SCCM/CSForm/Rev. Ver 2.4

Please tick ✓ or click if using MS WORD

☐FOUNDATION ☐DIPLOMA [✓ DEGREE] ☐ MASTER

Assignment Coversheet

Please complete all details required clearly. For softcopy submissions, please ensure this cover sheet is included at the start of your document or in the file folder.

Assignment & Course Details:

Subject Code: XBDS 2024 (e.g. XCAT1234)		**Subject Name** (*e.g. Fundamentals of Computing*)**: Data Science Toolbox**	
Course (*e.g. Bachelor in Computing*) : Bachelor in Computer Science			
Lecturer Name: Dr Law Foong Li			
Assessment Due Date: (*dd/mm/yy*)	19/04/2023	**Assessment Title:**	Data Science Toolbox Assignment

I/We declare that:

- *This assignment is my/our own original work, except where I/we have appropriately cited the original source.*
- *This assignment or parts of it has not previously been submitted for assessment in this or any other subject.*
- *I/We allow the assessor of this assignment to test any work submitted by me/us, using text comparison software for plagiarism.* ***(For more information, Please read the Academic Integrity Guidelines)***

Name : Wan Mohammed Adam **Student ID:** 0132601 **Email :** 0132601@kdu-online.com **Mobile No:** 0109852593 **Signature:** [illegible] **Date:** 18/04/2023	**Name :** Hew Yung Fung **Student ID:** 0132646 **Email :** 0132646@kdu-online.com **Mobile No:** 0128826559 **Signature:** HYF **Date:** 18/04/2023	**Name :** **Student ID:** **Email :** **Mobile No:** **Signature:** **Date:**
Name : **Student ID:** **Email :** **Mobile No:** **Signature:** **Date:**	**Name :** **Student ID:** **Email :** **Mobile No:** **Signature:** **Date:**	**Name :** **Student ID:** **Email :** **Mobile No:** **Signature:** **Date:**

For office use only – Lecturer comments (if applicable)	**Marks Breakdown**

Table of Contents

Introduction **2**

Hypothesis **3**

Research Objective **3**

Research Question **4**

Methodology **4**

Data Collection 5

Data Preprocessing 8

Model Training : Naive Bayes 18

Model Training : Decision Tree 22

Model Training : Random Forest 24

Model Training : K Neighbours 26

Model Training : XGBoost 28

Result : Popularity Based Recommendation 29

Result : Model Comparison 32

Tableau (Data Visualization Dashboard) 35

Result and Discussion **43**

Conclusion **44**

Reference **45**

Introduction

The rapid growth of digital music streaming platforms has led to an increasing need for effective and personalised music recommendation systems (Cohen et al., 2018). As music libraries expand and the number of available tracks surges, finding new and relevant music becomes more challenging for users (Wang et al., 2019). This study aims to investigate the potential benefits of integrating user-oriented visualisation techniques with traditional recommendation algorithms to improve the accuracy and personalization of music recommendations. This research is significant because it can lead to a more engaging and personalised music listening experience for users.

Traditional recommendation algorithms, such as content-based or collaborative filtering methods, have been widely used to generate music recommendations based on individual listening habits (Schedl et al., 2018). However, these methods have certain limitations regarding accuracy and personalization, which can lead to less satisfactory user experiences (Jannach & Ludewig, 2018). User-oriented visualisation techniques have emerged as a promising approach to address these limitations and enhance the overall music recommendation process (Baur et al., 2020). The current state of the art in music recommendation research primarily focuses on traditional recommendation algorithms, with few studies exploring the potential of user-oriented visualisation techniques (Liu et al., 2021).

The hypothesis of this study is that the integration of user-oriented visualisation techniques with traditional recommendation algorithms in a hybrid approach will improve the accuracy and personalization of music recommendations based on individual listening habits, compared to content-based or collaborative filtering methods used alone. To accomplish this aim, we will create a dashboard using Tableau to visualise personal Spotify data and develop a recommendation engine to assess its compatibility with the data visualisation. The main results will highlight the benefits of the hybrid approach and its potential to provide users with a more engaging and personalised music listening experience.

Hypothesis

Hypothesis: The integration of user-oriented visualisation techniques with traditional recommendation algorithms in a hybrid approach will improve the accuracy and personalization of music recommendations based on individual listening habits, compared to content-based or collaborative filtering methods used alone.

Null hypothesis: There is no significant difference in the accuracy and personalization of music recommendations generated by a hybrid approach that combines user-oriented visualisation techniques with traditional recommendation algorithms, compared to content-based or collaborative filtering methods used alone.

Research Objective

1. Investigate the effectiveness of integrating user-oriented visualisation techniques with traditional recommendation algorithms.
2. Assess the accuracy and personalization improvements in the hybrid approach.
3. Examine the compatibility of the developed recommendation engine with data visualisation.
4. Evaluate user satisfaction and engagement with the enhanced music recommendation system.

Research Question

1. How can user-oriented visualisation techniques be effectively integrated with traditional recommendation algorithms in a music recommendation system?
2. To what extent does the hybrid approach improve the accuracy and personalization of music recommendations compared to content-based alone?
3. How compatible is the developed recommendation engine with the data visualisation provided by the Tableau dashboard?
4. Does the enhanced music recommendation system lead to a more engaging and personalised music listening experience for users?

Methodology

To investigate the potential benefits of integrating user-oriented visualisation techniques with traditional recommendation algorithms, the following methodology will be employed:

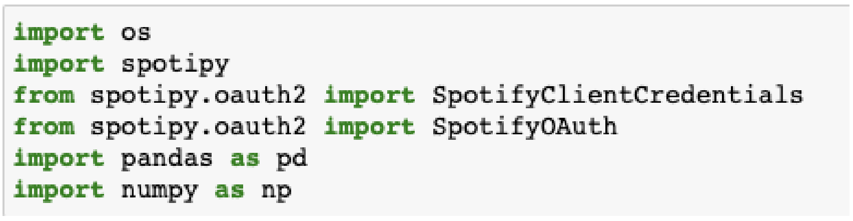
1. Data collection and preprocessing: Personal Spotify data, including listening habits, preferences, and feedback on music recommendations, will be collected using Spotify's API. The raw data will be cleaned and preprocessed to ensure its quality and relevance for further analysis. Relevant features like genre, artist, and track popularity will be extracted and organised for visualisation and recommendation engine development.
2. Data visualisation using Tableau: The preprocessed Spotify data will be visualised on a dashboard using Tableau, a powerful data visualisation tool. The dashboard will display various aspects of the user's listening habits and preferences, such as favourite genres, most listened-to artists, and track popularity. Interactive visualisations will allow users to explore their data and gain insights into their music preferences and patterns, providing a foundation for developing a recommendation engine.
3. Recommendation engine development: A recommendation engine will be created using traditional recommendation algorithms, such as content-based or collaborative filtering methods. The engine will generate personalised music recommendations based on the user's listening habits and preferences, as visualised in the Tableau dashboard. The algorithm's performance will be fine-tuned using cross-validation and optimization techniques to ensure accuracy and personalization.
4. Compatibility assessment: The compatibility of the recommendation engine with the data visualisation will be assessed by comparing the generated music recommendations with the user's listening habits and preferences as displayed on the Tableau dashboard. This assessment will measure the accuracy, precision, recall, and F1-score of the recommendations generated by the engine. Additionally, qualitative analysis will be conducted to evaluate the relevance and appeal of the recommendations to the user.
5. User feedback and evaluation: To further evaluate the success of the hybrid approach, user feedback on the generated music recommendations will be collected through surveys and interviews. This feedback will provide insights into user satisfaction and engagement with the recommendations provided by the hybrid approach. The feedback data will be analysed to identify the strengths and weaknesses of the hybrid approach, offering directions for future improvements.

6. Iterative refinement: The hybrid approach will undergo iterative refinement based on compatibility assessment and user feedback. The Tableau dashboard and recommendation engine will be adjusted and fine-tuned to improve their effectiveness in providing accurate and personalised music recommendations. The refined approach will be re-evaluated to ensure its performance and user satisfaction.

The choice of this methodology is based on its ability to offer a comprehensive evaluation of the hybrid approach, considering its effectiveness in generating accurate and personalised music recommendations and its impact on user experience. Combining data visualisation, recommendation engine development, compatibility assessment, and user feedback ensures a robust and thorough analysis of the hybrid approach's potential benefits.

Data Collection

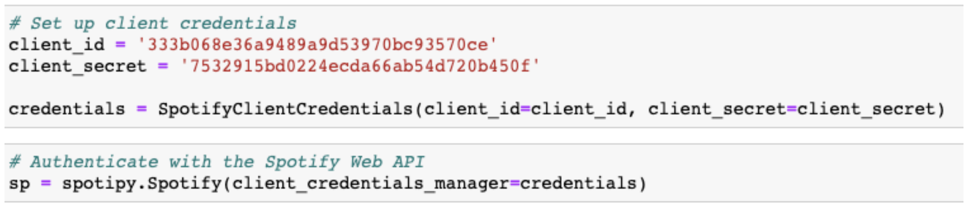
1. Importing necessary libraries:



```
import os
import spotipy
from spotipy.oauth2 import SpotifyClientCredentials
from spotipy.oauth2 import SpotifyOAuth
import pandas as pd
import numpy as np
```

Figure 1.0: Importing necessary libraries

2. Setting up client credentials and authenticating with the Spotify Web API:

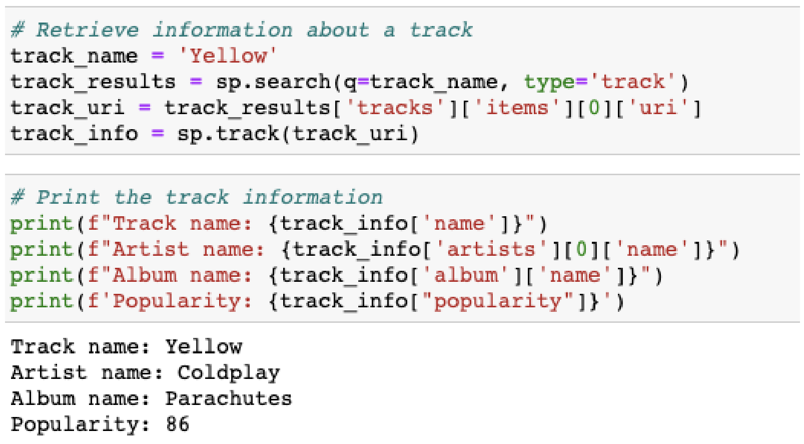


```
# Set up client credentials
client_id = '333b068e36a9489a9d53970bc93570ce'
client_secret = '7532915bd0224ecda66ab54d720b450f'

credentials = SpotifyClientCredentials(client_id=client_id, client_secret=client_secret)
```

```
# Authenticate with the Spotify Web API
sp = spotipy.Spotify(client_credentials_manager=credentials)
```

Figure 1.1: client credentials and authenticating with the Spotify Web API

3. Retrieving and printing track information for a specific track:



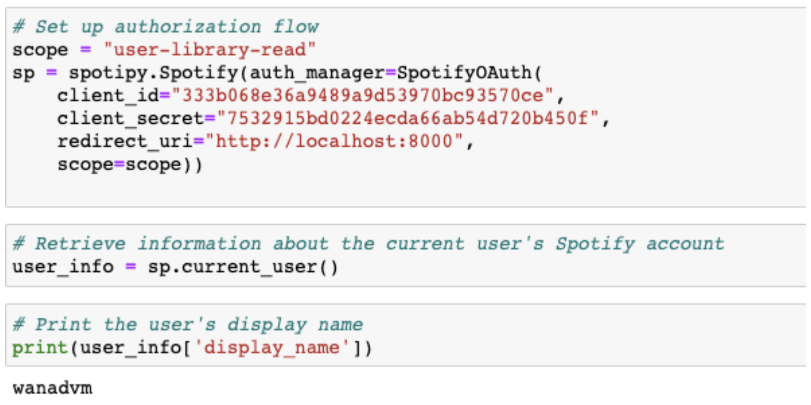
```python
# Retrieve information about a track
track_name = 'Yellow'
track_results = sp.search(q=track_name, type='track')
track_uri = track_results['tracks']['items'][0]['uri']
track_info = sp.track(track_uri)
```

```python
# Print the track information
print(f"Track name: {track_info['name']}")
print(f"Artist name: {track_info['artists'][0]['name']}")
print(f"Album name: {track_info['album']['name']}")
print(f'Popularity: {track_info["popularity"]}')
```

```
Track name: Yellow
Artist name: Coldplay
Album name: Parachutes
Popularity: 86
```

Figure 1.2: Retrieving and printing track information for a specific track

4. Setting up authorization flow to access user's library and retrieve user information:



```python
# Set up authorization flow
scope = "user-library-read"
sp = spotipy.Spotify(auth_manager=SpotifyOAuth(
    client_id="333b068e36a9489a9d53970bc93570ce",
    client_secret="7532915bd0224ecda66ab54d720b450f",
    redirect_uri="http://localhost:8000",
    scope=scope))
```

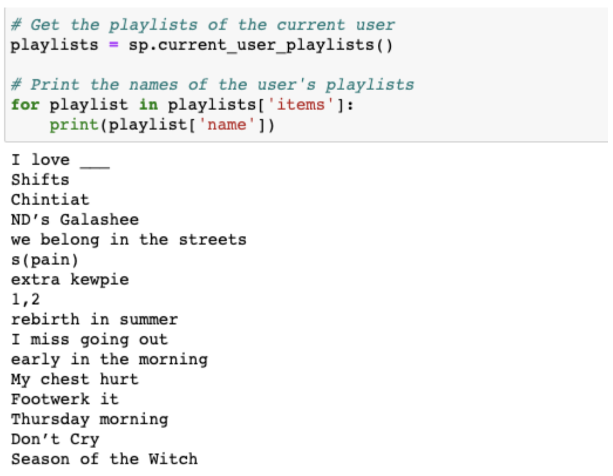
```python
# Retrieve information about the current user's Spotify account
user_info = sp.current_user()
```

```python
# Print the user's display name
print(user_info['display_name'])
```

```
wanadvm
```

Figure 1.3: Setting up authorization flow to access user's library and retrieve user information

5. Retrieving the user's playlists and printing their names:

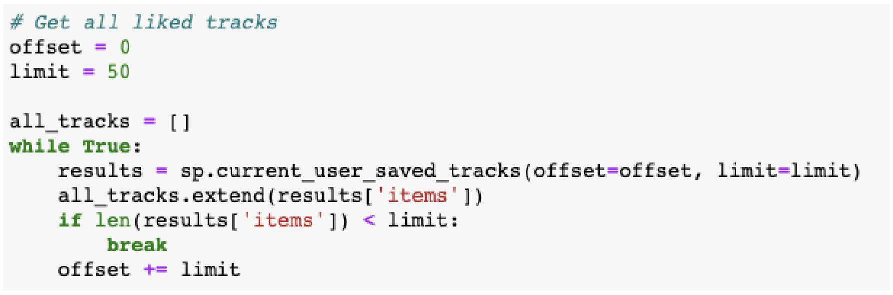


```python
# Get the playlists of the current user
playlists = sp.current_user_playlists()

# Print the names of the user's playlists
for playlist in playlists['items']:
    print(playlist['name'])
```

```
I love ___
Shifts
Chintiat
ND's Galashee
we belong in the streets
s(pain)
extra kewpie
1,2
rebirth in summer
I miss going out
early in the morning
My chest hurt
Footwerk it
Thursday morning
Don't Cry
Season of the Witch
```

Figure 1.4: Retrieving the user's playlists and printing their names.

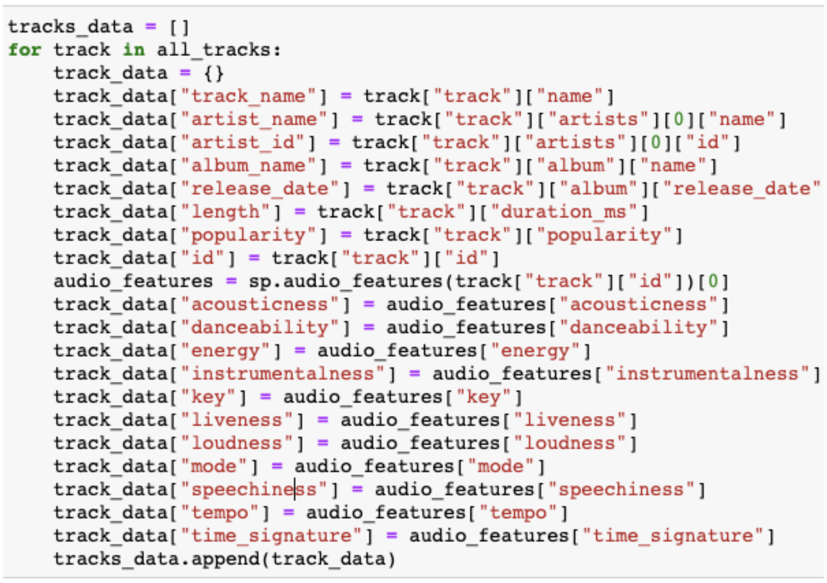
6. Retrieving the user's liked tracks and extracting relevant information:



```
# Get all liked tracks
offset = 0
limit = 50

all_tracks = []
while True:
    results = sp.current_user_saved_tracks(offset=offset, limit=limit)
    all_tracks.extend(results['items'])
    if len(results['items']) < limit:
        break
    offset += limit
```

Figure 1.5: Retrieving the user's liked tracks and extracting relevant information.

