# Final Project - Statical Analysis on Spotify Popularity Score

Shaoshao Xiong(sx24), Jay Chen(jc123)

2024-04-28

We wish to study what are the several most significant predictors of a track's popularity on Spotify, and how can we accurately predict these popularity scores based on track features and trends? This model could also help music companies to optimize their investment and focus on music style that is most likely to succeed so that they could maximize their revenue. Furthermore, this model would help the artists to realize which characteristics of their music is less popular, hence allow their future productions to better meet the market demand and achieve commercial success.

We look at two different studies by other researchers on the similar topic. The first study is "SpotHitPy: A Study For ML-Based Song Hit Prediction Using Spotify" by Ioannis et al., which presents a comprehensive analysis of predicting song popularity using machine learning techniques. The study used a data set that contains approximately 18,000 Spotify songs where 861 of them were in the BillBoard Top 100 between 2011 and 2021. Dataset's features includes id, artist, popularity, explicit, album type, danceability, emergy, key, loudness, mode speechiness, acousticness, instrumentalness, liveness, valence, tempo, duration_ms and time_signature. The researchers used multiple statistical learning methods and found out that Random Forest and Support Vector Machines were most effective, achieving an accuracy of approximately 86%. Random Forest achieved high precision on both the training and the test set, making it suitable for the Hit Song prediction problem, while Support Vector Machines had higher accuracy on the test set. This high level of predictive accuracy underscores the potential of machine learning in identifying future hits, providing valuable insights for artists and music producers about the traits that potentially lead to a song's commercial success.

The second study is "Music Popularity: Metrics, Characteristics, and Audio-based Prediction" by Junghyuk Lee and Jong-Seok Lee, which explores different aspects of music popularity and its predictability through audio features. The study used a dataset of 16,686 songs ranked in the Billboard Hot 100 chart between 1970 and 2014. The authors defined multiple popularity metrics and analyzed them using real-world chart data, then automatically predicted them using acoustic features. They developed classification models to predict these metrics using features like MPEG-7 audio features, Mel-frequency cepstral coefficients (MFCCs), and music complexity features including Harmony, Rhythm and Timbre. The study summarizes that the maximum rank of a song was highly related to its debut performance. The study uses SVMs and concluded that Complexity was superior to MFCC and MPEG. The study highlights the partial success in predicting music popularity metric and stated that it is necessary to attempt to improve the prediction performance in the future.

Both studies use a dataset of approximately 17,000 songs to predict the popularity of them, which share similarity with this study. Both studies aim to find out the relationship between a song's features and its market performance and both of them use SVM. The first study's approach is closer to our study, as both share similar features as predictors.

For our study, we choose a dataset from kaggle.com with link can be found in the reference page. There are 18 variables in the dataset with 232,725 observations (or tracks). To clean the data, we first read from the csv file, filter out the NA, NAN, and INF values. We then remove the irrelevant predictors, including artist_name, track_id, track_name, genre, key, mode, and time_signature, as we believe they may not be statistically significant in our analysis. Then we change the value two predictors, instrumentalness and

liveness, to be binary as suggested by the Spotify API documentation. Our first approach would be the simple linear regression.

```
library(tidyverse)
```

```
## Warning: package 'ggplot2' was built under R version 4.2.3
```

```
## Warning: package 'tidyr' was built under R version 4.2.3
```

```
## Warning: package 'readr' was built under R version 4.2.3
```

```
## Warning: package 'dplyr' was built under R version 4.2.3
```

```
## Warning: package 'stringr' was built under R version 4.2.3
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.0     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

```
library(caret)
```

```
## Loading required package: lattice
##
## Attaching package: 'caret'
##
## The following object is masked from 'package:purrr':
##
##     lift
```

```
raw_data <- read.csv("SpotifyFeatures.csv", header = TRUE,
                     sep = ",", quote = "\"", dec = ".")

cleaned_data <- raw_data %>% distinct(track_id, .keep_all = TRUE) %>%
filter_all(all_vars(!is.na(.) & !is.nan(.) & !is.infinite(.))) %>%
select (-artist_name, -track_id, -track_name, -genre, -key, -mode,
        -time_signature) %>%
mutate(instrumentalness = if_else(instrumentalness > 0.5, 1, 0)) %>%
mutate(liveness = if_else(liveness > 0.8, 1, 0))

filtered_data <- cleaned_data %>%
  filter(popularity >= 70)
head(filtered_data)
```

