji_count_per_action         : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ punctuations_per_action        : num  0 0.0769 0 0 0 ...
##  $ number_of_words_per_action     : num  0 0.154 0 0 0 ...
##  $ avgCompletion                  : num  0.46333 0.42947 0.34166 0.00574 0.45655 ...
##  $ avgTimeSpent                   : num  34.2 15.3 22 3 12.3 ...
##  $ avgDuration                    : num  54 96.2 83.1 523.1 53.8 ...
##  $ avgComments                    : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ creations                      : num  0 0.00847 0 0 0 ...
##  $ content_views                  : num  0.2 0.09322 0.00279 0.0084 0.20492 ...
##  $ num_of_comments                : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ weekends_trails_watched_per_day: num  0.0417 0.0127 0 0 0 ...
##  $ weekdays_trails_watched_per_day: num  0.025 0.018644 0.000557 0.001681 0.04918 ...
##  $ slot1_trails_watched_per_day   : num  0 0 0 0 0 ...
##  $ slot2_trails_watched_per_day   : num  0 0.08475 0.00279 0 0.0082 ...
##  $ slot3_trails_watched_per_day   : num  0.175 0 0 0 0.0574 ...
##  $ slot4_trails_watched_per_day   : num  0.0333 0.0339 0 0.0084 0.1803 ...
##  $ avgt2                          : num  0 82.5 0 0 0 ...
##  $ age_group                      : int  1 2 1 1 1 1 1 3 4 1 ...
```

```r
# Show basic stat of our data
summary(my_data_org)
```

```
##    Unnamed..0         userId              tier           gender     
##  Min.   :     0   Min.   :      27   Min.   :1.000   Min.   :1.000  
##  1st Qu.:135779   1st Qu.:35375992   1st Qu.:2.000   1st Qu.:1.000  
##  Median :271560   Median :43362702   Median :2.000   Median :1.000  
##  Mean   :271606   Mean   :42360955   Mean   :1.975   Mean   :1.213  
##  3rd Qu.:407431   3rd Qu.:53705229   3rd Qu.:2.000   3rd Qu.:1.000  
##  Max.   :543196   Max.   :79042026   Max.   :3.000   Max.   :2.000  
##  following_rate     followers_avg_age following_avg_age max_repetitive_punc
##  Min.   :  0.0000   Min.   :0.0000    Min.   :0.000     Min.   :  0.0000   
##  1st Qu.:  0.0000   1st Qu.:0.0000    1st Qu.:0.000     1st Qu.:  0.0000   
##  Median :  0.0000   Median :0.0000    Median :0.000     Median :  0.0000   
##  Mean   :  0.0822   Mean   :0.3475    Mean   :0.403     Mean   :  0.7397   
##  3rd Qu.:  0.0086   3rd Qu.:0.0000    3rd Qu.:0.000     3rd Qu.:  0.0000   
##  Max.   :895.3040   Max.   :4.0000    Max.   :4.000     Max.   :624.0000   
##  num_of_hashtags_per_action emoji_count_per_action punctuations_per_action
##  Min.   :0.0000000          Min.   :0.0000000      Min.   : 0.00000       
##  1st Qu.:0.0000000          1st Qu.:0.0000000      1st Qu.: 0.00000       
##  Median :0.0000000          Median :0.0000000      Median : 0.00000       
##  Mean   :0.0002772          Mean   :0.0009814      Mean   : 0.01281       
##  3rd Qu.:0.0000000          3rd Qu.:0.0000000      3rd Qu.: 0.00000       
##  Max.   :2.3333333          Max.   :3.0000000      Max.   :27.33333       
##  number_of_words_per_action avgCompletion        avgTimeSpent
```