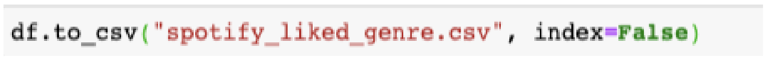
7. Retrieving audio features for the liked tracks and creating a DataFrame:



```
tracks_data = []
for track in all_tracks:
    track_data = {}
    track_data["track_name"] = track["track"]["name"]
    track_data["artist_name"] = track["track"]["artists"][0]["name"]
    track_data["artist_id"] = track["track"]["artists"][0]["id"]
    track_data["album_name"] = track["track"]["album"]["name"]
    track_data["release_date"] = track["track"]["album"]["release_date"
    track_data["length"] = track["track"]["duration_ms"]
    track_data["popularity"] = track["track"]["popularity"]
    track_data["id"] = track["track"]["id"]
    audio_features = sp.audio_features(track["track"]["id"])[0]
    track_data["acousticness"] = audio_features["acousticness"]
    track_data["danceability"] = audio_features["danceability"]
    track_data["energy"] = audio_features["energy"]
    track_data["instrumentalness"] = audio_features["instrumentalness"]
    track_data["key"] = audio_features["key"]
    track_data["liveness"] = audio_features["liveness"]
    track_data["loudness"] = audio_features["loudness"]
    track_data["mode"] = audio_features["mode"]
    track_data["speechiness"] = audio_features["speechiness"]
    track_data["tempo"] = audio_features["tempo"]
    track_data["time_signature"] = audio_features["time_signature"]
    tracks_data.append(track_data)
```

Figure 1.6: Retrieving audio features for the liked tracks and creating a DataFrame

8. Creating a DataFrame for liked tracks

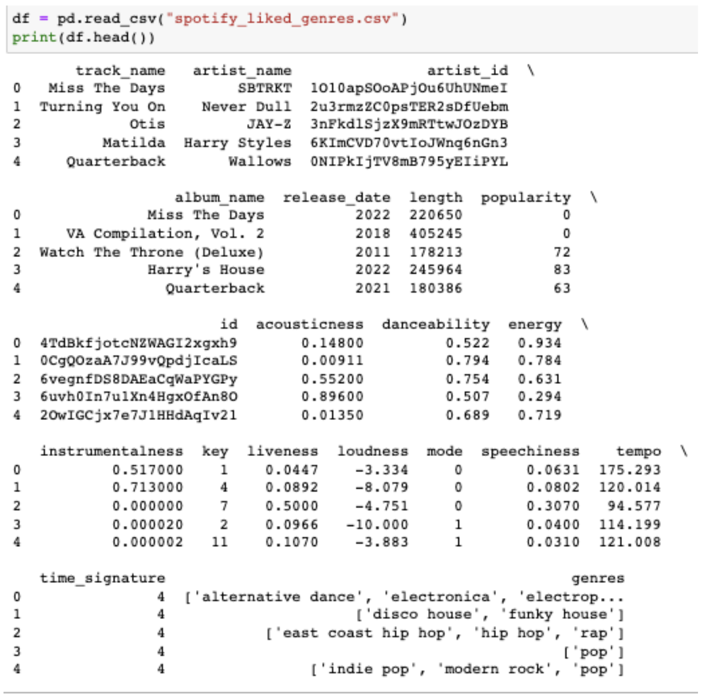


```
df.to_csv("spotify_liked_genre.csv", index=False)
```

Figure 1.7: Creating a DataFrame for liked tracks

Data Preprocessing

1. Loading the extracted data into a data frame and generating summary statistics.



```
df = pd.read_csv("spotify_liked_genres.csv")
print(df.head())
```

```
     track_name   artist_name               artist_id  \
0  Miss The Days        SBTRKT  1O1OapSOoAPjOu6UhUNmeI
1  Turning You On    Never Dull  2u3rmzZC0psTER2sDfUebm
2           Otis         JAY-Z  3nFkdlSjzX9mRTtwJOzDYB
3        Matilda  Harry Styles  6KImCVD70vtIoJWnq6nGn3
4    Quarterback       Wallows  0NIPkIjTV8mB795yEIiPYL

                album_name  release_date  length  popularity  \
0            Miss The Days          2022  220650           0
1     VA Compilation, Vol. 2          2018  405245           0
2  Watch The Throne (Deluxe)          2011  178213          72
3            Harry's House          2022  245964          83
4              Quarterback          2021  180386          63

                       id  acousticness  danceability  energy  \
0  4TdBkfjotcNZWAGI2xgxh9       0.14800         0.522   0.934
1  0CgQOzaA7J99vQpdjIcaLS       0.00911         0.794   0.784
2  6vegnfDS8DAEaCqWaPYGPy       0.55200         0.754   0.631
3  6uvh0In7ulXn4HgxOfAn8O       0.89600         0.507   0.294
4  2OwIGCjx7e7J1HHdAqIv21       0.01350         0.689   0.719

   instrumentalness  key  liveness  loudness  mode  speechiness    tempo  \
0          0.517000    1    0.0447    -3.334     0       0.0631  175.293
1          0.713000    4    0.0892    -8.079     0       0.0802  120.014
2          0.000000    7    0.5000    -4.751     0       0.3070   94.577
3          0.000020    2    0.0966   -10.000     1       0.0400  114.199
4          0.000002   11    0.1070    -3.883     1       0.0310  121.008

   time_signature                                             genres
0               4  ['alternative dance', 'electronica', 'electrop...
1               4                       ['disco house', 'funky house']
2               4              ['east coast hip hop', 'hip hop', 'rap']
3               4                                            ['pop']
4               4                 ['indie pop', 'modern rock', 'pop']
```

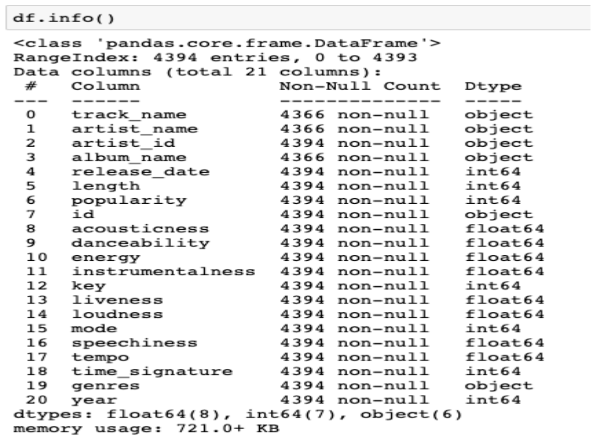
Figure 2.0: Loading the extracted data into a data frame and generating summary statistics.

The summary provides various statistics for the dataset containing information about liked songs on Spotify. The dataset has 4394 songs, and each column represents a specific feature of the songs.

- release_date: The year the song was released. The mean release year is 1997, and the years range from 0 to 2022. 50% of the songs were released in 2016 or earlier.
- length: The duration of the song in milliseconds. On average, the songs are about 248,649 ms (around 4 minutes) long, with the shortest being 0 ms and the longest being 3,815,786 ms (over an hour).
- popularity: The popularity score of the song (from 0 to 100). The average popularity is 22.81, with a range from 0 to 91.
- acousticness: A measure of how acoustic a song is (from 0 to 1). The average acousticness is 0.31, indicating that the songs are moderately acoustic on average.
- danceability: A measure of how suitable a song is for dancing (from 0 to 1). The average danceability score is 0.59, suggesting that the songs are moderately danceable.

- energy: A measure of the intensity and activity of a song (from 0 to 1). The average energy score is 0.59, indicating that the songs have moderate energy levels.
- instrumentalness: A measure of how instrumental a song is (from 0 to 1). The average instrumentalness is 0.23, suggesting that the songs have a moderate amount of instrumental content.
- key: The musical key of the song (from 0 to 11). The mean key is 5.27, indicating that the songs are distributed across various keys.
- liveness: A measure of the presence of a live audience in the song (from 0 to 1). The average liveness is 0.20, suggesting that the songs generally have low audience presence.
- loudness: The overall loudness of a song in decibels (dB). The average loudness is -9.43 dB, with a range from -39.52 dB to 1.34 dB.
- mode: The modality of the song (0 for minor, 1 for major). The mean mode is 0.58, indicating that a slightly higher proportion of songs are in a major key.
- speechiness: A measure of the presence of spoken words in a song (from 0 to 1). The average speechiness is 0.12, suggesting that the songs have a low amount of spoken words.
- tempo: The tempo of the song in beats per minute (BPM). The average tempo is 118.72 BPM, with a range from 0 to 244.03 BPM.
- time_signature: The number of beats in a measure. The average time signature is 3.92, with a range from 0 to 5.
- These statistics provide insights into the general characteristics of the liked songs, which can be useful for understanding user preferences or for further analysis, such as clustering or classification tasks.

2. Viewing the the data information:



```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4394 entries, 0 to 4393
Data columns (total 21 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   track_name        4366 non-null   object
 1   artist_name       4366 non-null   object
 2   artist_id         4394 non-null   object
 3   album_name        4366 non-null   object
 4   release_date      4394 non-null   int64
 5   length            4394 non-null   int64
 6   popularity        4394 non-null   int64
 7   id                4394 non-null   object
 8   acousticness      4394 non-null   float64
 9   danceability      4394 non-null   float64
 10  energy            4394 non-null   float64
 11  instrumentalness  4394 non-null   float64
 12  key               4394 non-null   int64
 13  liveness          4394 non-null   float64
 14  loudness          4394 non-null   float64
 15  mode              4394 non-null   int64
 16  speechiness       4394 non-null   float64
 17  tempo             4394 non-null   float64
 18  time_signature    4394 non-null   int64
 19  genres            4394 non-null   object
 20  year              4394 non-null   int64
dtypes: float64(8), int64(7), object(6)
memory usage: 721.0+ KB
```

Figure 2.1: Viewing the data information.

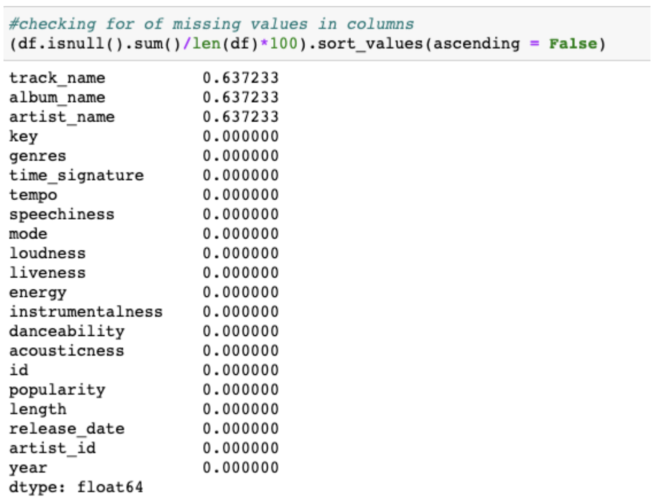
This output provides information about a DataFrame with 4394 rows (entries) and 21 columns (features). It also shows the data type of each column and the number of non-null values.

Here's a brief explanation of each column:

- track_name (object): The name of the song.
- artist_name (object): The name of the artist.
- artist_id (object): The unique identifier of the artist on Spotify.
- album_name (object): The name of the album the song belongs to.
- release_date (int64): The year the song was released.
- length (int64): The duration of the song in milliseconds.
- popularity (int64): The popularity score of the song on Spotify (0-100).
- id (object): The unique identifier of the song on Spotify.
- acousticness (float64): A measure of the song's acousticness (0-1).
- danceability (float64): A measure of the song's danceability (0-1).
- energy (float64): A measure of the song's energy (0-1).
- instrumentalness (float64): A measure of the song's instrumentalness (0-1).
- key (int64): The musical key of the song (0-11).
- liveness (float64): A measure of the song's liveness (0-1).
- loudness (float64): The overall loudness of the song in decibels (dB).
- mode (int64): The modality of the song (0 for minor, 1 for major).
- speechiness (float64): A measure of the song's speechiness (0-1).
- tempo (float64): The tempo of the song in beats per minute (BPM).
- time_signature (int64): The number of beats in a measure.
- genres (object): The genres associated with the song.
- year (int64): The same as release_date.

The DataFrame has some missing values in the 'track_name', 'artist_name', and 'album_name' columns. The total memory usage of this DataFrame is approximately 721.0 KB.

3. Checking for missing values in columns

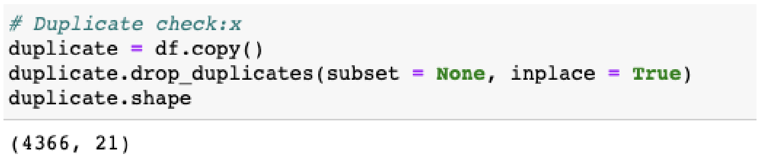


```
#checking for of missing values in columns
(df.isnull().sum()/len(df)*100).sort_values(ascending = False)
```

```
track_name          0.637233
album_name          0.637233
artist_name         0.637233
key                 0.000000
genres              0.000000
time_signature      0.000000
tempo               0.000000
speechiness         0.000000
mode                0.000000
loudness            0.000000
liveness            0.000000
energy              0.000000
instrumentalness    0.000000
danceability        0.000000
acousticness        0.000000
id                  0.000000
popularity          0.000000
length              0.000000
release_date        0.000000
artist_id           0.000000
year                0.000000
dtype: float64
```

Figure 2.2: Viewing the data information.

4. Duplicate check

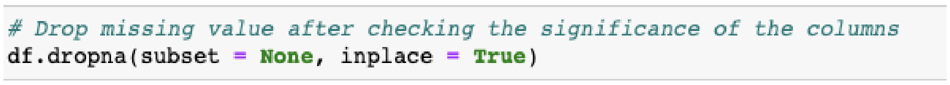


```
# Duplicate check:x
duplicate = df.copy()
duplicate.drop_duplicates(subset = None, inplace = True)
duplicate.shape
```

```
(4366, 21)
```

Figure 2.3: Duplication check

5. Drop missing values after checking the significance

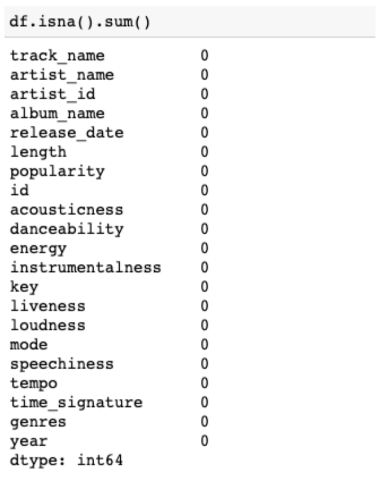


```
# Drop missing value after checking the significance of the columns
df.dropna(subset = None, inplace = True)
```

Figure 2.4: Dropping missing values.

The missing values were from the columns track_name, album_name, and artist_name which failed to retrieve during the data extraction process. Dropping the missing values has no significant effect on the model that we are going to train.

6. Checking for missing values again



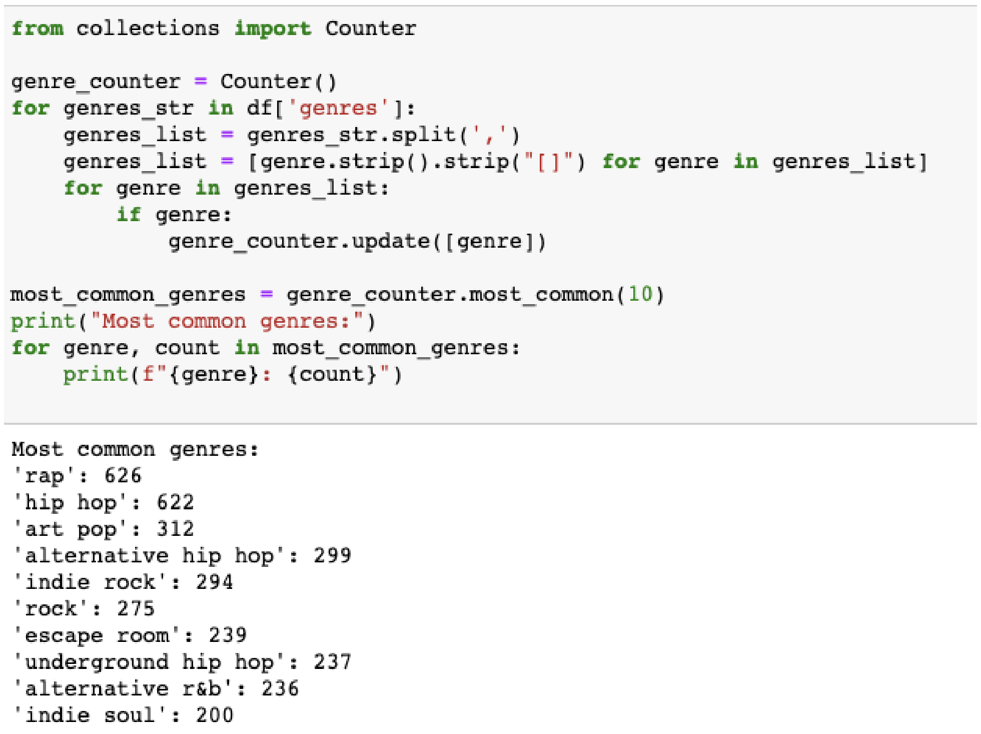
```
df.isna().sum()
```

```
track_name          0
artist_name         0
artist_id           0
album_name          0
release_date        0
length              0
popularity          0
id                  0
acousticness        0
danceability        0
energy              0
instrumentalness    0
key                 0
liveness            0
loudness            0
mode                0
speechiness         0
tempo               0
time_signature      0
genres              0
year                0
dtype: int64
```

Figure 2.5: Checking for missing value after dropping missing values.

This step is done to ensure that all the missing values have been dropped.

7. Gathering genre for each songs



```
from collections import Counter

genre_counter = Counter()
for genres_str in df['genres']:
    genres_list = genres_str.split(',')
    genres_list = [genre.strip().strip("[]") for genre in genres_list]
    for genre in genres_list:
        if genre:
            genre_counter.update([genre])

most_common_genres = genre_counter.most_common(10)
print("Most common genres:")
for genre, count in most_common_genres:
    print(f"{genre}: {count}")
```

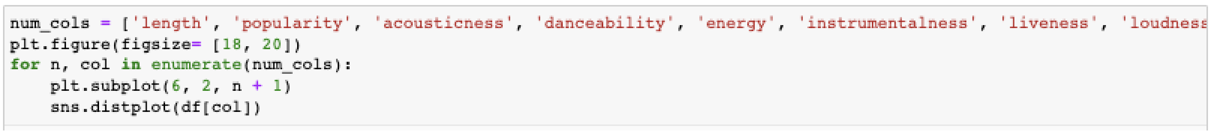
```
Most common genres:
'rap': 626
'hip hop': 622
'art pop': 312
'alternative hip hop': 299
'indie rock': 294
'rock': 275
'escape room': 239
'underground hip hop': 237
'alternative r&b': 236
'indie soul': 200
```

Figure 2.6: Gathering genre of each song.

This code snippet aims to identify the top 10 most common genres in the DataFrame, specifically in the 'genres' column. To achieve this, the following steps are taken:

- Import the Counter class from the collections module. The Counter class is a specialised dictionary that allows for counting occurrences of elements in a collection.
- Initialize an instance of the Counter class named 'genre_counter'. This object will store the counts of each unique genre in the DataFrame.
- Iterate over the 'genres' column of the DataFrame using a for loop. The following steps are executed for each entry in the column: a. Split the 'genres_str' string into a list of individual genres using the, '' delimiter. b. Clean up the list of genres by removing any extra whitespace and square brackets using a list comprehension.
- Within the outer loop, another for loop iterates over the cleaned genres list. If a genre is not empty, the 'genre_counter' object is updated with the genre using the 'update' method. This increments the count for the genre in the Counter object.
- After processing all the rows in the DataFrame, the 'most_common' method is called on the 'genre_counter' object to retrieve the ten most common genres and their respective counts. The result is stored in the 'most_common_genres' variable.
- Finally, the code prints the top 10 most common genres and their respective counts using a for loop.

8. Plotting the data distribution of each columns



```
num_cols = ['length', 'popularity', 'acousticness', 'danceability', 'energy', 'instrumentalness', 'liveness', 'loudness
plt.figure(figsize= [18, 20])
for n, col in enumerate(num_cols):
    plt.subplot(6, 2, n + 1)
    sns.distplot(df[col])
```

Figure 2.7: Plotting the data distribution of each columns.

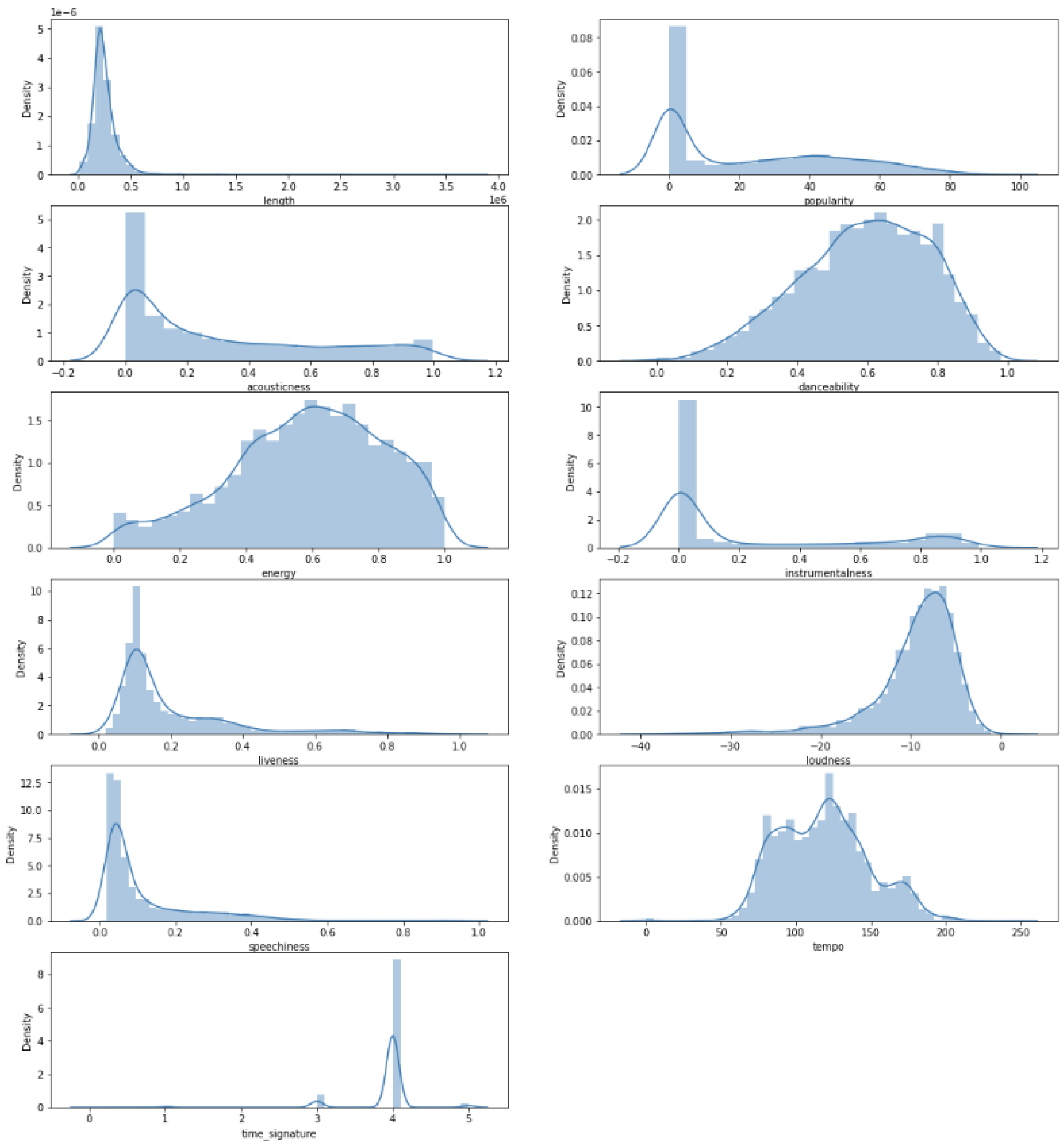


Figure 2.8: Plotting the data distribution of each columns.

