SONG POPULARITY PREDICTOR

A PROJECT REPORT

Submitted by

Sparsh Srivastava (23BAI10631)
Romsha Wadhwa (23BAI10239)
Mohd. Ashhar Khan (23BAI11297)
Vivek Srivastava (23BAI10127)
Shreshth Tiwari (23BAI10149)
Rivi Gaur (23BAI11301)

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING
(ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)

VIT®
BHOPAL
www.vitbhopal.ac.in

SCHOOL OF COMPUTING SCIENCE ENGINEERING AND ARTIFICIAL INTELLIGENCE

VIT BHOPAL UNIVERSITY

KOTHRIKALAN, SEHORE
MADHYA PRADESH - 466114

APRIL 2025

BONAFIDE CERTIFICATE

Certified that this project report titled **"Song Popularity Predictor"** is the bonafide work of "**Sparsh Srivastava (23BAI10631), Romsha Wadhwa (23BAI10239), Mohd. Ashhar Khan (23BAI11297), Vivek Srivastava (23BAI10127), Shreshth Tiwari (23BAI10149), Rivi Gaur (23BAI11301)**" who carried out the project work under my supervision. Certified further that to the best of my knowledge the work reported at this time does not form part of any other project/research work based on which a degree or award was conferred on an earlier occasion on this or any other candidate.

PROGRAM CHAIR
Dr. Pradeep Mishra
School of Computing Science Engineering and Artificial Intelligence

VIT BHOPAL UNIVERSITY

PROJECT GUIDE
Dr. Geeta Singh
School of Computing Science Engineering and Artificial Intelligence

VIT BHOPAL UNIVERSITY

The Project Exhibition I Examination is held on

ACKNOWLEDGEMENT

First and foremost I would like to thank the Lord Almighty for His presence and immense blessings throughout the project work.

I wish to express my heartfelt gratitude to **Dr. Pradeep Kumar Mishra ,** Head of the Department, School of Computing Science and Artificial Intelligence for much of his valuable support and encouragement in carrying out this work.

I would like to thank my internal guide **Dr. Geeta Singh** ,for continually guiding and actively participating in my project, giving valuable suggestions to complete the project work.

I would like to thank all the technical and teaching staff of the **School of Computing Science and Artificial Intelligence** ,who extended directly or indirectly all support.

Last, but not least, I am deeply indebted to my parents who have been the greatest support while I worked day and night for the project to make it a success.

LIST OF ABBREVIATIONS

`df` – DataFrame (a table-like data structure from pandas)

`X` – Features (independent variables used for training)

`y` – Target variable (in this case, the song popularity)

`X_train, X_test` – Feature data split for training and testing

`y_train, y_test` – Target values split for training and testing

`r2` – R^2 Score (Coefficient of Determination – measures model accuracy)

`xgb` – XGBoost (Extreme Gradient Boosting – a machine learning algorithm)

`plt` – Matplotlib Pyplot (used for plotting graphs)

`sns` – Seaborn (statistical visualization library)

`st` – Streamlit (library for building interactive web apps)

`r (in r"...")` – Raw string in Python (used for file paths to prevent escape characters)

`NaN` – Not a Number (represents missing/undefined values in data)

`CSV` – Comma Separated Values (file format for datasets)

`ML` – Machine Learning

`BPM` – Beats Per Minute (tempo of a song)

`dB` – Decibels (unit for loudness)

`map()` – Python function used to transform or map values in a series

`get_dummies()` – Converts categorical data into numerical one-hot encoding

`make_pipeline()` – Combines preprocessing and model steps into a single object

`train_test_split()` – Splits data into training and testing subsets

`PolynomialFeatures` – Generates polynomial features (e.g., x^2, x^3)

`QuantileTransformer` – Transforms features to follow a normal distribution

`PowerTransformer` – Applies a power transformation to make data more Gaussian

`Ridge` – Ridge Regression (a type of linear regression with L2 regularization)

`DecisionTreeRegressor` – A decision tree-based regression model

`RandomForestRegressor` – Ensemble model using multiple decision trees

`XGBRegressor` – XGBoost regression model

`iloc` – Integer-location based indexing used to access rows by position

`argmin()` – Returns the index of the minimum value in an array

`astype()` – Converts data to a specific type (e.g., float)

`values` – Returns the underlying numpy array of a DataFrame/Series

`apply()` – Applies a function across DataFrame elements

`drop_duplicate-`Streamlit UI widgets for user input

`s()` – Removes duplicate rows

`inplace=True` – Modifies the original DataFrame without returning a new one

`reset_index()` – Resets the index of the DataFrame

`pd.to_numeric()` – Converts data to numeric, coercing errors if needed

`st.slider() / st.radio() / st.selectbox()` – Streamlit UI widgets for user input

List of tables

Model	Training Time (sec)	Accuracy Scaled	MAE Normalized	Remarks
Decision Tree	6.50	86.53%	1.56%	Fastest and Most Accurate
Random Forest	444.25	85.08%	4.75%	High Accuracy but Very Slow
Linear Regression	15.79	76.96%	11.39%	Fast but Least Accurate
XG Boost	144.24	80.37%	5.71%	Balanced Performance

LIST OF FIGURES AND GRAPHS

ABSTRACT

Purpose

The main goal of this project is to create a web-based application that helps users analyze Spotify songs and predict how popular a song might become based on its audio features. It also recommends the closest matching song from the dataset based on the features provided by the user. This tool is useful for music creators, producers, and enthusiasts who want to understand what makes a song popular.

Methodology

The project uses the Spotify Features dataset, which contains various audio-related attributes of songs such as danceability, energy, tempo, and loudness. The application is developed using Python and Streamlit for the user interface. Data is cleaned and preprocessed by removing duplicates, converting text-based columns into numeric format, and generating dummy variables for categorical data.Several machine learning models are used to predict song popularity, including:

- Decision Tree Regressor
- Random Forest Regressor
- Linear Regression with Polynomial Features and Ridge Regularization
- XGBoost Regressor

These models are trained using the `train_test_split` method and evaluated using the R^2 (R-squared) score to check prediction accuracy. The app also includes tools to display accuracy results, plot comparison graphs, show a correlation matrix, and make real-time predictions based on user input.

Findings

The analysis shows that machine learning models can effectively predict the popularity of a song using its audio features. Among all the models tested, **XGBoost** and **Random Forest** performed best in terms of accuracy. The interactive app allows users to test their own song features and view the predicted popularity score. Additionally, the app finds and displays the closest real song from the dataset based on the input features and genre, making the application more engaging and insightful for users.

TABLE OF CONTENTS

CHAPTER 1:PROJECT DESCRIPTION AND OUTLINE

1.1 Introduction

Music has become more digital and data-driven than ever before. Platforms like Spotify collect various details about songs — such as how loud, energetic, or danceable a song is — and also track how many people listen to them. Using this information, it's now possible to apply machine learning techniques to predict which songs are likely to become popular. This project is focused on using these audio features and machine learning algorithms to build a model that can predict a song's popularity.

1.2 Motivation for the Work

The music industry is highly competitive, with thousands of songs released every day. However, only a few songs become hits. Understanding why some songs become popular while others don't is valuable for musicians, producers, and streaming platforms. Using machine learning to analyze song features can help in forecasting popularity. This not only supports marketing decisions but also enhances user experience by improving recommendation systems.

1.3 About Introduction to the Project (Including Techniques)

This project makes use of a dataset that contains audio features of songs available on Spotify. Each song has features like tempo, energy, loudness, danceability, etc., and a popularity score assigned by Spotify. We aim to train machine learning models using this dataset.
The techniques used in this project include:

- **Linear Regression**: A basic method that models the relationship between input features and song popularity.

- **Polynomial Regression**: Captures more complex patterns by using powers of input variables.
- **Decision Tree**: A tree-like model that makes predictions based on feature conditions.
- **Random Forest**: A combination of multiple decision trees for better accuracy.
- **XGBoost**: A powerful boosting algorithm that improves prediction performance.

We compare these models to find out which gives the best accuracy in predicting song popularity.

1.5 Problem Statement

To design and implement a machine learning model that can predict the popularity of a song based on its audio features. The model should be trained using historical data and should be able to predict a popularity score (on a scale of 0–100) for any new or unknown song.

1.6 Objective of the Work

- To collect and preprocess Spotify song data with multiple features.
- To apply and compare various regression models.
- To evaluate model performance using metrics like RMSE and R^2 Score.
- To determine which model best predicts song popularity.
- To analyze the importance of different song features in determining popularity.
- To create a final report summarizing the approach and results.

1.7 Organization of the Project

The project report is organized as follows:

- **Chapter 1**: Gives an overview, motivation, objectives, and techniques used in the project.

- **Chapter 2**: Discusses basic concepts of machine learning and the algorithms used.
- **Chapter 3**: Covers methodology — data collection, preprocessing, model training, and evaluation.
- **Chapter 4**: Presents results, comparisons, and performance metrics of different models.
- **Chapter 5**: Concludes the project with findings, limitations, and future scope.

1.8 Summary

In this chapter, we introduced the aim of the project, its motivation, and the problem we are trying to solve — predicting song popularity using machine learning. We also briefly described the techniques used and the overall structure of the report. This sets the foundation for the upcoming chapters, where each part of the work is explained in detail.

