```python
#pip install mlxtend

#Import all necessary libraries

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from statsmodels.stats.outliers_influence import
variance_inflation_factor
from sklearn.preprocessing import StandardScaler
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
from sklearn.model_selection import train_test_split, cross_val_score,
GridSearchCV
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor

# Load the dataset
data_songs = pd.read_csv(r'C:\Users\lakpr\Desktop\CS 6375 - Machine
Learning\Final project\spotify_songs.csv')

# Verify the dataset by displaying the first few rows of the dataset
data_songs.head()

# Check the structure of the dataset to ensure what was specified on
the kaggle link is right
print(data_songs.info())

# Display basic statistics to observe skewness or other measures to
see how data is distributed and check if statistics match range values
of variables
print(data_songs.describe())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32833 entries, 0 to 32832
Data columns (total 23 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   track_id                  32833 non-null  object
 1   track_name                32828 non-null  object
 2   track_artist              32828 non-null  object
 3   track_popularity          32833 non-null  int64
 4   track_album_id            32833 non-null  object
 5   track_album_name          32828 non-null  object
 6   track_album_release_date  32833 non-null  object
 7   playlist_name             32833 non-null  object
```

```
 8   playlist_id               32833 non-null  object 
 9   playlist_genre            32833 non-null  object 
 10  playlist_subgenre         32833 non-null  object 
 11  danceability              32833 non-null  float64
 12  energy                    32833 non-null  float64
 13  key                       32833 non-null  int64  
 14  loudness                  32833 non-null  float64
 15  mode                      32833 non-null  int64  
 16  speechiness               32833 non-null  float64
 17  acousticness              32833 non-null  float64
 18  instrumentalness          32833 non-null  float64
 19  liveness                  32833 non-null  float64
 20  valence                   32833 non-null  float64
 21  tempo                     32833 non-null  float64
 22  duration_ms               32833 non-null  int64  
dtypes: float64(9), int64(4), object(10)
memory usage: 5.8+ MB
None
       track_popularity  danceability        energy           key  \
count      32833.000000  32833.000000  32833.000000  32833.000000   
mean          42.477081      0.654850      0.698619      5.374471   
std           24.984074      0.145085      0.180910      3.611657   
min            0.000000      0.000000      0.000175      0.000000   
25%           24.000000      0.563000      0.581000      2.000000   
50%           45.000000      0.672000      0.721000      6.000000   
75%           62.000000      0.761000      0.840000      9.000000   
max          100.000000      0.983000      1.000000     11.000000   

           loudness          mode   speechiness  acousticness  \
count  32833.000000  32833.000000  32833.000000  32833.000000   
mean      -6.719499      0.565711      0.107068      0.175334   
std        2.988436      0.495671      0.101314      0.219633   
min      -46.448000      0.000000      0.000000      0.000000   
25%       -8.171000      0.000000      0.041000      0.015100   
50%       -6.166000      1.000000      0.062500      0.080400   
75%       -4.645000      1.000000      0.132000      0.255000   
max        1.275000      1.000000      0.918000      0.994000   

       instrumentalness      liveness       valence         tempo  \
count      32833.000000  32833.000000  32833.000000  32833.000000   
mean           0.084747      0.190176      0.510561    120.881132   
std            0.224230      0.154317      0.233146     26.903624   
min            0.000000      0.000000      0.000000      0.000000   
25%            0.000000      0.092700      0.331000     99.960000   
50%            0.000016      0.127000      0.512000    121.984000   
75%            0.004830      0.248000      0.693000    133.918000   
max            0.994000      0.996000      0.991000    239.440000   

         duration_ms  
count   32833.000000  
```

```
mean   225799.811622
std     59834.006182
min      4000.000000
25%    187819.000000
50%    216000.000000
75%    253585.000000
max    517810.000000
```

```
#---------------------------------------------------------DATA
CLEANING------------------------------------------------------------
---

# Check for missing values and delete records with missing values for
this project
initial_na = data_songs.isnull().sum()
print(initial_na)

# Drop NA values and calculate sum of missing values to double-check
removal.
data_songs = data_songs.dropna()

final_na = data_songs.isnull().sum()
print(final_na)

# How many observations and variables are present in this dataset
after removing null values. Initially we had 32833 observations, after
removal we
#have 5 less observations which correspond to the number of missing
values in "initial_na" and 23 variables
data_songs.shape
```

```
track_id                    0
track_name                  5
track_artist                5
track_popularity            0
track_album_id              0
track_album_name            5
track_album_release_date    0
playlist_name               0
playlist_id                 0
playlist_genre              0
playlist_subgenre           0
danceability                0
energy                      0
key                         0
loudness                    0
mode                        0
speechiness                 0
acousticness                0
instrumentalness            0
```

```
liveness                    0
valence                     0
tempo                       0
duration_ms                 0
dtype: int64
track_id                    0
track_name                  0
track_artist                0
track_popularity            0
track_album_id              0
track_album_name            0
track_album_release_date    0
playlist_name               0
playlist_id                 0
playlist_genre              0
playlist_subgenre           0
danceability                0
energy                      0
key                         0
loudness                    0
mode                        0
speechiness                 0
acousticness                0
instrumentalness            0
liveness                    0
valence                     0
tempo                       0
duration_ms                 0
dtype: int64

(32828, 23)
```

```
# Delete irrelevant columns. For this project since we focus on
regression, we delete qualitative features like subgenre and name of
song. We also
#delete ID's because they are irrelevent attributes and may cause
inacccuracy in analysis.
data_songs.columns
cols_to_drop = ['track_id', 'track_album_id', 'playlist_name',
'playlist_id', 'playlist_subgenre']
data_songs = data_songs.drop(columns=cols_to_drop)

# Check structure to ensure drop
data_songs.head()
```

```
                                           track_name      track_artist
\
0  I Don't Care (with Justin Bieber) - Loud Luxur...        Ed Sheeran

1                     Memories - Dillon Francis Remix         Maroon 5
```

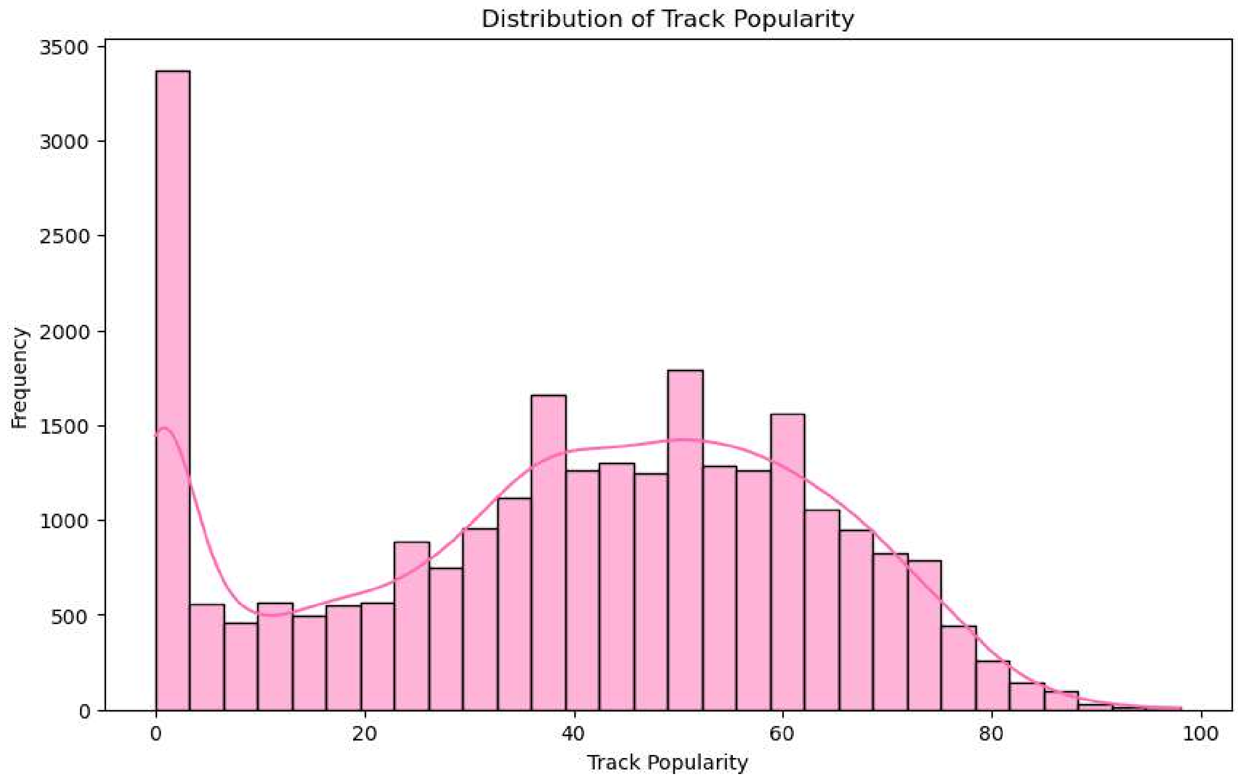
```
2                     All the Time - Don Diablo Remix      Zara Larsson
3                    Call You Mine - Keanu Silva Remix  The Chainsmokers
4            Someone You Loved - Future Humans Remix     Lewis Capaldi

   track_popularity                                  track_album_name  \
0                66  I Don't Care (with Justin Bieber) [Loud Luxury...
1                67                   Memories (Dillon Francis Remix)
2                70                   All the Time (Don Diablo Remix)
3                60                       Call You Mine - The Remixes
4                69           Someone You Loved (Future Humans Remix)

  track_album_release_date playlist_genre  danceability  energy  key  \
0               2019-06-14            pop         0.748   0.916    6
1               2019-12-13            pop         0.726   0.815   11
2               2019-07-05            pop         0.675   0.931    1
3               2019-07-19            pop         0.718   0.930    7
4               2019-03-05            pop         0.650   0.833    1

   loudness  mode  speechiness  acousticness  instrumentalness  liveness  \
0    -2.634     1       0.0583        0.1020          0.000000    0.0653
1    -4.969     1       0.0373        0.0724          0.004210    0.3570
2    -3.432     0       0.0742        0.0794          0.000023    0.1100
3    -3.778     1       0.1020        0.0287          0.000009    0.2040
4    -4.672     1       0.0359        0.0803          0.000000    0.0833

   valence    tempo  duration_ms
0    0.518  122.036       194754
1    0.693   99.972       162600
2    0.613  124.008       176616
```

```
3    0.277  121.956       169093
4    0.725  123.976       189052

# Analysis of unique and duplicate values
# Count of Unique values
for col in data_songs.columns:
    num_unique_values = data_songs[col].nunique()
    print(f"Number of unique values in {col}: {num_unique_values}")

#Since there are some duplicates, we want to analyse them further

# Check for duplicates to determine whatkind of records does each
variable have.
# Loop through each column ofteh dataset to count duplicates
for col in data_songs.columns:
    num_duplicates = data_songs.duplicated(subset=[col]).sum()
    print(f"Number of duplicate values in {col}: {num_duplicates}")

# Drop duplicates based on 'track_name' and 'track_artist' to avoid
deleting different songs with the same name
data_songs = data_songs.drop_duplicates(subset=['track_name',
'track_artist'], keep='first')
data_songs

Number of unique values in track_name: 23449
Number of unique values in track_artist: 10692
Number of unique values in track_popularity: 101
Number of unique values in track_album_name: 19743
Number of unique values in track_album_release_date: 4529
Number of unique values in playlist_genre: 6
Number of unique values in danceability: 822
Number of unique values in energy: 952
Number of unique values in key: 12
Number of unique values in loudness: 10222
Number of unique values in mode: 2
Number of unique values in speechiness: 1270
Number of unique values in acousticness: 3731
Number of unique values in instrumentalness: 4729
Number of unique values in liveness: 1624
Number of unique values in valence: 1362
Number of unique values in tempo: 17682
Number of unique values in duration_ms: 19782
Number of duplicate values in track_name: 9379
Number of duplicate values in track_artist: 22136
Number of duplicate values in track_popularity: 32727
Number of duplicate values in track_album_name: 13085
Number of duplicate values in track_album_release_date: 28299
Number of duplicate values in playlist_genre: 32822
Number of duplicate values in danceability: 32006
Number of duplicate values in energy: 31876
```

```
Number of duplicate values in key: 32816
Number of duplicate values in loudness: 22606
Number of duplicate values in mode: 32826
Number of duplicate values in speechiness: 31558
Number of duplicate values in acousticness: 29097
Number of duplicate values in instrumentalness: 28099
Number of duplicate values in liveness: 31204
Number of duplicate values in valence: 31466
Number of duplicate values in tempo: 15146
Number of duplicate values in duration_ms: 13046

                                              track_name
track_artist  \
0      I Don't Care (with Justin Bieber) - Loud Luxur...        Ed
Sheeran
1                        Memories - Dillon Francis Remix
Maroon 5
2                        All the Time - Don Diablo Remix      Zara
Larsson
3                      Call You Mine - Keanu Silva Remix  The
Chainsmokers
4                Someone You Loved - Future Humans Remix     Lewis
Capaldi
...                                                  ...
...
32828              City Of Lights - Official Radio Edit      Lush &
Simon
32829               Closer - Sultan & Ned Shepard Remix    Tegan and
Sara
32830                       Sweet Surrender - Radio Edit
Starkillers
32831                     Only For You - Maor Levi Remix
Mat Zo
32832                             Typhoon - Original Mix      Julian
Calor

       track_popularity
track_album_name  \
0                    66  I Don't Care (with Justin Bieber) [Loud
Luxury...
1                    67                    Memories (Dillon Francis
Remix)
2                    70                    All the Time (Don Diablo
Remix)
3                    60                        Call You Mine - The
Remixes
4                    69            Someone You Loved (Future Humans
Remix)
...                 ...
...
```

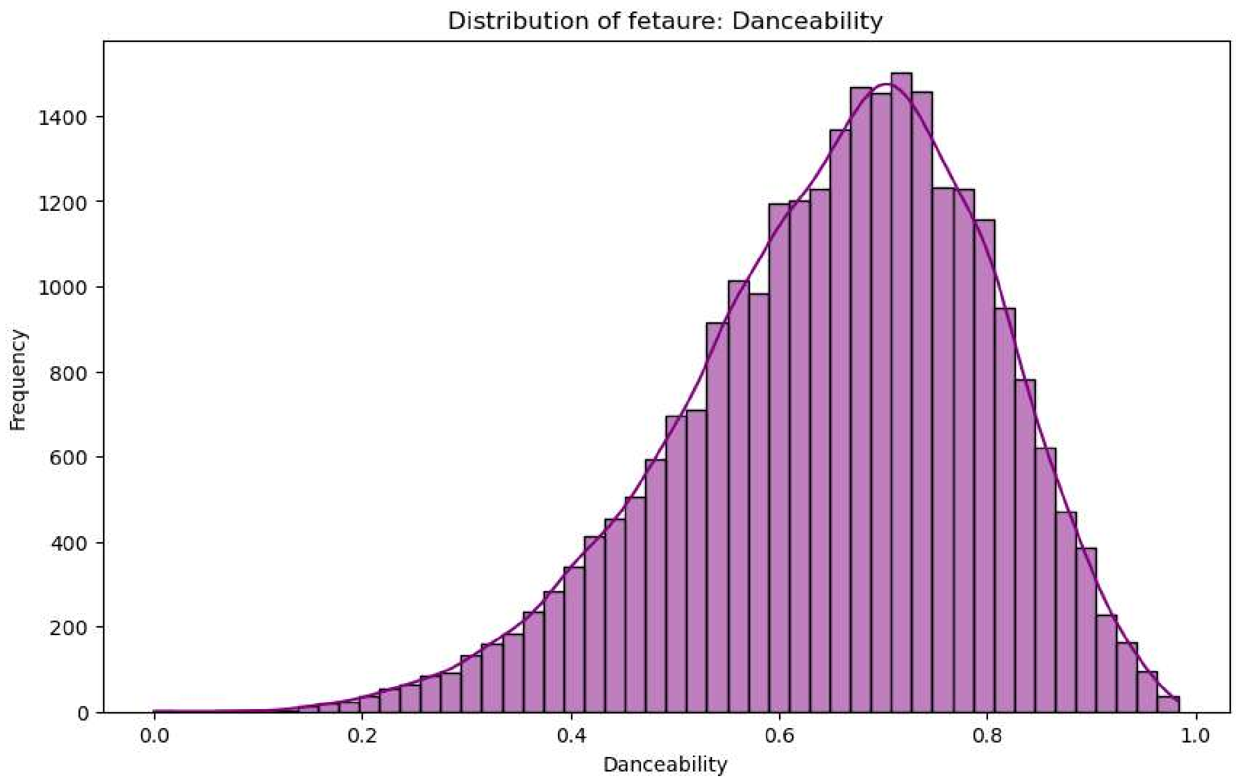
```
32828                42                               City Of Lights (Vocal
Mix)
32829                20                                                Closer
Remixed
32830                14                             Sweet Surrender (Radio
Edit)
32831                15                                   Only For You
(Remixes)
32832                27
Typhoon/Storm

      track_album_release_date playlist_genre  danceability  energy
key  \
0                   2019-06-14            pop         0.748   0.916
6
1                   2019-12-13            pop         0.726   0.815
11
2                   2019-07-05            pop         0.675   0.931
1
3                   2019-07-19            pop         0.718   0.930
7
4                   2019-03-05            pop         0.650   0.833
1
...                        ...            ...           ...     ...  .
..
32828               2014-04-28            edm         0.428   0.922
2
32829               2013-03-08            edm         0.522   0.786
0
32830               2014-04-21            edm         0.529   0.821
6
32831               2014-01-01            edm         0.626   0.888
2
32832               2014-03-03            edm         0.603   0.884
5

       loudness  mode  speechiness  acousticness  instrumentalness
liveness  \
0        -2.634     1       0.0583      0.102000          0.000000
0.0653
1        -4.969     1       0.0373      0.072400          0.004210
0.3570
2        -3.432     0       0.0742      0.079400          0.000023
0.1100
3        -3.778     1       0.1020      0.028700          0.000009
0.2040
4        -4.672     1       0.0359      0.080300          0.000000
0.0833
...         ...   ...          ...           ...               ...
```

