We will be using a dataset from Kaggle 'Spotify Tracks DB' that contains approximately 232,000 tracks and their attributes to train several machine learning models in order to find the common threads between popular songs.

In [1]:
```python
import warnings
warnings.filterwarnings('ignore')
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

In [2]:
```python
df = pd.read_csv("SpotifyFeatures.csv")
df.head()
```

Out[2]:

	genre	artist_name	track_name	track_id	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo
0	Movie	Henri Salvador	C'est beau de faire un Show	0BRjO6ga9RKCKjfDqeFgWV	0	0.611	0.389	99373	0.910	0.000	C#	0.3460	-1.828	Major	0.0525	166.96
1	Movie	Martin & les fées	Perdu d'avance (par Gad Elmaleh)	0BjC1NfoEOOusryehmNudP	1	0.246	0.590	137373	0.737	0.000	F#	0.1510	-5.559	Minor	0.0868	174.00
2	Movie	Joseph Williams	Don't Let Me Be Lonely Tonight	0CoSDzoNIKCRs124s9uTVy	3	0.952	0.663	170267	0.131	0.000	C	0.1030	-13.879	Minor	0.0362	99.48
3	Movie	Henri Salvador	Dis-moi Monsieur Gordon Cooper	0Gc6TVm52BwZD07Ki6tIvf	0	0.703	0.240	152427	0.326	0.000	C#	0.0985	-12.178	Major	0.0395	171.75
4	Movie	Fabien Nataf	Ouverture	0IuslXpMROHdEPvSl1fTQK	4	0.950	0.331	82625	0.225	0.123	F	0.2020	-21.150	Major	0.0456	140.57

In [3]:
```python
df.describe()
```

Out[3]:

	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	liveness	loudness	speechiness	tempo	valence
count	232725.000000	232725.000000	232725.000000	2.327250e+05	232725.000000	232725.000000	232725.000000	232725.000000	232725.000000	232725.000000	232725.000000
mean	41.127502	0.368560	0.554364	2.351223e+05	0.570958	0.148301	0.215009	-9.569885	0.120765	117.666585	0.454917
std	18.189948	0.354768	0.185608	1.189359e+05	0.263456	0.302768	0.198273	5.998204	0.185518	30.898907	0.260065
min	0.000000	0.000000	0.056900	1.538700e+04	0.000020	0.000000	0.009670	-52.457000	0.022200	30.379000	0.000000
25%	29.000000	0.037600	0.435000	1.828570e+05	0.385000	0.000000	0.097400	-11.771000	0.036700	92.959000	0.237000
50%	43.000000	0.232000	0.571000	2.204270e+05	0.605000	0.000044	0.128000	-7.762000	0.050100	115.778000	0.444000
75%	55.000000	0.722000	0.692000	2.657680e+05	0.787000	0.035800	0.264000	-5.501000	0.105000	139.054000	0.660000
max	100.000000	0.996000	0.989000	5.552917e+06	0.999000	0.999000	1.000000	3.744000	0.967000	242.903000	1.000000

In [4]:
```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 232725 entries, 0 to 232724
Data columns (total 18 columns):
 #   Column            Non-Null Count   Dtype  
---  ------            --------------   -----  
 0   genre             232725 non-null  object 
 1   artist_name       232725 non-null  object 
 2   track_name        232724 non-null  object 
 3   track_id          232725 non-null  object 
 4   popularity        232725 non-null  int64  
 5   acousticness      232725 non-null  float64
 6   danceability      232725 non-null  float64
 7   duration_ms       232725 non-null  int64  
 8   energy            232725 non-null  float64
 9   instrumentalness  232725 non-null  float64
 10  key               232725 non-null  object 
 11  liveness          232725 non-null  float64
 12  loudness          232725 non-null  float64
 13  mode              232725 non-null  object 
 14  speechiness       232725 non-null  float64
 15  tempo             232725 non-null  float64
 16  time_signature    232725 non-null  object 
 17  valence           232725 non-null  float64
dtypes: float64(9), int64(2), object(7)
memory usage: 32.0+ MB
```

We once again see that we have 232,725 tracks in the dataset with both categorical and numerical columns. In order to use the information from the categorical columns ('genre', 'artist_name', 'track_name', 'track_id', 'key', 'mode', 'time_signature') we will either need to represent them numerically by feature engineering or drop them to be able to train the models.

In [5]:

```python
#looking at different values contained within columns
for col in df.columns:
    print(f"Column: {col}")
    print(df[col].value_counts())
    print("--------------------")
```

```
Column: genre
genre
Comedy              9681
Soundtrack          9646
Indie               9543
Jazz                9441
Pop                 9386
Electronic          9377
Children’s Music    9353
Folk                9299
Hip-Hop             9295
Rock                9272
Alternative         9263
Classical           9256
Rap                 9232
World               9096
Soul                9089
Blues               9023
R&B                 8992
Anime               8936
Reggaeton           8927
Ska                 8874
```