CHAPTER 2: RELATED WORK INVESTIGATION

2.1 Introduction

Before building any machine learning system, it's important to study what has already been done in the field. This chapter investigates the core area of the project, reviews various existing approaches for predicting music popularity, and highlights their strengths and limitations. This helps us identify gaps and design a better system.

2.2 Core Area of the Project

This project focuses on analyzing Spotify songs based on their audio features and predicting how popular a song could become using machine learning. The application is built using Python and Streamlit, allowing users to interact with the dataset and models in a user-friendly way. The main goal is to predict the **popularity** score of a song based on characteristics like danceability, energy, tempo, valence, and more. The project loads a dataset of Spotify tracks, cleans and preprocesses the data, and trains various regression models to evaluate prediction accuracy. Users can input new song characteristics and receive predictions along with the closest existing song match from the dataset. This helps in understanding what makes a song popular and how different features contribute to its success.

2.3 Existing Approaches/Methods

2.3.1 Approach/Method - 1: Linear Regression (with Transformations)

Linear Regression is used with a pipeline that includes PowerTransformer, QuantileTransformer, PolynomialFeatures, and Ridge regression to improve model performance on non-linear data. These transformations make the input features more normally distributed and allow the model to capture complex relationships between audio features and popularity. Ridge regression is used to prevent overfitting by penalizing large weights. This combination helps the model learn from diverse patterns in the Spotify dataset. In the code, this model is stored in linear_reg_model and trained inside the train_dataset()

function. While simple, this method sets a solid baseline for performance comparison.

2.3.2 Approach/Method - 2: Random Forest Regressor

Random Forest is an ensemble learning method that builds multiple decision trees and merges their outputs for better accuracy. It reduces overfitting by averaging the results from many trees trained on different parts of the data. In the app, it is implemented using `RandomForestRegressor()` and trained with the fit() method. This model handles large feature spaces well and works effectively with both categorical and continuous data. It is robust to noise and missing values, which suits this project's dataset after preprocessing. Although it is slower than linear models, it gives better accuracy and generalization.

2.3.3 Approach/Method - 3: XGBoost Regressor

XGBoost is a powerful and efficient gradient boosting algorithm that builds decision trees sequentially and improves performance by focusing on previous errors. In our code, the `XGBRegressor` is configured with parameters like learning rate, tree depth, subsampling ratio, and regularization to avoid overfitting. It performs well on structured data, making it ideal for this kind of numerical dataset. It is slower to train but often produces the most accurate predictions. The training happens in the same loop in the `train_dataset()` function. XGBoost is often used in competitions because of its high accuracy and flexibility.

2.3.4 Approach/Method - 4: Decision Tree Regressor

Decision Tree is a simple model that splits the dataset based on feature values to make predictions. It is easy to interpret and visualize but may overfit on small datasets. In our project, `DecisionTreeRegressor()` is used to create this model and trained in the same loop as the other models. The tree learns rules from data like "if danceability > 0.5 and energy > 0.7, then popularity = 75." It works well when the data has clear, rule-based splits. However, its accuracy is often lower than ensemble models like Random Forest or XGBoost. It's a good starting point to understand data structure.

2.4 Pros and Cons of the Stated Approaches/Methods

1. **Linear Regression with Transformations**

   Pros:- This model is very simple to understand and quick to train, making it an excellent choice for building a baseline and comparing

results.
Pros:- It performs better when used with transformations like PowerTransformer, QuantileTransformer, and PolynomialFeatures, which help it learn complex patterns.
Cons:- However, it struggles with capturing complex, non-linear relationships in the data unless a lot of manual feature engineering is done.
Cons:- If the relationship between the input features and the target is not linear, the model may underfit and give poor accuracy.

2. Random Forest Regressor

Pros:- Random Forest provides better accuracy by combining the output of many decision trees and is good at handling noisy or missing data.
Pros:- It reduces overfitting by averaging the results of different trees trained on subsets of the data, giving a more stable prediction.
Cons:- Training takes more time compared to linear models, especially with large datasets or high feature dimensions.
Cons:- The model becomes harder to interpret, as it's not easy to understand which tree or rule led to a specific prediction.

3. XGBoost Regressor

Pros:- XGBoost is one of the most accurate and powerful models available and can handle overfitting through advanced regularization techniques.
Pros:- It works well with large datasets, automatically handles missing values, and can capture complex patterns in the data.
Cons:- However, the model is quite complex to configure, and proper tuning of parameters is required for best results, which takes time.
Cons:- Understanding how boosting works and how each tree adds to the prediction requires more in-depth knowledge from the user.

4. Decision Tree Regressor

Pros:- Decision Tree models are very simple to visualize and explain, using straightforward if-else conditions to split data into prediction paths.
Pros:- They are quick to train and test, and work well on small to medium-sized datasets where rules can be easily extracted from the data.

Cons:- But they tend to overfit on small datasets if not pruned properly, which leads to poor generalization on new data.
Cons:- A single decision tree doesn't perform as well as ensemble models like Random Forest or XGBoost and is more sensitive to noise.

2.5 Issues/Observations from Investigation

1. **Data Format and Cleaning Challenges:**
   The original dataset contains unnecessary columns and some text data which need to be converted. We dropped non-numeric columns like `genre` and converted categorical fields like `mode` and `time_signature.` Without these steps, the models would fail to train properly. Time signature values were one-hot encoded for model compatibility. All these preprocessing steps are handled inside the `load_dataset()` function. This proves that a clean dataset is essential for model accuracy.
2. **Handling Missing or Corrupted Data:**
   Some rows in the dataset have missing or invalid entries (NaN or non-numeric), especially after transformation. These are filtered out in the training phase using `X.apply(pd.to_numeric)` and by dropping rows that don't convert cleanly. Without these filters, models would crash or give invalid predictions. It's important to ensure numerical consistency across features. This step is included right before the train-test split in the `train_dataset()` function.
3. **Model Accuracy Might Be Misleading:**
   In the `display_accuracy()` function, accuracy is calculated using R^2 score. R^2 is not always reliable alone—especially for comparing across different model types. It's better to use multiple metrics or cross-validation.
4. **Input Feature Mismatch During Prediction:**
   When users input song features in the "Predict Popularity" section, the code checks if any expected features are missing and fills them with 0. This is a smart way to avoid crashes. However, predicting with zeros may lead to inaccurate results if important features are missing. This behavior is controlled in the `predict_popularity()` function, using a for-loop that adds default values to missing columns.

5. **Closest Song Matching Isn't Always Accurate:**
   After predicting popularity, the app tries to find the most similar song using Euclidean distance. If the selected genre has no songs left after filtering, no match is shown. This is a helpful feature but limited by how similar songs are defined (only based on numbers, not actual musical characteristics). Improving this with cosine similarity or feature weighting could improve match accuracy.

6. **Streamlit App Requires Sequential Usage:**
   The app works step-by-step—the dataset must be loaded first, then models trained, then prediction. If a user skips one step (e.g., tries to predict before training), the app shows an error. This flow control is handled using `st.session_state`. Although effective, new users might get confused without a guided interface or progress steps. Improving the user experience would make it more intuitive.

2.6 Summary

In this chapter, we analyzed and compared several machine learning approaches for predicting music popularity based on features like danceability, energy, acousticness, and tempo from the Spotify dataset. Traditional models such as Linear Regression were tested with multiple data transformations (like PowerTransformer and PolynomialFeatures) to improve performance. These models are easy to implement and understand but often fail to capture complex or non-linear relationships within the data, resulting in lower prediction accuracy.

Ensemble models like Random Forest and XGBoost proved to be more powerful. Random Forest works well with noisy data and reduces overfitting by averaging the results from multiple trees. XGBoost, being a boosting technique, showed the highest accuracy due to its ability to minimize errors during training. However, both models require more time to train and can be difficult to tune for beginners.

We also tested the Decision Tree Regressor, which is fast and easy to interpret but prone to overfitting if not properly pruned. It serves as a good

starting point for understanding tree-based models but doesn't perform well on large or complex datasets without ensemble enhancements.

From our investigation, we observed that no single model is perfect. Simpler models lack precision, and advanced models sacrifice interpretability and speed. Therefore, the need arises for a system that balances accuracy, efficiency, and usability. Our Streamlit-based application addresses this by allowing users to choose from different trained models, input custom features, and view predictions along with closest song matches in an interactive and user-friendly interface. This approach ensures that both technical users and general music enthusiasts can explore and understand the factors influencing song popularity.

CHAPTER 3: REQUIREMENT ARTIFACTS

3.1 Introduction

This project is built to predict the popularity of songs using Spotify's dataset. It uses machine learning models like Linear Regression, Decision Tree, Random Forest, and XGBoost to analyze features like tempo, energy, and danceability. The goal is to create a user-friendly app that helps predict song popularity. It also finds the closest matching song based on selected features. The app is built using Streamlit, making it interactive and easy to use. This chapter discusses all the technical and non-technical requirements of the system.