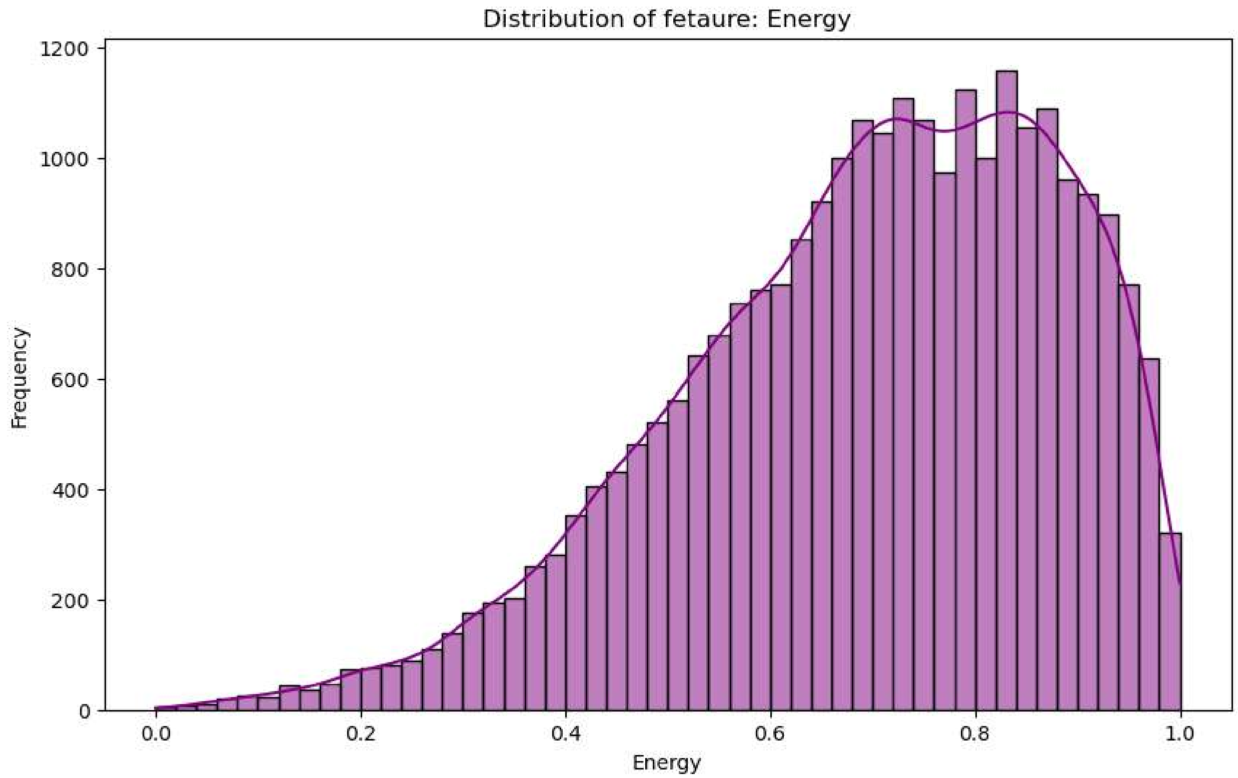
```
...
32828    -1.814     1       0.0936      0.076600          0.000000
0.0668
32829    -4.462     1       0.0420      0.001710          0.004270
0.3750
32830    -4.899     0       0.0481      0.108000          0.000001
0.1500
32831    -3.361     1       0.1090      0.007920          0.127000
0.3430
32832    -4.571     0       0.0385      0.000133          0.341000
0.7420

       valence    tempo  duration_ms
0       0.5180  122.036       194754
1       0.6930   99.972       162600
2       0.6130  124.008       176616
3       0.2770  121.956       169093
4       0.7250  123.976       189052
...        ...      ...          ...
32828   0.2100  128.170       204375
32829   0.4000  128.041       353120
32830   0.4360  127.989       210112
32831   0.3080  128.008       367432
32832   0.0894  127.984       337500

[26229 rows x 18 columns]

#---------------------------------------------------------EXPLORATORY
DATA
ANALYSIS-----------------------------------------------------------------
---

# Plotting the distribution of song popularity which is the response
variable we are interested in
plt.figure(figsize = (10,6))
sns.histplot(data_songs['track_popularity'], kde = True, bins = 30,
color = 'hotpink')
plt.title('Distribution of Track Popularity')
plt.xlabel('Track Popularity')
plt.ylabel('Frequency')
plt.show()

# Mostly bell shaped curve but one observation makes it skewed to the
right. We might have to transform our response variable. Posiible
considerations
#square root, cube root or log.
```

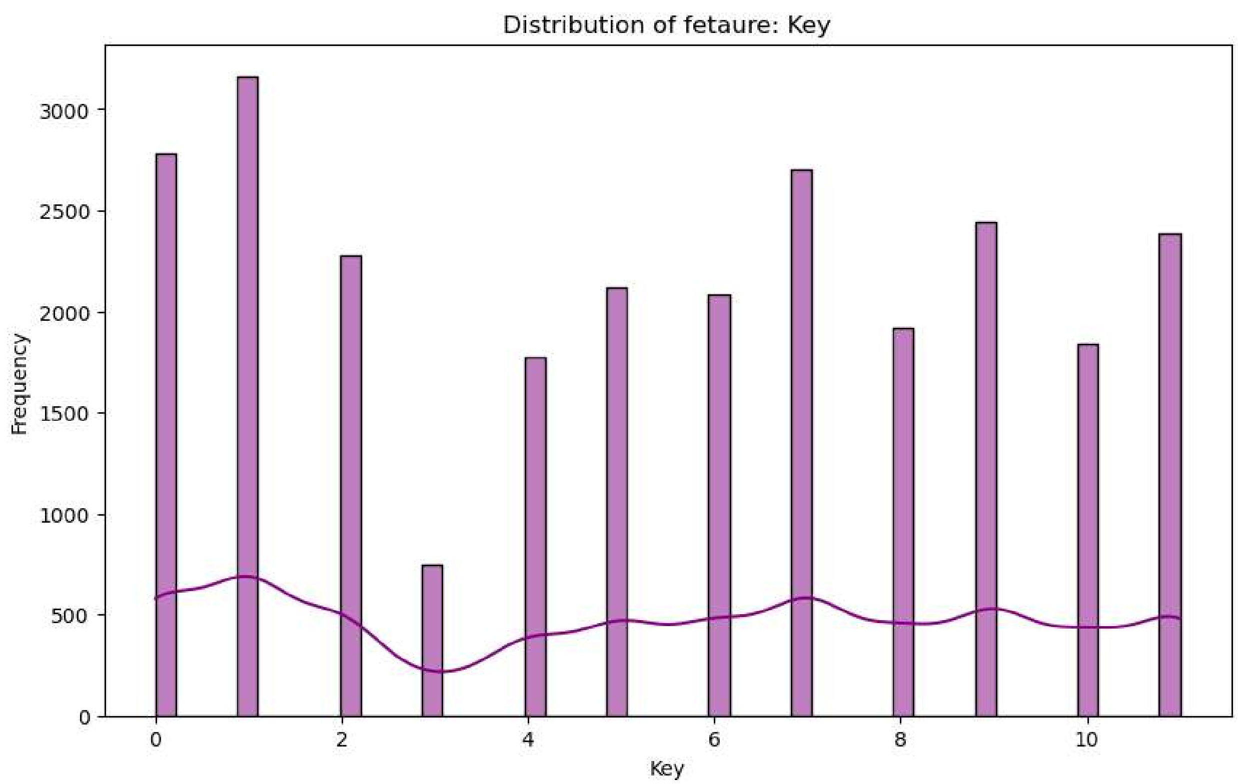


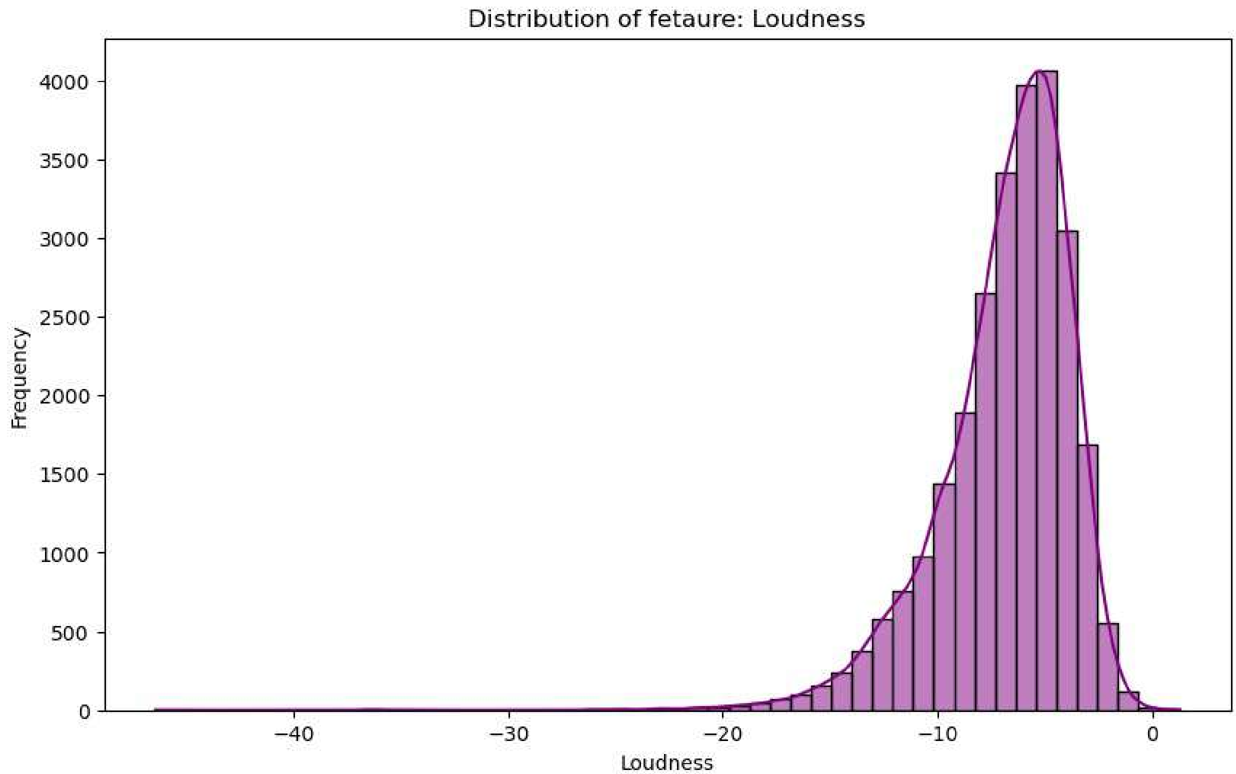

```
# Visualize distributions of other quantitative variables to notice
any patterns

# List of columns to plot histograms for are the ones we want to
consider as features.
col_to_plot = ['danceability', 'energy', 'key', 'loudness',
'speechiness', 'acousticness', 'instrumentalness', 'liveness',
'valence', 'tempo']

# Define a color for the histograms
hist_color = '#800080'

# Plot histograms for each column defined above
for column in col_to_plot:
    plt.figure(figsize = (10, 6))
    sns.histplot(data_songs[column], bins = 50, kde = True, color =
hist_color)
    plt.title(f'Distribution of fetaure: {column.capitalize()}')
    plt.xlabel(column.capitalize())
    plt.ylabel('Frequency')
    plt.show()

# All these variables are highly skewed. We will center and scale them
before fitting models using "StandardScaler" package.

# Even though we are not interested in genre classification it is good
```

```
to know number of songs per genre.
genre_counts = data_songs['playlist_genre'].value_counts()
genre_counts.plot(kind = 'bar')
plt.title('Number of Songs per Genre')
plt.xlabel('Genre')
plt.ylabel('Number of Songs')
plt.show()

# Proportion of songs in different genres
genre_counts.plot(kind = 'pie', autopct='%1.2f%%')
plt.title('Proportion of Songs in Different Genres')
plt.ylabel('')
plt.show()
```

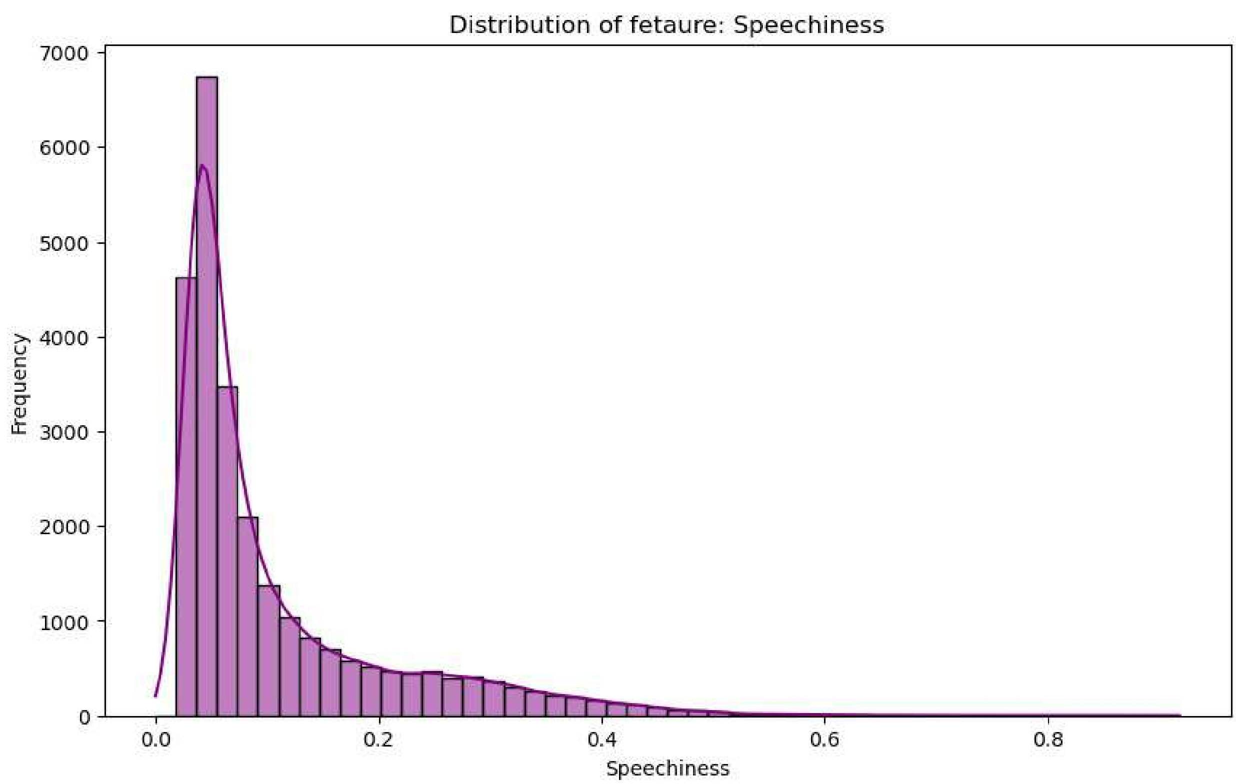



Distribution of fetaure: Energy
Frequency
Energy
0
200
400
600
800
1000
1200
0.0
0.2
0.4
0.6
0.8
1.0

