Mobile Strategy Game from Apple App Store

Prathit Shivade, Daehun Kim, Shenzeng Feng

Objective

The task in hand is to develop machine learning models based on the Kaggle data set in order to conduct predictive analysis.
The data set used for analysis is Mobile Strategy Games metadata from Apple App store: <https:// (https://) https://www.kaggle.com/tristan581/17k-apple-app-store-strategy-games (https://www.kaggle.com/tristan581/17k-apple-app-store-strategy-games)

Contribution

1. Predicting the Average User Rating - Daehun Kim
2. Predicting the Successful Apps - Prathit Shivade
3. Predicting the Average number of Daily Ratings - Shenzeng Feng

Please, set the directory file that you want and the data set is also attached.

```
setwd("C:/dataset")
```

Predicting the Average User Rating

Introduction

It was designed for predicting `Average User Rating` of mobile strategy games from App Store. It somehow means the popularity of game, so I would say that I also predicted the popularity. I have tried to use all the data as factor type by one-hot ending since the number of variables is too large to execute the model. I utilized various statistical values and used Random Forest and Gradient Boosting models.

Library and Read Data

```
library(caret)
library(skimr)
library(VIM)
library(ggthemes)
library(corrplot)
library(dplyr)
library(ggplot2)
library(e1071)
library(randomForest)
library(tidyverse)
library(gbm)

mobilegame <- read.csv("appstore_games.csv", header=T, stringsAsFactors = FALSE)
```

Preprocessing

Let's start with checking the data set and keep moving on to transform columns.

Data Set There are 18 columns and 17007 rows and two types of data type: character and numeric. We can see that there are also some useless columns for modeling such as URL,ID, and so on. First of all, we will investigate missing values.

```
skim(mobilegame)
```

Data summary

Name	mobilegame
Number of rows	17007
Number of columns	18

Column type frequency:	
character	13
numeric	5

Group variables	None

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
URL	0	1	44	256	0	16847	0

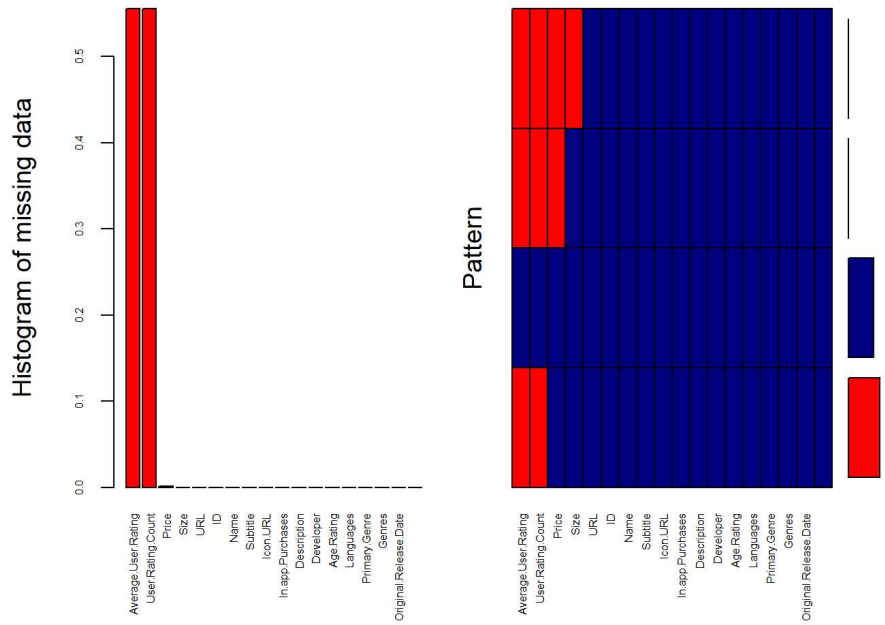
skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
Name	0	1	1	251	0	16847	0
Subtitle	0	1	0	153	11746	5011	0
Icon.URL	0	1	117	127	0	16847	0
In.app.Purchases	0	1	0	107	9324	3804	0
Description	0	1	6	12925	0	16473	0
Developer	0	1	2	86	0	8693	0
Age.Rating	0	1	2	3	0	4	0
Languages	0	1	0	442	60	991	0
Primary.Genre	0	1	4	17	0	21	0
Genres	0	1	15	77	0	1004	0
Original.Release.Date	0	1	9	10	0	3084	0
Current.Version.Release.Date	0	1	9	10	0	2512	0

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100
ID	0	1.00	1.059614e+09	299967589.36	284921427	899654330.0	1112286228.0	1286982837.0	1.475077e+09
Average.User.Rating	9446	0.44	4.060000e+00	0.75	1	3.5	4.5	4.5	5.000000e+00
User.Rating.Count	9446	0.44	3.306530e+03	42322.56	5	12.0	46.0	309.0	3.032734e+06
Price	24	1.00	8.100000e-01	7.84	0	0.0	0.0	0.0	1.799900e+02
Size	1	1.00	1.157064e+08	203647677.85	51328	22950144.0	56768954.0	133027072.0	4.005591e+09

Imputation Let's have a look at missing values through plot. we can see that most of missing values are in `AVerage User Rating` and `User Rating Count`. Additionally, the small amount of missing values are in `Price` and `Size`.

```
aggr_plot <- aggr(mobilegame,
                  col = c('navyblue','red'),
                  numbers = TRUE,
                  sortVars = TRUE,
                  labels = names(mobilegame),
                  cex.axis = .5,
                  gap = 3,
                  ylab = c("Histogram of missing data", "Pattern")
                  )
```



```
##
##  Variables sorted by number of missings:
##                       Variable        Count
##           Average.User.Rating 5.554184e-01
##             User.Rating.Count 5.554184e-01
##                         Price 1.411184e-03
##                          Size 5.879932e-05
##                           URL 0.000000e+00
##                            ID 0.000000e+00
##                          Name 0.000000e+00
##                      Subtitle 0.000000e+00
##                      Icon.URL 0.000000e+00
##              In.app.Purchases 0.000000e+00
##                   Description 0.000000e+00
##                     Developer 0.000000e+00
##                    Age.Rating 0.000000e+00
##                     Languages 0.000000e+00
##                 Primary.Genre 0.000000e+00
##                        Genres 0.000000e+00
##         Original.Release.Date 0.000000e+00
##  Current.Version.Release.Date 0.000000e+00
```

Based on the description from Kaggle, `Average User Rating` and `User Rating Count` columns are NA if it is lower than 5. So I will substitute NA with 0 and delete `Price` and `Size` rows which have NA since it is no information about missing values. Thus, the data set does not have any NAs now.

```
mobilegame$User.Rating.Count[is.na(mobilegame$User.Rating.Count)] <- 0
mobilegame$Average.User.Rating[is.na(mobilegame$Average.User.Rating)] <- 0
mobilegame <- na.omit(mobilegame)
sum(is.na(mobilegame))
```

```
## [1] 0
```

```
mobilegame_original <- mobilegame
```

Date The `Original Date Release` and `Current Version Release Date` columns are converted into a numeric type to be applied to Machine Learning model (The date format is changed depeding on a laptop so you might want to use "%d-%m-%y" format).

```
mobilegame$Original.Release.Date <- as.numeric(as.Date(mobilegame$Original.Release.Date, "%d/%m/%Y"))
mobilegame$Current.Version.Release.Date <- as.numeric(as.Date(mobilegame$Current.Version.Release.Date, "%d/%m/%Y"))
```

Size The Size column has too large values to be classified so that it had better being transformed into log 10 format.

```
mobilegame$Size <- log10(mobilegame$Size)
```

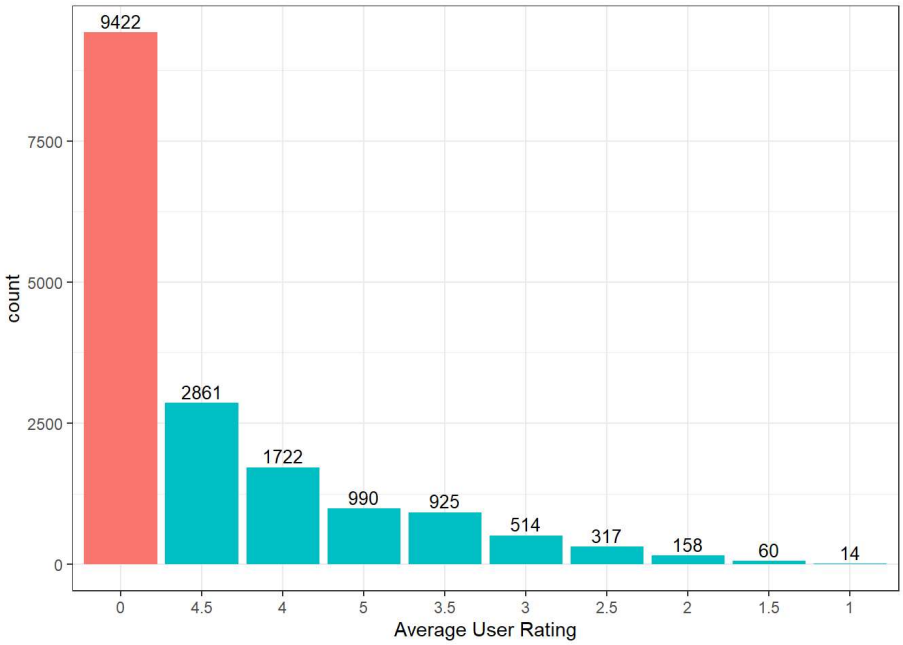
One-Hot Encoding As for the columns that have many categorical values, I will execute One-hot encoding to split them into each column that have either 1 or 0. 1 means that the game has that applicable attribute, on the other hand, 0 is vice versa. Here, the codes regarding one-hot encoding in `Gernes` are explained as below. (The other columns are similarly encoded and hided in the report (it is automatically run): `In App Purchases` / `Languages` / `Primary Genre` / `Age Rating` / `Price` / `Subtitle`)

```
mobilegame$Genres <- gsub(" & ", ".",mobilegame$Genres)
mobilegame$Genres <- gsub(" ", "",mobilegame$Genres)

df_unlisted_gerne <-data.frame(words = unlist(strsplit(mobilegame$Genres, split= ",")))
unique_genre <- unique(df_unlisted_gerne)
df_onehot_gerne <- data.frame(matrix(ncol = nrow(unique(df_unlisted_gerne)), nrow = nrow(mobilegame)))
colnames(df_onehot_gerne) <- unique_genre$words
df_onehot_gerne[is.na(df_onehot_gerne)] <- 0


df_onehot_name <- plyr::ldply(strsplit(mobilegame$Genres, split= ","), rbind)
df_onehot_name[,seq(1:length(df_onehot_name))] <- lapply(df_onehot_name[,seq(1:length(df_onehot_name))], as.character)
df_onehot_name[is.na(df_onehot_name)] <- "-"

#one-hot encoding data frame
for(i in 1:nrow(df_onehot_gerne)){
  for(j in 1:length(df_onehot_gerne)){
      for(k in 1:length(df_onehot_name)){
        if(colnames(df_onehot_gerne)[j] == df_onehot_name[i,k]){
          df_onehot_gerne[i,j] <- 1
        }
      }
  }
}

for(i in 1:length(df_onehot_gerne)){
  colnames(df_onehot_gerne)[i] <- paste0("Genre.",colnames(df_onehot_gerne)[i])
}
```