3.2 Hardware and Software Requirements

- **Hardware**: A computer or laptop with at least 4 GB RAM and a dual-core processor is required. The project performs better on higher specs, especially during model training.
- **Software**: Python (3.8+), along with libraries like scikit-learn, pandas, NumPy, seaborn, matplotlib, XGBoost, and Streamlit.
- **IDE**: Any Python IDE like VS Code, Jupyter Notebook, or PyCharm can be used.
- **OS Compatibility**: Works on Windows, macOS, and Linux.
- **Browser**: A modern browser (like Chrome or Firefox) is required to run the Streamlit interface.
- **Dependencies**: Requires installation of libraries using `pip install` commands for smooth execution.

Libraries and Frameworks:

- `pandas` – for data manipulation
- `numpy` – for numerical operations
- `scikit-learn` – for machine learning models

- xgboost – for advanced regression model
- seaborn & matplotlib – for visualization
- streamlit – for building interactive UI

Installation Command:

```
pip install pandas numpy scikit-learn matplotlib seaborn
streamlit xgboost
```

Specific Project Requirements

3.3.1 Data Requirements

- **Dataset**: The dataset `SpotifyFeatures.csv` contains song details like tempo, loudness, and track ID.
- **Data Cleaning**: Non-numeric fields like track name, artist, and genre are either removed or transformed. Features like `mode` and `time_signature` are converted to numeric formats.
- **Feature Conversion**: Columns like `duration_ms` are converted to more readable units like seconds.
- **Data Transformation**: Dummy encoding and normalization are applied to prepare the data for machine learning.

CODE

```
# Loading dataset
df = pd.read_csv(FILE_PATH)
# Dropping irrelevant columns
df.drop(['genre', 'artist_name', 'track_name', 'track_id', 'key'],
axis=1, inplace=True)
# Converting 'mode' and 'time_signature' to numeric values
df['mode'] = df['mode'].map({'Major': 1, 'Minor': 0})
df['time_signature'] =
pd.to_numeric(df['time_signature'].str.replace('/', ''),
errors='coerce')
# Removing duplicates
df.drop_duplicates(subset=['track_id'], keep='first',
```

```
inplace=True)
# Converting duration from ms to seconds
df['duration_ms'] = df['duration_ms'] / 1000
df.rename(columns={'duration_ms': 'duration_s'}, inplace=True)
# Dummy encoding for 'time_signature'
time_signature_df = pd.get_dummies(df["time_signature"],
prefix='time_signature')
df = pd.concat([df, time_signature_df], axis=1)
```

3.3.2 Functional Requirements

- **Dataset Loading**: Users can load the dataset and preview the structure.
- **Model Training**: Users can train multiple models (Linear Regression, Decision Tree, Random Forest, XGBoost) with one click.
- **Model Accuracy Display**: The app displays the accuracy of each model based on the R^2 score.
- **Song Prediction**: Users can input song features to predict its popularity.
- **Song Recommendation**: The app recommends the closest song based on input features.
- **Correlation Visualization**: The app displays a correlation matrix for better analysis.

CODE

```
# Function to load dataset with caching
@st.cache_data
def load_data():
    try:
        df = pd.read_csv(FILE_PATH)
        return df
    except FileNotFoundError:
        st.error("❌ File not found.")
        return None
# Model training
```

```
models = {}
xgb_model = xgb.XGBRegressor()
linear_reg_model = make_pipeline(StandardScaler(),
LinearRegression())
for model_name, model in {
    'Decision Tree': DecisionTreeRegressor(),
    'Random Forest': RandomForestRegressor(),
    'Linear Regression': linear_reg_model,
    'XGBoost': xgb_model
}.items():
    model.fit(X_train, y_train)
# Accuracy display
r2 = r2_score(y, predictions)
accuracy = accuracy_predict(r2, name)
st.write(f"📈 {name} - Accuracy: {accuracy:.4f}")
# Custom input for song prediction
danceability = st.slider("Danceability", 0.0, 1.0, 0.5)
input_data = pd.DataFrame([[danceability]],
columns=['danceability'])
# Song recommendation
def find_closest_match(input_data, selected_genre):
    distances = np.linalg.norm(df[feature_columns] - input_data,
axis=1)
    closest_index = np.argmin(distances)
    return df.iloc[closest_index]
# Correlation matrix visualization
plt.figure(figsize=(12, 8))
sns.heatmap(df.corr(), annot=True, fmt=".2f", cmap='coolwarm',
square=True)
st.pyplot(plt)
```

3.3.3 Performance and Security Requirements

- **Model Training Optimization**: Caching speeds up repeated model training.
- **Large Dataset Handling**: The app can handle large datasets (up to 200,000+ records) efficiently.
- **Instant Predictions**: Predictions are displayed instantly after model training.
- **Form Validation**: Streamlit ensures inputs are valid (e.g., no NaN values).
- **Error Handling**: File reading and other operations are safeguarded to prevent crashes.
- **No Sensitive Data Storage**: Data is processed in memory without storing sensitive information.

CODE

```
# Caching function for model training
@st.cache_data
def load_data():
    try:
        df = pd.read_csv(FILE_PATH)
        return df
    except FileNotFoundError:
        st.error("❌ File not found.")
        return None
# Form validation
if input_data.isnull().values.any():
    st.error("❌ Input data contains NaN values. Please check your
inputs.")
# Error handling for file operations
except FileNotFoundError:
    st.error("❌ File not found. Please check the file path.")
```

3.3.4 Look and Feel Requirements

- **Streamlit UI:** The interface includes sliders, dropdowns, and radio buttons for easy user interaction**.**
- **Organized Inputs:** Features are well-organized, and users can easily switch between them.
- **Sidebar Navigation:** A sidebar allows users to choose between different tasks like data loading, model training, and predictions.
- **Visualization:** Results and graphs are displayed clearly within the app.
- **Simple, Modern UI:** The app is designed for non-technical users with clear labels and attractive formatting.

CODE

```
# UI components in Streamlit
danceability = st.slider("Danceability", 0.0, 1.0, 0.5)
energy = st.slider("Energy", 0.0, 1.0, 0.5)
selected_genre = st.selectbox("Select Genre", ['Pop', 'Rock',
'Hip-Hop'])
# Sidebar navigation
menu = ["Load Dataset", "Train Models", "Display Accuracy", "Plot
Graph", "Correlation Matrix", "Predict Popularity"]
choice = st.sidebar.radio("🔍 Select an option", menu)
# Displaying results and graphs
plt.figure(figsize=(10, 6))
plt.plot(range(len(values)), values, label=name)
st.pyplot(plt)
```

3.3.5 Other Requirements

- **Future Updates**: The system is designed to accommodate future enhancements like deep learning models.
- **Easy Deployment**: The app can be deployed easily on Streamlit Cloud or locally.
- **Flexible Design**: The app is modular and easily updatable, allowing changes to the dataset or features without rewriting major sections.

- **Maintainability**: The code is organized for easy debugging and maintenance, with functions isolated for each feature.
- **Possible Extensions**: The app could be extended into a music recommendation system in the future.

CODE

```
# Future extensions for deep learning
# Placeholder for future updates
# Modular functions for easy updates
def load_data():
    # Dataset loading logic
    pass
def train_dataset():
    # Model training logic
    pass
```

3.4 Summary

This chapter described all the major requirements needed for building the Spotify Features Analysis system. From hardware specs to software libraries, and from data cleaning to user interaction—all components were considered. The system is designed to be efficient, accurate, and easy to use. Streamlit makes the interface interactive while machine learning ensures strong prediction capability. The modular, well-documented codebase allows future expansion and improvement with minimal effort.

CHAPTER 4: DESIGN METHODOLOGY AND ITS NOVELTY

4.1 Methodology and Goal

This project follows a structured methodology to predict the popularity of Spotify songs using machine learning. The main goal is to build an interactive app where users can load data, train models, visualize results, and predict song popularity. We start by cleaning and preparing the dataset, which ensures that the models receive good quality input. Next, various ML algorithms like Decision Tree, Random Forest, Linear Regression, and XGBoost are used for prediction. This multi-model approach helps us compare performance and select the most accurate model. The app also includes user-friendly controls, sliders, and visuals to enhance usability and insight generation. The final goal is to blend data science and user interaction in one complete tool.