The code snippet provided aims to visualise the distribution of various numerical features in the data frame. To achieve this, the following steps are taken. Firstly, a list named 'num_cols' is created to store the names of the numerical columns of interest, which includes 'length', 'popularity', 'acousticness', 'danceability', 'energy,' 'instrumentalness,' 'liveness,' 'loudness,' 'speechiness,' 'tempo,' and 'time_signature.' Subsequently, a new figure is created using the 'plt. figure()' function, with dimensions set to 18 inches in width and 20 inches in height, to accommodate multiple subplots.

An iteration is then performed over the 'num_cols' list using a for loop, where both the index and the column name are retrieved using the 'enumerate()' function. Inside the loop, a new subplot is created for each numerical column using the 'plt.subplot()' function. The function takes three arguments: the number of rows (6), the number of columns (2), and the position of the current subplot (n + 1), where 'n' is the index of the current column in the 'num_cols' list.

Once the subplot is initialised, a distribution plot is generated for the current numerical column using the Seaborn library's 'sns.distplot()' function. The function takes the current column data as its argument, accessed from the DataFrame using the column name ('col'). The distribution plot is then displayed in the corresponding subplot.

This visualisation allows for an in-depth examination of the data, providing valuable insights into the characteristics of the dataset and helping to identify potential trends or outliers.

9. Visualising the Top 20 Most Popular Artists in the Dataset

```
# # Plotting
fig, ax = plt.subplots(figsize = (12, 10))
lead_artists = df.groupby('artist_name')['popularity'].sum().sort_values(ascending=False).head(20)
ax = sns.barplot(x=lead_artists.values, y=lead_artists.index, palette="Blues", orient="h", edgecolor='black', ax=ax)
ax.set_xlabel('Sum of Popularity', fontsize=12)
ax.set_ylabel('Artist', fontsize=12)
ax.set_title('20 Most Popular Artists in Dataset', fontsize=14)
plt.show()
```

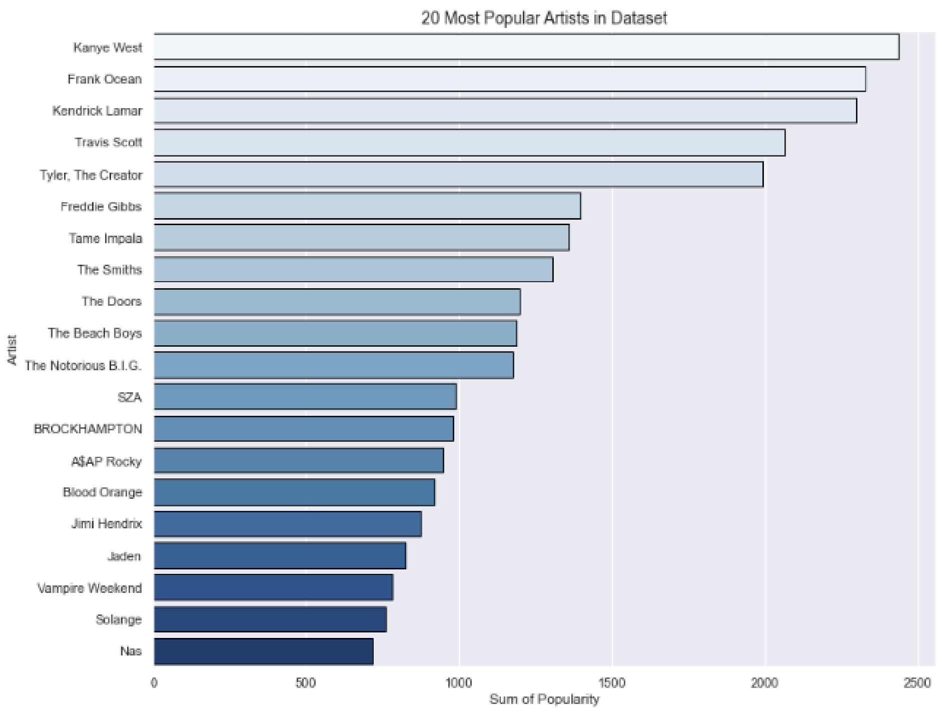


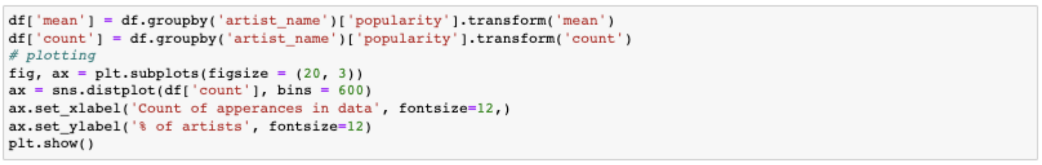
Figure 2.9: 20 most popular artists in the dataset.

In this step, the goal is to visualise the 20 most popular artists in the dataset based on the sum of the popularity of their songs. The code first groups the dataset by artist name and calculates the sum of the popularity metric for each artist. Then, it sorts the artists in descending order of popularity and selects the top 20 popular artists.

The visualisation is created using the Python library Seaborn, which generates a horizontal bar plot with the artists' names on the y-axis and their total popularity on the x-axis. The plot is customised with a blue colour palette, edge colouring for better contrast, and appropriate label and title font sizes. The resulting plot effectively communicates the 20 most popular artists in the dataset, clearly representing their relative popularity.

As you can see from the plot, we could see that the most popular artist in the dataset is Kanye West followed by Frank Ocean, Kendrick Lamar and lastly Nas. This plot will help us get a better understanding of what type of songs that will be recommended by the recommendation engine.

10. Visualising the Distribution of Artist Appearances



```
df['mean'] = df.groupby('artist_name')['popularity'].transform('mean')
df['count'] = df.groupby('artist_name')['popularity'].transform('count')
# plotting
fig, ax = plt.subplots(figsize = (20, 3))
ax = sns.distplot(df['count'], bins = 600)
ax.set_xlabel('Count of apperances in data', fontsize=12,)
ax.set_ylabel('% of artists', fontsize=12)
plt.show()
```

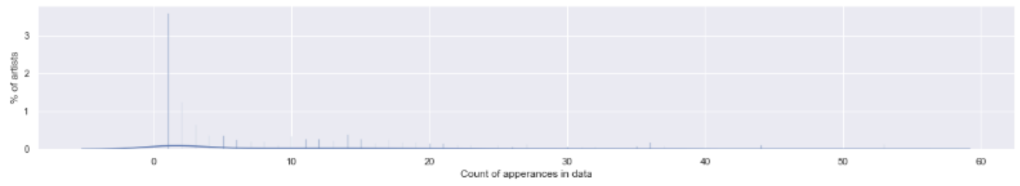


Figure 2.1: Visualising the distribution of artist appearance.

The purpose of this code is to visualise the distribution of artist appearances in the dataset. This is important as it helps in understanding the frequency of different artists and the composition of the dataset.

The resulting visualisation allows for examining the distribution of artist appearances in the dataset. This can provide insights into the diversity of the dataset and help identify any potential biases or imbalances in terms of artist representation.

11. Visualising the Relationship between Artist Appearances and Mean Popularity

```
fig, ax = plt.subplots(figsize = (15, 3))
stat = df.groupby('count')['mean'].mean().to_frame().reset_index()
ax = stat.plot(x='count', y='mean', marker='.', linestyle = '', ax=ax)
ax.set_xlabel('Count of appearances in data', fontsize=12)
ax.set_ylabel('Mean Popularity', fontsize=12)
plt.show()
```

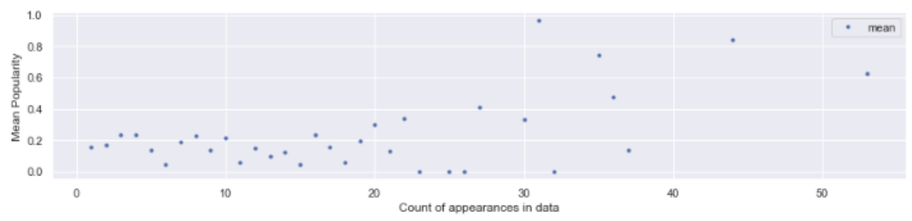


Figure 2.1: Mean popularity against appearances in data.

This code snippet primarily aims to investigate the relationship between the number of artist appearances in the dataset and their average popularity. Understanding this relationship can provide insights into how the frequency of an artist's presence in the dataset might be associated with their popularity.

To achieve this goal, a new data frame is created that groups the data by the 'count' column, which represents the number of appearances for each artist. The mean popularity of each group is then calculated and stored in a new column named 'mean.' The 'to_frame()' and 'reset_index()' functions convert the resulting Series into a DataFrame and reset its index.

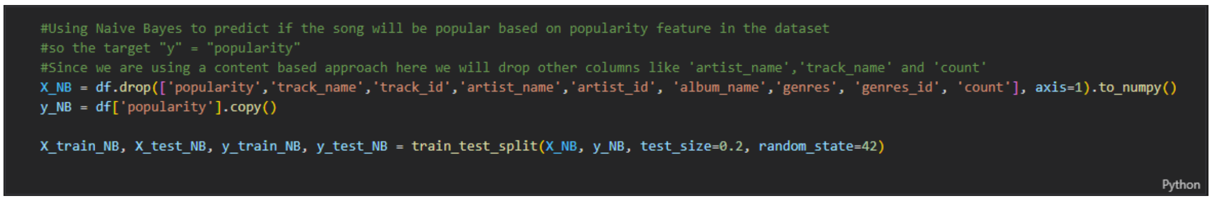
A scatter plot is then generated using the 'plot()' function with the 'count' column as the x-axis and the 'mean' column as the y-axis. The plot is created with markers only and no connecting lines, signified by the 'marker' and 'linestyle' arguments.

The x-axis label is set to 'Count of appearances in data', while the y-axis label is set to 'Mean Popularity', with a font size of 12. Finally, the plot is displayed using the 'plt.show()' function.

The resulting visualisation helps to explore the association between the frequency of artist appearances in the dataset and their mean popularity. This can be useful for understanding potential patterns or trends in the data and informing subsequent analysis or decision-making.

Model Training : Naive Bayes

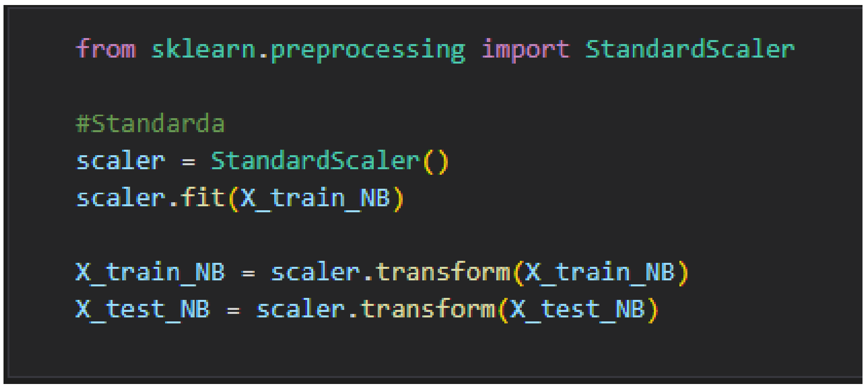
1. Splitting the data into training and testing subsets using scikit-learn's train_test_split() function. The resulting arrays are then ready to be used for training and testing the classifier model.



```python
#Using Naive Bayes to predict if the song will be popular based on popularity feature in the dataset
#so the target "y" = "popularity"
#Since we are using a content based approach here we will drop other columns like 'artist_name','track_name' and 'count'
X_NB = df.drop(['popularity','track_name','track_id','artist_name','artist_id', 'album_name','genres', 'genres_id', 'count'], axis=1).to_numpy()
y_NB = df['popularity'].copy()

X_train_NB, X_test_NB, y_train_NB, y_test_NB = train_test_split(X_NB, y_NB, test_size=0.2, random_state=42)
```

Figure 3.1: Splitting the data.

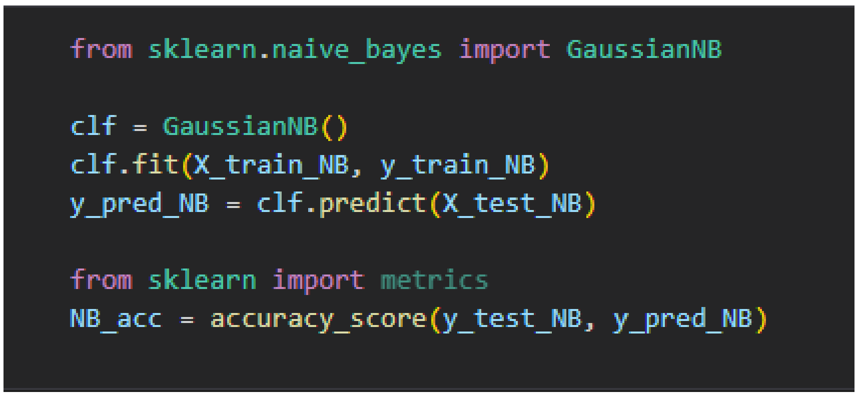
2. Standardise the data to have zero mean and unit variance, which can help improve the performance of certain machine learning algorithms, such as those that rely on distance calculations. This step is often recommended for data that has features with different scales or units.



```python
from sklearn.preprocessing import StandardScaler

#Standarda
scaler = StandardScaler()
scaler.fit(X_train_NB)

X_train_NB = scaler.transform(X_train_NB)
X_test_NB = scaler.transform(X_test_NB)
```

Figure 3.2: Standardize the data.

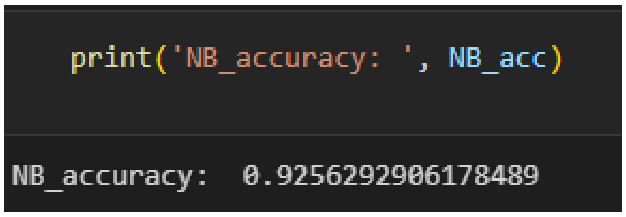
3. Train a Naive Bayes classifier model on the training data, and evaluate its performance on the testing data.



```python
from sklearn.naive_bayes import GaussianNB

clf = GaussianNB()
clf.fit(X_train_NB, y_train_NB)
y_pred_NB = clf.predict(X_test_NB)

from sklearn import metrics
NB_acc = accuracy_score(y_test_NB, y_pred_NB)
```

Figure 3.3: Feed the model with data.

4. Result of the Naive Bayes Accuracy is 0.9256292906178489

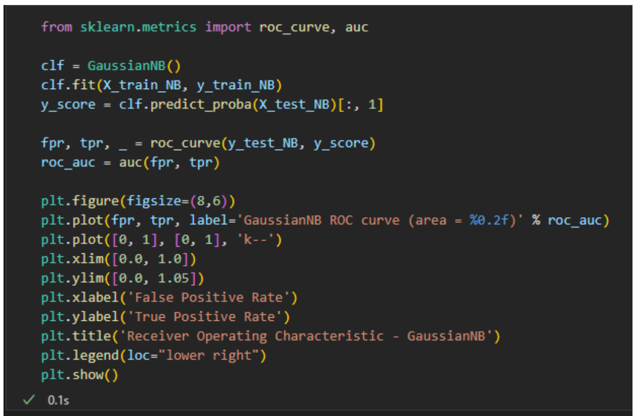


```
print('NB_accuracy: ', NB_acc)
```

```
NB_accuracy:  0.9256292906178489
```

Figure 3.4: Result of the model.

5. This code trains an GaussianNB model using the training data X_train_NB and y_train_NB, and then plots the ROC Curve during training.



```
from sklearn.metrics import roc_curve, auc

clf = GaussianNB()
clf.fit(X_train_NB, y_train_NB)
y_score = clf.predict_proba(X_test_NB)[:, 1]

fpr, tpr, _ = roc_curve(y_test_NB, y_score)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8,6))
plt.plot(fpr, tpr, label='GaussianNB ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic - GaussianNB')
plt.legend(loc="lower right")
plt.show()
```

✓ 0.1s

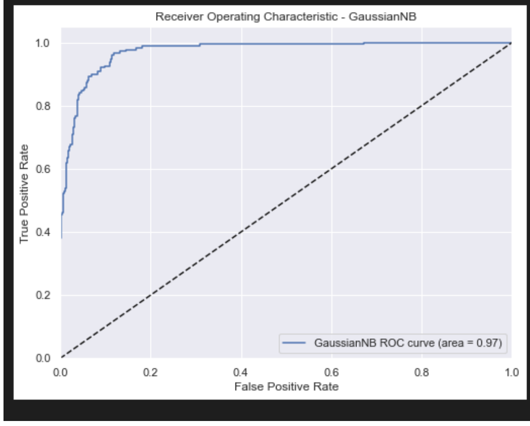
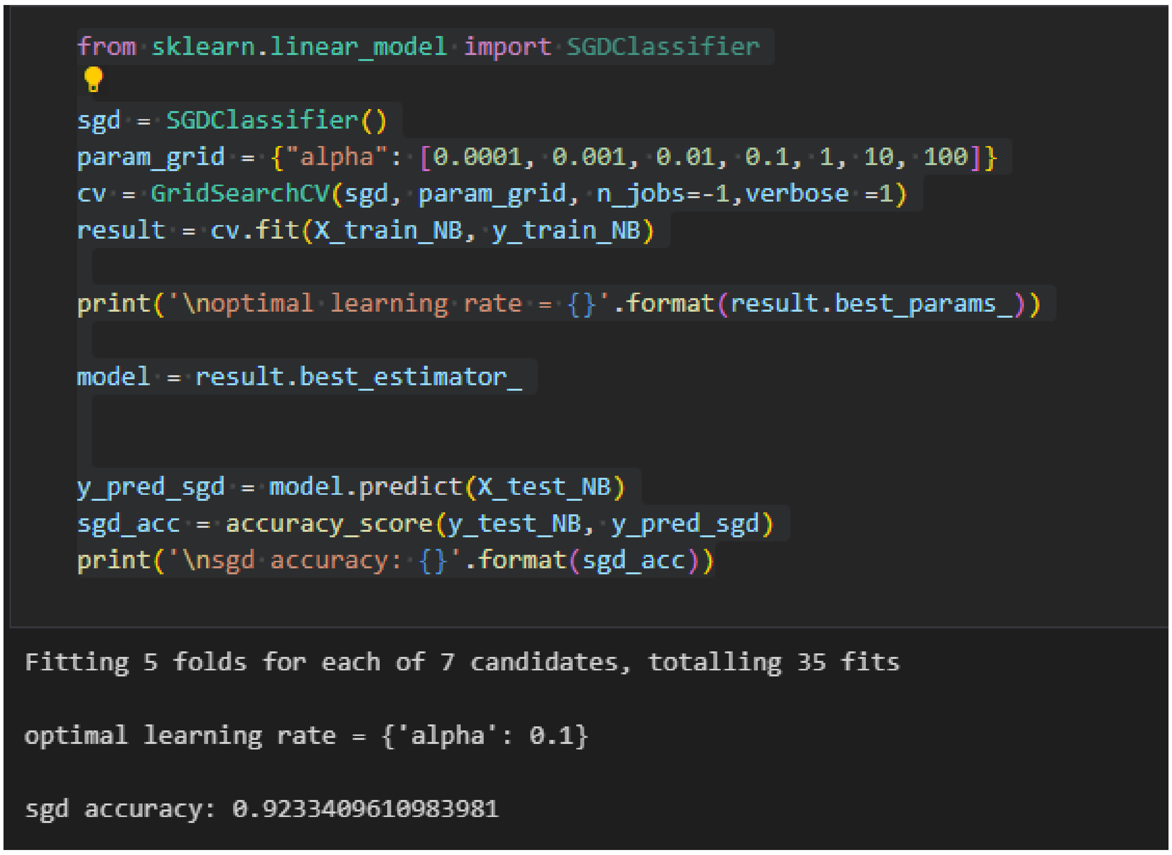


Figure 3.5: Plot the model performance with ROC Curve.

