Spotify Popularity Classification Analysis:

kite-luva

2025-05-24

Contents

```r
# Global options for code chunks
knitr::opts_chunk$set(echo = TRUE,
                      eval = TRUE,
                      message = FALSE,
                      warning = FALSE,
                      fig.align = 'center',
                      out.width = '80%',
                      fig_caption = TRUE
                      )
##############################==================================================
### CODE FOR AUC & ROC ###
##############################==================================================

# Load necessary libraries
library(tidyverse)
```

```
## -- 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   3.5.1     v tibble    3.2.1
## v lubridate 1.9.4     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 error
```

```r
library(caret)
```

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

```r
library(ranger)
library(lubridate)
library(pROC)
```

```
## Type 'citation("pROC")' for a citation.
##
## Attaching package: 'pROC'
##
## The following objects are masked from 'package:stats':
##
##     cov, smooth, var
```

```r
library(readr)
library(ggplot2)

# ---------------------------------------------------------------------------
# 1. Load & Clean the Data
# ---------------------------------------------------------------------------

# Load your dataset (adjust the file path accordingly)
spotify_charts_2024 <- read_csv("~/school docs/universal_top_spotify_songs.new.csv")
```

```
## Rows: 1750032 Columns: 25
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr   (5): spotify_id, name, artists, country, album_name
## dbl  (17): daily_rank, daily_movement, weekly_movement, popularity, duration...
## lgl   (1): is_explicit
## date  (2): snapshot_date, album_release_date
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
# Convert date columns and calculate difference in days
spotify_charts_2024 <- spotify_charts_2024 %>%
  mutate(snapshot_date = ymd(snapshot_date),
         album_release_date = ymd(album_release_date),
         days_out = as.numeric(snapshot_date - album_release_date))

# Remove duplicates based on the spotify_id column while retaining all columns
spotify_charts_2024 <- spotify_charts_2024 %>%
  distinct(spotify_id, .keep_all = TRUE)

# Remove unneeded columns
spotify_charts_2024 <- spotify_charts_2024 %>%
  select(-country, -snapshot_date, -name, -artists, -album_name, -album_release_date, -spotify_id)

# Convert 'is_explicit' (boolean) to integer
spotify_charts_2024$is_explicit <- as.integer(spotify_charts_2024$is_explicit)

# Handle missing values in numeric columns only
numeric_cols <- sapply(spotify_charts_2024, is.numeric)
```

```
spotify_charts_2024[numeric_cols] <- lapply(spotify_charts_2024[numeric_cols],
                                            function(x) ifelse(is.na(x), median(x, na.rm = TRUE), x))

# Standardize 'duration_ms' to minutes, then remove the original column
spotify_charts_2024 <- spotify_charts_2024 %>%
  mutate(duration_min = duration_ms / 60000) %>%
  select(-duration_ms)

# ------------------------------------------------------------------------------
# 2. Prepare Data for Classification
# ------------------------------------------------------------------------------

#remove popularity 0
spotify_charts_2024<- spotify_charts_2024 %>%
  filter(popularity != 0)

# Convert 'popularity' into a binary factor.
# This assigns popularity into two levels: "Low" and "High."
# 'make.names' ensures the levels are valid R variable names.
spotify_charts_2024 <- spotify_charts_2024 %>%
  mutate(popularity = ifelse(popularity >= 50, "High", "Low")) %>%
  mutate(popularity = make.names(popularity))

# Define feature columns (adjust these names if needed)
feature_columns <- c("daily_rank", "duration_min","daily_movement","weekly_movement",
                     "days_out", "is_explicit", "mode", "danceability", "energy", "loudness",
                     "speechiness", "acousticness", "instrumentalness", "time_signature",
                     "liveness", "valence", "key", "tempo")
View(spotify_charts_2024)
# Create a dataset with predictors and the target variable
class_data <- spotify_charts_2024 %>%
  select(all_of(feature_columns), popularity)

# Split the dataset into training (80%) and testing sets
set.seed(50)
trainIndex <- createDataPartition(class_data$popularity, p = 0.8, list = FALSE)
train_data  <- class_data[trainIndex, ]
test_data   <- class_data[-trainIndex, ]

head(train_data)
```

```
## # A tibble: 6 x 19
##   daily_rank duration_min daily_movement weekly_movement days_out is_explicit
##        <dbl>        <dbl>          <dbl>           <dbl>    <dbl>       <int>
## 1          1         4.19              0               1      191           0
## 2          3         2.96             -1               0       94           0
## 3          4         4.57             -1              -3      295           1
## 4          5         3.51              0               1      282           0
## 5          6         3.95              0               1       49           1
## 6          8         6.13              1               1       49           1
## # i 13 more variables: mode <dbl>, danceability <dbl>, energy <dbl>,
## #   loudness <dbl>, speechiness <dbl>, acousticness <dbl>,
## #   instrumentalness <dbl>, time_signature <dbl>, liveness <dbl>,
```

```
## #   valence <dbl>, key <dbl>, tempo <dbl>, popularity <chr>
```

```r
# ----------------------------------------------------------------------------
# 3. Train the Random Forest Classifier
# ----------------------------------------------------------------------------