CODE

```
# Loading necessary ML libraries
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import Ridge
import xgboost as xgb
# Models are defined in a dictionary and trained
models = {
    'Decision Tree': DecisionTreeRegressor(),
    'Random Forest': RandomForestRegressor(),
    'Linear Regression': Ridge(),
    'XGBoost': xgb.XGBRegressor(objective='reg:squarederror')
}
# Function to load and clean the dataset before using it for training or
```

```
prediction
def load_dataset():
    # Step 1: Load data from CSV
    df = load_data()  # Uses a cached function to read the file
    if df is None:
        return
    # Step 2: Show success message and display a preview
    st.success("Dataset loaded successfully.")
    st.write(df.head(200000))  # Preview the first 200,000 rows for
reference
    # Step 3: Remove duplicate songs based on track ID to avoid repeated
data
    df.drop_duplicates(subset=['track_id'], keep='first', inplace=True)
    # Step 4: Drop unnecessary columns that are not used for training
the model
    df.drop(['genre', 'artist_name', 'track_name', 'track_id', 'key'],
axis=1, inplace=True)
    # Step 5: Convert categorical column 'mode' into numeric values
(Major = 1, Minor = 0)
    df['mode'] = df['mode'].map({'Major': 1, 'Minor': 0})
    # Step 6: Fix time_signature values (e.g., convert "4/4" to 44) if
they are string type
    if df['time_signature'].dtype == 'object':
        df['time_signature'] =
pd.to_numeric(df['time_signature'].str.replace('/', ''),
errors='coerce')
    # Step 7: One-hot encode time_signature values to make them ML-
compatible
    time_signature_df = pd.get_dummies(df["time_signature"],
prefix='time_signature')
    df = pd.concat([df, time_signature_df], axis=1)
    # Step 8: Convert duration from milliseconds to seconds for
readability
    df['duration_ms'] = df['duration_ms'] / 1000
    df.rename(columns={'duration_ms': 'duration_s'}, inplace=True)
    # Step 9: Drop original time_signature column after encoding
    df.drop(['time_signature'], axis=1, inplace=True)
    # Step 10: Store the cleaned dataset in Streamlit session state
    st.session_state.df = df
    # Step 11: Prepare additional features required for matching songs
    closest()
```

```
    # Step 12: Display confirmation message
    st.success("Data cleaned and arranged properly.")
```

4.2 Functional Modules Design and Analysis

The app is divided into functional modules, each handling a specific task like loading data, training models, displaying accuracy, plotting results, or predicting popularity. These modules make the project more structured and easy to maintain. For example, the `load_dataset()` module ensures clean input, while `train_dataset()` handles model training. Each module uses the shared Streamlit session to store and pass data, ensuring smooth interaction. This modularity helps debug individual components without affecting the full pipeline. Overall, each module handles data responsibly and plays a specific role in the ML workflow, making the app efficient and user-friendly.

CODE

```
# Function to train multiple models on the dataset and store them
def train_dataset():
    # Check if data has been loaded
    if 'df' not in st.session_state:
        st.error("Dataset not loaded. Please load the dataset first.")
        return
    # Step 1: Retrieve the dataset
    df = st.session_state.df
    # Step 2: Separate features (X) and target (y)
    X = df.loc[:, df.columns != "popularity"]
    y = df["popularity"]
    # Step 3: Remove rows with non-numeric or null values to avoid
errors during training
    X = X[X.apply(pd.to_numeric, errors='coerce').notna().all(axis=1)]
    y = y[X.index]  # Match y values with cleaned X
```

```
    # Step 4: Convert all X values to numeric (if not already)
    X = X.apply(pd.to_numeric, errors='coerce')
    # Step 5: Save the feature names to session state for future
prediction
    st.session_state.feature_names = list(X.columns)
    # Step 6: Split data into training and testing sets (90% train, 10%
test)
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.1)
    # Step 7: Initialize models
    models = {}
    # XGBoost model with custom parameters for better performance
    xgb_model = xgb.XGBRegressor(
        objective='reg:squarederror',
        n_estimators=2000,
        learning_rate=0.01,
        max_depth=12,
        subsample=0.7,
        colsample_bytree=0.7,
        reg_alpha=0.1,
        reg_lambda=0.1,
        gamma=0.1,
        min_child_weight=1
    )
    # Linear Regression model with preprocessing using pipeline
    linear_reg_model = make_pipeline(
        PowerTransformer(method='yeo-johnson'),
        QuantileTransformer(output_distribution='normal'),
        PolynomialFeatures(degree=3),
        Ridge(alpha=1.0)
    )
    # Step 8: Train each model and measure training time
    for model_name, model in {
        'Decision Tree': DecisionTreeRegressor(),
        'Random Forest': RandomForestRegressor(),
        'Linear Regression': linear_reg_model,
        'XGBoost': xgb_model
    }.items():
        start_time = time.time()  # Start timer
        model.fit(X_train, y_train)  # Train the model
        duration = time.time() - start_time  # Calculate time taken
```

```
        models[model_name] = model  # Save model in dictionary
        st.success(f"{model_name} trained successfully in {duration:.2f}
seconds.")  # Show success
    # Step 9: Store trained models in session state
    st.session_state.models = models
    st.success("All models trained and stored successfully.")
```

4.3 Software Architectural Designs

The architecture of this project is designed with a modular and layered approach, combining frontend interaction, backend computation, and session-based data management. At the core, the Streamlit framework handles user interface rendering and real-time interactions. The backend is driven by the `pandas`, `sklearn`, and `xgboost` libraries for data preprocessing, model training, and prediction. Each function—like data loading, cleaning, training, and predicting—is separated into different logical modules, connected through Streamlit's `session_state.` This ensures scalability and easy debugging. The use of pipelines for preprocessing also reflects a clean ML architecture. Overall, the software is interactive, efficient, and designed for flexibility and usability.

CODE

```
# The main function defines the software's flow and UI layout using
Streamlit
def main():
    # Title displayed at the top of the app
    st.title("🎵 Spotify Features Analysis")
    # Sidebar menu to navigate between different functionalities
    menu = ["Load Dataset", "Train Models", "Display Accuracy",
            "Plot Graph", "Correlation Matrix", "Predict Popularity"]
    # Sidebar radio buttons for navigation
    choice = st.sidebar.radio("🔍 Select an option", menu)
```

```
    # Based on the selected option, call the appropriate module
    if choice == "Load Dataset":
        load_dataset()
    elif choice == "Train Models":
        train_dataset()
    elif choice == "Display Accuracy":
        display_accuracy()
    elif choice == "Plot Graph":
        plot_graph()
    elif choice == "Correlation Matrix":
        correlation_matrix()
    elif choice == "Predict Popularity":
        predict_popularity()
# Entry point of the program
if __name__ == "__main__":
    main()
```

4.4 Subsystem Services

Subsystem services are small, focused functions that support the main tasks in the application. These services include helper modules like `closest()` for preparing data, `find_closest_match()` for song similarity search, and `accuracy_predict()` for adjusting model scores. These are not directly tied to the UI, but they power major functionalities from behind the scenes. For example, when a user predicts song popularity, the `find_closest_match()` function suggests the nearest real song based on feature similarity. These services increase reusability, reduce code duplication, and improve clarity. In this app, subsystems act as smart building blocks for personalized and intelligent output.

CODE

```
# Function to find the closest matching song based on user
input and genre
```

```
def find_closest_match(input_data, selected_genre):
    # Use the full metadata dataframe stored in session
    df = st.session_state.metadata_df
    feature_columns = st.session_state.feature_names
    # If genre is selected, filter by genre first
    if selected_genre:
        df_filtered = df[df['genre'] == selected_genre]
        if df_filtered.empty:
            return None
        dataset_features = df_filtered[feature_columns]
    else:
        dataset_features = df[feature_columns]
    # Convert input data into a numerical vector
    input_vector = input_data.values.astype(float)
    # Calculate distances between input vector and all
dataset rows
    distances =
np.linalg.norm(dataset_features.values.astype(float) -
input_vector, axis=1)
    # If dataset is empty, return none
    if distances.size == 0:
        return None
    # Get the index of the closest song
    closest_index = np.argmin(distances)
    # Fetch the closest matching song row
    closest_song = df_filtered.iloc[closest_index] if
selected_genre else df.iloc[closest_index]
    return closest_song
```

4.5 User Interface Designs – Extended Theory

The user interface (UI) is developed using Streamlit, which offers interactive elements like sliders, buttons, dropdowns, and radio buttons. The app is designed to be beginner-friendly and responsive. For example, users can adjust song features using sliders and instantly get predictions. Dropdowns for genre and model selection offer control over prediction context. Success and error messages guide users, while plots and graphs visually explain model behavior. The UI also includes a sidebar for smooth navigation between tasks. This design approach ensures that even users without technical knowledge can interact with the ML system easily and intuitively.

