spotify popularity

kite-luva

2025-05-22

#================================# ## RF, XGBoost, and GBM PREDICTIVE CODES ## #================================#

library(skimr) library(tidyverse) library(dplyr) library(corrplot) library(caret) library(randomForest) library(ggplot2) library(writexl) library(openxlsx) library(rlang) library(GGally) library(grid) library(xgboost) library(gbm) library(recipes) library(lubridate)

Load the dataset

spotify_charts_2024 <- read_csv("~/school docs/universal_top_spotify_songs.new.csv")

###———————————— ##DATA CLEANING ###————————————-

market counts for each song

spotify_charts_2024 <- spotify_charts_2024 %>% group_by(spotify_id) %>% mutate(market_count = n_distinct(country, na.rm = TRUE)) %>% ungroup()

remove all duplicates, pick the most popular song(country)

spotify_charts_2024 <- spotify_charts_2024 %>% group_by(spotify_id) %>% mutate(other_charted_countries = paste(country[!duplicated(country)], collapse = ",")) %>% slice_max(order_by = popularity, n = 1, with_ties = FALSE) %>% ungroup()

Function to count the number of artists in each row

spotify_charts_2024$artist_count <- -sapply(strsplit(spotify_charts_2024artists, ","), length)

Convert character date columns to Date objects using mdy()

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))

change boolean into integers

spotify_charts_2024$is_explicit <- as.integer(spotify_charts_2024$is_explicit)

Standardize duration_ms (convert to minutes)

spotify_charts_2024$duration_ms <- spotify_charts_2024$duration_ms / 60000 colnames(spotify_charts_2024)[colnames(spoti
== "duration_ms"] <- "duration_min"

Remove unneeded columns

spotify_modelling <- spotify_charts_2024 %>% select(-country, -other_charted_countries, -snapshot_date,
-name, -artists, -album_name, -album_release_date, -spotify_id)

check for missing values

colSums(is.na(spotify_modelling))

#Handle missing values spotify_modelling <- spotify_modelling %>% mutate_all(~ifelse(is.na(.), mean(.,
na.rm = TRUE), .))

arrange the columns

spotify_modelling <- spotify_modelling %>% select(popularity, days_out, artist_count, market_count,
daily_rank, daily_movement, weekly_movement, duration_min, is_explicit, mode, danceability, energy,
loudness, speechiness, acousticness, instrumentalness, liveness, valence, tempo, key, time_signature)

#remove popularity 0 spotify_modelling <- spotify_modelling %>% filter(popularity != 0) %>% ar-
range(desc(popularity))

#———————————— ### DESCRIPTIVE STATISTICS #————————————

Function to compute basic statistics

spotify_stats <- function(column) { stats <- c(Mean = mean(column, na.rm = TRUE), Median = me-
dian(column, na.rm = TRUE), SD = sd(column, na.rm = TRUE), Variance = var(column, na.rm = TRUE),
IQR = IQR(column, na.rm = TRUE)) return(stats) }

Loop through columns and compute statistics

stats_results <- lapply(spotify_modelling, spotify_stats) names(stats_results) <- colnames(spotify_modelling)

Convert the list of statistics to a data frame for better printing

stats_table <- as.data.frame(stats_results) print(stats_table)

Generate the four scatter plot matrices

plot1 <- ggpairs(spotify_modelling, columns = 1:5) plot2 <- ggpairs(spotify_modelling, columns = 6:10) plot3 <- ggpairs(spotify_modelling, columns = 11:15) plot4 <- ggpairs(spotify_modelling, columns = 16:20)

Create a layout matrix

layout_matrix <- matrix(c(1,2,3,4), nrow = 2, byrow = TRUE)

Open a graphics device and use grid.newpage() to manually arrange plots

grid.newpage() pushViewport(viewport(layout = grid.layout(nrow = 2, ncol = 2)))