```
##  Min.   :  0.0000          Min.   :0.0006531   Min.   :       1
##  1st Qu.:  0.0000          1st Qu.:0.1996755   1st Qu.:       6
##  Median :  0.0000          Median :0.3297429   Median :       8
##  Mean   :  0.1792          Mean   :0.3415820   Mean   :     109
##  3rd Qu.:  0.1502          3rd Qu.:0.4604917   3rd Qu.:      13
##  Max.   :262.6667          Max.   :1.0000000   Max.   :38266041
##   avgDuration        avgComments         creations        content_views
##  Min.   :   0.233   Min.   :   0.000   Min.   : 0.00000   Min.   : 0.00089
##  1st Qu.:  30.724   1st Qu.:   0.000   1st Qu.: 0.00000   1st Qu.: 0.04065
##  Median :  62.501   Median :   0.000   Median : 0.00000   Median : 0.12403
##  Mean   :  83.105   Mean   :   0.321   Mean   : 0.01707   Mean   : 0.39101
##  3rd Qu.: 112.246   3rd Qu.:   0.000   3rd Qu.: 0.00909   3rd Qu.: 0.36449
##  Max.   :7541.026   Max.   :3228.000   Max.   :63.38889   Max.   :75.66228
##  num_of_comments    weekends_trails_watched_per_day
##  Min.   :0.000000   Min.   : 0.000000
##  1st Qu.:0.000000   1st Qu.: 0.000000
##  Median :0.000000   Median : 0.003968
##  Mean   :0.002009   Mean   : 0.074353
##  3rd Qu.:0.000000   3rd Qu.: 0.060000
##  Max.   :8.196850   Max.   :17.201754
##  weekdays_trails_watched_per_day slot1_trails_watched_per_day
##  Min.   : 0.000000               Min.   : 0.00000
##  1st Qu.: 0.002265               1st Qu.: 0.00000
##  Median : 0.015873               Median : 0.00000
##  Mean   : 0.066927               Mean   : 0.03286
##  3rd Qu.: 0.059016               3rd Qu.: 0.00000
##  Max.   :18.756140               Max.   :19.61290
##  slot2_trails_watched_per_day slot3_trails_watched_per_day
##  Min.   : 0.00000             Min.   : 0.00000
##  1st Qu.: 0.00000             1st Qu.: 0.00000
##  Median : 0.01183             Median : 0.01456
##  Mean   : 0.14170             Mean   : 0.15040
##  3rd Qu.: 0.11719             3rd Qu.: 0.11864
##  Max.   :27.90598             Max.   :45.08333
##  slot4_trails_watched_per_day     avgt2            age_group
##  Min.   : 0.00000             Min.   :     0.0   Min.   :1.000
##  1st Qu.: 0.00000             1st Qu.:     0.0   1st Qu.:1.000
##  Median : 0.01587             Median :     0.0   Median :1.000
##  Mean   : 0.15838             Mean   :   164.8   Mean   :1.742
##  3rd Qu.: 0.12240             3rd Qu.:   178.7   3rd Qu.:2.000
##  Max.   :55.15385             Max.   :39304.0   Max.   :4.000
```

```r
# Ensure no duplicated userId
n_occur <- data.frame(table(my_data_org$userId))
n_occur[n_occur$Freq > 1,]
```