Exploratory Analysis

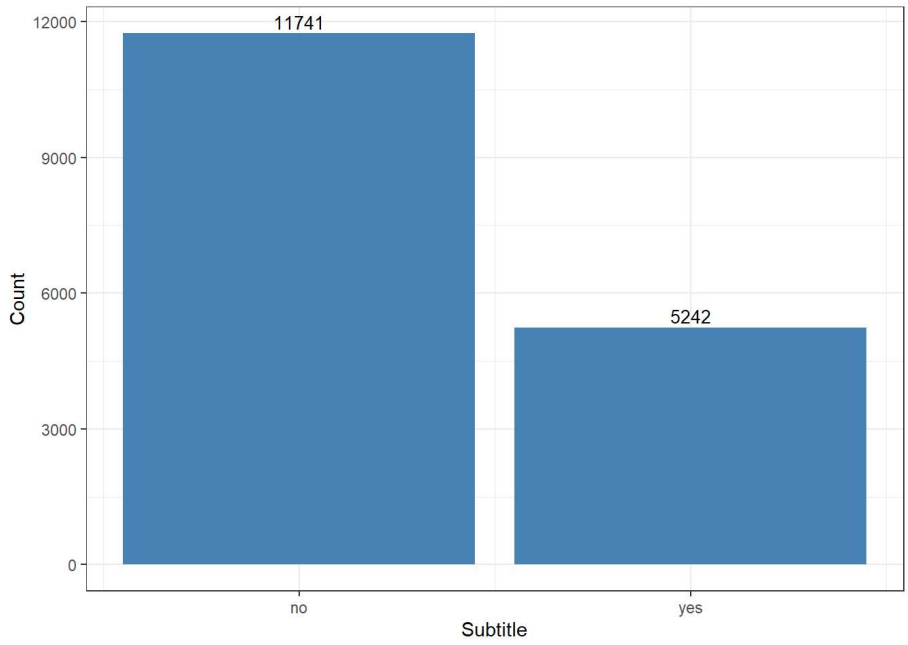
The Distribution of Average Rating Looking at the below graph, 0 rating is dominant so we need to separate data into several types for prediction, enabling the data have various rating values.

```
mobilegame %>%
  count(Average.User.Rating) %>%
  ggplot(aes(x = reorder(Average.User.Rating,-n), y =n)) +
  geom_bar(aes(fill=n<5000), stat = "identity") +
  labs(x = "Average User Rating", y = "count") +
  theme_bw() +
  theme(legend.position="none") +
  geom_text(aes(label=n), vjust=-0.3, color="black", size=3.5)
```



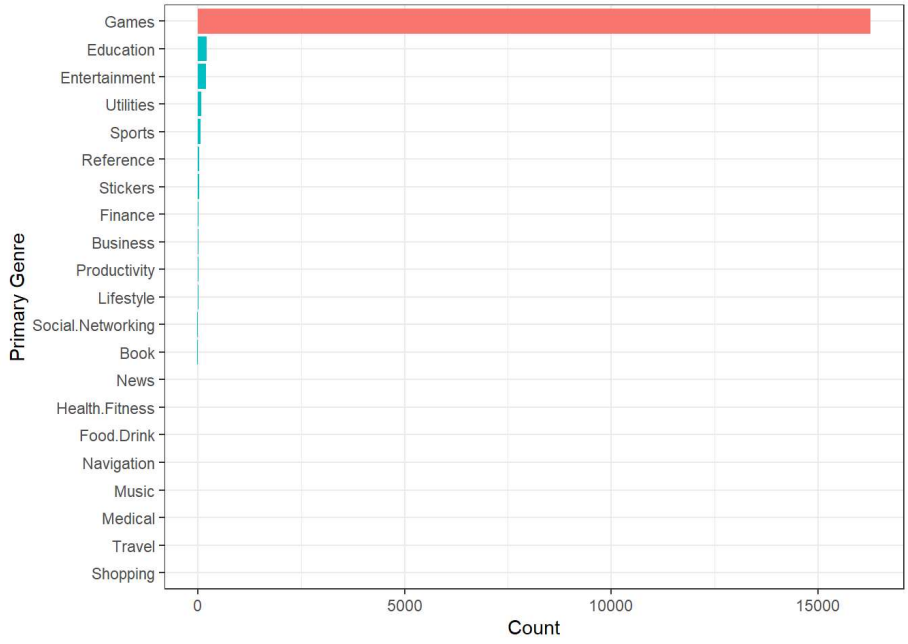
The number of games that has subtitle or not Around 33% of games have subtitle. Hence, it can be one of useful features to predict.

```
df_subtitle %>%
  count(subtitle.o) %>%
  ggplot(aes(x=subtitle.o, y = n)) +
  geom_bar(stat='identity', fill = "steelblue") +
  scale_x_continuous(breaks = c(0,1), labels = c("no","yes")) +
  geom_text(aes(label=n), vjust=-0.3, color="black", size=3.5) +
  labs(x = "Subtitle", y = "Count") +
  theme_bw()
```



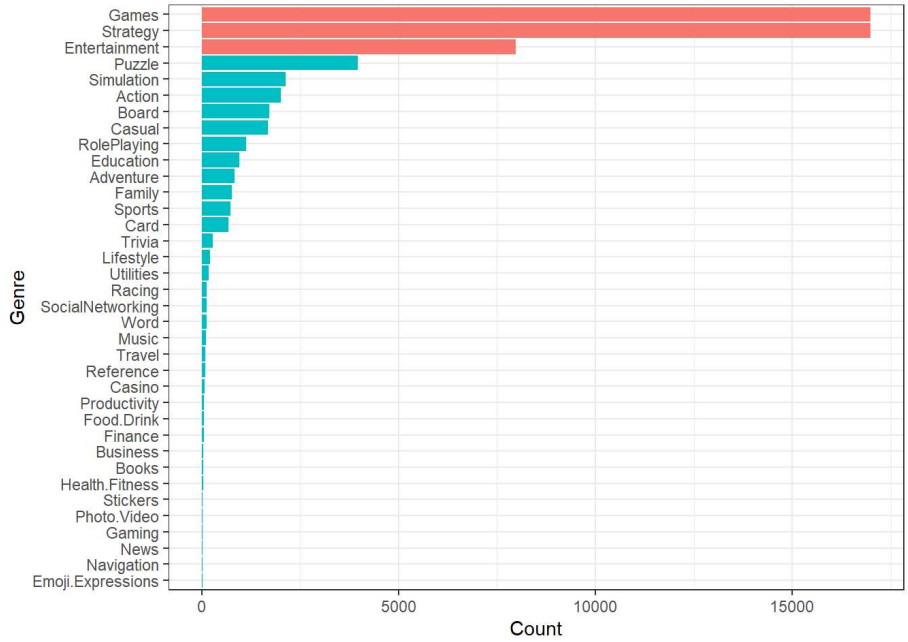
The bar graph of Primary Gerne Notably, Games section account for almost all of games.

```
df_onehot_primaryname %>%
  count(Primary.Genre) %>%
  ggplot(aes(x = reorder(Primary.Genre,n), y =n)) +
  coord_flip() +
  geom_bar(aes(fill=n<5000), stat = "identity") +
  labs(x = "Primary Genre", y = "Count") +
  theme_bw() +
  theme(legend.position="none")
```



The bar graph of Genre Games, Strategy, and Entertainment account for the majority of genres.

```
df_unlisted_gerne %>%
  count(words) %>%
  filter(n > 10) %>%
  ggplot(aes(x = reorder(words,n), y =n)) +
  coord_flip() +
  geom_bar(aes(fill=n<5000), stat = "identity") +
  labs(x = "Genre", y = "Count") +
  theme_bw() +
  theme(legend.position="none")
```



Correlation There is a slight correlation between Size and Average.User.Rating. It would implies that large games have great quality at some point such as graphic.

```
cor_game <- mobilegame[c("Price","Average.User.Rating","User.Rating.Count","Size")]
cor_game <- na.omit(cor_game)

col <- colorRampPalette(c("#BB4444", "#EE9988", "#FFFFFF", "#77AADD", "#4477AA"))
cor_value <- cor(cor_game)
corrplot(cor_value, method="color", col=col(200),
         order="hclust",
         addCoef.col = "black",
         tl.col="black", tl.srt=45,
)
```

Predictive Analysis

New Data Frame for Modelling Now, we are ready to predict Average User Rating. Before prediction, let's make new data frame combining all data frames that we made in the previous steps with mobile game and deleting useless columns.Then, make Average User Rating column as factor to implement classification. I will use Gradient Boosting and Random Forest, and create train data as 70% and test data as 30%.

```
mobilegame$Average.User.Rating <- as.factor(mobilegame$Average.User.Rating)
mobilegame <- cbind(mobilegame,df_onehot_gerne,df_onehot_apppurchase,df_onehot_lang,df_subtitle,df_onehot_primary,df_onehot_
age,df_onehot_price)
mobilegame <- mobilegame %>%
               select(-URL,-ID,-Description,-In.app.Purchases,-Developer,-Icon.URL,-Name,-Subtitle,-Languages,-Genres,-Prim
ary.Genre,-Age.Rating,-Price)

set.seed(1234)
in_train <- createDataPartition(mobilegame$Average.User.Rating, p = 0.7, list = FALSE)
game_train <- mobilegame[in_train, ]
game_test <- mobilegame[-in_train, ]
```

Automated Parameter Tuning Before modeling, the adequate parameters was calculated by using caret. I will use ntree as default and mtry as 150.

```
set.seed(1234)
rf_training <- train(Average.User.Rating ~.,
                 data=game_train,
                 method="rf",
                 trControl=trainControl(method="repeatedcv", number=5))
rf_training
```

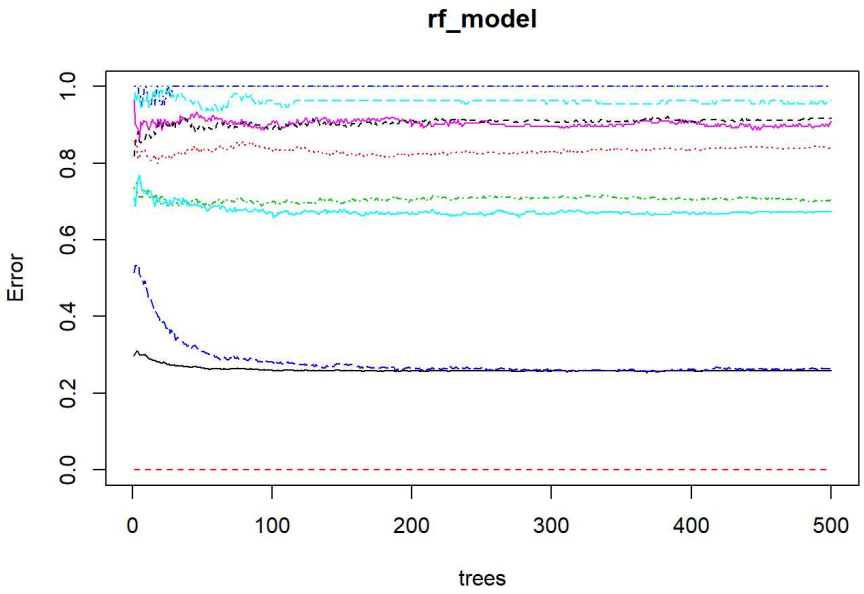
```
## Random Forest
##
## 11891 samples
##   299 predictor
##    10 classes: '0', '1', '1.5', '2', '2.5', '3', '3.5', '4', '4.5', '5'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 1 times)
## Summary of sample sizes: 9513, 9514, 9510, 9514, 9513
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##     2   0.5547054  0.0000000
##   150   0.7433338  0.5961376
##   299   0.7372789  0.5871296
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 150.
```

