music-recommendation-system-1

March 31, 2024

1 Muhammad Talha

[1]:
```
import os
import sys
from tempfile import NamedTemporaryFile
from urllib.request import urlopen
from urllib.parse import unquote, urlparse
from urllib.error import HTTPError
from zipfile import ZipFile
import tarfile
import shutil

CHUNK_SIZE = 40960
DATA_SOURCE_MAPPING = 'spotify-dataset:https%3A%2F%2Fstorage.googleapis.
 ↪com%2Fkaggle-data-sets%2F1800580%2F2936818%2Fbundle%2Farchive.
 ↪zip%3FX-Goog-Algorithm%3DGOOG4-RSA-SHA256%26X-Goog-Credential%3Dgcp-kaggle-com%2540kaggle-1
 ↪iam.gserviceaccount.
 ↪com%252F20240331%252Fauto%252Fstorage%252Fgoog4_request%26X-Goog-Date%3D20240331T095747Z%26

KAGGLE_INPUT_PATH='/kaggle/input'
KAGGLE_WORKING_PATH='/kaggle/working'
KAGGLE_SYMLINK='kaggle'

!umount /kaggle/input/ 2> /dev/null
shutil.rmtree('/kaggle/input', ignore_errors=True)
os.makedirs(KAGGLE_INPUT_PATH, 0o777, exist_ok=True)
os.makedirs(KAGGLE_WORKING_PATH, 0o777, exist_ok=True)

try:
  os.symlink(KAGGLE_INPUT_PATH, os.path.join("..", 'input'),␣
 ↪target_is_directory=True)
except FileExistsError:
  pass
try:
  os.symlink(KAGGLE_WORKING_PATH, os.path.join("..", 'working'),␣
 ↪target_is_directory=True)
except FileExistsError:
```

```
  pass

for data_source_mapping in DATA_SOURCE_MAPPING.split(','):
    directory, download_url_encoded = data_source_mapping.split(':')
    download_url = unquote(download_url_encoded)
    filename = urlparse(download_url).path
    destination_path = os.path.join(KAGGLE_INPUT_PATH, directory)
    try:
        with urlopen(download_url) as fileres, NamedTemporaryFile() as tfile:
            total_length = fileres.headers['content-length']
            print(f'Downloading {directory}, {total_length} bytes compressed')
            dl = 0
            data = fileres.read(CHUNK_SIZE)
            while len(data) > 0:
                dl += len(data)
                tfile.write(data)
                done = int(50 * dl / int(total_length))
                sys.stdout.write(f"\r[{'=' * done}{' ' * (50-done)}] {dl} bytes␣
 ↪downloaded")
                sys.stdout.flush()
                data = fileres.read(CHUNK_SIZE)
            if filename.endswith('.zip'):
              with ZipFile(tfile) as zfile:
                zfile.extractall(destination_path)
            else:
              with tarfile.open(tfile.name) as tarfile:
                tarfile.extractall(destination_path)
            print(f'\nDownloaded and uncompressed: {directory}')
    except HTTPError as e:
        print(f'Failed to load (likely expired) {download_url} to path␣
 ↪{destination_path}')
        continue
    except OSError as e:
        print(f'Failed to load {download_url} to path {destination_path}')
        continue

print('Data source import complete.')
```

```
Downloading spotify-dataset, 17275602 bytes compressed
[==================================================] 17275602 bytes downloaded
Downloaded and uncompressed: spotify-dataset
Data source import complete.
```

[2]:
```
# This Python 3 environment comes with many helpful analytics libraries␣
 ↪installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/
 ↪docker-python
```

```
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list␣
↪all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that␣
↪gets preserved as output when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be saved␣
↪outside of the current session
```

```
/kaggle/input/spotify-dataset/data/data.csv
/kaggle/input/spotify-dataset/data/data_w_genres.csv
/kaggle/input/spotify-dataset/data/data_by_year.csv
/kaggle/input/spotify-dataset/data/data_by_artist.csv
/kaggle/input/spotify-dataset/data/data_by_genres.csv
```

2 Importing Necessary Libraries

[3]:
```
import numpy as np
import pandas as pd

import seaborn as sns
import plotly.express as px
import plotly.graph_objs as go
import matplotlib.pyplot as plt
from wordcloud import WordCloud

from collections import defaultdict
from scipy.spatial.distance import cdist
from sklearn.preprocessing import MinMaxScaler, StandardScaler

import warnings
warnings.filterwarnings("ignore")
```

3 Importing Datasets

[4]:
```
# Saving data from csv to pandas dataframe
data = pd.read_csv("../input/spotify-dataset/data/data.csv")
genre_data = pd.read_csv('../input/spotify-dataset/data/data_by_genres.csv')
year_data = pd.read_csv('../input/spotify-dataset/data/data_by_year.csv')
artist_data = pd.read_csv('../input/spotify-dataset/data/data_by_artist.csv')
```

4 Understanding the Datasets

[5]:
```
data.sample(5)
```

[5]:
```
        valence  year  acousticness  \
144489    0.251  1963        0.8460
3886      0.561  1941        0.8940
76634     0.091  1939        0.9950
134281    0.551  1987        0.9950
86304     0.799  1990        0.0128

                                                  artists  danceability  \
144489                                  ['Living Strings']         0.249
3886                                       ["Anita O'Day"]         0.599
76634                                        ['Duo A & G']         0.292
134281  ['Benny Goodman Trio', 'Benny Goodman', 'Teddy…        0.676
86304                                           ['Sybil']         0.676

        duration_ms  energy  explicit                      id  \
144489       192147  0.2350         0  6S4mWt72qIda3FEX54q0uL
3886         190867  0.2870         0  2YxJFkhOj3lsfbuny4QFef
76634        183853  0.1160         0  6WWjQ7QmMXDfuz8f3y64Rw
134281       207400  0.0905         0  44A3gCwhADwREEqnG2Yj5o
86304        255467  0.5490         0  7C2PSApP1jxFFpPgggCAaG

        instrumentalness  key  liveness  loudness  mode  \
144489          0.842000    1     0.199   -16.379     1
3886            0.000028   10     0.298    -8.834     1
76634           0.378000    3     0.202   -17.709     1
134281          0.896000   10     0.122   -21.941     0
86304           0.000004    5     0.096   -12.357     1

                                          name  popularity release_date  \
144489                         White Christmas          12   1963-10-01
3886                                 Slow Down           6         1941
76634                        Over the rainbow            0         1939
134281  Body and Soul - 1996 Remastered - Take 2        32   1987-09-01
86304                          Make It Easy On Me       48         1990
```

```
        speechiness    tempo
144489       0.0299   90.331
3886         0.0312  102.692
76634        0.0397   74.678
134281       0.0387   95.975
86304        0.0585  100.069
```