```
## [1] Var1 Freq
## <0 rows> (or 0-length row.names)
```

There are no duplicated record in our data set and it is free from NA. We can move on to the next step.

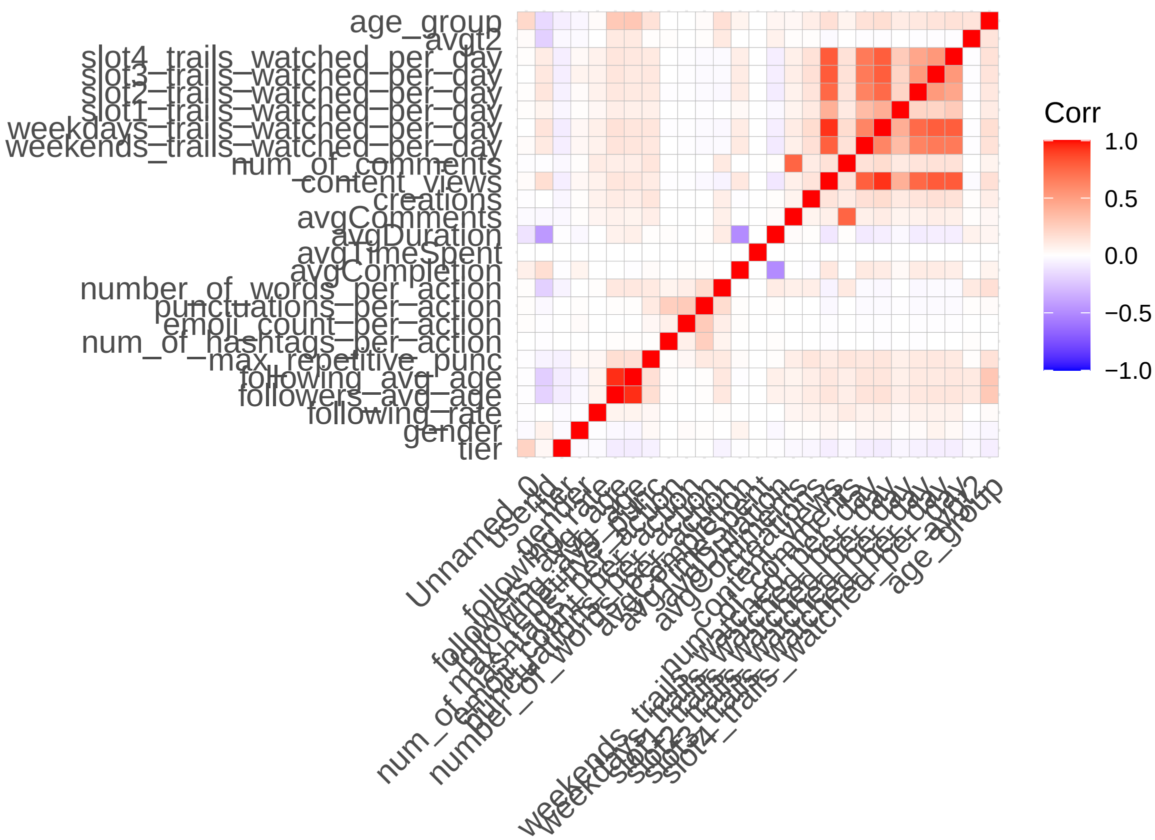
Data visualization

In the summary of our data set, we see that features in our data set are either Integer or Numeric. The variance of some features is very large. The features "tier" and "gender" should convert to categorical type, and the outcome should also be categorical, too.

Some of our features have a very similar named. Features such as "slot1_trails_watched_per_day", "slot2_trails_watched_per_day". Features as such may be highly correlated to each other.

The correlation graph shows their correlation.

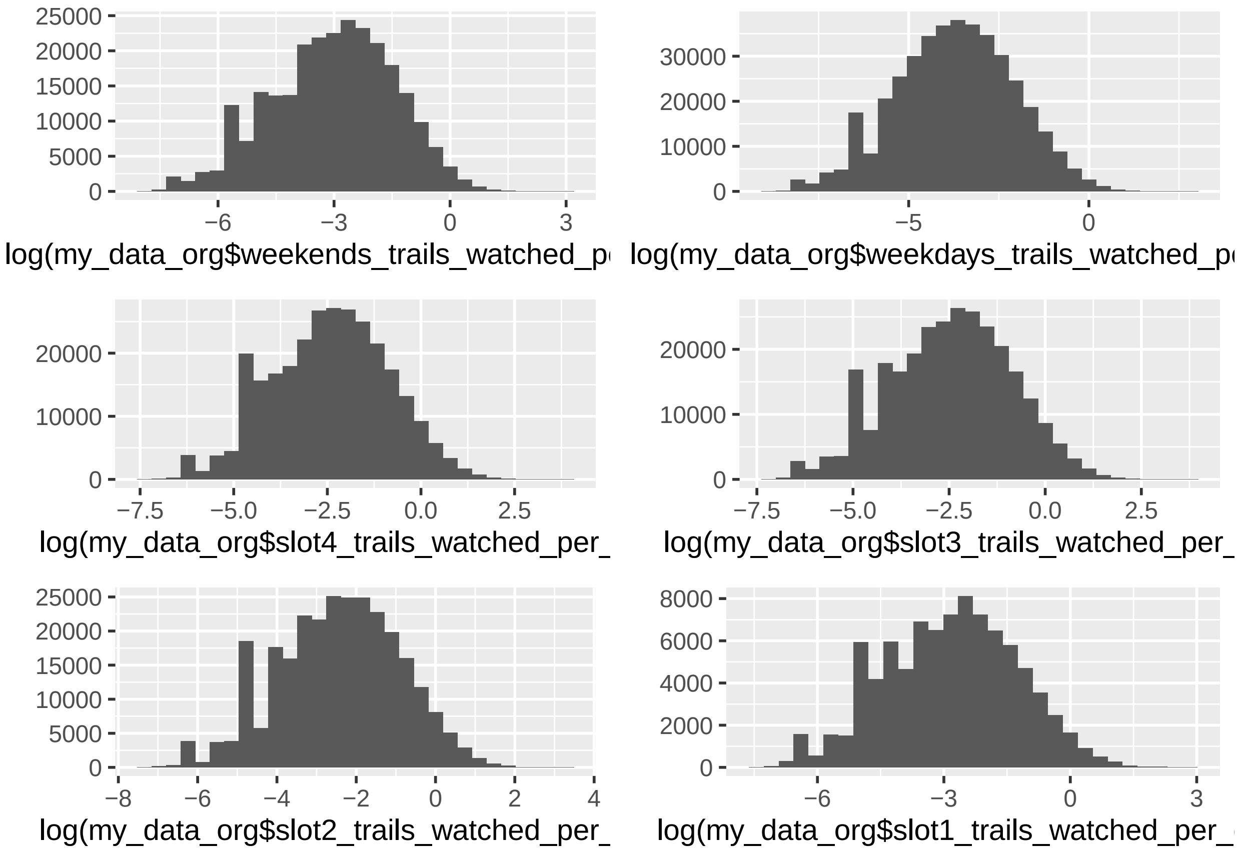
```
mydata.cor = round(cor(my_data_org), 3)