```
Reggae              8771
Dance               8701
Country             8664
Opera               8280
Movie               7806
Children's Music    5403
A Capella            119
Name: count, dtype: int64
--------------------
Column: artist_name
artist_name
Giuseppe Verdi              1394
Giacomo Puccini             1137
Kimbo Children's Music       971
Nobuo Uematsu                825
Richard Wagner               804
                            ... 
Zubin Mehta                    1
Shawn Lane                     1
Claudio Arrau                  1
Charles Daellenbach            1
Jr Thomas & The Volcanos       1
Name: count, Length: 14564, dtype: int64
--------------------
Column: track_name
track_name
Home                                                               100
You                                                                 71
Intro                                                               69
Stay                                                                63
Wake Up                                                             59
                                                                  ... 
Siegfried / Zweiter Aufzug: Vorspiel                                 1
Die Walküre / Zweiter Aufzug: "Siegmund! Sieh auf mich!"             1
Puccini: Tosca, Act 1: "Ah! Finalmente!" (Angelotti) [Live]          1
Harpsichord Sonata No. 11 in F Major: I. Moderato                    1
You Don't Have To Hurt No More                                       1
Name: count, Length: 148614, dtype: int64
--------------------
Column: track_id
track_id
3R73Y7X53MIQZWnKloWq5i    8
0wY9rA9fJkuESyYm9uzVK5    8
6sVQNUvcVFTXvlk3ec0ngd    8
0UE0RhnRaEYsiYgXpyLoZc    8
6AIte2Iej1QKlaofpjCzW1    8
                         ..
2sERVoTuQG14MKze0PuLZd    1
2rQCKDafhIA6GKPGiZsyfI    1
150kk5S2hUULZD4yApAwSH    1
7cq0WqlooYmk0u2EQeA85S    1
34XO9RwPMKjbvRry54QzWn    1
Name: count, Length: 176774, dtype: int64
--------------------
Column: popularity
popularity
0      6312
50     5415
```

```
53     5414
51     5401
52     5342
       ...
96        8
94        7
99        4
98        3
100       2
Name: count, Length: 101, dtype: int64
--------------------
Column: acousticness
acousticness
0.995000    851
0.994000    701
0.992000    682
0.993000    646
0.991000    597
           ...
0.000038      1
0.000008      1
0.000085      1
0.000006      1
0.000007      1
Name: count, Length: 4734, dtype: int64
--------------------
Column: danceability
danceability
0.5970    558
0.5470    544
0.5890    542
0.6100    542
0.6220    540
         ...
0.0922      1
0.9700      1
0.0868      1
0.0572      1
0.0570      1
Name: count, Length: 1295, dtype: int64
--------------------
Column: duration_ms
duration_ms
240000    138
180000    120
192000    115
216000     99
200000     85
         ...
200699      1
297501      1
255965      1
233502      1
282447      1
Name: count, Length: 70749, dtype: int64
--------------------
Column: energy
energy
```

```
0.72100    417
0.67500    403
0.72000    392
0.68600    389
0.73800    389
          ...
0.00641      1
0.00714      1
0.00229      1
0.00599      1
0.00633      1
Name: count, Length: 2517, dtype: int64
--------------------
Column: instrumentalness
instrumentalness
0.00000    79236
0.91200      235
0.91000      230
0.92300      222
0.91800      222
           ...
0.06460        1
0.00966        1
0.99800        1
0.99900        1
0.00888        1
Name: count, Length: 5400, dtype: int64
--------------------
Column: key
key
C     27583
G     26390
D     24077
C#    23201
A     22671
F     20279
B     17661
E     17390
A#    15526
F#    15222
G#    15159
D#     7566
Name: count, dtype: int64
--------------------
Column: liveness
liveness
0.1110    2860
0.1100    2702
0.1080    2608
0.1090    2537
0.1070    2451
          ...
0.0180       1
0.0155       1
0.0240       1
0.0105       1
0.0189       1
Name: count, Length: 1732, dtype: int64
```

```
--------------------
Column: loudness
loudness
-5.318     57
-5.460     52
-5.131     51
-5.428     51
-5.480     50
           ..
-26.914     1
-35.951     1
-30.688     1
-29.939     1
-18.792     1
Name: count, Length: 27923, dtype: int64
--------------------
Column: mode
mode
Major    151744
Minor     80981
Name: count, dtype: int64
--------------------
Column: speechiness
speechiness
0.0374    663
0.0332    654
0.0337    652
0.0363    650
0.0343    642
         ...
0.5840      1
0.7160      1
0.6880      1
0.9670      1
0.5860      1
Name: count, Length: 1641, dtype: int64
--------------------
Column: tempo
tempo
120.016    61
100.003    60
100.014    60
120.008    59
120.003    59
           ..
173.292     1
177.825     1
76.708      1
64.120      1
175.666     1
Name: count, Length: 78512, dtype: int64
--------------------
Column: time_signature
time_signature
4/4    200760
3/4     24111
5/4      5238
1/4      2608
```