```
##   popularity acousticness danceability duration_ms energy instrumentalness
## 1         71       0.7350        0.501      257733  0.378                0
## 2         76       0.0233        0.845      187521  0.709                0
## 3         70       0.4100        0.721      213750  0.881                0
## 4         72       0.3230        0.637      218947  0.730                0
## 5         70       0.1980        0.781      269493  0.745                0
## 6         70       0.5300        0.679      264200  0.440                0
##   liveness loudness speechiness   tempo valence
## 1        0   -9.370      0.0290 119.987  0.1780
## 2        0   -4.547      0.0714  98.062  0.6200
## 3        0   -2.528      0.3420 127.759  0.6430
## 4        0   -5.380      0.0874  93.867  0.7320
## 5        0   -5.810      0.0332 129.998  0.3260
## 6        0   -9.313      0.3510 109.959  0.0336
```

We create indices for the training set. Splitting 70 percent of the data as the training set and 30 percent of the data as the test set.

```
set.seed(123)

# Create indices for the training set
trainIndex <- createDataPartition(cleaned_data$popularity,
                                  p = 0.7, list = FALSE, times = 1)
train_data <- cleaned_data[trainIndex, ]
test_data <- cleaned_data[-trainIndex, ]

ols = lm(popularity ~ .,  data = train_data)
summary(ols)
```

```
##
## Call:
## lm(formula = popularity ~ ., data = train_data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -52.524 -10.400   0.849  10.587  61.761
##
## Coefficients:
##                    Estimate Std. Error t value Pr(>|t|)
## (Intercept)       4.588e+01  4.542e-01 101.019   <2e-16 ***
## acousticness     -1.083e+01  2.063e-01 -52.502   <2e-16 ***
## danceability      1.788e+01  3.232e-01  55.325   <2e-16 ***
## duration_ms       3.169e-06  3.435e-07   9.227   <2e-16 ***
## energy           -3.652e+00  3.666e-01  -9.962   <2e-16 ***
## instrumentalness -1.166e+00  1.352e-01  -8.624   <2e-16 ***
## liveness         -3.440e+00  2.544e-01 -13.520   <2e-16 ***
## loudness          5.493e-01  1.419e-02  38.717   <2e-16 ***
## speechiness      -1.045e+01  2.799e-01 -37.323   <2e-16 ***
## tempo            -2.551e-03  1.489e-03  -1.713   0.0868 .
## valence          -1.344e+01  2.231e-01 -60.211   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## Residual standard error: 15.6 on 123733 degrees of freedom
## Multiple R-squared:  0.1959, Adjusted R-squared:  0.1958
## F-statistic:  3014 on 10 and 123733 DF,  p-value: < 2.2e-16
```

```
set.seed(123)
predictions <- predict(ols, newdata=test_data)
mse <- mean((predictions - test_data$popularity)^2)
print(mse)
```

```
## [1] 243.9313
```

From the summary we could see that every one of the coefficients in our Linear Regression model is statistically significant. Linear regression gives a MSE of 243.93 and a Adjusted R-squared of 0.1958, which implies that the model did not capture many factors that might influence the popularity score. Howeve, based on the statistics, we find all predictors except tempo to be significant because of the extremely small p-value. We now check if our select of predictors are reasonable by applying lasso regression.