# Remove Username and user ID from the data set.  They are not useful in our graph.
remove_cols <- c('Unnamed..0','userId')
trimmed_my_data.cor <- mydata.cor[, !(colnames(my_data_org)%in%remove_cols), drop=FALSE]
ggcorrplot(trimmed_my_data.cor )
```



We pick those columns that are highly correlated and plot their histograms. The variances of features tell us that the small value in graphs will be hard to detect. So we use logarithms to help us to express large numbers.

```
p1 <- qplot(log(my_data_org$weekends_trails_watched_per_day), bins = 30)
p2 <- qplot(log(my_data_org$weekdays_trails_watched_per_day), bins = 30)
p3 <- qplot(log(my_data_org$slot4_trails_watched_per_day), bins = 30)
p4 <- qplot(log(my_data_org$slot3_trails_watched_per_day), bins = 30)
p5 <- qplot(log(my_data_org$slot2_trails_watched_per_day), bins = 30)
```

```
p6 <- qplot(log(my_data_org$slot1_trails_watched_per_day), bins = 30)

grid.arrange(p1, p2, p3, p4, p5, p6, ncol=2)
```



Data Modeling

The outcome of our data set is age_group. It contains 1, 2, 3 and 4. Each of this value represents an age group of the mobile app user. Without doubt, this is a classification problem. The original data type of age_group provided in the data set is an Integer. We convert it to factor in the above section. We will try to use Random Forest, KNN and XGBoost algorithm on this data set.

Random Forest model

The training set contains over 400k entries. RF algorithm requires powerful CPU to generate results within reasonable time. Random Forest model also provides different arguments for us to fine tune the execution in order to search for best result. This requires powerful CPU to run with a size of training set like this. A sample of 10,000 entries from the training set will be used to run the algorithm.

```
# Test with smaller set 10,000 obs
set.seed(1, sample.kind="Rounding")

index <- sample(1:nrow(my_data_org), 10000, replace = FALSE)
my_data_shortlist <- my_data_org[index,]

# Remove userId and username
remove_cols <- c('Unnamed..0','userId')

# define column number of outcome
n_col <- 25

# make a copy of our sample data
my_data <- my_data_shortlist[, !(colnames(my_data_shortlist)%in%remove_cols), drop=FALSE]

# verify data set
# head(my_data)

# Change target variables and outcome into factor
my_data$tier <- as.factor(my_data$tier)
my_data$gender <- as.factor(my_data$gender)
my_data$age_group <- as.factor(my_data$age_group)

# Define test set and training set
test_index <- createDataPartition(y = my_data$age_group, times = 1, p = 0.1, list = FALSE)
train_set <- my_data[-test_index, ]
test_set <- my_data[test_index, ]

# Define predictors and response variables in the training set
train_x <- train_set[, -n_col]
train_y <- train_set[,n_col]

