+ Code + Text

Group Number: 35

Student ID's: 1006031820, 1003556831, 1009692110

Student Names: Berke Derin Berktay, Paris Wang, Dareen Christabel

Instructor: Maher Elshakankiri

Course code: INF1340

Course name: Programming for Data Science

University of Toronto

Final Project

Please refer to the README file & API documentation for information regarding the program.

PART 1: DATA CLEANING

Set Up Dataframe

```
[1] import pandas as pd
    import numpy as np
    import seaborn as sns
    import matplotlib.pyplot as plt
    import statsmodels.api as sm
    from sklearn.preprocessing import LabelEncoder
    from statsmodels.formula.api import ols
```

```
[2] from google.colab import drive
    drive.mount('/content/drive') # reading the data from google drive
    df = pd.read_csv('/content/drive/MyDrive/spotify-2023.csv', encoding='iso-8859-1')
    df.head()
```

```
Mounted at /content/drive
```

	track_name	artist(s)_name	artist_count	released_year	released_month	released_day	in_spotify_playlists	in_spotify_charts	streams	in_apple_playlists	...	bpm	key	mode	danceability_%	valence_%	energy_%	acou
0	Seven (feat. Latto) (Explicit Ver.)	Latto, Jung Kook	2	2023	7	14	553	147	141381703	43	...	125	B	Major	80	89	83	
1	LALA	Myke Towers	1	2023	3	23	1474	48	133716286	48	...	92	C#	Major	71	61	74	
2	vampire	Olivia Rodrigo	1	2023	6	30	1397	113	140003974	94	...	138	F	Major	51	32	53	
3	Cruel Summer	Taylor Swift	1	2019	8	23	7858	100	800840817	116	...	170	A	Major	55	58	72	
4	WHERE SHE GOES	Bad Bunny	1	2023	5	18	3133	50	303236322	84	...	144	A	Minor	65	23	80	

5 rows × 24 columns

```
[3] df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 953 entries, 0 to 952
Data columns (total 24 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   track_name            953 non-null    object
 1   artist(s)_name        953 non-null    object
 2   artist_count          953 non-null    int64
 6   in_spotify_playlists  953 non-null    int64
 7   in_spotify_charts     953 non-null    int64
 8   streams               952 non-null    float64
 9   in_apple_playlists    953 non-null    int64
 10  in_apple_charts       953 non-null    int64
 11  in_deezer_playlists   953 non-null    int64
 12  in_deezer_charts      953 non-null    int64
 13  in_shazam_charts      903 non-null    float64
 14  bpm                   953 non-null    int64
 15  key                   858 non-null    object
 16  mode                  953 non-null    object
 17  danceability_%        953 non-null    int64
 18  valence_%             953 non-null    int64
 19  energy_%              953 non-null    int64
 20  acousticness_%        953 non-null    int64
 21  instrumentalness_%    953 non-null    int64
 22  liveness_%            953 non-null    int64
 23  speechiness_%         953 non-null    int64
dtypes: float64(2), int64(18), object(4)
memory usage: 178.8+ KB
```

Check Missing Values

```
[6] # we are checking for na values to see if there are anomalies
    empty_cells = df.isna().sum()
    print(empty_cells)
```

```
track_name              0
artist(s)_name          0
artist_count            0
released_year           0
released_month          0
released_day            0
in_spotify_playlists    0
in_spotify_charts       0
streams                 1
in_apple_playlists      0
in_apple_charts         0
in_deezer_playlists     0
in_deezer_charts        0
in_shazam_charts       50
bpm                     0
key                    95
mode                    0
danceability_%          0
valence_%               0
energy_%                0
acousticness_%          0
instrumentalness_%      0
liveness_%              0
speechiness_%           0
dtype: int64
```

As we can see, streams column has one missing value.

```
[7] # check the row with missing streams
    missing_streams_rows = df[df['streams'].isnull()]
    missing_streams_rows
```

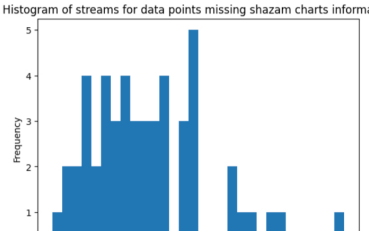
	track_name	artist(s)_name	artist_count	released_year	released_month	released_day	in_spotify_playlists	in_spotify_charts	streams	in_apple_playlists	...	bpm	key	mode	danceability_%	valence_%	energy
574	Love Grows (Where My Rosemary Goes)	Edison Lighthouse	1	1970	1	1	2877	0	NaN	16	...	110	A	Major	53	75	

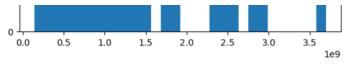
1 rows × 24 columns

50%	1.000000	2020.000000	6.500000	14.000000	10236.500000	2.500000	1.199093e+09	168.000000	82.000000	521.500000	0.000000	NaN	122.500000	65.500
75%	1.000000	2021.000000	9.000000	21.000000	16200.500000	17.000000	1.726432e+09	257.250000	122.000000	960.000000	6.000000	NaN	143.250000	74.000
max	7.000000	2022.000000	12.000000	31.000000	43899.000000	130.000000	3.703895e+09	672.000000	199.000000	6508.000000	46.000000	NaN	206.000000	86.000

```
[10] # plot histogram
     missing_in_shazam_charts_rows['streams'].plot(kind='hist', bins=30)
     plt.title("Histogram of streams for data points missing shazam charts information")
```

```
Text(0.5, 1.0, 'Histogram of streams for data points missing shazam charts information')
```






```
[11] # Get rows with missing values in the 'key' column
missing_key_rows = df[df['key'].isnull()]

# get some information about these rows
missing_key_rows.describe()
```

	artist_count	released_year	released_month	released_day	in_spotify_playlists	in_spotify_charts	streams	in_apple_playlists	in_apple_charts	in_deezer_playlists	in_deezer_charts	in_shazam_charts	bpm	danceability
count	95.000000	95.000000	95.000000	95.000000	95.000000	95.000000	9.500000e+01	95.000000	95.000000	95.000000	95.000000	86.000000	95.000000	95.000
mean	1.600000	2018.210526	6.105263	15.789474	5180.621053	12.589474	5.211925e+08	72.221053	60.252632	297.589474	3.115789	83.209302	119.947368	64.378
std	1.124202	11.251378	3.559446	8.151354	7546.135952	22.896852	5.260315e+08	85.969463	52.126272	908.679653	8.084478	220.711903	26.899881	14.244
min	1.000000	1952.000000	1.000000	1.000000	166.000000	0.000000	1.159939e+07	0.000000	0.000000	0.000000	0.000000	0.000000	65.000000	28.000
25%	1.000000	2020.000000	3.000000	10.000000	995.500000	0.000000	1.838929e+08	16.500000	14.500000	21.000000	0.000000	0.000000	96.000000	54.500
50%	1.000000	2022.000000	6.000000	17.000000	1963.000000	4.000000	3.114824e+08	35.000000	52.000000	61.000000	0.000000	2.000000	120.000000	65.000
75%	2.000000	2022.000000	9.000000	21.000000	5657.000000	16.000000	6.383046e+08	103.500000	96.500000	163.500000	2.000000	33.000000	138.000000	75.000
max	8.000000	2023.000000	12.000000	31.000000	42798.000000	115.000000	2.322580e+09	437.000000	227.000000	6508.000000	58.000000	1173.000000	206.000000	91.000

```
[12] # plot histogram
missing_key_rows['streams'].plot(kind='hist', bins=30)
plt.title("Histogram of streams for data points missing key information")
```

```
Text(0.5, 1.0, 'Histogram of streams for data points missing key information')
 13  in_shazam_charts      816 non-null    float64
 14  bpm                   816 non-null    int64
 15  key                   816 non-null    object
 16  mode                  816 non-null    object
 17  danceability_%        816 non-null    int64
 18  valence_%             816 non-null    int64
 19  energy_%              816 non-null    int64
 20  acousticness_%        816 non-null    int64
 21  instrumentalness_%    816 non-null    int64
 22  liveness_%            816 non-null    int64
 23  speechiness_%         816 non-null    int64
dtypes: float64(2), int64(18), object(4)
memory usage: 159.4+ KB
```

Feature Transformation

Merge Release Day, Month, and Year to Release Date

```
[15] # we might need a new column where we have the complete date as a date type for each track on top of them having the year, day, and month separately
     # let's name this new column release_date
     df['release_date'] = pd.to_datetime({'year': df['released_year'],
                                          'month': df['released_month'],
                                          'day': df['released_day']})
     df['release_date']
```

```
<ipython-input-15-692a29565f62>:3: 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
  df['release_date'] = pd.to_datetime({'year': df['released_year'],
0     2023-07-14
1     2023-03-23
2     2023-06-30
3     2019-08-23
4     2023-05-18
         ...
948   2022-11-03
949   2022-10-21
950   2022-11-03
951   2022-10-20
952   2022-11-04
Name: release_date, Length: 816, dtype: datetime64[ns]
```

Create Average Monthly Stream Value

```
[16] # To overcome the time advantage of these songs, we will create a new column "streams-per-month" based on the months since the song's release.
     # since the dataset updated the last time 3 months ago, we will take the end date for the calculations as August 1st, 2023
     # we also consider the months that has passed ever since the song's released into our calculation
     from pandas.tseries.offsets import MonthEnd
     import numpy as np

     df['streams'] = pd.to_numeric(df['streams'], errors='coerce')

     reference_date = pd.Timestamp('2023-09-01')

     df['months_passed'] = ((reference_date - df['release_date']) / np.timedelta64(1, 'M')).astype(int)
     df['avg_streams_per_month'] = df['streams'] / df['months_passed'].replace(0, np.nan)
     df = df.drop(columns=['months_passed'])
     df['avg_streams_per_month']
```

```
[18] for i in range(5,9):
       df['artist_count'].replace(i, 4, inplace=True)

     # check value counts again
     df['artist_count'].value_counts()
```

```
1    493
2    224
3     75
4     24
Name: artist_count, dtype: int64
```