This dataset appears to contain information about various musical tracks, with each row representing a single track and its attributes.

1. **valence**: A measure of the musical positiveness conveyed by a track, ranging from 0 to 1.
2. **year**: The year when the track was released.
3. **acousticness**: A measure of the acoustic characteristics of the track, ranging from 0 to 1.
4. **artists**: Names of the artists or group associated with the track.
5. **danceability**: A measure of how suitable a track is for dancing, ranging from 0 to 1.
6. **duration_ms**: The duration of the track in milliseconds.
7. **energy**: Represents the energy level of the track, ranging from 0 to 1.
8. **explicit**: Indicates whether the track contains explicit content (1 for explicit, 0 for not explicit).
9. **id**: A unique identifier for the track.
10. **instrumentalness**: Indicates the likelihood of the track containing no vocals, ranging from 0 to 1.
11. **key**: The key the track is in.
12. **liveness**: A measure of the presence of a live audience in the track, ranging from 0 to 1.
13. **loudness**: The overall loudness of the track in decibels (dB).
14. **mode**: Indicates the modality of the track (major or minor, 1 for major, 0 for minor).
15. **name**: The name of the track.
16. **popularity**: A measure of the popularity of the track, ranging from 0 to 100.
17. **release_date**: The date when the track was released.
18. **speechiness**: Detects the presence of spoken words in the track, ranging from 0 to 1.
19. **tempo**: The tempo of the track in beats per minute (BPM).

The dataset seems to encompass a variety of music genres, spanning different years of release, with each track having distinct attributes related to its musical characteristics and metadata.

[6]:
```
genre_data.sample(5)
```

[6]:
```
      mode                 genres  acousticness  danceability  \
1338     1         icelandic indie      0.733876      0.514616
1051     1      folclor colombiano      0.267626      0.769304
1321     0                  humppa      0.673000      0.775000
1117     1           funk mexicano      0.103563      0.449667
2237     0  progressive power metal      0.000031      0.223000

        duration_ms    energy  instrumentalness  liveness   loudness  \
1338  239498.184211  0.314205          0.005528  0.120432 -11.336605
1051  227693.739130  0.592391          0.004813  0.092065  -7.851609
```

```
1321  151480.000000  0.487000          0.000000  0.203000 -11.048000
1117  235676.666667  0.663500          0.090306  0.103967  -6.007667
2237  716347.000000  0.885000          0.001050  0.243000  -3.687000

      speechiness       tempo   valence  popularity  key
1338     0.044009  102.885563  0.261375   54.778947    7
1051     0.106878  109.587609  0.925130   43.086957    0
1321     0.122000  127.385000  0.921000   38.000000    2
1117     0.045733  123.825833  0.437833   56.833333    3
2237     0.057000  101.795000  0.143000   44.000000    9
```

[7]:
```python
year_data.sample(5)
```

[7]:
```
    mode  year  acousticness  danceability    duration_ms    energy  \
21     1  1942      0.852934      0.464634  222361.168252  0.256079
87     1  2008      0.249192      0.579193  240107.315601  0.671461
3      1  1924      0.940200      0.549894  191046.707627  0.344347
95     1  2016      0.284171      0.600202  221396.510295  0.592855
23     1  1944      0.907653      0.500174  245555.586436  0.253441

    instrumentalness  liveness   loudness  speechiness       tempo   valence  \
21          0.392882  0.212878 -15.029032     0.083678  106.008398  0.477409
87          0.063662  0.198431  -6.843804     0.077356  123.509934  0.527542
3           0.581701  0.235219 -14.231343     0.092089  120.689572  0.663725
95          0.093984  0.181170  -8.061056     0.104313  118.652630  0.431532
23          0.449292  0.238772 -14.582056     0.173283  105.963930  0.540695

    popularity  key
21    1.126635    7
87   50.630179    0
3     0.661017   10
95   59.647190    0
23    3.192819   10
```

This dataset appears to contain information about various music tracks, with each row representing a different track.

1. **Mode**: This column indicates a categorical variable representing some mode or category associated with the music tracks.
2. **Genres**: This column specifies the genre or style of music.
3. **Acousticness**: A numeric value representing the level of acousticness in the track, ranging from 0 to 1, where 0 indicates not acoustic at all and 1 indicates highly acoustic.
4. **Danceability**: A numeric value representing how suitable a track is for dancing, ranging from 0 to 1, where 0 indicates not danceable and 1 indicates highly danceable.
5. **Duration_ms**: The duration of the track in milliseconds.
6. **Energy**: A numeric value representing the energy of the track, likely in the musical sense, ranging from 0 to 1.
7. **Instrumentalness**: A numeric value representing the likelihood that the track contains no

vocals, ranging from 0 to 1.
8. **Liveness**: A numeric value representing the probability that the track was performed live, ranging from 0 to 1.
9. **Loudness**: The overall loudness of the track in decibels (dB).
10. **Speechiness**: A numeric value representing the presence of spoken words in the track, ranging from 0 to 1.
11. **Tempo**: The tempo of the track in beats per minute (BPM).
12. **Valence**: A numeric value representing the musical positiveness conveyed by a track, ranging from 0 to 1, where 0 indicates negative and 1 indicates positive.
13. **Popularity**: A numeric value representing the popularity of the track.
14. **Key**: A categorical variable representing the key of the track.