Print each ggmatrix object in its respective location

print(plot1, vp = viewport(layout.pos.row = 1, layout.pos.col = 1)) print(plot2, vp = viewport(layout.pos.row = 1, layout.pos.col = 2)) print(plot3, vp = viewport(layout.pos.row = 2, layout.pos.col = 1)) print(plot4, vp = viewport(layout.pos.row = 2, layout.pos.col = 2))

check popularity distribution

ggplot(data=spotify_modelling)+ geom_bar(mapping=aes(x=popularity, fill = "skyblue"))+ ggtitle('popularity dist. after cleaning')

#————————————- ### CORRELATION #————————————- # Select the audio feature columns audio_features <- spotify_modelling %>% select(popularity, market_count, daily_rank, daily_movement, weekly_movement, days_out, artist_count, duration_min, is_explicit, mode, danceability, energy, loudness, speechiness, acousticness, instrumentalness, time_signature, liveness, valence, key, tempo) # Compute correlation matrix correlation_matrix <- cor(audio_features, use = "complete.obs") # Plot heatmap with correlation values corrplot(correlation_matrix, method = "color", tl.col = "black", tl.srt = 45, type = "upper", addCoef.col = "black", number.cex = 0.7, number.digits = 3, main = "Correlation Heatmap", mar = c(0, 0, 2, 0))

#———————————— ### CARET DATA PARTITION #————————————

Define feature columns and target variable

target_column <- "popularity"

X <- spotify_modelling %>% select(-popularity) y <- spotify_modelling$popularity

Split the data into training and testing sets

set.seed(50) trainIndex <- createDataPartition(y, p = 0.8, list = FALSE) X_train <- X[trainIndex,] X_test <- X[-trainIndex,] y_train <- y[trainIndex] y_test <- y[-trainIndex]

train.dframe_features <- as.data.frame(X_train) test.dframe_features <- as.data.frame(X_test) train.dframe_target <- as.data.frame(y_train) test.dframe_target <- as.data.frame(y_test) train.dframe <- cbind(train.dframe_features, y_train) test.dframe <- cbind(test.dframe_features, y_test)

testing popularity distribution for training data

ggplot(data=train.dframe)+ geom_bar(mapping=aes(x=y_train),fill = "skyblue")+ ggtitle('popularity dist. for training data'))

testing popularity distribution for testing data

ggplot(data=test.dframe)+ geom_bar(mapping=aes(x=y_test) ,fill = "skyblue")+ ggtitle('popularity dist. for testing data'))

#———————————— ## DATA PREPARATION FOR MODELING #————————————

Create a preprocessing recipe

preprocess_recipe <- recipe(x = train.dframe_features, y = y_train) %>% # Center and scale numeric predictors step_center(all_numeric(), -all_outcomes()) %>% step_scale(all_numeric(), -all_outcomes()) %>% # Handle categorical variables step_dummy(all_nominal(), -all_outcomes())

Apply preprocessing

prep_recipe <- prep(preprocess_recipe, training = train.dframe_features, y = y_train) train_processed <- bake(prep_recipe, new_data = train.dframe_features) test_processed <- bake(prep_recipe, new_data = test.dframe_features)

print("Processed Training Data (First few rows):") print(head(train_processed)) print("Processed Testing Data (First few rows):") print(head(test_processed))

Combine processed features with the target variable

train_processed <- cbind(train_processed, y_train) test_processed <- cbind(test_processed, y_test) colnames(train_processed)[ncol(train_processed)] <- "y_train" colnames(test_processed)[ncol(test_processed)] <- "y_test"

#————————————- ### RANDOM FOREST MODEL #————————————

Hyperparameter tuning using caret

control_rf <- trainControl(method = “cv”, number = 5) grid_rf <- expand.grid(mtry = c(5, 7, 9, 11), min.node.size = c(3, 5, 7), splitrule = “variance”) rf_model <- train(y_train ~ ., data = train_processed, method = “ranger”, trControl = control_rf, tuneGrid = grid_rf, importance = “permutation”, num.trees = 200) # model results print(rf_model$results) $print(rf_model$bestTune) importance_matrix_rf <- varImp(rf_model)

save RF model

saveRDS(rf_model, “rf_model.rds”)