# The 'twoClassSummary' along with 'metric = "ROC"' will use the ROC AUC for tuning.

rf_model_class <- train(popularity ~ .,
                        data = train_data,
                        method = "ranger",
                        trControl = trainControl(method = "cv",
                                                 number = 5,
                                                 classProbs = TRUE,
                                                 summaryFunction = twoClassSummary),
                        tuneGrid =  expand.grid(mtry = c(5, 7, 9, 11),
                                                min.node.size = c(1, 3, 5),
                                                splitrule = "gini"),
                        num.trees = 200,
                        metric = "ROC")
print(rf_model_class)
```

```
## Random Forest
##
## 16117 samples
##    18 predictor
##     2 classes: 'High', 'Low'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 12894, 12893, 12894, 12893, 12894
## Resampling results across tuning parameters:
##
##   mtry  min.node.size  ROC        Sens       Spec
##    5    1              0.8567575  0.9327640  0.5642857
##    5    3              0.8567663  0.9324161  0.5642857
##    5    5              0.8566084  0.9325032  0.5616883
##    7    1              0.8548563  0.9311115  0.5634199
##    7    3              0.8554429  0.9318942  0.5632035
##    7    5              0.8562770  0.9318074  0.5658009
##    9    1              0.8546391  0.9318074  0.5651515
##    9    3              0.8544226  0.9309376  0.5675325
##    9    5              0.8551210  0.9319814  0.5599567
##   11    1              0.8532295  0.9324162  0.5634199
##   11    3              0.8533833  0.9309376  0.5662338
##   11    5              0.8540395  0.9307637  0.5673160
##
## Tuning parameter 'splitrule' was held constant at a value of gini
## ROC was used to select the optimal model using the largest value.
## The final values used for the model were mtry = 5, splitrule = gini
##  and min.node.size = 3.
```

```r
# ------------------------------------------------------------------------------
# 4. Compute and Plot AUC & ROC Curve
# ------------------------------------------------------------------------------

# Generate predicted probabilities on the test set.
# We request probabilities (type = "prob") for both "Low" and "High" classes.
rf_pred_probs <- predict(rf_model_class, newdata = test_data, type = "prob")
head(rf_pred_probs)
```

```