PART 2: DESCRIPTIVE ANALYTICS, DATA VISUALIZATION AND TAKEAWAYS

```
[19] # to get an overview of the data set after cleaning
     df.describe()
```

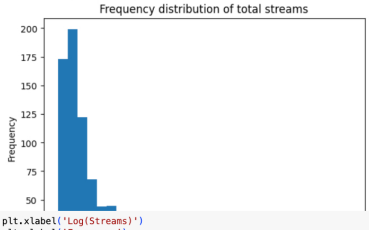
	artist_count	released_year	released_month	released_day	in_spotify_playlists	in_spotify_charts	streams	in_apple_playlists	in_apple_charts	in_deezer_playlists	...	in_shazam_charts	bpm	danceability_%	valence_%
count	816.000000	816.000000	816.000000	816.000000	816.000000	816.000000	8.160000e+02	816.000000	816.000000	816.000000	...	816.000000	816.000000	816.000000	816.000000
mean	1.546569	2018.517157	6.024510	13.712010	4852.316176	11.736520	4.689858e+08	60.215686	49.534314	372.053922	...	57.622549	122.580882	67.409314	51.172794
std	0.780302	10.701973	3.570415	9.294719	7745.565488	18.624555	5.231267e+08	74.953565	49.570536	1157.759053	...	153.616316	28.188643	14.688808	23.620742
min	1.000000	1930.000000	1.000000	1.000000	31.000000	0.000000	2.762000e+03	0.000000	0.000000	0.000000	...	0.000000	65.000000	23.000000	4.000000
25%	1.000000	2021.000000	3.000000	5.000000	829.000000	0.000000	1.342848e+08	12.000000	6.000000	12.750000	...	0.000000	99.000000	57.000000	32.000000
50%	1.000000	2022.000000	5.000000	13.000000	2037.500000	3.000000	2.638368e+08	32.000000	34.500000	39.000000	...	3.000000	120.000000	70.000000	51.000000
75%	2.000000	2022.000000	9.000000	22.000000	4890.750000	16.000000	6.011986e+08	78.250000	84.000000	142.250000	...	38.250000	141.250000	79.000000	70.000000
max	4.000000	2023.000000	12.000000	31.000000	52898.000000	147.000000	3.562544e+09	532.000000	275.000000	12367.000000	...	1451.000000	206.000000	96.000000	97.000000

8 rows × 21 columns

Total Streams & Average Monthly Streams

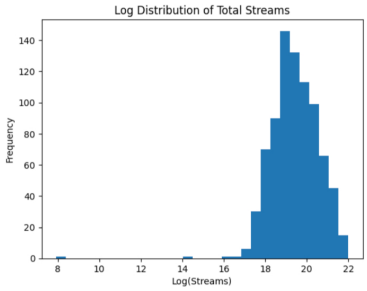
```
[20] # let's look at the distribution of the dependent variable "stream"
     df['streams'].plot(kind='hist', bins=30)
     plt.title("Frequency distribution of total streams")
```

```
Text(0.5, 1.0, 'Frequency distribution of total streams')
```




```
plt.xlabel('Log(Streams)')
plt.ylabel('Frequency')
```

```
plt.title("Log Distribution of Total Streams")

# Show the plot
plt.show()
```



We can see from the frequency distribution that the shape of streams is nearly normal.

Next, let's find the outliers using the lognormal distribution of streams

```
[23] Q1 = np.log1p(df['streams']).quantile(0.25)
     Q3 = np.log1p(df['streams']).quantile(0.75)
     IQR = Q3 - Q1

     # Define the upper and lower bounds to identify outliers
     lower_bound = Q1 - 1.5 * IQR
     upper_bound = Q3 + 1.5 * IQR

     # Identify rows with outliers
     outliers = df[(np.log1p(df['streams']) < lower_bound) | (np.log1p(df['streams']) > upper_bound)]
     outliers
```

	track_name	artist(s)_name	artist_count	released_year	released_month	released_day	in_spotify_playlists	in_spotify_charts	streams	in_apple_playlists	...	mode	danceability_%	valence_%	ener
123	Que Vuelvas	Carin Leon, Grupo Frontera	2	2022	12	9	763	26	2762.0	21	...	Major	49	78	
142	Gol Bolinha, Gol Quadrado 2	Mc Pedrinho, DJ 900	2	2023	6	1	293	8	11956641.0	5	...	Minor	93	68	
393	Jhoome Jo Pathaan	Arijit Singh, Vishal Dadlani, Sukriti Kakar, V...	4	2022	12	22	138	4	1365184.0	13	...	Major	82	62	

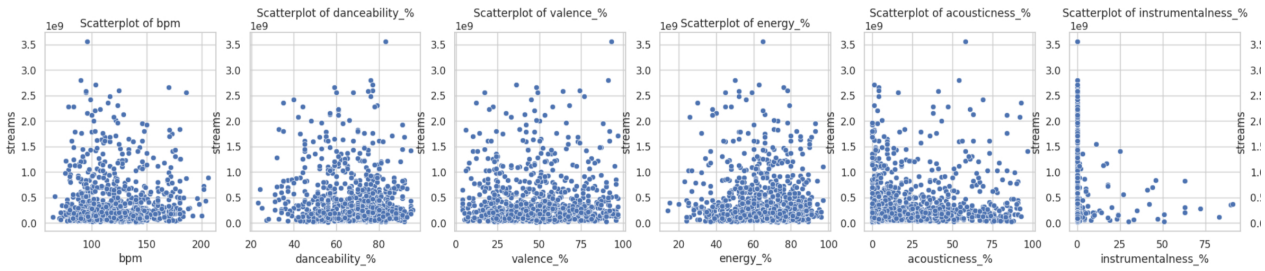
3 rows × 26 columns

Remove Outliers

```
[24] df = df.drop(outliers.index)
```

We also graphed the relationship between total streams and these song features using a scatterplot.

```
[27] sns.set(style="whitegrid")

     # Set up subplots
     fig, axes = plt.subplots(nrows=1, ncols=len(song_features), figsize=(32, 4))

     # Create side-by-side scatterplots
     for i, feature in enumerate(song_features):
         sns.scatterplot(x=feature, y='streams', data=df, ax=axes[i])
         axes[i].set_title(f'Scatterplot of {feature}')
         axes[i].set_xlabel(feature)
         axes[i].set_ylabel('streams')
```

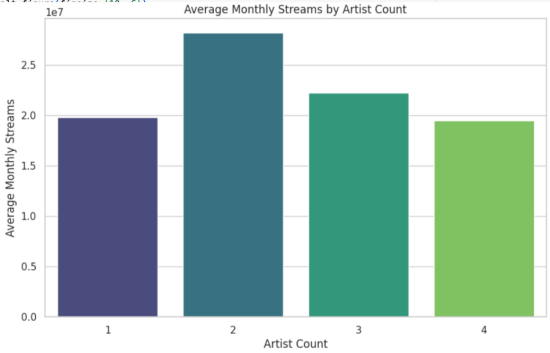


From the scatterplots of artistic features such as danceability and energy of the songs vs streams, we can see that

- These are no songs on the most streamed list that are less than 20% danceable.
- Most songs on the list have a 0% instrumentalness, which means most of the top 1000 songs have vocals in it.
- Most top streamed songs tend to have low liveness, which means they were recorded instead of live versions.
- There is no obvious association betweeen bpm and streams. Danceability and energy, though having similar patterns, also have no association with streams. Sam for valence, which describes the musical positivity conveyed by a piece of music.

Artist Count

```
[28] # What about streams by artist count? Does having more artists attract more listeners?
     streams_by_artist_count = df.groupby('artist_count')['streams'].mean().reset_index()

     # Sort the data by streams in descending order for a more meaningful plot
     streams_by_artist_count = streams_by_artist_count.sort_values(by='streams', ascending=False)

     # Create a bar plot
```

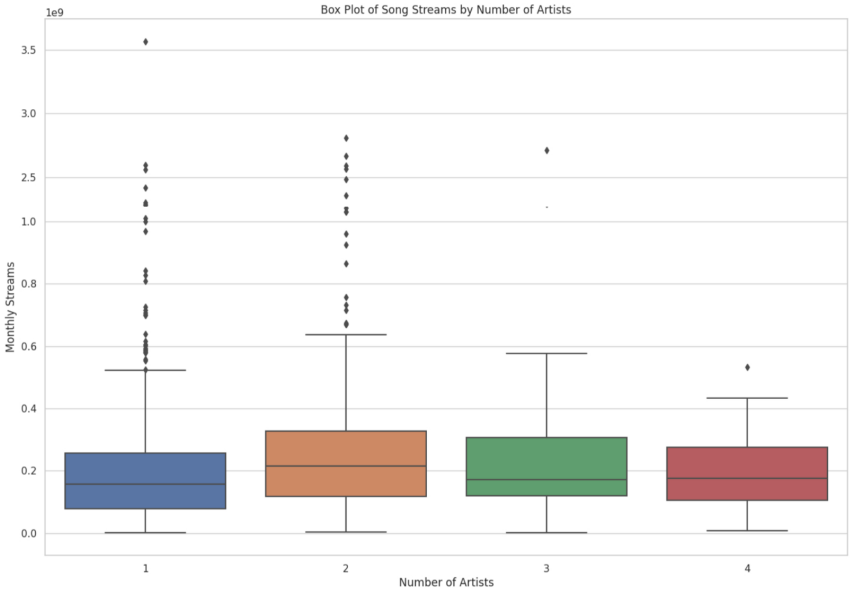


From the plot, we can see that after removing the time advantages of these songs (the fact that older songs have had more time to accumulate streams on the platform), there is not obvious patterns in artist count and song streams.