```
library(glmnet)
```

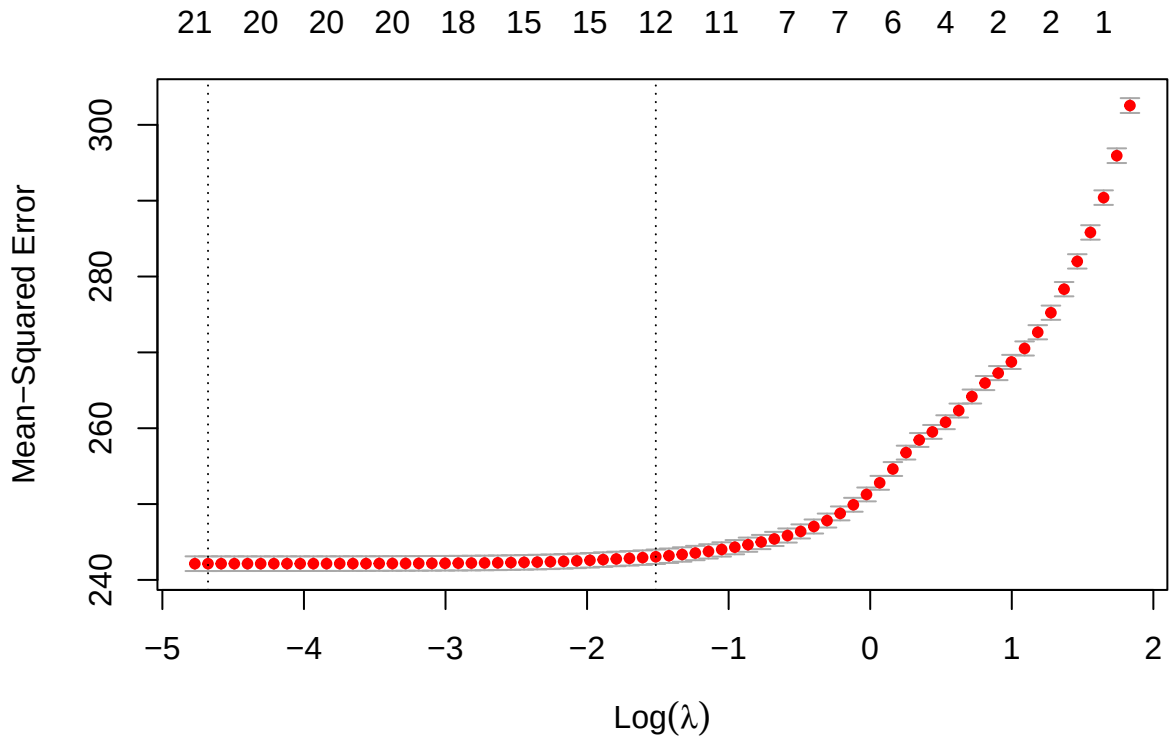
```
## Loading required package: Matrix

##
## Attaching package: 'Matrix'

## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack

## Loaded glmnet 4.1-8
```

```
library(fastDummies)
```

```
## Thank you for using fastDummies!

## To acknowledge our work, please cite the package:

## Kaplan, J. & Schlegel, B. (2023). fastDummies: Fast Creation of Dummy (Binary) Columns and Rows from
```

```
set.seed(123)

lasso_data <- raw_data %>% distinct(track_id, .keep_all = TRUE) %>%
filter_all(all_vars(!is.na(.) & !is.nan(.) & !is.infinite(.))) %>%
select (-artist_name, -track_id, -track_name, -genre, -time_signature) %>%
mutate(liveness = if_else(liveness > 0.8, 1, 0),
       instrumentalness = if_else(instrumentalness > 0.5, 1, 0),
       mode_major = as.integer(mode == "Major"),
       mode_minor = as.integer(mode == "Minor"),
      )%>%
dummy_cols(select_columns = "key", remove_first_dummy = TRUE,
ignore_na = TRUE) %>% select(-key, -mode)
```

```r
# Create indices for the training set
lasso_trainIndex <- createDataPartition (lasso_data$popularity, p = 0.7,
                                         list = FALSE, times = 1)
lasso_train_data <- lasso_data[lasso_trainIndex, ]
lasso_test_data <- lasso_data[-lasso_trainIndex, ]


X <- as.matrix(lasso_train_data
               [, -which(names(lasso_train_data) == "popularity")])
Y <- as.vector(lasso_train_data$popularity)


lasso.model <- glmnet(X, Y, alpha=1)

cv.lasso <- cv.glmnet(X, Y, alpha=1)
# plot the cross validation result
plot(cv.lasso)
```




```r
best.lambda <- cv.lasso$lambda.min

newX <- as.matrix(lasso_test_data
                  [, -which(names(lasso_test_data) == "popularity")])
newY <- as.vector(lasso_test_data$popularity)