# Define predictors and response variables in the test set
test_x <- test_set[, -n_col]
test_y <- test_set[, n_col]
```

Random forest model allows us to fine tune the model for best result. We will try to find the best mtry value. mtry represents number of variables randomly sampled as candidates at each split.

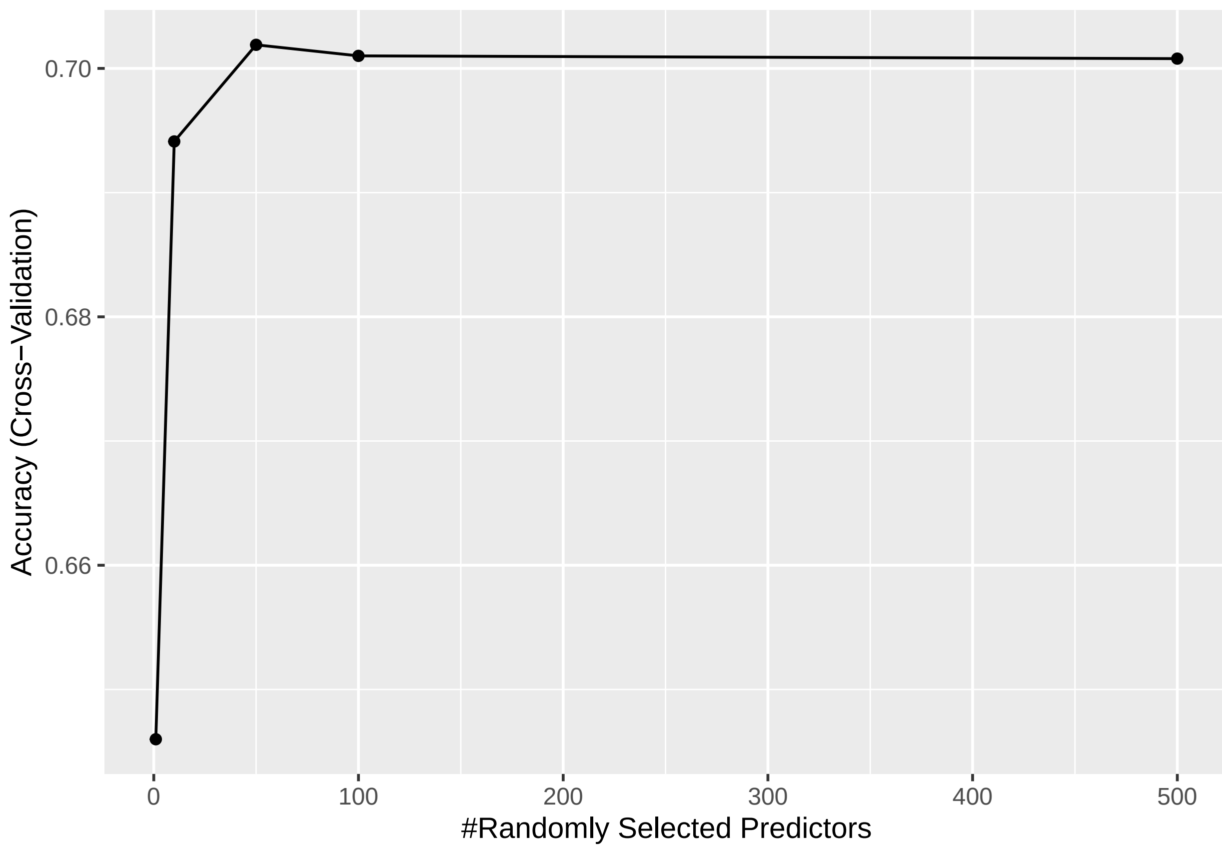
```
control <- trainControl(method="cv", number=5, search ="grid")
grid <- data.frame(mtry=c(1,10,50,100,500))

# Run the model
train_rf <- train(age_group ~ .,
                  data = train_set,
                  method="rf",
                  trControl=control,
                  tuneGrid = grid,
                  metric = "Accuracy",
                  importance = TRUE)
```

```
## Warning in (function (kind = NULL, normal.kind = NULL, sample.kind = NULL) :
## non-uniform 'Rounding' sampler used
```

```
# Print the results
print(train_rf)
```

```
## Random Forest
##
## 8997 samples
##   24 predictor
##    4 classes: '1', '2', '3', '4'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 7197, 7197, 7198, 7198, 7198
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##     1   0.6459929  0.1324726
##    10   0.6941191  0.4597086
##    50   0.7018989  0.4750521
##   100   0.7010095  0.4723499
##   500   0.7007877  0.4719727
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 50.
```

