PREDICTIVE ANALYTICS

Northeastern University

ALY6020, SPRING 2020

MODULE 6 PROJECT ASSIGNMENT

WEEK 6: COLLABORATIVE PROJECT

SUBMITTED BY: SHIVANI ADSAR, VASU AMBASANA, SURYA TEJA DEVAKI, RONAK DUDUSKAR

NUID: 001399374, 001085863, 001060897, 001058327

SUBMITTED TO: PROF. NA YU

DATE: 05/11/2020

Introduction

The project assignment aims at providing practical experience of working on a dataset to perform various machine learning algorithms. The algorithms have been implemented on “Spotify Hit Predictor” dataset for performing predictions. Spotify is an International Media Service provider acts as a platform for media such as Music, Video, Podcasts and this dataset contain music records between 1960 to 2019. Our Goal is to understand the impact of various features for predicting “Hit” and “Flop” tracks which determines the success of the music album. The algorithms used are Naïve Bayes Classification, K-Nearest Neighbors, Logistic Regression, Decision Trees and Random Forest, which have helped in predicting the accuracy of our dataset.

Analysis

Spotify Hit Predictor Dataset

The Spotify Dataset has been taken from the website, Kaggle. This dataset has 41106 instances and 19 attributes. The data consists of collection of features of tracks fetched from Spotify. The tracks are labelled ‘1’ and ‘0’ (‘Hit’ or ‘Flop’) depending on some criteria. Moreover, the dataset has been cleaned to perform analysis. [4]

Naïve Bayes Algorithm

Naïve Bayes is a supervised machine learning algorithm which uses the Bayes Theorem by assuming a strong independence between the variables.

Importing the Dataset

- The Spotify dataset has been imported initially into a dataframe. Using the “str” function, the structure of the dataset has been analysed into 41106 observations and 19 variables.
- The “Category” is a character vector and is a categorical variable. Hence the variable has been factorized. [1]

```
> spotify_raw$Category <- factor(spotify_raw$Category)
> str(spotify_raw$Category)
 Factor w/ 2 levels "Flop","Hit": 2 1 1 1 1 1 1 2 1 2 ...
> table(spotify_raw$Category)

 Flop   Hit
20553 20553
```

Fig.1: Factorization of ‘Category’ Variable

We can observe that the “Category” variable has been factorized into “Flop” and “Hit”. So, there are 20553 Flop tracks and 20553 Hit tracks.

Analysis on Dataset

- The texts are collected and saved in a Corpus of documents and then the first 3 documents from the corpus are analyzed.

```
> spotify_corpus <- Corpus(VectorSource(spotify_raw$artist))
> print(spotify_corpus)
<<SimpleCorpus>>
Metadata:  corpus specific: 1, document level (indexed): 0
Content:  documents: 41106
> inspect(spotify_corpus[1:3])
<<SimpleCorpus>>
Metadata:  corpus specific: 1, document level (indexed): 0
Content:  documents: 3

[1] Garland Green     Serge Gainsbourg Lord Melody
```

Fig.2: Corpus of the text messages

We have created a corpus of tracks and inspected the first three documents. Corpus is a collection of documents which makes it easier for document retrieval. The corpus contains a collection of 41106 documents.

- We have used the "tm" package which is used for performing text mining. The data is cleaned by using the "tm_map" function which transforms the tm_corpus function.

```
install.packages("tm")
library(tm)
spotify_corpus <- Corpus(VectorSource(spotify_raw$artist))
print(spotify_corpus)
inspect(spotify_corpus[1:3])
corpus_clean <- tm_map(spotify_corpus, tolower)
corpus_clean <- tm_map(corpus_clean, removeNumbers)
corpus_clean <- tm_map(corpus_clean, removeWords, stopwords())
corpus_clean <- tm_map(corpus_clean, removePunctuation)
corpus_clean <- tm_map(corpus_clean, stripWhitespace)
corpus_clean
nrow(spotify_raw)
spotify_dtm <- DocumentTermMatrix(corpus_clean)
```

Fig.3: Data Cleaning

We can see that the data has been cleaned to remove some stop words such as "and", "to", "an"etc., the data has been converted to lower case, numbers and punctuations have been removed.

- Using the process of "tokenization", the messages have been split into individual tokens or words. We have used the DocumentTermMatrix() function which takes the cleaned corpus as the input parameter and creates a matrix. This matrix shows the frequency of words occurring in every document. [2]

```
> spotify_dtm <- DocumentTermMatrix(corpus_clean)
```

The DocumentTermMatrix() will convert the corpus into tokens and return a sparse matrix, which will be used for performing further analysis.

Splitting into Training and Testing Dataset

- The data is split into 70 % training and 30% testing datasets, where we train and build our model on the training dataset and validate using the test dataset.[1]
- First, we will split the raw dataset:

```
> spotify_raw_train <- spotify_raw[1:30000, ]
> spotify_raw_test  <- spotify_raw[30001:41106, ]
```

Fig.4: Splitting of raw data

- We have split the document term matrix

```
> spotify_dtm_train <- spotify_dtm[1:30000, ]
> spotify_dtm_test  <- spotify_dtm[30001:41106, ]
```

Fig.5: Splitting of Document Term Matrix

We can see that the first 30000 of the data is used for training and the rest of 11105 data is used for testing.

- Further, We have split the corpus into training and testing datasets

```
> spotify_corpus_train <- corpus_clean[1:30000]
> spotify_corpus_test  <- corpus_clean[30001:41106]
```

Fig.6: Splitting of Corpus

- In order to view the proportion of Hit and Flop data in the training and testing data, we have used the prop.table() function.[1]

```
> prop.table(table(spotify_raw_train$Category))

     Flop       Hit 
0.4997667 0.5002333 
> prop.table(table(spotify_raw_test$Category))

     Flop       Hit 
0.5006303 0.4993697 
```

Fig.7: Proportion of Hit and Flop

We can observe that the data in both the testing and training is almost the same with Flop tracks being approximately 50%, hence the data is split equally.

- We would be using the sparse matrix to train our model for Naïve Bayes classification algorithm. Since, all the features from the sparse matrix will not be considered for classification, will use the findFreqTerms() which will be used for finding the frequent occurring terms in the document. The function takes the matrix as input and then returns a character vector. The results of the function are stored in a dictionary.

```
> findFreqTerms(spotify_dtm_train, 5)
  [1] "garland"    "green"      "gainsbourg" "serge"      "lord"       "melody"    
  [7] "celia"      "cruz"       "susheela"   "ennio"      "morricone"  "ant\xe3�nio"
 [13] "carlos"     "jobim"      "johnson"    "marv"       "caetano"    "veloso"    
 [19] "beach"      "boys"       "goldsmith"  "jerry"      "orbison"    "roy"       
 [25] "gonzaga"    "luiz"       "dean"       "jimmy"      "mary"       "wells"     
> spotify_dict <- c(findFreqTerms(spotify_dtm_train, 5))
> spotify_train <- DocumentTermMatrix(spotify_corpus_train,    list(dictionary = spotify_dict))
> spotify_test  <- DocumentTermMatrix(spotify_corpus_test,    list(dictionary = spotify_dict))
```

Fig.8: Frequency Matrix

Naïve Bayes Classifier

- The Naïve Bayes classifier works well with categorical values. We can see that the values in the matrix are character vector values, hence we need to convert them into factor values.

```
> convert_counts <- function(x) {
+   x <- ifelse(x > 0, 1, 0)
+   x <- factor(x, levels = c(0, 1), labels = c("No", "Yes"))
+ }
```

Fig. 9: Conversion of vector into factor values

We can observe that, for factorization, we have converted the values of x, if greater than 0, it will be replaced with 1, else will remain 0. So, the labels of "No" and "Yes" have been given accordingly.

- Further, we have used the apply() function to convert the counts of train and test data for factorization, using Margin = 2 as we are selecting the columns.

```
> spotify_train <- apply(spotify_train, MARGIN = 2, convert_counts)
> spotify_test  <- apply(spotify_test, MARGIN = 2, convert_counts)
```

Fig.10: Converting into factors for Train and Test samples

- For performing the Naïve Bayes classifier, we need to install the "e1071" CRAN packages.

```
> spotify_classifier <- naiveBayes(spotify_train, spotify_raw_train$Category)
> spotify_test_pred <- predict(spotify_classifier, spotify_test)
```

Fig.11: Naïve Bayes Classifier

- We have built the Naïve Bayes classifier on the model and used predict() function for predictions. Since we need to evaluate our predictions, we will be comparing with the unseen data which has been stored in spotify_test, whereas spotify_classifier is our trained classifer.
- Now, we will be comparing the predicted values with actual values by using the CrossTable() function which is available in the CRAN gmodels package, used for model fitting.

```
install.packages("gmodels")
library(gmodels)
CrossTable(spotify_test_pred, spotify_raw_test$Category, prop.chisq = FALSE, prop.t = FALSE,    dnn = c('predicted', 'actual'))
```

Fig.16: Cross Table function

```
Total Observations in Table:  11106

             | actual
   predicted |      Flop |       Hit | Row Total |
-------------|-----------|-----------|-----------|
        Flop |      1151 |       905 |      2056 |
             |     0.560 |     0.440 |     0.185 |
             |     0.207 |     0.163 |           |
-------------|-----------|-----------|-----------|
         Hit |      4409 |      4641 |      9050 |
             |     0.487 |     0.513 |     0.815 |
             |     0.793 |     0.837 |           |
-------------|-----------|-----------|-----------|
Column Total |      5560 |      5546 |     11106 |
             |     0.501 |     0.499 |           |
-------------|-----------|-----------|-----------|
```

Fig. 12: Cross Table Output

We can observe that, out of a total of 5560 Flop tracks, 4409 Flop tracks were incorrectly classified as Hit. Whereas, 905 out of a total of 5546 Hit tracks were incorrectly classified as Flop. Also, 1151 Flop tracks and 4641 Hit tracks were correctly classified. The accuracy by considering True Negatives and True Positives is 52.15%.