Random Forest Random Forest is useful for classification as well as regression with powerful performance. so it can be the first option for prediction.

```
set.seed(1234)
rf_model <- randomForest(Average.User.Rating ~., data = game_train, mtry = 150)
rf_model
```

```
##
## Call:
##  randomForest(formula = Average.User.Rating ~ ., data = game_train,      mtry = 150)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 150
##
##         OOB estimate of  error rate: 25.95%
## Confusion matrix:
##        0 1 1.5 2 2.5  3 3.5   4  4.5   5 class.error
## 0   6596 0   0 0   0  0   0   0    0   0   0.0000000
## 1      0 0   0 0   0  0   2   1    4   3   1.0000000
## 1.5    0 0   0 1   3  4   3   7   11  13   1.0000000
## 2      0 0   0 4   7 14   9  21   27  29   0.9639640
## 2.5    0 0   2 4  20 24  31  50   72  19   0.9099099
## 3      0 0   2 4  17 30  52 106  115  34   0.9166667
## 3.5    0 0   0 4  13 30 103 192  260  46   0.8410494
## 4      0 0   0 4   7 30  90 357  636  82   0.7039801
## 4.5    0 0   0 3   8 15  66 303 1469 139   0.2666001
## 5      0 0   0 2   2  5  22  81  355 226   0.6738817
```

```
plot(rf_model)
```




```
rf_predict <- predict(rf_model, game_test)
confusionMatrix(rf_predict,game_test$Average.User.Rating)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1  1.5    2  2.5    3  3.5    4  4.5    5
##        0   2826    0    0    0    0    0    0    0    0    0
##        1      0    1    0    0    0    0    0    0    0    0
##        1.5    0    0    1    1    0    0    0    0    0    0
##        2      0    1    0    1    1    2    1    0    0    3
##        2.5    0    0    2    1    4   13    6    4    1    0
##        3      0    0    1    4   10   18   20   10    3    3
##        3.5    0    1    4    5   17   28   34   35   23    2
##        4      0    0    4   12   26   36  103  144  123   47
##        4.5    0    0    4   18   28   41   90  286  649  144
##        5      0    1    2    5    9   16   23   37   59   98
##
## Overall Statistics
##
##                Accuracy : 0.7416
##                  95% CI : (0.7293, 0.7535)
##     No Information Rate : 0.555
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.5933
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0  Class: 1 Class: 1.5  Class: 2 Class: 2.5
## Sensitivity             1.000 0.2500000  0.0555556 0.0212766  0.0421053
## Specificity             1.000 1.0000000  0.9998029 0.9984143  0.9945968
## Pos Pred Value          1.000 1.0000000  0.5000000 0.1111111  0.1290323
## Neg Pred Value          1.000 0.9994107  0.9966601 0.9909502  0.9820194
## Prevalence              0.555 0.0007855  0.0035350 0.0092302  0.0186567
## Detection Rate          0.555 0.0001964  0.0001964 0.0001964  0.0007855
## Detection Prevalence    0.555 0.0001964  0.0003928 0.0017675  0.0060880
## Balanced Accuracy       1.000 0.6250000  0.5276792 0.5098454  0.5183510
##                      Class: 3 Class: 3.5 Class: 4 Class: 4.5 Class: 5
## Sensitivity          0.116883   0.122744  0.27907     0.7564  0.32997
## Specificity          0.989672   0.976116  0.92330     0.8557  0.96830
## Pos Pred Value       0.260870   0.228188  0.29091     0.5151  0.39200
## Neg Pred Value       0.972925   0.950840  0.91908     0.9455  0.95890
## Prevalence           0.030244   0.054399  0.10134     0.1685  0.05833
## Detection Rate       0.003535   0.006677  0.02828     0.1275  0.01925
## Detection Prevalence 0.013551   0.029262  0.09721     0.2474  0.04910
## Balanced Accuracy    0.553278   0.549430  0.60118     0.8061  0.64913
```

Gradient Boosting Here, it is Gradient Boosting model, which is also suitable for classification, using multinomial distribution by loss. The prediction results generate numerical values so we will convert those figures as class types. Then, we can recognize that the accuracy is highly similar to that of Random Forest.

```
set.seed(1234)
gb_model <- gbm(Average.User.Rating ~ . ,
                data = game_train,
                distribution = "multinomial",
                n.trees = 500,
                shrinkage = 0.01,
                interaction.depth = 4)
gb_model
```

```
## gbm(formula = Average.User.Rating ~ ., distribution = "multinomial",
##     data = game_train, n.trees = 500, interaction.depth = 4,
##     shrinkage = 0.01)
## A gradient boosted model with multinomial loss function.
## 500 iterations were performed.
## There were 299 predictors of which 109 had non-zero influence.
```

```
gb_predict <- predict(gb_model, game_test, n.trees = 500, type = "response")
labels = colnames(gb_predict)[apply(gb_predict, 1, which.max)]
confusionMatrix(as.factor(labels),game_test$Average.User.Rating)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1  1.5    2  2.5    3  3.5    4  4.5    5
##        0   2826    0    0    0    0    0    0    0    0    0
##        1      0    0    1    0    0    0    0    0    0    0
##        1.5    0    0    0    0    0    0    0    0    0    0
##        2      0    0    1    0    0    1    1    0    1    1
##        2.5    0    0    0    0    1    4    5    3    1    0
##        3      0    1    1    1    2   10   11    5    1    1
##        3.5    0    0    3    5    7   26   24   25    8    2
##        4      0    1    3    7   26   30   77   98   78   25
##        4.5    0    1    7   27   48   69  134  356  724  177
##        5      0    1    2    7   11   14   25   29   45   91
##
## Overall Statistics
##
##                Accuracy : 0.7412
##                  95% CI : (0.7289, 0.7532)
##     No Information Rate : 0.555
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.5893
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0  Class: 1 Class: 1.5  Class: 2 Class: 2.5
## Sensitivity             1.000 0.0000000   0.000000 0.0000000  0.0105263
## Specificity             1.000 0.9998035   1.000000 0.9990089  0.9973984
## Pos Pred Value          1.000 0.0000000        NaN 0.0000000  0.0714286
## Neg Pred Value          1.000 0.9992143   0.996465 0.9907608  0.9814888
## Prevalence              0.555 0.0007855   0.003535 0.0092302  0.0186567
## Detection Rate          0.555 0.0000000   0.000000 0.0000000  0.0001964
## Detection Prevalence    0.555 0.0001964   0.000000 0.0009819  0.0027494
## Balanced Accuracy       1.000 0.4999017   0.500000 0.4995045  0.5039624
##                      Class: 3 Class: 3.5 Class: 4 Class: 4.5 Class: 5
## Sensitivity          0.064935   0.086643  0.18992     0.8438  0.30640
## Specificity          0.995342   0.984216  0.94602     0.8066  0.97205
## Pos Pred Value       0.303030   0.240000  0.28406     0.4692  0.40444
## Neg Pred Value       0.971536   0.949319  0.91194     0.9622  0.95767
## Prevalence           0.030244   0.054399  0.10134     0.1685  0.05833
## Detection Rate       0.001964   0.004713  0.01925     0.1422  0.01787
## Detection Prevalence 0.006481   0.019639  0.06775     0.3030  0.04419
## Balanced Accuracy    0.530139   0.535429  0.56797     0.8252  0.63923
```

Random Forest without data partition The sequential data set results in the higher accuracy because it seems mostly the rating values have either 0 or 4.5, meaning it is easier to predict than the previous data set.

```
in_train2 <- 1:12741
game_train2 <- mobilegame[in_train2, ]
game_test2 <- mobilegame[-in_train2, ]

set.seed(1234)
rf_model2 <- randomForest(Average.User.Rating ~., data = game_train2)
rf_predict2 <- predict(rf_model2, game_test)
confusionMatrix(rf_predict2,game_test$Average.User.Rating)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1  1.5    2  2.5    3  3.5    4  4.5    5
##        0   2826    1    3    2    9   10   10    3    2   19
##        1      0    0    0    0    0    0    0    0    0    0
##        1.5    0    0    3    0    0    0    0    0    0    0
##        2      0    0    0    8    0    0    0    0    0    0
##        2.5    0    0    0    0    6    0    0    0    0    0
##        3      0    0    0    0    3   24    0    0    0    0
##        3.5    0    1    2    1    3   12   71    3    2    3
##        4      0    0    2    5   14   15   27  225   16    8
##        4.5    0    2    6   29   58   91  167  282  832  218
##        5      0    0    2    2    2    2    2    3    6   49
##
## Overall Statistics
##
##                Accuracy : 0.7942
##                  95% CI : (0.7828, 0.8052)
##     No Information Rate : 0.555
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.6689
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0  Class: 1 Class: 1.5 Class: 2 Class: 2.5
## Sensitivity            1.0000 0.0000000  0.1666667 0.170213   0.063158
## Specificity            0.9740 1.0000000  1.0000000 1.000000   1.000000
## Pos Pred Value         0.9795       NaN  1.0000000 1.000000   1.000000
## Neg Pred Value         1.0000 0.9992145  0.9970525 0.992329   0.982501
## Prevalence             0.5550 0.0007855  0.0035350 0.009230   0.018657
## Detection Rate         0.5550 0.0000000  0.0005892 0.001571   0.001178
## Detection Prevalence   0.5666 0.0000000  0.0005892 0.001571   0.001178
## Balanced Accuracy      0.9870 0.5000000  0.5833333 0.585106   0.531579
##                      Class: 3 Class: 3.5 Class: 4 Class: 4.5 Class: 5
## Sensitivity          0.155844    0.25632  0.43605     0.9697 0.164983
## Specificity          0.999392    0.99439  0.98099     0.7985 0.996038
## Pos Pred Value       0.888889    0.72449  0.72115     0.4938 0.720588
## Neg Pred Value       0.974334    0.95875  0.93912     0.9924 0.950637
## Prevalence           0.030244    0.05440  0.10134     0.1685 0.058327
## Detection Rate       0.004713    0.01394  0.04419     0.1634 0.009623
## Detection Prevalence 0.005302    0.01925  0.06127     0.3309 0.013354
## Balanced Accuracy    0.577618    0.62536  0.70852     0.8841 0.580510
```

Conclusion for Average User Rating

The Average User Rating values were predicted. The data set has great many of variables so I used one-hot encoding primarily. Random Forest and Gradient Boosting represent both around 75% accuracy and we can see that the models depend on the format of training data set: partitional and sequential. I tried SVM, bagging, and so on but they are not suitable for this data set showing lower or a bit similar accuracy.

Predicting the Successful Apps

Introduction

The model is designed to predict the success of a game. A success of a game depends on many factors such as developer, genre, size, etc but most importantly on the **Average User Rating** and any game having an average user rating of greater than or equal to 4.5 is a successful game. The question to answer using the predictive analysis is **Can you predict successful games?** ***

Loading the Libraries

```
library(ggthemes)
library(skimr)
library(factoextra)
library(cluster)
library(dplyr)
library(randomForest)
library(caret)
library(naniar)
library(highcharter)
library(VIM)
library(mice)
library(tidyr)
library(reshape2)
library(stringr)
```

Reading the Data

```
appstore_games_og <- read.csv("appstore_games.csv", header=TRUE, stringsAsFactors = FALSE)
```

Checking the Data Structure and Abnormalities

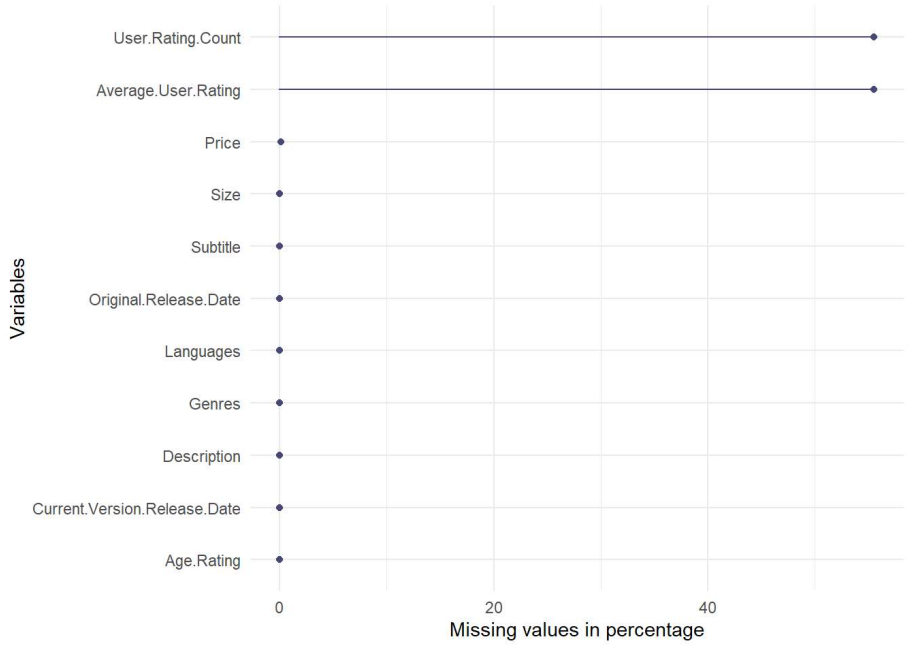
A quick view at the data set shows that there are 10 columns with 17007 rows. A total of 13 factor variables and 5 numeric variables are present. The factor variables have complete data with no missing values where are the numeric variables such as `Average.User.Rating` and `User.Rating.Count` have 9446 missing variables. `Price` and `Size` have less amount of missing variables.