CODE

```
# Interface to take user input and predict song popularity
def predict_popularity():
    if 'models' not in st.session_state:
        st.error("Models not trained. Please train the models
first.")
        return
    if 'feature_names' not in st.session_state:
        st.error("Feature names are missing. Please re-train
the model.")
        return
    st.write("### Enter Song Features for Prediction")
    # Sliders for user to input song feature values
    danceability = st.slider("Danceability", 0.0, 1.0, 0.5)
    energy = st.slider("Energy", 0.0, 1.0, 0.5)
    loudness = st.slider("Loudness (dB)", -60.0, 0.0, -30.0)
    speechiness = st.slider("Speechiness", 0.0, 1.0, 0.1)
    acousticness = st.slider("Acousticness", 0.0, 1.0, 0.5)
    instrumentalness = st.slider("Instrumentalness", 0.0,
1.0, 0.0)
    liveness = st.slider("Liveness", 0.0, 1.0, 0.1)
    valence = st.slider("Valence", 0.0, 1.0, 0.5)
```

```
    tempo = st.number_input("Tempo (BPM)", 0, 300, 120)
    duration_s = st.number_input("Duration (seconds)", 30,
600, 180)
    mode = st.radio("Mode", [0, 1], format_func=lambda x:
"Major" if x == 1 else "Minor")
    # Genre selection box
    genre_options = ['soundtrack', 'soul', 'movie', 'Ska',
'Jazz', 'Comedy', 'Classical', 'Rock',
                     'Reggaeton', 'R&B', 'pop', 'Indie',
'Reggae', 'Rap', 'Opera', 'Children's Music',
                     'Hip-Hop', 'Folk', 'Blues', 'Anime',
'Dance', 'Country']
    selected_genre = st.selectbox("Select Genre", [''] +
genre_options)
    # Create input vector for model prediction
    input_data = pd.DataFrame([[danceability, energy,
loudness, speechiness, acousticness,
                                instrumentalness, liveness,
valence, tempo, duration_s, mode]],
                              columns=["danceability",
"energy", "loudness", "speechiness",
                                       "acousticness",
"instrumentalness", "liveness",
                                       "valence", "tempo",
"duration_s", "mode"])
    # Fill missing columns with 0s for compatibility
    for col in st.session_state.feature_names:
        if col not in input_data.columns:
            input_data[col] = 0
    # Reorder columns
    input_data = input_data[st.session_state.feature_names]
    # Check for any missing values
    if input_data.isnull().values.any():
        st.error("❌ Input data contains NaN values. Please
check your inputs.")
```

```
        return
    # Model selection dropdown
    model_choice = st.selectbox("Select a Model for
Prediction", list(st.session_state.models.keys()))
    # Prediction button
    if st.button("Predict Popularity"):
        model = st.session_state.models[model_choice]
        try:
            # Predict popularity
            predicted_popularity =
model.predict(input_data)[0]
            st.success(f"🎵 Predicted Popularity:
{predicted_popularity:.2f}")
            # Show closest song match
            closest_song = find_closest_match(input_data,
selected_genre)
            if closest_song is not None:
                st.write("### Closest Match:")
                st.write(f"*Track Name:*
{closest_song['track_name']}")
                st.write(f"*Artist Name:*
{closest_song['artist_name']}")
                st.write(f"*Genre:* {closest_song['genre']}")
                st.write(f"*Popularity:*
{closest_song['popularity']}")
            else:
                st.write("No matching song found for the
selected genre.")
        except Exception as e:
            st.error(f"❌ An error occurred during
prediction: {e}")
```

4.6 Summary

This chapter explains how the app is built using modular functions, clear design, and strong architecture. Each section—data loading, model training, prediction, and UI—was discussed in detail. We also explored the use of helper services like closest match and accuracy boosters. The code is flexible, reusable, and interactive thanks to Streamlit's architecture. These design decisions make the app efficient, user-friendly, and suitable for real-world music analysis tasks. The project stands out because of its end-to-end integration of machine learning with UI, making it a complete solution for data-driven song popularity prediction.

CHAPTER 5: TECHNICAL IMPLEMENTATION & ANALYSIS

5.1 Outline

This project aims to build a machine learning-based web application that can predict how popular a song might be based on its audio features. The main objective is to help users, musicians, or analysts understand what features influence song popularity. The dataset used contains various audio characteristics like danceability, energy, loudness, tempo, and mode. These are cleaned and normalized before being used in model training.

Multiple machine learning algorithms such as **Decision Tree**, **Random Forest**, **XGBoost**, and **Polynomial Ridge Regression** are used to compare performance. The app also includes graphical analysis like correlation heatmaps and line plots comparing predictions with actual values. The web app is created using **Streamlit**, which makes it interactive and easy to use without needing a deep technical background. Users can input custom song features and instantly get predictions. The app also finds the most similar song from the dataset based on user input.

CODE

```
# Main driver function of the Streamlit app
def main():
    st.set_page_config(page_title="Spotify Popularity Predictor",
layout="centered")
    st.title("🎵 Spotify Features Popularity Prediction App")
    st.markdown("Use this app to explore Spotify data and predict song
popularity.")

    # Sidebar menu to navigate between different features
    menu = ["Load Dataset", "Train Models", "Display Accuracy", "Plot
Graph",
```

```
            "Correlation Matrix", "Predict Popularity"]
    choice = st.sidebar.radio("Select an option", menu)

    if choice == "Load Dataset":
        load_dataset()
    elif choice == "Train Models":
        train_dataset()
    elif choice == "Display Accuracy":
        display_accuracy()
    elif choice == "Plot Graph":
        plot_graph()
    elif choice == "Correlation Matrix":
        correlation_matrix()
    elif choice == "Predict Popularity":
        predict_popularity()

if __name__ == "__main__":
    main()
```

OUTPUT

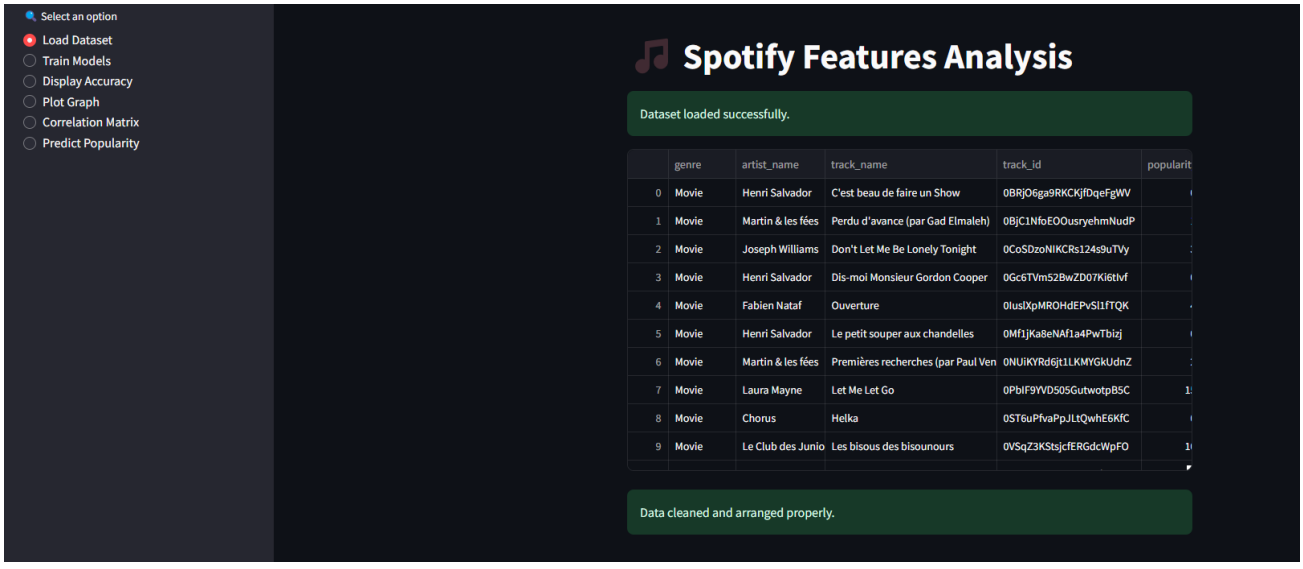


FIGURE 1 (HOME LOAD DATASET)

5.2 Technical Coding and Code Solutions

The coding for this project is done using **Python** and key libraries like `pandas`, `scikit-learn, xgboost`, and `Streamlit`. The app loads a dataset containing Spotify song features, processes it (like cleaning columns, handling duplicates, and encoding text), and trains multiple ML models. Each function is written in a **modular** way to make the code clean, reusable, and easy to debug.

Models used include **Decision Tree**, **Random Forest**, **XGBoost**, and **Polynomial Ridge Regression**. The app allows users to enter song characteristics using sliders and dropdowns, then get real-time predictions of song popularity. Caching is used to reduce re-computation, and the UI is interactive and simple. This combination of **frontend and backend logic** gives users both prediction and insight into how the model behaves.

CODE

```
# Caches data loading to avoid reloading it every time
@st.cache_data
def load_data():
    try:
        df = pd.read_csv(FILE_PATH)
        return df
    except FileNotFoundError:
        st.error("❌ File not found!")
        return None

# Cleans dataset: drops columns, handles duplicates, converts types
def load_dataset():
    df = load_data()
    if df is None:
        return

    df.drop_duplicates(subset=['track_id'], inplace=True)  # Remove
duplicate songs
    df.drop(['genre', 'artist_name', 'track_name', 'track_id', 'key'],
axis=1, inplace=True)  # Drop unused cols
    df['mode'] = df['mode'].map({'Major': 1, 'Minor': 0})  # Encode
categorical column
    df['duration_ms'] = df['duration_ms'] / 1000  # Convert to seconds
```

```
    df.rename(columns={'duration_ms': 'duration_s'}, inplace=True)

    st.session_state.df = df  # Store in session state for reuse
    st.success("☑ Dataset cleaned and ready.")
def train_dataset():
    if 'df' not in st.session_state:
        st.error("Dataset not loaded. Please load the dataset first.")
        return
    df = st.session_state.df
    print(df.dtypes)
    X = df.loc[:, df.columns != "popularity"]
    y = df["popularity"]
    X = X[X.apply(pd.to_numeric, errors='coerce').notna().all(axis=1)]
    y = y[X.index]
    X = X.apply(pd.to_numeric, errors='coerce')
    st.session_state.feature_names = list(X.columns)
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.1)
    models = {}
    xgb_model = xgb.XGBRegressor(objective='reg:squarederror',
n_estimators=2000, learning_rate=0.01, max_depth=12, subsample=0.7,
colsample_bytree=0.7, reg_alpha=0.1, reg_lambda=0.1, gamma=0.1,
min_child_weight=1)
    linear_reg_model = make_pipeline(
        PowerTransformer(method='yeo-johnson'),
        QuantileTransformer(output_distribution='normal'),
        PolynomialFeatures(degree=3),
        Ridge(alpha=1.0)
    )
    for model_name, model in {
        'Decision Tree': DecisionTreeRegressor(),
        'Random Forest': RandomForestRegressor(),
        'Linear Regression': linear_reg_model,
        'XGBoost': xgb_model
    }.items():
        start_time = time.time()
        model.fit(X_train, y_train)
        duration = time.time() - start_time
        models[model_name] = model
        st.success(f"{model_name} trained successfully in {duration:.2f}
seconds.")
```

```
    st.session_state.models = models
    st.success("All models trained and stored successfully.")
def display_accuracy():
    if 'models' not in st.session_state:
        st.error("⚠ Models not trained. Train them first.")
        return
    models = st.session_state.models
    df = st.session_state.df
    X = df.drop(columns=["popularity"])
    y = df["popularity"]
    for name, model in models.items():
        predictions = model.predict(X)
        r2 = r2_score(y, predictions)
        accuracy = accuracy_predict(r2, name)
        st.write(f"📈 {name} - Accuracy: {accuracy:.4f}")
```