```
ggplot(train_rf)
```




```r
best_mtry <- train_rf$bestTune
print(best_mtry)
```

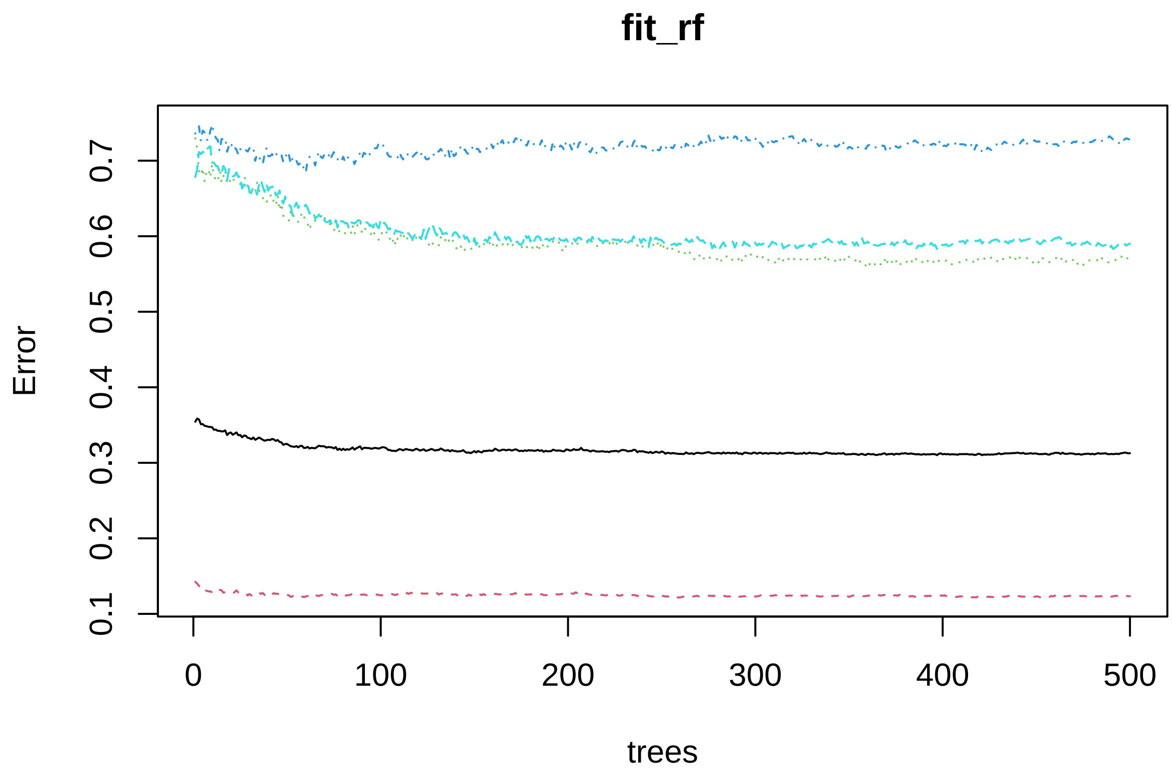
```
##   mtry
## 3   50
```

The best value of mtry is 50. There are other parameters we can fine tune in RF model such as nTree, nodesize and maxnodes.

The nodesize controls the size of terminals nodes during node splitting while training a tree. Nodes with fewer than nodesize objects are not split, and therefore become terminal nodes.

The maxnodes controls the maximum number of terminal nodes trees in the forest can have. If not given, trees are grown to the maximum possible (subject to limits by nodesize).

```r
fit_rf <- randomForest(train_x, train_y,
                       minNode = train_rf$bestTune$mtry)
plot(fit_rf)
```




```
y_hat_rf <- predict(fit_rf, test_set)
cm_rf <- confusionMatrix(y_hat_rf, test_y)
print(cm_rf$overall["Accuracy"])
```

```
##  Accuracy
## 0.6849452
```

KNN model

KNN model is another computing power hunger model. It will take extensive amount of time to execute with a size like 400k entries and over 20 features.

```
# define column number of outcome
n_col <- 25

# make a copy of our sample data
my_data <- my_data_shortlist[, !(colnames(my_data_shortlist)%in%remove_cols), drop=FALSE]

# The normalization function is created
nor <-function(x) { (x -min(x))/(max(x)-min(x))   }

# Normalized features for KNN
select_features <- seq(1:(n_col-1))
my_data_norm <- as.data.frame(lapply(my_data[, select_features], nor))
```

```r
# Generated new normalized data set.  Outcome is appended back.
my_data_norm <- data.frame(my_data_norm, my_data$age_group)
col_names <- colnames(my_data_norm[,select_features])
colnames(my_data_norm) <- c(col_names, "age_group")

my_data_norm$age_group <- as.factor(my_data_norm$age_group)
str(my_data_norm)
```