Storing the Required Data

The data has many columns such as URL, Icon.URL, Name, etc which are not useful for the analysis. Hence only the columns required for the analysis are selected.

```
appstore_games <- appstore_games_og %>% select(Average.User.Rating, User.Rating.Count, Price, Size, Subtitle, Description, G
enres, Languages, Age.Rating, Price, Original.Release.Date, Current.Version.Release.Date)
```

Checking the Percentage of the Missing Data



`Average.User.Rating` and `User.Rating.Count` have 55.54% missing values and small percentage of missing values in `Price` and `Size`.
***

Data Imputation - Handling the Missing Values

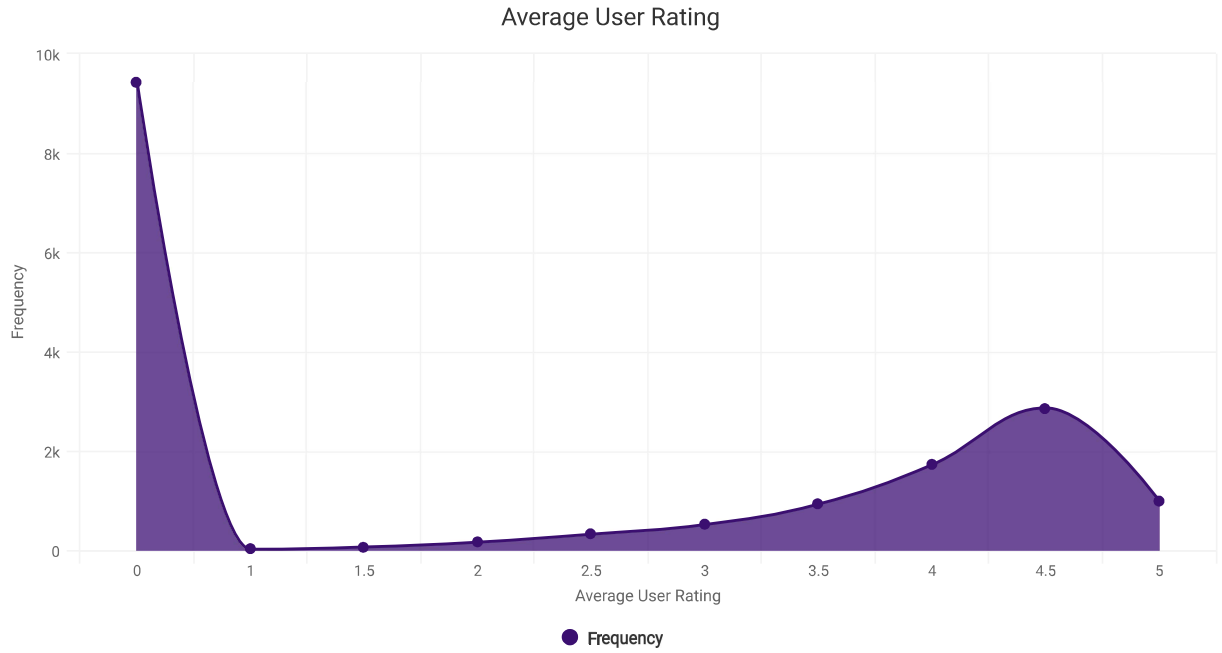
Based on the description from Kaggle, `Average.User.Rating` requires at least 5 ratings in order to display the Average.User.Rating, hence replacing the NA values with 0 in order to keep the remaining data required for analysis. Similarly, `User.Rating.Count` columns are NA if the number of ratings are below 5, thus substituting the values of NA with 0. `Price` and `Size` rows have NA but the percentage of missing values is

not significant and since there is no information about missing values hence for the sake of analysis, I drop these values. Thus, the data is now complete for conducting the predictive analysis.

```
appstore_games$User.Rating.Count[is.na(appstore_games$User.Rating.Count)] <- 0
appstore_games$Average.User.Rating[is.na(appstore_games$Average.User.Rating)] <- 0
appstore_games <- na.omit(appstore_games)
```

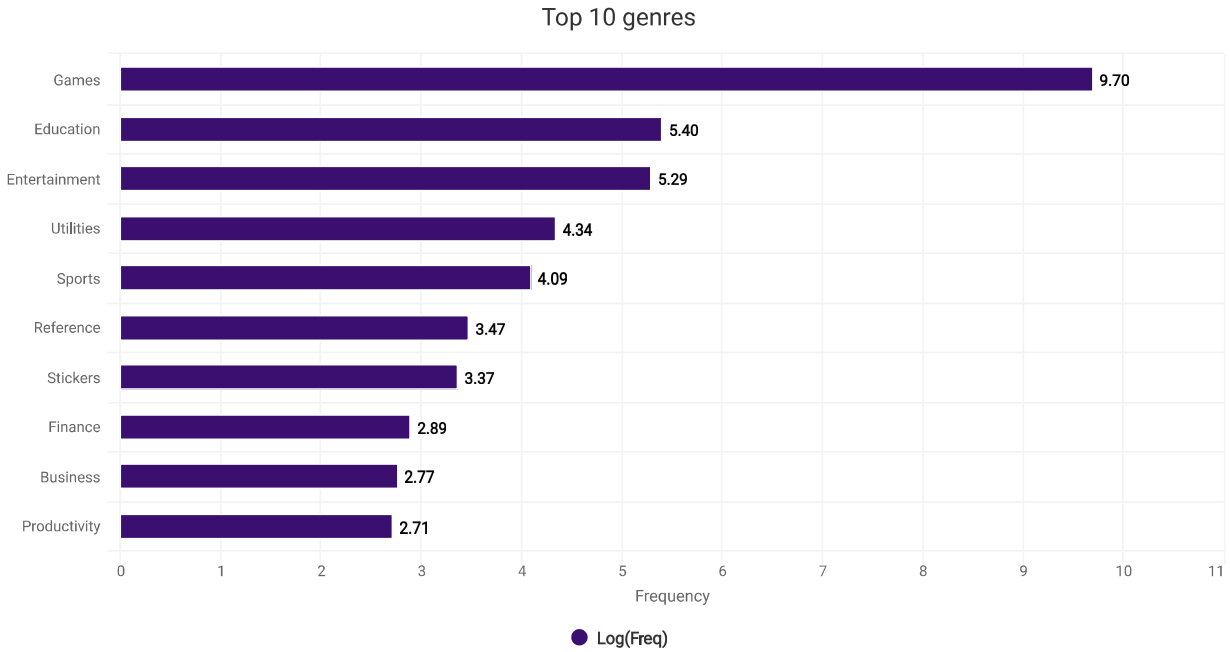
Exploratory Analysis

Exploring the average user rating



Since, most of the values are accumulated near the zero, hence there are a lot of unsuccessful apps.

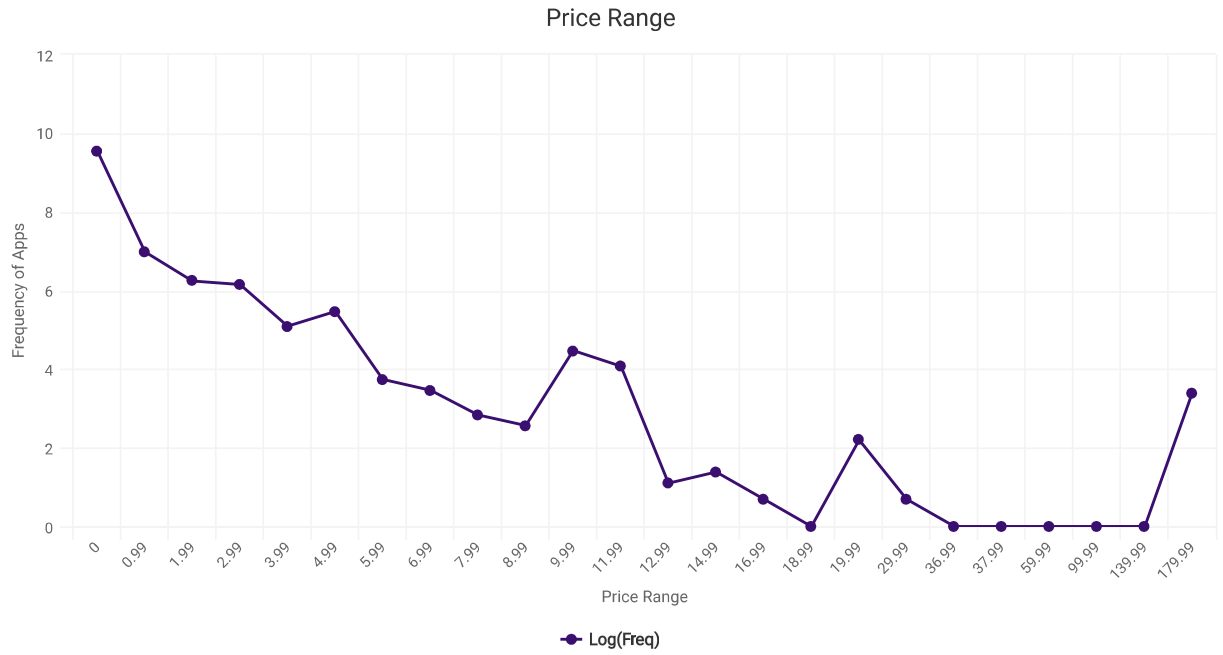
Exploring the genres



The games genre is present in almost most of the apps. Followed by the genres shows in the bar chart above.

Exploring the price range

Most of the apps on the appstore are free. As the price goes on increasing the frequency of the app decreases.



***

Data Preparation for Analysis

The data has many words in the `Description` and hence not considering the `Description` column data would lead to loss of some important data. The number of words in the description column are counted to check the traits of the successful apps. In the similar fashion, the number of words in the subtitle column are stored in separate data frame.

```
word_count_description <- sapply(gregexpr("\\S+", appstore_games$Description), length)
word_count_subtitle <- str_count(appstore_games$Subtitle, '\\s+')+1
subtitle_present <- ifelse(word_count_subtitle <= 1 , 0 , 1)
```

The number of genres and number of languages for each game are counted and stored in separate data frames.

```
genre_count <- sapply(gregexpr("\\S+", appstore_games$Genres), length)
language_count <- sapply(gregexpr("\\S+", appstore_games$Languages), length)
```

A new column is created which indicate whether the app is price free or not. **1** indicates the app is free and **0** indicates the app is paid.

```
price_free <- ifelse(appstore_games$Price >0, 0, 1)
```

An app is considered to be successful if the `Average.User.Rating` is greater than or equal to 4.5. Hence, substituting the values of `Average.User.Rating` column with **Successful** and **Unsuccessful** Apps.

```
successful_app <- ifelse(appstore_games$Average.User.Rating >= 4.5, 1,0)
successful_app[successful_app == 0] <- "Unsuccessful"
successful_app[successful_app == 1] <- "Successful"
```

Simplifying the `Age.Rating` column and removing the + sign.

```
age_rating <- appstore_games %>% separate(Age.Rating, c("Age.Rating"), extra = "drop")
age_rating <- age_rating$Age.Rating
```

The size of the app is a huge number. Thus, for conducting the analysis the size is transformed to the size of log10.

```
size <- log10(appstore_games$Size)
```