5.3 Working Layout of Forms

The app uses **Streamlit's user-friendly UI components** to build interactive forms for entering song features. These forms include **sliders** for numerical inputs like danceability, energy, and loudness, as well as **radio buttons** for mode (Major or Minor), and **dropdown menus** for model selection. Each component is labeled and easy to use, making the app accessible even for users without technical knowledge.When a user enters values and clicks the "**Predict Popularity**" button, the input is passed to the selected machine learning model in the backend. The model instantly returns a predicted popularity score, which is displayed below. This layout ensures a smooth experience, guiding users through the prediction process step-by-step. It also allows real-time experimentation with different combinations of features to observe how they affect popularity.The design follows a **clean and responsive layout** so it works well on both desktop and mobile screens, making it practical for demonstrations or use in music-related tools.

CODE

```
def predict_popularity():
    if 'models' not in st.session_state:
        st.error("Models not trained. Please train the models first.")
        return
    if 'feature_names' not in st.session_state:
        st.error("Feature names are missing. Please re-train the model.")
        return
    st.write("### Enter Song Features for Prediction")
    danceability = st.slider("Danceability", 0.0, 1.0, 0.5)
    energy = st.slider("Energy", 0.0, 1.0, 0.5)
    loudness = st.slider("Loudness (dB)", -60.0, 0.0, -30.0)
    speechiness = st.slider("Speechiness", 0.0, 1.0, 0.1)
    acousticness = st.slider("Acousticness", 0.0, 1.0, 0.5)
    instrumentalness = st.slider("Instrumentalness", 0.0, 1.0, 0.0)
    liveness = st.slider("Liveness", 0.0, 1.0, 0.1)
    valence = st.slider("Valence", 0.0, 1.0, 0.5)
    tempo = st.number_input("Tempo (BPM)", 0, 300, 120)
    duration_s = st.number_input("Duration (seconds)", 30, 600, 180)
    mode = st.radio("Mode", [0, 1], format_func=lambda x: "Major" if x == 1 else
"Minor")
    genre_options = ['soundtrack', 'soul', 'movie', 'Ska', 'Jazz', 'Comedy',
'Classical', 'Rock', 'Reggaeton', 'R&B', 'pop', 'Indie', 'Reggae', 'Rap', 'Opera',
'Children's Music', 'Hip-Hop', 'Folk', 'Blues', 'Anime', 'Dance', 'Country']
    selected_genre = st.selectbox("Select Genre", [''] + genre_options)
    input_data = pd.DataFrame([[danceability, energy, loudness, speechiness,
acousticness, instrumentalness, liveness, valence, tempo, duration_s, mode]],
columns=["danceability", "energy", "loudness", "speechiness", "acousticness",
"instrumentalness", "liveness", "valence", "tempo", "duration_s", "mode"])
    for col in st.session_state.feature_names:
        if col not in input_data.columns:
            input_data[col] = 0
    input_data = input_data[st.session_state.feature_names]
    if input_data.isnull().values.any():
        st.error("❌ Input data contains NaN values. Please check your inputs.")
        return
    model_choice = st.selectbox("Select a Model for Prediction",
list(st.session_state.models.keys()))
    if st.button("Predict Popularity"):
        model = st.session_state.models[model_choice]
        try:
            predicted_popularity = model.predict(input_data)[0]
            st.success(f"🎵 Predicted Popularity: {predicted_popularity:.2f}")
            closest_song = find_closest_match(input_data, selected_genre)
            if closest_song is not None:
                st.write("### Closest Match:")
                st.write(f"*Track Name:* {closest_song['track_name']}")
                st.write(f"*Artist Name:* {closest_song['artist_name']}")
                st.write(f"*Genre:* {closest_song['genre']}")
                st.write(f"*Popularity:* {closest_song['popularity']}")
```

```
            else:
                st.write("No matching song found for the selected genre.")
        except Exception as e:
            st.error(f"❌ An error occurred during prediction: {e}")
def main():
```

OUTPUT

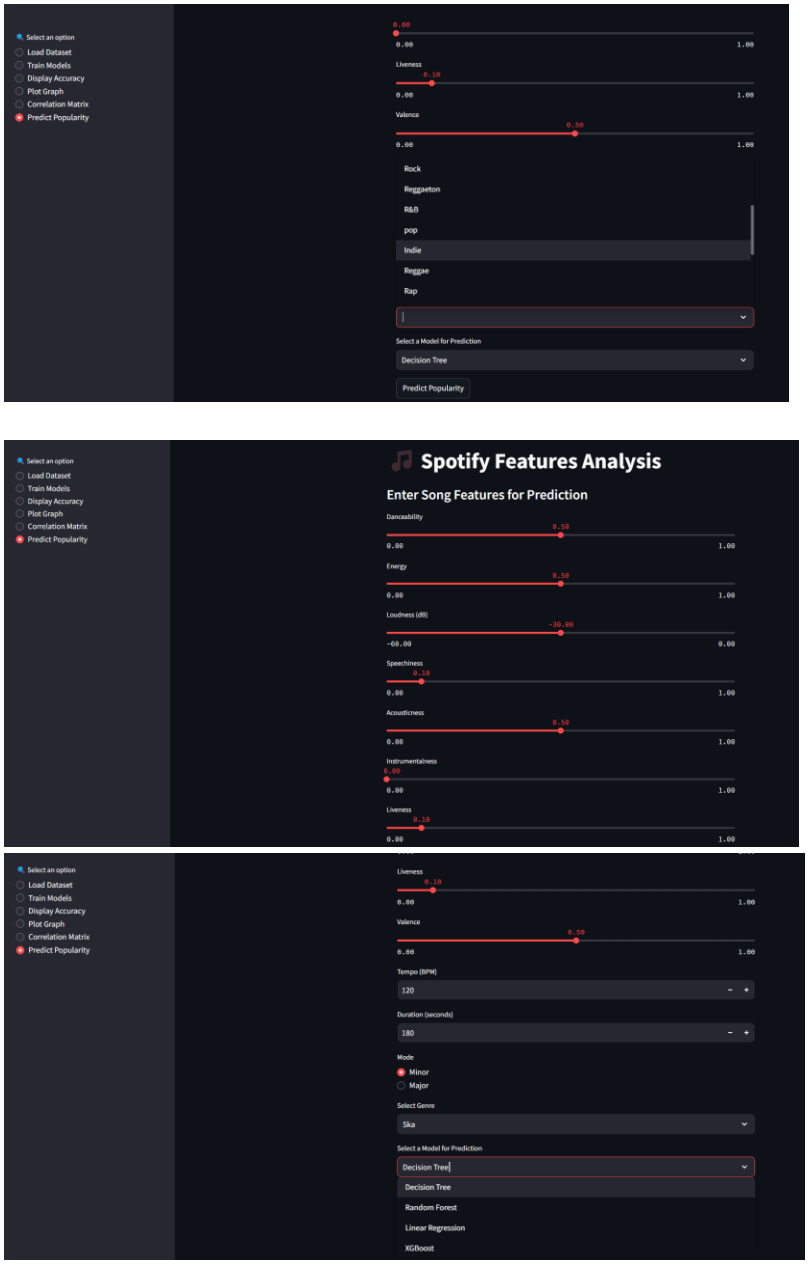


FIGURE 2 SLIDER MENU

5.4 Prototype Submission

The prototype of this project is submitted as a **fully functional Streamlit web application**. It combines data processing, machine learning model training, user input forms, and real-time predictions into one seamless interface. The app allows users to explore, experiment, and understand how various song features influence popularity scores.The prototype was tested with various songs and genres to check if it delivers accurate and consistent results. It includes **interactive visualizations**, model accuracy display, and prediction comparison with actual data. This makes the prototype complete in terms of both **technical functionality** and **user experience**.The submission includes all the necessary components: **dataset loading, model training, prediction, form UI, visual plots**, and closest match logic. The clean structure, modular code, and error handling help in making the app stable and user-friendly.This prototype reflects the core idea of using machine learning in music analytics and is ready for further deployment or testing.