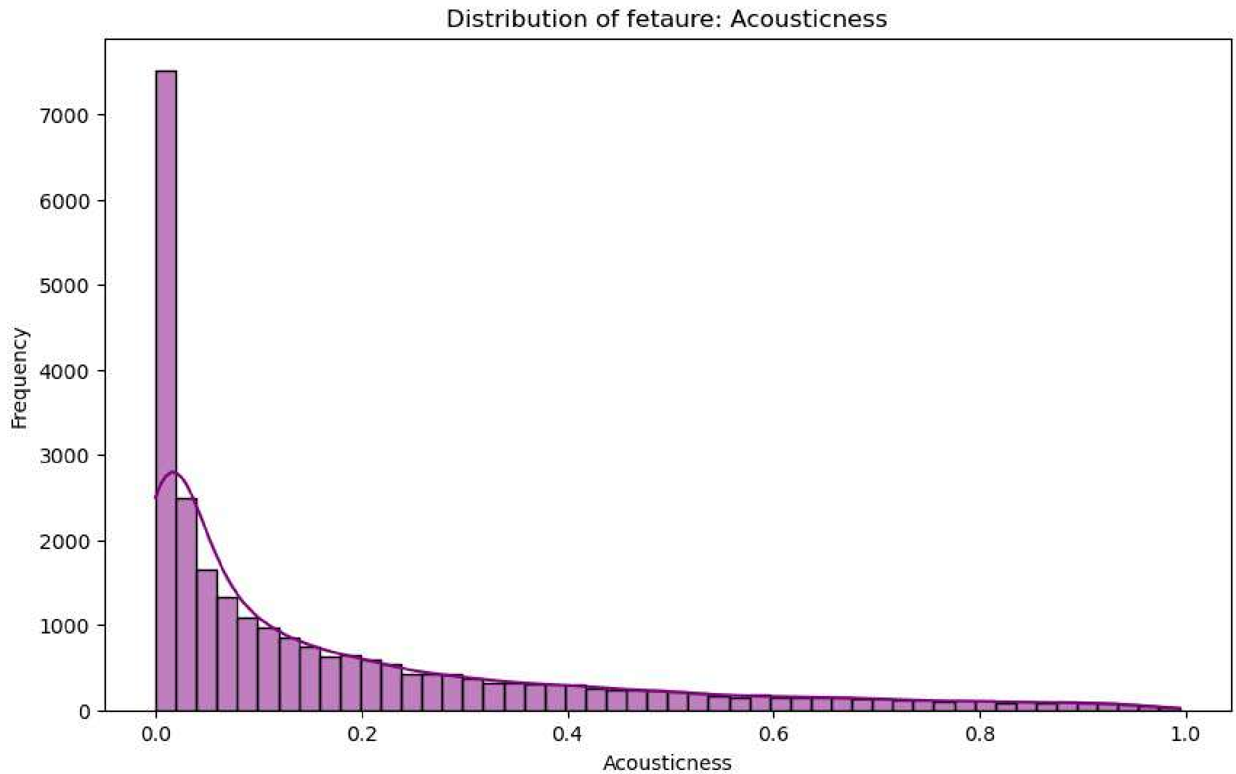

Distribution of fetaure: Key
Frequency
Key
0
500
1000
1500
2000
2500
3000
0
2
4
6
8
10



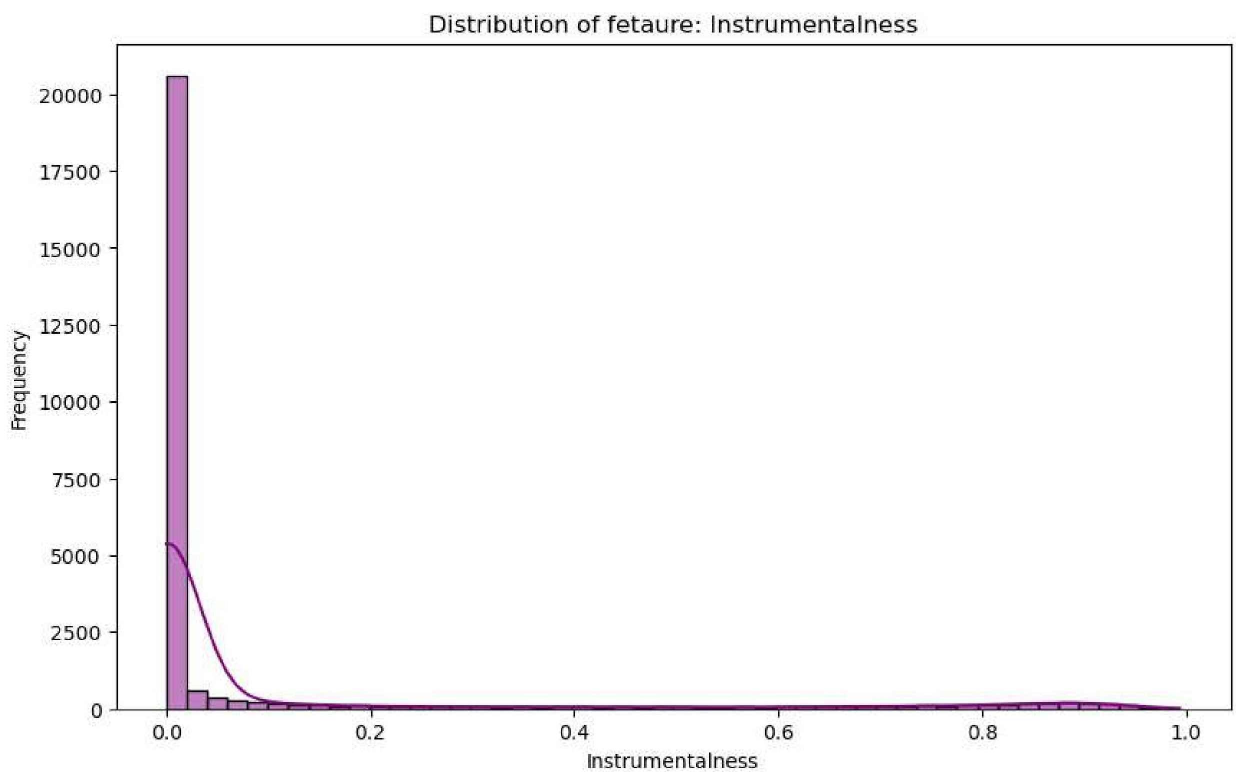
Distribution of fetaure: Loudness
Frequency
Loudness
0
500
1000
1500
2000
2500
3000
3500
4000
−40
−30
−20
−10
0



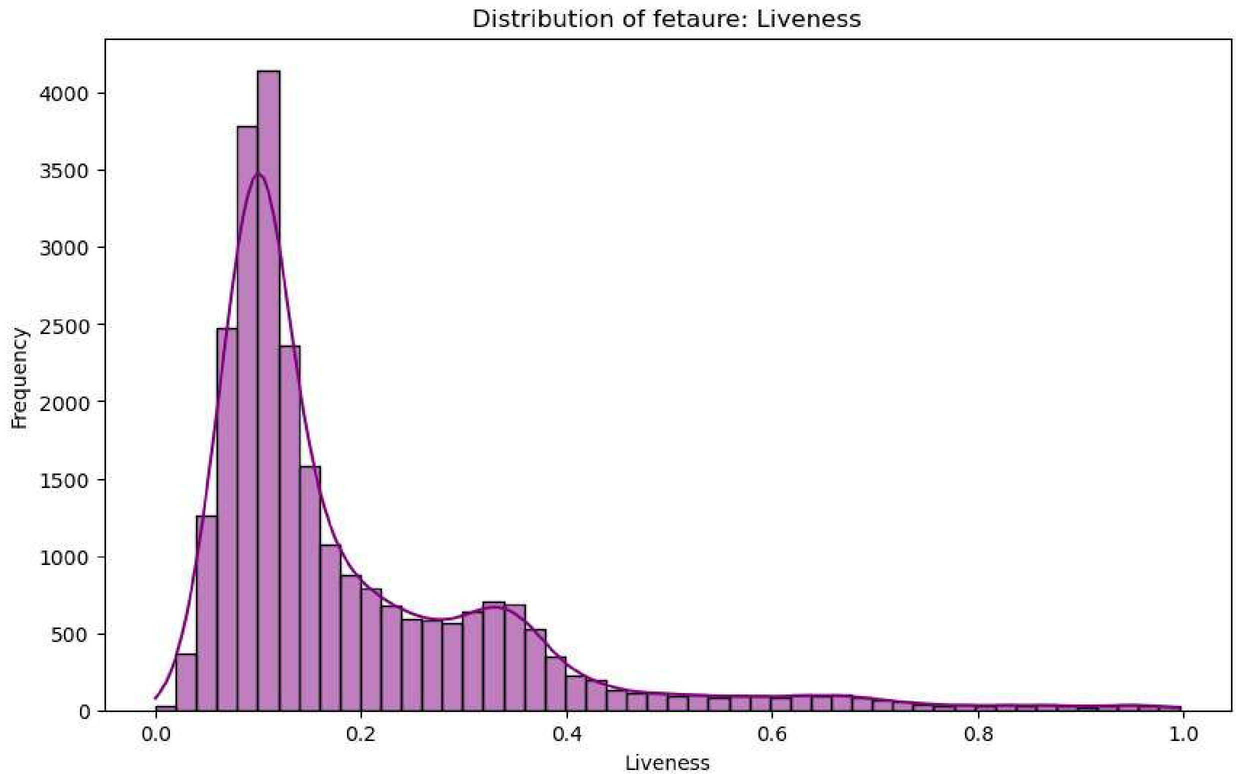
Distribution of fetaure: Speechiness
Frequency
Speechiness
0
1000
2000
3000
4000
5000
6000
7000
0.0
0.2
0.4
0.6
0.8

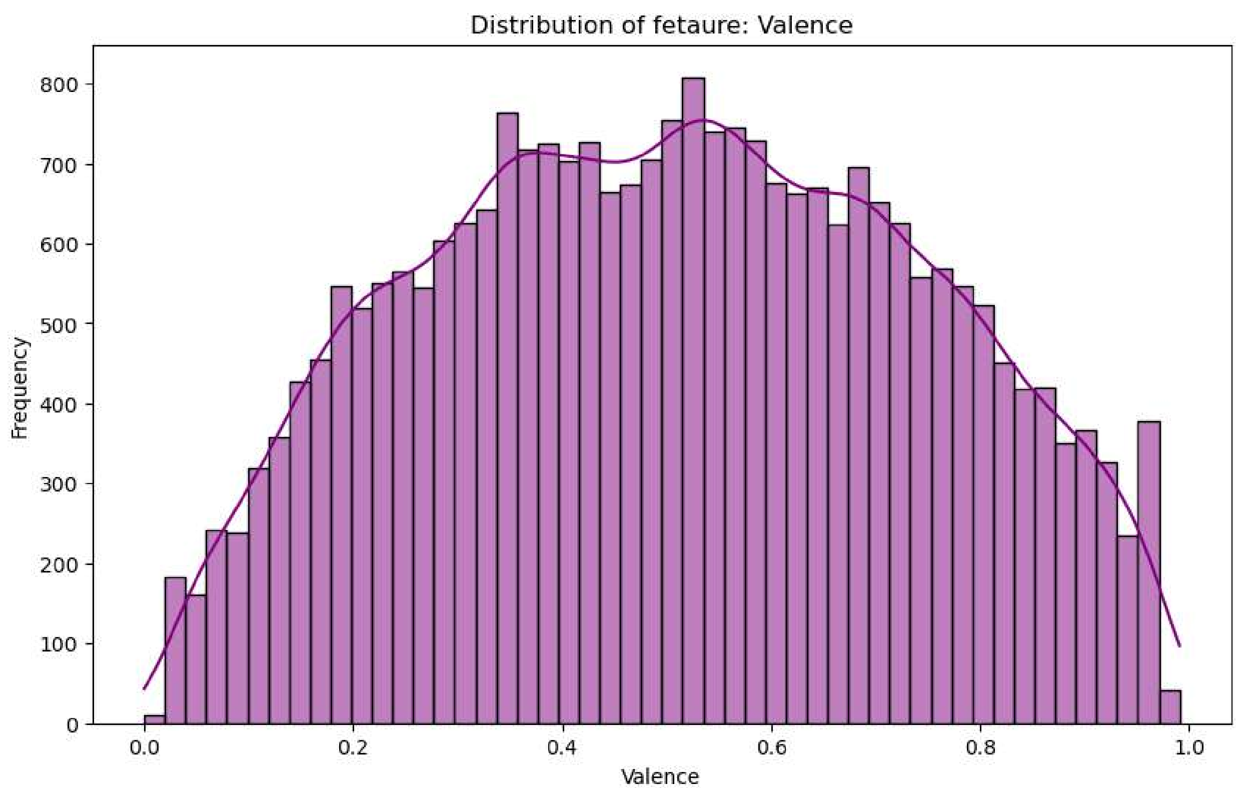


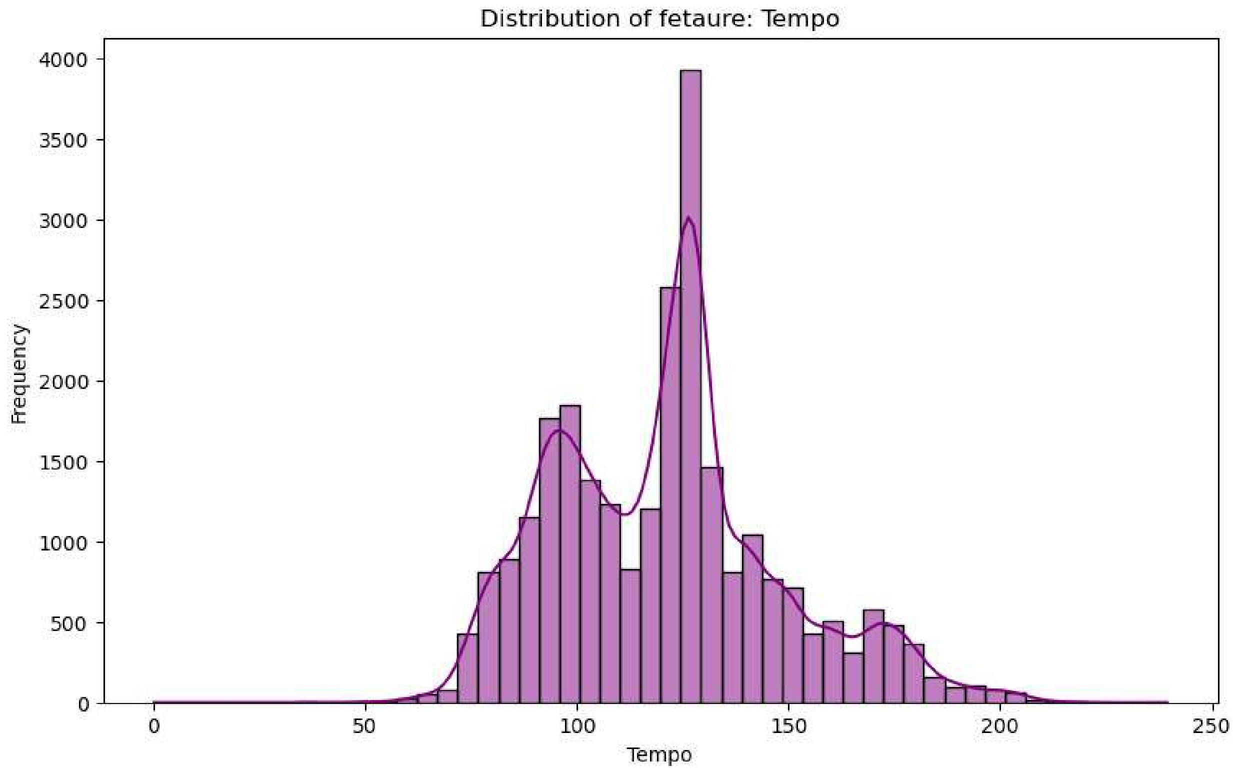
Distribution of fetaure: Acousticness
Frequency
0
1000
2000
3000
4000
5000
6000
7000
0.0
0.2
0.4
0.6
0.8
1.0
Acousticness