```
## 'data.frame':    10000 obs. of  25 variables:
##  $ tier                            : num  0.5 0 0 0.5 0 0.5 0.5 0.5 0.5 0.5 ...
##  $ gender                          : num  1 1 0 0 0 0 0 0 0 0 ...
##  $ following_rate                  : num  0.00 0.00 1.10e-03 0.00 7.06e-05 ...
##  $ followers_avg_age               : num  0 0 0.57 0 0 ...
##  $ following_avg_age               : num  0 0 0.692 0 0 ...
##  $ max_repetitive_punc             : num  0.0132 0 0.0132 0 0 ...
##  $ num_of_hashtags_per_action      : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ emoji_count_per_action          : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ punctuations_per_action         : num  0 0 0.00605 0 0 ...
##  $ number_of_words_per_action      : num  0.10168 0.01816 0.20616 0.03631 0.00807 ...
##  $ avgCompletion                   : num  0.589 0.28 0.429 0.808 0.426 ...
##  $ avgTimeSpent                    : num  0.002089 0.000587 0.003555 0.001076 0.091395 ...
##  $ avgDuration                     : num  0.00808 0.00394 0.03997 0.00196 0.01009 ...
##  $ avgComments                     : num  0 0 0.0469 0 0 ...
##  $ creations                       : num  0.00135 0.00267 0.01021 0.00135 0.00534 ...
##  $ content_views                   : num  0.003396 0.007223 0.004083 0.000805 0.009364 ...
##  $ num_of_comments                 : num  0 0 0.0208 0 0 ...
##  $ weekends_trails_watched_per_day: num  0.00661 0.00437 0.00198 0.00176 0 ...
##  $ weekdays_trails_watched_per_day: num  0 0.00234 0.00163 0 0.00684 ...
##  $ slot1_trails_watched_per_day   : num  0 0 0.00117 0 0.02138 ...
##  $ slot2_trails_watched_per_day   : num  0 0.00228 0.0012 0 0.00512 ...
##  $ slot3_trails_watched_per_day   : num  0 0.000501 0.000469 0.000505 0 ...
##  $ slot4_trails_watched_per_day   : num  0.00928 0.00798 0.00392 0.00124 0 ...
##  $ avgt2                           : num  0.000113 0 0.008926 0.004223 0.004863 ...
##  $ age_group                       : Factor w/ 4 levels "1","2","3","4": 3 3 4 1 1 1 1 3 1 1 ...
```

```r
# Create normalized test set and training set using test_index generated in the
# above to ensure consistency of our training and test data set
set.seed(1, sample.kind="Rounding")
```

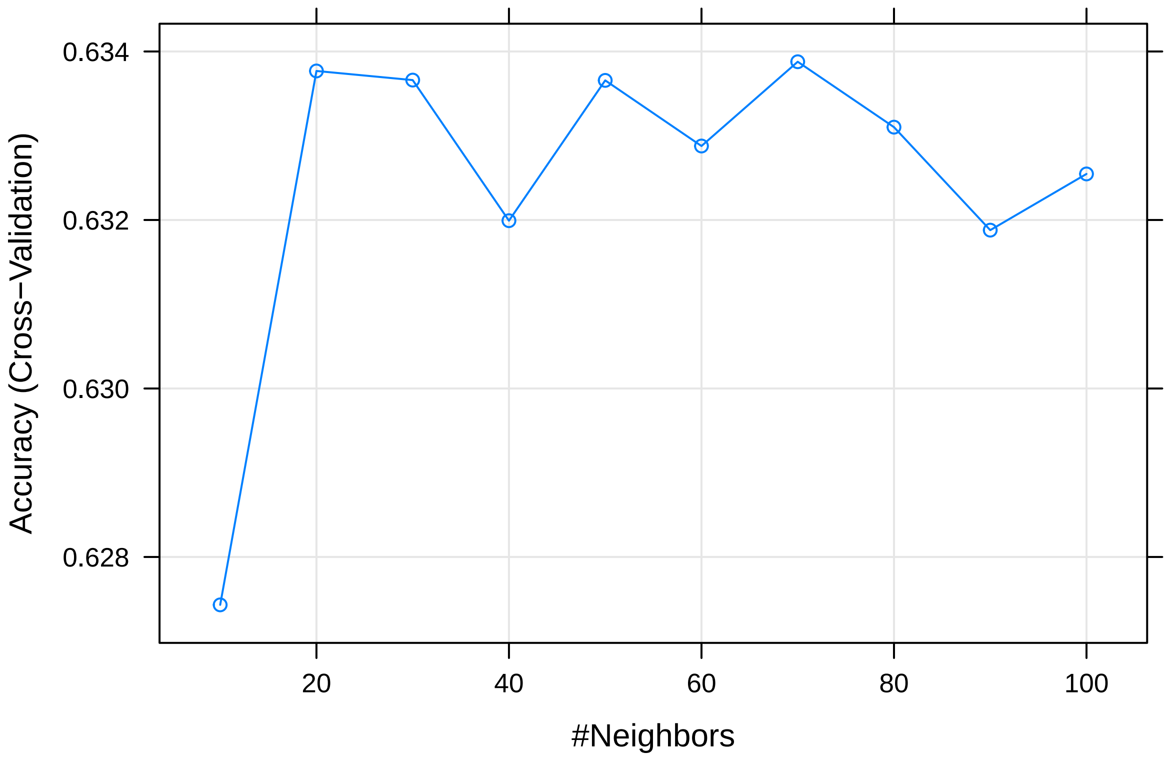
```
## Warning in set.seed(1, sample.kind = "Rounding"): non-uniform 'Rounding' sampler
## used
```