- We can observe that the 4409 Flop tracks were incorrectly classified as Hit, this could be a major problem as the classifier will predict Flop tracks to be Hit. Hence we need to improve the performance because sometimes the classifier classifies a particular word to be a Flop even if it occurred once.
- We will build a Naïve Bayes model and set a laplace =1.

```
> spotify_classifier2 <- naiveBayes(spotify_train, spotify_raw_train$Category,    laplace = 1)
> spotify_test_pred2 <- predict(spotify_classifier2, spotify_test)
> CrossTable(spotify_test_pred2, spotify_raw_test$Category,    prop.chisq = FALSE, prop.t = FALSE, prop.r = FALSE,    dnn = c('predi
cted', 'actual'))
```

Fig.18: Naïve Bayes Classifier with Laplace

```
Total Observations in Table:  11106

             | actual
   predicted |      Flop |       Hit | Row Total |
-------------|-----------|-----------|-----------|
        Flop |      1340 |       445 |      1785 |
             |     0.241 |     0.080 |           |
-------------|-----------|-----------|-----------|
         Hit |      4220 |      5101 |      9321 |
             |     0.759 |     0.920 |           |
-------------|-----------|-----------|-----------|
Column Total |      5560 |      5546 |     11106 |
             |     0.501 |     0.499 |           |
-------------|-----------|-----------|-----------|
```

Fig.13: Cross Table Output of Laplace

We can observe a small improvement in the classification of False Negatives from 4409 to 4220 and False Positives from 905 to 445. The accuracy has improved to 58%.

- We have observed that the Naïve Bayes classifier is an efficient technique for classification of Hit and Flop tracks with an accuracy of 58 %. However, this algorithm is not suitable for the Spotify dataset as the accuracy is 58%. Hence, the dataset can give a better accuracy with a different machine learning algorithm.

Logistic Regression

Logistic Regression is a supervised machine learning regression algorithm used for binary classification. It is a method used for fitting the categorical variables in the *y=f(x)* curve.[6]

Importing the Data

- The Spotify dataset has been imported initially into a dataframe. Using the "str" function, the structure of the dataset has been analysed into 41106 observations and 19 variables.

```
> str(spotify)
'data.frame':	41106 obs. of  19 variables:
 $ track           : Factor w/ 35860 levels "''Bang Bang'' (My Baby Shot Me Down)",..: 15698 15092 19548 19773 32929 3185 25778 1172
1 5105 13650 ...
 $ artist          : Factor w/ 11904 levels "'In The Heights' Original Broadway Company",..: 3846 9091 6309 1857 7829 3287 556 6769
 1673 9993 ...
 $ uri             : Factor w/ 40560 levels "spotify:track:000JBwAOq5d9lNNSnovPYg",..: 6704 28137 35593 37412 9112 15889 31479 1564
 5706 7300 ...
 $ danceability    : num  0.417 0.498 0.657 0.59 0.515 0.697 0.662 0.72 0.545 0.511 ...
 $ energy          : num  0.62 0.505 0.649 0.545 0.765 0.673 0.272 0.624 0.22 0.603 ...
 $ key             : int  3 3 5 7 11 0 0 5 2 2 ...
 $ loudness        : num  -7.73 -12.47 -13.39 -12.06 -3.52 ...
 $ mode            : int  1 1 1 0 0 1 1 0 0 1 ...
 $ speechiness     : num  0.0403 0.0337 0.038 0.104 0.124 0.0266 0.0313 0.0473 0.0828 0.028 ...
 $ acousticness    : num  0.49 0.018 0.846 0.706 0.857 0.714 0.36 0.795 0.582 0.0385 ...
 $ instrumentalness: num  0.00 1.07e-01 4.42e-06 2.46e-02 8.72e-04 9.19e-01 2.28e-01 0.00 2.39e-01 1.67e-06 ...
 $ liveness        : num  0.0779 0.176 0.119 0.061 0.213 0.122 0.0963 0.488 0.269 0.142 ...
 $ valence         : num  0.845 0.797 0.908 0.967 0.906 0.778 0.591 0.887 0.386 0.685 ...
 $ tempo           : num  186 102 116 106 115 ...
 $ duration_ms     : int  173533 213613 223960 157907 245600 167667 134360 160040 158413 157293 ...
 $ time_signature  : int  3 4 4 4 4 4 4 4 4 4 ...
```

Fig.14: Structure of 'Spotify' dataset

The "str" function is used to view the structure of the dataset for understanding the variables for further analysis.

Check the Class Bias

- We have checked the class bias to view the proportion of 1's and 0's in our target variable. [2]

```
> #Check class bias
> table(spotify$target)

    0     1 
20553 20553 
```

Fig.15: Checking the Class Bias

As observed, there is a class bias with 0 having 20553 values and 1 having 20553 values. Hence, there is no biased observed amongst the target variables.

Creating Testing and Training Samples

- The data is split into 80 % training and 20% testing datasets, where we train and build our model on the training dataset and validate using the test dataset.[5]
- The set.seed(9999) function takes 9999 random samples and repeats them in a sequence.

```
> set.seed(9999)
> index<-spotify[sample(nrow(spotify),20000),]
> partioned.spotify<-createDataPartition(
+   index$target,
+   times=1,
+   p=0.8,
+   list=F
+ )
> spotify_training=index[partioned.spotify,]
> spotify_test=index[-partioned.spotify,]
```

We have used the createDataPartition() function that splits the data into testing and training samples. We have used 20000 of the dataset's rows for analysis. This data has been

Fig.16: Allocation of Training and Testing Sample

Information Values

- We have used the 'smbinning' package which performs score modeling and discretization, ie. conversion of continuous variables into bins. This gives a better understanding of the distribution with a binary variable.
- The continuous and factor variables are segregated and then combined in a dataframe.

```
> library(smbinning)
> factor_vars <- c ("target","mode")
> continuous_vars <- c("danceability","energy","key","loudness","speechiness","acousticness","instrumentalness","liveness","valenc
e","duration_ms","time_signature","chorus_hit")
> iv_df <- data.frame(VARS=c(factor_vars, continuous_vars), IV=numeric(14))  # init for IV results
```

Fig.17: Segregation of Continuous and Factor variables

- Further, we have computed the information values for categorical variables.
  This has been performed for categorization of hit and flop tracks.
  Moreover we have used the WOE, Weight of Evidence and IV, Information Values as they help in performing exploratory analysis for binary classifiers. They help to establish linear and non-linear relationships amongst variables. Thus, finding correlations between the dependent and independent variables. [2]

```
> for(factor_var in factor_vars){
+   smb <- smbinning.factor(spotify_training, y="target", x=factor_var)  # WOE table
+   if(class(smb) != "character"){ # heck if some error occured
+     iv_df[iv_df$VARS == factor_var, "IV"] <- smb$iv
+   }
+ }
```

We have created the WOE, 'Weight of Evidence' table that transforms the factors into binary classifications by calculation of weights.

Fig.18: Information Value for Categorical Variables

- Similarly, we have calculated the information values for continuous variables.

```
> # compute IV for continuous vars
> for(continuous_var in continuous_vars){
+   smb <- smbinning(spotify_training, y="target", x=continuous_var)  # WOE table
+  if(class(smb) != "character"){  # any error while calculating scores.
+     iv_df[iv_df$VARS == continuous_var, "IV"] <- smb$iv
+   }
+ }
```

Fig.19: Information Value for Continuous Variables

- We have performed sorting of the information values by using the order() function, in the decreasing order.

```
> iv_df <- iv_df[order(-iv_df$IV), ]  # sort
> iv_df
               VARS     IV
9  instrumentalness 1.1096
8      acousticness 0.7759
4            energy 0.5936
3      danceability 0.5859
6          loudness 0.4838
11          valence 0.4506
12      duration_ms 0.3674
7       speechiness 0.1208
13   time_signature 0.0706
10         liveness 0.0356
14       chorus_hit 0.0267
1            target 0.0000
2              mode 0.0000
5               key 0.0000
```

Fig.20: Sorted Information Values

As observed, the Information Values for variables like, instrumentalness, acousticness, energy, danceability etc. is higher. So, instrumentalness has the highest Information Value of 1.109 and chorus_hit has the lowest value. Also, these variables would be significant for our predictions.