Each row provides data for a specific track, including its characteristics such as genre, tempo, energy, and popularity, among others. The dataset seems to cover a variety of music genres and styles, along with their corresponding attributes.

[8]:
```
artist_data.sample(5)
```

[8]:
```
       mode  count  acousticness         artists  danceability  duration_ms  \
19482     1      2       0.00004      Pestilence        0.3060     279093.0
3541      1      2       0.98800     Buell Kazee        0.5700     191187.0
21656     0     10       0.34507        SG Lewis        0.6316     240617.7
21489     1      4       0.32400      Rudy Mills        0.6925     190595.0
18211     1      4       0.99300  Nikos Perdikis        0.4085     194253.0

       energy  instrumentalness  liveness  loudness  speechiness     tempo  \
19482  0.9890          0.000410   0.27000   -5.0870      0.10900  103.0060
3541   0.3590          0.000168   0.41600  -11.9340      0.03960  137.6800
21656  0.5142          0.069837   0.22416   -9.6500      0.17007  114.1151
21489  0.6855          0.134840   0.17895   -7.2445      0.06775   81.7680
18211  0.5170          0.036465   0.40300   -9.1530      0.04115  102.6185

       valence  popularity  key
19482   0.1340        38.0    2
3541    0.8780        13.0    3
21656   0.4877        59.3   11
21489   0.8875        27.5   10
18211   0.6590         0.0    5
```

This dataset appears to contain information about different musical tracks.

1. **Mode**: Indicates the modality of the track (0 = Minor, 1 = Major).
2. **Count**: Represents the count of something related to the track, but the meaning is not explicitly clear from the provided snippet.
3. **Acousticness**: A measure of the acoustic characteristics of the track, where 0 represents low acousticness and 1 represents high acousticness.
4. **Artists**: The name of the artist or artists associated with the track.
5. **Danceability**: Represents how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity.

6. **Duration_ms**: The duration of the track in milliseconds.
7. **Energy**: Represents the intensity and activity of the track. Typically, energetic tracks feel fast, loud, and noisy.
8. **Instrumentalness**: Indicates the likelihood of the track containing no vocals. A high value suggests the track is instrumental.
9. **Liveness**: Represents the presence of a live audience in the recording. A value closer to 1 suggests higher liveness.
10. **Loudness**: The overall loudness of the track in decibels (dB).
11. **Speechiness**: Detects the presence of spoken words in a track. The more exclusively speech-like the recording, the closer to 1.0 the attribute value.
12. **Tempo**: The overall estimated tempo of the track in beats per minute (BPM).
13. **Valence**: Describes the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g., happy, cheerful), while tracks with low valence sound more negative (e.g., sad, depressed).
14. **Popularity**: Represents the popularity of the track, likely based on metrics such as play counts or listener engagement.
15. **Key**: The key of the track.

Each row in the dataset seems to represent a different track, with each column providing specific information about that track.

[9]:
```
# Combining the datasets for convinent access
datasets = [("data", data), ("genre_data", genre_data), ("year_data",␣
↪year_data), ("artist_data", artist_data)]
```

[11]:
```
# Typecasting columns for better understanding of the datasets
data['year'] = pd.to_datetime(data['year'], format='%Y')
data['release_date'] = pd.to_datetime(data['release_date'])
year_data['year'] = pd.to_datetime(year_data['year'], format='%Y')
```

[ ]:
```
for name, df in datasets:
    # print some info about the datasets
    print(f"Info about the dataset: {name}")
    print("-"*30)
    print(df.info())
    print()
```

4.1 Checking for Missing Values

[12]:
```
for name, df in datasets:
    print(f"Missing Values in: {name}")
    print("-"*30)
    print(df.isnull().sum())
    print()
```

```
Missing Values in: data
------------------------------
valence             0
```

```
year                0
acousticness        0
artists             0
danceability        0
duration_ms         0
energy              0
explicit            0
id                  0
instrumentalness    0
key                 0
liveness            0
loudness            0
mode                0
name                0
popularity          0
release_date        0
speechiness         0
tempo               0
dtype: int64

Missing Values in: genre_data
------------------------------
mode                0
genres              0
acousticness        0
danceability        0
duration_ms         0
energy              0
instrumentalness    0
liveness            0
loudness            0
speechiness         0
tempo               0
valence             0
popularity          0
key                 0
dtype: int64

Missing Values in: year_data
------------------------------
mode                0
year                0
acousticness        0
danceability        0
duration_ms         0
energy              0
instrumentalness    0
liveness            0
```

```
loudness            0
speechiness         0
tempo               0
valence             0
popularity          0
key                 0
dtype: int64

Missing Values in: artist_data
------------------------------
mode                0
count               0
acousticness        0
artists             0
danceability        0
duration_ms         0
energy              0
instrumentalness    0
liveness            0
loudness            0
speechiness         0
tempo               0
valence             0
popularity          0
key                 0
dtype: int64
```

4.2 Checking for Duplicate Values

[13]:
```
for name, df in datasets:
    print(f"Duplicates in the dataset: {name}")
    print("-"*30)
    print(df.duplicated(keep=False).sum())
    print()
```

```
Duplicates in the dataset: data
------------------------------
0

Duplicates in the dataset: genre_data
------------------------------
0

Duplicates in the dataset: year_data
------------------------------
0
```

```
Duplicates in the dataset: artist_data
------------------------------
0
```