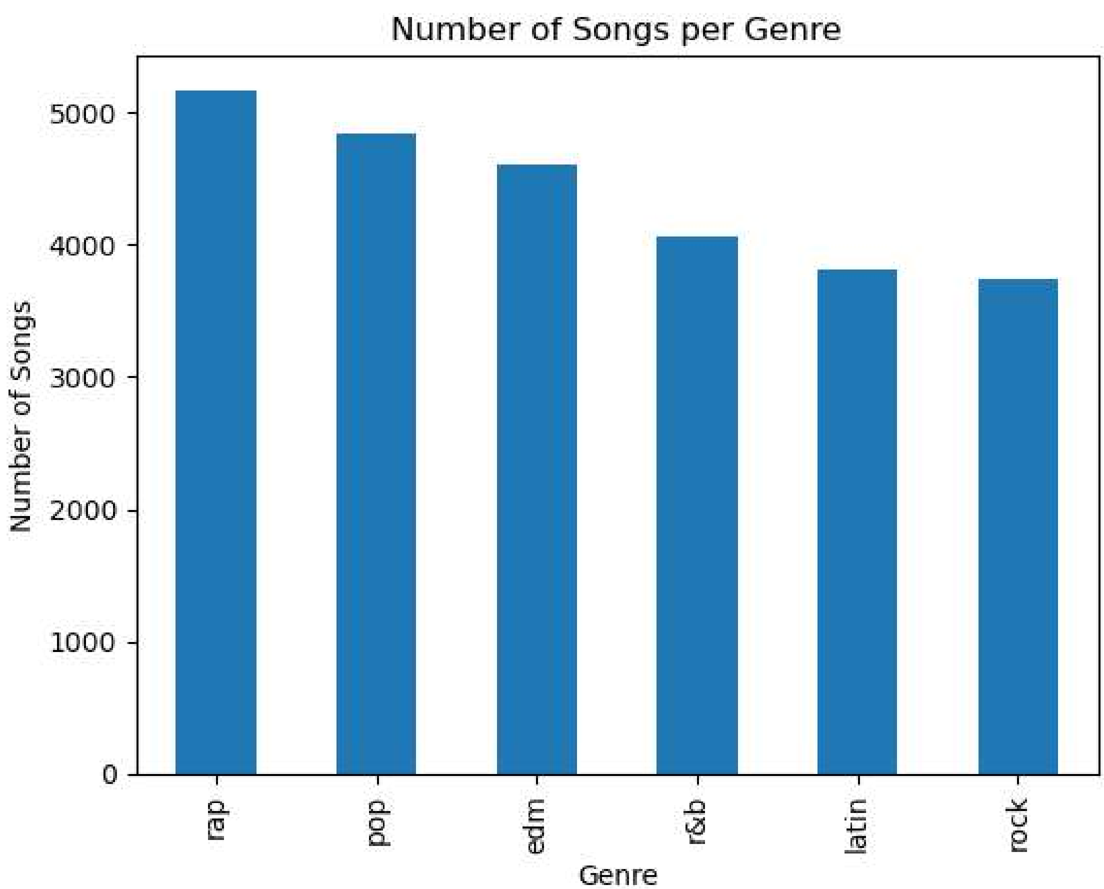


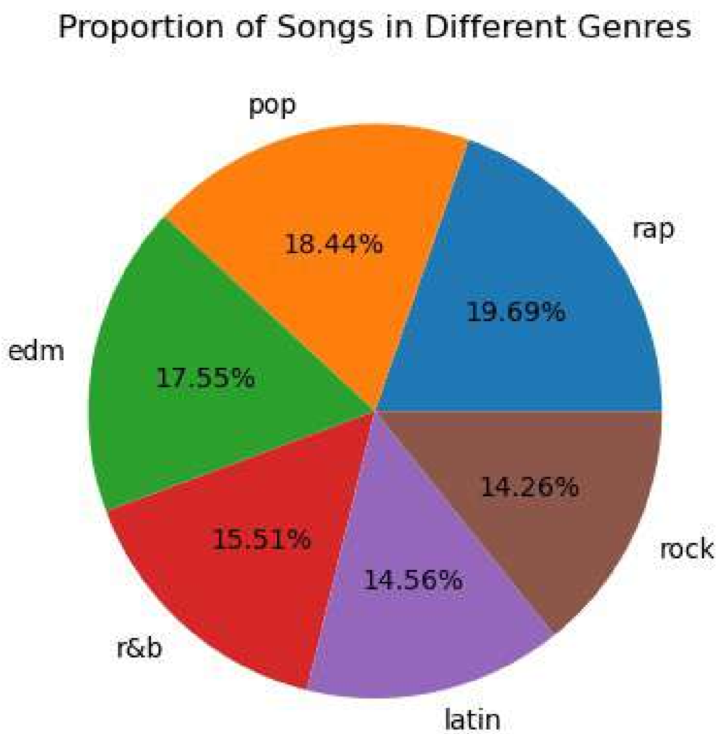
Distribution of fetaure: Instrumentalness
Frequency
0
2500
5000
7500
10000
12500
15000
17500
20000
0.0
0.2
0.4
0.6
0.8
1.0
Instrumentalness


Distribution of fetaure: Liveness
Frequency
Liveness
0
500
1000
1500
2000
2500
3000
3500
4000
0.0
0.2
0.4
0.6
0.8
1.0


Distribution of fetaure: Valence
Frequency
Valence
0
100
200
300
400
500
600
700
800
0.0
0.2
0.4
0.6
0.8
1.0


Distribution of fetaure: Tempo
Frequency
Tempo
0
500
1000
1500
2000
2500
3000
3500
4000
0
50
100
150
200
250

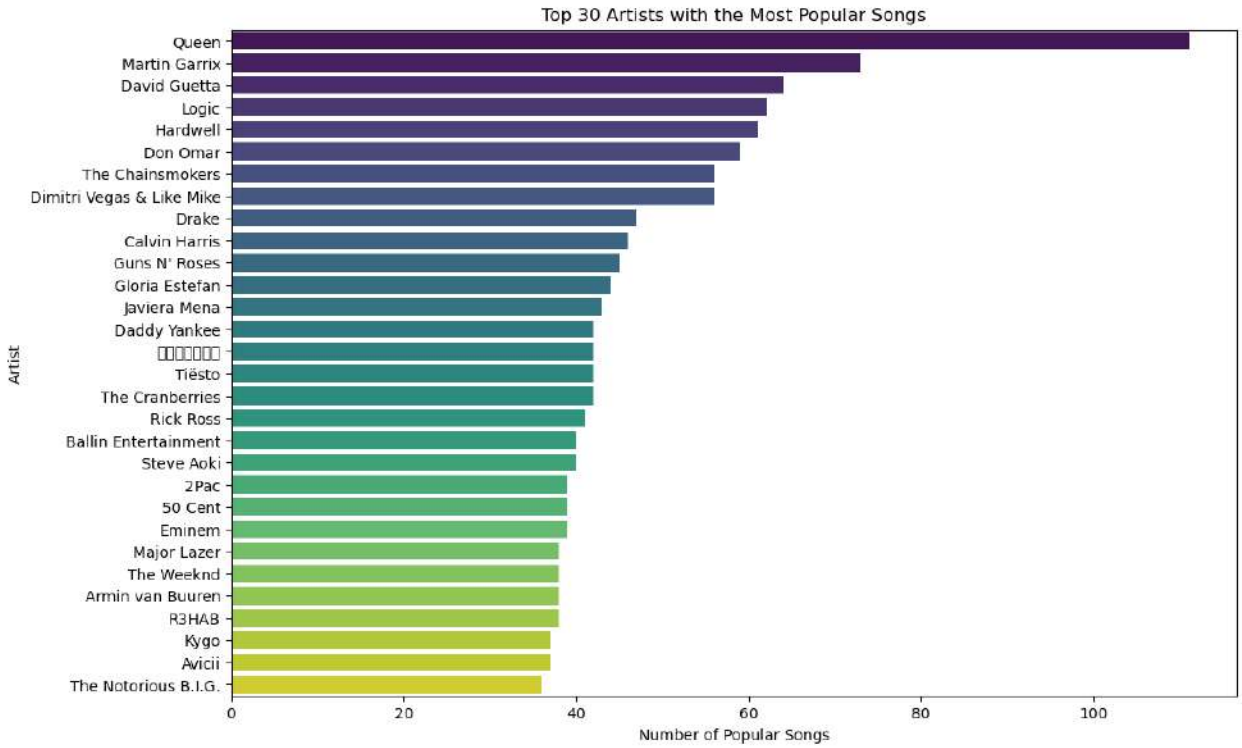

Number of Songs per Genre
Number of Songs
Genre
0
1000
2000
3000
4000
5000
rap
pop
edm
r&b
latin
rock



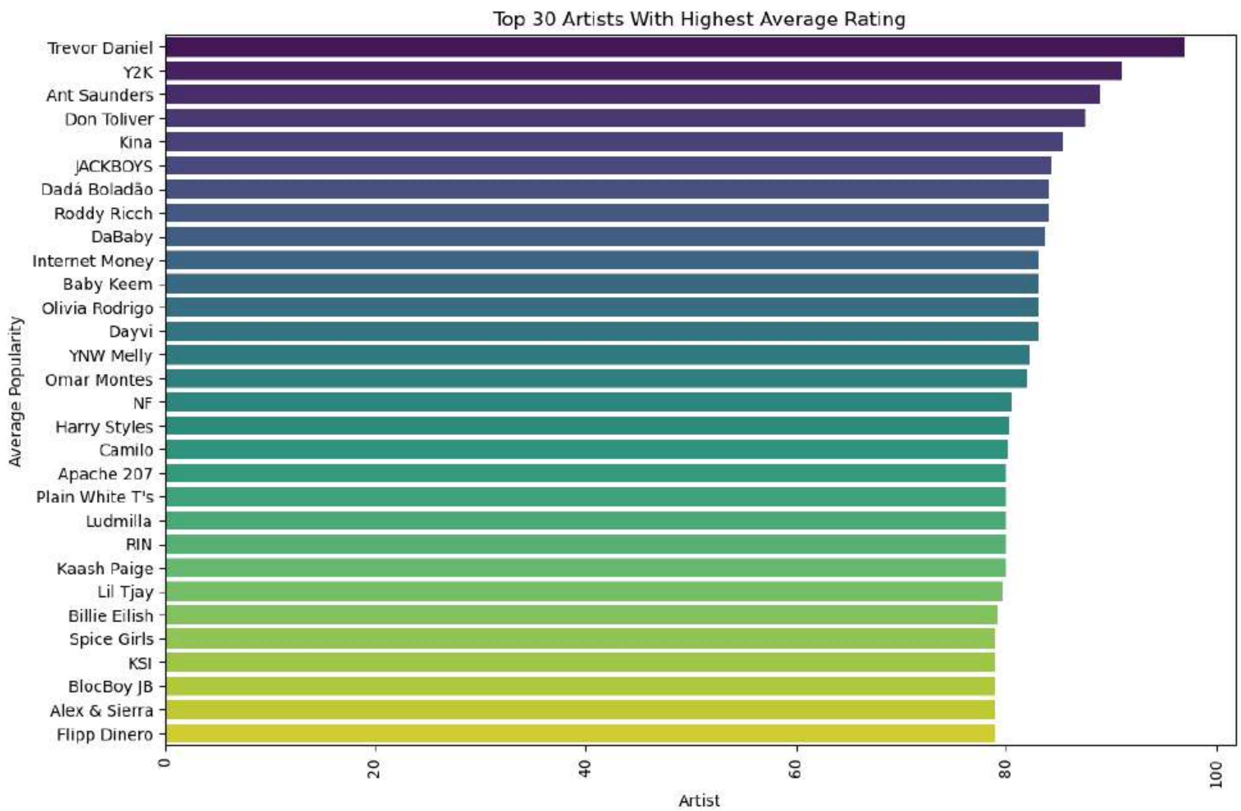

```
# Count the number of popular songs for each artist to visualize track
popularity.
artist_pop = data_songs['track_artist'].value_counts()

# Plot the top 30 artists with the most popular songs for
visualization
top_artists = artist_pop.head(30)
plt.figure(figsize = (12, 8))
sns.barplot(x = top_artists.values, y = top_artists.index, palette =
'viridis')
plt.title('Top 30 Artists with the Most Popular Songs')
plt.xlabel('Number of Popular Songs')
plt.ylabel('Artist')
plt.show()

# Plot list of top 30 artists with the highest average rating
sample_avg = data_songs.groupby(by = 'track_artist')
['track_popularity'].mean().sort_values(ascending=False).head(30).rese
t_index()
plt.figure(figsize = (12,8))
sns.barplot(sample_avg, y = 'track_artist', x = 'track_popularity',
palette = 'viridis')
plt.title('Top 30 Artists With Highest Average Rating')
plt.xlabel('Artist')
plt.ylabel('Average Popularity')
```

```
plt.xticks(rotation = 90)
plt.show()
```

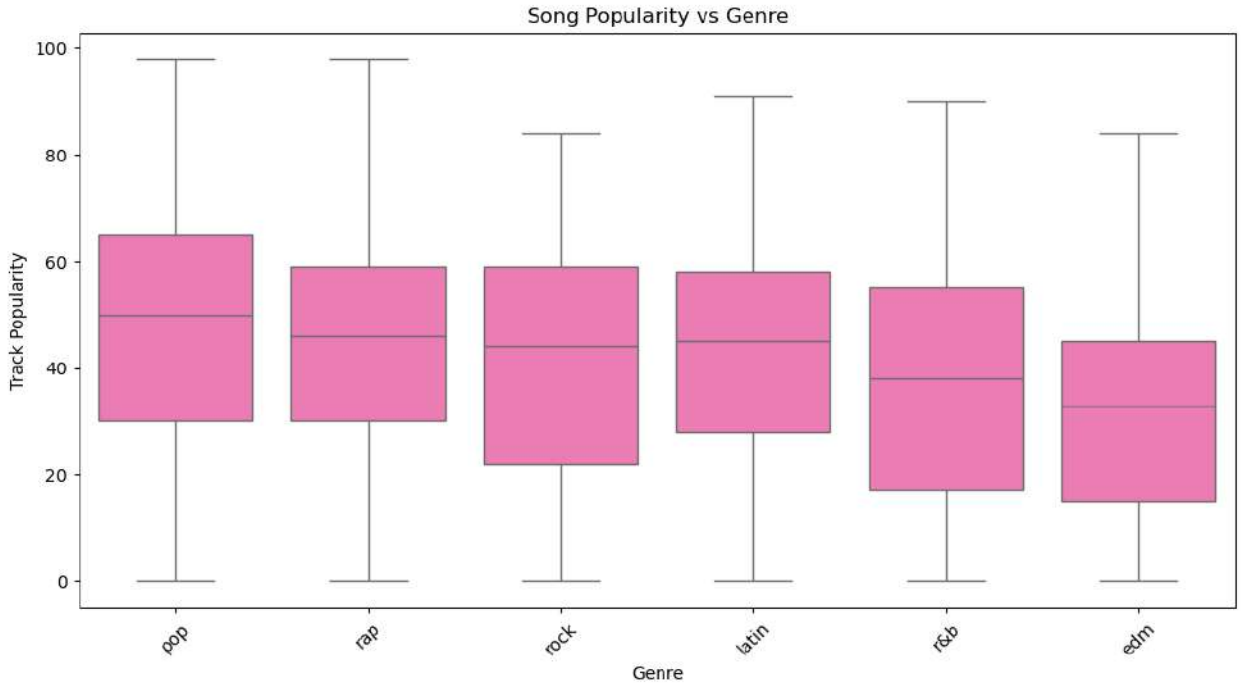
```
C:\Users\lakpr\AppData\Local\Temp\ipykernel_11520\4074497368.py:7:
FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be
removed in v0.14.0. Assign the `y` variable to `hue` and set
`legend=False` for the same effect.

  sns.barplot(x = top_artists.values, y = top_artists.index, palette =
'viridis')
C:\Users\lakpr\anaconda3\Lib\site-packages\IPython\core\
pylabtools.py:170: UserWarning: Glyph 12458 (\N{KATAKANA LETTER O})
missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
C:\Users\lakpr\anaconda3\Lib\site-packages\IPython\core\
pylabtools.py:170: UserWarning: Glyph 12513 (\N{KATAKANA LETTER ME})
missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
C:\Users\lakpr\anaconda3\Lib\site-packages\IPython\core\
pylabtools.py:170: UserWarning: Glyph 12460 (\N{KATAKANA LETTER GA})
missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
C:\Users\lakpr\anaconda3\Lib\site-packages\IPython\core\
pylabtools.py:170: UserWarning: Glyph 12488 (\N{KATAKANA LETTER TO})
missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
C:\Users\lakpr\anaconda3\Lib\site-packages\IPython\core\
pylabtools.py:170: UserWarning: Glyph 12521 (\N{KATAKANA LETTER RA})
missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
C:\Users\lakpr\anaconda3\Lib\site-packages\IPython\core\
pylabtools.py:170: UserWarning: Glyph 12452 (\N{KATAKANA LETTER I})
missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
C:\Users\lakpr\anaconda3\Lib\site-packages\IPython\core\
pylabtools.py:170: UserWarning: Glyph 12502 (\N{KATAKANA LETTER BU})
missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
```




```
C:\Users\lakpr\AppData\Local\Temp\ipykernel_11520\4074497368.py:16:
FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be
removed in v0.14.0. Assign the `y` variable to `hue` and set
`legend=False` for the same effect.

  sns.barplot(sample_avg, y = 'track_artist', x = 'track_popularity',
palette = 'viridis')
```




```
# Boxplot to show the relationship between genre and song popularity
to identify possible outliers and influential points
plt.figure(figsize = (12, 6))
sns.boxplot(x = 'playlist_genre', y = 'track_popularity', data =
data_songs, color = 'hotpink')
plt.title('Relationship between Genre and Song Popularity')
plt.xticks(rotation = 45)
plt.title('Song Popularity vs Genre')
plt.xlabel('Genre')
plt.ylabel('Track Popularity')
plt.show()
```