Instead, let's look at box plots.

```
[29] plt.figure(figsize=(15, 10))
```

```
[30] plt.figure(figsize=(10, 10))
     sns.boxplot(x='artist_count', y='streams', data=df)

     plt.title('Box Plot of Song Streams by Number of Artists')
     plt.xlabel('Number of Artists')
     plt.ylabel('Streams')

     plt.show()
```



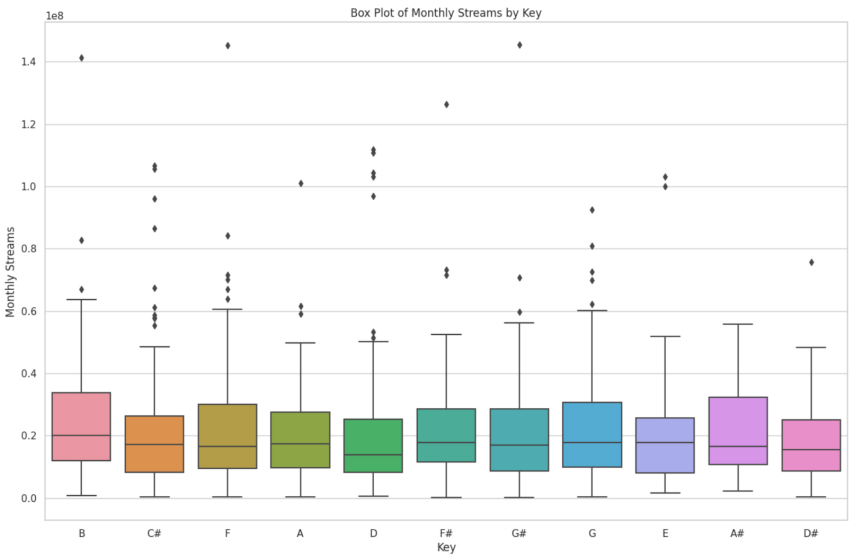
We can see from the box plots that although average streams for songs look similar across artist counts, the spread looks very different. There appears to be a pattern of the standard deviations of streams as artist counts grew, however, this is most likely due to most songs have less than 3 artists.

Again, there is no obvious patterns for average monthly streams across artist counts, though the average monthly streams for different artist counts look very different. In part 3, we will perfom ANOVA to determine if the average are really different across groups given the difference in spread (aka standard deviation).

Key & Mode

```
[32] # we can also visualize the most commonly and rarely used song key
     key_counts = df['key'].value_counts()
     key_counts
```

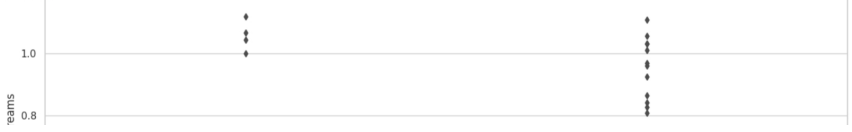
```
C#    115
G      90
F      87
G#     85
D      78
B      76
A      70
F#     69
E      59
A#     54
D#     30
Name: key, dtype: int64
```

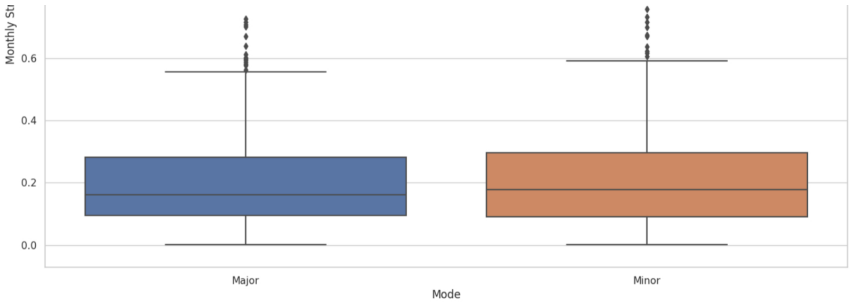



```
[35] # we can do the same for mode
     mode_counts = df['mode'].value_counts()
     mode_counts
```

```
Major    449
Minor    364
Name: mode, dtype: int64
```

```
[36] plt.figure(figsize=(10, 6))
     sns.barplot(x=mode_counts.index, y=mode_counts)
     plt.title('Count of Songs by Mode')
     plt.xlabel('Song Mode')
     plt.ylabel('Count')
     plt.xticks(rotation=70)
     plt.show()
```





Pivot Table of Artistic Features

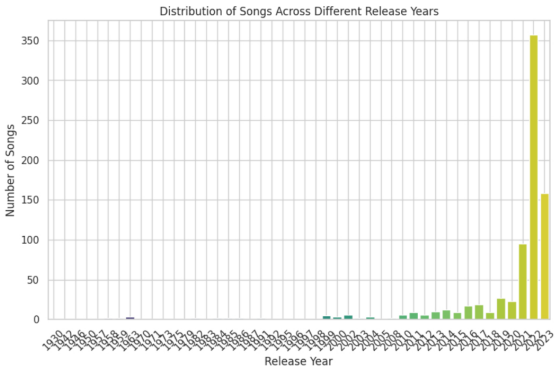
After doing some research, we realized that the mode of a song plays a significant role in shaping the mood and vibe of the music.

```
[38] pivot_table_result = pd.pivot_table(df,
                                         values=['danceability_%',
            'valence_%', 'energy_%', 'acousticness_%',
            'liveness_%', 'speechiness_%'],
                                         index=['mode'],
                                         aggfunc='mean')
     pivot_table_result.round(1)
```

mode	acousticness_%	danceability_%	energy_%	liveness_%	speechiness_%	valence_%
Major	27.9	65.5	63.3	18.3	9.7	49.5
Minor	24.4	69.7	65.6	18.0	11.5	53.1

Let's visualize the relationships between streams and other numerical variables that are not related to the intrinsic artistic characteristics of a song.

```
[42] #Distribution of songs across different release years
     release_year_distribution = df['released_year'].value_counts().sort_index()

     plt.figure(figsize=(10, 6))
     sns.barplot(x=release_year_distribution.index, y=release_year_distribution.values, palette="viridis")
     plt.title("Distribution of Songs Across Different Release Years")
     plt.xlabel("Release Year")
     plt.ylabel("Number of Songs")
     plt.xticks(rotation=45)
     plt.grid(True)
     plt.show()
```



PART 3: DIAGNOSTIC ANALYTICS, DATA VISUALIZATION AND TAKEAWAYS

ANOVA

After visualizing data distribution by categorical variables (artist_count, key and mode), we wanted to test if the streams and avg monthly streams are truly different across different levels of the categorical variables

Artist Count vs Streams

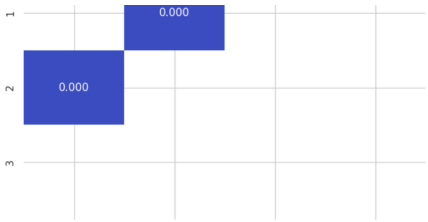
As we can see from the box plots in part 2, there is a lot of noise for songs with artist count > 4. For this reason, we decided to do an ANOVA

```
Collecting scikit-posthocs
  Downloading scikit_posthocs-0.8.0-py3-none-any.whl (32 kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from scikit-posthocs) (1.23.5)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from scikit-posthocs) (1.11.4)
Requirement already satisfied: statsmodels in /usr/local/lib/python3.10/dist-packages (from scikit-posthocs) (0.14.0)
Requirement already satisfied: pandas>=0.20.0 in /usr/local/lib/python3.10/dist-packages (from scikit-posthocs) (1.5.3)
Requirement already satisfied: seaborn in /usr/local/lib/python3.10/dist-packages (from scikit-posthocs) (0.12.2)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (from scikit-posthocs) (3.7.1)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.20.0->scikit-posthocs) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.20.0->scikit-posthocs) (2023.3.post1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->scikit-posthocs) (1.2.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib->scikit-posthocs) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->scikit-posthocs) (4.46.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->scikit-posthocs) (1.4.5)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->scikit-posthocs) (23.2)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->scikit-posthocs) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->scikit-posthocs) (3.1.1)
Requirement already satisfied: patsy>=0.5.2 in /usr/local/lib/python3.10/dist-packages (from statsmodels->scikit-posthocs) (0.5.4)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from patsy>=0.5.2->statsmodels->scikit-posthocs) (1.16.0)
Installing collected packages: scikit-posthocs
Successfully installed scikit-posthocs-0.8.0
```

```
[46] import scikit_posthocs as sp
     p_values1 = sp.posthoc_dunn(df, val_col='avg_streams_per_month', group_col='artist_count')
     print(p_values1)
```

```
          1         2         3         4
1  1.000000  0.000002  0.125544  0.576665
2  0.000002  1.000000  0.150235  0.230016
3  0.125544  0.150235  1.000000  0.766505
4  0.576665  0.230016  0.766505  1.000000
```

```
[47] alpha = 0.05
     mask = p_values1 < alpha

     plt.figure(figsize=(8, 6))
     sns.heatmap(p_values1, annot=True, cmap='coolwarm', fmt=".3f", mask=~mask, cbar=False)

     plt.title("Dunn Test p-values of artist count and average streams per month")
     plt.xlabel("Artist Count")
     plt.show()
```



Given the p-value of 3.622793e-07, we have enough evidence to reject to null hypothesis that the group means of **avg monthly streams** are the same and conclude with 95% confidence that at least one of the months has a different group mean for average monthly streams than the rest.