<u>Building Logit Models and Prediction</u>

- We have used the glm() function with family='binomial' to build the logistic regression model in our training data. The left side of the '~' sign shows our independent variable, target and the variables on the right side of the sign show the dependent variables, like instrumentalness, loudness, liveness, acousticness, valence, and speechiness. [2]

```
> logitMod <- glm(target ~ instrumentalness +loudness +liveness+acousticness+valence+speechiness, data=spotify_training, family=binomial(link="logit"))
```

Fig.21: Logistic Model using glm()

- We have used the predict function which is used for predicting the logs of the target variable.

```
> predicted <- predict(logitMod, spotify_test, type="response")  # predicted scores
```

Fig.22: Prediction on model

The predict() function gives us the values between 0 and 1.

- In order to perform optimization on our model, we have used the optimal prediction cutoff. The cutoff score is 0.5 for models by default, for improving the accuracy in training and testing samples, we have used the optimalcutoff() function. This function provides optimal cutoff by improving prediction of 1's and 0's and minimize misclassification errors.

- We have installed ‘InformationValue’ library for performing analysis on performance of model and binary classifications.

```
> library(InformationValue)
> optcutoff <- optimalCutoff(spotify_test$target, predicted)[1]
> optcutoff
[1] 0.453365
```

Fig.23: Optimisation on model

The optimal cutoff is seen to be 0.45 for our model.

Model Diagnostics

- Now, we have analysed the diagnostic using the summary() function.

```
> summary(logitMod)

Call:
glm(formula = target ~ instrumentalness + loudness + liveness +
    acousticness + valence + speechiness, family = binomial(link = "logit"),
    data = spotify_training)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.9381  -1.0598  -0.1091   0.9300   3.0652

Coefficients:
                 Estimate Std. Error z value             Pr(>|z|)
(Intercept)       0.97692    0.07069  13.820 <0.0000000000000002 ***
instrumentalness -3.80179    0.11086 -34.292 <0.0000000000000002 ***
loudness          0.05004    0.00491  10.193 <0.0000000000000002 ***
liveness         -0.89742    0.10494  -8.552 <0.0000000000000002 ***
acousticness     -0.97821    0.06623 -14.771 <0.0000000000000002 ***
valence           1.06085    0.07294  14.544 <0.0000000000000002 ***
speechiness      -2.21035    0.22275  -9.923 <0.0000000000000002 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 22180  on 15999  degrees of freedom
Residual deviance: 17823  on 15993  degrees of freedom
AIC: 17837

Number of Fisher Scoring iterations: 5
```

We can interpret the beta coefficients, standard error, z-value and p-value. We can view entries for each category as the glm() function considers each category to be independent binary variable. The AIC is 17837 and fisher iterations are 5.

Fig.24: Summary of logitMod

- We need to view the misclassification error which shows the percentage mismatch between actual and predicted variables.

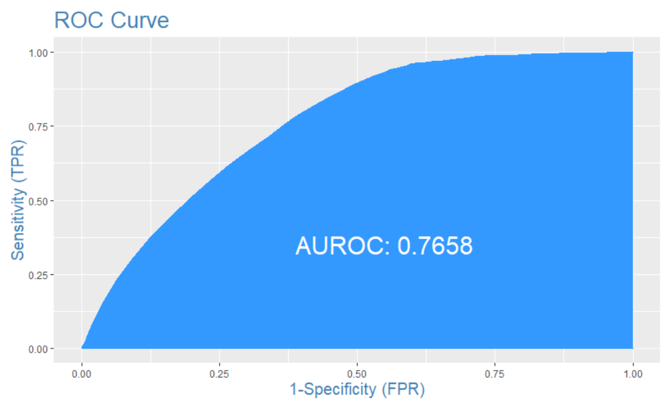
```
> misClassError(spotify_test$target, predicted, threshold = optcutoff)
[1] 0.2982
```

Fig.25: Misclassification error

We can see that, our model has 0.29 error, lower the error, better is the prediction.

- Further, we have plotted the ROC Curve, Receiver Operating Characteristics Curve which shows the diagnostics of a binary classifier by plotting the true positives against false positives.[3]
- In order for a ROC curve to be good for a model, the curve should rise steeply and should have greater area under the curve as the cutoff is lowered, by indicating True Positive Rate increases faster than False Positive Rate.

```
> plotROC(spotify_test$target, predicted)
```



It can be seen that area under the ROC curve is 76.58% which is good for our model.

Fig.26: ROC Curve

- We have computed the Concordance for our model, which gives the percentage of number of actual positive's (1's) in comparison to the negative's (0's), if the positive's are more than the negative's, the model is said to be concordant. Higher the concordance, better is the model. [2]

```
> Concordance(spotify_test$target, predicted)
$Concordance
[1] 0.7658524
```

The concordance in our model, is 76.58%, which is good for our model.

Fig.27: Concordance

- The Sensitivity is the percentage of 1's or positive's that are correctly predicted by our model. While, the Specificity is the percentage of 0's or Negative's correctly predicted by our model.

```
> # Sensitivity
> sensitivity(spotify_test$target, predicted, threshold = optcutoff)
[1] 0.87333
> #Specificity
> specificity(spotify_test$target, predicted, threshold = optcutoff)
[1] 0.5265285
```

Fig.28: Sensitivity and Specificity

As seen, the Sensitivity is 87.33% and Specificity is 52.65%.

- Confusion Matrix shows the performance of a classification model.

```
> #Confusion Matrix
> confusionMatrix(spotify_test$target, predicted, threshold = optcutoff)
     0    1
0 1042  256
1  937 1765
```

Fig.29: Confusion Matrix

- The True Negatives ie. 1042 show we predicted No, and it was a Flop track. While, False Positives is 256 as they were predicted to be Hit tracks but actually were Flop tracks. Moreover, we predicted 937 as False Negatives and those tracks were Hit tracks. In addition, 1765 were True Positives, as their cases were predicted correctly as Hit tracks. The accuracy of the model is predicted to be 70.17%.

- After performing analysis on the Spotify's dataset for predicting Spotify's Hit and Flop tracks using Logistic Regression, it was noted that the accuracy of the model is highest as 76.58% .

<u>Decision Tree Algorithm</u>

Decision Tree is a classification concept in Machine Learning that applies similar strategy to **"Divide and Conquer"** or **"Recursive partitioning"** the data into smaller subsets and then identifying patterns later that can be used for prediction. The data is then represented logical structures enabling it to be understood by people without statistical knowledge.

<u>Importing the Data</u>

- First step here will be getting our dataset ready to work upon. To do that we'll import the data using **read.csv()**. This will look like follows :-

```
> Spotify <- read.csv("Spotify(1960-2019)Hits_Prediction.csv", stringsAsFactors = FALSE)
```

- As we've seen earlier, our dataset "Spotify" contains Track, Artist and URI information, we don't need that for the making of our Decision tree, so we'll simply remove those columns for the sake of Decision Trees.

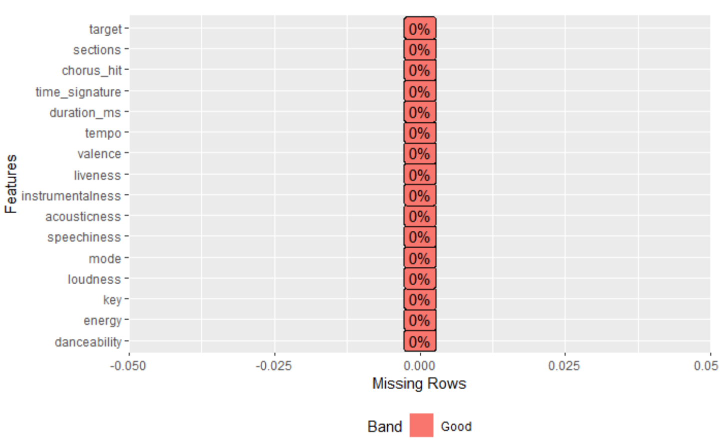
```
> Spotify <- Spotify[,-3]
> Spotify <- Spotify[, -2]
> Spotify <- Spotify[,-1]
```

These lines of code will remove first three columns from "Spotify".

Fig. 30: Removal of unnecessary columns

- To check for missing values in our Dataset, we used **plot_missing()** function which is provided by the package **DataExplorer.**

```
> plot_missing(Spotify)
```



The plot shows the total proportion of missing values in all of the Features of our Dataset.

Our Dataset seems to be cleaned as we can see 0% data is missing in all the columns.

Fig.31: Data Cleaning

We can look at the proportion of Flops and Hits i.e. "0"s and "1"s in our Dataset :

```
> table(Spotify$target)

    0     1 
20553 20553 
```

Fig.32: Proportion of Flops and Hits

Our target variable shows that there are equal number of Flops as Hits.

Creating Training and Testing Data

- We'll select values from the original Dataframe to create Training and Test Datasets. We selected 35000 values for the training dataset and the remaining 6106 in test dataset which can be used later to validate our model.

```
> decision_train <- Spotify[1:35000, ]
> decision_test <- Spotify[35001:41106, ]
> prop.table(table(decision_train$target))

        0         1 
0.4999143 0.5000857 
> prop.table(table(decision_test$target))

        0         1 
0.5004913 0.4995087 
```

The variables are then loaded into the global environment.