4.3 Checking the Unique Values

[14]:
```
for name, df in datasets:
    print(f"Unique Values in: {name}")
    print("-"*30)
    print(df.nunique())
    print()
```

```
Unique Values in: data
------------------------------
valence                   1733
year                       100
acousticness              4689
artists                  34088
danceability              1240
duration_ms              51755
energy                    2332
explicit                     2
id                      170653
instrumentalness          5401
key                         12
liveness                  1740
loudness                 25410
mode                         2
name                    133638
popularity                 100
release_date             10968
speechiness               1626
tempo                    84694
dtype: int64

Unique Values in: genre_data
------------------------------
mode                    2
genres               2973
acousticness         2798
danceability         2725
duration_ms          2872
energy               2778
instrumentalness     2731
liveness             2709
loudness             2873
speechiness          2707
```

```
tempo               2872
valence             2745
popularity          2188
key                   12
dtype: int64

Unique Values in: year_data
------------------------------
mode                  1
year                100
acousticness        100
danceability        100
duration_ms         100
energy              100
instrumentalness    100
liveness            100
loudness            100
speechiness         100
tempo               100
valence             100
popularity          100
key                   7
dtype: int64

Unique Values in: artist_data
------------------------------
mode                    2
count                 379
acousticness        14127
artists             28680
danceability        10650
duration_ms         23960
energy              12126
instrumentalness    15517
liveness            12156
loudness            21862
speechiness         10950
tempo               24801
valence             11882
popularity           4663
key                    12
dtype: int64
```

5 Data Visualization

5.1 Popularity Treads Over the Years

[15]:
```
# Popularity Trends Over Years
fig = px.line(year_data, x='year', y='popularity', title='Popularity Trends␣
↪Over the Years', labels={'year': 'Years --->', 'popularity': "Popularity␣
↪--->"})
fig.show()
```

This plot shows the trend of popularity of music over the years. The x-axis represents the years, while the y-axis represents the popularity of the music.

The line plot visually illustrates how the popularity of music has changed over time. By observing the trend of the line, one can infer whether music popularity has increased, decreased, or remained relatively stable over the specified time period.

Overall, the Popularity of music kept increasing.

5.2 Number of Songs Released per Decade

[16]:
```
# Converting release_date to datetime and extract decade
data['release_decade'] = (data['release_date'].dt.year // 10) * 10

# Counting the number of songs per decade
decade_counts = data['release_decade'].value_counts().sort_index()

# Creating a bar chart for songs per decade
fig = px.bar(x=decade_counts.index, y=decade_counts.values, labels={'x':␣
↪'Decade --->', 'y': 'Number of Songs --->', 'color':"Color"},
             title='Number of Songs Released per Decade', color=decade_counts.
↪index, color_continuous_scale='Rainbow')
fig.update_layout(xaxis_type='category')
fig.show()
```

The above code converts the release dates into decades and then counts the number of songs released in each decade. The resulting plot is a bar chart showing the number of songs released per decade. Each bar represents a decade, and the height of the bar indicates the number of songs released during that decade.

The x-axis represents decades, and the y-axis represents the number of songs. The plot provides an overview of the distribution of songs over different decades, which could help in analyzing trends or patterns in music production over time.

5.3 Changes in Tempo Over the Years

[17]:
```
# Changes in Tempo Over the Years
fig = px.scatter(year_data, x='year', y='tempo', color='tempo',␣
 ↪size='popularity',
                 title='Changes in Tempo Over the Years', labels={'tempo':␣
 ↪'Tempo --->', "year":"Years --->"})
fig.show()
```

From the above code, it is a scatter plot generated to visualize the changes in tempo over the years. Here's what can be inferred:

- **X-axis (year)**: This represents the years over which the tempo data is being analyzed. Each point on the plot likely corresponds to a specific year.
- **Y-axis (tempo)**: This represents the tempo of the music tracks. Tempo is typically measured in beats per minute (BPM). Each point on the plot likely corresponds to the tempo of tracks in a particular year.
- **Color**: The color of the points might represent the tempo values, providing a visual distinction between different tempo ranges or values.
- **Size**: The size of the points might represent the popularity of the tracks. Larger points may indicate more popular tracks, while smaller points may represent less popular ones.
- **Title**: The title of the plot indicates that it visualizes the changes in tempo over the years.

This plot allows viewers to see trends or patterns in tempo changes across different years, and potentially how tempo correlates with track popularity.

5.4 Changes in Energy and Acousticness Over Years

[18]:
```
# Energy and Acousticness Over Years
fig = go.Figure()

fig.add_trace(go.Scatter(x=year_data['year'], y=year_data['energy'],␣
 ↪mode='lines', name='Energy'))
fig.add_trace(go.Scatter(x=year_data['year'], y=year_data['acousticness'],␣
 ↪mode='lines', name='Acousticness'))

fig.update_layout(title='Energy and Acousticness Over Years',␣
 ↪xaxis_title='Years --->', yaxis_title='Values --->')
fig.show()
```

The plot displays how the energy and acousticness of music tracks have changed over the years.

- The x-axis represents the years, indicating the timeframe over which the data has been collected.
- The y-axis represents the values of energy and acousticness.
- There are two lines on the plot:

- The first line (labeled "Energy") represents the trend of energy levels in music tracks over the years.
- The second line (labeled "Acousticness") represents the trend of acoustic characteristics in music tracks over the years.

By observing the plot, one can interpret how these two attributes have varied over time, providing insights into potential trends or patterns in music production or consumption.

5.5 Changes in Speechiness and Instrumentalness Over Years

[19]:
```
# Speechiness and Instrumentalness Over Years
fig = go.Figure()

fig.add_trace(go.Scatter(x=year_data['year'], y=year_data['speechiness'],␣
 ↪mode='lines', name='Speechiness'))
fig.add_trace(go.Scatter(x=year_data['year'], y=year_data['instrumentalness'],␣
 ↪mode='lines', name='Instrumentalness'))

fig.update_layout(title='Speechiness and Instrumentalness Over Years',␣
 ↪xaxis_title='Years --->', yaxis_title='Values --->')
fig.show()
```

The plot displays how the values of speechiness and instrumentalness change over the years. Each line represents one of these attributes, with the x-axis indicating the years and the y-axis representing the values of speechiness and instrumentalness.