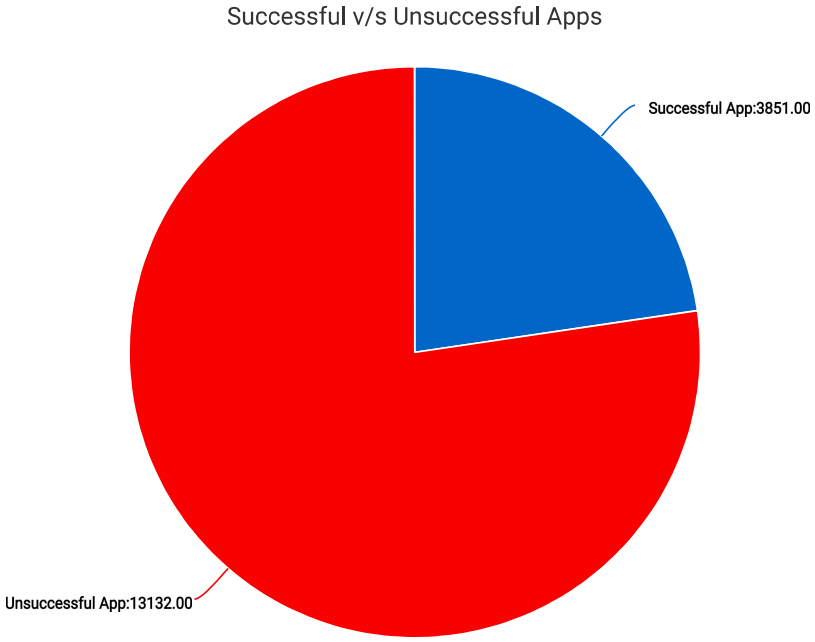
Separating the year from the `Original.Release.Date` and `Current.Version.Release.Date` and storing the data.

```
year_release <- format(as.Date(appstore_games$Original.Release.Date, format="%d/%m/%Y"),"%Y")
year_updated<- format(as.Date(appstore_games$Current.Version.Release.Date, format="%d/%m/%Y"),"%Y")
```

Combining all the single data sets into a single data frame which will be used for conducting the predictive analysis.

```
appstore_games_clean <- data.frame(successful_app ,appstore_games$User.Rating.Count, appstore_games$Price, size, subtitle_pr
esent, word_count_description, genre_count, language_count, age_rating, price_free, year_release, year_updated)
colnames(appstore_games_clean) <- c("Successful_App","User_Rating_Count", "Price", "Size", "Subtitle_Present", "Description_
Length", "Genre_Count", "Language_Count", "Age_Rating", "Price_Free", "Original_Release_Year", "Year_Updated")

appstore_games_clean <- appstore_games_clean %>%
  mutate(Original_Release_Year = str_sub(Original_Release_Year, 3, -1))

appstore_games_clean <- appstore_games_clean %>%
  mutate(Year_Updated = str_sub(Year_Updated, 3, -1))
```

Successful v/s Unsuccessful App Ratio



***

Predictive Analysis

Data Structure

The structure of the data is shown below. Changing the characters into numeric for the sake of analysis.

```
str(appstore_games_clean)
```

```
## 'data.frame':    16983 obs. of  12 variables:
##  $ Successful_App       : Factor w/ 2 levels "Successful","Unsuccessful": 2 2 2 2 2 2 2 2 2 2 ...
##  $ User_Rating_Count    : num  3553 284 8376 190394 28 ...
##  $ Price                : num  2.99 1.99 0 0 2.99 0 0 0.99 0 0 ...
##  $ Size                 : num  7.2 7.09 5.83 7.33 7.54 ...
##  $ Subtitle_Present     : num  0 0 0 0 0 1 0 0 0 0 ...
##  $ Description_Length   : int  259 204 97 267 365 368 109 129 61 83 ...
##  $ Genre_Count          : int  3 3 3 3 4 4 4 3 4 3 ...
##  $ Language_Count       : int  17 1 1 17 15 1 1 1 1 1 ...
##  $ Age_Rating           : Factor w/ 4 levels "12","17","4",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ Price_Free           : num  0 0 1 1 0 1 1 0 1 1 ...
##  $ Original_Release_Year: chr  "08" "08" "08" "08" ...
##  $ Year_Updated         : chr  "17" "18" "17" "17" ...
```

```
appstore_games_clean$Original_Release_Year <- as.numeric(appstore_games_clean$Original_Release_Year)
appstore_games_clean$Year_Updated <- as.numeric(appstore_games_clean$Year_Updated)
```

The `Successful_App` column is a factor which is good because this column is to be predicted.

Splitting the training and testing data

The data is split into 2 categories, training and testing data sets. 80% data is stored into train data set while the remaining 20% is stored in test data set.

```
set.seed(123)
in_train <- createDataPartition(appstore_games_clean$Successful_App, p = 0.80, list = FALSE)

train <- appstore_games_clean[in_train, ]
test <- appstore_games_clean[-in_train, ]
```

Running the random forest

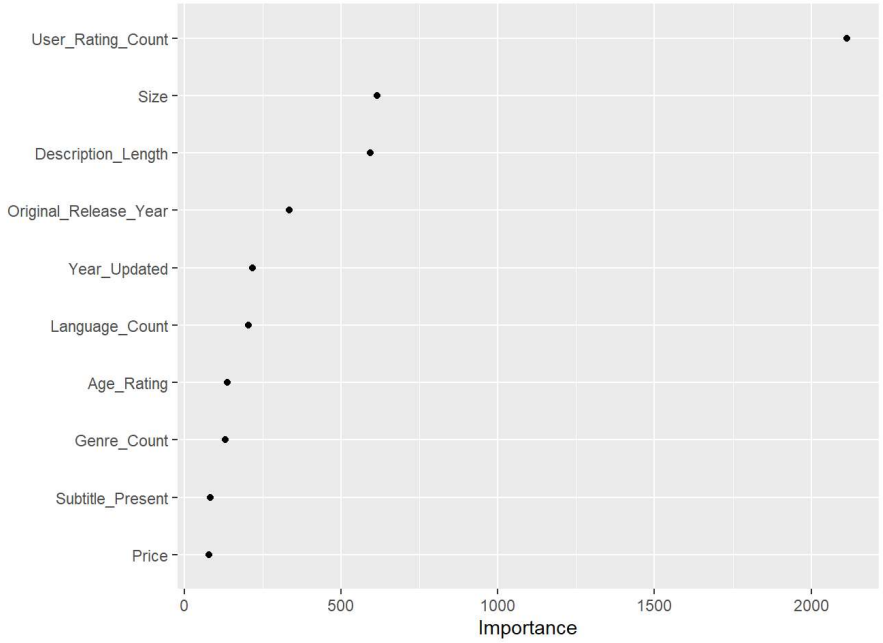
Running the random forest on the training data set.

```
set.seed(123)
rf1 = randomForest(Successful_App~., data = train)
rf1
```

```
##
## Call:
##  randomForest(formula = Successful_App ~ ., data = train)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 14.82%
## Confusion matrix:
##              Successful Unsuccessful class.error
## Successful         2090          991  0.32164882
## Unsuccessful       1022         9484  0.09727775
```

The Random Forest has considered 500 trees with an mtry of 3. The out-of-bag error is 14.82%.

The important variables which the random forest has considered are shown in the graph below.



Checking the predictions by the random forest on the training data set and plotting the confusion matrix to check the **Accuracy** of the predictions.

```
set.seed(123)
prediction <- predict(rf1, train)
confusionMatrix(prediction , train$Successful_App)
```

```
## Confusion Matrix and Statistics
##
##               Reference
## Prediction     Successful Unsuccessful
##   Successful         3078            1
##   Unsuccessful          3        10505
##
##                Accuracy : 0.9997
##                  95% CI : (0.9992, 0.9999)
##     No Information Rate : 0.7732
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9992
##
##  Mcnemar's Test P-Value : 0.6171
##
##             Sensitivity : 0.9990
##             Specificity : 0.9999
##          Pos Pred Value : 0.9997
##          Neg Pred Value : 0.9997
##              Prevalence : 0.2268
##          Detection Rate : 0.2265
##    Detection Prevalence : 0.2266
##       Balanced Accuracy : 0.9995
##
##        'Positive' Class : Successful
##
```

From the accuracy it can be observed that the **Random Forest** is able to predict the train data with an accuracy of 99.97% having a 95% Confidence Interval that ranges between of 99.92% to 99.99% accuracy. The accuracy is quite high and there might be a chance of over-fitting of the model.

Predicting the outcomes

The random forest is used to predict the successful apps for the testing data set. The results can be seen in the confusion matrix.

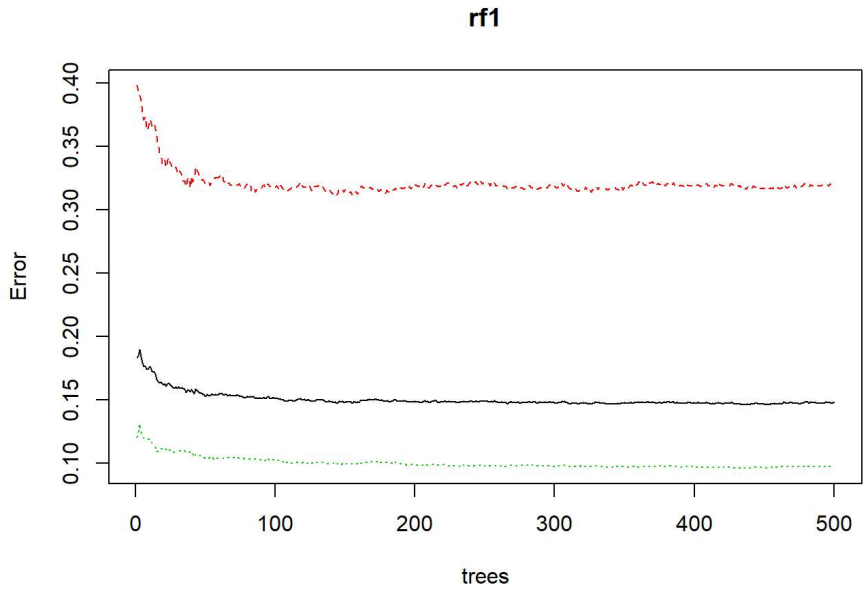
```
set.seed(123)
prediction_test <- predict(rf1, test)
confusionMatrix(prediction_test , test$Successful_App)
```

```
## Confusion Matrix and Statistics
##
##               Reference
## Prediction     Successful Unsuccessful
##   Successful          534           243
##   Unsuccessful        236         2383
##
##                Accuracy : 0.859
##                  95% CI : (0.8468, 0.8705)
##     No Information Rate : 0.7733
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.599
##
##  Mcnemar's Test P-Value : 0.784
##
##             Sensitivity : 0.6935
##             Specificity : 0.9075
##          Pos Pred Value : 0.6873
##          Neg Pred Value : 0.9099
##              Prevalence : 0.2267
##          Detection Rate : 0.1572
##    Detection Prevalence : 0.2288
##       Balanced Accuracy : 0.8005
##
##        'Positive' Class : Successful
##
```

Looking at the results, the random forest has successfully predicted with an accuracy of 85.9% having 95% Confidence Interval between 84.68% to 87.05% accuracy. ***

Manual Tuning to Avoid Over fitting.

Tuning the number of trees.