Prop.table() will show the proportion of 0 and 1 in both training and test datasets.

Fig.33: Proportions of Train and Test Samples

- The proportion of the target data in both Training and Test Dataset is always checked in order to avoid Biased performance of our Decision Tree Model.

Model Building

- We will now use the C5.0 algorithm provided by the package "c50". Install the package with install.packages("C50") and can be loaded into the R session by using library(c50).
- For the first iteration of our Hits predictor model, we'll use the default C5.0 configuration as shown in the following code. The 16th column is the class variable, target, so we have to exclude it from the training data frame as an independent variable, but we'll supply it as the target factor vector for classification.

```
> Spotify_decision_model <- C5.0(decision_train[-16], factor(decision_train$target))
> Spotify_decision_model

Call:
C5.0.default(x = decision_train[-16], y = factor(decision_train$target))

Classification Tree
Number of samples: 35000 
Number of predictors: 15 

Tree size: 390 
```

Fig.34: Decision Model

Our Spotify_decision_model object now contains a C5.0 configuration with decision tree object. We can see some basic data about the tree in the output.

- Our classifier contains :

decision_train as the data frame containing training data and the factor vector decision_train$target with the class for each row in the training data

- The above output shows our Decision Tree is 390 decisions deep!! To see all the decisions , we can use the summary() function on the model :

```
> summary(Spotify_decision_model)

Call:
C5.0.default(x = decision_train[-16], y = factor(decision_train$target))


C5.0 [Release 2.07 GPL Edition]         Sun May 10 15:42:38 2020
-------------------------------

Class specified by attribute `outcome'

Read 35000 cases (16 attributes) from undefined.data

Decision tree:

instrumentalness > 0.091:
:...danceability > 0.556:
:   :...duration_ms <= 102121: 0 (128)
:   :   duration_ms > 102121:
:   :   :...instrumentalness <= 0.593:
:   :       :...acousticness > 0.734:
:   :       :   :...valence <= 0.829: 0 (149/18)
:   :       :   :   valence > 0.829:
:   :       :   :   :...speechiness <= 0.0568: 1 (14/5)
:   :       :   :       speechiness > 0.0568: 0 (8)
:   :       :   acousticness <= 0.734:
:   :       :   :...danceability > 0.692:
:   :       :       :...instrumentalness > 0.485: 0 (65/26)
:   :       :       :   instrumentalness <= 0.485:
:   :       :       :   :...loudness <= -14.493:
:   :       :       :       :...speechiness > 0.0444: 0 (34/8)
:   :       :       :       :   speechiness <= 0.0444:
:   :       :       :       :   :...tempo <= 126.316: 1 (13)
:   :       :       :       :       tempo > 126.316: 0 (5/1)
:   :       :       :       loudness > -14.493:
:   :       :       :       :...mode > 0: 1 (181/48)
:   :       :       :           mode <= 0:
:   :       :       :           :...valence > 0.572: 1 (108/28)
:   :       :       :               valence <= 0.572:
:   :       :       :               :...tempo <= 97.587: 1 (6/1)
:   :       :       :                   tempo > 97.587: 0 (24/7)
:   :       :       danceability <= 0.692:
:   :       :       :...valence <= 0.666: 0 (239/75)
:   :       :           valence > 0.666:
:   :       :           :...sections <= 8: 0 (78/27)
:   :       :               sections > 8:
:   :       :               :...energy <= 0.516:
:   :       :                   :...chorus_hit > 36.62034: 0 (16)
:   :       :                   :   chorus_hit <= 36.62034:
:   :       :                   :   :...liveness > 0.204: 0 (5)
:   :       :                   :       liveness <= 0.204:
:   :       :                   :       :...sections <= 15: 1 (12/2)
:   :       :                   :           sections > 15: 0 (2)
:   :       :                   energy > 0.516:
:   :       :                   :...mode > 0: 1 (129/56)
```

Fig.35: Decision Rules

We have 35,000 observations in our training data and we know that our decision tree is 390 decisions deep. It may take some time to run of low memory machines.

- Our evaluation on training data of 35,000 cases shows there is 19.3% error rate in classifying.
- The attribute usage section shows percentage of data used from that column.
- The summary function also displays the confusion matrix for the model, which helps in understanding the incorrectly classified fields in training data.

```
Evaluation on training data (35000 cases):

            Decision Tree
          ----------------
          Size      Errors

           390 6759(19.3%)   <<


           (a)   (b)    <-classified as
          ----  ----
         13372  4125    (a): class 0
          2634 14869    (b): class 1


        Attribute usage:

        100.00% instrumentalness
         86.53% duration_ms
         79.26% acousticness
         76.81% danceability
         76.74% loudness
         72.46% energy
         55.77% time_signature
         45.99% speechiness
         41.92% mode
         34.42% sections
         30.07% valence
          7.87% liveness
          7.47% chorus_hit
          5.39% tempo
          4.07% key
```

Fig.36: Decision Tree for Training Data

- Similarly, we can plot the decision tree using plot() function :

```
> plot(Spotify_decision_model)
```

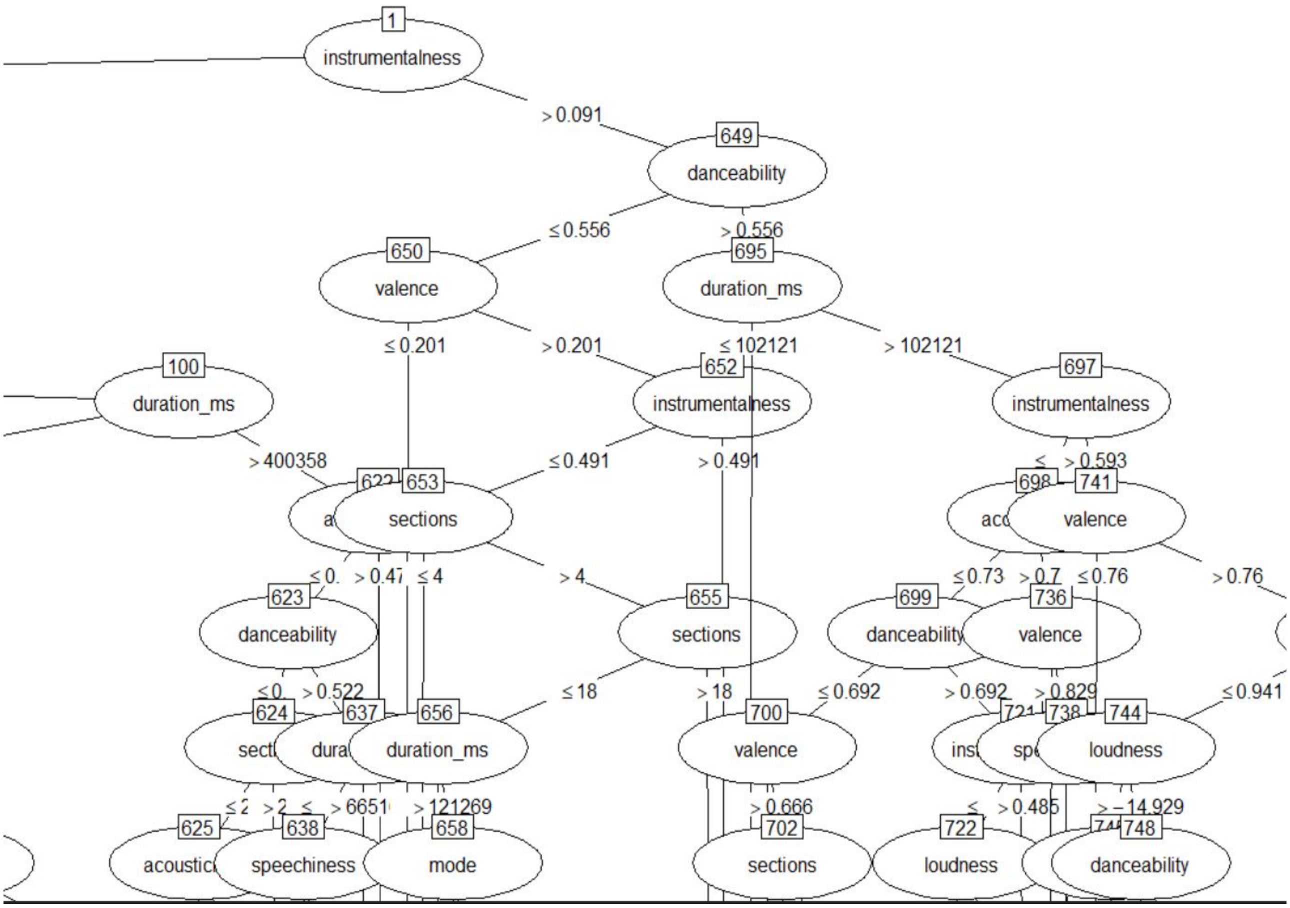


Fig.:37: Decision Model

- Since, decision trees have the tendency to overfit the model and that is why we need to evaluate decision trees on training and testing data.
- The figure that we will get after running the plot command will not be completely visible as it contains 35,000 observations.