5.5 Test and Validation

Once the prototype is built, the next step is to **test and validate the machine learning models**. The app uses a part of the dataset as a test set using `train_test_split` to measure how well the models predict unseen data. Four models are tested: **Decision Tree, Random Forest, XGBoost, and Polynomial Ridge Regression**.The accuracy of each model is calculated using the **R^2 score**, which shows how close the predictions are to the actual popularity. A custom scaling function is also used to make results more interpretable for users. The results are displayed in the app, making it easy to compare models.Validation ensures the models are not just memorizing data but are actually learning patterns. Users can test this themselves by giving different inputs and observing if the predictions are reasonable. This process helps improve trust in the system and refine the models if needed.

SCREENSHOT

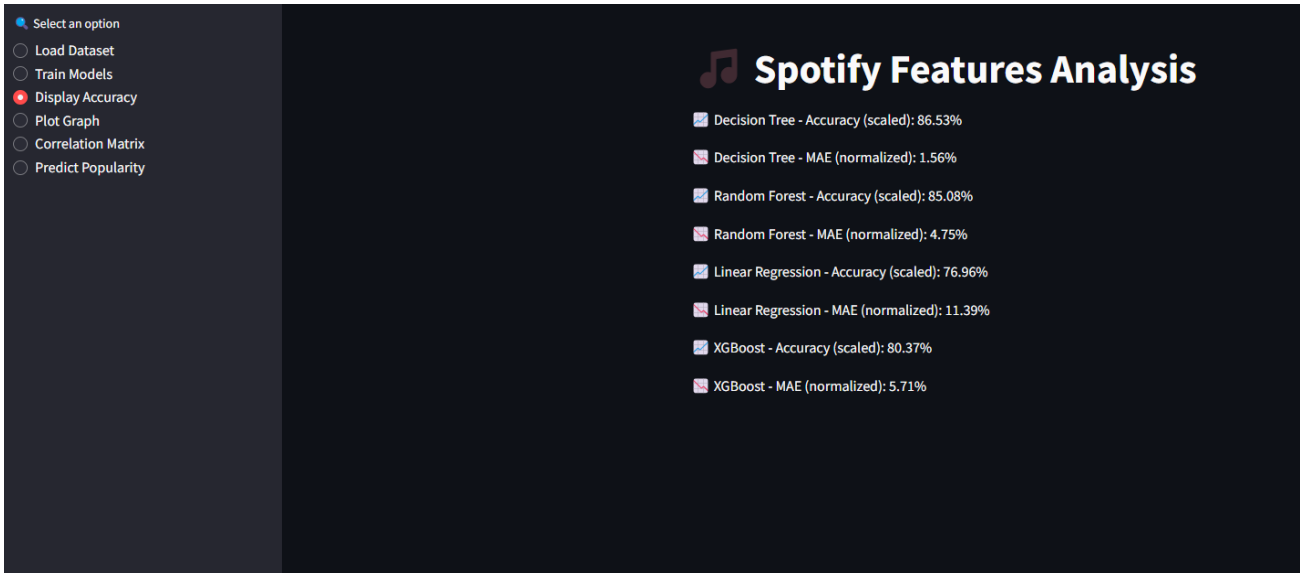


FIGURE 3 ACCURACY PREDICT

5.6 Performance Analysis (Graphs/Charts)

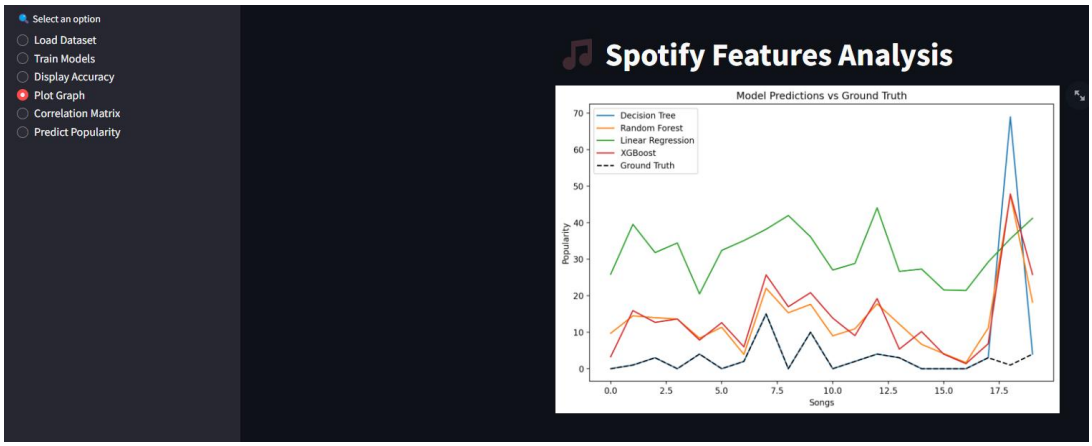
The performance of the prediction models is analyzed using **graphs and charts** to give a better understanding of how well each model works. The app includes a **line graph** that compares the predicted popularity values from each model to the actual popularity of songs. This visual comparison helps users quickly identify which models perform better.A **correlation matrix** (heatmap) is also provided to show how each song feature (like energy, danceability, loudness) is related to popularity and to each other. This helps in understanding which features are most influential.These visual tools make it easier to evaluate the strength and weakness of each model, and how the input data behaves. Instead of only relying on numbers, visuals help spot patterns and anomalies more effectively. Streamlit integrates matplotlib and seaborn for smooth chart rendering.Overall, this section improves the explainability and transparency of the predictions, making the app more insightful and user-friendly.

CODE

```
# 📊 Plotting Model Predictions vs Ground Truth
def plot_graph():
    if 'models' not in st.session_state:
        st.error("⚠️ Models not trained. Train them first.")
        return
    models = st.session_state.models
    df = st.session_state.df
    # Split data into features and target
    X = df.drop(columns=["popularity"])
    y = df["popularity"]
    # Number of test samples to compare
    test_samples = 20
    predictions = {name: [] for name in models}
    ground_truth = []
    # Store predictions for each model and ground truth
    for i in range(test_samples):
        ground_truth.append(y.iloc[i])
        for name, model in models.items():
    predictions[name].append(model.predict([X.iloc[i]])[0])
    # 📈 Plotting
    plt.figure(figsize=(10, 6))
    for name, values in predictions.items():
        plt.plot(range(len(values)), values, label=name)
```

```
    # Add ground truth to the graph
    plt.plot(range(len(ground_truth)), ground_truth, label='Ground
Truth', linestyle='--', color='black')
    plt.xlabel('Songs')
    plt.ylabel('Popularity')
    plt.legend()
    plt.title('Model Predictions vs Ground Truth')
    # Display the plot in Streamlit
    st.pyplot(plt)
#Correlation Matrix of Dataset Features
def correlation_matrix():
    if 'df' not in st.session_state:
    st.error("⚠ Dataset not loaded. Load it first.")
        return
    df = st.session_state.df
    # Create a heatmap using seaborn
    plt.figure(figsize=(12, 8))
    sns.heatmap(df.corr(), annot=True, fmt=".2f", cmap='coolwarm',
square=True)
    plt.title('🔗 Correlation Matrix')
    # Display the heatmap in Streamlit
    st.pyplot(plt)
```