As the number of trees grow the

OOB Error starts to drop down and then becomes constant. Hence, to find the correct trees which will give maximum prediction accuracy without over fitting, the following loop is run.

```
control <- trainControl(method="repeatedcv", number=10, repeats=3, search="grid")
tunegrid <- expand.grid(.mtry=c(sqrt(ncol(train))))
modellist <- list()
set.seed(123)
for (ntree in c(200, 300, 400, 500 ,600, 700)) {
  set.seed(123)
  fit <- train(Successful_App~., data=train, method="rf", metric="Accuracy", tuneGrid=tunegrid, trControl=control, ntree=ntr
ee)
  key <- toString(ntree)
  modellist[[key]] <- fit
}
results <- resamples(modellist)
summary(results)
```

```
##
## Call:
## summary.resamples(object = results)
##
## Models: 200, 300, 400, 500, 600, 700
## Number of resamples: 30
##
## Accuracy
##          Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 200 0.8375000 0.8463944 0.8520971 0.8524329 0.8579323 0.8689249    0
## 300 0.8338235 0.8485741 0.8520971 0.8525799 0.8555393 0.8682855    0
## 400 0.8360294 0.8498067 0.8528330 0.8538066 0.8580897 0.8668138    0
## 500 0.8360294 0.8490705 0.8524109 0.8535859 0.8583517 0.8682855    0
## 600 0.8375000 0.8490705 0.8527788 0.8534139 0.8583780 0.8667158    0
## 700 0.8352941 0.8490705 0.8520971 0.8532668 0.8586156 0.8681885    0
##
## Kappa
##          Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## 200 0.5367276 0.5664547 0.5769332 0.5800310 0.5940489 0.6262771    0
## 300 0.5213192 0.5696466 0.5821981 0.5802162 0.5889580 0.6196242    0
## 400 0.5293051 0.5743579 0.5832832 0.5831736 0.5981229 0.6152027    0
## 500 0.5303867 0.5721342 0.5825601 0.5828459 0.5972525 0.6212088    0
## 600 0.5335266 0.5729682 0.5820242 0.5824819 0.5943562 0.6195419    0
## 700 0.5266493 0.5684122 0.5815975 0.5817401 0.5925946 0.6228796    0
```

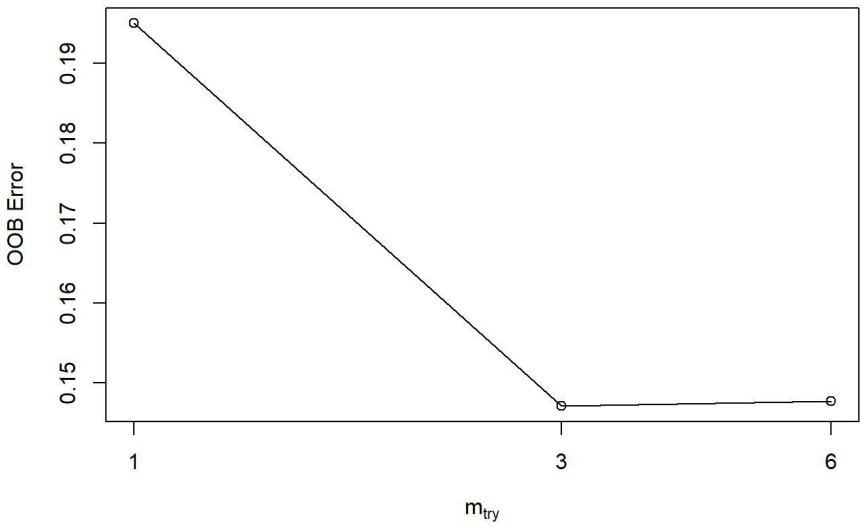
From the summary of the results, the accuracy is more or less the same for all the trees but the accuracy is best when the trees = 300.
Thus 300 trees are selected to run the random forest.

Tuning the mtry.

Using 300 trees, tuning the random forest to obtain the best mtry.

```
t <- tuneRF(train[,-1], train[,1],
            stepFactor = 0.5,
            plot = TRUE,
            ntreeTry = 300,
            trace = TRUE,
            improve = 0.5)
```

```
## mtry = 3  OOB error = 14.71%
## Searching left ...
## mtry = 6     OOB error = 14.77%
## -0.004002001 0.5
## Searching right ...
## mtry = 1     OOB error = 19.5%
## -0.3251626 0.5
```



The OOB Error is low when mtry is 3.
Thus, tuning the random forest using **trees = 300** and **mtry = 3** to obtain a good fit for the model. ***

Tuned Random Forest

```
set.seed(123)
rf2 = randomForest(Successful_App~., data = train, ntree = 300, mtry = 3, importance = TRUE, proximity = TRUE)
rf2
```

```
##
## Call:
##  randomForest(formula = Successful_App ~ ., data = train, ntree = 300,      mtry = 3, importance = TRUE, proximity = TRU
E)
##                Type of random forest: classification
##                      Number of trees: 300
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 14.69%
## Confusion matrix:
##              Successful Unsuccessful class.error
## Successful         2112          969  0.31450828
## Unsuccessful       1027         9479  0.09775366
```

The OOB error is 14.69% which is less than the previous one. Checking the predictions by the random forest on the training data set. Plotting the confusion matrix to check the **Accuracy** of the predictions.

Predictions using tuned random forest

```
set.seed(123)
prediction2 <- predict(rf2, train)
confusionMatrix(prediction2 , train$Successful_App)
```

```
## Confusion Matrix and Statistics
##
##              Reference
## Prediction     Successful Unsuccessful
##   Successful         3079            1
##   Unsuccessful          2        10505
##
##                Accuracy : 0.9998
##                  95% CI : (0.9994, 1)
##     No Information Rate : 0.7732
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9994
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.9994
##             Specificity : 0.9999
##          Pos Pred Value : 0.9997
##          Neg Pred Value : 0.9998
##              Prevalence : 0.2268
##          Detection Rate : 0.2266
##    Detection Prevalence : 0.2267
##       Balanced Accuracy : 0.9996
##
##        'Positive' Class : Successful
##
```

The accuracy is 99.98% for the training data set. With 95% Confidence Interval the accuracy ranges between 99.98% to 100%.

Predicting the outcomes

```
set.seed(123)
prediction_test2 <- predict(rf2, test)
confusionMatrix(prediction_test2 , test$Successful_App)
```

```
## Confusion Matrix and Statistics
##
##              Reference
## Prediction     Successful Unsuccessful
##   Successful          532          243
##   Unsuccessful        238         2383
##
##                Accuracy : 0.8584
##                  95% CI : (0.8462, 0.8699)
##     No Information Rate : 0.7733
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.597
##
##  Mcnemar's Test P-Value : 0.8553
##
##             Sensitivity : 0.6909
##             Specificity : 0.9075
##          Pos Pred Value : 0.6865
##          Neg Pred Value : 0.9092
##              Prevalence : 0.2267
##          Detection Rate : 0.1567
##    Detection Prevalence : 0.2282
##       Balanced Accuracy : 0.7992
##
##        'Positive' Class : Successful
##
```

Looking at the results, the random forest has successfully predicted with an accuracy of 85.84%. With 95% Confidence Interval the app has an accuracy ranging between 84.62% to 86.99%.
The model has a perfect fit and has a good accuracy of predicting the successful apps. ***

Conclusion for Successful Apps

Using the data set from Kaggle, a predictive model was build to predict the successful apps. Random forest for classification was used to train the data and predict the outcome with the test data. Overall, the model built and is tuned manually to avoid any over fitting. The model predicts with an accuracy of 85.57%.

Predicting the Average number of Daily Ratings

Introduction of Neural Network

This part of the code is aiming at training regression artificial neural network models to predict the average number of daily ratings for the games in order to further determine their popularity. The main methodologies that have been used in this part are One-hot Encoding, Principle Component Analysis (PCA), and Regression Neural Network. This part of analysis will focus on seven columns which are 'User Rating Count', 'Size', 'Price', 'Age Rating', 'Genres', 'Languages' and 'Original Release Date' (which will be processed as existence time) in the original data set. At the end, it will generate a set of ANN models with three different activation functions and various number of hidden nodes and by comparing their sum of squared errors (SSE), it will figure out the most suitable one.

Library and Read Data

```
library(tidyverse)
library(neuralnet)
library(GGally)
library(mltools)
library(caret)
library(ggplot2)
library(foreach)
library(doSNOW)
library(sigmoid)

game_data <- read_csv("appstore_games.csv")
```

Explanatory Analysis

Data Processing

1. Replace all the 'NA' values in 'User Rating Count' column with 0

```
game_data[is.na(game_data[,7]),7] <- 0
```

2. Split the genre and language columns

```
game_data$genre_list <- lapply(game_data$Genres, strsplit, split=", ")
game_data$lang_list <- lapply(game_data$Languages, strsplit, split=", ")
```

3. Get all the values for genre, languages and age ratings. This will be used to create a new data frame for one-hot encoding operation. There are 46 genres, 115 languages and 4 age ratings in total.

```
genres <- unique(unlist(game_data[1:nrow(game_data), 19][[1]]))
genres <- genres[!is.na(genres)]
genres_names <- paste("genre_", genres, sep = '')


langs <- unique(unlist(game_data[1:nrow(game_data), 20][[1]]))
langs <- langs[!is.na(langs)]
langs_names <- paste("language_", langs, sep = '')

ages <- unique(game_data$`Age Rating`)
ages_names <- c('age_above_4','age_above_9','age_above_12','age_above_17')
```

One-hot Encoding

Define a function for checking whether a game has a feature of a certain genre, language or age rating. It helps to transform 'Genre', 'Language' and 'Age Rating' columns into binary (0/1) values.

```
checklist <- function(theList, genre){
  sapply(theList, `%in%`, x = genre)
}
```

1. Create a new Data Frame with wanted columns for storing the data after One-hot encoding.

```
column_names <- c('rating_count', 'price', 'size', genres_names[2:length(genres_names)], langs_names, ages_names, 'release_d
ays')
column_names <- gsub(" ", "_", column_names)
column_names <- gsub("&", "and", column_names)
game_data_new <- data.frame(matrix(ncol = length(column_names), nrow = nrow(game_data)))

colnames(game_data_new) <- column_names
```

2. In this part, I have further processed the data. The 'User Rating Count' column has been processed into daily values and 'Genre', 'Language' and 'Age Rating' columns have been set into 0/1 values by using one-hot encoding method.

```
game_data_new[,1] <- game_data$`User Rating Count`/(as.numeric((as.Date("2019/08/03") - as.Date(game_data$`Original Release
 Date`, "%d/%m/%Y"))) + 1)
game_data_new[,2] <- game_data$Price
game_data_new[,3] <- game_data$Size

# one-hot encoding, transforming all the characters into binary values.
for(i in 4:49){
  game_data_new[,i] <- 1*sapply(game_data$genre_list , checklist, genre = genres[i - 3])
}

for(j in 50:164){
  game_data_new[,j] <- 1*sapply(game_data$lang_list , checklist, genre = langs[j - 49])
}

for (k in 165:168){
  game_data_new[,k] <- 1*sapply(game_data$`Age Rating`, `==`,  x = ages[k - 164])
}

# process the 'Original Release Date' of a game into the number of days it has existed
game_data_new[,169] <- (as.numeric((as.Date("2019/08/03") - as.Date(game_data$`Original Release Date`, "%d/%m/%Y"))))
```

3. Remove rows with NA values

```
game_data_new <- na.omit(game_data_new)
```

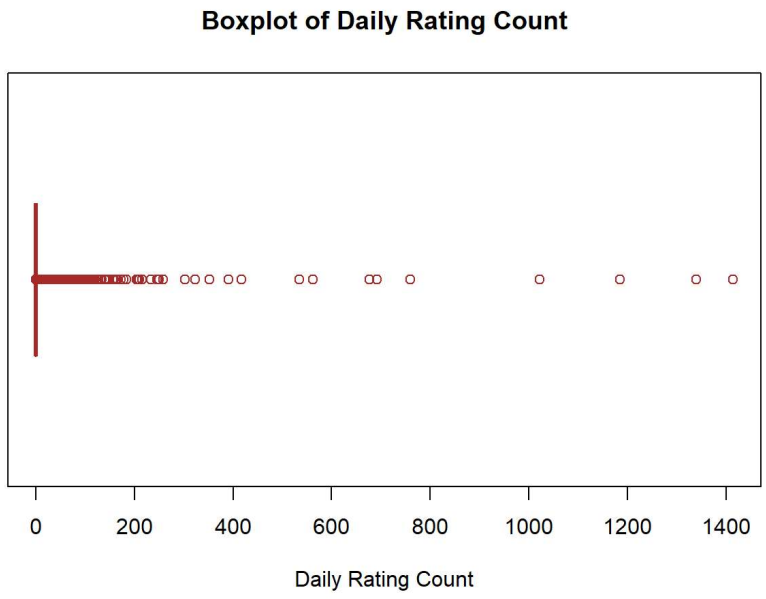
Processing outliers

Outliers have large impacts on the training of regression neural network models. From the quartiles and box plot below, it is noticeable that the original data set is highly concentrated and most of the games have less than 1 average daily ratings while a small group of games have a huge amount of ratings. This is likely to impact the accuracy of the final result.

```
# calculating the quartile for this data set
ordered_count <- game_data_new[order(game_data_new[,1]),1]
q1 <- ordered_count[round(length(ordered_count)/4,0)]
q3 <- ordered_count[round(length(ordered_count)/4*3,0)]
upperWhiskers <- q3 + 1.5*(q3-q1)


basic_info <- data.frame(
  number <- nrow(game_data_new),
  min <- min(game_data_new[,1]),
  max <- max(game_data_new[,1]),
  Q1 <- q1,
  Q3 <- q3,
  upperWhisker <- upperWhiskers
)

colnames(basic_info) <- c("Total Number", "Min", "Max", "First Quartile", "Third Quartile", "Upper Whisker")
basic_info
```

```
##   Total Number Min      Max First Quartile Third Quartile Upper Whisker
## 1        16983   0 1414.332              0     0.03038936    0.07597341
```

```
boxplot(game_data_new[,1],
        col = "orange",
        border = "brown",
        horizontal = TRUE,
        xlab="Daily Rating Count",
        main = "Boxplot of Daily Rating Count"
        )
```