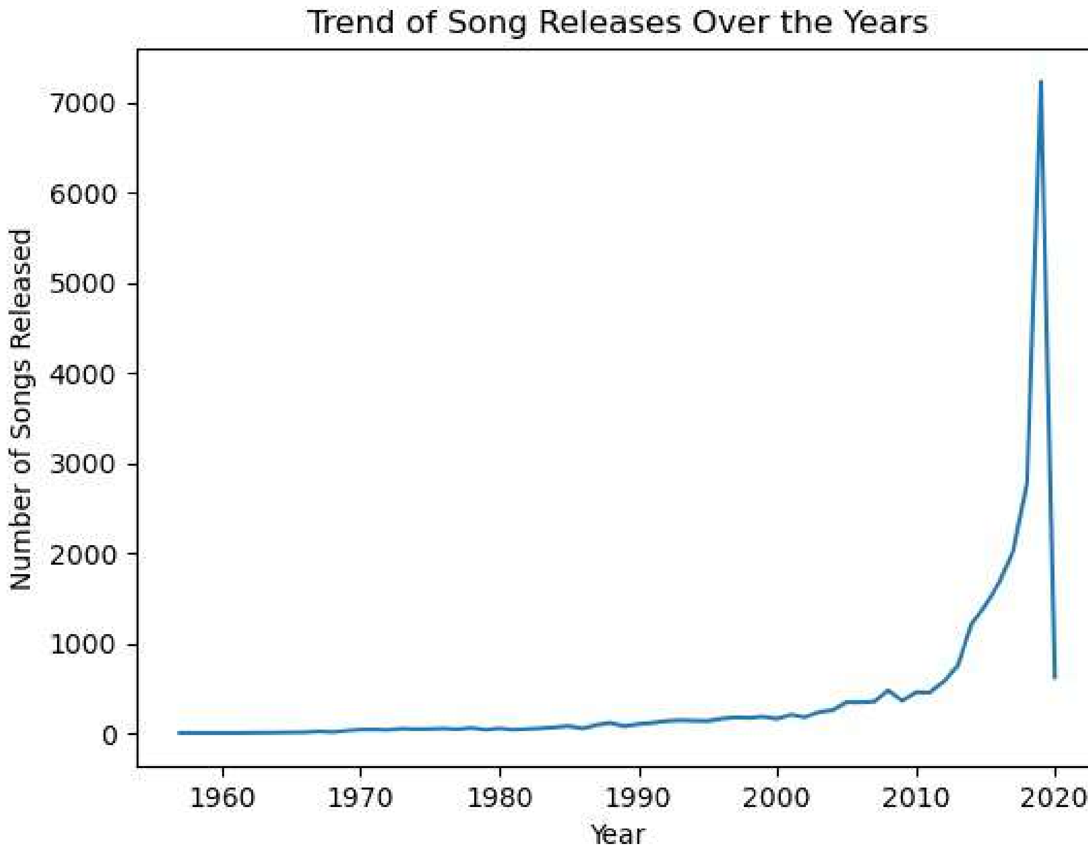



```
# Count the number of songs for each artist to see which artists have
the most songs
artist_song_count = data_songs['track_artist'].value_counts()

# Convert the Series to a DataFrame for better display
artist_song_count_df = artist_song_count.reset_index()
artist_song_count_df.columns = ['Artist', 'Number of Songs']

# Display the new dataframe of values
print(artist_song_count_df)

# Find the artist with the most songs
max_songs_artist = artist_song_count.idxmax()
max_count = artist_song_count.max()

# Find the artist with the least songs
min_songs_artist = artist_song_count.idxmin()
min_count = artist_song_count.min()

# Display the results
print(f"Artist with the most songs: {max_songs_artist} ({max_count}
songs)")
print(f"Artist with the least songs: {min_songs_artist} ({min_count}
song)")

# Notice that there are many artists with one song but only the first
artist with the least number of songs id displayed.
```

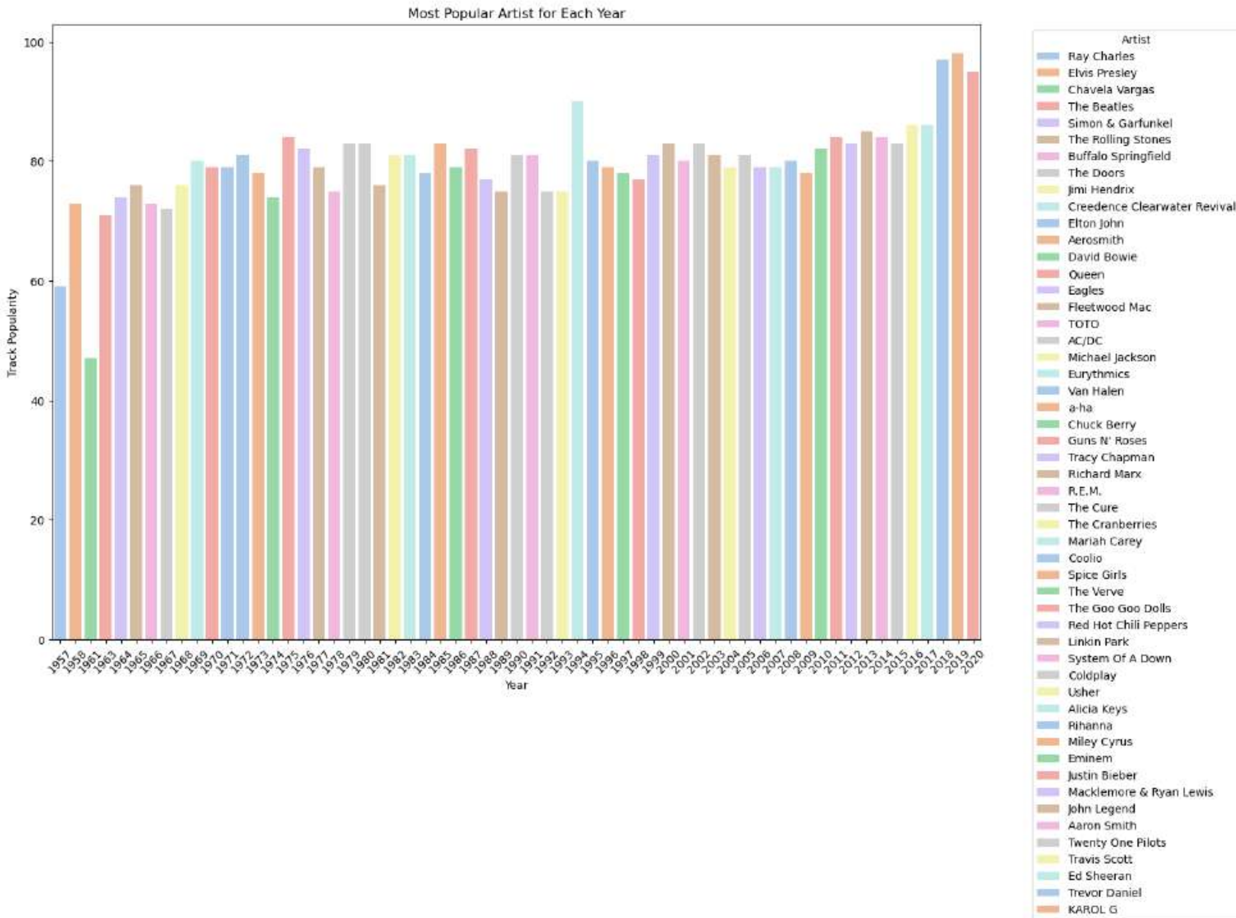
```
                Artist  Number of Songs
0                Queen              111
1       Martin Garrix               73
2        David Guetta               64
3                Logic               62
4             Hardwell               61
...                ...              ...
10687           Irvine                1
10688  Andrew W. Boss                 1
10689   Scott Krokoff                 1
10690   John Belthoff                 1
10691           Mat Zo                1

[10692 rows x 2 columns]
Artist with the most songs: Queen (111 songs)
Artist with the least songs: Kevcody (1 song)

# Convert the 'track_album_release_date' column to datetime format,
handling different date formats for easy plotting and
#interpretation.
data_songs['track_album_release_date'] =
pd.to_datetime(data_songs['track_album_release_date'], errors =
'coerce')

# Drop rows with NaT values if present
data_songs = data_songs.dropna(subset = ['track_album_release_date'])

# Extract the year from the 'track_album_release_date' column and
store year only. We don't consider days and months of the year in this
project.
data_songs['release_year'] =
data_songs['track_album_release_date'].dt.year

# Plot the trend of song releases over the years.
release_trend = data_songs['release_year'].value_counts().sort_index()
release_trend.plot(kind = 'line')
plt.title('Trend of Song Releases Over the Years')
plt.xlabel('Year')
plt.ylabel('Number of Songs Released')
plt.show()

# Find the most popular artist for each year
mostpop_art_year = data_songs.loc[data_songs.groupby('release_year')
['track_popularity'].idxmax()]

# Plot the most popular artist for each year
plt.figure(figsize = (15, 10))
sns.barplot(x = 'release_year', y = 'track_popularity', hue =
'track_artist', data = mostpop_art_year, palette = 'pastel')
plt.title('Most Popular Artist for Each Year')
```

```
plt.xlabel('Year')
plt.ylabel('Track Popularity')
plt.xticks(rotation = 45)
plt.legend(title = 'Artist', bbox_to_anchor = (1.05, 1), loc = 'upper
left')
plt.show()

# Calculate the average number of songs per artist
avg_songs_art = artist_song_count.mean()

# Plot the average number of songs per artist = 3
plt.figure(figsize = (8, 6))
sns.barplot(x = ['Average Number of Songs per Artist'], y=
[avg_songs_art], palette='pastel')
plt.title('Average Number of Songs per Artist')
plt.ylabel('Average Number of Songs')
plt.show()
```

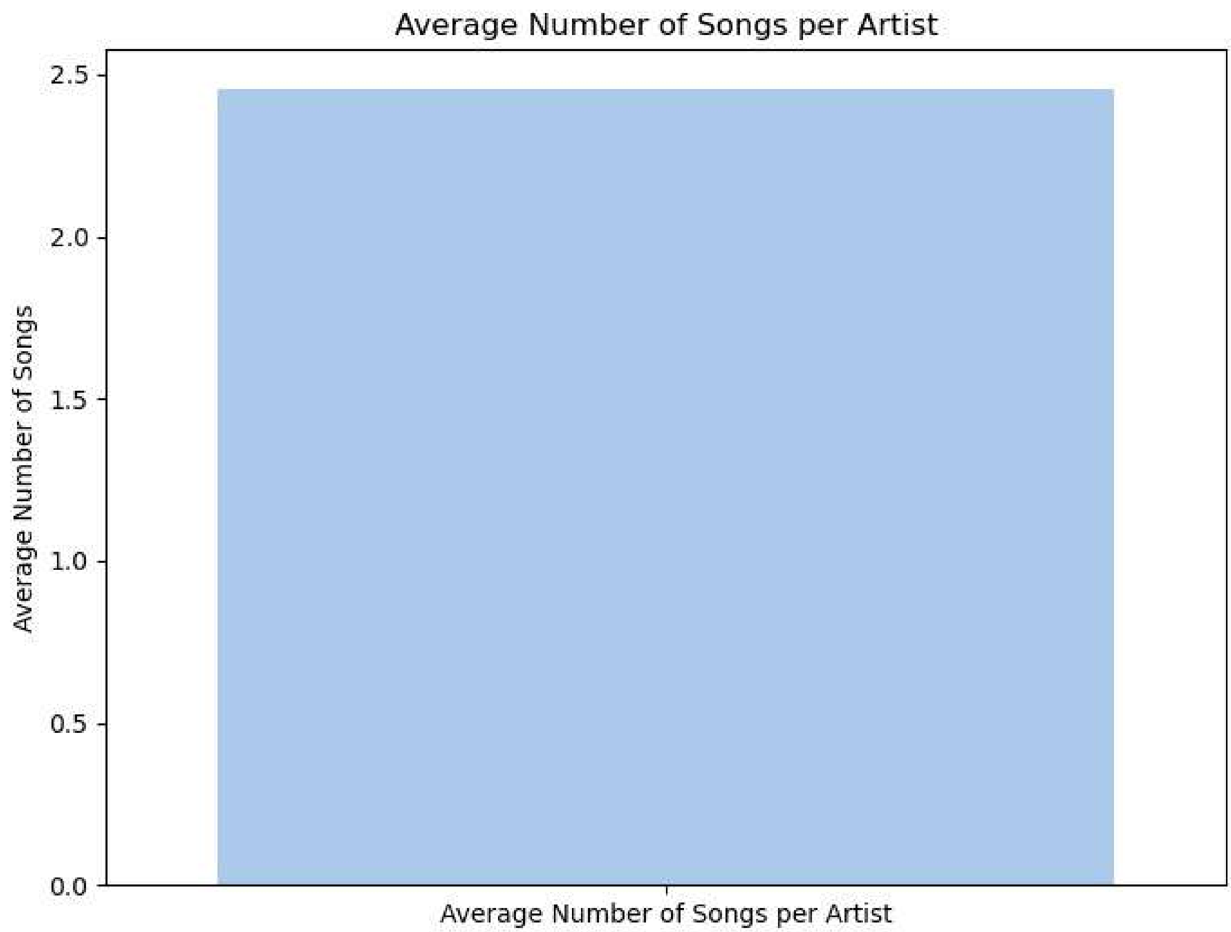
```
C:\Users\lakpr\AppData\Local\Temp\ipykernel_11520\126333889.py:9:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
  data_songs['release_year'] =
data_songs['track_album_release_date'].dt.year
```