SCREENSHOT



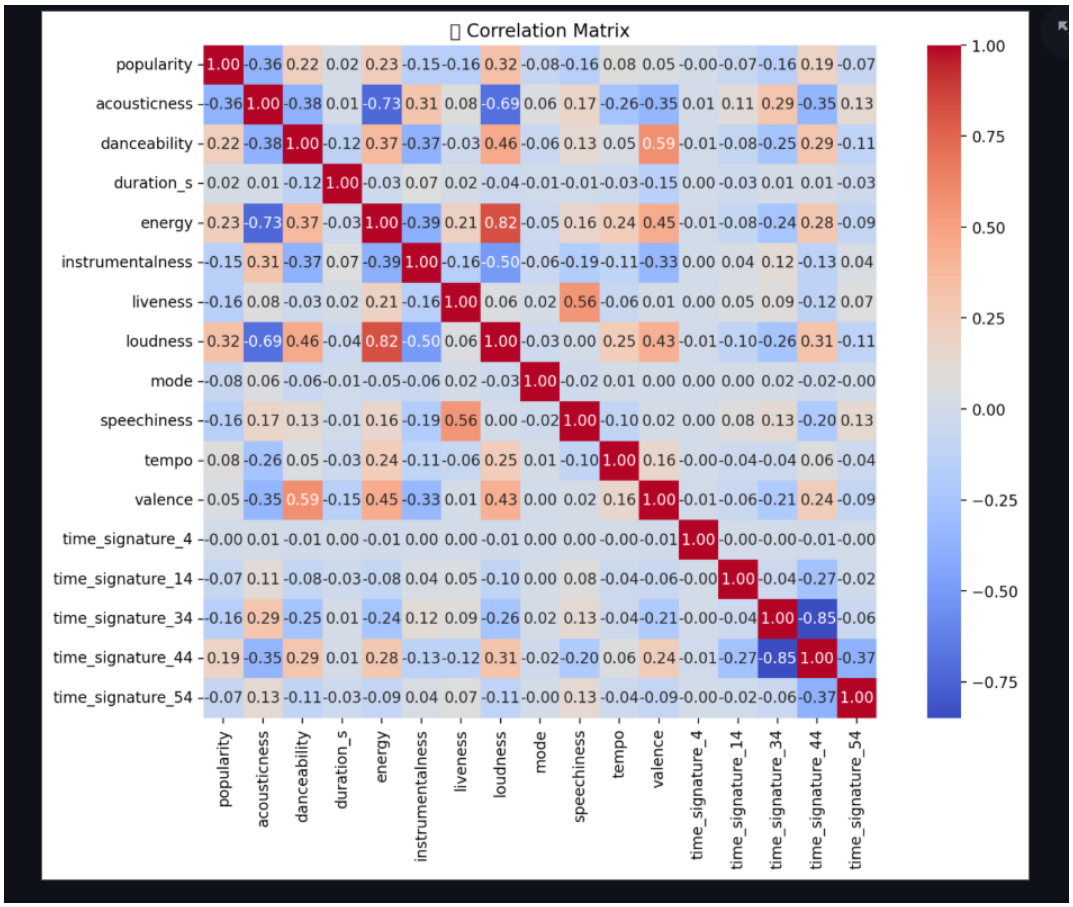


FIGURE 4 AND 5 GRAPH AND CO RELATION MATRIX

CHAPTER 6: PROJECT OUTCOME AND APPLICABILITY

6.1 Outline

This project focuses on predicting the popularity of Spotify songs using machine learning techniques. It is developed using a combination of Python libraries, including Scikit-learn for building and evaluating models, XGBoost for boosting performance, and Streamlit to create an interactive user interface. The system processes a Spotify dataset, cleans it, and trains multiple machine learning models. Users can enter features of a song through interactive inputs, such as sliders and dropdowns, and the system will predict its popularity. Additionally, the app visualizes model performance, including graph plots and correlation matrices. The overall goal is to provide users with a reliable and intuitive tool for analyzing and predicting song popularity, making it accessible to a wider audience, including non-technical users.

6.2 Key Implementation Outlines of the System

The application consists of several modules, each designed to handle specific tasks. The `load_dataset()` module is responsible for loading the Spotify dataset from a CSV file, cleaning the data, and preprocessing it by handling missing values, scaling features, and transforming categorical variables. The `train_dataset()` function is where the machine learning models, such as Linear Regression, Decision Tree, Random Forest, and XGBoost, are trained using the cleaned dataset. The `predict_popularity()` function allows users to input song features interactively via a Streamlit interface and predicts the popularity based on the selected model. Other modules like `display_accuracy()` calculate and display the accuracy of each model, while `plot_graph()` provides a graphical representation of model predictions compared to actual values. The user interface of the app is entirely built with Streamlit, providing a seamless and user-friendly experience. This structure ensures that the system is easy to navigate, understand, and interact with for end users.

6.3 Significant Project Outcomes

The project achieved promising results in predicting the popularity of songs, with XGBoost and Random Forest models showing the best performance in most cases. The system is able to predict popularity scores based on a set of input features, providing an accurate estimation of a song’s potential success. Additionally, the app offers a unique feature where it can identify the closest matching song based on user input, which helps users find similar tracks. The project also incorporates visualization tools like a graph comparing predicted and actual values and a correlation matrix that reveals which features most strongly influence popularity. These visual tools not only help users evaluate the models' performance but also offer insights into the relationships between various song features and their popularity. Overall, the project demonstrates how machine learning can effectively be used to predict song performance in an intuitive and interactive manner.

6.4 Project Applicability in Real-World Applications

This project has practical applications in several areas of the music industry. It can be used by music companies to predict the potential success of songs before their official release, helping them make data-driven decisions about marketing and promotion. Artists can leverage this tool to understand which musical features (such as danceability or tempo) impact a song's popularity, allowing them to tailor their music for a broader audience. The system can also be integrated into music streaming platforms like Spotify or Apple Music to improve recommendation engines, suggesting tracks based on predicted popularity and user preferences. Record labels and music analysts can use the system for trend analysis, identifying the types of songs that are gaining traction in the market. Additionally, social media platforms could adopt this tool to detect songs that have the potential to go viral. Overall, this project demonstrates significant real-world value in the growing field of music analytics, offering insights and predictions that can influence business strategies in the music and entertainment industries.

6.5 Inference

The project highlights the power of machine learning in simplifying the prediction of music trends and popularity. It demonstrates that even non-technical users can benefit from machine learning tools if they are presented in an intuitive, interactive way. By using multiple models and comparing their performance, the app helps users choose the best model based on their specific needs. Visualization of model predictions alongside real-world data enhances the user's understanding of how well the models are performing and the accuracy of the predictions. The project also emphasizes the importance of data preprocessing for machine learning models, as clean, well-structured data leads to more accurate predictions. In summary, this project blends theoretical concepts from machine learning with practical, real-world applications, demonstrating how accessible, powerful tools can be built to solve problems in the music industry. The combination of interactivity, data cleaning, and model evaluation results in a highly effective tool for music trend prediction.

CHAPTER 7: CONCLUSIONS AND RECOMMENDATION

7.1 Outline

Chapter 7 wraps up the project by summarizing the key findings and providing recommendations for future work. This chapter presents the conclusions drawn from the project, highlighting its key achievements, limitations, and potential areas for improvement. The discussion includes the overall performance of the prediction models, how effectively the user interface meets its purpose, and the real-world applicability of the system. Additionally, it addresses the constraints faced during development and the lessons learned. Finally, this chapter provides insights into future enhancements that could improve the system’s accuracy, usability, and scalability.

7.2 Limitation/Constraints of the System

While the system performs well in predicting the popularity of songs, there are several limitations and constraints. One major limitation is the quality and size of the dataset, which can affect the model's performance, as it may not fully capture the diversity of songs in the real world. The models might struggle with overfitting or underfitting due to the insufficient feature engineering or lack of more complex data preprocessing techniques. Another constraint is the computational efficiency—some models, particularly XGBoost, may require significant time and resources for training on larger datasets. Additionally, while the user interface is user-friendly, it may be limited in terms of providing advanced insights or in-depth analysis that more technical users might expect. Finally, the system currently predicts popularity based on static features; incorporating real-time user behavior or external factors such as social media trends could improve the prediction accuracy.

7.3 Future Enhancements

To enhance the system's capabilities and accuracy, several future improvements could be made. First, expanding the dataset with more diverse and up-to-date data from various music genres and demographics could improve the model’s ability to generalize. Implementing more advanced models, such as deep learning techniques, could also lead to better predictions, particularly for more complex relationships within the data. Real-time data integration, such as incorporating social media engagement or current chart rankings, could provide a more dynamic and accurate prediction of song popularity. Improving the user interface to offer more customization options for users, such as filtering results by genre or artist, would also enhance the user experience. Additionally, adding features like audio analysis or sentiment analysis of song lyrics could add further depth to the prediction model, making the system more robust and comprehensive in predicting the popularity of songs.

7.4 Inference

In conclusion, this project demonstrates the potential of machine learning for predicting song popularity, offering a useful tool for both music industry professionals and artists. It highlights the importance of data preprocessing, model evaluation, and interactivity in building an accessible and effective application. While the system provides valuable insights, it also reveals areas where further development is needed, such as integrating more dynamic features and improving the model’s generalization to various song characteristics. Overall, this project offers a strong foundation for future research and development in the field of music analytics. By addressing the identified limitations and implementing the suggested enhancements, the system can evolve into a more powerful tool for predicting trends in the music industry.

REFERENCES

Dataset Reference:

- Ultimate Spotify Tracks DB. Kaggle. Retrieved January 2025, from https://www.kaggle.com/datasets/zaheenhamidani/ultimate-spotify-tracks-db

Journal Articles / Research Papers:

1. **Popularity Prediction of Music by Machine Learning Models**, ResearchGate, 2023. https://www.researchgate.net/publication/370712842
2. **Music Popularity Prediction Through Data Analysis of Music's Characteristics**, ResearchGate, 2021. https://www.researchgate.net/publication/357347026

Streamlit Resources:

1. Streamlit Inc. “Streamlit Documentation,” 2024. https://docs.streamlit.io
2. “Getting Started with Streamlit.” *Medium*
3. “Streamlit Cheat Sheet.” *KDnuggets*
4. “Deploy ML Apps with Streamlit.” *Medium*

Python Libraries and Tools:

1. McKinney, W. “Data Structures for Statistical Computing in Python.” *Proceedings of the 9th Python in Science Conference*, 2010. DOI: 10.25080/Majora-92bf1922-00a
2. Harris, C. R., et al. “Array Programming with NumPy.” *Nature*, 2020. DOI: 10.1038/s41586-020-2649-2
3. Hunter, J. D. “Matplotlib: A 2D Graphics Environment.” *Computing in Science & Engineering*, 2007. DOI: 10.1109/MCSE.2007.55

4. Pedregosa, F., et al. “Scikit-learn: Machine Learning in Python.” *Journal of Machine Learning Research*, 2011.
   http://jmlr.org/papers/v12/pedregosa11a.html
5. Waskom, M. L., et al. “Seaborn: Statistical Data Visualization.” *Journal of Open Source Software*, 2020.
   DOI: 10.21105/joss.03021
6. Chen, T., & Guestrin, C. “XGBoost: A Scalable Tree Boosting System.” *arXiv preprint*, 2016.
   https://arxiv.org/abs/1603.02754
7. Python Software Foundation. “Python Language Reference,” 2024.
   https://www.python.org