Predictions on test set

rf_pred <- predict(rf_model, newdata= test_processed)

Evaluate model performance

MAE_rf <- mean(abs(rf_pred - test_processed$y_test)) $RMSE_rf <- sqrt(mean((rf_pred - test_processed$y_test)^2))$ R_squared_rf <- cor(rf_pred, test_processed$y_test)^2

print(“Random Forest - Mean Absolute Error (MAE):”, MAE_rf) print(“Random Forest - Root Mean Squared Error (RMSE):”, RMSE_rf) print(“Random Forest - R-squared (R²):”, R_squared_rf)

Convert the importance matrix to a data frame for easier handling

importance_df_rf <- data.frame(Feature = rownames(importance_matrix_rf$importance), $Importance = importance_matrix_rf$importance[, 1] # Access the first column of importance) # sort by importance: sorted_importance_df_rf <- importance_df_rf[order(importance_df_rf$Importance, decreasing = TRUE),] print(sorted_importance_df_rf)

#————————————- ### XGBOOST MODEL #————————————

Prepare data for XGBoost

dtrain <- xgb.DMatrix(data = as.matrix(train_processed %>% select(-y_train)), label = train_processed$y_train) $dtest <- xgb.DMatrix(data = as.matrix(test_processed$y_test)$

Hyperparameter tuning using caret

control_xgb <- trainControl(method = “cv”, number = 5) grid_xgb <- expand.grid(nrounds = c(100, 200), max_depth = c(3, 5, 7), eta = c(0.01, 0.05, 0.1), gamma = 0, colsample_bytree = c(0.7, 0.9), min_child_weight = 1, subsample = 0.8)

xgb_model <- train(y_train ~ ., data = train_processed, method = “xgbTree”, trControl = control_xgb, tuneGrid = grid_xgb, verbose = FALSE)

Model results

print(xgb_model$results)print($xgb_model$bestTune) importance_matrix_xgb <- varImp(xgb_model)

save XGBoost model

saveRDS(xgb_model, "xgb_model.rds")

Predictions on test set

xgb_pred <- predict(xgb_model, newdata = test_processed %>% select(-y_test))

Evaluate model performance

MAE_xgb <- mean(abs(xgb_pred - test_processed$y_test))$RMSE_xgb <- sqrt(mean((xgb_pred - test_processed$y_test)^2)) R_squared_xgb <- cor(xgb_pred, test_processed$y_test)^2

print("XGBoost - Mean Absolute Error (MAE):", MAE_xgb) print("XGBoost - Root Mean Squared Error (RMSE):", RMSE_xgb) print("XGBoost - R-squared (R^2):", R_squared_xgb)

Convert the importance matrix to a data frame for easier handling

importance_df_xgb <- data.frame(Feature = rownames(importance_matrix_xgb$importance), Importance = importance_matrix_xgb$importance[, 1] # Access the first column of importance) # sort by importance: sorted_importance_df_xgb <- importance_df_xgb[order(importance_df_xgb$Importance, decreasing = TRUE),] print(sorted_importance_df_xgb)

#————————————- ### GBM MODEL #————————————

Hyperparameter tuning using caret

control_gbm <- trainControl(method = "cv", number = 5) grid_gbm <- expand.grid(n.trees = c(100, 200), interaction.depth = c(3, 5), shrinkage = c(0.01, 0.05), n.minobsinnode = c(10, 20))

gbm_model <- train(y_train ~ ., data = train_processed, method = "gbm", trControl = control_gbm, tuneGrid = grid_gbm, verbose = FALSE)

Model results

print(gbm_model$results)print($gbm_model$bestTune) importance_matrix_gbm <- varImp(gbm_model)

save GBM model

saveRDS(gbm_model, "gbm_model.rds")