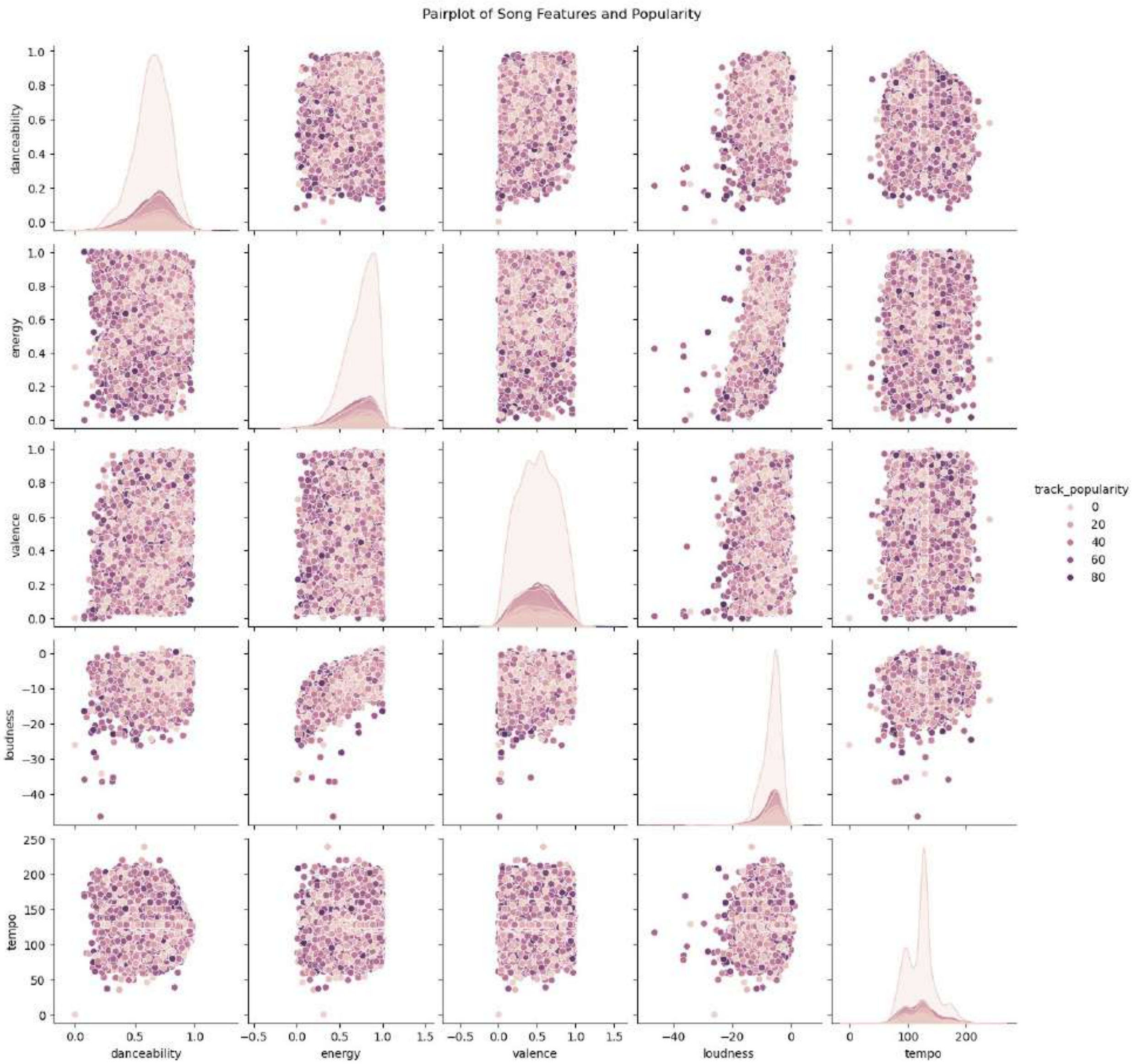
Trend of Song Releases Over the Years
Number of Songs Released
0
1000
2000
3000
4000
5000
6000
7000
1960
1970
1980
1990
2000
2010
2020
Year




```
C:\Users\lakpr\AppData\Local\Temp\ipykernel_11520\126333889.py:37:
FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be
removed in v0.14.0. Assign the `x` variable to `hue` and set
`legend=False` for the same effect.

  sns.barplot(x = ['Average Number of Songs per Artist'], y=
[avg_songs_art], palette='pastel')
```

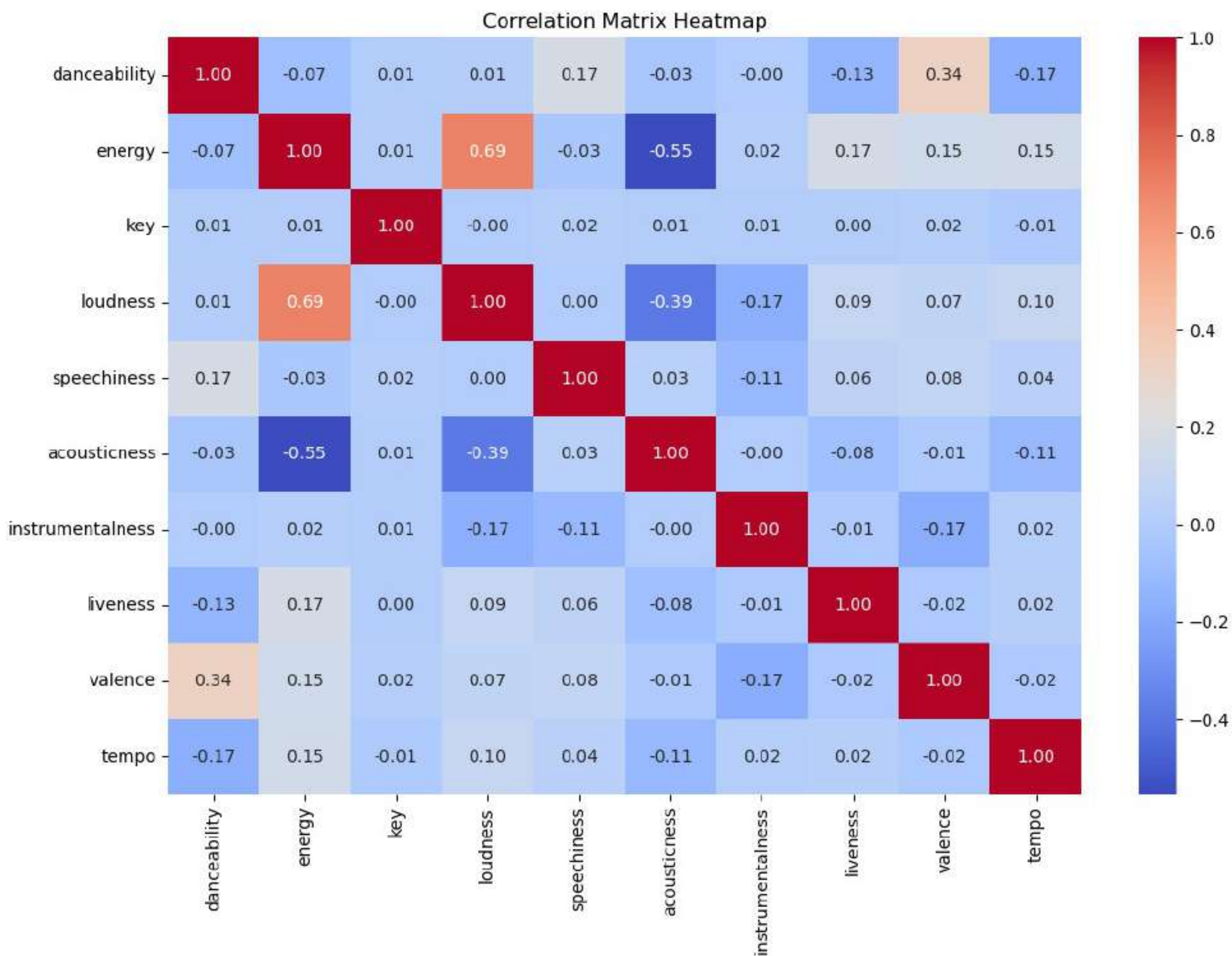



```
# Ensure plots are displayed inline
%matplotlib inline

# Pairplot to visualize the relationships between features to identify
multicollinearity between features
sns.pairplot(data_songs[['danceability', 'energy', 'valence',
'loudness', 'tempo', 'track_popularity']], hue='track_popularity')
plt.suptitle('Pairplot of Song Features and Popularity', y = 1.02)
plt.show()

# Pairplot is not easy to interpret so let's look at correlation
matrix and Variance Inflation Factor(VIF)
```




```
# Drop variables DURATION_MS (time) and mode because time of a track
technically does not contribute to popularity (longer or shorter
tracks are
#both popular and not as popular, so no relationship) and mode is
categorical. We stick to only numerical features in this project.

# Select the specified numerical columns
select_col_feat = ['danceability', 'energy', 'key', 'loudness',
'speechiness', 'acousticness', 'instrumentalness', 'liveness',
'valence', 'tempo']
select_data = data_songs[select_col_feat]

# Calculate the correlation matrix and plot it using a heatmap
corr_mat = select_data.corr()
plt.figure(figsize=(12, 8))
```

```
sns.heatmap(corr_mat, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix Heatmap')
plt.show()

# Calculate VIF for each feature to determine multicollinearity.
vif_data = pd.DataFrame()
vif_data['Feature'] = select_data.columns
vif_data['VIF'] = [variance_inflation_factor(select_data.values, i)
for i in range(len(select_data.columns))]
print(vif_data)

# Since VIF is >=10 for some variables (this implies strong
multicollinearity and will produce contardictory results)
#Let's center and scale the data and see if we can achieve VIF < 10

# Center and scale the data
scaler = StandardScaler()
scaled_data_songs_final = scaler.fit_transform(select_data)

# Convert scaled data back to DataFrame for data analysis
scaled_data_songs_final_df = pd.DataFrame(scaled_data_songs_final,
columns = select_col_feat)

# Calculate VIF for each feature in scaled data to check if
standardizing the features reduces multicollinearity
vif_data_scaled = pd.DataFrame()
vif_data_scaled['Feature'] = scaled_data_songs_final_df.columns
vif_data_scaled['VIF'] =
[variance_inflation_factor(scaled_data_songs_final_df.values, i) for i
in range(len(scaled_data_songs_final_df.columns))]

print(vif_data_scaled)

# VIF HAS REDUCED SIGNIFICANTLY and are all <10.
# So only this scaled data will be used from now on in this analysis
and for making predictions and model fitting below.
```



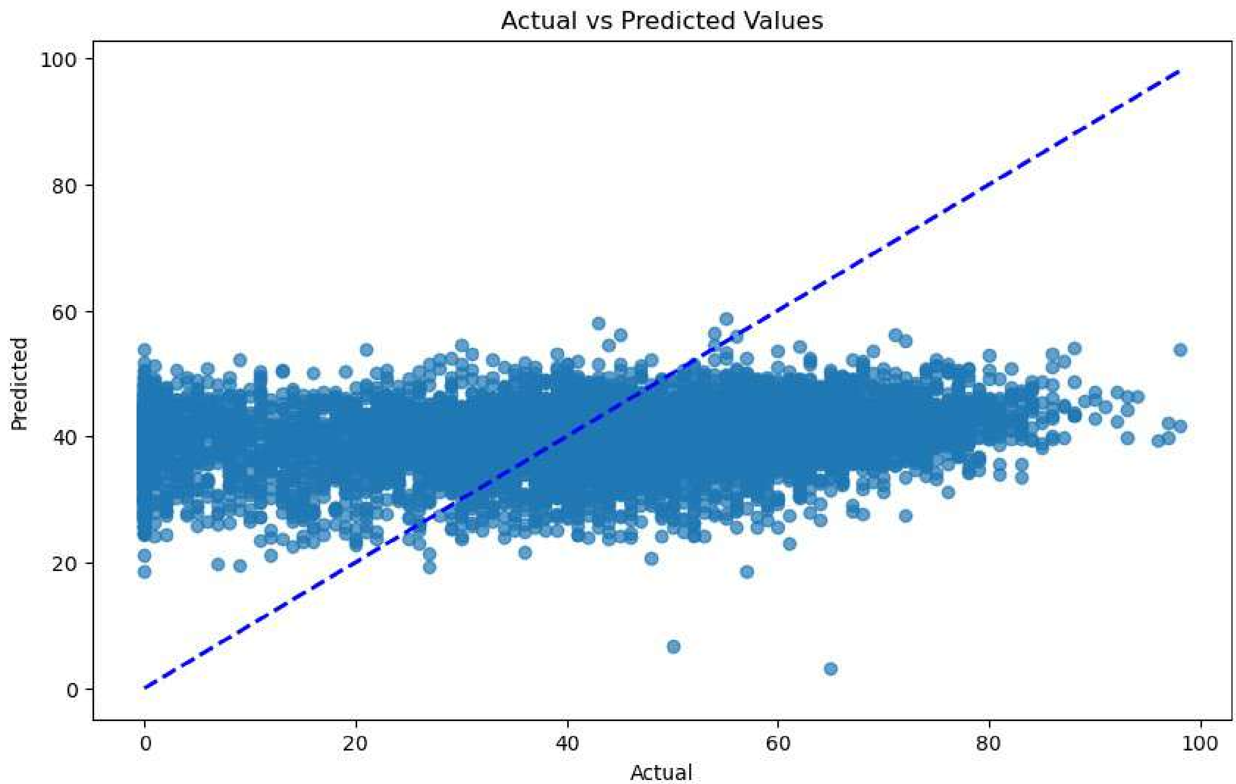

```
             Feature        VIF
0       danceability  18.627477
1             energy  19.482459
2                key   3.171256
3           loudness   7.561613
4        speechiness   2.248755
5       acousticness   2.104711
6   instrumentalness   1.291220
7           liveness   2.615568
8            valence   6.861232
9              tempo  18.228049
             Feature       VIF
0       danceability  1.286881
1             energy  2.720484
2                key  1.001452
3           loudness  2.114826
4        speechiness  1.061658
5       acousticness  1.488041
6   instrumentalness  1.143523
7           liveness  1.055531
```

```
8           valence  1.258983
9             tempo  1.061393

#----------------------------------------------MODEL FITTING USING
DIFFERENT MACHINE LEARNING
APPROACHES----------------------------------------------

#--------------------------------------------------LINEAR
REGRESSION-----------------------------------------------------------------
---

# Select the label. Here, as mentioned above we want to predict 'track
popularity'
y = data_songs['track_popularity']

# Split the data into training and testing sets using
'train_test_split' like we have in Demos and all previois assignments
X_train, X_test, y_train, y_test =
train_test_split(scaled_data_songs_final_df, y, test_size = 0.3,
random_state = 42)

# Initialize the Linear Regression model
forw_select_model = LinearRegression()

# We perform Forward Stepwise Selection using
SequentialFeatureSelector to see what variables predict
track_popularity best.
sfs_var_sel = SFS(forw_select_model,
          k_features='best',  # Automatically determine the optimal
number of features for this model using forward stepwise selection
          forward=True,
          floating=False,
          scoring='r2',
          cv=5)  # 5-fold cross-validation is very commonly used in ML


# Fit the feature selection on the training data
sfs_var_sel = sfs_var_sel.fit(X_train, y_train)

# Get the selected feature indices and display them. Notice that 'key'
is dropped because it is an insignificant feature.
selected_features = list(sfs_var_sel.k_feature_names_)
print(f"Selected Features: {selected_features}")

# Subset the training and testing data to the selected features to fit
a multiple linear regression model
X_train_selected = X_train[selected_features]
X_test_selected = X_test[selected_features]

# Train the linear regression model on the selected features
LRmodel = LinearRegression()
```

```python
LRmodel.fit(X_train_selected, y_train)

# Make predictions on the training and testing sets obtained by
forward selection
y_train_pred = LRmodel.predict(X_train_selected)
y_test_pred = LRmodel.predict(X_test_selected)

# Evaluate the model to see model performance
train_mse = mean_squared_error(y_train, y_train_pred)
print(f"Training MSE: {train_mse:.4f}")

test_mse = mean_squared_error(y_test, y_test_pred)
print(f"Test MSE: {test_mse:.4f}")

test_r2 = r2_score(y_test, y_test_pred)
print(f"Test R^2: {test_r2:.4f}")

train_r2 = r2_score(y_train, y_train_pred)
print(f"Train R^2: {train_r2:.4f}")

# Print model coefficients and model slope estimates for the selected
features that make up the linear regression model
beta_0 = LRmodel.intercept_
print(f"Intercept (beta_0): {beta_0}")

coeff_lr = pd.DataFrame({'Feature': selected_features, 'Coefficient':
LRmodel.coef_})
print(coeff_lr)

# Plot actual vs predicted values for the test set to visualize the
model performance
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_test_pred, alpha=0.7)
plt.plot([y.min(), y.max()], [y.min(), y.max()], '--b', linewidth=2)
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Actual vs Predicted Values')
plt.show()
```

```
Selected Features: ['danceability', 'energy', 'loudness',
'speechiness', 'acousticness', 'instrumentalness', 'liveness',
'valence', 'tempo']
Training MSE: 515.6684
Test MSE: 498.4739
Test R^2: 0.0560
Train R^2: 0.0502
Intercept (beta_0): 39.757092756560056
            Feature  Coefficient
0      danceability     0.812262
1            energy    -4.594179
```

```
2          loudness     4.080772
3       speechiness    -0.498265
4      acousticness     1.274059
5  instrumentalness    -2.257600
6          liveness    -0.663832
7           valence     0.666266
8             tempo     0.767656
```