- Speechiness: This measures the presence of spoken words in the track. A higher value suggests more spoken words or a higher proportion of speech-like content in the music.
- Instrumentalness: This indicates the likelihood of the track containing no vocals. A higher value suggests a higher probability that the track is instrumental.

By observing the trends of these attributes over the years, one can infer how speechiness and instrumentalness have evolved in the music industry over time. For instance, if speechiness shows an increasing trend while instrumentalness decreases, it might suggest a shift towards more vocal-oriented music. Conversely, if instrumentalness increases while speechiness decreases, it might indicate a rise in instrumental music.

5.6 Genres Analysis

5.6.1 Top Genres by Popularity

[20]:
```
# Genre Analysis: Top Genres by Popularity
top_10_genre_data = genre_data.nlargest(10, 'popularity')

fig = px.bar(top_10_genre_data, x='popularity', y='genres', orientation='h',
             title='Top Genres by Popularity', color='genres',␣
 ↪labels={'popularity': 'Popularity --->', "genres":"Genres --->"})
fig.show()
```

From this plot, the horizontal bars indicate the popularity of each genre, with the genre names labeled on the y-axis and the corresponding popularity values labeled on the x-axis. Each bar's color represents a different genre. The title of the plot is "Top Genres by Popularity", indicating the purpose of the visualization. Overall, it provides a visual representation of the popularity ranking of different music genres.

The most popular genre according to this graph is Basshall, followed by South African House, Trap Venezolano and Turkish EDM. The least popular genres are Guaracha, Circuit, Afro Soul and Afroswing.

5.6.2 Danceability Distribution for Top 10 Popular Genres

[ ]:
```
# Genre Analysis: Danceability Distribution for Top 10 Popular Genres
fig = px.bar(top_10_genre_data, x='genres', y='danceability', color='genres',
             title='Danceability Distribution for Top 10 Popular Genres',␣
 ↪labels={'genres': 'Genres --->', "danceability":"Daneability --->"})
fig.show()
```

This code generates a bar plot using Plotly Express (`px.bar`). The plot displays the distribution of danceability for the top 10 popular genres. Each bar represents a genre, and the height of the bar corresponds to the average danceability score for that genre. The x-axis represents the genres, the y-axis represents the danceability score, and each genre is color-coded for easier differentiation. The title of the plot is "Danceability Distribution for Top 10 Popular Genres", and custom labels are provided for the axes.

Seems like "alberta hip hop" has the most daceability amoung all the other genres.

5.6.3 Energy Distribution for Top 10 Popular Genres

[21]:
```
# Genre Analysis: Energy Distribution for Top 10 Popular Genres
fig = px.bar(top_10_genre_data, x='genres', y='energy', color='genres',
             title='Energy Distribution for Top 10 Popular Genres',␣
 ↪labels={'genres': 'Genres --->', "energy":"Energy --->"})
fig.show()
```

From this plot, it is an analysis of the energy distribution across the top 10 popular genres. Each bar represents a different genre, with the x-axis showing the genre names and the y-axis representing the energy level. The color of each bar corresponds to the respective genre.

The plot provides insight into how the energy levels vary across different music genres, helping to visualize which genres tend to have higher or lower energy levels based on the dataset being analyzed.

5.6.4 Valence Distribution for Top 10 Popular Genres

[22]:
```
# Genre Analysis: Valence Distribution for Top 10 Popular Genres
fig = px.bar(top_10_genre_data, x='genres', y='valence', color='genres',
             title='Valence Distribution for Top 10 Popular Genres',␣
 ↪labels={'genres': 'Genres --->', "valence":"Valence --->"})
```

```
fig.show()
```

From the provided code snippet, it seems that a bar plot is being generated to visualize the valence distribution for the top 10 popular genres.

1. **X-axis**: Represents different genres.
2. **Y-axis**: Represents the valence (musical positiveness) associated with each genre.
3. **Color**: Each bar is colored according to the corresponding genre, making it easier to distinguish between different genres.

By observing this plot, one can analyze how the valence varies across different genres, and which genres tend to have higher or lower valence ratings among the top 10 popular genres.

5.6.5 Acousticness Distribution for Top 10 Popular Genres

[23]:
```
# Genre Analysis: Acousticness Distribution for Top 10 Popular Genres
fig = px.bar(top_10_genre_data, x='genres', y='acousticness', color='genres',
             title='Acousticness Distribution for Top 10 Popular Genres',␣
 ↪labels={'genres': 'Genres --->', "acousticness":"Acousticness --->"})
fig.show()
```

This plot is a bar chart showing the distribution of acousticness across the top 10 popular genres. Each bar represents a genre, with the height of the bar indicating the average acousticness value for that genre. The x-axis represents the genres, while the y-axis represents the acousticness values. Additionally, each genre is color-coded for better visualization. The title of the plot indicates that it's specifically analyzing the acousticness distribution for the top 10 popular genres.

5.6.6 Instrumentalness Distribution for Top 10 Popular Genres

[24]:
```
# Genre Analysis: Instrumentalness Distribution for Top 10 Popular Genres
fig = px.bar(top_10_genre_data, x='genres', y='instrumentalness',␣
 ↪color='genres',
             title='Instrumentalness Distribution for Top 10 Popular Genres',␣
 ↪labels={'genres': 'Genres --->', "instrumentalness":"Instrumentalness --->"})
fig.show()
```

From this plot, it is analyzing the distribution of instrumentalness across the top 10 popular genres. Each genre is represented on the x-axis, and the instrumentalness value for each genre is shown on the y-axis.

- The bars represent the instrumentalness value for each genre.
- The color of the bars indicates the corresponding genre.
- The title of the plot suggests that it's focusing on the instrumentalness distribution across these genres.