```r
train_set <- my_data_norm[-test_index, ]
test_set <- my_data_norm[test_index, ]

# Define predictors and response variables in the training set
train_x <- train_set[, -n_col]
train_y <- train_set[,n_col]

# Define predictors and response variables in the test set
test_x <- test_set[, -n_col]
test_y <- test_set[, n_col]
```

```r
tune_grid <- data.frame(k = seq(10, 100, 10))
control <- trainControl(method = "cv", number = 10, p = .9)

train_knn <- train(age_group ~ ., data = train_set,
                   method = "knn",
                   tuneGrid = tune_grid,
                   trControl = control)
```

```
## Warning in (function (kind = NULL, normal.kind = NULL, sample.kind = NULL) :
## non-uniform 'Rounding' sampler used
```

```r
plot(train_knn)
```




```r
best_tune_knn <- train_knn$bestTune

y_hat_knn <- predict(train_knn, test_set, type = "raw")
cm_knn <- confusionMatrix(y_hat_knn, test_y)$overall["Accuracy"]
print(cm_knn)
```

```
##  Accuracy
## 0.6370887
```

XGBoost model

XBBoost is short for eXtreme Gradient Boosting package. It is a very popular model in Kaggle and data science professional because it is fast, and it can utilized modern GPU power to speed up its execution.

```r
# Assign train and test data set, with the same test index for consistency
train_set <- my_data[-test_index, ]
test_set <- my_data[test_index, ]

# Define predictors and response variables in the training set
train_x <- data.matrix(train_set[, -n_col])
train_y <- train_set[,n_col]

# Define predictors and response variables in the test set
test_x <- data.matrix(test_set[, -n_col])
test_y <- test_set[, n_col]

# Define final training and testing sets for XGboost
set.seed(1, sample.kind="Rounding")
```

```
## Warning in set.seed(1, sample.kind = "Rounding"): non-uniform 'Rounding' sampler
## used
```

```r
xgb_train <- xgb.DMatrix(data = train_x, label = train_y)
xgb_test <- xgb.DMatrix(data = test_x, label = test_y)

#  Define watchlist
watchlist <- list(train=xgb_train, test=xgb_test)

# Tune number of rounds
N_rounds <- seq(10, 5000, 250)
preds6 <- sapply(N_rounds, function(n){
  first_model <- xgb.train(data = xgb_train, max.depth = 6, watchlist=watchlist, nrounds = n, verbose =
  pred_y <- predict(first_model, as.matrix(as.integer(test_y)))
  pred_y_hat <- round(pred_y)

  u <- union(pred_y_hat, test_y)
  t <- table(factor(pred_y_hat, u), factor(test_y, u))
  cm <- confusionMatrix(t)
  cm_accuracy <- cm$overall["Accuracy"]
  result <- c(6, n, cm_accuracy)
})
preds6 <- t(preds6)
colnames(preds6) <- c('depths', 'rounds', 'Accuarcy')
preds6
```

```
##       depths rounds  Accuarcy
##  [1,]      6     10 0.6251246
##  [2,]      6    260 0.6251246
##  [3,]      6    510 0.6251246
##  [4,]      6    760 0.6251246
##  [5,]      6   1010 0.6251246
##  [6,]      6   1260 0.6251246
```

```
##  [7,]      6   1510 0.6251246
##  [8,]      6   1760 0.6251246
##  [9,]      6   2010 0.6251246
## [10,]      6   2260 0.6251246
## [11,]      6   2510 0.6251246
## [12,]      6   2760 0.6251246
## [13,]      6   3010 0.6251246
## [14,]      6   3260 0.6251246
## [15,]      6   3510 0.6251246
## [16,]      6   3760 0.6251246
## [17,]      6   4010 0.6251246
## [18,]      6   4260 0.6251246
## [19,]      6   4510 0.6251246
## [20,]      6   4760 0.6251246
```

Conclusion

All three models provided accuracy approximately 0.6 and 0.7. XGboost is the fastest among these three algorithms. All these three algorithms allow us to fine tune the model with various parameters. We can refine our results with their tuned parameters to achieve a higher accuracy than current level of 0.6 and 0.7.

```
##           model  accuracy
## 1 Random Forest 0.6849452
## 2           KNN 0.6370887
## 3       XGBoost 0.6251246
```