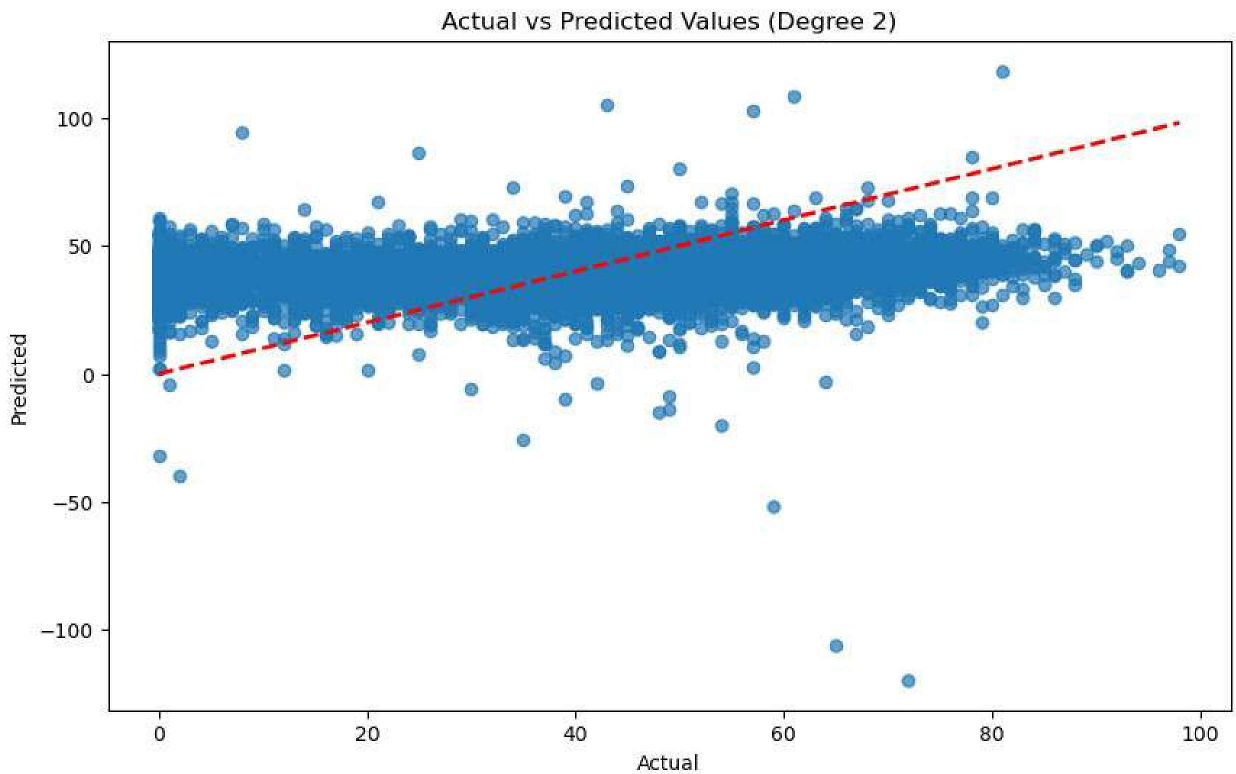



```
# Function to evaluate models with polynomial features of varying
degrees to improve model performance
def eval_poly_model(degree):
    print(f"\nEvaluating model with Polynomial Degree: {degree}")

    # Create a pipeline with polynomial feature generation and linear
regression
    pipeline = Pipeline([
        ('poly_features', PolynomialFeatures(degree=degree,
include_bias=False)),
        ('linear_regression', LinearRegression())
    ])

    # Perform forward stepwise selection using
SequentialFeatureSelector on polynomial features just like in multiple
linear regression above
    sfs = SFS(pipeline['linear_regression'],
```

```
              k_features='best',  # Automatically determine the
optimal number of features
              forward=True,
              floating=False,
              scoring='r2',
              cv=5)  # 5-fold cross-validation

    # Fit the selector on the polynomial features
    sfs = sfs.fit(X_train, y_train)

    # Get selected features for the polynomial degree
    selected_features = list(sfs.k_feature_names_)
    print(f"Selected Features for Degree {degree}:
{selected_features}")

    # Fit the pipeline with selected features
    X_train_poly = PolynomialFeatures(degree=degree,
include_bias=False).fit_transform(X_train[selected_features])
    X_test_poly = PolynomialFeatures(degree=degree,
include_bias=False).fit_transform(X_test[selected_features])

    # Train the linear regression model
    pipeline.fit(X_train_poly, y_train)

    # Make predictions
    y_train_pred = pipeline.predict(X_train_poly)
    y_test_pred = pipeline.predict(X_test_poly)

    # Evaluate model performance using same criterion as in linear
regression
    train_mse = mean_squared_error(y_train, y_train_pred)
    print(f"Training MSE: {train_mse:.4f}")
    test_mse = mean_squared_error(y_test, y_test_pred)
    print(f"Test MSE: {test_mse:.4f}")
    r2 = r2_score(y_test, y_test_pred)
    print(f"R^2: {r2:.4f}")

    # Plot actual vs predicted values for the test set
    plt.figure(figsize=(10, 6))
    plt.scatter(y_test, y_test_pred, alpha=0.7)
    plt.plot([y.min(), y.max()], [y.min(), y.max()], '--r',
linewidth=2)
    plt.xlabel('Actual')
    plt.ylabel('Predicted')
    plt.title(f'Actual vs Predicted Values (Degree {degree})')
    plt.show()

# Iterate over polynomial degrees 2, 3, 4 and 5
```

```
for degree in [2, 3, 4, 5]:
    eval_poly_model(degree)
```

```
Evaluating model with Polynomial Degree: 2
Selected Features for Degree 2: ['danceability', 'energy', 'loudness', 
'speechiness', 'acousticness', 'instrumentalness', 'liveness', 
'valence', 'tempo']
Training MSE: 477.0132
Test MSE: 515.8839
R^2: 0.0230
```




```
Evaluating model with Polynomial Degree: 3
Selected Features for Degree 3: ['danceability', 'energy', 'loudness', 
'speechiness', 'acousticness', 'instrumentalness', 'liveness', 
'valence', 'tempo']

---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
Cell In[119], line 56
     54 # Iterate over polynomial degrees 2, 3, 4 and 5
     55 for degree in [2, 3, 4, 5]:
---> 56     eval_poly_model(degree)
```

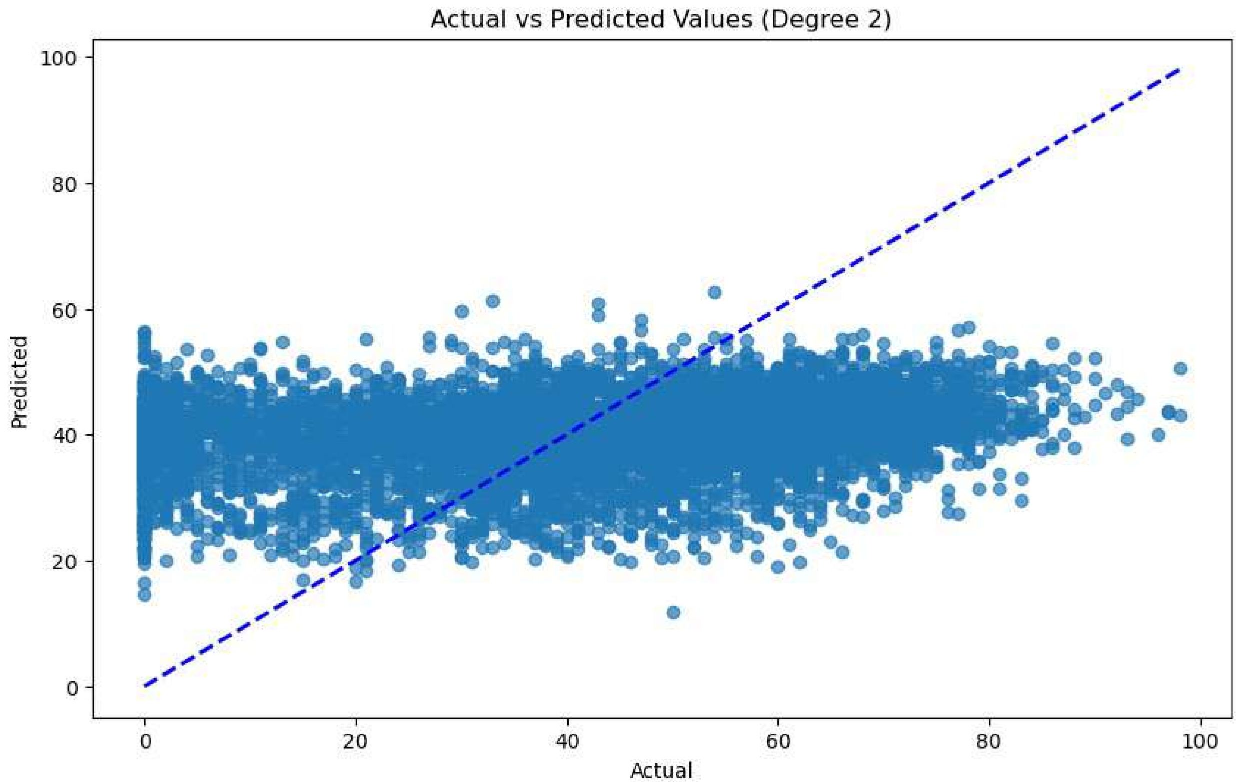
```
Cell In[119], line 31, in eval_poly_model(degree)
     28 X_test_poly = PolynomialFeatures(degree=degree,
include_bias=False).fit_transform(X_test[selected_features])
     30 # Train the linear regression model
---> 31 pipeline.fit(X_train_poly, y_train)
     33 # Make predictions
     34 y_train_pred = pipeline.predict(X_train_poly)

File ~\anaconda3\Lib\site-packages\sklearn\base.py:1473, in
_fit_context.<locals>.decorator.<locals>.wrapper(estimator, *args,
**kwargs)
   1466     estimator._validate_params()
   1468 with config_context(
   1469     skip_parameter_validation=(
   1470         prefer_skip_nested_validation or
global_skip_validation
   1471     )
   1472 ):
-> 1473     return fit_method(estimator, *args, **kwargs)

File ~\anaconda3\Lib\site-packages\sklearn\pipeline.py:469, in
Pipeline.fit(self, X, y, **params)
    426 """Fit the model.
    427
    428 Fit all the transformers one after the other and sequentially
transform the
   (...)
    466     Pipeline with fitted steps.
    467 """
    468 routed_params = self._check_method_params(method="fit",
props=params)
--> 469 Xt = self._fit(X, y, routed_params)
    470 with _print_elapsed_time("Pipeline",
self._log_message(len(self.steps) - 1)):
    471     if self._final_estimator != "passthrough":

File ~\anaconda3\Lib\site-packages\sklearn\pipeline.py:406, in
Pipeline._fit(self, X, y, routed_params)
    404     cloned_transformer = clone(transformer)
    405 # Fit or load from cache the current transformer
--> 406 X, fitted_transformer = fit_transform_one_cached(
    407     cloned_transformer,
    408     X,
    409     y,
    410     None,
    411     message_clsname="Pipeline",
    412     message=self._log_message(step_idx),
    413     params=routed_params[name],
    414 )
```

```
    415 # Replace the transformer of the step with the fitted
    416 # transformer. This is necessary when loading the transformer
    417 # from the cache.
    418 self.steps[step_idx] = (name, fitted_transformer)

File ~\anaconda3\Lib\site-packages\joblib\memory.py:312, in
NotMemorizedFunc.__call__(self, *args, **kwargs)
    311 def __call__(self, *args, **kwargs):
--> 312     return self.func(*args, **kwargs)

File ~\anaconda3\Lib\site-packages\sklearn\pipeline.py:1310, in
_fit_transform_one(transformer, X, y, weight, message_clsname,
message, params)
   1308 with _print_elapsed_time(message_clsname, message):
   1309     if hasattr(transformer, "fit_transform"):
-> 1310         res = transformer.fit_transform(X, y,
**params.get("fit_transform", {}))
   1311     else:
   1312         res = transformer.fit(X, y, **params.get("fit",
{})).transform(
   1313             X, **params.get("transform", {})
   1314         )

File ~\anaconda3\Lib\site-packages\sklearn\utils\_set_output.py:313,
in _wrap_method_output.<locals>.wrapped(self, X, *args, **kwargs)
    311 @wraps(f)
    312 def wrapped(self, X, *args, **kwargs):
--> 313     data_to_wrap = f(self, X, *args, **kwargs)
    314     if isinstance(data_to_wrap, tuple):
    315         # only wrap the first output for cross decomposition
    316         return_tuple = (
    317             _wrap_data_with_container(method, data_to_wrap[0],
X, self),
    318             *data_to_wrap[1:],
    319         )

File ~\anaconda3\Lib\site-packages\sklearn\base.py:1101, in
TransformerMixin.fit_transform(self, X, y, **fit_params)
   1098     return self.fit(X, **fit_params).transform(X)
   1099 else:
   1100     # fit method of arity 2 (supervised transformation)
-> 1101     return self.fit(X, y, **fit_params).transform(X)

File ~\anaconda3\Lib\site-packages\sklearn\utils\_set_output.py:313,
in _wrap_method_output.<locals>.wrapped(self, X, *args, **kwargs)
    311 @wraps(f)
    312 def wrapped(self, X, *args, **kwargs):
--> 313     data_to_wrap = f(self, X, *args, **kwargs)
    314     if isinstance(data_to_wrap, tuple):
    315         # only wrap the first output for cross decomposition
```

```
    316         return_tuple = (
    317             _wrap_data_with_container(method, data_to_wrap[0], 
X, self),
    318             *data_to_wrap[1:],
    319         )

File ~\anaconda3\Lib\site-packages\sklearn\preprocessing\
_polynomial.py:508, in PolynomialFeatures.transform(self, X)
    504     XP = sparse.hstack(columns, dtype=X.dtype).tocsc()
    505 else:
    506     # Do as if _min_degree = 0 and cut down array after the
    507     # computation, i.e. use _n_out_full instead of 
n_output_features_.
--> 508     XP = np.empty(
    509         shape=(n_samples, self._n_out_full), dtype=X.dtype, 
order=self.order
    510     )
    512     # What follows is a faster implementation of:
    513     # for i, comb in enumerate(combinations):
    514     #     XP[:, i] = X[:, comb].prod(1)
   (...)
    524 
    525     # degree 0 term
    526     if self.include_bias:

MemoryError: Unable to allocate 232. GiB for an array with shape 
(17320, 1798939) and data type float64

# Stick to generating polynomial features for degree 2 only since 
degree > 2 is leading to exponential growth and taking up lots of 
memory.

poly2 = PolynomialFeatures(degree=2, include_bias=False)
X_train_poly = poly2.fit_transform(X_train)
X_test_poly = poly2.transform(X_test)

# Train the linear regression model
LRpolymodel = LinearRegression()
LRpolymodel.fit(X_train_poly, y_train)

# make predictions on the labels
y_train_pred = LRpolymodel.predict(X_train_poly)
y_test_pred = LRpolymodel.predict(X_test_poly)

# Evaluation metrics
train_mse = mean_squared_error(y_train, y_train_pred)
print(f"Training MSE: {train_mse:.4f}")
test_mse = mean_squared_error(y_test, y_test_pred)
print(f"Test MSE: {test_mse:.4f}")
r2_test = r2_score(y_test, y_test_pred)
```

```
print(f"Test R^2: {r2_test:.4f}")
r2_train = r2_score(y_train, y_train_pred)
print(f"Train R^2: {r2_train:.4f}")

# Intercept and coefficient estimates of the polynomial regression
model
beta_0 = LRpolymodel.intercept_
print(f"Intercept (beta_0): {beta_0}")
coefficients = LRpolymodel.coef_
features = poly2.get_feature_names_out(X_train.columns)
coeff_df = pd.DataFrame({'Feature': features, 'Coefficient':
coefficients})
print(coeff_df)

# Plot actual vs predicted values for the test set for visualizing
performance
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_test_pred, alpha=0.7)
plt.plot([y.min(), y.max()], [y.min(), y.max()], '--b', linewidth=2)
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title(f'Actual vs Predicted Values (Degree {2})')
plt.show()
```