This plot provides insights into how instrumental various genres are within the top 10 popular genres, helping to understand the musical characteristics and preferences associated with each genre.

5.7 Artists Analysis

5.7.1 Average Attributes for Top 10 Popular Artists

[25]:
```
# Artist Analysis: Average Attributes for Top 10 Popular Artists
top_10_artist_data = artist_data.nlargest(10, 'popularity')

fig = px.bar(top_10_artist_data, x='popularity', y='artists', orientation='h',␣
 ↪color='artists',
             title='Top Artists by Popularity', labels={'popularity':␣
 ↪'Popularity --->', "artists":"Artists --->"})
fig.show()
```

This plot visualizes the average attributes for the top 10 popular artists. The x-axis represents the popularity score, while the y-axis displays the names of the artists. Each horizontal bar represents one artist, with the length of the bar indicating their popularity score. The color of each bar corresponds to the artist it represents. The title of the plot is "Top Artists by Popularity", and the labels on the axes clarify what they represent.

Overall, this plot provides a visual comparison of the popularity of the top 10 artists in the dataset.

5.7.2 Speechiness vs. Instrumentalness for Top 10 Popular Artists

[26]:
```
# Artist Analysis: Speechiness vs. Instrumentalness for Top 10 Popular Artists

fig = px.scatter(top_10_artist_data, x='speechiness', y='instrumentalness',␣
 ↪color='artists',
                 size='popularity', hover_name='artists',
                 title='Speechiness vs. Instrumentalness for Top Artists',␣
 ↪labels={'speechiness': 'Speechiness --->', "instrumentalness":
 ↪"Instrumentalness --->", "artists":"Artists"})
fig.show()
```

The plot visualizes the relationship between two audio features, "Speechiness" and "Instrumentalness", for the top 10 popular artists.

- **X-axis (speechiness)**: This represents the speech-like quality of the audio tracks. A higher value indicates more presence of spoken words in the music.
- **Y-axis (instrumentalness)**: This indicates the likelihood of the audio tracks being instrumental, i.e., containing no vocals. A higher value suggests a higher probability of the track being instrumental.
- **Color**: Each point on the scatter plot corresponds to a specific artist. The color differentiation helps identify different artists.
- **Size**: The size of each point might represent the popularity of the artist or their tracks. Larger points indicate higher popularity.
- **Hover**: Hovering over a point would likely display additional information, such as the name of the artist.

This plot essentially allows us to explore the speechiness and instrumentalness characteristics of tracks by the top 10 popular artists, providing insights into their musical style preferences.

5.7.3 Danceability vs. Energy for Top 10 Popular Artists

[27]:
```
# Artist Analysis: Danceability vs. Energy for Top 10 Popular Artists
fig = px.scatter(top_10_artist_data, x='danceability', y='energy',␣
 ↪color='artists',
                 size='popularity', hover_name='artists',
                 title='Danceability vs. Energy for Top 10 Popular Artists',␣
 ↪labels={'danceability': 'Danceability --->', "energy":"Energy --->",␣
 ↪"artists":"Artists"})
fig.show()
```

From this plot, we are analyzing the relationship between danceability and energy for the top 10 popular artists. Each point on the scatter plot represents an artist, positioned based on their respective danceability and energy scores. The color of the points distinguishes different artists, and the size of the points represents the popularity of the artists.

By examining the distribution and patterns of the points, we can infer how danceability and energy correlate among the top 10 popular artists. If there are clusters of points in certain regions of the plot, it could suggest common trends or tendencies among these artists regarding the danceability and energy levels of their music. Additionally, by considering the popularity represented by the size of the points, we can identify whether there's any relationship between these musical attributes and an artist's popularity.

5.8 Songs Analysis

5.8.1 Top Songs by Popularity

[28]:
```
# Song Analysis: Top Songs by Popularity
top_songs = data.nlargest(10, 'popularity')

fig = px.bar(top_songs, x='popularity', y='name', orientation='h',
             title='Top Songs by Popularity', color='name',␣
 ↪labels={'popularity': 'Popularity --->', "name":"Name --->"})
fig.show()
```

This code generates a horizontal bar plot showing the top 10 songs based on their popularity. The data for this plot is sourced from the DataFrame, and the top 10 songs are selected using the `nlargest()` function based on the 'popularity' column. The plot displays the popularity of each song on the y-axis and the name of the song on the x-axis. Each bar in the plot is colored differently based on the name of the song.

5.8.2 Danceability vs. Energy for Top 10 Popular Songs

[29]:
```
# Song Analysis: Danceability vs. Energy for Top 10 Popular Songs
fig = px.scatter(top_songs, x='danceability', y='energy', color='popularity',
                 size='popularity', hover_name='name',
                 title='Danceability vs. Energy for Top Songs',␣
 ↪labels={'danceability': 'Danceability --->', "energy":"Energy --->",␣
 ↪"popularity":"Popularity"})
fig.show()
```

The plot visualizes the relationship between danceability and energy for the top 10 popular songs. Each point on the scatter plot represents a song, with its position determined by its danceability (x-axis) and energy (y-axis).

- Danceability: This metric represents how suitable a song is for dancing based on various musical elements.
- Energy: Indicates the intensity and activity level of the song.

Additionally, the color and size of the points represent the popularity of each song. A higher popularity value is likely represented by a larger and/or differently colored point.

The plot helps in understanding if there's any correlation or pattern between danceability, energy, and popularity among the top 10 songs. For example, it could reveal whether more danceable or energetic songs tend to be more popular, or if there's a diverse range of danceability and energy levels among the top songs.