lasso.pred <- predict(lasso.model, s=best.lambda, newx=newX)
```

```r
lasso_mse <- mean((lasso.pred - newY)^2)
print(lasso_mse)
```

```
## [1] 242.5021
```

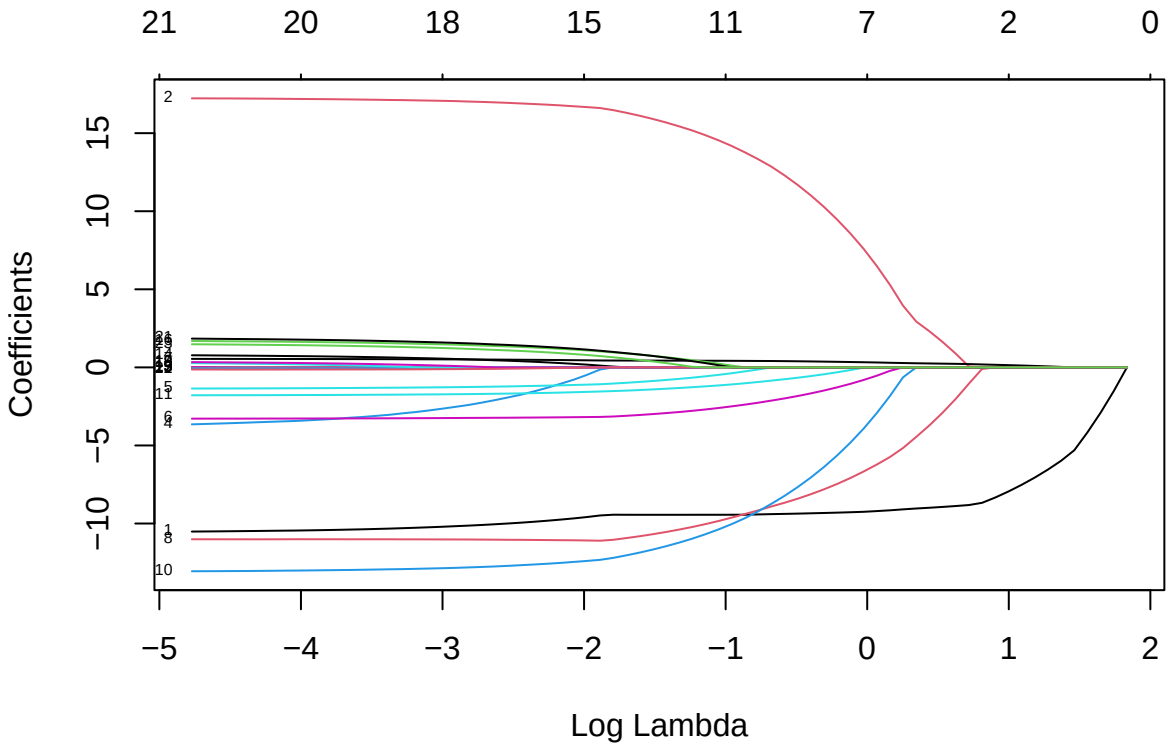
Comparing the lasso mse(242.502) with linear regression mse(243.93), we observer a difference of about 1.4. Given the minimal difference, we may deduce that either the variables dropped by the Lasso model were not significantly contributing to the prediction, or the dataset does not have issues with multicollinearity or overfitting that Lasso is specifically designed to handle. To confirm our assumption, we print the coefficients and plot the coefficient paths.

```r
# Use the best lambda to extract coefficients
lasso_coefficients <- coef(lasso.model, s = best.lambda)
print(lasso_coefficients)
```

```
## 24 x 1 sparse Matrix of class "dgCMatrix"
##                              s1
## (Intercept)       4.663660e+01
## acousticness     -1.050847e+01
## danceability      1.722738e+01
## duration_ms       3.008902e-06
## energy           -3.623711e+00
## instrumentalness -1.352636e+00
## liveness         -3.282267e+00
## loudness          5.434029e-01
## speechiness      -1.100807e+01
## tempo            -1.695123e-03
## valence          -1.305276e+01
## mode_major       -1.785139e+00
## mode_minor        .
## key_A#            1.694263e-02
## key_B             7.697375e-01
## key_C            -3.053821e-02
## key_C#            1.683312e+00
## key_D            -6.241871e-02
## key_D#            2.626232e-01
## key_E             3.317445e-01
## key_F             .
## key_F#            1.834873e+00
## key_G            -1.359445e-01
## key_G#            1.475289e+00
```

The values of the coefficients suggests that features like danceability and loudness have positive effects on a track's popularity, whereas features such as acousticness, energy, speechiness, and valence typically reduce it. The effects of different musical keys on popularity also vary, with some keys associated with higher or lower popularity relative to the baseline.