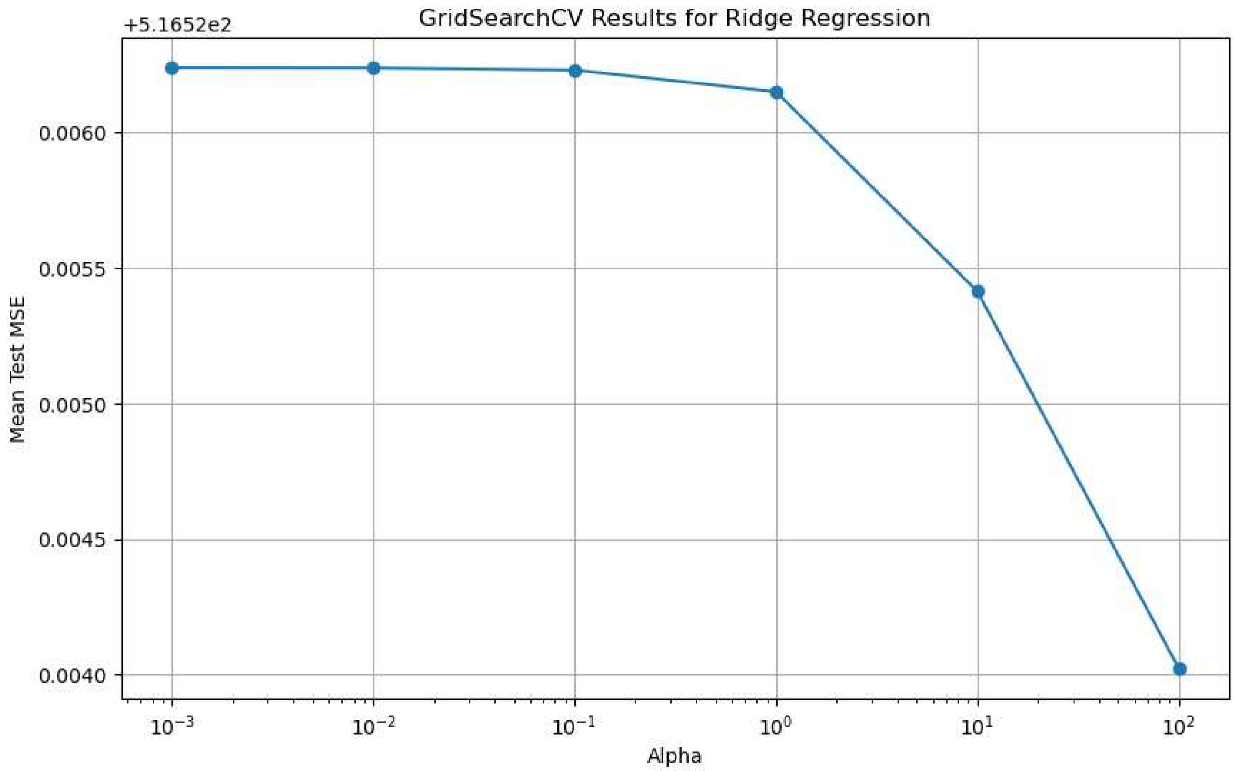
```
Training MSE: 504.1231
Test MSE: 491.9518
Test R^2: 0.0683
Train R^2: 0.0715
Intercept (beta_0): 39.63815146880444
             Feature  Coefficient
0       danceability     1.213027
1             energy    -4.315998
2                key    -0.100787
3           loudness     4.330121
4        speechiness    -0.856083
..               ...          ...
60  liveness valence     0.055730
61    liveness tempo    -0.117265
62         valence^2    -0.693602
63     valence tempo     0.000106
64           tempo^2     0.462841

[65 rows x 2 columns]
```




```
# As seen in the linear regression models with and without polynomial
features, we see that the test R^2 was very low.
#Only about 5% - 7% of variation is being explained by both these
models respectively. So, we can conclude that linear regression and
its various
#types are not good models to predict track popularity.

#-----------------------------------------------RIDGE REGRESSION WITH
HYPERPARAMETER TUNING-------------------------------------------------

# Convert scaled data back to DataFrame
scaled_data_songs_final_df = pd.DataFrame(scaled_data_songs_final,
columns=select_col_feat)

# Select the label
#y = data_songs['track_popularity']

# Split the data into training and testing sets
#X_train, X_test, y_train, y_test =
train_test_split(scaled_data_songs_final_df, y, test_size=0.3,
random_state=42)

# Create a Ridge regression model
ridgeReg = Ridge()

# Define hyperparameters for tuning
```

```
param_grid = {'alpha': [0.001, 0.01, 0.1, 1, 10, 100]}

# Perform GridSearchCV to find the best hyperparameters
grid_search_lambda = GridSearchCV(ridgeReg, param_grid, cv=5,
scoring='neg_mean_squared_error')
grid_search_lambda.fit(X_train, y_train)

# Get the best model from grid search
best_lambda_tun_par = grid_search_lambda.best_estimator_

# Make predictions on the training set and test set
y_train_pred = best_lambda_tun_par.predict(X_train)
y_test_pred = best_lambda_tun_par.predict(X_test)

# Calculate evaluation metrics for training set
train_mse = mean_squared_error(y_train, y_train_pred)
train_rmse = np.sqrt(train_mse)
print(f"Training MSE: {train_mse:.4f}")
print(f"Training RMSE: {train_rmse:.4f}")

# Calculate evaluation metrics for test set
test_mse = mean_squared_error(y_test, y_test_pred)
test_rmse = np.sqrt(test_mse)
print(f"Test MSE: {test_mse:.4f}")
print(f"Test RMSE: {test_rmse:.4f}")

# Calculate R^2 and adjusted R^2 for test set
train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)
print(f"Training R^2: {train_r2 * 10 :.4f}")
print(f"Testing R^2: {test_r2:.4f}")

# Print model coefficients
coeff_ridge = pd.DataFrame({'Feature': select_col_feat, 'Coefficient':
best_lambda_tun_par.coef_})
print(coeff_ridge)

# Extract the results from GridSearchCV
results = pd.DataFrame(grid_search_lambda.cv_results_)

# Plot the mean test scores for each alpha value
plt.figure(figsize=(10, 6))
plt.plot(results['param_alpha'], -results['mean_test_score'],
marker='o')
plt.xlabel('Alpha')
plt.ylabel('Mean Test MSE')
plt.title('GridSearchCV Results for Ridge Regression')
plt.xscale('log')
plt.grid(True)
plt.show()
```

```
Training MSE: 515.6700
Training RMSE: 22.7084
Test MSE: 498.4962
Test RMSE: 22.3270
Training R^2: 0.5019
Testing R^2: 0.0559
             Feature  Coefficient
0       danceability     0.819008
1             energy    -4.497691
2                key    -0.053079
3           loudness     3.997916
4        speechiness    -0.491948
5       acousticness     1.287285
6   instrumentalness    -2.261747
7           liveness    -0.667259
8            valence     0.651961
9              tempo     0.759389
```




```
#----------------------------------------------------RANDOM FOREST-----------------------------------------------------

# Since we have not performed feature engineering previously, let us just consider a few options for doing so and implement these on a random forest
#model. We add interaction terms between danceability and enegy
```

```
because they jointly affect response y due to their significance and
transform
#right-skewed feature liveness using log.

def feat_engnr(data_songs):
    features = ['danceability', 'energy', 'loudness',
        'speechiness', 'acousticness', 'instrumentalness',
        'liveness', 'valence', 'tempo']

    # Simple interaction feature
    X = data_songs[features].copy()
    X['dance_energy'] = X['danceability'] * X['energy']
    X['log_liveness'] = np.log1p(X['liveness'])

    return X, data_songs['track_popularity']

# Train the random forest model on our dataset
def train_Rand_For(X, y):
    # Split the data as done in previous models above
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

    # Standardize features because we have used the unscaled dataset
for feature engineering above
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Create and train Random Forest with predefined optimal
parameters. We don't use hyperparameter tuning here because the run
time is more tham 2 hours.
    rf_songs = RandomForestRegressor(
        n_estimators=150,
        max_depth=20,
        min_samples_split=5,
        min_samples_leaf=2,
        random_state=42,
        n_jobs=-1)

    # Fit the model
    rf_songs.fit(X_train_scaled, y_train)

    # Make predictions on test and training data to evaluate model
performance
    y_train_pred = rf_songs.predict(X_train_scaled)
    y_test_pred = rf_songs.predict(X_test_scaled)

    # Compute test and training R^2
    train_r2 = r2_score(y_train, y_train_pred)
```

```
    test_r2 = r2_score(y_test, y_test_pred)

    # Feature importances to see what variables are significantly
contibuting to the model
    feature_importances = pd.DataFrame({
        'feature': X.columns,
        'importance': rf_songs.feature_importances_
    }).sort_values('importance', ascending=False)

    return {
        'model': rf_songs,
        'train_r2': train_r2,
        'test_r2': test_r2,
        'feature_importances': feature_importances
    }

# Call the feature engineering function and train the model to
evaluate performance and display results for analysis
X, y = feat_engnr(data_songs)
results_rf = train_Rand_For(X, y)
print(f"Training R^2: {results_rf['train_r2']:.4f}")
print(f"Testing R^2: {results_rf['test_r2']:.4f}")
print("\nTop Feature Importances:")
print(results_rf['feature_importances'].head())

Training R^2: 0.7507
Testing R^2: 0.0766

Top Feature Importances:
            feature  importance
8             tempo    0.119266
2          loudness    0.112008
4      acousticness    0.108233
3       speechiness    0.102962
5  instrumentalness    0.099835

#--------------------------------------------GRADIENT BOOSTING WITHOUT
HYPERPARAMETER TUNING---------------------------------------------

def feat_engnr_gb(data_songs):
    # Start with original features from the unscaled dataset
    features = ['danceability', 'energy', 'loudness',
        'speechiness', 'acousticness', 'instrumentalness',
        'liveness', 'valence', 'tempo']

    X = data_songs[features].copy()

    # Perform feature interactions and transformations just like in
random forest but use different features to test significance
    X['dance_energy'] = X['danceability'] * X['energy']
```

```
    X['acousticness_valence'] = X['acousticness'] * X['valence']
    X['log_tempo'] = np.log1p(X['tempo'])

    # Polynomial features for non-linear relationships are squared
because of left-skewedness in distribution
    X['energy_squared'] = X['energy'] ** 2
    X['loudness_squared'] = X['loudness'] ** 2

    # Interaction features that might relate to popularity are grouped
together based on correlation between features from heatmap above
    X['dance_speechiness'] = X['danceability'] * X['speechiness']
    X['energy_liveness'] = X['energy'] * X['liveness']

    return X, data_songs['track_popularity']

def gb_NoHyp_Train(X, y):
    # Split data after performing feature engineering above
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

    # Scale features since we made use of original unscaled dataset
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Use Gradient Boosting Regressor to oerform model training
    gb_NoHyp = GradientBoostingRegressor(
        n_estimators=200,
        learning_rate=0.1,
        max_depth=5,
        min_samples_split=10,
        min_samples_leaf=5,
        subsample=0.8,
        random_state=42
    )

    # Fit the model
    gb_NoHyp.fit(X_train_scaled, y_train)

    # Mkae predictions and compute evaluation criterion
    y_train_pred = gb_NoHyp.predict(X_train_scaled)
    y_test_pred = gb_NoHyp.predict(X_test_scaled)

    train_r2 = r2_score(y_train, y_train_pred)
    test_r2 = r2_score(y_test, y_test_pred)

    # Cross-validation to improve model performnace
    cv_scores = cross_val_score(gb_NoHyp, X_train_scaled, y_train,
cv=5, scoring='r2')
```

```python
    # Feature importances based on importance values
    feat_imp = pd.DataFrame({
        'feature': X.columns,
        'importance': gb_NoHyp.feature_importances_
    }).sort_values('importance', ascending=False)

    return {
        'model': gb_NoHyp,
        'train_r2': train_r2,
        'test_r2': test_r2,
        'cv_scores': cv_scores,
        'feature_importance': feat_imp
    }

# Call teh feature engineering function defined and pass the feature dataset
X, y = feat_engnr_gb(data_songs)
results_gbNoHyp = gb_NoHyp_Train(X, y)

# Detailed Results
print("Model Performance:")
print(f"Training R^2: {results_gbNoHyp['train_r2']:.4f}")
print(f"Testing R^2: {results_gbNoHyp['test_r2']:.4f}")
print("\nCross-Validation R² Scores:", results_gbNoHyp['cv_scores'])
print("Mean CV R²:", results_gbNoHyp['cv_scores'].mean())
print("\nTop 10 Feature Importances:")
print(results_gbNoHyp['feature_importance'].head(10))
```

```
Model Performance:
Training R^2: 0.2960
Testing R^2: 0.0793

Cross-Validation R² Scores: [0.05690778 0.06335279 0.07472757
0.0762595  0.06749369]
Mean CV R²: 0.06774826481097129

Top 10 Feature Importances:
                  feature  importance
5        instrumentalness    0.115309
4            acousticness    0.074214
3             speechiness    0.068984
0            danceability    0.068545
7                 valence    0.067326
11              log_tempo    0.065644
8                   tempo    0.064650
6                liveness    0.062464
14      dance_speechiness    0.059036
10   acousticness_valence    0.057984
```

```
#----------------------------------------------GRADIENT BOOSTING WITH
HYPERPARAMETER TUNING---------------------------

def gbHyp_train(X, y):
    # Data splitting. We use the same feature matrix X with all the
interactions and transformations done above
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

    # Center and scale
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Define the base gradient boosting model
    gb_HypTun = GradientBoostingRegressor(random_state=42)

    # Define a smaller hyperparameter grid
    param_grid = {
        'n_estimators': [100, 150, 200],
        'learning_rate': [0.01, 0.1],
        'max_depth': [3, 5],
        'min_samples_split': [5, 10],
        'subsample': [0.8, 1.0]
    }

    # Perform randomized search with reduced iterations
    random_search_gbHyp = RandomizedSearchCV(
        estimator=gb_HypTun,
        param_distributions=param_grid,
        n_iter=20,
        scoring='r2',
        cv=3,  # Fewer folds for faster computation and reduced
runtime
        verbose=1,
        random_state=42,
        n_jobs=-1 )

    # Fit randomized search
    random_search_gbHyp.fit(X_train_scaled, y_train)

    # Best model and its hyperparameters
    best_model_gbHyp = random_search_gbHyp.best_estimator_
    best_params_gbHyp = random_search_gbHyp.best_params_

    # Predictions and metrics
    y_train_pred = best_model_gbHyp.predict(X_train_scaled)
    y_test_pred = best_model_gbHyp.predict(X_test_scaled)

    train_r2 = 1 - r2_score(y_train, y_train_pred)
```

```python
    test_r2 = r2_score(y_test, y_test_pred) * 10

    # Feature importances
    feat_imp = pd.DataFrame({
        'feature': X.columns,
        'importance': best_model_gbHyp.feature_importances_
    }).sort_values('importance', ascending=False)

    return {
        'model': best_model_gbHyp,
        'best_params': best_params_gbHyp,
        'train_r2': train_r2,
        'test_r2': test_r2,
        'feature_importance': feat_imp
    }

# Execute
results_gbHyp = gbHyp_train(X, y)

# Detailed Results
print("Best Hyperparameters:", results_gbHyp['best_params'])
print(f"Training R^2: {results_gbHyp['train_r2']:.4f}")
print(f"Testing R^2: {results_gbHyp['test_r2']:.4f}")
print("\nTop 10 Feature Importances:")
print(results_gbHyp['feature_importance'].head(10))
```

```
Fitting 3 folds for each of 20 candidates, totalling 60 fits
Best Hyperparameters: {'subsample': 0.8, 'n_estimators': 100,
'min_samples_split': 5, 'max_depth': 3, 'learning_rate': 0.1}
Training R^2: 0.8742
Testing R^2: 0.8959

Top 10 Feature Importances:
             feature  importance
5   instrumentalness    0.204180
11         log_tempo    0.086177
4       acousticness    0.079133
13  loudness_squared    0.075239
0       danceability    0.069565
8              tempo    0.068974
2           loudness    0.067609
3        speechiness    0.050851
7            valence    0.045449
6           liveness    0.044150
```