<u>Model Building</u>

- We have applied our decision tree to the test dataset, predict() function provided by caret package in R that will help us do so and will create a vector of predicted class values which can be used later in Cross Table evaluation .
- To use CrossTable() function, we need to install gmodels package.

```
> decision_pred <- predict(Spotify_decision_model, decision_test)
> CrossTable(decision_test$target, decision_pred,
+            prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
+            dnn = c('Actual Target','Predicted Target'))

   Cell Contents
|-------------------------|
|                       N |
|         N / Table Total |
|-------------------------|

Total Observations in Table:  6106

              | Predicted Target
Actual Target |         0 |         1 | Row Total |
--------------|-----------|-----------|-----------|
            0 |      2024 |      1032 |      3056 |
              |     0.331 |     0.169 |           |
--------------|-----------|-----------|-----------|
            1 |       378 |      2672 |      3050 |
              |     0.062 |     0.438 |           |
--------------|-----------|-----------|-----------|
 Column Total |      2402 |      3704 |      6106 |
--------------|-----------|-----------|-----------|
```

We can observe that the True Positives, 2024 were correctly classified out of 2402 values.

The True Negatives of 2672 were correctly classified as 3704.

However, the False Positives were 1032 and False Negatives were 378 out of a total of 3704 and 2402 respectively. The accuracy is

Fig.38: Confusion Matrix

- The above function creates a vector of predicted values which are further compared with the actual values. The proportions, prop.c and prop.r has been set to “False” to remove row and column percentages from the table.
- The accuracy of our model is 2024 + 2672 = 4,696 ÷ 6106, which will be 76.90% .

<u>Adaptive Boosting</u>

- A way to improve the C5.0 algorithm’s accuracy is by introducing the adaptive boosting in which maximum decision trees are built and best class is selected.
- We have used “trials” which will use different decision trees in the algorithm. If the trials stop improving the accuracy, the algorithm will stop adding trials.

```
> decision_boost <- C5.0(decision_train[-16],factor(decision_train$target), trials = 10)
  decision_boost

Call:
C5.0.default(x = decision_train[-16], y = factor(decision_train$target), trials = 10)

Classification Tree
Number of samples: 35000
Number of predictors: 15

Number of boosting iterations: 10
Average tree size: 238.9
```

Fig.39: Adaptive Boosting

```
Evaluation on training data (35000 cases):

Trial        Decision Tree
-----      ----------------
           Size      Errors

   0        390  6759(19.3%)
   1        158  8044(23.0%)
   2        200  8644(24.7%)
   3        173  8702(24.9%)
   4        181  9585(27.4%)
   5        182 10124(28.9%)
   6        223  9018(25.8%)
   7        249  8341(23.8%)
   8        313  7648(21.9%)
   9        320  7134(20.4%)
boost            4994(14.3%)   <<


            (a)   (b)    <-classified as
           ----  ----
          14286  3211    (a): class 0
           1783 15720    (b): class 1


        Attribute usage:

        100.00% instrumentalness
        100.00% valence
        100.00% duration_ms
        100.00% chorus_hit
         99.05% acousticness
         98.13% danceability
         96.80% loudness
         93.88% sections
         89.47% tempo
         88.33% speechiness
         88.21% energy
         86.49% time_signature
         81.33% mode
         74.43% liveness
         73.71% key


Time: 7.4 secs
```

After Boosting, our number of trees reduced to 239 and our 10 number of boosting trials gave 5% of better accuracy i.e. of 14.3% on 10th trial.

Also, the attribute usage is more now on the training dataset after we performed boosting.

Fig.40: Decision Rule for Adaptive Boosting

- Applying our Boosted model on Test Dataset :

```
> decision_boost_pred <- predict(decision_boost, decision_test)
> CrossTable(decision_test$target, decision_boost_pred,
+            prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
+            dnn = c('Actual Target','Predicted Target'))


   Cell Contents
|-------------------------|
|                       N |
|         N / Table Total |
|-------------------------|


Total Observations in Table:  6106


              | Predicted Target
Actual Target |         0 |         1 | Row Total |
--------------|-----------|-----------|-----------|
            0 |      2067 |       989 |      3056 |
              |     0.339 |     0.162 |           |
--------------|-----------|-----------|-----------|
            1 |       337 |      2713 |      3050 |
              |     0.055 |     0.444 |           |
--------------|-----------|-----------|-----------|
 Column Total |      2404 |      3702 |      6106 |
--------------|-----------|-----------|-----------|
```

We can see that our model's accuracy is 78.28% after performing adaptive boosting.

Fig.41: Confusion Matrix for Adaptive Boosted Model

- The accuracy of our model can be calculated by adding 2067 and 2713 divided by total number of observations i.e. 6106. We got 78.28% which is a little improvement on the previous model without boosting.

Cost Matrix

• The C5.0 algorithm allows us to assign penalties to different types of errors in order to guide tree not to make costly errors. The penalties are designated in a matrix known as Cost Matrix :

```
> error_cost <- matrix(c(0,2,1,0), nrow = 2)
> error_cost
     [,1] [,2]
[1,]    0    1
[2,]    2    0
```

Fig.42: Error Cost Matrix

```
> decision_cost_pred <- predict(decision_cost, decision_test)
> CrossTable(decision_test$target, decision_cost_pred,
+            prop.chisq = FALSE, prop.r = FALSE, prop.c = FALSE,
+            dnn = c('Actual Target','Predicted Target'))


   Cell Contents
|-------------------------|
|                       N |
|         N / Table Total |
|-------------------------|


Total Observations in Table:  6106


              | Predicted Target
Actual Target |         0 |         1 | Row Total |
--------------|-----------|-----------|-----------|
            0 |      2397 |       659 |      3056 |
              |     0.393 |     0.108 |           |
--------------|-----------|-----------|-----------|
            1 |       942 |      2108 |      3050 |
              |     0.154 |     0.345 |           |
--------------|-----------|-----------|-----------|
 Column Total |      3339 |      2767 |      6106 |
--------------|-----------|-----------|-----------|
```

Note: The 1 shows Hits and 2 indicates Flops defaulted value. The rows show predicted values and columns indicate actual values. It can be seen that “false negative” has a cost of 2 while “false positive” has a cost of 1.

Fig.43: Confusion Matrix

- It can be observed that this model predicts more incorrect results of 32% than the boosting algorithm that predicted 23%. This model has incorrectly classified only 25% of defaults, whereas the previous algorithms wrongly classified 50% of defaults incorrectly.

Random Forest Algorithm

Random Forest is considered as one of the most important Machine Learning Algorithm that is used in Classification and Regression of Data. The idea behind Random Forest is that it generates multiple small decision trees from a random subset from the original data. Then it combines the result and outputs the one with most number of individual votes by the trees.

Importing the Dataset

- The library “randomForest” allows us to use Random Forest Algorithm in R. This package is used for classification and regression analysis.
- The library “MASS” contains the Dataframe that we are taking into consideration.

Then we’ll load the dataframe using read.csv() function.

```
> R_Spotify <- read.csv("Spotify(1960-2019)Hits_Prediction.csv", stringsAsFactors = FALSE)
> R_Spotify <- R_Spotify[,-3]
> R_Spotify <- R_Spotify[,-2]
> R_Spotify <- R_Spotify[,-1]
```

Fig.44: Cleaning Dataset

- The next three lines of code will remove the first three columns from the dataset which are Track, Artist and URI column. These three columns won't affect any of our model analysis result.

```
> R_Spotify$target <- as.factor(R_Spotify$target)
  summary(R_Spotify)
  table(R_Spotify$target)
```

```
 danceability        energy               key            loudness             mode
 Min.   :0.0000   Min.   :0.000251   Min.   : 0.000   Min.   :-49.253   Min.   :0.0000
 1st Qu.:0.4200   1st Qu.:0.396000   1st Qu.: 2.000   1st Qu.:-12.816   1st Qu.:0.0000
 Median :0.5520   Median :0.601000   Median : 5.000   Median : -9.257   Median :1.0000
 Mean   :0.5397   Mean   :0.579545   Mean   : 5.214   Mean   :-10.222   Mean   :0.6934
 3rd Qu.:0.6690   3rd Qu.:0.787000   3rd Qu.: 8.000   3rd Qu.: -6.374   3rd Qu.:1.0000
 Max.   :0.9880   Max.   :1.000000   Max.   :11.000   Max.   :  3.744   Max.   :1.0000
  speechiness      acousticness     instrumentalness      liveness          valence
 Min.   :0.00000   Min.   :0.0000   Min.   :0.00000   Min.   :0.0130   Min.   :0.0000
 1st Qu.:0.03370   1st Qu.:0.0394   1st Qu.:0.00000   1st Qu.:0.0940   1st Qu.:0.3300
 Median :0.04340   Median :0.2580   Median :0.00012   Median :0.1320   Median :0.5580
 Mean   :0.07296   Mean   :0.3642   Mean   :0.15442   Mean   :0.2015   Mean   :0.5424
 3rd Qu.:0.06980   3rd Qu.:0.6760   3rd Qu.:0.06125   3rd Qu.:0.2610   3rd Qu.:0.7680
 Max.   :0.96000   Max.   :0.9960   Max.   :1.00000   Max.   :0.9990   Max.   :0.9960
     tempo         duration_ms      time_signature    chorus_hit        sections
 Min.   :  0.0   Min.   :  15168   Min.   :0.000   Min.   :  0.00   Min.   :  0.00
 1st Qu.: 97.4   1st Qu.: 172928   1st Qu.:4.000   1st Qu.: 27.60   1st Qu.:  8.00
 Median :117.6   Median : 217907   Median :4.000   Median : 35.85   Median : 10.00
 Mean   :119.3   Mean   : 234878   Mean   :3.894   Mean   : 40.11   Mean   : 10.48
 3rd Qu.:136.5   3rd Qu.: 266773   3rd Qu.:4.000   3rd Qu.: 47.63   3rd Qu.: 12.00
 Max.   :241.4   Max.   :4170227   Max.   :5.000   Max.   :433.18   Max.   :169.00
 target
 0:20553
 1:20553

    0     1
20553 20553
```

We can see summary of out Data Frame and also it can be noted that our Target variable got converted into factor. There are equal proportion of Hits and Flops in the Data.