What about total streams?

```
[50] # Fit the model with total streams and perform ANOVA
     model_month_total = ols('streams ~ C(released_month)', data=df).fit()
     anova_table_month_total = sm.stats.anova_lm(model_month_total, typ=2)

     # display ANOVA table
     print(anova_table_month_total)
```

```
                       sum_sq     df         F        PR(>F)
C(released_month)  1.582141e+19   11.0  5.577412  1.175394e-08
Residual           2.065629e+20  801.0       NaN           NaN
```

Given the p-value of 1.175394e-08, we have enough evidence to reject to null hypothesis that the group means of **total streams** are the same and conclude with 95% confidence that at least one of the months has a different group mean for average monthly streams than the rest.

ANOVA Post-Hoc Test

From our ANOVA analysis, we concluded that at least one of the group means of **avg monthly streams** and **total streams** differ from each other. Thus, we initiated a post-hoc test to see which groups have different means.

```
[51] import scikit_posthocs as sp
     p_values2 = sp.posthoc_dunn(df, val_col='avg_streams_per_month', group_col='released_month')
     print(p_values2)
```

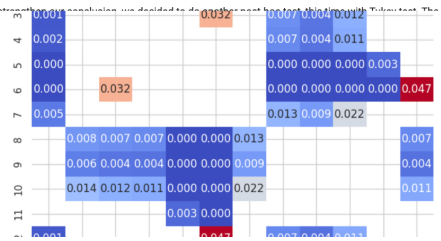
```
             1            2            3            4            5   \
1   1.000000e+00  0.000006  1.365643e-08  8.237629e-10  1.601646e-07
2   5.846530e-06  1.000000  5.852460e-01  2.566313e-01  7.255836e-01
3   1.365643e-08  0.585246  1.000000e+00  5.112836e-01  2.954276e-01
4   8.237629e-10  0.256631  5.112836e-01  1.000000e+00  8.569238e-02
5   1.601646e-07  0.725584  2.954276e-01  8.569238e-02  1.000000e+00
6   4.219422e-10  0.299250  5.904533e-01  8.913544e-01  1.019035e-01
7   3.160427e-10  0.091867  2.007922e-01  5.122981e-01  2.231383e-02
8   1.006140e-03  0.515768  2.368398e-01  8.876380e-02  6.701490e-01
9   3.171465e-04  0.535142  2.352152e-01  8.280720e-02  7.029220e-01
10  2.657194e-07  0.768359  7.998986e-01  3.796234e-01  4.706596e-01
11  7.170172e-03  0.066615  1.012153e-02  1.801339e-03  7.028965e-02
12  1.119647e-07  0.713054  8.567782e-01  4.148929e-01  4.145976e-01

             6            7         8         9            10        11  \
1   4.219422e-10  3.160427e-10  0.001006  0.000317  2.657194e-07  0.007170
2   2.992499e-01  9.186661e-02  0.515768  0.535142  7.683586e-01  0.066615
3   5.904533e-01  2.007922e-01  0.236840  0.235215  7.998986e-01  0.010122
4   8.913544e-01  5.122981e-01  0.888764  0.882807  3.796234e-01  0.001801
5   1.019035e-01  2.231383e-02  0.670149  0.702922  4.706596e-01  0.070290
6   1.000000e+00  4.242009e-01  0.103966  0.097432  4.408174e-01  0.002017
7   4.242009e-01  1.000000e+00  0.029157  0.025683  1.422546e-01  0.000397
8   1.039655e-01  2.915727e-02  1.000000  0.952095  3.473105e-01  0.325601
9   9.743234e-02  2.568334e-02  0.952095  1.000000  3.536922e-01  0.266666
10  4.408174e-01  1.422546e-01  0.347310  0.353692  1.000000e+00  0.025587
11  2.016743e-03  3.969342e-04  0.325601  0.266666  2.558706e-02  1.000000
12  4.811659e-01  1.573213e-01  0.311788  0.315465  9.414605e-01  0.019640

             12
1   1.119647e-07
2   7.130545e-01
```

average monthly stream that couldn't have occured due to random chance alone. Some notable ones are **January with all of the months**, as well as **July-November**.

```
[53] from statsmodels.stats.multicomp import pairwise_tukeyhsd

     tukey_results = pairwise_tukeyhsd(endog=df['avg_streams_per_month'], groups=df['released_month'], alpha=0.05)
     print(tukey_results)
```

```
              Multiple Comparison of Means - Tukey HSD, FWER=0.05
=================================================================================
group1 group2   meandiff    p-adj      lower          upper       reject
---------------------------------------------------------------------------------
     1      2  10850032.5177 0.0463      82064.164 21618000.8714   True
     1      3  12809626.5893 0.0011    3094372.4131 22524880.7655   True
     1      4  13421498.2249  0.001    3250150.9412 23592845.5087   True
     1      5   9830681.9418 0.0102    1237944.7899 18423419.0938   True
     1      6  12345334.0338 0.0022    2589320.4462 22101347.6214   True
     1      7  19827575.7709    0.0    8602756.2626 31052395.2791   True
     1      8   7072065.877 0.7517   -5052984.8561   19197116.61  False
     1      9   7637819.0636 0.5403   -3672617.3066 18948255.4337  False
     1     10  11457085.072 0.0126    1285737.7882 21628432.3557   True
     1     11   4193332.9892 0.9686   -5828975.5067 14215641.4851  False
     1     12  11944555.2627 0.0057    1922246.7668 21966863.7585   True
     2      3   1959594.0716    1.0   -9695738.02 13614926.1633  False
     2      4   2571465.7073 0.9999   -9466678.1706 14609609.5851  False
     2      5  -1019350.5759    1.0  -11756935.6964  9718234.5446  False
     2      6   1495301.5161    1.0  -10194027.1036 13184630.1358  False
     2      7   8977543.2532 0.4962   -3962981.2899 21918067.7962  False
     2      8  -3777966.6407 0.9991  -17506675.8931  9950742.6117  False
     2      9  -3212213.4541 0.9997  -16227073.1668  9802646.2586  False
     2     10   607052.5543    1.0  -11431091.3236 12645196.4321  False
     2     11  -6656699.5285 0.8002  -18569183.3183  5255784.2614  False
     2     12    1094522.745    1.0  -10817961.0449 13007006.5348  False
     3      4    611871.6356    1.0  -10494609.7724 11718353.0437  False
     3      5  -2978944.6475 0.9975   -12660512.468   6702623.173  False
     3      6   -464292.5555    1.0  -11191707.1939 10263122.0828  False
     3      7   7017949.1816 0.7564   -5060717.8266 19096616.1897  False
     3      8  -5737560.7124 0.9517  -18657099.3643   7181977.9396  False
     3      9  -5171807.5257 0.9646  -17330080.2067   6986465.1552  False
     3     10  -1352541.5174    1.0  -12459022.9254   9753939.8907  False
     3     11  -8616293.6001  0.296   -19586448.17   2353860.9698  False
     3     12   -865071.3266    1.0  -11835225.8966 10105083.2433  False
     4      5  -3590816.2831 0.9916  -13729992.6462   6548360.0799  False
     4      6  -1076164.1912    1.0  -12218316.8863 10065988.5039  False
     4      7   6406077.5459 0.8741   -6042389.7714 18854544.8633  False
     4      8  -6349432.348 0.9198  -19615351.8728   6916487.1768  False
     4      9  -5783679.1614 0.9368  -18309402.1522   6742043.8294  False
     4     10  -1964413.153    1.0  -13471978.974    9543152.668  False
     4     11  -9228165.2357 0.2491  -20604211.6995   2147881.228  False
     4     12  -1476942.9623    1.0  -12852989.426   9899103.5015  False
     5      6   2514652.092 0.9995   -7207816.3639 12237120.5478  False
     5      7   9996893.8291 0.1335    -1198782.334 21192569.9921  False
     5      8  -2758616.0649 0.9999  -14856692.2356   9339460.1059  False
     5      9  -2192862.8782    1.0  -13474377.0779   9088651.3215  False
     5     10   1626403.1301    1.0   -8512773.2329 11765579.4932  False
     5     11  -5637348.9526 0.7897  -15627006.5714   4352308.6662  False
     5     12   2113873.3209 0.9999   -7875784.298 12103530.9397  False
     6      7   7482241.7371 0.6762   -4629233.5696 19593717.0438  False
     6      8  -5273268.1568 0.9744  -18223485.0041   7676948.6905  False
     6      9  -4707514.9702  0.983  -16898381.7153   7483351.7749  False
     6     10   -888248.9618    1.0  -12030401.6569 10253903.7333  False
     6     11  -8152001.0445 0.3884  -19158268.7453   2854266.6562  False
     6     12   -400778.7711    1.0  -11407046.4718 10605488.9296  False
     7      8 -12755509.8939 0.1202  -26845395.9213   1334376.1334  False
     7      9 -12189756.7073 0.1155  -25585054.5341   1205541.1195  False
     7     10  -8370490.6989 0.5472  -20818958.0163   4077976.6184  False
```

To strengthen our conclusion, we decided to do another post hoc test this time with Tukey test. The results show the same conclusion as Dunn