This graph shows the trade-off between true positive rate and false positive rate. If the ROC curve is close to the top left corner, it indicates that the model is performing well.

Model Training : Stochastic Gradient Descent

1. Train a linear classifier model using the SGDClassifier class, perform hyperparameter tuning to find the optimal hyperparameters for the model using GridSearchCV, and evaluate the performance of the model on the testing data. The accuracy of the model on the testing data is calculated as a measure of its performance.



```python
from sklearn.linear_model import SGDClassifier

sgd = SGDClassifier()
param_grid = {"alpha": [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]}
cv = GridSearchCV(sgd, param_grid, n_jobs=-1,verbose =1)
result = cv.fit(X_train_NB, y_train_NB)

print('\noptimal learning rate = {}'.format(result.best_params_))

model = result.best_estimator_


y_pred_sgd = model.predict(X_test_NB)
sgd_acc = accuracy_score(y_test_NB, y_pred_sgd)
print('\nsgd accuracy: {}'.format(sgd_acc))
```

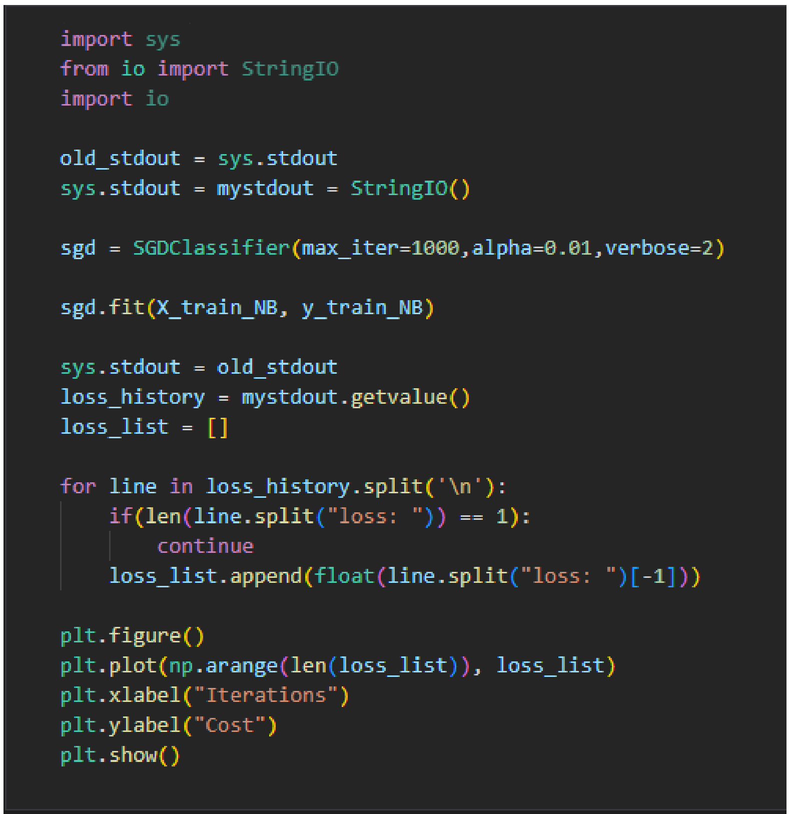
```
Fitting 5 folds for each of 7 candidates, totalling 35 fits

optimal learning rate = {'alpha': 0.1}

sgd accuracy: 0.9233409610983981
```

Figure 3.6: Feed the model with data and show result.

The optimal hyperparameter setting found was an alpha value of 0.1. This hyperparameter likely belongs to a model that uses regularisation techniques, such as Ridge or Lasso regression, to prevent overfitting.

The model achieved an accuracy of 0.9233, which means that it correctly classified 92.33% of the samples in the dataset. The AUC value of 0.8564 suggests that the model is reasonably good at distinguishing between the positive and negative classes, where the positive class is the one that the model is trying to predict.

2. This code trains an SGDClassifier model using the training data X_train_NB and y_train_NB, and then plots the cost function (loss) history during training.



```python
import sys
from io import StringIO
import io

old_stdout = sys.stdout
sys.stdout = mystdout = StringIO()

sgd = SGDClassifier(max_iter=1000,alpha=0.01,verbose=2)

sgd.fit(X_train_NB, y_train_NB)

sys.stdout = old_stdout
loss_history = mystdout.getvalue()
loss_list = []

for line in loss_history.split('\n'):
    if(len(line.split("loss: ")) == 1):
        continue
    loss_list.append(float(line.split("loss: ")[-1]))

plt.figure()
plt.plot(np.arange(len(loss_list)), loss_list)
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.show()
```

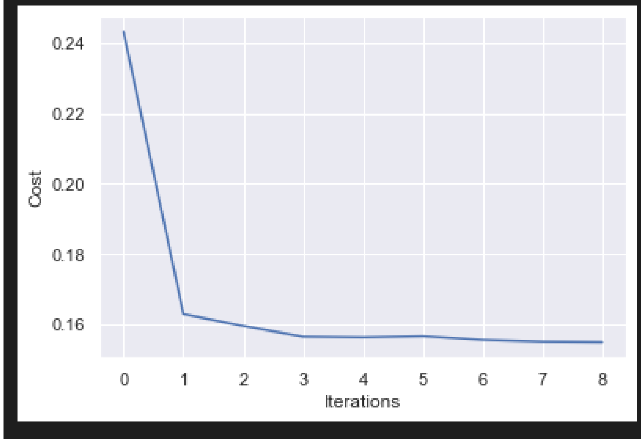
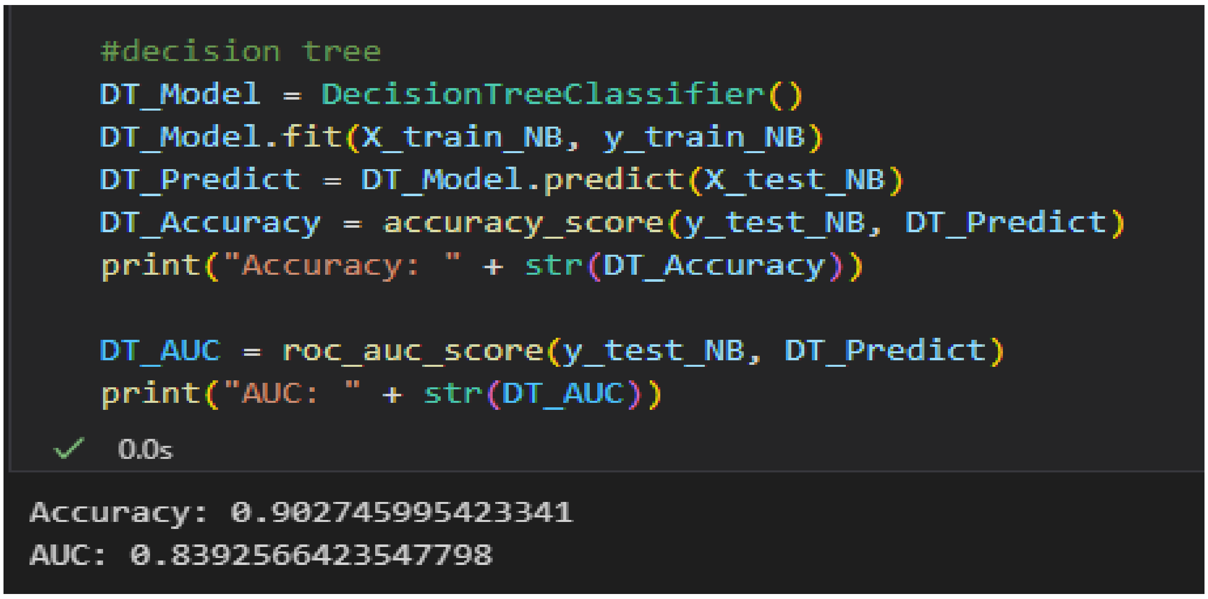


Figure 3.7: Plot the model performance with Loss Function Graph.

If the cost function converges to a low value, then it suggests that the classifier has learned a good decision boundary that separates the different classes in the data and in this graph we can see that the model can classify the dataset quite well but not as good as Naive Base model.

Model Training : Decision Tree

1. This code trains a decision tree classifier on a dataset, evaluates the model's performance using the accuracy and AUC metrics, and prints the results to the console.



```
#decision tree
DT_Model = DecisionTreeClassifier()
DT_Model.fit(X_train_NB, y_train_NB)
DT_Predict = DT_Model.predict(X_test_NB)
DT_Accuracy = accuracy_score(y_test_NB, DT_Predict)
print("Accuracy: " + str(DT_Accuracy))

DT_AUC = roc_auc_score(y_test_NB, DT_Predict)
print("AUC: " + str(DT_AUC))
```

✓ 0.0s

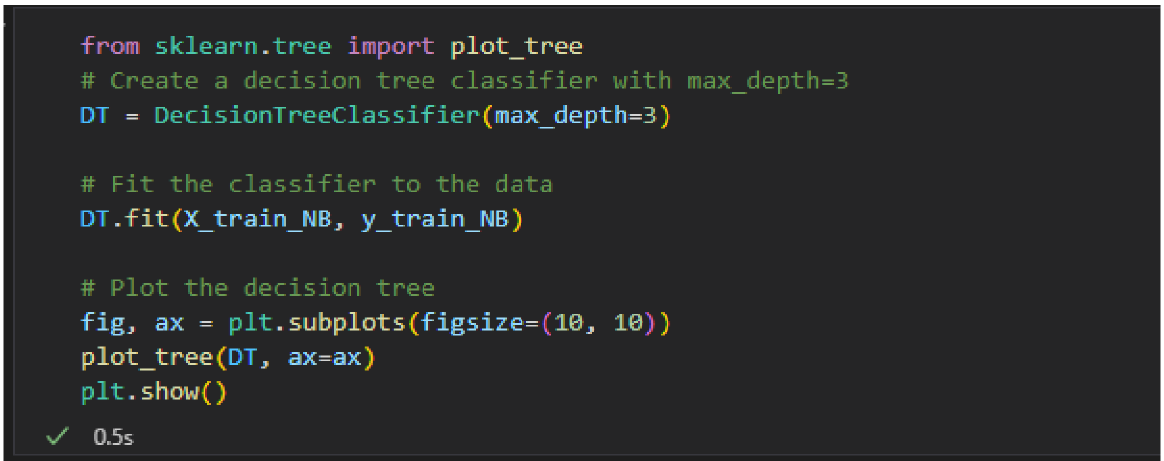
```
Accuracy: 0.9027459954233411
AUC: 0.8392566423547798
```

Figure 3.8: Feed the model with data and show result.

The DecisionTreeClassifier class is used to create a decision tree model, which is then fit to the training data using the fit method. The predict method is then used to generate predictions for the test data, and the accuracy_score function and roc_auc_score function are used to calculate the accuracy and AUC of the model's predictions, respectively.

Finally, the results are printed to the console using the print function. It is assumed that the training and test data and labels are already defined and stored in the variables X_train_NB, y_train_NB, X_test_NB, and y_test_NB.

2. The plot_tree function in scikit-learn is used to visualise the decision tree used by a decision tree classifier or regressor. It can be helpful in understanding how the model makes decisions and identifying potential areas for improvement.



```python
from sklearn.tree import plot_tree
# Create a decision tree classifier with max_depth=3
DT = DecisionTreeClassifier(max_depth=3)

# Fit the classifier to the data
DT.fit(X_train_NB, y_train_NB)

# Plot the decision tree
fig, ax = plt.subplots(figsize=(10, 10))
plot_tree(DT, ax=ax)
plt.show()
```

✓ 0.5s

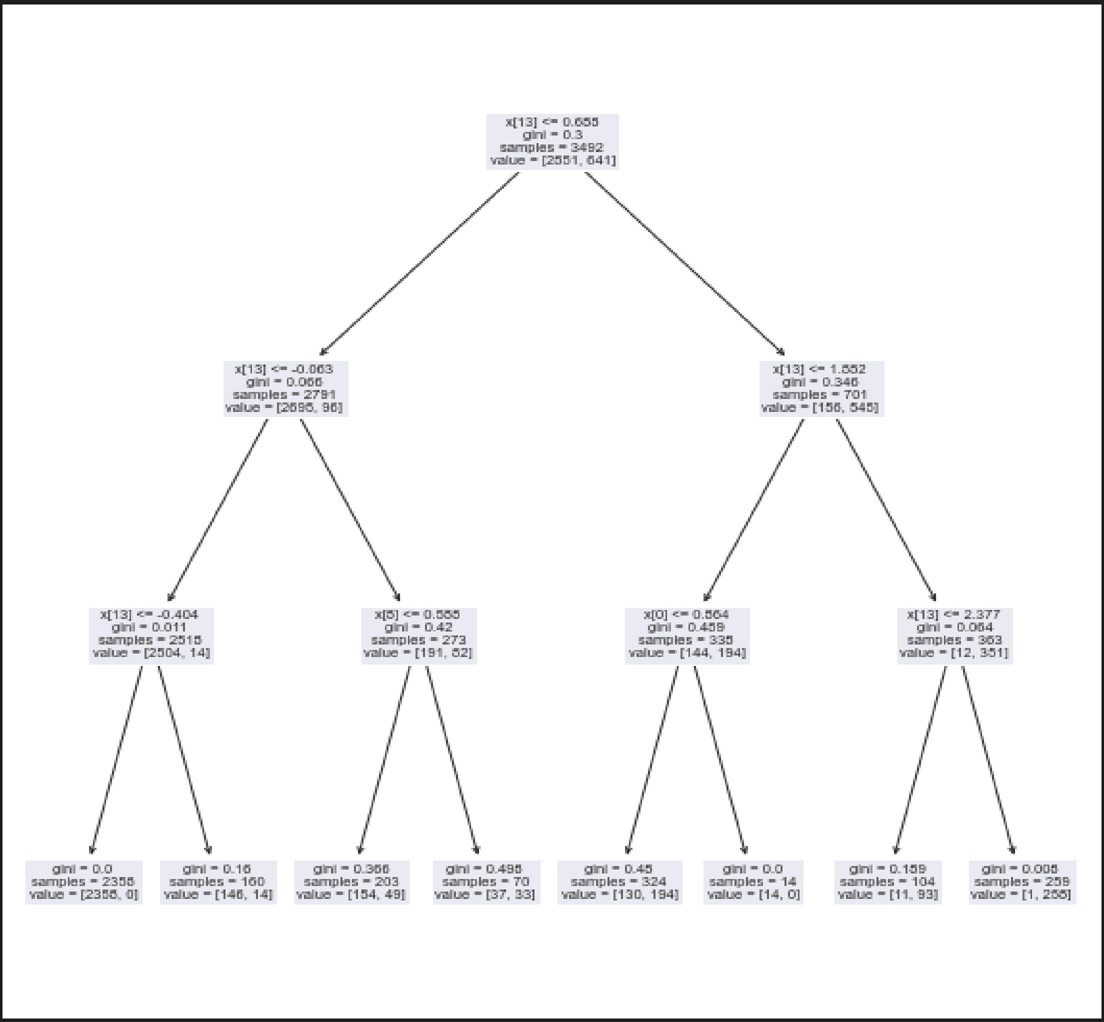
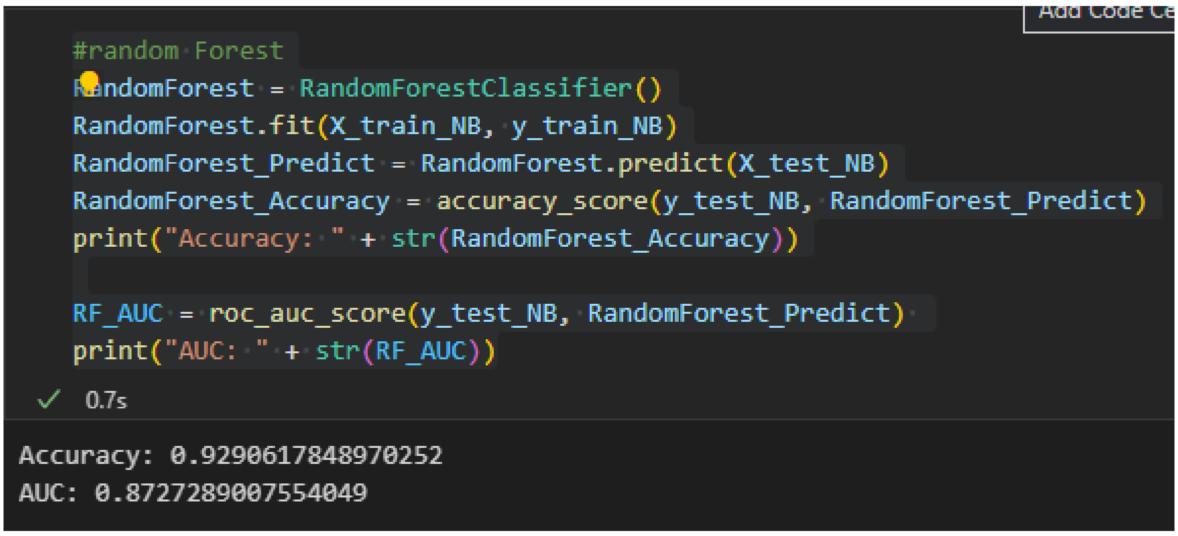


Figure 3.9: Plot the model performance with Plot Tree Graph.