5.8.3 Speechiness vs. Instrumentalness for Top 10 Popular Songs

[30]:
```
# Song Analysis: Speechiness vs. Instrumentalness for Top 10 Popular Songs
fig = px.scatter(top_songs, x='speechiness', y='instrumentalness',␣
 ↪color='popularity',
                 size='popularity', hover_name='name',
                 title='Speechiness vs. Instrumentalness for Top Songs',␣
 ↪labels={'speechiness': 'Speechiness --->', "instrumentalness":
 ↪"Instrumentalness --->", "popularity":"Popularity"})
fig.show()
```

This plot shows a scatter plot visualizing the relationship between speechiness and instrumentalness of the top 10 popular songs. Here's what can be inferred from the plot:

- **X-axis (Speechiness)**: This represents the degree of speech-like elements present in the songs. Higher values indicate tracks with more spoken words or vocal elements.
- **Y-axis (Instrumentalness)**: This axis measures the absence of vocal content in the songs. Higher values suggest tracks that are more instrumental, i.e., containing little to no vocals.
- **Color**: The color of each data point represents the popularity of the song. This could be a gradient scale where darker colors indicate higher popularity.
- **Size**: The size of each data point may also correspond to the popularity of the song. Larger points likely indicate more popular songs.

- **Hover Information**: When hovering over each point, additional information such as the name of the song may be displayed.

The plot aims to visualize whether there's any discernible pattern or correlation between the speechiness and instrumentalness of the top 10 popular songs. It helps in understanding the characteristics of these songs in terms of vocal content and popularity.

6 Building the Music Recommender System

[31]:
```
# Convert year column back
data['year'] = data['year'].dt.year
```

[32]:
```
# List of numerical columns to consider for similarity calculations
number_cols = ['valence', 'year', 'acousticness', 'danceability',␣
↪'duration_ms', 'energy', 'explicit', 'year',
               'instrumentalness', 'key', 'liveness', 'loudness', 'mode',␣
↪'popularity', 'speechiness', 'tempo']
```

[33]:
```
# Function to retrieve song data for a given song name
def get_song_data(name, data):
    try:
        return data[data['name'].str.lower() == name].iloc[0]
        return song_data
    except IndexError:
        return None
```

This code defines a function called `get_song_data` that retrieves data for a given song name from a dataset. Here's a short explanation of what it does:

- **Input**: It takes two parameters: `name` (the name of the song) and `data` (the dataset containing song information).
- **Operation**:
  - It tries to find the song in the dataset by matching the lowercase version of the song name with the lowercase version of the names in the dataset.
  - If a match is found, it returns the data (row) for that song.
  - If no match is found, it returns `None`.
- **Output**: It returns either the data for the song if it's found in the dataset or `None` if the song is not found.

[34]:
```
# Function to calculate the mean vector of a list of songs
def get_mean_vector(song_list, data):
    song_vectors = []
    for song in song_list:
        song_data = get_song_data(song['name'], data)
        if song_data is None:
```

```
            print('Warning: {} does not exist in the dataset'.
↪format(song['name']))
            return None
        song_vector = song_data[number_cols].values
        song_vectors.append(song_vector)
    song_matrix = np.array(list(song_vectors))
    return np.mean(song_matrix, axis=0)
```

This code defines a function `get_mean_vector()` which calculates the mean vector of a list of songs. Here's what it does:

1. **Input**:
   - `song_list`: A list of dictionaries where each dictionary represents a song, typically containing information like the song's name.
   - `data`: The dataset containing information about songs.
2. **Processing**:
   - It iterates through each song in the `song_list`.
   - For each song, it retrieves the corresponding data from the dataset using a function called `get_song_data()`.
   - If the song is not found in the dataset, it prints a warning message and returns `None`.
   - Otherwise, it extracts the numerical data (presumably features of the song) from the dataset and appends it to a list called `song_vectors`.
   - It converts the list of vectors (`song_vectors`) into a numpy array called `song_matrix`.
   - Finally, it calculates the mean vector of all the song vectors along each dimension (column) and returns it.
3. **Output**:
   - The function returns the mean vector of the songs in the input list.

In summary, this function provides a way to compute the average numerical features of a collection of songs, which could be useful for various analytical purposes.

[35]:
```
# Function to flatten a list of dictionaries into a single dictionary
def flatten_dict_list(dict_list):
    flattened_dict = defaultdict()
    for key in dict_list[0].keys():
        flattened_dict[key] = []
    for dictionary in dict_list:
        for key, value in dictionary.items():
            flattened_dict[key].append(value)
    return flattened_dict
```

This Python code defines a function called `flatten_dict_list`. It takes a list of dictionaries (`dict_list`) as input. The function iterates through each dictionary in the list and flattens it into a single dictionary (`flattened_dict`). It initializes `flattened_dict` as an empty defaultdict and then iterates over each key-value pair in each dictionary, appending the values to lists corresponding to their respective keys in `flattened_dict`. Finally, it returns the flattened dictionary. Essentially, it transforms a list of dictionaries into a single dictionary where each key holds a list of values from the original dictionaries under that key.

[36]:
```
# Normalize the song data using Min-Max Scaler
min_max_scaler = MinMaxScaler()
normalized_data = min_max_scaler.fit_transform(data[number_cols])

# Standardize the normalized data using Standard Scaler
standard_scaler = StandardScaler()
scaled_normalized_data = standard_scaler.fit_transform(normalized_data)
```

This code performs two types of scaling on the song data:

1. **Min-Max Scaling (Normalization):** The data is scaled between a specified range (usually 0 and 1) using Min-Max Scaler. This ensures that all features have the same scale, preserving the relative differences between them.
2. **Standardization:** After normalization, the data is standardized using Standard Scaler. This process standardizes the features by removing the mean and scaling to unit variance. It transforms the data so that it has a mean of 0 and a standard deviation of 1.

In summary, these two steps ensure that the song data is both normalized (within a specified range) and standardized (with a mean of 0 and a standard deviation of 1), making it suitable for various machine learning algorithms that require standardized input data.