Fig.45: Conversion to Factor

<u>Splitting the dataset into Train and Test Samples</u>

- We can split the data into 80:20 ratio, 80% of data will be for training purpose and the remaining 20% can be used for validating our results.

```
> set.seed(9999)
> index <- R_Spotify[sample(nrow(R_Spotify), 22000), ]
> partitioned.Spotify <- createDataPartition(
+         index$target,
+         times = 1,
+         p = 0.8,
+         list = F
+ )
>
> forest_train = index[partitioned.Spotify, ]
> forest_test = index[-partitioned.Spotify, ]
> R_Spotify$target <- as.factor(R_Spotify$target)
  summary(R_Spotify)
  table(R_Spotify$target)
```

It can be noted that we included only 22000 results as Random Forest eats up a lot of resources and it is hard to run Random Forest Algorithm on big data sets on a regular machine.

Fig.46: Train and Test Allocation

- Applying Random Forest algorithm to build the model. To apply random forest, we have to install the library 'randomForest'. Then we'll fit the model on training data.

```
> spotify_forest_model <- randomForest(target ~. , data = forest_train)
> spotify_forest_model

Call:
 randomForest(formula = target ~ ., data = forest_train)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 3

        OOB estimate of  error rate: 21.57%
Confusion matrix:
     0    1 class.error
0 6464 2363   0.2677014
1 1434 7340   0.1634374
```

Fig.47: Random Forest

- Here in the above output, we can see that the error rate Out-of-Bag is 21.57%
- Out of Bag data is the data that has been left out in the original dataset while taking random samples for training dataset from the original dataset.
- These selected samples are also called Bootstrap sample and the prediction error using the data which is not Bootstrap sample is the OOB error rate.

Now we'll summarize the attributes of Random Forest :

```
> summary(spotify_forest_model)
                Length Class  Mode     
call                3  -none- call     
type                1  -none- character
predicted       17601  factor numeric  
err.rate         1500  -none- numeric  
confusion           6  -none- numeric  
votes           35202  matrix numeric  
oob.times       17601  -none- numeric  
classes             2  -none- character
importance         15  -none- numeric  
importanceSD        0  -none- NULL     
localImportance     0  -none- NULL     
proximity           0  -none- NULL     
ntree               1  -none- numeric  
mtry                1  -none- numeric  
forest             14  -none- list     
y               17601  factor numeric  
test                0  -none- NULL     
inbag               0  -none- NULL     
terms               3  terms  call     
```

The summary of the random Forest function can be used to see Length, Class and Mode of all the features under consideration in your Random Forest Model.

Fig.48: Summary of Model

Creating Confusion Matrix :

```
> spotify_forest_model$confusion
     0    1 class.error
0 6464 2363   0.2677014
1 1434 7340   0.1634374
```

Fig.49: Confusion Matrix for Model

The above results show that 6464 Flops, 7340 Hits have been classified properly and correctly to the respective classes. Also, we can observe that the class 1, i.e. 'Hits' state has highest error of 26.7% and the class 0 has lowest, i.e. 'Flops'.

- The number of trees a randomForest algorithm make is depicted by ntree function, the default number is 500. We have possibilities to check how many number of trees do we need : ntree refers to the number of trees that grow in a Random Forest. We'll tune our Model for better accuracy.
- Below is the result of plot :

```
> plot(Spotify_forest_model)
```

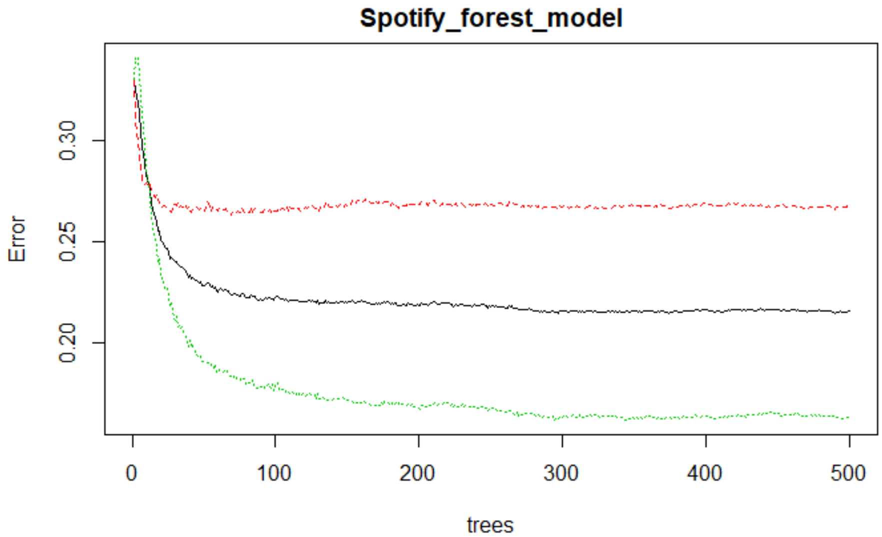


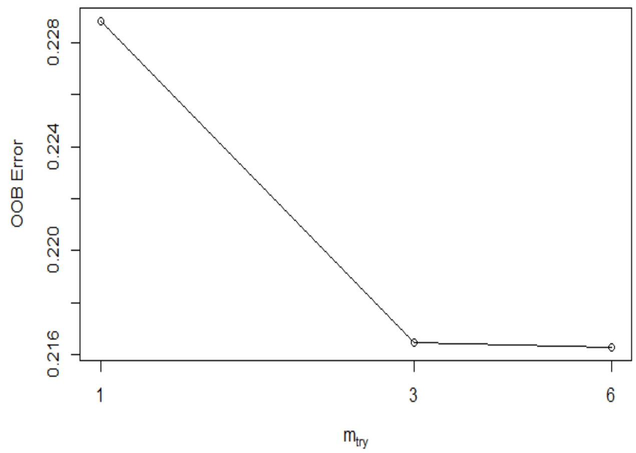
Fig.50: Plot for trees in our model

- By looking at the output in the plot, we can see that the error line gets constant from number of trees reaches 350 and more, therefore we'll give the value of 350 to ntree. Tuning our model for better accuracy :

```
> tuneRF(forest_train[,-16], forest_train$target,
+        stepFactor = 0.5,
+        plot = TRUE,
+        ntreeTry = 350,
+        trace = TRUE,
+        improve = 0.05)
mtry = 3  OOB error = 21.65%
Searching left ...
mtry = 6        OOB error = 21.63%
0.0007874016 0.05
Searching right ...
mtry = 1        OOB error = 22.89%
-0.05721785 0.05
      mtry  OOBError
1.OOB    1 0.2288506
3.OOB    3 0.2164650
6.OOB    6 0.2162945
```

The tuneRF will look for a better mtry value on both sides of the default value. We can see that the error rate slightly decreases when mtry is 6.

Fig.51: Tuning our model



We can see that the OOB error rate is constant when mtry is 3 and further decreases by a little bit when it is 6.