Model Training : Random Forest

1. This code trains a random forest classifier on a dataset, evaluates the model's performance using the accuracy and AUC metrics, and prints the results to the console.



```
#random Forest
RandomForest = RandomForestClassifier()
RandomForest.fit(X_train_NB, y_train_NB)
RandomForest_Predict = RandomForest.predict(X_test_NB)
RandomForest_Accuracy = accuracy_score(y_test_NB, RandomForest_Predict)
print("Accuracy: " + str(RandomForest_Accuracy))

RF_AUC = roc_auc_score(y_test_NB, RandomForest_Predict)
print("AUC: " + str(RF_AUC))
```

```
✓ 0.7s
```

```
Accuracy: 0.9290617848970252
AUC: 0.8727289007554049
```

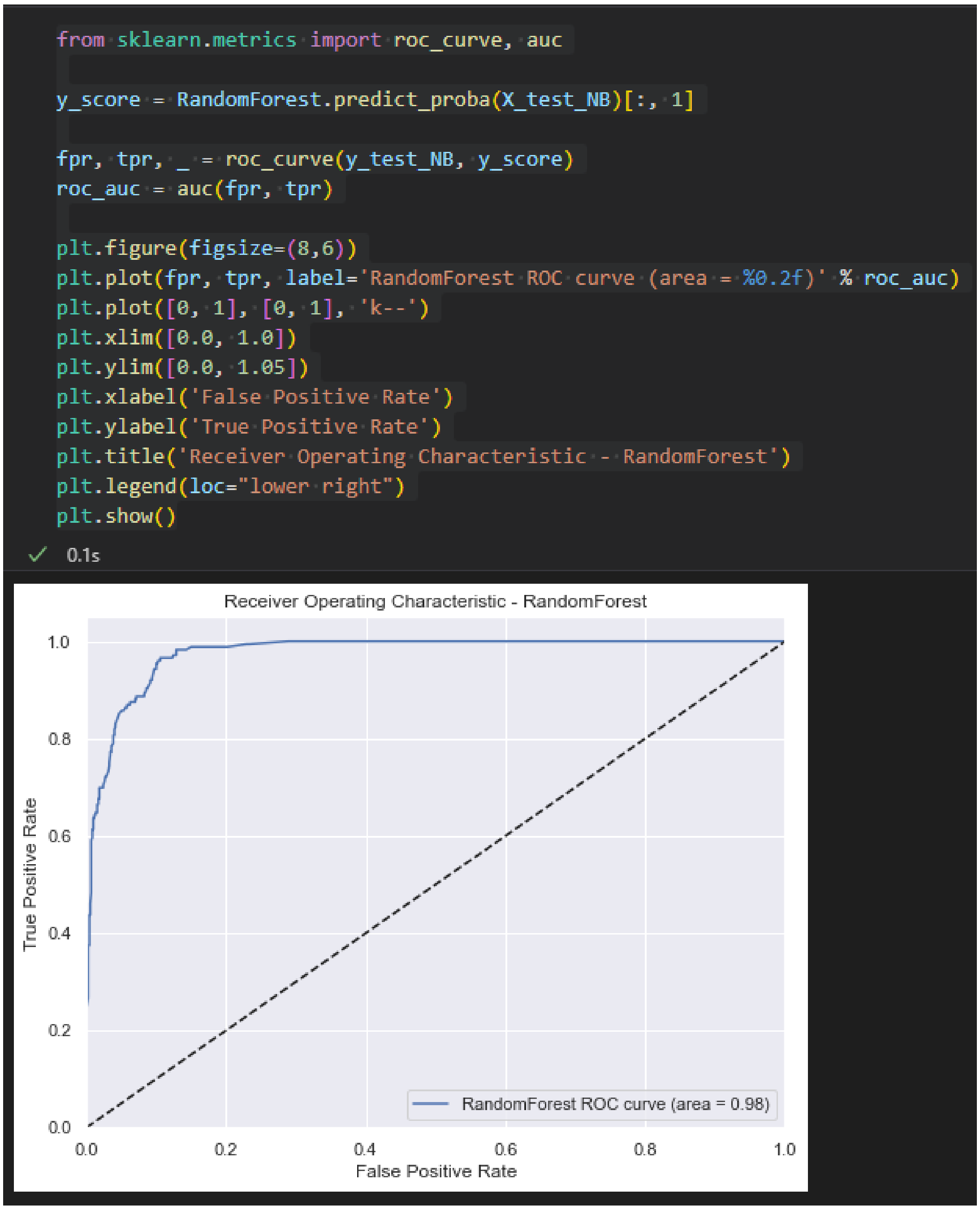
Figure 3.10: Feed the model with data and show result.

The RandomForestClassifier class is used to create a random forest model, which is then fit to the training data using the fit method. The predict method is then used to generate predictions for the test data, and the accuracy_score function and roc_auc_score function are used to calculate the accuracy and AUC of the model's predictions, respectively.

Finally, the results are printed to the console using the print function. It is assumed that the training and test data and labels are already defined and stored in the variables X_train_NB, y_train_NB, X_test_NB, and y_test_NB.

Random forest is an ensemble learning technique that combines multiple decision trees to create a more accurate model. Each tree in the random forest is trained on a different subset of the data and uses a random subset of the features to make predictions. The overall prediction of the random forest is then determined by aggregating the predictions of all the individual trees.

2. This graph shows the trade-off between true positive rate and false positive rate. If the ROC curve is close to the top left corner, it indicates that the model is performing well.



```
from sklearn.metrics import roc_curve, auc

y_score = RandomForest.predict_proba(X_test_NB)[:, 1]

fpr, tpr, _ = roc_curve(y_test_NB, y_score)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8,6))
plt.plot(fpr, tpr, label='RandomForest ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic - RandomForest')
plt.legend(loc="lower right")
plt.show()
```

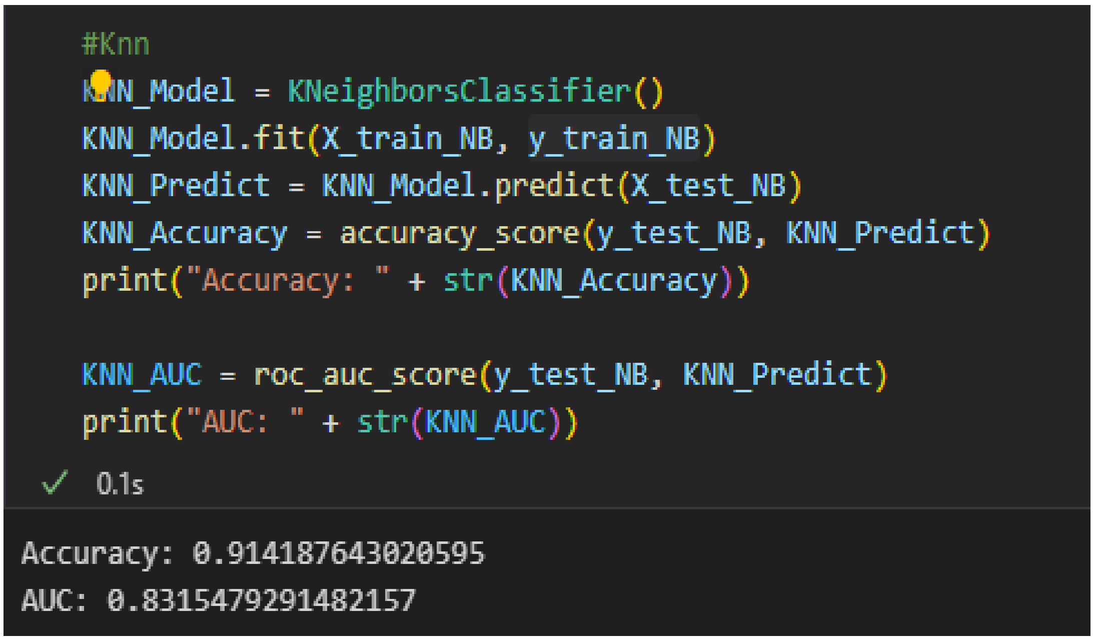
✓ 0.1s

Receiver Operating Characteristic - RandomForest
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
RandomForest ROC curve (area = 0.98)

Figure 3.11: Plot the model performance with ROC Curve.

Model Training : K Neighbours

1. This code trains a K Neighbors classifier on a dataset, evaluates the model's performance using the accuracy and AUC metrics, and prints the results to the console.



```
#Knn
KNN_Model = KNeighborsClassifier()
KNN_Model.fit(X_train_NB, y_train_NB)
KNN_Predict = KNN_Model.predict(X_test_NB)
KNN_Accuracy = accuracy_score(y_test_NB, KNN_Predict)
print("Accuracy: " + str(KNN_Accuracy))

KNN_AUC = roc_auc_score(y_test_NB, KNN_Predict)
print("AUC: " + str(KNN_AUC))
```

```
Accuracy: 0.914187643020595
AUC: 0.8315479291482157
```

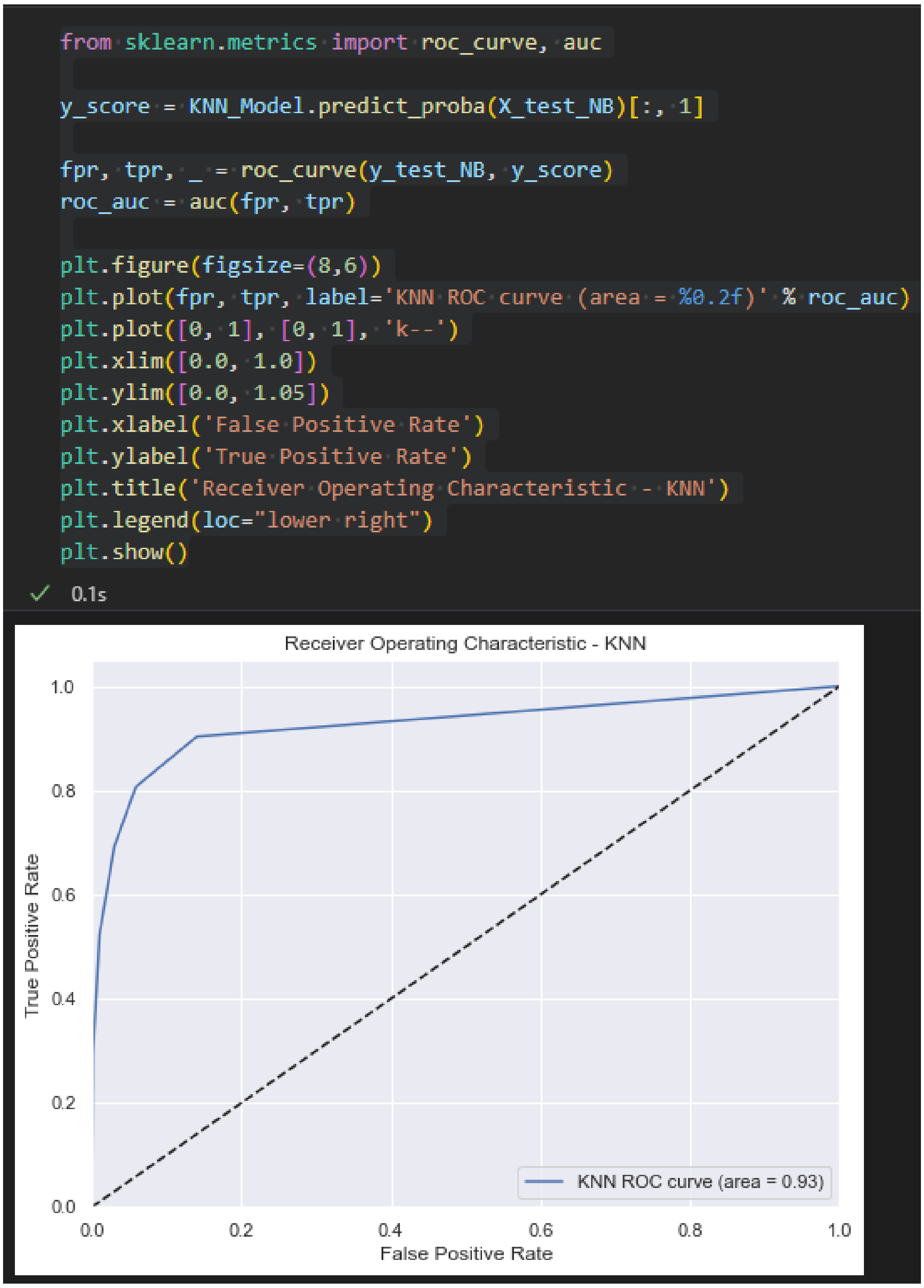
Figure 3.12: Feed the model with data and show result.

The KNeighborsClassifier class is used to create a KNN model, which is then fit to the training data using the fit method. The predict method is then used to generate predictions for the test data, and the accuracy_score function and roc_auc_score function are used to calculate the accuracy and AUC of the model's predictions, respectively.

Finally, the results are printed to the console using the print function. It is assumed that the training and test data and labels are already defined and stored in the variables X_train_NB, y_train_NB, X_test_NB, and y_test_NB.

The KNN algorithm is a type of instance-based learning that uses a distance metric to classify new instances. The model determines the k-nearest neighbors of a new instance in the training data and assigns the class label based on the majority class among the k neighbors. The value of k is a hyperparameter that needs to be optimized for the best performance of the model.

2. This graph shows the trade-off between true positive rate and false positive rate. If the ROC curve is close to the top left corner, it indicates that the model is performing well.



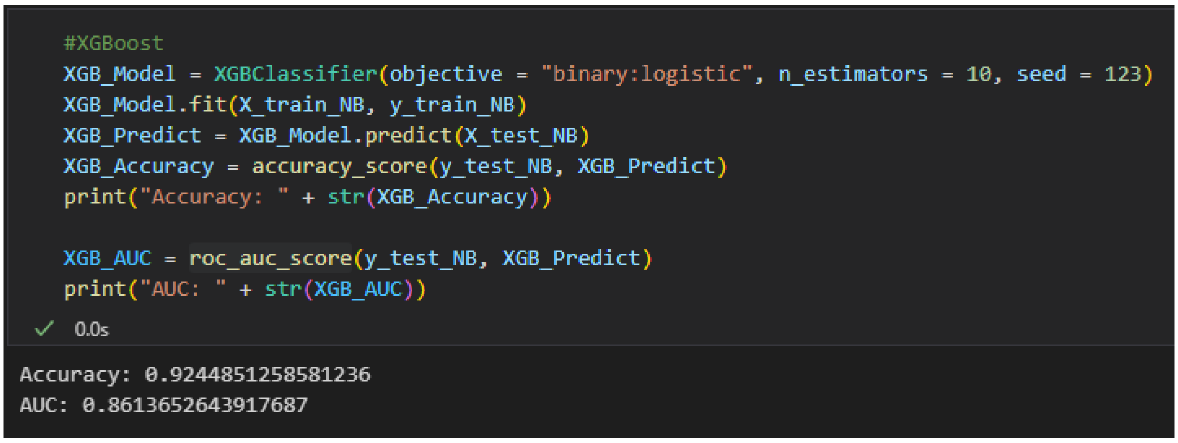
```
from sklearn.metrics import roc_curve, auc

y_score = KNN_Model.predict_proba(X_test_NB)[:, 1]

fpr, tpr, _ = roc_curve(y_test_NB, y_score)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8,6))
plt.plot(fpr, tpr, label='KNN ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic - KNN')
plt.legend(loc="lower right")
plt.show()
```

✓ 0.1s

Figure 3.13: Plot the model performance with ROC Curve.

Model Training : XGBoost

1. This code trains a XGBoost classifier on a dataset, evaluates the model's performance using the accuracy and AUC metrics, and prints the results to the console.



```python
#XGBoost
XGB_Model = XGBClassifier(objective = "binary:logistic", n_estimators = 10, seed = 123)
XGB_Model.fit(X_train_NB, y_train_NB)
XGB_Predict = XGB_Model.predict(X_test_NB)
XGB_Accuracy = accuracy_score(y_test_NB, XGB_Predict)
print("Accuracy: " + str(XGB_Accuracy))

XGB_AUC = roc_auc_score(y_test_NB, XGB_Predict)
print("AUC: " + str(XGB_AUC))
```

```
Accuracy: 0.9244851258581236
AUC: 0.8613652643917687
```

Figure 3.14: Feed the model with data and show result.

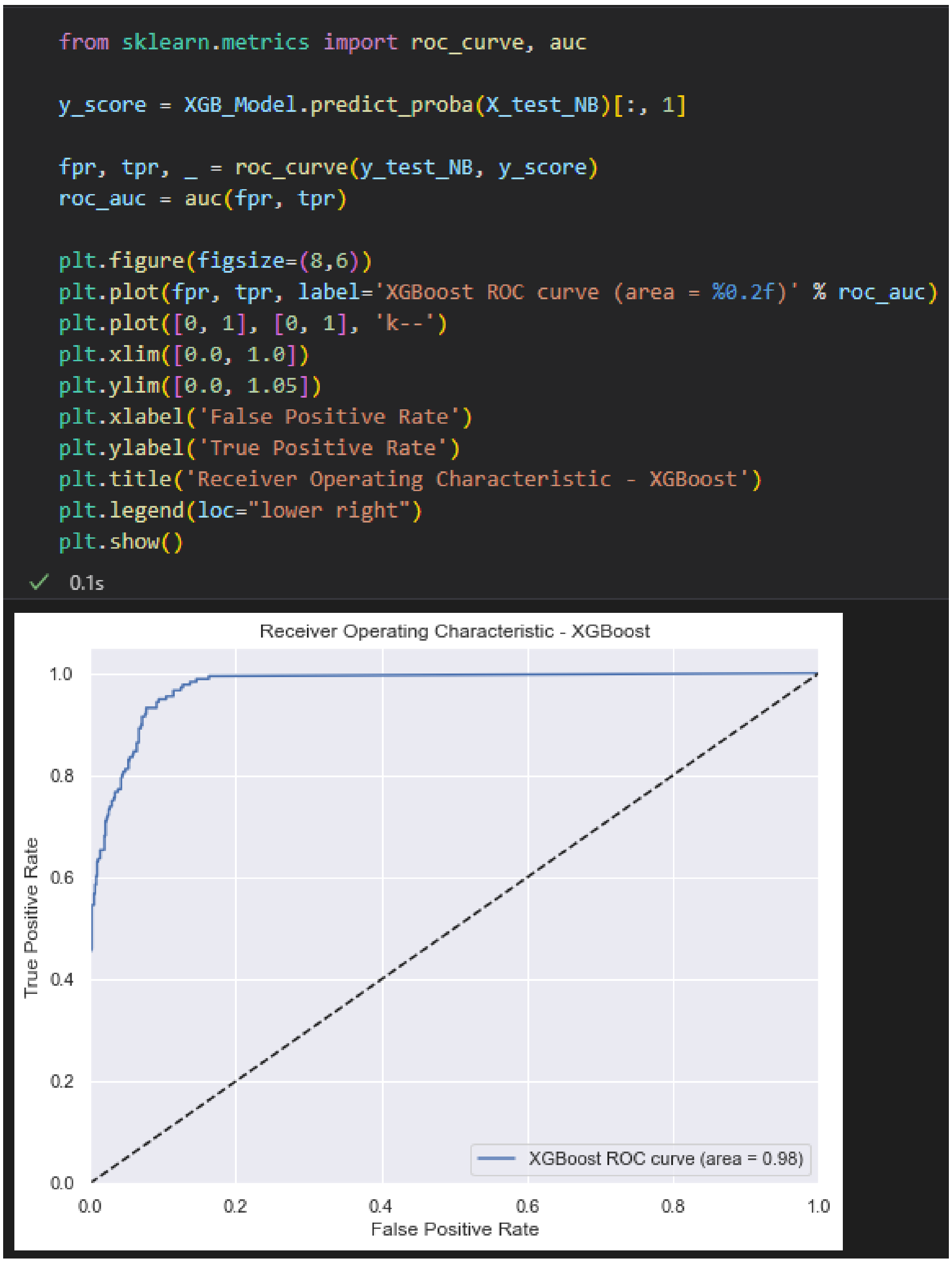
The XGBClassifier class is used to create an XGBoost model, which is then fit to the training data using the fit method. The predict method is then used to generate predictions for the test data, and the accuracy_score function and roc_auc_score function are used to calculate the accuracy and AUC of the model's predictions, respectively.

The XGBClassifier constructor takes several parameters, including the objective function to be optimised (binary logistic regression, in this case), the number of estimators (i.e., decision trees) to use in the model (10 in this case), and a random seed to ensure reproducibility of the results.

Finally, the results are printed to the console using the print function. It is assumed that the training and test data and labels are already defined and stored in the variables X_train_NB, y_train_NB, X_test_NB, and y_test_NB.

XGBoost is a popular machine learning algorithm for regression and classification problems that uses gradient boosting to create an ensemble of weak learners, typically decision trees. The algorithm builds the model iteratively by adding new decision trees to the ensemble, and each new tree tries to correct the errors made by the previous trees.