In this part of analysis, it is decided to remove the outlier rows with more than 400 ratings per day.

```
cleaned_data_400 <- filter(game_data_new, game_data_new[,1] <= 400)
```

Rescale the Data

Before PCA and Regression Neural network training, the data needs to be re scaled into a range between 0 to 1. Note that, when using the 'tanh' activation function for training ANN models, it will be re scaled again into a range between -1 to 1.

```
scale01 <- function(x){
  (x - min(x)) / (max(x) - min(x))
}

# rescale the data set
game_data_new_mutated <- cleaned_data_400 %>%
  mutate(rating_count = scale01(rating_count),
         price = scale01(price), size = scale01(size),
         release_days = scale01(release_days))
```

Principle Component Analysis (PCA)

Principle Component Analysis here is for: 1. Reducing the dimensionality - in the original data set, after one-hot encoding, it contains 168 variables and this can lead to a large amount of running time. PCA is able to reduce the number of dimension. 2. Avoiding over fitting - by doing PCA, it can alleviate the over fitting problem for the regression neural network training.

```
# Do the Principle Component Analysis
myPC <- prcomp(game_data_new_mutated[, -1])
```

PCA results - It can be seen from the following table that after PC113, the cumulative percentage has already reached 1, which means the rest of components are irrelevant. However, with 113 variables, it can still cost 4 to 5 hours for the later ANN training. According to the graph below (in which the bars represent for the proportion that each components account for while the red line stands for the cumulative proportion), the top 36 components have already included most of the information (95%). Thus, in this program, I will choose only the top 36 components for neural network training.

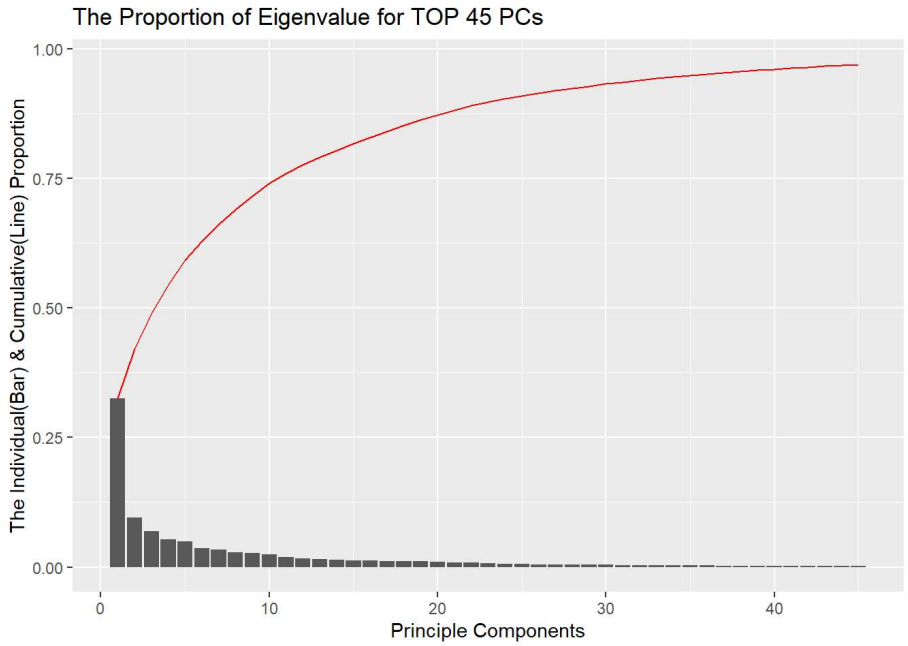
```
ev <- myPC$sdev^2
sumev <- sum(ev)

# calculate the eigenvalue and proportion for each component
PCA_values <- data.frame(
  names <- paste("PC", 1:168, sep = ""),
  number <- c(1:168),
  eigenvalue <- ev,
  Percentage <- ev/sum(ev),
  percent_cum <- cumsum(ev)/sum(ev)
)

# print out the table and draw the plot
colnames(PCA_values) <- c('PC_Name', "PC", 'Eigenvalue', 'Percentage', 'Cumulative_Percentage')
PCA_values_table <- PCA_values[,c(1,3:5)]
PCA_values_table[,2:4] <- round(PCA_values_table[,2:4],4)
PCA_values_table
```

```
##     PC_Name Eigenvalue Percentage Cumulative_Percentage
## 1       PC1     1.1941     0.3256                0.3256
## 2       PC2     0.3496     0.0953                0.4210
## 3       PC3     0.2530     0.0690                0.4900
## 4       PC4     0.1968     0.0537                0.5436
## 5       PC5     0.1786     0.0487                0.5924
## 6       PC6     0.1327     0.0362                0.6285
## 7       PC7     0.1212     0.0330                0.6616
## 8       PC8     0.1016     0.0277                0.6893
## 9       PC9     0.0998     0.0272                0.7165
## 10     PC10     0.0883     0.0241                0.7406
## 11     PC11     0.0701     0.0191                0.7597
## 12     PC12     0.0599     0.0163                0.7761
## 13     PC13     0.0538     0.0147                0.7907
## 14     PC14     0.0502     0.0137                0.8044
## 15     PC15     0.0478     0.0130                0.8174
## 16     PC16     0.0447     0.0122                0.8296
## 17     PC17     0.0426     0.0116                0.8413
## 18     PC18     0.0408     0.0111                0.8524
## 19     PC19     0.0394     0.0108                0.8631
## 20     PC20     0.0363     0.0099                0.8730
## 21     PC21     0.0325     0.0088                0.8819
## 22     PC22     0.0308     0.0084                0.8903
## 23     PC23     0.0258     0.0070                0.8973
## 24     PC24     0.0235     0.0064                0.9037
## 25     PC25     0.0225     0.0061                0.9099
## 26     PC26     0.0187     0.0051                0.9150
## 27     PC27     0.0174     0.0047                0.9197
## 28     PC28     0.0163     0.0044                0.9241
## 29     PC29     0.0152     0.0041                0.9283
## 30     PC30     0.0146     0.0040                0.9323
## 31     PC31     0.0140     0.0038                0.9361
## 32     PC32     0.0135     0.0037                0.9398
## 33     PC33     0.0125     0.0034                0.9432
## 34     PC34     0.0101     0.0028                0.9460
## 35     PC35     0.0101     0.0027                0.9487
## 36     PC36     0.0099     0.0027                0.9514
## 37     PC37     0.0098     0.0027                0.9541
## 38     PC38     0.0088     0.0024                0.9565
## 39     PC39     0.0084     0.0023                0.9588
## 40     PC40     0.0079     0.0022                0.9609
## 41     PC41     0.0075     0.0021                0.9630
## 42     PC42     0.0072     0.0020                0.9649
## 43     PC43     0.0070     0.0019                0.9668
## 44     PC44     0.0064     0.0017                0.9686
## 45     PC45     0.0061     0.0017                0.9702
## 46     PC46     0.0058     0.0016                0.9718
## 47     PC47     0.0054     0.0015                0.9733
## 48     PC48     0.0053     0.0014                0.9747
## 49     PC49     0.0053     0.0014                0.9762
## 50     PC50     0.0051     0.0014                0.9776
## 51     PC51     0.0048     0.0013                0.9789
## 52     PC52     0.0041     0.0011                0.9800
## 53     PC53     0.0037     0.0010                0.9810
## 54     PC54     0.0037     0.0010                0.9820
## 55     PC55     0.0036     0.0010                0.9830
## 56     PC56     0.0036     0.0010                0.9840
## 57     PC57     0.0035     0.0010                0.9849
## 58     PC58     0.0035     0.0009                0.9859
## 59     PC59     0.0033     0.0009                0.9868
## 60     PC60     0.0032     0.0009                0.9877
## 61     PC61     0.0030     0.0008                0.9885
## 62     PC62     0.0029     0.0008                0.9893
## 63     PC63     0.0028     0.0008                0.9900
## 64     PC64     0.0026     0.0007                0.9907
## 65     PC65     0.0025     0.0007                0.9914
## 66     PC66     0.0023     0.0006                0.9920
## 67     PC67     0.0022     0.0006                0.9926
## 68     PC68     0.0021     0.0006                0.9932
## 69     PC69     0.0020     0.0005                0.9938
## 70     PC70     0.0018     0.0005                0.9943
## 71     PC71     0.0018     0.0005                0.9948
## 72     PC72     0.0017     0.0005                0.9952
## 73     PC73     0.0016     0.0004                0.9957
## 74     PC74     0.0015     0.0004                0.9961
## 75     PC75     0.0013     0.0003                0.9964
## 76     PC76     0.0012     0.0003                0.9967
## 77     PC77     0.0011     0.0003                0.9970
## 78     PC78     0.0009     0.0002                0.9973
```

```
## 79     PC79     0.0009     0.0002              0.9975
## 80     PC80     0.0008     0.0002              0.9977
## 81     PC81     0.0007     0.0002              0.9979
## 82     PC82     0.0006     0.0002              0.9981
## 83     PC83     0.0006     0.0002              0.9982
## 84     PC84     0.0006     0.0002              0.9984
## 85     PC85     0.0005     0.0001              0.9985
## 86     PC86     0.0005     0.0001              0.9987
## 87     PC87     0.0004     0.0001              0.9988
## 88     PC88     0.0004     0.0001              0.9989
## 89     PC89     0.0003     0.0001              0.9990
## 90     PC90     0.0003     0.0001              0.9991
## 91     PC91     0.0003     0.0001              0.9991
## 92     PC92     0.0002     0.0001              0.9992
## 93     PC93     0.0002     0.0001              0.9993
## 94     PC94     0.0002     0.0001              0.9993
## 95     PC95     0.0002     0.0001              0.9994
## 96     PC96     0.0002     0.0001              0.9994
## 97     PC97     0.0002     0.0000              0.9995
## 98     PC98     0.0002     0.0000              0.9995
## 99     PC99     0.0002     0.0000              0.9996
## 100   PC100     0.0002     0.0000              0.9996
## 101   PC101     0.0001     0.0000              0.9997
## 102   PC102     0.0001     0.0000              0.9997
## 103   PC103     0.0001     0.0000              0.9997
## 104   PC104     0.0001     0.0000              0.9998
## 105   PC105     0.0001     0.0000              0.9998
## 106   PC106     0.0001     0.0000              0.9998
## 107   PC107     0.0001     0.0000              0.9998
## 108   PC108     0.0001     0.0000              0.9999
## 109   PC109     0.0001     0.0000              0.9999
## 110   PC110     0.0001     0.0000              0.9999
## 111   PC111     0.0001     0.0000              0.9999
## 112   PC112     0.0001     0.0000              0.9999
## 113   PC113     0.0001     0.0000              0.9999
## 114   PC114     0.0000     0.0000              1.0000
## 115   PC115     0.0000     0.0000              1.0000
## 116   PC116     0.0000     0.0000              1.0000
## 117   PC117     0.0000     0.0000              1.0000
## 118   PC118     0.0000     0.0000              1.0000
## 119   PC119     0.0000     0.0000              1.0000
## 120   PC120     0.0000     0.0000              1.0000
## 121   PC121     0.0000     0.0000              1.0000
## 122   PC122     0.0000     0.0000              1.0000
## 123   PC123     0.0000     0.0000              1.0000
## 124   PC124     0.0000     0.0000              1.0000
## 125   PC125     0.0000     0.0000              1.0000
## 126   PC126     0.0000     0.0000              1.0000
## 127   PC127     0.0000     0.0000              1.0000
## 128   PC128     0.0000     0.0000              1.0000
## 129   PC129     0.0000     0.0000              1.0000
## 130   PC130     0.0000     0.0000              1.0000
## 131   PC131     0.0000     0.0000              1.0000
## 132   PC132     0.0000     0.0000              1.0000
## 133   PC133     0.0000     0.0000              1.0000
## 134   PC134     0.0000     0.0000              1.0000
## 135   PC135     0.0000     0.0000              1.0000
## 136   PC136     0.0000     0.0000              1.0000
## 137   PC137     0.0000     0.0000              1.0000
## 138   PC138     0.0000     0.0000              1.0000
## 139   PC139     0.0000     0.0000              1.0000
## 140   PC140     0.0000     0.0000              1.0000
## 141   PC141     0.0000     0.0000              1.0000
## 142   PC142     0.0000     0.0000              1.0000
## 143   PC143     0.0000     0.0000              1.0000
## 144   PC144     0.0000     0.0000              1.0000
## 145   PC145     0.0000     0.0000              1.0000
## 146   PC146     0.0000     0.0000              1.0000
## 147   PC147     0.0000     0.0000              1.0000
## 148   PC148     0.0000     0.0000              1.0000
## 149   PC149     0.0000     0.0000              1.0000
## 150   PC150     0.0000     0.0000              1.0000
## 151   PC151     0.0000     0.0000              1.0000
## 152   PC152     0.0000     0.0000              1.0000
## 153   PC153     0.0000     0.0000              1.0000
## 154   PC154     0.0000     0.0000              1.0000
## 155   PC155     0.0000     0.0000              1.0000
## 156   PC156     0.0000     0.0000              1.0000
## 157   PC157     0.0000     0.0000              1.0000
## 158   PC158     0.0000     0.0000              1.0000
```

```
## 159   PC159     0.0000     0.0000               1.0000
## 160   PC160     0.0000     0.0000               1.0000
## 161   PC161     0.0000     0.0000               1.0000
## 162   PC162     0.0000     0.0000               1.0000
## 163   PC163     0.0000     0.0000               1.0000
## 164   PC164     0.0000     0.0000               1.0000
## 165   PC165     0.0000     0.0000               1.0000
## 166   PC166     0.0000     0.0000               1.0000
## 167   PC167     0.0000     0.0000               1.0000
## 168   PC168     0.0000     0.0000               1.0000
```

```
ggplot(data = PCA_values[1:45,]) +
  geom_col(aes(x = PC, y = Percentage)) +
  geom_line(aes(x = PC, y = Cumulative_Percentage), color='red', size=0.5) +
  labs(title="The Proportion of Eigenvalue for TOP 45 PCs") +
  xlab("Principle Components") +
  ylab("The Individual(Bar) & Cumulative(Line) Proportion")
```