```r
# Plot the coefficient paths
plot(lasso.model, xvar = "lambda", label = T)
```




From the graph, we observe that at higher values of log(lambda), many coefficients are shrunk to zero, suggesting that the model is selecting a smaller subset of features. Also, a few paths remain non-zero across a wide range of lambda values before reaching zero, indicating that these features are influential across various degrees of regularization and may be strong predictors. Whereas the fact that some coefficient paths cross the zero line quite early as lambda increases suggests that these features are less important and are eliminated by the Lasso model early in the regularization process. To sum up, we may conclude that our choice of variables for the linear regression is very reasonable and appropriate, with all variables being significant as shown by the graphs and coefficient values.

We move on to Random Forest. We consider Random Forest because it can provide insights into which features are most important in predicting track popularity, which aligns with our goal to understand what features influence popularity. Random Forest also could handle a mix of binary, categorical, and numerical features well.

```
library(randomForest)
```

```
## randomForest 4.7-1.1

## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:dplyr':
##
##     combine
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin
```

```
set.seed(123)
rf_model <- randomForest(popularity ~ ., data = train_data,
                         ntree = 100, mtry = 3)
# Print summary of the model
print(summary(rf_model))
```

```
##                 Length Class  Mode
## call                 5 -none- call
## type                 1 -none- character
## predicted       123744 -none- numeric
## mse                100 -none- numeric
## rsq                100 -none- numeric
## oob.times       123744 -none- numeric
## importance          10 -none- numeric
## importanceSD         0 -none- NULL
## localImportance      0 -none- NULL
## proximity            0 -none- NULL
## ntree                1 -none- numeric
## mtry                 1 -none- numeric
## forest              11 -none- list
## coefs                0 -none- NULL
## y               123744 -none- numeric
## test                 0 -none- NULL
## inbag                0 -none- NULL
## terms                3 terms  call
```

```
# Predicting on test data
predictions <- predict(rf_model, newdata = test_data)

# Calculate Mean Squared Error
mse <- mean((predictions - test_data$popularity)^2)
print(paste("Mean Squared Error:", mse))
```

```
## [1] "Mean Squared Error: 201.326466327961"
```

As we can see from the result, MSE becomes 151.5233, which is lower than both the Linear Regression model and the lasso, showing that Random Forest is the best model in evaluating popularity score.

In the next section of our analysis, instead of predicting the numerical value of our popularity score, we want to classify songs as "hits" and "non-hits" songs based on their popularity score. We have decided that a song could be counted as hits if their popularity score is in the 99th percentile of the whole population, which is greater than 70 in this case. We split the dataset into "hits"(1) and "non-hits"(0) and perform logistic regression. Before doing the logistic regression, we mandatorily balance the dataset because the initial data's non-hits significantly outnumber hits. Then we employed logistic regression and trained the model on the balanced dataset and evaluated it against a test set from the original data.

```
library(caret)
library(dplyr)
set.seed(123)
```

```r
cleaned_data <- cleaned_data %>%
  mutate(hit = ifelse(popularity > 70, 1, 0))
# Split the data into training and test sets
splitIndex <- createDataPartition(cleaned_data$hit, p = 0.7, list = FALSE)
train_data <- cleaned_data[splitIndex, ]
test_data <- cleaned_data[-splitIndex, ]

# Balance the training dataset
hits <- train_data %>% filter(hit == 1)
non_hits <- train_data %>% filter(hit == 0)
test_data$hit <- factor(test_data$hit, levels = c(0, 1))
# Downsample to ensure a balance dataset
non_hits_downsampled <- non_hits %>%
  sample_n(nrow(hits))

balanced_train_data <- bind_rows(hits, non_hits_downsampled)

# Fit logistic regression model on the balanced data
logit_model_balanced <- glm(as.factor(hit) ~ . - popularity,
                            data = balanced_train_data, family = 'binomial')

# Calculate test predictions and error rate
test_predictions_prob_balanced <- predict (logit_model_balanced,
                                           newdata = test_data,
                                           type = "response")
test_predictions_balanced <- ifelse(test_predictions_prob_balanced > 0.5, 1, 0)
test_error_rate_balanced <- mean(test_predictions_balanced != test_data$hit)
test_accuracy_balanced <- 1 - test_error_rate_balanced