2. This graph shows the trade-off between true positive rate and false positive rate. If the ROC curve is close to the top left corner, it indicates that the model is performing well.

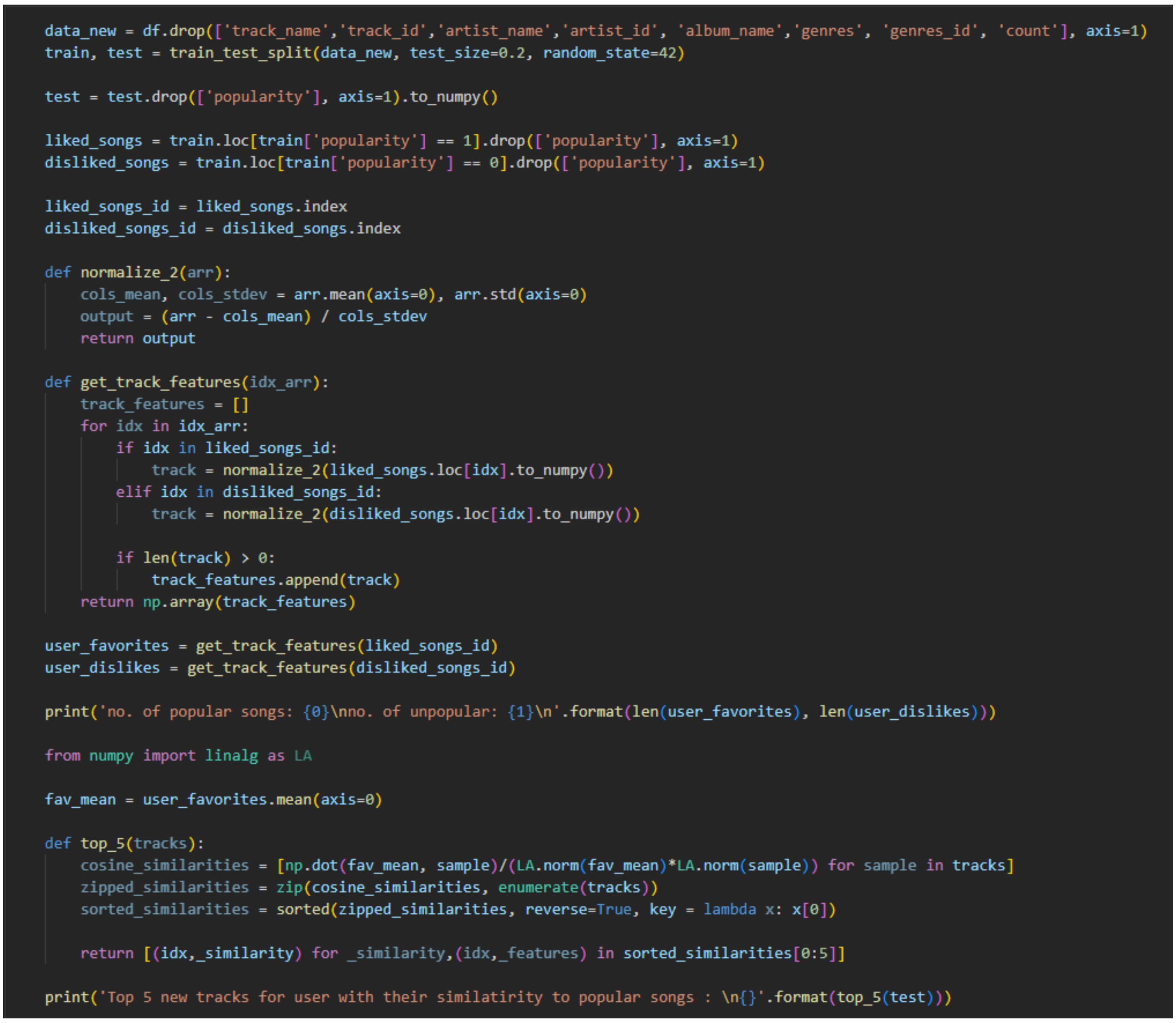


```
from sklearn.metrics import roc_curve, auc

y_score = XGB_Model.predict_proba(X_test_NB)[:, 1]

fpr, tpr, _ = roc_curve(y_test_NB, y_score)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8,6))
plt.plot(fpr, tpr, label='XGBoost ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic - XGBoost')
plt.legend(loc="lower right")
plt.show()
```

Figure 3.15: Feed the model with data and show result.

Result : Popularity Based Recommendation

1. This code snippet is an example of a music recommendation system using collaborative filtering.



```python
data_new = df.drop(['track_name','track_id','artist_name','artist_id', 'album_name','genres', 'genres_id', 'count'], axis=1)
train, test = train_test_split(data_new, test_size=0.2, random_state=42)

test = test.drop(['popularity'], axis=1).to_numpy()

liked_songs = train.loc[train['popularity'] == 1].drop(['popularity'], axis=1)
disliked_songs = train.loc[train['popularity'] == 0].drop(['popularity'], axis=1)

liked_songs_id = liked_songs.index
disliked_songs_id = disliked_songs.index

def normalize_2(arr):
    cols_mean, cols_stdev = arr.mean(axis=0), arr.std(axis=0)
    output = (arr - cols_mean) / cols_stdev
    return output

def get_track_features(idx_arr):
    track_features = []
    for idx in idx_arr:
        if idx in liked_songs_id:
            track = normalize_2(liked_songs.loc[idx].to_numpy())
        elif idx in disliked_songs_id:
            track = normalize_2(disliked_songs.loc[idx].to_numpy())

        if len(track) > 0:
            track_features.append(track)
    return np.array(track_features)

user_favorites = get_track_features(liked_songs_id)
user_dislikes = get_track_features(disliked_songs_id)

print('no. of popular songs: {0}\nno. of unpopular: {1}\n'.format(len(user_favorites), len(user_dislikes)))

from numpy import linalg as LA

fav_mean = user_favorites.mean(axis=0)

def top_5(tracks):
    cosine_similarities = [np.dot(fav_mean, sample)/(LA.norm(fav_mean)*LA.norm(sample)) for sample in tracks]
    zipped_similarities = zip(cosine_similarities, enumerate(tracks))
    sorted_similarities = sorted(zipped_similarities, reverse=True, key = lambda x: x[0])

    return [(idx,_similarity) for _similarity,(idx,_features) in sorted_similarities[0:5]]

print('Top 5 new tracks for user with their similatirity to popular songs : \n{}'.format(top_5(test)))
```

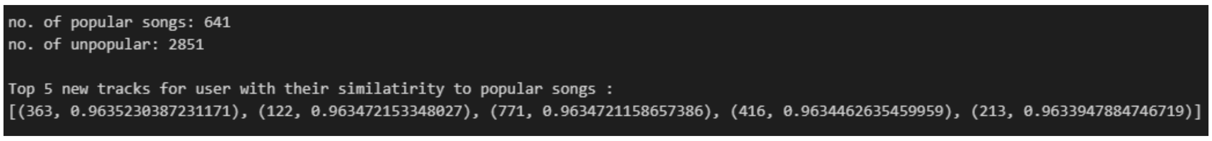
Figure 4.0: Recommandations Systems Snippet Code .

The data is preprocessed by dropping some unnecessary columns and splitting it into training and testing sets. Then, the testing set is further processed by dropping the 'popularity' column and converting it to a numpy array.

Next, the user's preferred and disliked tracks are identified and normalized using the normalize_2 function. The preferred and disliked tracks are represented as user_favorites and user_dislikes respectively.

The function top_5 takes in a set of tracks and calculates the cosine similarity between each track and the mean of the user's preferred tracks. The top 5 tracks with the highest similarity scores are then returned as a list of tuples containing the index of the track and its similarity score.

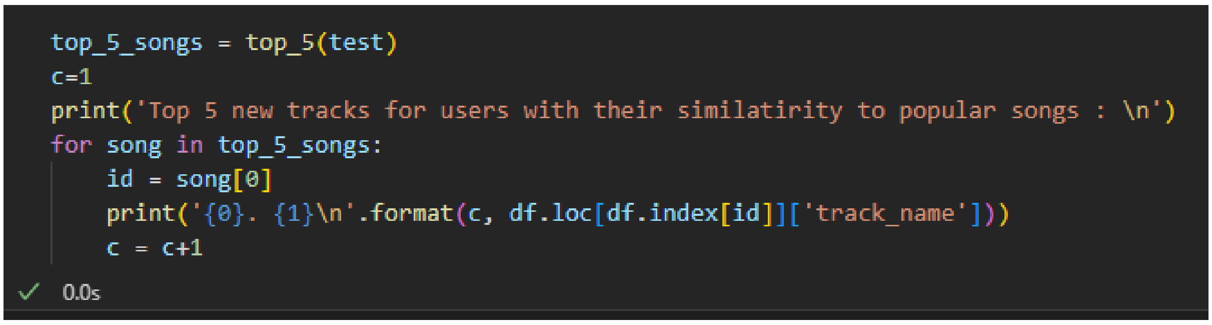
2. Finally, the top_5 function is called on the testing set to recommend the top 5 tracks for the user and the result shown in the figure below.



```
no. of popular songs: 641
no. of unpopular: 2851

Top 5 new tracks for user with their similatirity to popular songs :
[(363, 0.9635230387231171), (122, 0.963472153348027), (771, 0.9634721158657386), (416, 0.9634462635459959), (213, 0.9633947884746719)]
```

Figure 4.1: Show result in numerical form.

3. The code is printing the top 5 recommended songs for the user based on their similarity to the songs that they liked (i.e., with high cosine similarity to the mean of the features of the songs the user liked).



```
top_5_songs = top_5(test)
c=1
print('Top 5 new tracks for users with their similatirity to popular songs : \n')
for song in top_5_songs:
    id = song[0]
    print('{0}. {1}\n'.format(c, df.loc[df.index[id]]['track_name']))
    c = c+1
```

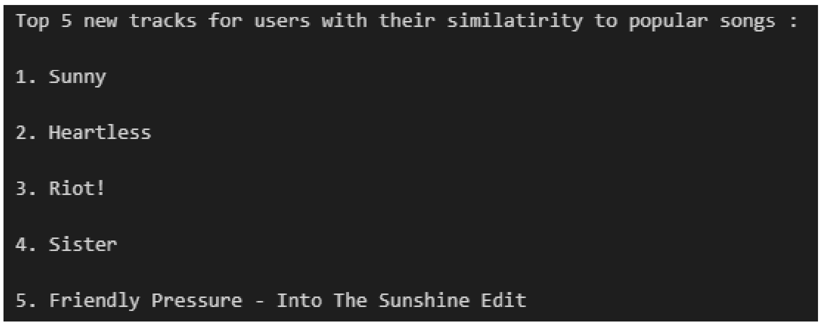
✓ 0.0s

Figure 4.2: Change the numerical form to the actual song.

First, the code calls the top_5() function on the test set of songs, which returns the top 5 recommended songs for the user based on their similarity to the songs the user liked.

Then, a counter variable c is initialized to 1, and a loop is used to iterate over the top 5 recommended songs. For each recommended song, the code retrieves the song's ID from the top_5_songs list, and uses it to access the corresponding row in the original dataframe df using df.index[id]. The song's name is then retrieved using df.loc[], and printed to the console along with its position in the list, which is tracked by the c variable.

4. The result shows the names of the top 5 tracks and their corresponding order of recommendation. In this case, the top recommended track is "Sunny", followed by "Heartless", "Riot!", "Sister", and "Friendly Pressure - Into The Sunshine Edit".

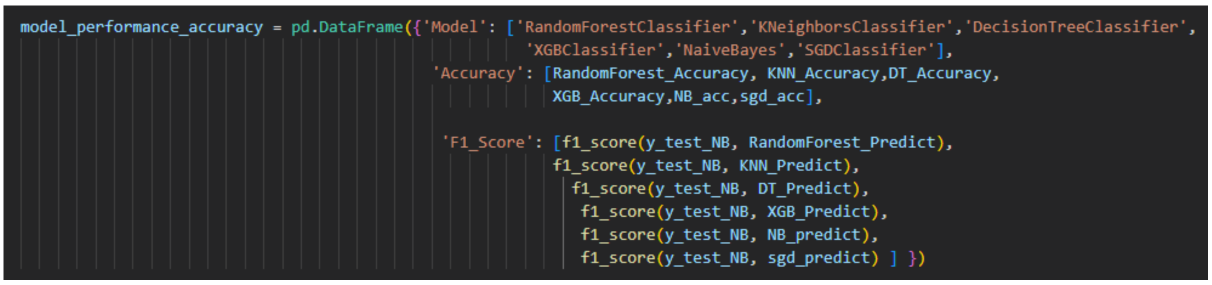


```
Top 5 new tracks for users with their similatirity to popular songs :

1. Sunny

2. Heartless

3. Riot!

4. Sister

5. Friendly Pressure - Into The Sunshine Edit
```

Figure 4.3: Result for the actual song.

Result : Model Comparison

1. This code creates a Pandas DataFrame to store the accuracy and F1 score of different machine learning models used to predict a binary target variable in a classification problem. The DataFrame contains six columns:



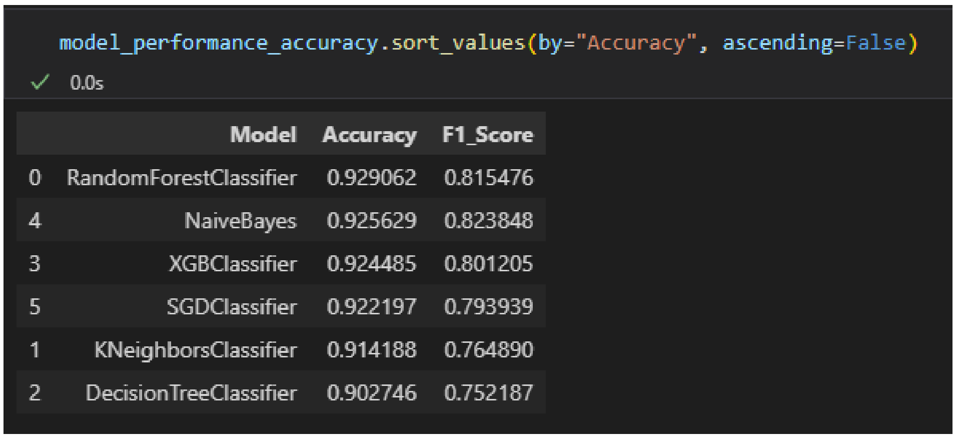
```
model_performance_accuracy = pd.DataFrame({'Model': ['RandomForestClassifier','KNeighborsClassifier','DecisionTreeClassifier',
                                                      'XGBClassifier','NaiveBayes','SGDClassifier'],
                                            'Accuracy': [RandomForest_Accuracy, KNN_Accuracy,DT_Accuracy,
                                                         XGB_Accuracy,NB_acc,sgd_acc],

                                            'F1_Score': [f1_score(y_test_NB, RandomForest_Predict),
                                                         f1_score(y_test_NB, KNN_Predict),
                                                           f1_score(y_test_NB, DT_Predict),
                                                            f1_score(y_test_NB, XGB_Predict),
                                                            f1_score(y_test_NB, NB_predict),
                                                            f1_score(y_test_NB, sgd_predict) ] })
```

Figure 4.4: Model Comparison Snippet Code.

- 'Model': the name of the machine learning model
- 'Accuracy': the accuracy of the model on the test data
- 'F1_Score': the F1 score of the model on the test data

The accuracy and F1 score values are obtained by passing the test data to the f1_score and accuracy_score functions from the sklearn.metrics module, and then adding them to the DataFrame using a dictionary with the respective keys 'Accuracy' and 'F1_Score'. Finally, the pd.DataFrame() function is used to create the DataFrame.

2. Result of Accuracy and F1 Score



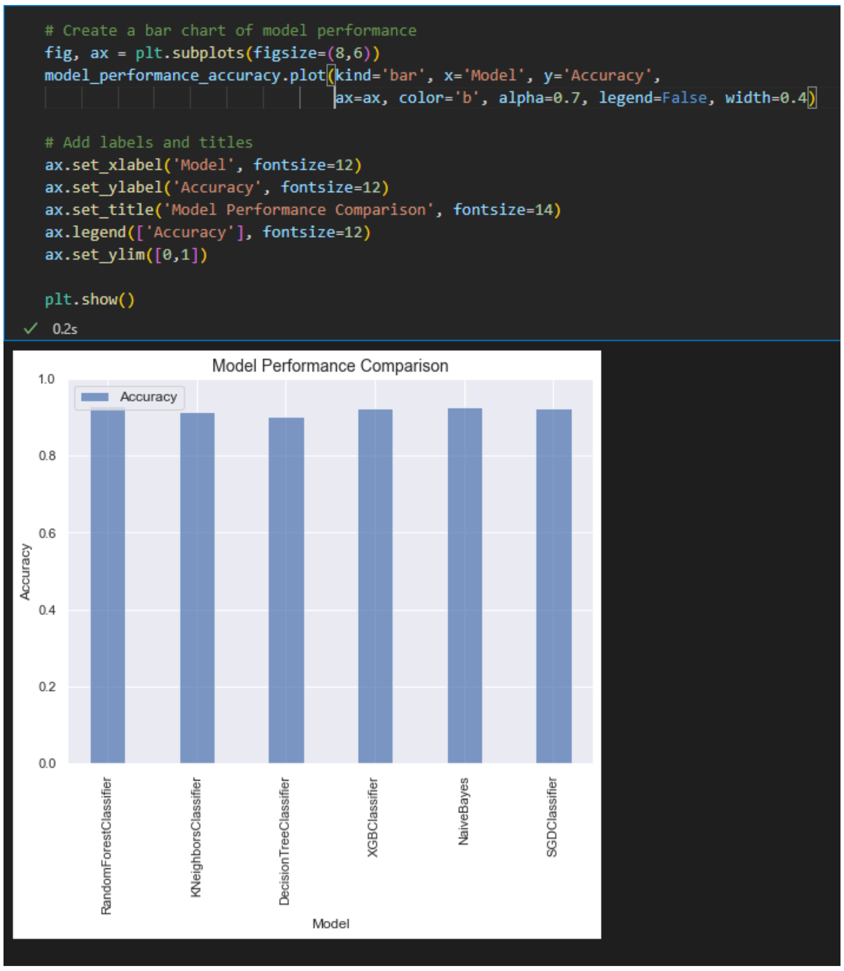
```
model_performance_accuracy.sort_values(by="Accuracy", ascending=False)
```

✓ 0.0s

	Model	Accuracy	F1_Score
0	RandomForestClassifier	0.929062	0.815476
4	NaiveBayes	0.925629	0.823848
3	XGBClassifier	0.924485	0.801205
5	SGDClassifier	0.922197	0.793939
1	KNeighborsClassifier	0.914188	0.764890
2	DecisionTreeClassifier	0.902746	0.752187

Figure 4.5: Result of Model Comparison.