##   High  Low
## 1 0.88 0.12
## 2 0.93 0.07
## 3 0.89 0.11
## 4 0.88 0.12
## 5 0.91 0.09
## 6 0.80 0.20
```

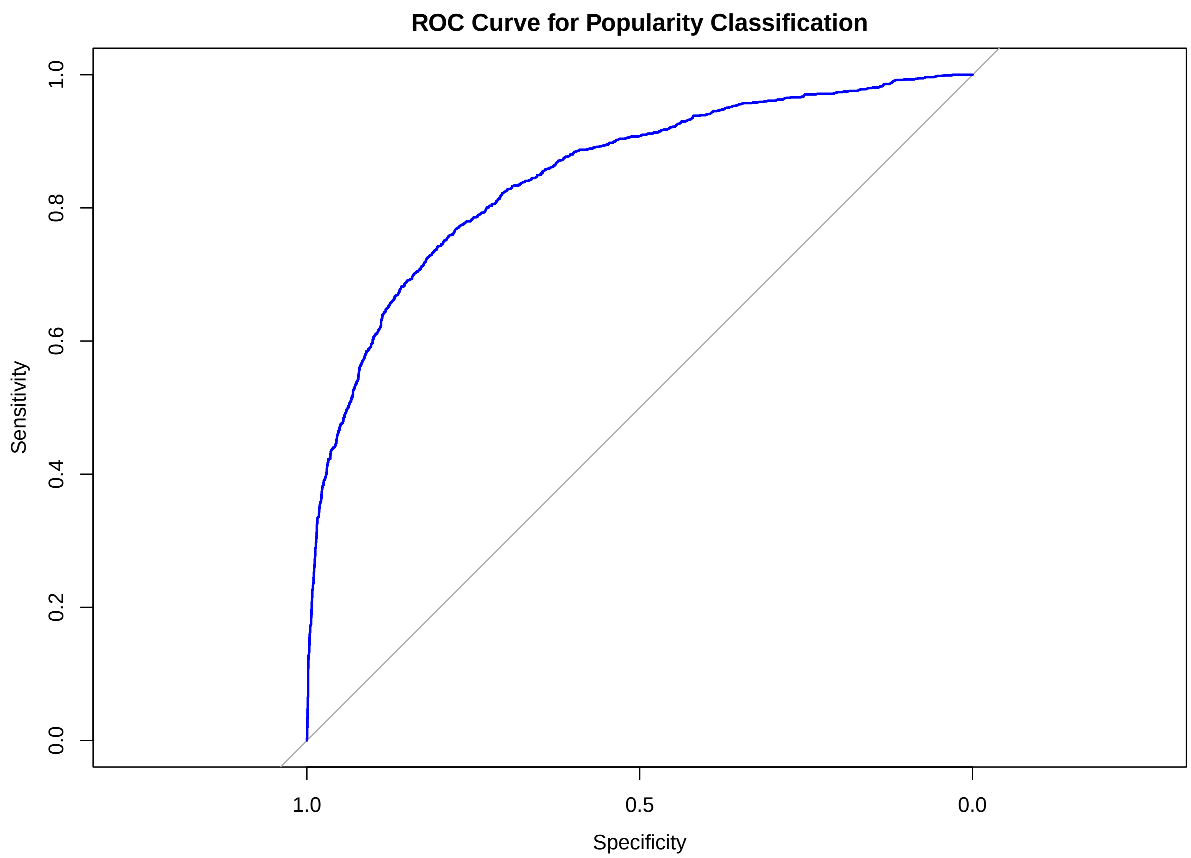
```r
prob_values <- rf_pred_probs$High
# Compute the ROC curve.
# Here, we consider the probability for the "High" class as the predictor.
roc_obj <- roc(response = test_data$popularity, predictor = rf_pred_probs[,"High"])
```

```
## Setting levels: control = High, case = Low
## Setting direction: controls > cases
```

```r
# Calculate the AUC and print it.
auc_value <- auc(roc_obj)
cat("AUC:", auc_value, "\n")
```

```
## AUC: 0.8484169
```

```r
# Plot the ROC curve.
plot(roc_obj, col = "blue", main = "ROC Curve for Popularity Classification")
```

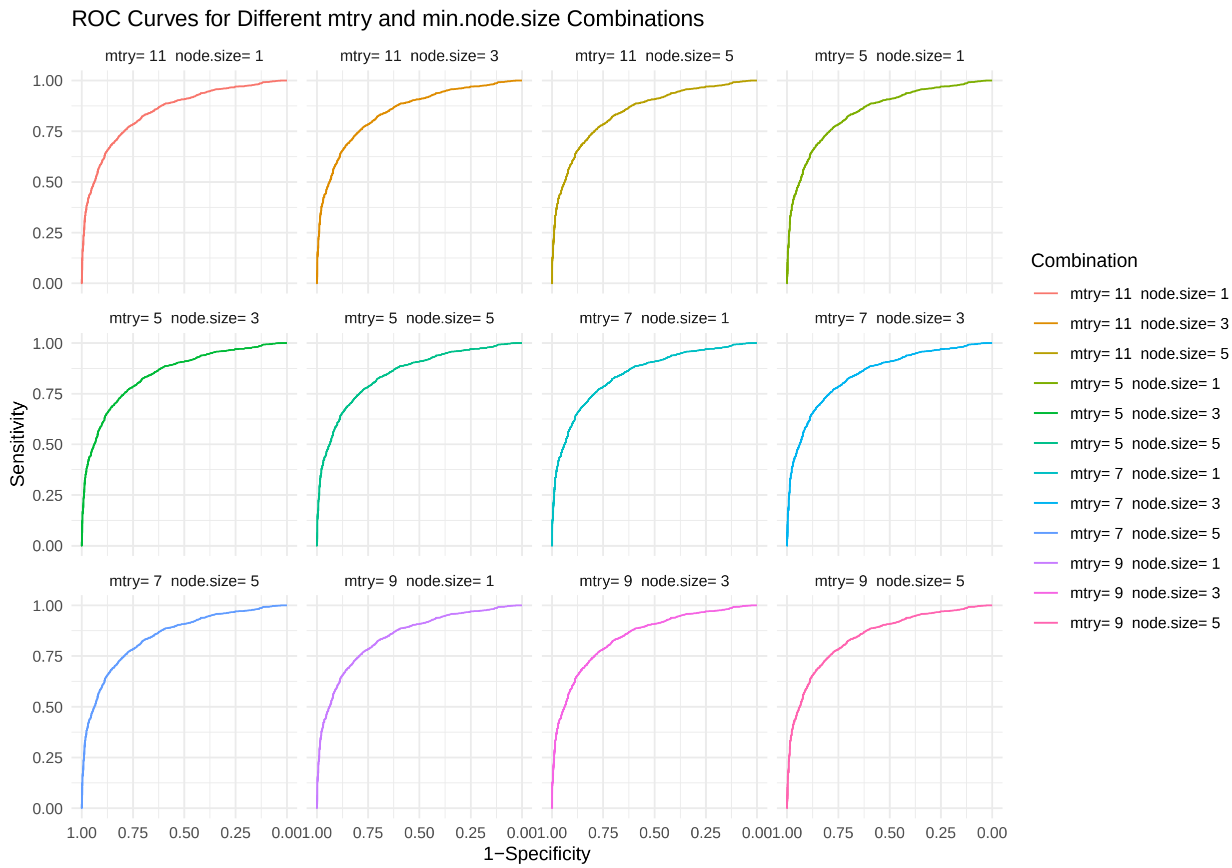



```r
#===========================================================================
###########################################################################
#===========================================================================
#Create an empty list to store ROC curves
roc_list <- list()
# Loop through each tuning parameter combination
for(i in 1:nrow(rf_model_class$results)){
  # extract parameters
  mtry_val <- rf_model_class$results$mtry[i]
  node_size_val <- rf_model_class$results$min.node.size[i]
  # make predictions
  predictions <- predict(rf_model_class, newdata = test_data, type= "prob")
  # Compute ROC curve
  roc_curve <- roc(test_data$popularity, predictions[, "High"])
  # Store the ROC curve with parameter labels
  roc_list[[paste("mtry=", mtry_val, " node.size=", node_size_val)]] <- roc_curve
}
```

```
## Setting levels: control = High, case = Low
## Setting direction: controls > cases
## Setting levels: control = High, case = Low
## Setting direction: controls > cases
## Setting levels: control = High, case = Low
## Setting direction: controls > cases
## Setting levels: control = High, case = Low
```

```
## Setting direction: controls > cases
## Setting levels: control = High, case = Low
## Setting direction: controls > cases
## Setting levels: control = High, case = Low
## Setting direction: controls > cases
## Setting levels: control = High, case = Low
## Setting direction: controls > cases
## Setting levels: control = High, case = Low
## Setting direction: controls > cases
## Setting levels: control = High, case = Low
## Setting direction: controls > cases
## Setting levels: control = High, case = Low
## Setting direction: controls > cases
## Setting levels: control = High, case = Low
## Setting direction: controls > cases
## Setting levels: control = High, case = Low
## Setting direction: controls > cases
```