Predictive Analysis

Data set is divided into 80%/20% for training and testing, respectively.

Regression Neural Network

For training the ANN model, three activation functions ('logistic', 'tanh' and 'relu') have been used. And for each function, it will try to fit in single and double hidden layer models. Packages 'doSnow' and 'foreach' have been used to parallelly train the models and reduce the running time.

Below is a single-layer ANN, with logistic Activation Function.

```
act <- c("logistic", "tanh", "relu")
k = 1
cores = 4

# parallelly run the code to shorten the training time
cl = registerDoSNOW(makeCluster(cores-1, type = "SOCK"))
SSE_list_log_1 <- foreach (i = 1:4,
                     .combine='rbind',
                     .packages=('neuralnet'),
                     .errorhandling = "remove") %dopar% {
                       set.seed(12321)
                       pca_ann <- neuralnet(V1 ~ .,
                                            data = pca_train,
                                            hidden = c(i),
                                            err.fct = "sse",
                                            act.fct = 'logistic',
                                            rep = 1)

                       # calculating the training and test SSE
                       train_sse <- sum((unlist(pca_ann$net.result) - pca_train[, 1])^2)/2

                       model_name <- paste(act[k], i,  sep = "-")
                       pca_test_output <- compute(pca_ann, pca_test[, 2:37])$net.result
                       test_sse <- sum((pca_test_output - pca_test[, 1])^2)/2
                       sse <- cbind(model_name,train_sse,test_sse)
                       return(sse)
                     }
```

Similar codes are applied to single and double layer models with all three activation functions.

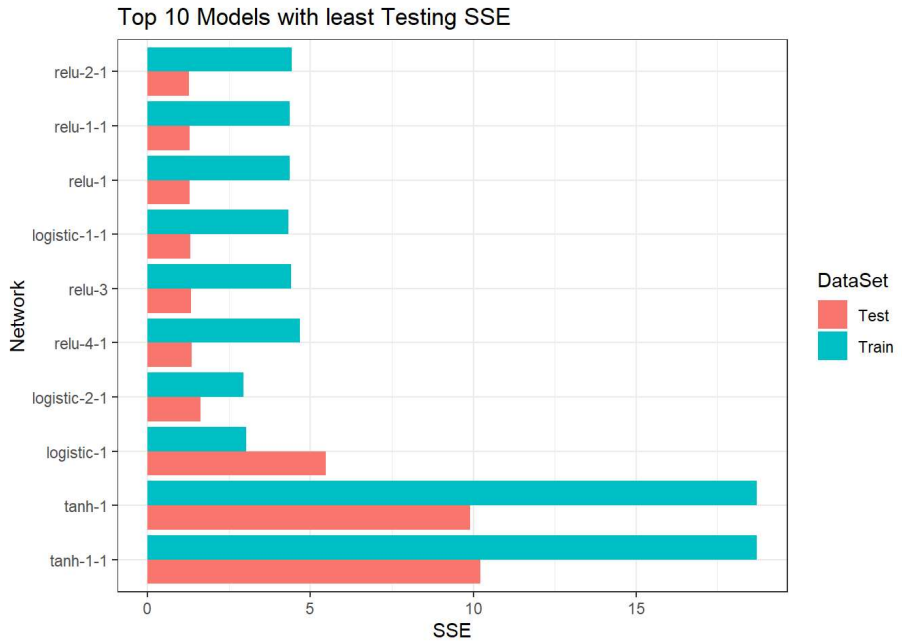
The Sum of Squared Error for each models

After the training for all the models, the sum of squared error for each model has been collected and integrated into the below table. For clearer visualization, a bar chart has been generated to compare the accuracy of each model.

```
SSE <- rbind(SSE_list_log_1, SSE_list_tan_1, SSE_list_relu_1, SSE_list_log_2, SSE_list_tan_2, SSE_list_relu_2)
SSE <- SSE[order(as.numeric(SSE[,3])),]

SSE_table <- SSE
SSE_table[,2:3] <- round(as.numeric(SSE_table[,2:3]),4)
SSE_table
```

```
##       model_name     train_sse test_sse
##  [1,] "relu-2-1"     "4.4175"  "1.2692"
##  [2,] "relu-1-1"     "4.3674"  "1.2845"
##  [3,] "relu-1"       "4.3636"  "1.2863"
##  [4,] "logistic-1-1" "4.3302"  "1.3114"
##  [5,] "relu-3"       "4.3985"  "1.3437"
##  [6,] "relu-4-1"     "4.6848"  "1.3551"
##  [7,] "logistic-2-1" "2.9513"  "1.622"
##  [8,] "logistic-1"   "3.0163"  "5.4701"
##  [9,] "tanh-1"       "18.7054" "9.9091"
## [10,] "tanh-1-1"     "18.7046" "10.2111"
## [11,] "relu-2"       "2.9685"  "27.0222"
## [12,] "tanh-4-1"     "16.2084" "33.2502"
## [13,] "tanh-2-1"     "15.8841" "34.4318"
## [14,] "tanh-3-1"     "10.0999" "61.6693"
## [15,] "tanh-2"       "15.8003" "70.1327"
## [16,] "tanh-4"       "15.3158" "127.8687"
```

```
tb <- tibble(Network = rep(SSE[1:10 ,1], time = 2),
       DataSet = rep(c("Train", "Test"), each = 10),
       SSE = c(round(as.numeric(SSE[1:10,2]), 2), round(as.numeric(SSE[1:10, 3]),2))
       )
tb$Network <- factor(tb$Network, levels = tb$Network[10:1])

ggplot(data = tb, aes(Network, SSE, fill = DataSet)) +
  geom_col(position = "dodge") + theme_bw() +
  ggtitle("Top 10 Models with least Testing SSE") +
  coord_flip()
```

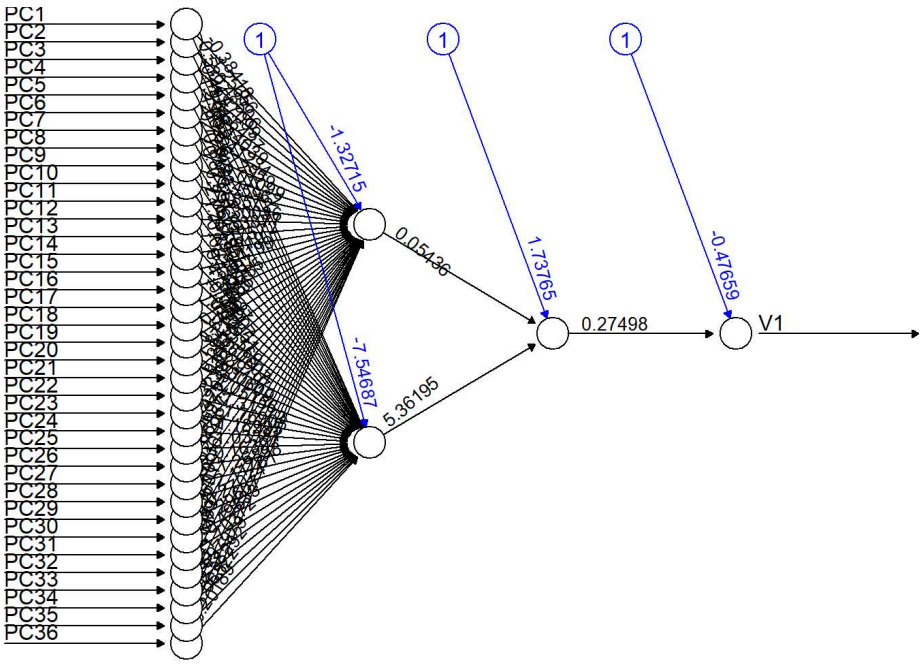


Conclusion For Regression Neural Network

As shown in the table and graph above, the 'relu' (rectified linear unit) activation function has the best performance on this data set for predicting the user rating count. Models with 'logistic' activation function seems to be a little over fitted as they have a decent performance for training data set but for testing, they do not have corresponding performance. Function 'tanh' is not quite suitable for this prediction with both its training SSE and testing SSE are significantly larger than that of the other two types of models. Among the 'relu' models, the top three have decent performances for both training and testing data set. However, the second and third have too few neurons in the hidden layers which may cause underfitting problem. Thus, the final model is decided to be relu-2-1.

Model 'relu-2-1' is shown below:

```
set.seed(12345)
Best_result <- neuralnet(V1 ~ .,
                         data = pca_train,
                         hidden = c(2,1),
                         err.fct = "sse",
                         act.fct = relu,
                         rep = 5)
#Select the best model from 5 replications
plot(Best_result, rep = "best")
```



Conclusion

The data from the Kaggle data base has been thoroughly analyzed using a lot of **Exploratory** and **Predictive** Analysis. Three different question were explored and successfully predicted by the group members using different **MACHINE LEARNING** techniques.