Introduction:

As part of my internship , I was tasked with building a machine learning model to predict whether a song is popular based on streaming data from platforms like Spotify, YouTube, TikTok, and more. The dataset provided contained real metrics for top songs in 2024, which made the task feel both relevant and interesting. My goal was to prepare the data, explore different classification models, and evaluate their performance using standard metrics. Since I'm still new to machine learning, this project was a great opportunity to apply what I've been learning and get more comfortable working with real data, writing Python code in Google Colab, and understanding how different models behave. Throughout the process, I tried to keep things organized and focused on understanding *why* each step mattered — not just getting results, but actually learning from them.

Part 1: Data Preparation and Preprocessing:

In the first part of the project, I worked on preparing the dataset before applying any machine learning models. I started by uploading the Spotify 2024 dataset into Google Colab using pandas, which allowed me to explore the data and get a sense of the features available. Since I wanted to build a classification model, I created a new column called 'popular' by converting the spotify_popularity score into a binary label: if the score was 0.7 or higher, the song was labeled as popular (1), otherwise it was not popular (0). This helped turn the problem into something classification models could handle. After that, I selected six features that I believed were most related to popularity, such as playlist counts and platform view numbers from Spotify, YouTube, TikTok, and others. These became my input variables, while the new 'popular' column became the target. Since these features had very different value ranges (for example, YouTube views could be in the millions while others were much smaller), I normalized them using StandardScaler so that all features had the same scale. This was important for making sure the models trained fairly and accurately. Finally, I split the dataset into training and testing sets using a 50/50 split with random_state=10 so that the results would be consistent every time I ran the code.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

from google.colab import files
uploaded = files.upload()

df = pd.read_csv('/content/final_spotify_data_final.csv')


df['popular'] = df['spotify_popularity'].apply(lambda x: 1 if x >= 0.7 else 0)


features = [
    'spotify_playlist_count', 'spotify_playlist_reach', 'tiktok_views',
    'apple_music_playlist_count', 'youtube_views', 'deezer_playlist_count'
]

X = df[features]
y = df['popular']


scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)


X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.5, random_state=10)
```

Part 2: Checking Class Distribution

To understand how balanced or imbalanced the dataset was, I used value_counts() on the 'popular' column I created earlier. This showed how many songs were labeled as popular (1) versus not popular (0). It was important to check this before training any models because if one class heavily outnumbers the other, it can affect how well the model performs and may require adjustments later in the pipeline.

print(df['popular'].value_counts())

popular

0 2809

1 1791

Name: count, dtype: int64

Part 3: Selecting Features and Splitting the Data:
n this step, I selected six features from the dataset that I believed were most relevant to predicting a song's popularity. These included various playlist counts and view metrics from platforms like Spotify, TikTok, YouTube, and Apple Music. I then used these features as the input variables (X), while the popular column was used as the target (y). After that, I split the dataset into training and testing sets using a 70% test size and a random_state of 10. This allowed me to reserve part of the data for evaluating how well the models perform on unseen songs.

```
features = [
    'spotify_playlist_count', 'spotify_playlist_reach',
    'tiktok_views', 'apple_music_playlist_count',
    'youtube_views', 'deezer_playlist_count'
]

X = df[features]
y = df['popular']


from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.7, random_state=10
)
```

Part 4: Training and Evaluating Logistic Regression:

Here, I used the Logistic Regression model as my first classifier. I initialized the model and trained it using the training data (X_train, y_train). Once the model was trained, I used it to make predictions on the test data and then evaluated its performance using classification_report. This report gave me useful metrics like precision, recall, F1-score, and accuracy for both classes (popular and not popular)

```
from sklearn.linear_model import LogisticRegression
log_model = LogisticRegression()
log_model.fit(X_train, y_train)
y_pred_log = log_model.predict(X_test)

from sklearn.metrics import classification_report
print("Logistic Regression:\n", classification_report(y_test, y_pred_log))
```

```
Logistic Regression:
               precision    recall  f1-score   support

           0       0.75      0.94      0.84      1941
           1       0.86      0.53      0.66      1279

    accuracy                           0.78      3220
   macro avg       0.81      0.74      0.75      3220
weighted avg       0.79      0.78      0.77      3220
```

Part 6: Training and Evaluating Decision Tree:

Next, I used a Decision Tree classifier to model the data. I initialized the model with a fixed random_state for consistent results and trained it using the training dataset. After training, I used the model to predict the test set and evaluated the results using classification_report. Decision Trees work by splitting the data into branches based on feature values, making them easy to understand and visualize

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report


tree = DecisionTreeClassifier(random_state=42)


tree.fit(X_train, y_train)


y_pred_tree = tree.predict(X_test)


print("Decision Tree Results:\n")
print(classification_report(y_test, y_pred_tree))
```

```
Decision Tree Results:

              precision    recall  f1-score   support

           0       0.79      0.79      0.79      1941
           1       0.68      0.68      0.68      1279

    accuracy                           0.74      3220
   macro avg       0.73      0.73      0.73      3220
weighted avg       0.74      0.74      0.74      3220
```

Part 7: Training and Evaluating Random Forest:

finally, I trained a Random Forest classifier, which is an ensemble method made up of many individual Decision Trees. I set the number of trees (n_estimators) to 100 and used a random_state for consistency. After fitting the model on the training data, I predicted on the test set and printed out the evaluation results using classification_report. Random Forest usually performs better than a single Decision Tree because it combines the results of multiple trees, reducing overfitting and improving overall accuracy

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix


rf = RandomForestClassifier(n_estimators=100, random_state=42)

rf.fit(X_train, y_train)


y_pred_rf = rf.predict(X_test)


print("Random Forest Results:\n")
print(classification_report(y_test, y_pred_rf))

```

```
Random Forest Results:

              precision    recall  f1-score   support

           0       0.80      0.87      0.84      1941
           1       0.78      0.68      0.72      1279

    accuracy                           0.80      3220
   macro avg       0.79      0.78      0.78      3220
weighted avg       0.79      0.80      0.79      3220
```

Conclusion:

Working on this project helped me understand the full machine learning process — from preparing real-world data to training and comparing different models. I started by exploring and cleaning the dataset, then transformed it into a format suitable for classification. I experimented with several models including Logistic Regression, K-Nearest Neighbors, Decision Tree, and Random Forest. Each model gave me a different perspective on how algorithms work and what affects their performance. In the end, Random Forest delivered the most balanced and accurate results, likely because it combines the strengths of multiple decision trees