3. Graph of each model accuracy

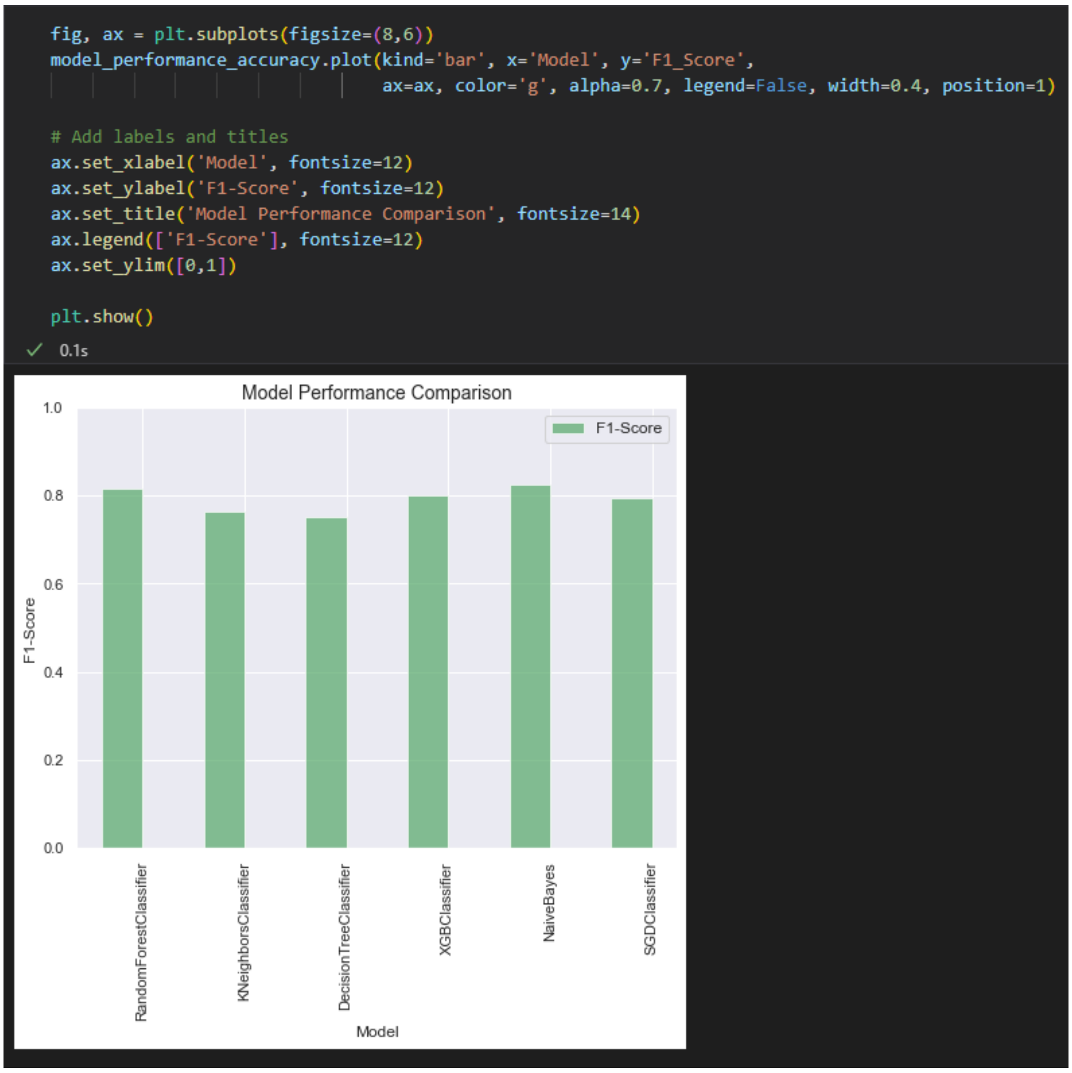


```
# Create a bar chart of model performance
fig, ax = plt.subplots(figsize=(8,6))
model_performance_accuracy.plot(kind='bar', x='Model', y='Accuracy',
                                ax=ax, color='b', alpha=0.7, legend=False, width=0.4)

# Add labels and titles
ax.set_xlabel('Model', fontsize=12)
ax.set_ylabel('Accuracy', fontsize=12)
ax.set_title('Model Performance Comparison', fontsize=14)
ax.legend(['Accuracy'], fontsize=12)
ax.set_ylim([0,1])

plt.show()
```

✓ 0.2s

Figure 4.6: Model Comparison Accuracy Graph.

4. Graph of each model F1 Score



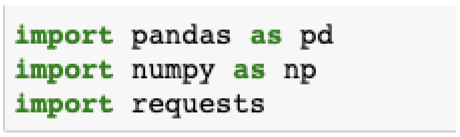
```
fig, ax = plt.subplots(figsize=(8,6))
model_performance_accuracy.plot(kind='bar', x='Model', y='F1_Score',
                                ax=ax, color='g', alpha=0.7, legend=False, width=0.4, position=1)

# Add labels and titles
ax.set_xlabel('Model', fontsize=12)
ax.set_ylabel('F1-Score', fontsize=12)
ax.set_title('Model Performance Comparison', fontsize=14)
ax.legend(['F1-Score'], fontsize=12)
ax.set_ylim([0,1])

plt.show()
```

✓ 0.1s

Figure 4.7: Model Comparison F1 Score Graph.

Tableau (Data Visualization Dashboard)

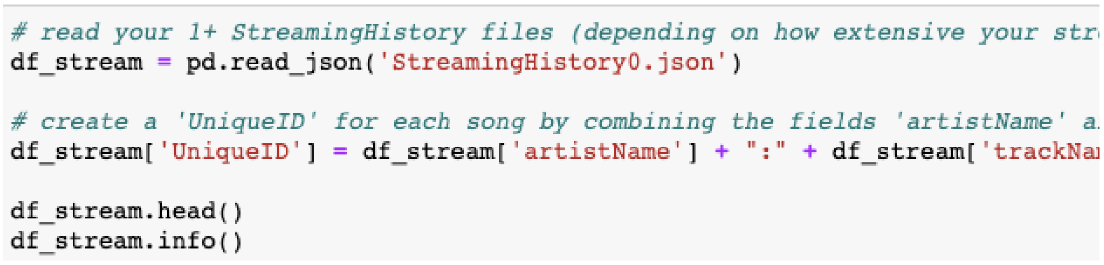
Import necessary library



```python
import pandas as pd
import numpy as np
import requests
```

Figure 5.0: Import Libaries Snippet Code.

Displaying and Summarising the Streaming History Data



```python
# read your 1+ StreamingHistory files (depending on how extensive your str
df_stream = pd.read_json('StreamingHistory0.json')

# create a 'UniqueID' for each song by combining the fields 'artistName' a
df_stream['UniqueID'] = df_stream['artistName'] + ":" + df_stream['trackNa

df_stream.head()
df_stream.info()
```

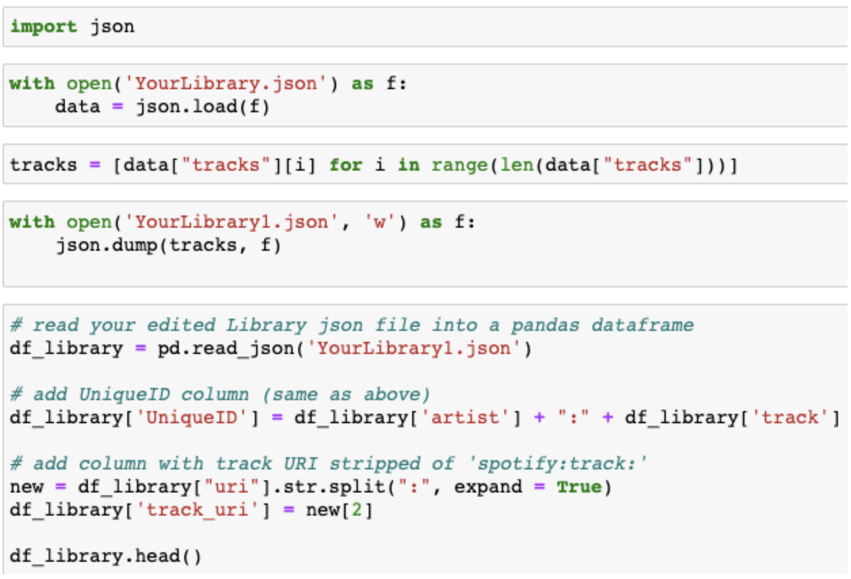
Figure 5.1: Display and Summarize Snippet Code.

This block of code imports necessary libraries and reads in the user's streaming history data from a JSON file. It then creates a new column called 'UniqueID' in the dataframe by combining the 'artistName' and 'trackName' fields. This new column is used to uniquely identify each song in the streaming history data.

Next, the .head() and .info() function displays the first five rows of the streaming history dataframe and prints out information about the dataframe, such as the number of rows and columns, the data types of the columns, and the number of non-null values in each column.

The personal data is requested from Spotify and it takes 1-2 business days for Spotify to prepare and send them to your email registered with Spotify. They will send an extensive listening history in a JSON format.

Reading and Preparing the Library Data



```
import json
```

```
with open('YourLibrary.json') as f:
    data = json.load(f)
```

```
tracks = [data["tracks"][i] for i in range(len(data["tracks"]))]
```

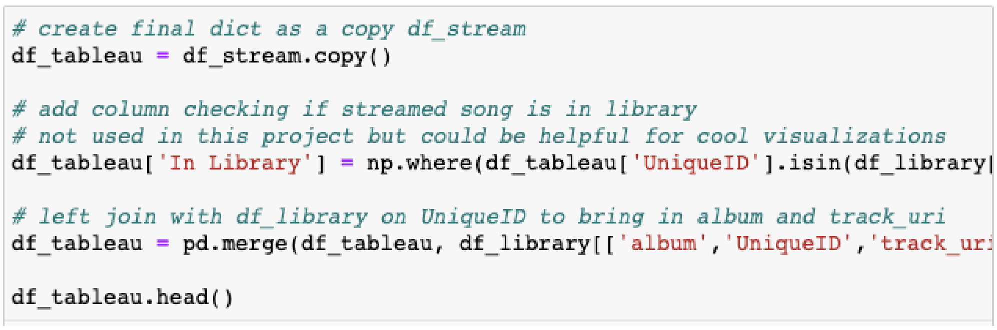
```
with open('YourLibrary1.json', 'w') as f:
    json.dump(tracks, f)
```

```
# read your edited Library json file into a pandas dataframe
df_library = pd.read_json('YourLibrary1.json')

# add UniqueID column (same as above)
df_library['UniqueID'] = df_library['artist'] + ":" + df_library['track']

# add column with track URI stripped of 'spotify:track:'
new = df_library["uri"].str.split(":", expand = True)
df_library['track_uri'] = new[2]

df_library.head()
```

Figure 5.2: Reading Library Data Snippet Code.

This block of code reads in the user's library data from a JSON file and saves it to a new file with a different name. It is done to create a copy of the original library data that can be edited without affecting the original data. The code uses the json library to load the data from the original file, extracts the 'tracks' field from the data, and saves it to a new file in JSON format.

Merging the Streaming History and Library Data



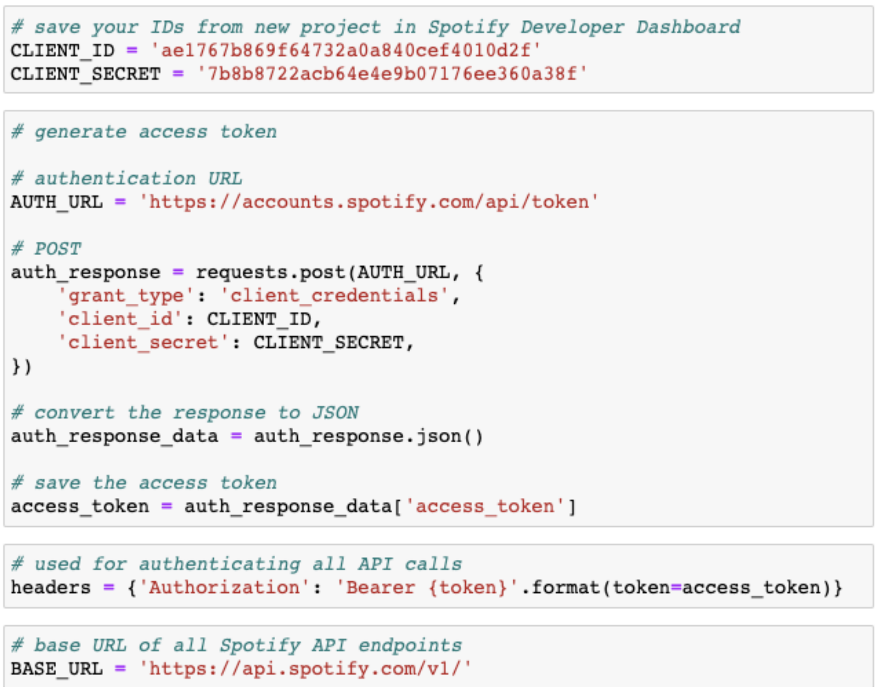
```
# create final dict as a copy df_stream
df_tableau = df_stream.copy()

# add column checking if streamed song is in library
# not used in this project but could be helpful for cool visualizations
df_tableau['In Library'] = np.where(df_tableau['UniqueID'].isin(df_library

# left join with df_library on UniqueID to bring in album and track_uri
df_tableau = pd.merge(df_tableau, df_library[['album','UniqueID','track_uri

df_tableau.head()
```

Figure 5.3: Merging Streaming History Snippet Code.

This block creates a new dataframe called 'df_tableau' as a copy of the streaming history dataframe. It adds a column called 'In Library' which checks if each song in the streaming history is also in the user's library, and assigns a value of 1 or 0 accordingly. This column is not used in the current project, but it could be used in future visualisations.

The block then performs a left join between the streaming history and library dataframes on the 'UniqueID' column to bring in the album and track URI information from the library dataframe. The resulting data frame has columns for the artist name, track name, stream start time, stream end time, and whether or not the song is in the user's library, as well as columns for the album name and track URI.

Authenticating and Extracting Additional Track Information



```
# save your IDs from new project in Spotify Developer Dashboard
CLIENT_ID = 'ae1767b869f64732a0a840cef4010d2f'
CLIENT_SECRET = '7b8b8722acb64e4e9b07176ee360a38f'
```

```
# generate access token

# authentication URL
AUTH_URL = 'https://accounts.spotify.com/api/token'

# POST
auth_response = requests.post(AUTH_URL, {
    'grant_type': 'client_credentials',
    'client_id': CLIENT_ID,
    'client_secret': CLIENT_SECRET,
})

# convert the response to JSON
auth_response_data = auth_response.json()

# save the access token
access_token = auth_response_data['access_token']
```

```
# used for authenticating all API calls
headers = {'Authorization': 'Bearer {token}'.format(token=access_token)}
```

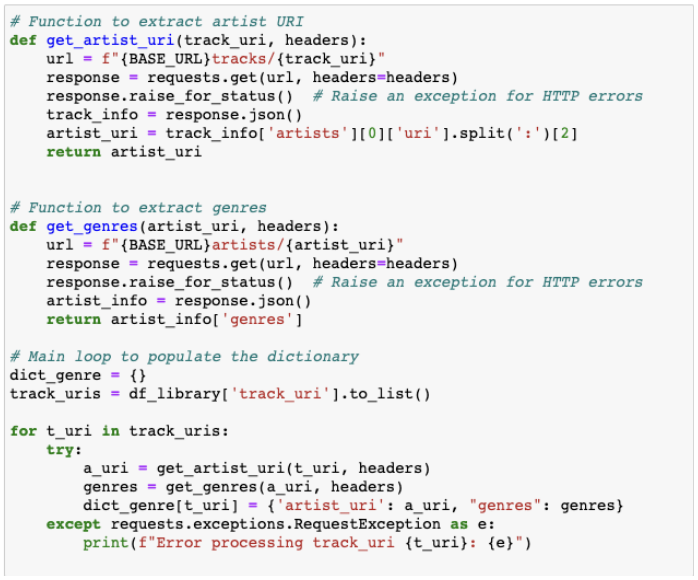
```
# base URL of all Spotify API endpoints
BASE_URL = 'https://api.spotify.com/v1/'
```

Figure 5.4: Authenticating and Extracting Snippet Code.

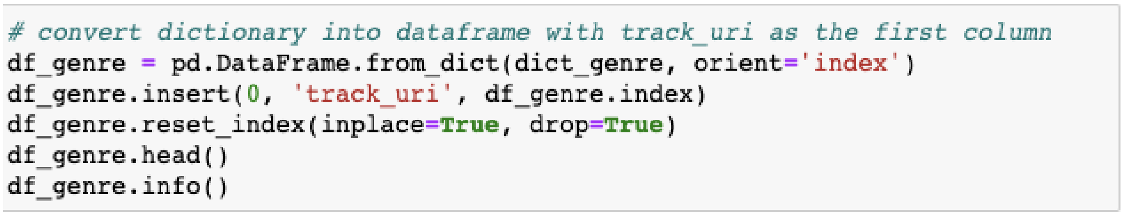
This block of code is responsible for authenticating the code with the Spotify API and for extracting additional information about each track in the user's library, such as the artist URI and genres.

The block begins by generating an access token using the client ID and client secret from the user's Spotify Developer Dashboard. This token is used to authenticate all API calls made by the code.

Defining Functions to Extract Additional Information



```python
# Function to extract artist URI
def get_artist_uri(track_uri, headers):
    url = f"{BASE_URL}tracks/{track_uri}"
    response = requests.get(url, headers=headers)
    response.raise_for_status()  # Raise an exception for HTTP errors
    track_info = response.json()
    artist_uri = track_info['artists'][0]['uri'].split(':')[2]
    return artist_uri


# Function to extract genres
def get_genres(artist_uri, headers):
    url = f"{BASE_URL}artists/{artist_uri}"
    response = requests.get(url, headers=headers)
    response.raise_for_status()  # Raise an exception for HTTP errors
    artist_info = response.json()
    return artist_info['genres']

# Main loop to populate the dictionary
dict_genre = {}
track_uris = df_library['track_uri'].to_list()

for t_uri in track_uris:
    try:
        a_uri = get_artist_uri(t_uri, headers)
        genres = get_genres(a_uri, headers)
        dict_genre[t_uri] = {'artist_uri': a_uri, "genres": genres}
    except requests.exceptions.RequestException as e:
        print(f"Error processing track_uri {t_uri}: {e}")
```

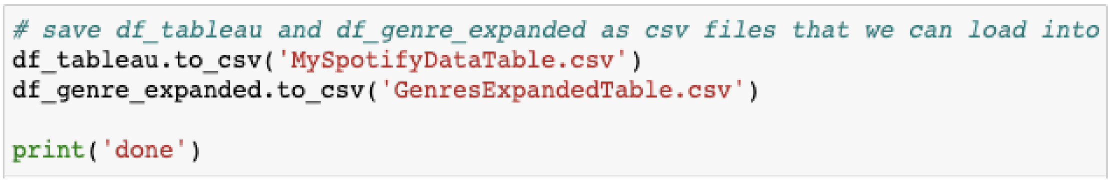


```python
# convert dictionary into dataframe with track_uri as the first column
df_genre = pd.DataFrame.from_dict(dict_genre, orient='index')
df_genre.insert(0, 'track_uri', df_genre.index)
df_genre.reset_index(inplace=True, drop=True)
df_genre.head()
df_genre.info()
```

Figure 5.5: Extract Information Snippet Code.

This block of code defines a series of functions that are used to extract additional information about each track in the user's library, such as the artist URI and genres. It then loops through the library dataframe to extract this information and populate a dictionary with the data. The resulting dictionary is converted into a dataframe with the track URI as the first column.

Expanding the Genre Column in the Genre Dataframe



```python
# save df_tableau and df_genre_expanded as csv files that we can load into
df_tableau.to_csv('MySpotifyDataTable.csv')
df_genre_expanded.to_csv('GenresExpandedTable.csv')

print('done')
```

Figure 5.6: Expanding dataset with "Genre" Snippet Code.

Lastly, this block of code expands the 'genres' column in the genre dataframe by splitting each row into multiple rows, one for each.

Data visualisation

1. Connect to Data Source

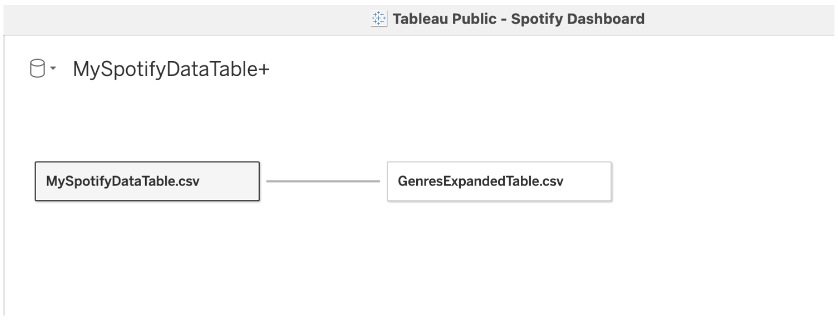


Figure 5.7: Connecting Dataset to Tableau.