Predictions on test set

gbm_pred <- predict(gbm_model, newdata = test_processed %>% select(-y_test))

Evaluate model performance

MAE_gbm <- mean(abs(gbm_pred - test_processed$y_test))$RMSE_gbm <- -sqrt(mean((gbm_pred - test_processed$y_test)^2))$ R_squared_gbm <- cor(gbm_pred, test_processed$y_test)^2

print("Gradient Boosting Machine - Mean Absolute Error (MAE):", MAE_gbm) print("Gradient Boosting Machine - Root Mean Squared Error (RMSE):", RMSE_gbm) print("Gradient Boosting Machine - R-squared (R^2):", R_squared_gbm)

Convert the importance matrix to a data frame for easier handling

importance_df_gbm <- data.frame(Feature = rownames(importance_matrix_gbm$importance), $Importance = importance_matrix_gbm$importance[, 1]) # sort by importance: sorted_importance_df_gbm <- importance_df_gbm[order(importance_df_gbm$Importance, decreasing = TRUE),] print(sorted_importance_df_gbm)

#————————— ## MODEL COMPARISON: #————————-

Create a data frame to compare model performance

model_comparison <- data.frame(Model = c("Random Forest", "XGBoost", "GBM"), MAE = c(MAE_rf, MAE_xgb, MAE_gbm), RMSE = c(RMSE_rf, RMSE_xgb, RMSE_gbm), R_squared = c(R_squared_rf, R_squared_xgb, R_squared_gbm))

print("Model Performance Comparison:") print(model_comparison)

Create a bar plot for R-squared comparison

ggplot(model_comparison, aes(x = Model, y = R_squared, fill = Model)) + geom_bar(stat = "identity") + geom_text(aes(label = round(R_squared, 3)), vjust = -0.3) + labs(title = "R-squared Comparison of Models", y = "R-squared") + theme_minimal()

Create a bar plot for RMSE comparison

ggplot(model_comparison, aes(x = Model, y = RMSE, fill = Model)) + geom_bar(stat = "identity") + geom_text(aes(label = round(RMSE, 3)), vjust = -0.3) + labs(title = "RMSE Comparison of Models", y = "RMSE") + theme_minimal()

Create a bar plot for MAE comparison

ggplot(model_comparison, aes(x = Model, y = MAE, fill = Model)) + geom_bar(stat = "identity") + geom_text(aes(label = round(MAE, 3)), vjust = -0.3) + labs(title = "MAE Comparison of Models", y = "MAE") + theme_minimal()

#————————————— ## ACTUAL VS PREDICTED PLOTS: #————————————

Random Forest

plot_data_rf <- data.frame(Actual = test_processed$y_test, Predicted = rf_pred) ggplot(plot_data_rf, aes(x = Actual, y = Predicted)) + geom_point(color = “blue”, alpha = 0.6) + geom_smooth(method = ‘lm’, col=‘red’) + labs(title = “Actual vs Predicted Popularity (Random Forest)”, x = “Actual Popularity”, y = “Predicted Popularity”) + theme_minimal()

XGBoost

plot_data_xgb <- data.frame(Actual = test_processed$y_test, Predicted = xgb_pred) ggplot(plot_data_xgb, aes(x = Actual, y = Predicted)) + geom_point(color = “green”, alpha = 0.6) + geom_smooth(method = ‘lm’, col=‘red’) + labs(title = “Actual vs Predicted Popularity (XGBoost)”, x = “Actual Popularity”, y = “Predicted Popularity”) + theme_minimal()

GBM

plot_data_gbm <- data.frame(Actual = test_processed$y_test, Predicted = gbm_pred) ggplot(plot_data_gbm, aes(x = Actual, y = Predicted)) + geom_point(color = “purple”, alpha = 0.6) + geom_smooth(method = ‘lm’, col=‘red’) + labs(title = “Actual vs Predicted Popularity (GBM)”, x = “Actual Popularity”, y = “Predicted Popularity”) + theme_minimal()