```r
###
roc_data <- do.call(rbind, lapply(names(roc_list), function(label) {
  data.frame(
    Specificity = roc_list[[label]]$specificities,
    Sensitivity = roc_list[[label]]$sensitivities,
    Combination = label
  )
}))
# Plot ROC curves with facets for each combination
ggplot(roc_data, aes(x = Specificity, y = Sensitivity, color = Combination)) +
  geom_line() +
  labs(
    title = "ROC Curves for Different mtry and min.node.size Combinations",
    x = "1-Specificity", y = "Sensitivity"
  ) +
  theme_minimal() +
  facet_wrap(~Combination) +
  scale_x_reverse()
```




```
###------------------------------------------------------------------------------

####=================================================================######
# Define thresholds
upper_threshold_very_high <- 0.75
upper_threshold_high <- 0.5
lower_threshold_very_low <- 0.25

# Function to classify songs based on probability thresholds
classify_popularity <- function(prob_value) {
  if (prob_value >= upper_threshold_very_high) {
    return("Very High")
  } else if (prob_value >= upper_threshold_high) {
    return("High")
  } else if (prob_value < lower_threshold_very_low) {
    return("Very Low")
  } else if (prob_value < upper_threshold_high) {
    return("Low")
  } else {
    return("Uncertain")
  }
}
# Apply function to probability values
test_data$Predicted_Popularity <- sapply(rf_pred_probs$High, classify_popularity)
```

```r
# Combine actual popularity and predicted probabilities
results_df <- data.frame(
  Actual = test_data$popularity,
  High_Probability = rf_pred_probs$High,
  Low_Probability = rf_pred_probs$Low,
  Prediction_probability = test_data$Predicted_Popularity
)

# View the first few rows
print(head(results_df))
```

```
##   Actual High_Probability Low_Probability Prediction_probability
## 1   High             0.88            0.12              Very High
## 2   High             0.93            0.07              Very High
## 3   High             0.89            0.11              Very High
## 4   High             0.88            0.12              Very High
## 5   High             0.91            0.09              Very High
## 6   High             0.80            0.20              Very High
```

```r
###==================================================================================================+++
```