Fig.51: OOB Error Rate Plot

- Now fitting the random model on training data after tuning the model by giving the value of ntree = 350 and mtry = 6.
- Here, the parameter stepFactor suggests that , mtry value inflates or deflates by this number. ntreeTry shows the number of trees used at the tuning step. Improve suggests the improvement in OOB error must be by this much for search to continue. Trace is used to whether to print the progress of the search. Plot parameter is to specify whether to plot OOB function of mtry.

```
> set.seed(4444)
> Spotify_forest_model <- randomForest(target ~. , data = forest_train,
+                                       ntree = 350,
+                                       mtry = 6,
+                                       importance = TRUE,
+                                       proximity = TRUE)
> Spotify_forest_model

Call:
 randomForest(formula = target ~ ., data = forest_train, ntree = 350,      mtry = 6, importance = TRUE, proximity = TRUE)
               Type of random forest: classification
                     Number of trees: 350
No. of variables tried at each split: 6

        OOB estimate of  error rate: 21.39%
Confusion matrix:
     0    1 class.error
0 6493 2334   0.2644160
1 1431 7343   0.1630955
```

We found a slight improvement but negligible though. i.e. of 0.15%

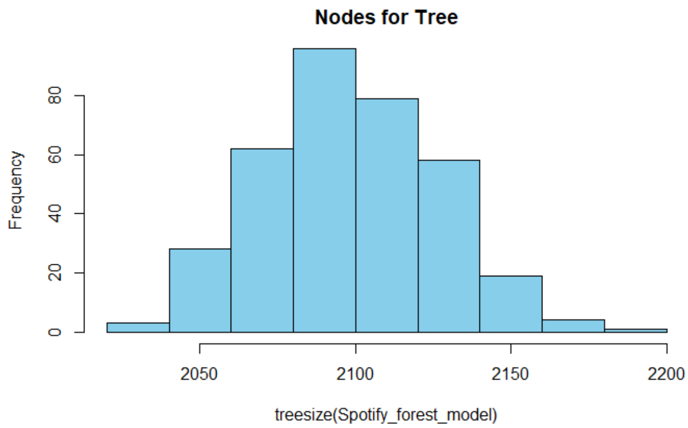
Fig.52: Random Forest on Model

- It can be observed that after tuning our model, the error rate has slightly decreased to 21.39%. Therefor the accuracy is 79.61%.

- We can see the number of nodes in our tree by following code :

```
> hist(treesize(Spotify_forest_model), main = "Nodes for Tree",
+       col = "skyblue")
```

- Output of the treesize() function is as shown below:



We can see that the maximum frequency for the number of nodes could be found in the range **2075-2125.**

Fig.52: Plot for treesize() function

- Quantifying the values of each predictors against our Target Variable, i.e. 'Target' :

```
> importance(Spotify_forest_model)
                         0          1 MeanDecreaseAccuracy MeanDecreaseGini
danceability     40.053849  87.229488            92.985363        931.64745
energy           29.117052  54.310466            74.770830        700.98835
key               6.161149   1.072719             5.332661        239.52996
loudness         10.523240  53.432581            52.582606        590.38885
mode              4.146454  29.585669            25.386047         86.78283
speechiness      34.832592  60.565076            68.205454        688.41919
acousticness     30.422202 101.011441           113.010647       1021.09762
instrumentalness 79.873077 196.115034           191.233188       1668.01490
liveness          3.229609  14.878011            14.071758        438.92512
valence          24.413996  49.725564            56.919054        586.93514
tempo            -2.029530  35.501386            28.410696        476.85603
duration_ms       1.999075  83.207843            76.403564        599.74691
time_signature   -8.552536  28.974511            27.096223         61.04398
chorus_hit        3.247722  12.464635            11.469455        429.14994
sections         11.531172  34.343989            37.839236        276.34604
```

Fig.53: Importance Matrix

- To find out which predictors were actually used in our Random Forest model, we'll use varUsed() function offered by randomForest package :

```
> varUsed(Spotify_forest_model)
 [1] 59961 59062 35850 58597  8561 62772 62262 47388 57738 60665 60268 60704  7288 57487 35422
```

- It can be noted that Instrumentalness is the one that is highly dependent.

Performing Predictions

- To make predictions we'll install 'caret' package in R. Then we'll use the predict() function on our Training Data First :

```
> #Making Predicitons
> forest_pred <- predict(Spotify_forest_model, forest_train)
> head(forest_pred,20)
13609  1673  5203 35860 25791 22518 39937 16636  2927  4903 38998   723 19773 39361 13443 20164 15535 39503 29711 27415
    0     0     1     1     1     1     1     0     1     1     0     1     0     1     1     1     0     0     1     0
Levels: 0 1
> #getting actual values
> head(forest_train$target, 20)
 [1] 0 0 1 1 1 1 1 0 1 1 0 1 0 1 1 1 0 0 1 0
Levels: 0 1
```

Fig.54: Predictions on random Forest Model

- We can see that the actual and predicted values are similar. Let's create a confusion matrix and check for accuracy based on our training data :

```
> #Confusion matrix for train data
> confusionMatrix(forest_pred, forest_train$target)
Confusion Matrix and Statistics

          Reference
Prediction    0    1
         0 8824    3
         1    3 8771

               Accuracy : 0.9997
                 95% CI : (0.9993, 0.9999)
    No Information Rate : 0.5015
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.9993

 Mcnemar's Test P-Value : 1

            Sensitivity : 0.9997
            Specificity : 0.9997
         Pos Pred Value : 0.9997
         Neg Pred Value : 0.9997
             Prevalence : 0.5015
         Detection Rate : 0.5013
   Detection Prevalence : 0.5015
      Balanced Accuracy : 0.9997

       'Positive' Class : 0
```

The accuracy of our Random Forest model on Training dataset is 99.97% and only 6 values were incorrectly classified.

Fig.55: Confusion Matrix

- Now, creating a confusion matrix and checking accuracy based on the test data :

```
> #Predicting and building confusion matrix on Test Data.
> forest_pred1 <- predict(Spotify_forest_model, forest_test)
> confusionMatrix(forest_pred1, forest_test$target)
Confusion Matrix and Statistics

          Reference
Prediction    0    1
         0 1611  390
         1  595 1803

               Accuracy : 0.7761
                 95% CI : (0.7635, 0.7883)
    No Information Rate : 0.5015
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.5523

 Mcnemar's Test P-Value : 8.033e-11

            Sensitivity : 0.7303
            Specificity : 0.8222
         Pos Pred Value : 0.8051
         Neg Pred Value : 0.7519
             Prevalence : 0.5015
         Detection Rate : 0.3662
   Detection Prevalence : 0.4549
      Balanced Accuracy : 0.7762

       'Positive' Class : 0
```

Fig.56: Confusion Matrix on Test Model

We got 77.61% accuracy after running our model on Test Data which is good.

Hence, we got 77.61% accuracy on our test data with 95% confidence interval in the range 92%-96%. Our predictions for target variable to predict Hits and Flops were pretty well.

K- Nearest Neighbors Algorithm

The K-Nearest Neighbors Algorithm is a supervised machine learning algorithm which is used for classification and regression predictions. The algorithm works by storing the available cases and then classifying the cases based upon the Euclidean distance between the data points.

Importing the Dataset

- The spotify dataset has been imported initially into a dataframe. Using the “str” function, the structure of the dataset has been analysed.
- We have selected the feature, “target” for understanding the number of Hit and Flop tracks and further factorized by labelling them.

```
> # table of target
> table(spotify$target)

    0     1
20553 20553
```

Fig.57: Data in “target”

We can observe that there are 20553 records are Hit and 20553 are Flop.

Analysis on Dataset

- The percentages of the labelled values, Hit and Flop were analysed using the “prop.table” function.

```
> # table or proportions with more informative labels
> round(prop.table(table(spotify$target)) * 100, digits = 1)

Flop  Hit
  50   50
```

Fig.58: Percentages of Labelled Values

As seen, the percentage of records having Hit and Flop tracks, which is equally distributed, as 50%.

- We have selected the three variables to understand and analyze certain numeric parameters.

```
> summary(spotify)
  danceability        energy              key              mode       
 Min.   :0.0000   Min.   :0.000251   Min.   : 0.000   Min.   :0.0000  
 1st Qu.:0.4200   1st Qu.:0.396000   1st Qu.: 2.000   1st Qu.:0.0000  
 Median :0.5520   Median :0.601000   Median : 5.000   Median :1.0000  
 Mean   :0.5397   Mean   :0.579545   Mean   : 5.214   Mean   :0.6934  
 3rd Qu.:0.6690   3rd Qu.:0.787000   3rd Qu.: 8.000   3rd Qu.:1.0000  
 Max.   :0.9880   Max.   :1.000000   Max.   :11.000   Max.   :1.0000  
```

We know that KNN works primarily on the numeric measurement of datapoints.

Fig.59: Summary of numeric parameters selected

- We can see that the values are not uniform which may give biased results and inaccurate predictions.

Normalisation

- Hence, we will normalize the data in order to have standardized and uniform values. Using the normalization function which will use the difference of the variable from the minimum value and divide by the range, the values will be standardized.

```
> normalize
function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
}
```

Using the normalization function which will use the difference of the variable from the minimum value and divide by the range, the values will be standardized.

Fig.60: Normalization

- We have checked the working of normalization function.

```
> # test normalization function - result should be identical
> normalize(c(1, 2, 3, 4, 5))
[1] 0.00 0.25 0.50 0.75 1.00
> normalize(c(10, 20, 30, 40, 50))
[1] 0.00 0.25 0.50 0.75 1.00
```

Fig.61: Verification Normalization Function

We can see that the values have been normalized for different set of input parameters.

- In order to perform the normalization function for all the values, we have used the "lapply" function which will return the same length for all the values.

```
spotify_n <- as.data.frame(lapply(spotify[1:14], normalize))
```

Fig.62: Normalisation for all values

- The data has been divided into Training and Testing Data for better prediction.

```
> # create training and test data
> spotify_train <- spotify_n[1:28775, ]
> spotify_test <- spotify_n[28776:41106, ]
```

As seen, the first 28775 columns have been sed for training the dataset and the remaining columns are tested on the remaining columns, with a 70% on train and 30% on Test samples.