# Print results
print(paste("Test Error Rate (Balanced):", test_error_rate_balanced))
```

```
## [1] "Test Error Rate (Balanced): 0.366363704932871"
```

```r
print(paste("Test Accuracy (Balanced):", test_accuracy_balanced))
```

```
## [1] "Test Accuracy (Balanced): 0.633636295067129"
```

```r
# Create a confusion matrix for the balanced model and print it
confusionMatrix_balanced <- confusionMatrix(
  as.factor(test_predictions_balanced), as.factor(test_data$hit))
print(confusionMatrix_balanced)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction     0     1
##          0 32839   233
##          1 19196   764
##
##                Accuracy : 0.6336
##                  95% CI : (0.6295, 0.6377)
```

```
##     No Information Rate : 0.9812
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.0385
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.63109
##             Specificity : 0.76630
##          Pos Pred Value : 0.99295
##          Neg Pred Value : 0.03828
##              Prevalence : 0.98120
##          Detection Rate : 0.61923
##    Detection Prevalence : 0.62362
##       Balanced Accuracy : 0.69870
##
##        'Positive' Class : 0
##
```

```r
balanced_hit_counts <- table(balanced_train_data$hit)
print(balanced_hit_counts)
```

```
##
##    0    1
## 2263 2263
```

We eventually received a test error rate of 0.366. While logistic regression provides a baseline model for predicting song popularity as hits or non-hits, the results suggest that a more complex model might be required to enhance predictive accuracy.

We then extended our analysis to Support Vector Machine(SVM), which is a more complicated classification technique that could potentially handle non-linear relationships with kernels. We trained the SVM model on the balanced dataset and evaluated the model with the original, unbalanced test dataset.

```r
library(e1071)
```

```
## Warning: package 'e1071' was built under R version 4.2.3
```

```r
cleaned_data <- cleaned_data %>%
  mutate(hit = ifelse(popularity > 70, 1, 0))

# Fit SVM model on the balanced data
svm_model <- svm(as.factor(hit) ~ . - popularity, data = balanced_train_data,
                 method = 'C-classification', kernel = 'radial')

# Predict on test data
test_predictions <- predict(svm_model,
                    newdata = test_data[-which(names(test_data) == "hit")])

# Calculate test predictions and error rate
test_error_rate <- mean(as.numeric(test_predictions) - 1 != test_data$hit)
test_accuracy <- 1 - test_error_rate
```

```r
# Print results
print(paste("Test Error Rate:", test_error_rate))
```

```
## [1] "Test Error Rate: 0.374830291144969"
```

```r
print(paste("Test Accuracy:", test_accuracy))
```

```
## [1] "Test Accuracy: 0.625169708855031"
```

```r
# Create a confusion matrix for the SVM model and print it
confusionMatrix <- confusionMatrix(as.factor(test_predictions),
                                   as.factor(test_data$hit))
print(confusionMatrix)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction     0     1
##          0 32337   180
##          1 19698   817
##
##                Accuracy : 0.6252
##                  95% CI : (0.621, 0.6293)
##     No Information Rate : 0.9812
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.0416
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.62145
##             Specificity : 0.81946
##          Pos Pred Value : 0.99446
##          Neg Pred Value : 0.03982
##              Prevalence : 0.98120
##          Detection Rate : 0.60976
##    Detection Prevalence : 0.61316
##       Balanced Accuracy : 0.72045
##
##        'Positive' Class : 0
##
```

The result gives us a test error rate of 0.375, which is slightly higher than the logistic regression model.

We now will consider a versatile method, K-Nearest Neighbors. To choose the best k value, we use define a range of k to try and use 10-fold cross-validation to avoid overfitting.

```r
library(caret)
library(class)
control <- trainControl(method = "cv", number = 10)
```

```r
k_values <- data.frame(k = c(1, 5, 10, 20, 50, 100))

balanced_train_data$hit <- factor(balanced_train_data$hit, levels = c(0, 1))

# Train the model
knn <- train(hit ~ ., data = balanced_train_data, method = "knn",
                 tuneGrid = k_values, trControl = control)

#Print the results
print(knn)
```

```
## k-Nearest Neighbors
##
## 4526 samples
##   11 predictor
##    2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 4073, 4073, 4074, 4074, 4074, 4072, ...
## Resampling results across tuning parameters:
##
##   k    Accuracy   Kappa
##     1  0.6493589  0.2986903
##     5  0.6000929  0.2001571
##    10  0.5841720  0.1683257
##    20  0.5870564  0.1741031
##    50  0.6058397  0.2116842
##   100  0.6087046  0.2174102
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 1.
```

```r
summary(knn)
```

```
##             Length Class      Mode
## learn        2     -none-     list
## k            1     -none-     numeric
## theDots      0     -none-     list
## xNames      11     -none-     character
## problemType  1     -none-     character
## tuneValue    1     data.frame list
## obsLevels    2     -none-     character
## param        0     -none-     list
```

From the result, we can see that at k=1, the highest accuracy is reported at approximately 64.93%. This suggests moderate effectiveness of the model. Here, k=1 may be slightly overfitting as it captures too much noise or outliers in the data, yet it still provides the best accuracy among the tested k values. As k increases, the training error rate generally decreases, indicating that including more neighbors dilutes the prediction quality for our dataset.