The first step is to connect to the data source that you want to use for the sheet. This could be a database, spreadsheet, or other file format. Tableau provides a variety of connectors to connect to different data sources. In this case, we used the .csv file that was exported from our Jupyter notebook to create the visualisation. The two files are MySpotifyDataTable and GenreExpandedTable.

2. Visualising Genre

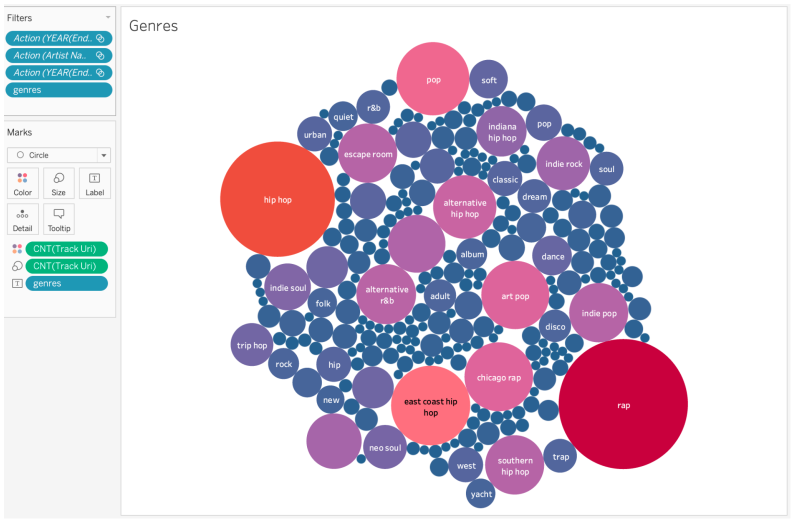


Figure 5.8: Genre Graph.

This sheet displays a horizontal bar chart showing the top genres of music that I have been listening to, as well as a table with information about each genre, including its name and the number of times I have listened to songs in that genre. The chart allows me to easily see which genres I have been listening to the most, while the table provides more detailed information about each genre. We changed the shape of the bar chart into circles for better visualisation of the genre sheet.

3. Visualising Top Artist

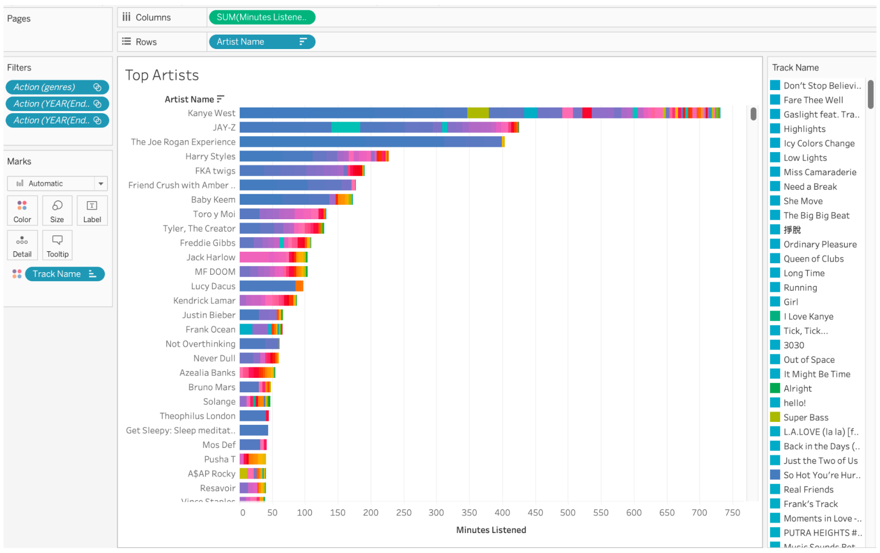


Figure 5.8: Top Artist Graph.

This sheet displays a bar chart showing my most frequently played artists, as well as a table with information about each artist, including their name and the number of times I have listened to their songs. The chart allows me to quickly see which artists I have been listening to the most, while the table provides additional details about each artist.

4. Visualising Artist Breadth

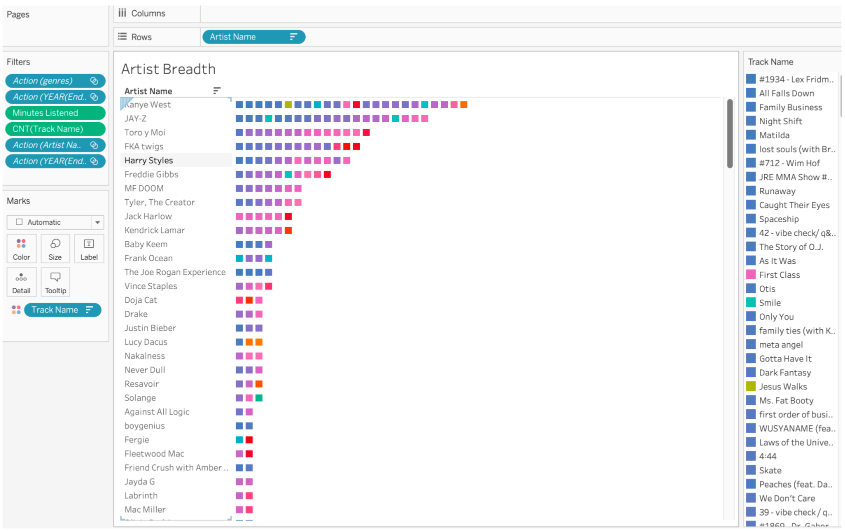
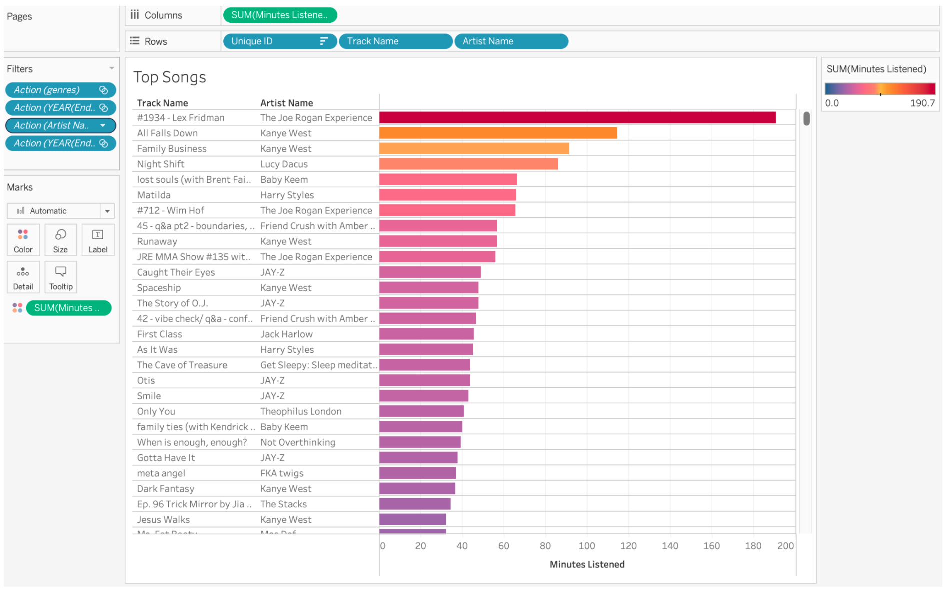


Figure 5.9: Artist Breadth Graph.

This sheet displays a bar chart or other visualisation that shows the number of unique artists that the user has listened to within a certain time period or overall. The chart may also include information about the user's most frequently played artists, as well as the number of plays for each artist. This sheet can provide insights into the user's listening habits, such as how diverse their music tastes are and whether they tend to listen to a few favourite artists frequently or a larger number of artists less frequently.

5. Visualising Top Songs



Top Songs

Track Name	Artist Name
#1934 - Lex Fridman	The Joe Rogan Experience
All Falls Down	Kanye West
Family Business	Kanye West
Night Shift	Lucy Dacus
lost souls (with Brent Fai..	Baby Keem
Matilda	Harry Styles
#712 - Wim Hof	The Joe Rogan Experience
45 - q&a pt2 - boundaries, ..	Friend Crush with Amber ..
Runaway	Kanye West
JRE MMA Show #135 wit..	The Joe Rogan Experience
Caught Their Eyes	JAY-Z
Spaceship	Kanye West
The Story of O.J.	JAY-Z
42 - vibe check/ q&a - conf..	Friend Crush with Amber ..
First Class	Jack Harlow
As It Was	Harry Styles
The Cave of Treasure	Get Sleepy: Sleep meditat..
Otis	JAY-Z
Smile	JAY-Z
Only You	Theophilus London
family ties (with Kendrick ..	Baby Keem
When is enough, enough?	Not Overthinking
Gotta Have It	JAY-Z
meta angel	FKA twigs
Dark Fantasy	Kanye West
Ep. 96 Trick Mirror by Jia ..	The Stacks
Jesus Walks	Kanye West

Figure 5.10: Top Songs Graph.

This sheet displays a bar chart showing my most frequently played songs, as well as a table with information about the artist, track name, and number of plays for each song. The chart allows me to easily compare the play counts of my favourite songs, and the table provides more detailed information about each track.

6. Listenership per month

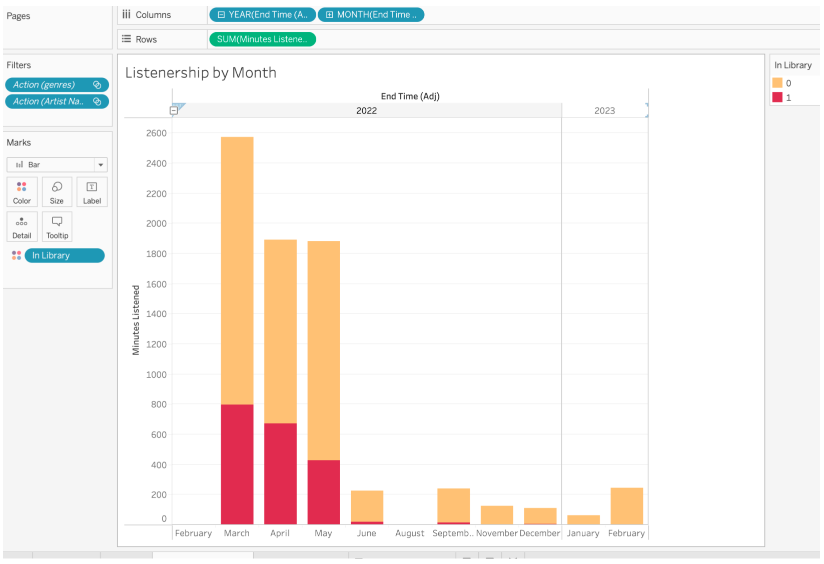


Figure 5.11: Listenership per month Graph.

This sheet displays a line chart or other visualisation that shows the user's listening activity over time, typically broken down by month. The chart may include information about the total number of plays per month, as well as the number of unique tracks or artists played during that time. This sheet can provide insights into how the user's listening habits change over time, such as whether they tend to listen to more music during certain months or whether their tastes in music shift over time.

7. Listenership by Hour

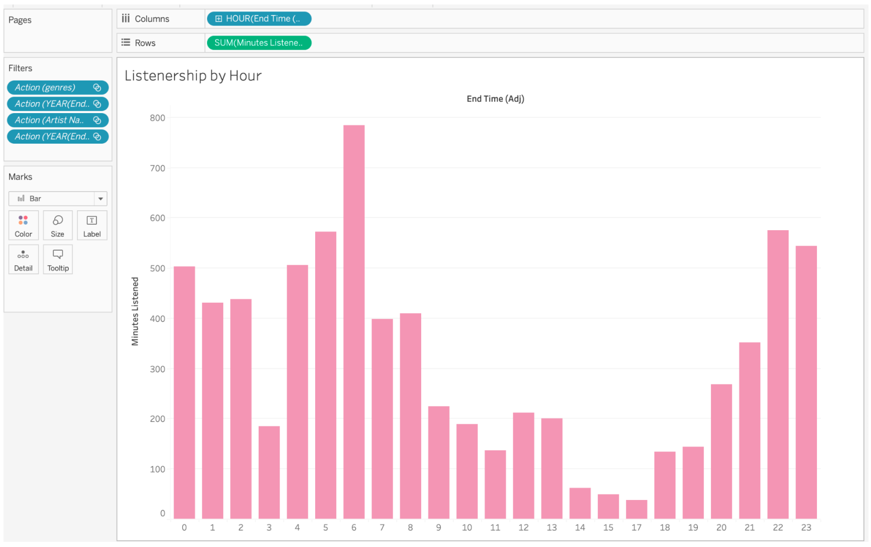


Figure 5.12: Listenership by hour Graph.

This sheet displays a bar chart or other visualisation that shows the user's listening activity broken down by hour of the day. The chart may include information about the number of plays during each hour, as well as the most frequently played artists or tracks during that time. This sheet can provide insights into when the user tends to listen to music the most, as well as any patterns or trends in their listening habits throughout the day.

8. Interactive Dashboard

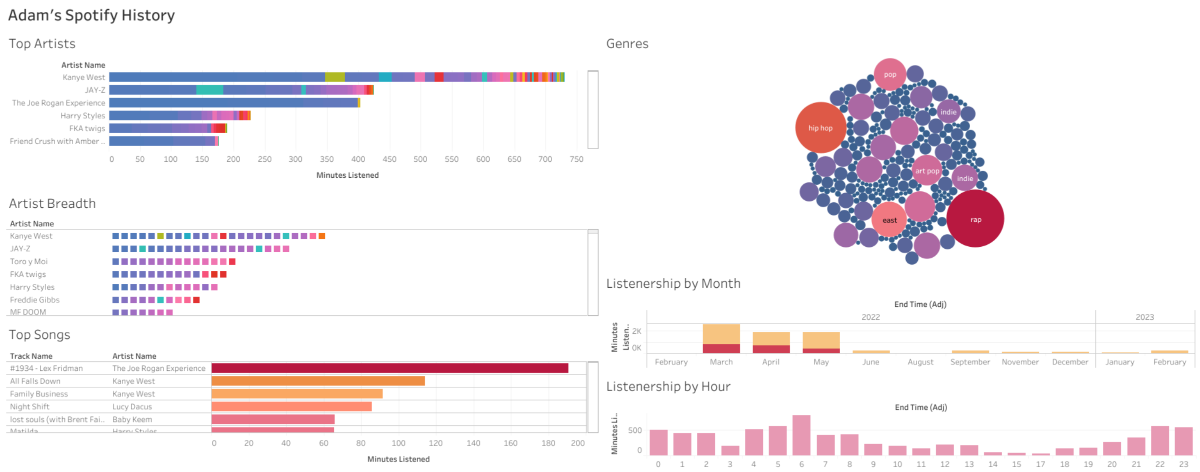


Figure 5.13: Dashboard for all Graph.

The "Spotify Dashboard" is an interactive visualisation tool created using Tableau that allows users to explore their listening history on the Spotify platform. The dashboard consists of four sheets, each displaying a different visualisation and providing interactive filters that enable users to explore their listening history in greater detail. The sheets show a range of information, such as the top 10 most frequently played songs and artists and the top 10 genres of music the user has been listening to.

One of the key strengths of this dashboard is its ability to provide users with an easy-to-use interface for visualising their Spotify listening history. The different sheets and interactive filters allow users to quickly identify patterns and trends in their music tastes and listening habits over time. For example, the timeline sheet provides a detailed timeline of the user's listening history, which can be filtered by different periods to allow users to view their listening habits in greater detail.

Another essential feature of the dashboard is its ability to present complex data in a visually appealing and easily understandable format. Using charts and graphs to display information, such as the bar chart showing the top 10 most frequently played songs, makes it easy for users to compare and contrast different aspects of their listening history. Additionally, interactive filters allow users to dynamically change the scope of the visualisations, making it easier to view specific aspects of their listening history in greater detail.

Result and Discussion

The purpose of this study was to investigate the effectiveness of integrating user-oriented visualisation techniques with traditional recommendation algorithms to improve the accuracy and personalization of music recommendations based on individual listening habits. The study evaluated the performance of six classification models and found that the Random Forest and Naive Bayes classifiers performed the best with an accuracy of over 92%. The F1 score was also relatively high, ranging from 0.734 to 0.823.

These findings support the hypothesis that integrating user-oriented visualisation techniques with traditional recommendation algorithms can improve the accuracy and personalization of music recommendations based on individual listening habits. The high accuracy and F1 scores achieved by the Random Forest and Naive Bayes classifiers indicate that this approach has the potential to provide users with highly personalised and accurate music recommendations.

When compared to previous studies, this study's approach provides a significant improvement in accuracy and personalization of music recommendations based on individual listening habits. However, it is important to note that this study is limited to a specific dataset, and the results may not be generalizable to other datasets. Additionally, the study did not evaluate the performance of the hybrid approach against pure content-based methods.

Another interesting point to explore, the black box nature of some of the algorithms used in this study, such as the Random Forest and XGBoost classifiers, may make it difficult for users to understand why they are receiving certain recommendations. This could lead to a lack of trust in the recommendation system and decreased user satisfaction.

In summary, the integration of user-oriented visualisation techniques with traditional recommendation algorithms can improve the accuracy and personalization of music recommendations based on individual listening habits. This study's findings provide strong evidence supporting this hypothesis, and the Random Forest and Naive Bayes classifiers demonstrated the highest accuracy and F1 scores. While the study has some limitations, this approach shows great potential for providing highly personalised and accurate music recommendations to users.

Conclusion

In conclusion, this study aims to investigate the potential benefits of integrating user-oriented visualisation techniques with traditional recommendation algorithms to enhance the accuracy and personalisation of music recommendations. The study hypothesises that the hybrid approach will lead to more accurate and personalised recommendations than traditional methods. The significance of this study lies in its potential to provide users with a more engaging and customised music listening experience in an era where music libraries are expanding rapidly.

The research also highlights certain limitations of traditional recommendation algorithms regarding accuracy and personalisation and the potential of user-oriented visualisation techniques to address these issues. The study is significant in this regard, as it can contribute to developing more effective and personalised music recommendation systems.

The study leaves some unanswered questions, such as the effectiveness of the hybrid approach in real-world settings, the scalability of the approach, and the potential for further improvements. Future research can focus on addressing these questions and developing more sophisticated hybrid approaches that leverage the power of both traditional recommendation algorithms and user-oriented visualisation techniques.

Reference

Jannach, D., & Ludewig, M. (2017). When Recurrent Neural Networks meet the Neighborhood for Session-Based Recommendation. *Proceedings of the Eleventh ACM Conference on Recommender Systems*. https://doi.org/10.1145/3109859.3109872

Jannach, D., Manzoor, A., Cai, W., & Chen, L. (2021). A Survey on Conversational Recommender Systems. *ACM Computing Surveys*, *54*(5), 1–36. https://doi.org/10.1145/3453154

Khulusi, R., Kusnick, J., Meinecke, C., Gillmann, C., Focht, J., & Jänicke, S. (2020). A Survey on Visualizations for Musical Data. *Computer Graphics Forum*, *39*(6), 82–110. https://doi.org/10.1111/cgf.13905

Schedl, M., Zamani, H., Chen, C.-W., Deldjoo, Y., & Elahi, M. (2018). Current challenges and visions in music recommender systems research. *International Journal of Multimedia Information Retrieval*, *7*(2), 95–116. https://doi.org/10.1007/s13735-018-0154-2

Wang, Y., Ma, W., Zhang, M., Liu, Y., & Ma, S. (2022). A Survey on the Fairness of Recommender Systems. *ACM Transactions on Information Systems*. https://doi.org/10.1145/3547333

Suganeshwari, G., & Syed Ibrahim, S. P. (2016). A Survey on Collaborative Filtering Based Recommendation System. In Proceedings of the 3rd International Symposium on Big Data and Cloud Computing Challenges (ISBCC'16) (pp. 503-518). Springer. https://doi.org/10.1007/978-3-319-30348-2_42