Fig.63: Training and Testing Datasets

K-Nearest Neighbor

- After the initial data analysis is completed, the training for KNN classifier is performed. The "class" package is used which classifies the datapoints for KNN algorithm. The algorithm selects the nearest neighbors based on the Euclidean distance of each datapoint and as per the number of clusters specified.[1]

```
> spotify_test_pred <- knn(train = spotify_train, test = spotify_test,cl = spotify_t
rain_labels, k=200)
```

Fig.64: KNN Classifier

We can built the KNN classifier on the training and testing data considering 200 clusters.

- Since the training data is built upon 41106 columns, we have used the number of clusters as 200 which is almost equivalent to the square root of that number.

- We have checked the performance of the model by comparing with the predicted values and test values in the testing dataset. Hence, the Cross Table function included in the "gmodels" library will help in comparisons of two vectors.

```
> # Create the cross tabulation of predicted vs. actual
> CrossTable(x = spotify_test_labels, y = spotify_test_pred,
+            prop.chisq=FALSE)
```

Fig.9: Cross Table

```
                     | spotify_test_pred 
 spotify_test_labels |      Flop |       Hit | Row Total | 
---------------------|-----------|-----------|-----------|
                Flop |      3633 |      2536 |      6169 | 
                     |     0.589 |     0.411 |     0.500 | 
                     |     0.911 |     0.304 |           | 
                     |     0.295 |     0.206 |           | 
---------------------|-----------|-----------|-----------|
                 Hit |       355 |      5807 |      6162 | 
                     |     0.058 |     0.942 |     0.500 | 
                     |     0.089 |     0.696 |           | 
                     |     0.029 |     0.471 |           | 
---------------------|-----------|-----------|-----------|
        Column Total |      3988 |      8343 |     12331 | 
                     |     0.323 |     0.677 |           | 
---------------------|-----------|-----------|-----------|
```

Fig. 65: Confusion Matrix

True Negative: The top left value of 3633 has been accurately been identified as Flop out of 3988 values.

True Positive: The values of 5807 of 8343 was correctly identified as Hit.

False Negative: The value of 355 shows that the predicted value was Flop while the label was Hit.

False Positive: The model precited 2536 values for tracks that were Hit but actually Flop.

- We can see that the model predicted 9440 out of 12331 values incorrectly and the accuracy was seen to be 76.55%.
- The improvement of prediction can be very well performed by Normalization. However, normalization, does not compress the middle values for uniformity always. Hence, outliers are not taking into account as the extreme values of compressed. Therefore, Z-score standardization can be useful in such cases.

<u>Z-Score Standardization</u>

- Z-score standardization uses the scale() function to re-scale the values for better optimization.

```
> # use the scale() function to z-score standardize a data frame
> spotify_z <- as.data.frame(scale(spotify[-15]))
```

Fig.11: Re-scaling using Z-score

The Z-score value should be 0 as, we can see the mean value is 0 in summary.

```
> summary(spotify_z$target)
Length  Class   Mode
     0   NULL   NULL
```

Fig.66: Summary of Z-Score function

- As performed earlier, we will be dividing the data into testing and training data labels, then perform the KNN classifier and built the cross table.

```
Total Observations in Table:  12331 

 
                    | spotify_test_pred 
spotify_test_labels |      Flop |       Hit | Row Total | 
--------------------|-----------|-----------|-----------|
               Flop |      3344 |      2825 |      6169 | 
                    |     0.542 |     0.458 |     0.500 | 
                    |     0.923 |     0.324 |           | 
                    |     0.271 |     0.229 |           | 
--------------------|-----------|-----------|-----------|
                Hit |       278 |      5884 |      6162 | 
                    |     0.045 |     0.955 |     0.500 | 
                    |     0.077 |     0.676 |           | 
                    |     0.023 |     0.477 |           | 
--------------------|-----------|-----------|-----------|
       Column Total |      3622 |      8709 |     12331 | 
                    |     0.294 |     0.706 |           | 
--------------------|-----------|-----------|-----------|
```

We can see that, the False Negative values were incorrectly classified as 278, Hence the accuracy has reduced to 75%.

Fig.67: Confusion Matrix using Z-Score

- Testing for several values of K has been done as follows on the same function:

```
> spotify_test_pred <- knn(train = spotify_train, test = spotify_test, cl = spotify_
train_labels, k=210)
                    | spotify_test_pred 
spotify_test_labels |      Flop |       Hit | Row Total | 
--------------------|-----------|-----------|-----------|
               Flop |      3618 |      2551 |      6169 | 
                    |     0.586 |     0.414 |     0.500 | 
                    |     0.910 |     0.305 |           | 
                    |     0.293 |     0.207 |           | 
--------------------|-----------|-----------|-----------|
                Hit |       358 |      5804 |      6162 | 
                    |     0.058 |     0.942 |     0.500 | 
                    |     0.090 |     0.695 |           | 
                    |     0.029 |     0.471 |           | 
--------------------|-----------|-----------|-----------|
       Column Total |      3976 |      8355 |     12331 | 
                    |     0.322 |     0.678 |           | 
--------------------|-----------|-----------|-----------|
```

Fig.68: Confusion Matrix for K=210

```
                     | spotify_test_pred
spotify_test_labels  |      Flop |       Hit | Row Total |
---------------------|-----------|-----------|-----------|
                Flop |      3650 |      2519 |      6169 |
                     |     0.592 |     0.408 |     0.500 |
                     |     0.911 |     0.303 |           |
                     |     0.296 |     0.204 |           |
---------------------|-----------|-----------|-----------|
                 Hit |       355 |      5807 |      6162 |
                     |     0.058 |     0.942 |     0.500 |
                     |     0.089 |     0.697 |           |
                     |     0.029 |     0.471 |           |
---------------------|-----------|-----------|-----------|
        Column Total |      4005 |      8326 |     12331 |
                     |     0.325 |     0.675 |           |
---------------------|-----------|-----------|-----------|
```

Fig.69: Confusion Matrix for K=190

```
                     | spotify_test_pred
spotify_test_labels  |      Flop |       Hit | Row Total |
---------------------|-----------|-----------|-----------|
                Flop |      3668 |      2501 |      6169 |
                     |     0.595 |     0.405 |     0.500 |
                     |     0.913 |     0.301 |           |
                     |     0.297 |     0.203 |           |
---------------------|-----------|-----------|-----------|
                 Hit |       351 |      5811 |      6162 |
                     |     0.057 |     0.943 |     0.500 |
                     |     0.087 |     0.699 |           |
                     |     0.028 |     0.471 |           |
---------------------|-----------|-----------|-----------|
        Column Total |      4019 |      8312 |     12331 |
                     |     0.326 |     0.674 |           |
---------------------|-----------|-----------|-----------|
```

Fig.70: Confusion Matrix for K=180

- As we can see, the algorithm gives more accurate results for K=180 with accuracy of approximately 77% as compared to other values.

K- Values	***Percentage of Accuracy***
210	76.40
190	76.69
180	76.87
200	76.55

Fig.71: Table showing the Accuracies of K-values

Conclusion

- As per the analysis on the Spotify dataset, we can see that Naïve Bayes classifier is a very effective algorithm, the accuracy is 52.16% and model performance of the algorithm was improved by using Laplace to 58%. However, we have observed that the accuracy still needs improvement and the Naïve Bayes algorithm is not suitable for our dataset.
- Moreover, after performing K-Nearest Neighbors algorithm on the dataset, the accuracy was observed to be 77% for 180 Kth value. This algorithm has performed better compared to Naïve Bayes algorithm.
- The logistic regression was performed, as we need to predict the tracks that were "Hit" and "Flop", which are categorical variables in our dataset. The accuracy was predicted to be 70.17%.
- The Decision Tree algorithm has used decision rules for predictions with an accuracy of 76.90%. After performing Adaptive Boosting, the accuracy was improved to 78.28%.
- Furthermore, Random Forest has performed predictions by merging many decision trees and used the bagging method with an accuracy of 77.61%. The accuracy was improved to 79.61% after tuning the data.
- It can be observed that, Random Forest has given the best accuracy of approximately 80% in performing predictions.

References

[1] Lantz, B. (2015). Machine learning with R: learn how to use R to apply powerful machine learning methods and gain an insight into real-world applications. Birmingham: Packt Publ.

[2] lakshmi25npathi. (2019, June 19). Sentiment Analysis of IMDB Movie Reviews. Retrieved from https://www.kaggle.com/lakshmi25npathi/sentiment-analysis-of-imdb-movie-reviews

[3] Jeffrey Strickland (2014). Predictive Analytics using R. Retrieved from –

https://www.slideshare.net/JeffreyStricklandPhD/predictiveanalyticsusingrredc

[4] Ansari, F. (2020, April 25). The Spotify Hit Predictor Dataset (1960-2019). Retrieved May 11, 2020, from https://www.kaggle.com/theoverman/the-spotify-hit-predictor-dataset

[5] Prabhakaran, S. (n.d.). eval(ez_write_tag([[728,90],'r_statistics_co-box-3','ezslot_4',109,'0','0']));Logistic Regression. Retrieved from http://r-statistics.co/Logistic-Regression-With-R.html