Again, from our Dunn test results, we concluded that there are multiple group pairs that have a statistically significant mean group difference of total streams that couldn't have occured due to random chance alone. Some notable ones are **January-May, January-June, May-August, May-September, May-October, June-August, June-September, June-October, and June-November**.

```
[56] from statsmodels.stats.multicomp import pairwise_tukeyhsd

     tukey_results = pairwise_tukeyhsd(endog=df['streams'], groups=df['released_month'], alpha=0.05)
     print(tukey_results)
```

```
            Multiple Comparison of Means - Tukey HSD, FWER=0.05
==========================================================================
group1 group2     meandiff    p-adj       lower           upper      reject
--------------------------------------------------------------------------
     1      2 -369729565.7645 0.0009 -647242503.3708  -92216628.1582   True
     1      3 -309825286.0237 0.0032  -560207590.047  -59442982.0005   True
     1      4 -326615595.0972 0.0028 -588752366.9953  -64478823.1991   True
     1      5 -371778685.2046    0.0 -593231391.1186 -150325979.2905   True
     1      6 -376370813.2024 0.0001   -627803572.02 -124938054.3847   True
     1      7 -313647752.8203 0.0205 -602934694.7504  -24360810.8902   True
     1      8 -133927519.3186 0.9626 -446415292.0386  178560253.4014  False
     1      9    2433780.2958    1.0 -289059686.2155  293927246.8071  False
     1     10 -189156895.8909 0.4312  -451293667.789   72979876.0072  False
     1     11 -253179136.0244 0.0608  -511474868.417    5116596.3683  False
     1     12 -354690938.7819 0.0005 -612986671.1745  -96395206.3893   True
     2      3   59904279.7408    1.0 -240477866.7601  360286426.2416  False
     2      4   43113970.6673    1.0 -267134031.6625   353361972.997  False
     2      5   -2049119.4401    1.0 -278779017.9424  274680779.0622  False
     2      6   -6641247.4379    1.0 -307899555.1658    294617060.29  False
     2      7   56081812.9442    1.0 -277422415.7242  389586041.6126  False
     2      8  235802046.4459 0.5615 -118015341.2953   589619434.187  False
     2      9  372163346.0603 0.0154   36743345.4831  707583346.6375   True
     2     10  180572669.8736 0.7543 -129675332.4561  490820672.2033  False
     2     11  116550429.7401 0.9852 -190459050.7986  423559910.2789  False
     2     12   15038626.9826    1.0 -291970853.5562  322048107.5213  False
     3      4  -16790309.0735    1.0 -303027432.0655  269446813.9185  False
     3      5  -61953399.1808 0.9997 -311467535.7628  187560737.4011  False
     3      6  -66545527.1787 0.9997 -343013311.2046  209922256.8473  False
     3      7   -3822466.7966    1.0 -315114834.4653  307469900.8722  False
     3      8  175897766.7051 0.8532 -157065611.9075  508861145.3177  False
     3      9  312259066.3195 0.0518   -1084905.0863  625603037.7253  False
```

4	WHERE SHE GOES	Bad Bunny	1	2023	5	18	3133	50	303236322.0	84	...	65	23	80	14

5 rows × 27 columns

```
[59] model_key = ols('avg_streams_per_month ~ C(key_encoded)', data=df_encoded).fit()
     anova_table_key = sm.stats.anova_lm(model_key, typ=2)
     print(anova_table_key)
```

```
                      sum_sq     df         F    PR(>F)
C(key_encoded)  1.770306e+15   10.0  0.430651  0.931948
Residual        3.296834e+17  802.0       NaN       NaN
```

Given the p-value of 0.931948, we can conclude with 95% confidence that we don't have enough evidence to reject to null hypothesis that the group means of **avg monthly streams** are the same.

```
[60] model_key = ols('streams ~ C(key_encoded)', data=df_encoded).fit()
     anova_table_key = sm.stats.anova_lm(model_key, typ=2)
     print(anova_table_key)
```

```
                      sum_sq     df         F    PR(>F)
C(key_encoded)  2.474988e+18   10.0  0.902618  0.530166
Residual        2.199093e+20  802.0       NaN       NaN
```

Given the p-value of 0.530166, we can conclude with 95% confidence that we don't have enough evidence to reject to null hypothesis that the group means of **total streams** are the same.

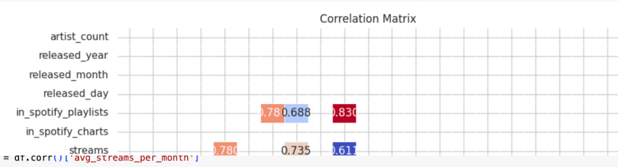
Correlation Matrix

Now, lets look into the correlation of our variables. Initially, we were curious to see if there's any statistical significant correlations between any of our variables. We decided to create a correlation matrix for our dataframe.

```
[61] correlation_table = df.corr()
```

```
<ipython-input-61-abc2845c9d45>:1: FutureWarning: The default value of numeric_only in DataFrame.corr is deprecated. In a future version, it will default to False. Select only valid columns or specify the value of numeric_only to si
  correlation_table = df.corr()
```

```
[62] alpha = 0.6
     mask = (correlation_table > alpha) & (correlation_table != 1)

     plt.figure(figsize=(10, 8))
     sns.heatmap(correlation_table, annot=True, cmap='coolwarm', fmt=".3f", linewidths=.5, mask=~mask, cbar=False)
     plt.title('Correlation Matrix')
     plt.show()
```




```
[64] cm2 = df.corr()['avg_streams_per_month']
```

```
<ipython-input-64-e2d94fef3d0d>:1: FutureWarning: The default value of numeric_only in DataFrame.corr is deprecated. In a future version, it will default to False. Select only valid columns or specify the value of numeric_only to si
  cm2 = df.corr()['avg_streams_per_month']
```

```
[65] cm3 = df.corr()['in_spotify_playlists']
```

```
<ipython-input-65-03ef00f9e8cc>:1: FutureWarning: The default value of numeric_only in DataFrame.corr is deprecated. In a future version, it will default to False. Select only valid columns or specify the value of numeric_only to si
  cm3 = df.corr()['in_spotify_playlists']
```

```
[66] cm4 = df.corr()['in_apple_playlists']
```

```
<ipython-input-66-c9389852bfba>:1: FutureWarning: The default value of numeric_only in DataFrame.corr is deprecated. In a future version, it will default to False. Select only valid columns or specify the value of numeric_only to si
  cm4 = df.corr()['in_apple_playlists']
```

```
[67] cm5 = df.corr()['in_deezer_playlists']
```

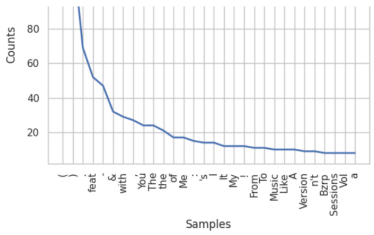
```
<ipython-input-67-3b038bd88798>:1: FutureWarning: The default value of numeric_only in DataFrame.corr is deprecated. In a future version, it will default to False. Select only valid columns or specify the value of numeric_only to si
  cm5 = df.corr()['in_deezer_playlists']
```

```
[68] merged_correlation = pd.merge(cm1,pd.merge(cm2,pd.merge(cm3,pd.merge(cm4,cm5,left_index=True, right_index=True),left_index=True, right_index=True),left_index=True, right_index=True),left_index=True, right_index=True)
```

```
[69] merged_correlation
```

	streams	avg_streams_per_month	in_spotify_playlists	in_apple_playlists	in_deezer_playlists
artist_count	-0.102661	0.085969	-0.079090	-0.005493	-0.066191
released_year	-0.241968	0.298373	-0.396631	-0.206678	-0.321351
released_month	-0.042567	0.060385	-0.121155	-0.032861	-0.101673
released_day	-0.026251	0.091297	-0.109086	-0.013831	-0.095215
in_spotify_playlists	0.780133	-0.038218	1.000000	0.687895	0.829630
in_spotify_charts	0.214624	0.584705	0.141383	0.208088	0.077816
streams	1.000000	0.196402	0.780133	0.734877	0.610804
in_apple_playlists	0.734877	0.244319	0.687895	1.000000	0.468295
in_apple_charts	0.270934	0.475916	0.216529	0.365589	0.137514
in_deezer_playlists	0.610804	-0.107665	0.829630	0.468295	1.000000
in_deezer_charts	0.185538	0.575623	0.101712	0.327533	0.039391
in_shazam_charts	0.015480	0.304467	0.081967	0.138688	0.084599
bpm	-0.024512	0.022851	-0.033990	0.006004	-0.037973
danceability_%	-0.092134	0.139045	-0.096547	-0.011813	-0.069808
valence_%	-0.048620	0.035383	-0.027980	0.043838	-0.007445
energy_%	-0.035882	0.045424	0.036462	0.040736	0.082210

acousticness_%	-0.006125	-0.044520	-0.065192	-0.071772	-0.075632
instrumentalness_%	-0.033718	-0.048019	-0.025055	-0.055014	-0.012346
liveness_%	-0.056675	-0.004038	-0.051942	-0.065083	-0.024266
speechiness_%	-0.099833	-0.054566	-0.077687	-0.097856	-0.057123
avg_streams_per_month	0.196402	1.000000	-0.038218	0.244319	-0.107665




```
<Axes: xlabel='Samples', ylabel='Counts'>
```

```
[75] # Only keep tokens that have letters
words1 = [token for token in tokens if token.isalpha()]
```

```
[76] # Fold words to lower case
words2 = [word.lower() for word in words1]
```

```
# Get the stop words list
nltk.download('stopwords')
from nltk.corpus import stopwords
stopWords = stopwords.words('english')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

```
[78] # Filter out stop words from reviews
# Removing commonly occurring words
words3 = [word for word in words2 if word not in stopWords]
```

```
[79] # Count # of occurrences of each word
wordFrequency3 = nltk.FreqDist(words3)
```