```r
# Make prediction on new data
knn_predictions <- predict(knn, newdata = test_data)

# Create a confusion matrix for the balanced model and print it
knn_confusion_matrix <- confusionMatrix(as.factor(knn_predictions),
                                        as.factor(test_data$hit))
print(knn_confusion_matrix)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction     0     1
##          0 32395   320
##          1 19640   677
##
##                Accuracy : 0.6236
##                  95% CI : (0.6195, 0.6277)
##     No Information Rate : 0.9812
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.0287
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.62256
##             Specificity : 0.67904
##          Pos Pred Value : 0.99022
##          Neg Pred Value : 0.03332
##              Prevalence : 0.98120
##          Detection Rate : 0.61086
##    Detection Prevalence : 0.61689
##       Balanced Accuracy : 0.65080
##
##        'Positive' Class : 0
##
```

Based on confusion matrix of the actual test, we have a accuracy of 62.36%, or test error rate of 37.64%. If we look at the predictive values, the high value of 0.99022 indicates that when the model predicts class 0, it is correct about 99.02% of the time. However, the negative predictive values are very low at 0.03332, suggesting that when the model predicts class 1, it is correct only about 3.33% of the time. Overall, the kNN model shows limited effectiveness, with modest accuracy and a strong bias towards predicting the majority class. Its predictive performance for class 1 is notably poor, as indicated by the very low negative predictive value.

Finally, we employed the Random Forest as a more complicated classification technique to predict the song as hits versus not-hits. Random Forest is a robust ensemble learning method known for its high accuracy and ability to handle overfitting through constructing a multitude of decision trees at training time. We trained the Random Forest model on the balanced dataset and evaluated the model with the original, unbalanced test dataset. The model was configured with 500 trees and considered three variables at each split. The result gives us a test error rate of 0.355, which is the best among three classification models and indicates the effectiveness of Random Forest in this application.

```r
library(randomForest)
# fit Random Forest model on the balanced data
rf_model <- randomForest(as.factor(hit) ~ . - popularity,
                         data = balanced_train_data, ntree = 500, mtry = 3)
test_predictions <- predict(rf_model, newdata = test_data)

# calculate test predictions and error rate
test_error_rate <- mean(as.numeric(test_predictions) - 1 != test_data$hit)
test_accuracy <- 1 - test_error_rate