[37]:
```
# Function to recommend songs based on a list of seed songs
def recommend_songs(seed_songs, data, n_recommendations=10):
    metadata_cols = ['name', 'artists', 'year']
    song_center = get_mean_vector(seed_songs, data)

    # Return an empty list if song_center is missing
    if song_center is None:
        return []

    # Normalize the song center
    normalized_song_center = min_max_scaler.transform([song_center])

    # Standardize the normalized song center
    scaled_normalized_song_center = standard_scaler.
↪transform(normalized_song_center)

    # Calculate Euclidean distances and get recommendations
    distances = cdist(scaled_normalized_song_center, scaled_normalized_data,␣
↪'euclidean')
    index = np.argsort(distances)[0]

    # Filter out seed songs and duplicates, then get the top n_recommendations
    rec_songs = []
    for i in index:
        song_name = data.iloc[i]['name']
        if song_name not in [song['name'] for song in seed_songs] and song_name␣
↪not in [song['name'] for song in rec_songs]:
```

```
            rec_songs.append(data.iloc[i])
            if len(rec_songs) == n_recommendations:
                break

    return pd.DataFrame(rec_songs)[metadata_cols].to_dict(orient='records')
```

This code defines a function `recommend_songs()` that generates song recommendations based on a list of seed songs and a dataset containing information about various songs.

Here's a breakdown of what the code does:

1. **Input Parameters**:
   - `seed_songs`: A list of seed songs represented as dictionaries containing song information like name, artists, and year.
   - `data`: The dataset containing information about songs.
   - `n_recommendations`: The number of song recommendations to generate (default is 10).
2. **Extract Metadata Columns**:
   - `metadata_cols`: Defines a list of columns to include in the final recommendation, such as name, artists, and year.
3. **Calculate Song Center**:
   - Calls a function `get_mean_vector()` to calculate the mean vector of the seed songs in the dataset.
4. **Normalization and Standardization**:
   - Normalizes the song center using min-max scaling.
   - Standardizes the normalized song center using standard scaling.
5. **Calculate Distances and Get Recommendations**:
   - Calculates Euclidean distances between the standardized song center and all songs in the dataset.
   - Sorts the distances to find the closest songs.
   - Filters out seed songs and duplicates from the recommendations.
   - Selects the top `n_recommendations` songs based on distance.
6. **Return Recommendations**:
   - Returns the recommendations as a list of dictionaries, where each dictionary contains metadata of a recommended song (name, artists, year).

Overall, this function takes a set of seed songs, calculates their mean representation, compares it with other songs in the dataset, and returns a list of recommended songs that are most similar to the provided seed songs.

[38]:
```
# List of seed songs (replace with your own seed songs)
seed_songs = [
    {'name': 'Come As You Are'},
    {'name': 'Smells Like Teen Spirit'},
    # Add more seed songs as needed
]
seed_songs = [{'name': name['name'].lower()} for name in seed_songs]

# Number of recommended songs
n_recommendations = 10
```

```
# Call the recommend_songs function
recommended_songs = recommend_songs(seed_songs, data, n_recommendations)

# Convert the recommended songs to a DataFrame
recommended_df = pd.DataFrame(recommended_songs)

# Print the recommended songs
for idx, song in enumerate(recommended_songs, start=1):
    print(f"{idx}. {song['name']} by {song['artists']} ({song['year']})")
```

```
1. No Excuses by ['Alice In Chains'] (1994)
2. Come As You Are by ['Nirvana'] (1991)
3. Smells Like Teen Spirit by ['Nirvana'] (1991)
4. Born in the U.S.A. by ['Bruce Springsteen'] (1984)
5. Breakfast At Tiffany's by ['Deep Blue Something'] (1995)
6. Malibu by ['Hole'] (1998)
7. Fuel by ['Metallica'] (1997)
8. Sleep Now In the Fire by ['Rage Against The Machine'] (1999)
9. When You're Gone by ['Bryan Adams', 'Melanie C'] (1998)
10. Outshined by ['Soundgarden'] (1991)
```

This code performs the following tasks:

1. **Seed Songs Definition**: It defines a list named `seed_songs`, which contains dictionaries representing seed songs. Each dictionary has a key-value pair where the key is 'name' and the value is the name of a song. It's then converted to lowercase to ensure consistency.
2. **Number of Recommendations**: It sets the variable `n_recommendations` to the number of recommended songs desired, in this case, 10.
3. **Call to recommend_songs Function**: It calls a hypothetical function named `recommend_songs` with the seed songs, the dataset (`data`), and the number of recommendations as arguments. This function presumably returns a list of recommended songs based on the provided seed songs and the dataset.
4. **Conversion to DataFrame**: It converts the list of recommended songs into a pandas DataFrame named `recommended_df`.
5. **Printing Recommended Songs**: It iterates through the list of recommended songs and prints each song's name, artists, and year of release. The `enumerate()` function is used to iterate through the list with an index starting from 1 for better readability.

In summary, this code takes a list of seed songs, generates recommendations based on those seeds and a dataset of songs, converts the recommendations into a DataFrame, and then prints out the recommended songs with their details.

[39]:
```
# Create a bar plot of recommended songs by name
recommended_df['text'] = recommended_df.apply(lambda row: f"{row.name + 1}.␣
↪{row['name']} by {row['artists']} ({row['year']})", axis=1)
```

```
fig = px.bar(recommended_df, y='name', x=range(n_recommendations, 0, -1),␣
 ↪title='Recommended Songs', orientation='h', color='name', text='text')
fig.update_layout(xaxis_title='Recommendation Rank', yaxis_title='Songs',␣
 ↪showlegend=False, uniformtext_minsize=20, uniformtext_mode='show',␣
 ↪yaxis_showticklabels=False, height=1000)
fig.update_traces(width=1)
fig.show()
```

Dear Readers and Visitors,

Thank you for taking the time to explore this Jupyter notebook! We hope you've found the content informative and engaging. Your support means a lot to us.

We encourage you to leave your thoughts, comments, and feedback in the comments section.

If you found this notebook valuable, we kindly ask you to consider liking it and sharing it with others who might also benefit from it.

Once again, thank you for visiting!!