```
[80] # Visualize the refined word frequency
wordFrequency3.plot(30)
```


```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 813 entries, 0 to 952
Data columns (total 36 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   track_name            813 non-null    object
 1   artist(s)_name        813 non-null    object
 2   artist_count          813 non-null    int64
 3   released_year         813 non-null    int64
 4   released_month        813 non-null    int64
 5   released_day          813 non-null    int64
 6   in_spotify_playlists  813 non-null    int64
 7   in_spotify_charts     813 non-null    int64
 8   streams               813 non-null    float64
 9   in_apple_playlists    813 non-null    int64
 10  in_apple_charts       813 non-null    int64
 11  in_deezer_playlists   813 non-null    int64
 12  in_deezer_charts      813 non-null    int64
 13  in_shazam_charts      813 non-null    float64
 14  bpm                   813 non-null    int64
 15  danceability_%        813 non-null    int64
 16  valence_%             813 non-null    int64
 17  energy_%              813 non-null    int64
 18  acousticness_%        813 non-null    int64
 19  instrumentalness_%    813 non-null    int64
 20  liveness_%            813 non-null    int64
 21  speechiness_%         813 non-null    int64
 22  release_date          813 non-null    datetime64[ns]
 23  key_A                 813 non-null    uint8
 24  key_A#                813 non-null    uint8
 25  key_B                 813 non-null    uint8
 26  key_C#                813 non-null    uint8
 27  key_D                 813 non-null    uint8
 28  key_D#                813 non-null    uint8
 29  key_E                 813 non-null    uint8
 30  key_F                 813 non-null    uint8
 31  key_F#                813 non-null    uint8
 32  key_G                 813 non-null    uint8
 33  key_G#                813 non-null    uint8
 34  mode_Major            813 non-null    uint8
 35  mode_Minor            813 non-null    uint8
dtypes: datetime64[ns](1), float64(2), int64(18), object(2), uint8(13)
memory usage: 195.0+ KB
```

Multi-Linear Regression

Starting with exploring linear relationsips, we decided to fit a multi-linear regression model and evaluate the accuracy of the model using cross validation methods.

```
[83] # sklearn libraries for preprocessing, model selection, and metrics
from sklearn.preprocessing import StandardScaler  # To standardize features
from sklearn.linear_model import LinearRegression  # Linear regression model
from sklearn.model_selection import train_test_split  # To split the data into training and test sets
from sklearn.model_selection import KFold  # For K-fold cross-validation
from sklearn.metrics import make_scorer, confusion_matrix  # For creating custom metrics and confusion matrices
from sklearn.model_selection import learning_curve  # To generate a learning curve
```

```
[84] # Split dataset into features and target.
# drop one of each one hot encoded columns to avoid perfect multi-linearity
X = df1.drop(['streams','track_name', 'artist(s)_name', 'release_date','key_A','mode_Major'],axis=1)
y = df1['streams']
```

```
const                 4.519e+08   1.17e+07     38.636      0.000    4.29e+08    4.75e+08
released_year        -1.791e+07   1.25e+07     -1.438      0.151   -4.24e+07    6.55e+06
released_month        4.056e+07   1.39e+07      2.920      0.004    1.33e+07    6.78e+07
released_day          3.399e+06   1.21e+07      0.280      0.779   -2.04e+07    2.72e+07
in_spotify_playlists  1.696e+07   1.21e+07      1.400      0.162   -6.84e+06    4.08e+07
in_spotify_charts     2.373e+08   3.01e+07      7.884      0.000    1.78e+08    2.96e+08
in_apple_playlists    9.891e+07   1.83e+07      5.401      0.000    6.29e+07    1.35e+08
in_apple_charts       2.012e+08   2.01e+07      9.989      0.000    1.62e+08    2.41e+08
in_deezer_playlists   7.714e+05   1.54e+07      0.050      0.960   -2.94e+07    3.1e+07
in_deezer_charts      5.344e+07   2.32e+07      2.299      0.022    7.77e+06    9.91e+07
in_shazam_charts     -2.35e+07   1.59e+07     -1.475      0.141   -5.48e+07    7.8e+06
bpm                  -1.006e+08   1.53e+07     -6.569      0.000   -1.31e+08   -7.05e+07
danceability_%       -6.143e+06   1.21e+07     -0.507      0.612   -2.99e+07    1.76e+07
valence_%            -1.553e+07   1.46e+07     -1.060      0.289   -4.43e+07    1.32e+07
energy_%             -8.526e+06    1.5e+07     -0.567      0.571    -3.8e+07     2.1e+07
acousticness_%       -2.61e+07    1.6e+07     -1.629      0.104   -5.76e+07    5.38e+06
instrumentalness_%     2.5e+07   1.54e+07      1.625      0.105   -5.23e+06    5.52e+07
liveness_%           -3.18e+06   1.22e+07     -0.261      0.794   -2.71e+07    2.07e+07
speechiness_%        -4.653e+06   1.22e+07     -0.383      0.702   -2.85e+07    1.92e+07
artist_count         -6.445e+06   1.23e+07     -0.524      0.600   -3.06e+07    1.77e+07
key_A#               -2.838e+06   1.53e+07     -0.186      0.853   -3.28e+07    2.71e+07
key_B                 1.423e+06   1.63e+07      0.087      0.931   -3.07e+07    3.35e+07
key_C#               -7.504e+05   1.75e+07     -0.043      0.966   -3.52e+07    3.37e+07
key_D                -3.09e+05   1.65e+07     -0.019      0.985   -3.27e+07    3.21e+07
key_D#                2.437e+06   1.38e+07      0.177      0.860   -2.47e+07    2.96e+07
key_E                 6.661e+06   1.54e+07      0.433      0.665   -2.35e+07    3.69e+07
key_F                -1.507e+07   1.65e+07     -0.911      0.363   -4.76e+07    1.74e+07
key_F#               -1.202e+07   1.56e+07     -0.771      0.441   -4.26e+07    1.86e+07
key_G                -1.733e+07   1.69e+07     -1.025      0.306   -5.05e+07    1.59e+07
```

```
key_G#              -8.416e+06   1.66e+07     -0.508      0.612    -4.1e+07    2.41e+07
mode_Minor           6.875e+06   1.26e+07      0.545      0.586   -1.79e+07    3.17e+07
==============================================================================
Omnibus:                      218.824   Durbin-Watson:                   2.035
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             1645.999
Skew:                           1.499   Prob(JB):                         0.00
Kurtosis:                      10.774   Cond. No.                         5.93
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

Cross-Validation

```
[90] from sklearn.model_selection import KFold
     from sklearn.model_selection import GridSearchCV
     from sklearn.pipeline import make_pipeline
     from sklearn.model_selection import cross_val_score
```

```
     # Perform hyperparameter tuning to find the best hyperparameters for the linear regression model.

     # k-fold Cross-validation
     # we cannot implement cross validation using statsmode
     # so to get cross validation score, we switch to sklearn's linear regression model
     # fit_intercept = True by default so no need to add constant term
     lm = LinearRegression()

     # fit skleanr linear regression on training set
     lm.fit(X_train, y_train)

     # used trained sklearn linear regression model "lm" to predict mpg for testing data
     predictions = lm.predict(X_test)

     R2 = cross_val_score(lm, X_train, y_train, scoring='r2', cv=10) #default 5-fold cross validation
```

```
                       'danceability_%', 'valence_%', 'energy_%', 'liveness_%', 'speechiness_%',
                       'artist_count',
                       'key_A#', 'key_B', 'key_C#', 'key_D', 'key_D#', 'key_E','key_F', 'key_F#', 'key_G', 'key_G#', 'mode_Minor',
                       'released_day']
```

```
[94] # Split dataset into features and target after dropping statistically insignificant variables
     # drop one of each one hot encoded columns to avoid perfect multi-linearity
     X_2 = df1.drop(['streams','track_name', 'artist(s)_name', 'release_date','key_A','mode_Major'],axis=1).drop(columns_to_drop, axis=1)
     y = df1['streams']
```

```
[95] # Split dataset into training and test sets.
     X_train_2, X_test_2, y_train, y_test = train_test_split(X_2, y, random_state=42, test_size=0.3)

     X_train_2.columns
```

```
Index(['released_year', 'released_month', 'in_spotify_playlists',
       'in_spotify_charts', 'in_apple_playlists', 'in_apple_charts',
       'in_deezer_charts', 'in_shazam_charts', 'bpm', 'acousticness_%',
       'instrumentalness_%'],
      dtype='object')
```

```
     # Scale the feature data.
     scaler = StandardScaler()

     X_train_2 = scaler.fit_transform(X_train_2)
     X_test_2 = scaler.transform(X_test_2)
```

```
[97] # Set up the environment to ignore warnings.
     import warnings
     warnings.filterwarnings("ignore")
```

```
[98] # add a constant term for intercept
     X_train_const_2 = sm.add_constant(X_train_2)

     # Create a linear regression model and train it with the training data.
     model2 = sm.OLS(y_train, X_train_const_2)
     result = model2.fit()
```

```
[99] print(result.summary(xname=['const',
                                 'released_year', 'released_month',
                                 'in_spotify_playlists', 'in_spotify_charts','in_apple_playlists', 'in_apple_charts', 'in_deezer_charts','in_shazam_charts',
                                 'bpm', 'acousticness_%', 'instrumentalness_%']))
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                streams   R-squared:                       0.716
Model:                            OLS   Adj. R-squared:                  0.710
Method:                 Least Squares   F-statistic:                     127.6
Date:                Wed, 13 Dec 2023   Prob (F-statistic):          2.90e-144
Time:                        22:15:52   Log-Likelihood:                -11869.
No. Observations:                 569   AIC:                         2.376e+04
Df Residuals:                     557   BIC:                         2.381e+04
Df Model:                          11
Covariance Type:            nonrobust
==========================================================================================
                             coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------------------
const                   4.519e+08   1.17e+07     38.498      0.000    4.29e+08    4.75e+08
released_year           4.566e+07   1.33e+07      3.433      0.001    1.95e+07    7.18e+07
released_month          4.289e+06   1.19e+07      0.360      0.719   -1.91e+07    2.77e+07
in_spotify_playlists     2.96e+08   1.83e+07     16.192      0.000     2.6e+08    3.32e+08
in_spotify_charts       -8.251e+07   1.77e+07     -5.047      0.000   -1.17e+08   -5.03e+07
```

```
grid_search = GridSearchCV(ridge, param_grid, scoring='r2', cv=5)
grid_search.fit(X_train, y_train)

# Get the best hyperparameters
best_alpha = grid_search.best_params_['alpha']

# Train the model with the best hyperparameters
best_ridge_model = Ridge(alpha=best_alpha)
best_ridge_model.fit(X_train, y_train)

# Evaluate the model on the validation set
y_val_pred = best_ridge_model.predict(X_test)
#mse = mean_squared_error(y_test, predictions)
r2 = r2_score(y_test, predictions)

# calculate adj r2
n = len(X_train)
p = len(X.columns)
adj_r2 = 1- ((1-r2) * (n-1)/(n-p-1))

print(f"Best alpha: {best_alpha}")
print(f"R-squared on validation set: {r2}")
print(f"Adjusted R-squared on validation set: {adj_r2}")
```

```
Best alpha: 10
R-squared on validation set: 0.6858362677990533
Adjusted R-squared on validation set: 0.6683178440703759
```

After attempting linear regression model with only selective statsitically significant variables, the R-squared of the model which in this case evaluates how much variation is accounted in the model did not change much. This tells us that linear regression is not a good model for predicting streams.