# print results
print(paste("Random Forest Test Error Rate:", test_error_rate))
```

```
## [1] "Random Forest Test Error Rate: 0.314979634937396"
```

```r
print(paste("Random Forest Test Accuracy:", test_accuracy))
```

```
## [1] "Random Forest Test Accuracy: 0.685020365062604"
```

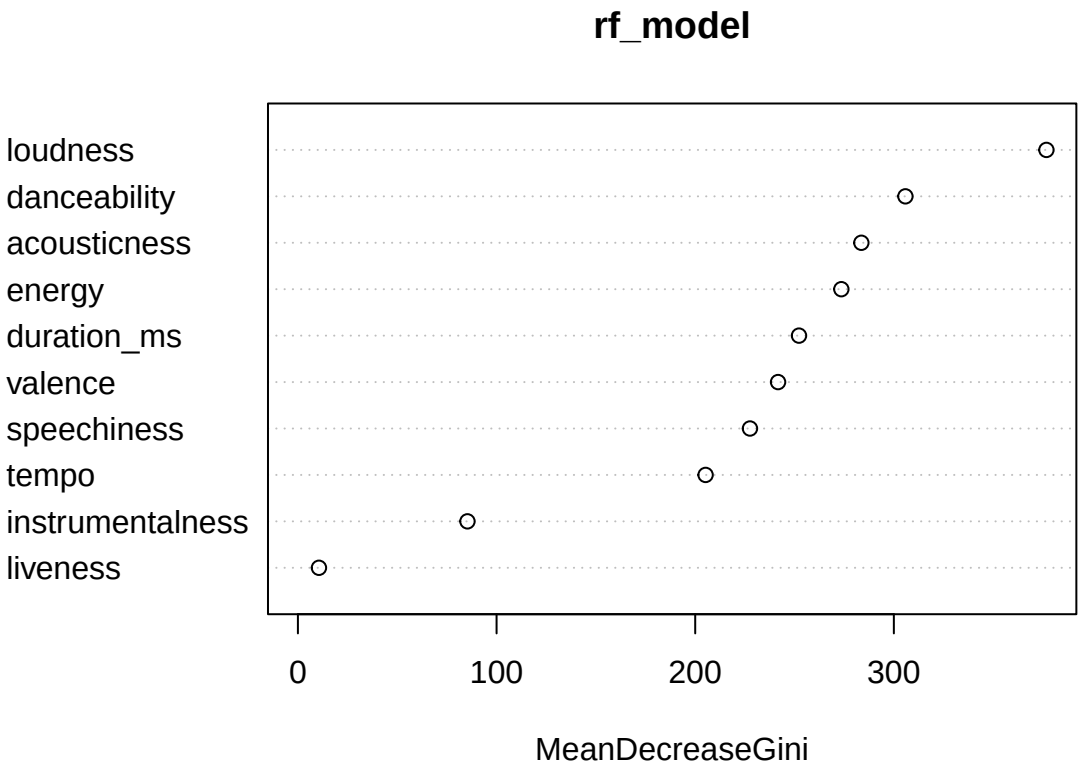
```r
confusionMatrix <- confusionMatrix(as.factor(test_predictions),
                                   as.factor(test_data$hit))
print(confusionMatrix)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction     0     1
##          0 35557   226
##          1 16478   771
##
##                Accuracy : 0.685
##                  95% CI : (0.681, 0.689)
##     No Information Rate : 0.9812
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.0508
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.6833
##             Specificity : 0.7733
##          Pos Pred Value : 0.9937
##          Neg Pred Value : 0.0447
##              Prevalence : 0.9812
##          Detection Rate : 0.6705
##    Detection Prevalence : 0.6747
##       Balanced Accuracy : 0.7283
##
##        'Positive' Class : 0
##
```

```r
feature_importance <- importance(rf_model)
print(feature_importance)
```

```
##                   MeanDecreaseGini
## acousticness             283.63718
## danceability             305.81068
## duration_ms              252.24177
## energy                   273.55901
## instrumentalness          85.33954
## liveness                  10.58994
## loudness                 376.80008
## speechiness              227.56660
## tempo                    205.22839
## valence                  241.71583
```

```r
# Plotting feature importance for better visualization
varImpPlot(rf_model)
```




Based on the result of the feature importance scorse of the random forest model, features such as loudness, acousticness, and danceability emerged as the most critical, with high importance scores. These features are followed by energy, speechiness, and others that also contribute notably but to a lesser extent. Conversely, liveness and instrumentalness displayed minimal impact, suggesting their limited role in the model's predictive accuracy.

Overall, the study demonstrates the application of machine learning techniques in predicting the popularity of songs on Spotify. Among the models tested, the Random Forest classifier is the most effective in both

regression and classification testing, achieving the lowest error rate and the highest accuracy. This suggests that for complex predictive problems like song popularity, Random Forest is valuable because the relationships between predictors and outcomes are not straightforward, and the model is very good at explaining non-linear data and avoiding overfitting. And from the results of random forest, we may conclude that loudness, acousticness, danceability, energy, speechiness, duration_ms, tempo, valence are the most signifanct predictors in our analysis. This project not only highlights the potential of advanced statistical techniques in the entertainment industry but also show one of the most effective strategies of prediction.

Reference Page: Dimolitsas, Ioannis, et al. (PDF) Spothitpy: A Study for ML-Based Song Hit Prediction Using Spotify, School of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece, www.researchgate.net/publication/367280936_SpotHitPy_A_Study_For_ML-Based_Song_Hit_Prediction_Using_Spotify. Accessed 29 Apr. 2024.

Lee, Junghyuk, and Jong-Seok Lee. Music Popularity: Metrics, Characteristics, and Audio-Based Prediction, 2018, arxiv.org/pdf/1812.00551.pdf.

Link to dataset: https://www.kaggle.com/datasets/zaheenhamidani/ultimate-spotify-tracks-db