Logistic Regression

Since we have a lot of discrete variables in our data set, it makes sense that linear regression is not the ideal model. Instead we decided to try logistic regression for classification.

First, we need to define the binary outcome variable. In this case, we wanted to predict what makes a song a "top hit", which is the top 25%/first quartile of number of streams on Spotify.

Define Outcome Variable: Top Hits

```
[102] # let's use the top 25% streams as "meg hits"
      streamsQ1 = df1['streams'].quantile(0.75)
      streamsQ1
```

```
601863821.0
```

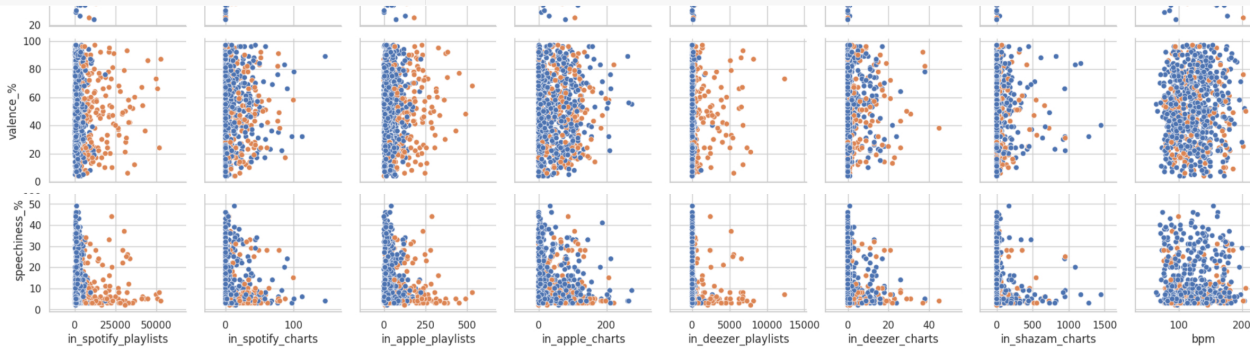
```
[103] # Create a new data frame for logistic regression
      df2 = df1.copy()

      # Create a new binary variable named "TopHits" where the value is 1 if streams is above top 25% streams.
      TopHits = np.where(df2['streams'] > streamsQ1, 1, 0)

      # Add column to dataframe.
```

```
df2['TopHits'] = TopHits
 29  key_C               813 non-null    uint8
 30  key_F#              813 non-null    uint8
 31  key_G               813 non-null    uint8
 32  key_G#              813 non-null    uint8
 33  mode_Major          813 non-null    uint8
 34  mode_Minor          813 non-null    uint8
 35  TopHits             813 non-null    int64
dtypes: datetime64[ns](1), float64(1), int64(19), object(2), uint8(13)
memory usage: 195.0+ KB
```

```
[105] # get a list of all columns that are numerical
numerical_variables = ['in_spotify_charts',
 'in_apple_playlists',
 'in_apple_charts',
 'in_deezer_playlists',
 'in_deezer_charts',
 'in_shazam_charts',
 'bpm',
 'danceability_%',
 'valence_%',
 'energy_%',
 'acousticness_%',
 'instrumentalness_%',
 'liveness_%',
 'speechiness_%']
numerical_variables
```

```
['in_spotify_playlists',
 'in_spotify_charts',
 'in_apple_playlists',
 'in_apple_charts',
 'in_deezer_playlists',
 'in_deezer_charts',
 'in_shazam_charts',
 'bpm',
 'danceability_%',
 'valence_%',
 'energy_%',
 'acousticness_%',
 'instrumentalness_%',
 'liveness_%',
 'speechiness_%']
```

```
[106] # let's visualize our data distribution using paired plot for all continuous data
sns.pairplot(df2,
             x_vars = numerical_variables,
             y_vars = numerical_variables,
             hue="TopHits")
```



Fit Logistic Regression ML Model with One-Hot

```
[107] # Import the necessary libraries for data processing and machine learning.
import numpy as np  # For numerical operations.
import pandas as pd  # For data manipulation.
import statsmodels.api as sm
import statsmodels.formula.api as smf
from statsmodels.api import OLS

# sklearn libraries for preprocessing, model selection, and metrics.
from sklearn.preprocessing import StandardScaler  # To standardize features.
from sklearn.linear_model import LogisticRegression  # Logistic regression model.
from sklearn.model_selection import train_test_split  # To split the data into training and test sets.
from sklearn.model_selection import KFold  # For K-fold cross-validation.
from sklearn.metrics import make_scorer, confusion_matrix  # For creating custom metrics and confusion matrices.
from sklearn.model_selection import learning_curve  # To generate a learning curve.

# Libraries for visualization.
import matplotlib.pyplot as plt  # For plotting.
import seaborn as sns  # For attractive and informative statistical graphics.
sns.set_style('whitegrid')  # Set the style of seaborn plots to 'whitegrid'.

# This command makes sure that plots are displayed inline in the Jupyter Notebook.
%matplotlib inline
```

```
[108] # Split dataset into features and target
# delete 'key_A','mode_Major' to avoid perfect multilinearity
X = df2.drop(['TopHits','track_name', 'artist(s)_name', 'release_date','key_A','mode_Major'],axis=1)
y = df2['TopHits']
```

```
[109] # Check X to make sure the right features are selected
X
```

	artist_count	released_year	released_month	released_day	in_spotify_playlists	in_spotify_charts	in_apple_playlists	in_apple_charts	in_deezer_playlists	in_deezer_charts	...	key_B	key_C#	key_D	key_D#	key_E	key_F	key_F#
0	2	2023	7	14	553	147	43	263	45	10	...	1	0	0	0	0	0	0
1	1	2023	3	23	1474	48	48	126	58	14	...	0	1	0	0	0	0	0
2	1	2023	6	30	1397	113	94	207	91	14	...	0	0	0	0	0	1	0
3	1	2019	8	23	7858	100	116	207	125	12	...	0	0	0	0	0	0	0
4	1	2023	5	18	3133	50	84	133	87	15	...	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
948	1	2022	11	3	953	0	61	13	37	1	...	0	0	0	0	0	0	0

```
'key_G#',
'mode_Minor']
```

```
# Split dataset into training and test sets.
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42, test_size=0.3)

# Scale the feature data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Add a constant term to the independent variables
# this is to avoid data leakage due to scaling feature data
# to allow for a baseline probability (intercept)
# this makes our model more flexible and interpretable
X_train_const = sm.add_constant(X_train)

# Fit logistic regression model
model = sm.Logit(y_train, X_train_const)
result = model.fit()

# Print the model summary and accuracy
print(result.summary(xname=['const','artist_count',
 'released_year',
 'released_month',
 'released_day',
 'in_spotify_playlists',
 'in_spotify_charts',
 'in_apple_playlists',
 'in_apple_charts',
 'in_deezer_playlists',
 'in_deezer_charts',
 'in_shazam_charts',
 'bpm',
```

```
    'danceability_%',
    'valence_%',
    'energy_%',
    'acousticness_%',
    'instrumentalness_%',
    'liveness_%',
    'speechiness_%',
    'key_A#',
    'key_B',
    'key_C#',
    'key_D',
    'key_D#',
    'key_E',
    'key_F',
    'key_F#',
    'key_G',
    'key_G#',
    'mode_Minor']))
```

```
Optimization terminated successfully.
         Current function value: 0.201593
         Iterations 10
                           Logit Regression Results
==============================================================================
Dep. Variable:                TopHits   No. Observations:                  569
Model:                          Logit   Df Residuals:                      538
Method:                           MLE   Df Model:                           30
Date:                Wed, 13 Dec 2023   Pseudo R-squ.:                 0.6249
Time:                        22:17:35   Log-Likelihood:                -114.71
```

```
# Perform the K-Fold cross-validation
for train_idx, test_idx in kfold.split(X):
    X_train, X_test = X.values[train_idx], X.values[test_idx]
    y_train, y_test = y.values[train_idx], y.values[test_idx]

    # Scale the features within the train/test split to prevent data leakage
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    # Train the model and make predictions
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)

    cm = confusion_matrix(y_test, predictions)

    if cm.shape == (1, 1):
      # If the confusion matrix has only one element, set TN, FP, FN, TP accordingly
      TN, FP, FN, TP = 0, 0, 0, cm[0, 0]

    else:
      # Compute the accuracy for the current fold
      TN = cm[0][0]
      FP = cm[0][1]
      FN = cm[1][0]
      TP = cm[1][1]

    total = float(TN + FP + FN + TP)
    ACC = (TP + TN) / float(total)

    # Store the accuracy in the array
    accuracy[np_idx] = ACC*100
    np_idx += 1

    # Print the accuracy for the current fold
    print("Fold {}: Accuracy: {}%".format(np_idx, round(ACC*100,3)))

# Print the average accuracy across all folds
print("Average Score for logistic regression with one-hot encoding: {}%({}%)".format(round(np.mean(accuracy),3),round(np.std(accuracy),3)))
```

```
Fold 1: Accuracy: 85.366%
Fold 2: Accuracy: 89.024%
Fold 3: Accuracy: 93.902%
Fold 4: Accuracy: 90.123%
Fold 5: Accuracy: 80.247%
Fold 6: Accuracy: 87.654%
Fold 7: Accuracy: 85.185%
Fold 8: Accuracy: 91.358%
Fold 9: Accuracy: 95.062%
Fold 10: Accuracy: 95.062%
Average Score for logistic regression with one-hot encoding: 89.298%(4.579%)
```

The logistic regression model is around 90% accurate at predicting if a song is a "top hit", a better accuracy score than linear regression.

Random Forest Regression

We also wanted to attempt a non-linear machine learning model and compare results.

```
[114] # Create a new data frame for random forest regression
```

```
 21  instrumentalness_%     813 non-null    float64
 22  liveness_%             813 non-null    float64
 23  speechiness_%          813 non-null    float64
 24  release_date           813 non-null    datetime64[ns]
 25  avg_streams_per_month  813 non-null    float64
dtypes: datetime64[ns](1), float64(10), int64(13), object(2)
memory usage: 203.8+ KB
```

Perform Predictions

Now we are ready to perform random forest regression

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import GridSearchCV


X = df4.drop(['track_name', 'artist(s)_name', 'release_date','streams'],axis=1)
y = df4['streams']

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

rf_reg = RandomForestRegressor()
rf_reg.fit(X_train, y_train)

# Predictions on the validation set first
y_val_pred = rf_reg.predict(X_val)

mse = mean_squared_error(y_val, y_val_pred)
mae = mean_absolute_error(y_val, y_val_pred)
r2 = r2_score(y_val, y_val_pred)

print(f"Mean Squared Error (MSE): {mse}")
print(f"Mean Absolute Error (MAE): {mae}")
print(f"R^2 Score: {r2}")
```

```
Mean Squared Error (MSE): 3.088075954331991e+16
Mean Absolute Error (MAE): 101494879.90606557
R^2 Score: 0.8739800716896832
```

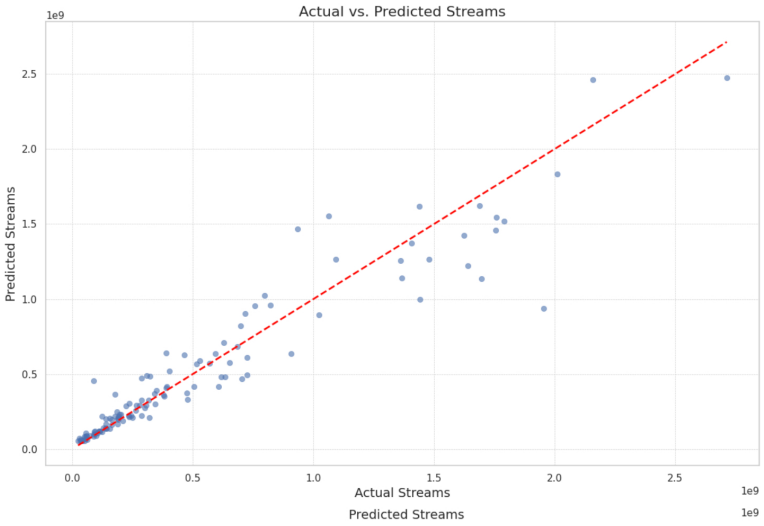
```
[119] # Now, after predictions on the validation set, now we predict on the test set
    y_test_pred = rf_reg.predict(X_test)

    mse_test = mean_squared_error(y_test, y_test_pred)
    mae_test = mean_absolute_error(y_test, y_test_pred)
    r2_test = r2_score(y_test, y_test_pred)

    print(f"Test Set Evaluation:")
    print(f"Mean Squared Error (MSE): {mse_test}")
    print(f"Mean Absolute Error (MAE): {mae_test}")
    print(f"R^2 Score: {r2_test}")

    # compute the level of importance/influence for each variable
    feature_importances = rf_reg.feature_importances_
    features_df = pd.DataFrame({
        'Feature': X.columns.to_list(),
        'Importance': feature_importances
    })
```

data compared to the validation data.

This suggests that the model's performance is better on the test set or that the model generalizes well to new, unseen data, which contradicts what the MAE says about our model

While R^2 is a useful metric, it's important to consider other evaluation metrics and conduct a comprehensive analysis of the model's performance.

```
[120] #Now, let us plot the Actual vs. Predicted Streams to see how our model has performed
      plt.figure(figsize=(12, 8))
      sns.scatterplot(x=y_test, y=y_test_pred, alpha=0.6, edgecolor=None)
      plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red', linestyle='--', lw=2)
      plt.xlabel('Actual Streams', fontsize=14)
      plt.ylabel('Predicted Streams', fontsize=14)
      plt.title('Actual vs. Predicted Streams', fontsize=16)
      plt.grid(True, which='both', linestyle='--', linewidth=0.5)
      plt.tight_layout()
      plt.show()

      #Here, especially for the less listened to songs, we see that we have reached a relatively decent accuracy based off of looking at this graphs.
      #Note the numbers at the axises are in billions
```



Here, we have reached a relatively decent accuracy based off of looking at this graphs.

However, we do have some outliers on the residual plot. The shape of the residual plot is also fan-shaped. This pattern may indicate that there is relationships that are not captured by the model.

**Note the numbers at the x-axis are in billions and the numbers on the y-axis are in millions.

PART 5: PRESCRIPTIVE ANALYTICS, DATA VISUALIZATION AND TAKEAWAYS

- We could experiment with other machine learning models like Gradient Boosting, XGBoost, or neural networks to compare performance.
- We could incorporate sentiment analysis of social media buzz around songs or artists to see if it correlates with streaming numbers.
- We could explore differences in streaming behavior across different geographic regions or demographic groups.
- We could utilize grid search or randomized search for hyperparameter tuning of the random forest model to further optimize its performance.

To strengthen the credibility of our analysis results (and in extension, our recommendations), we explored previous research done on music popularity. In addition, we wanted to know where our current results stand amongst what is currently known. To our surprise, Random Forest model is consistently the highest performing (most accurate) machine learning algorithm for music popularity predictions, compared to other approaches such as: Linear Regression, Logistic Regression, Neural Networks, Lasso Regression, Deep Learning, Boosting Tree, etc. (Arora, Rani, & Saxena, 2022; Colley et al., 2022; Essa et al., 2022; Gao, 2021; Khan et al., 2022). Several researchers were even successful in creating a Random Forest model with a 95.37% accuracy rate (Gulmatico et al., 2022). Based on previous research, we believe our Random Forest model has potential, and that its superior performance against our Linear Regression model isn't unusual.

In reviewing further research, it is evident that many external factors have profound effects on music streaming. Research shows that there is a strong correlation between numbers of followers and royalties from music streaming specifically. Set in the context of the high level of income precarity being experienced amongst independent artists, this finding suggests that a positive co-evolution is occurring between social media platforms and music streaming with potentially significant increases in artist royalty incomes resulting from increases in followers (Watson, A., Watson, J. B., & Tompkins, L., 2023). This combined with the importance of being in not only Spotify playlists but also playlists on other music streaming platforms that we've found through our model suggests that -- to achieve streaming success, network and marketing is a key factor. Based on our data analysis and industry research, we propose the following recommendations for musicians:

Leverage Social Media Platforms for Increased Followership:

Engage actively on social media platforms to increase your followership. As evidenced by previous research, a positive correlation exists between the number of followers and artist royalty incomes from music streaming. Utilize platforms such as Instagram, Twitter, and Facebook to connect with your audience, share updates, and promote your music. Optimize Presence on Spotify Playlists:

Our model highlights the importance of being featured in Spotify playlists for increased music streaming. Work towards getting your songs included in relevant playlists to enhance discoverability. Collaborate with curators, promote your music to influencers, and ensure your songs align with the themes of popular playlists.

Diversify Across Multiple Music Streaming Platforms:

Extend your reach beyond Spotify and explore other music streaming platforms. Our analysis suggests that being featured in playlists on various platforms contributes to overall streaming success. Platforms like Apple Music, Deezer, and YouTube Music offer additional opportunities to reach diverse audiences. Collaborate with Other Artists and Curators:

Network within the music industry by collaborating with other artists and connecting with playlist curators. Cross-promotions and collaborative projects can broaden your audience and increase your visibility. Building relationships with curators enhances the chances of getting featured in curated playlists. Strategic Marketing Campaigns:

Develop and execute strategic marketing campaigns to promote your music. Utilize online advertising, influencer partnerships, and content marketing to reach a wider audience. Tailor your campaigns to target specific demographics and genres to maximize impact.

Engage with Fans and Collect Feedback:

Foster a strong connection with your fanbase by actively engaging with them on social media and other platforms. Encourage feedback, conduct polls, and consider fan preferences in your music production. Building a loyal fan community contributes to sustained streaming success. Monitor Streaming Analytics and Iterate Strategies:

Regularly analyze streaming analytics to assess the impact of your efforts. Monitor which strategies contribute most to streaming numbers and adjust your approach accordingly. Iterative improvements based on data insights are crucial for long-term success.

In conclusion, success in music streaming is not solely determined by the quality of the music but also by strategic networking and marketing

```
3   released_year          953 non-null    int64
```