```
0/4         8
Name: count, dtype: int64
--------------------
Column: valence
valence
0.9610    479
0.9620    403
0.9630    368
0.3700    363
0.3580    363
         ...
0.0205      1
0.9950      1
0.0193      1
0.0247      1
0.0248      1
Name: count, Length: 1692, dtype: int64
--------------------
```

There are a couple things that stand out in the value counts of the columns. First one is that we have the "Children's Music" genre showing up twice and we have duplicated values in the track_id column

Addressing "Children's Music" Character Discrepancy

In [6]:
```python
df['genre'].value_counts()
```

Out[6]:
```
genre
Comedy              9681
Soundtrack          9646
Indie               9543
Jazz                9441
Pop                 9386
Electronic          9377
Children’s Music    9353
Folk                9299
Hip-Hop             9295
Rock                9272
Alternative         9263
Classical           9256
Rap                 9232
World               9096
Soul                9089
Blues               9023
R&B                 8992
Anime               8936
Reggaeton           8927
Ska                 8874
Reggae              8771
Dance               8701
Country             8664
Opera               8280
Movie               7806
Children's Music    5403
A Capella            119
Name: count, dtype: int64
```

There are 2 types of "Children's Music" values in the genres due to the character used for apostrophe. Since both of these values are meant to show the same thing we need to merge them and achieve consistency.

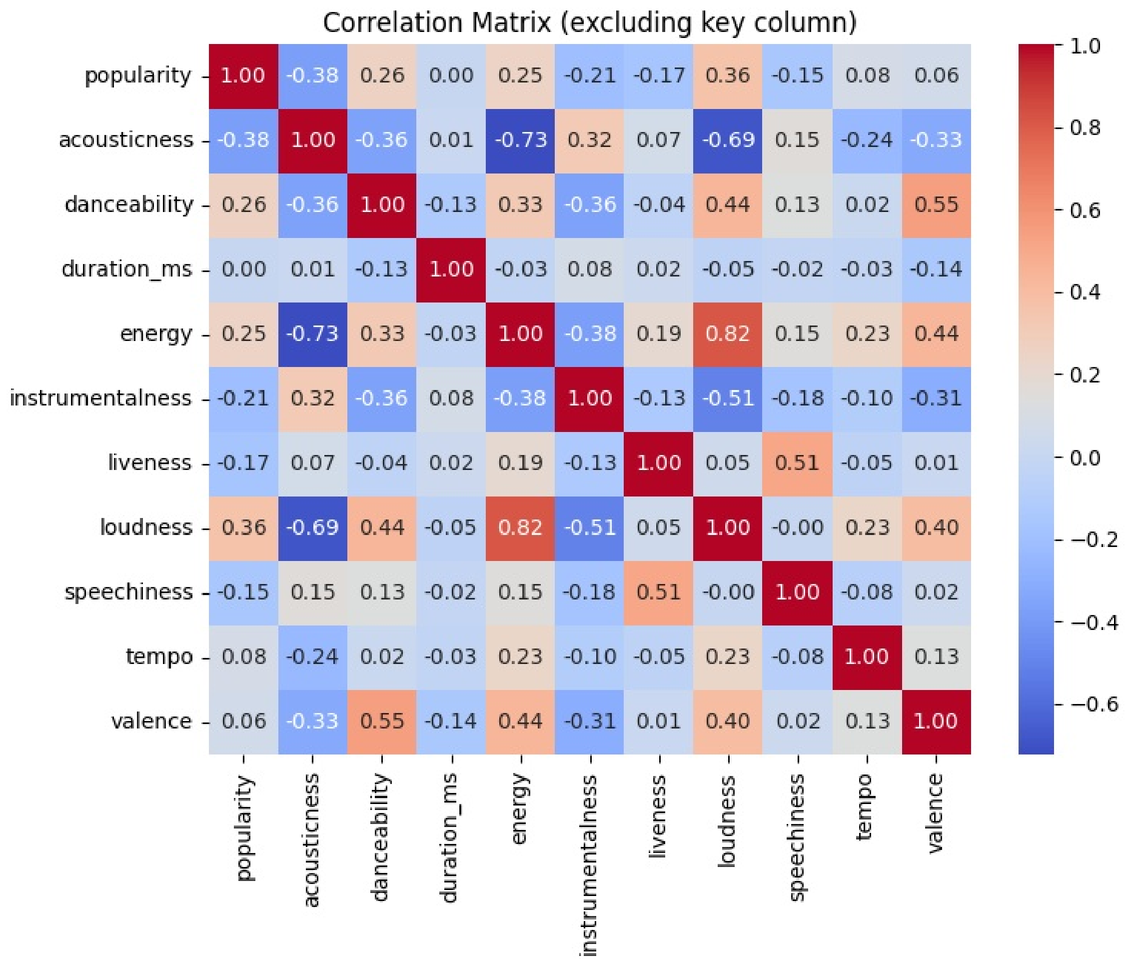
In [7]:
```
df.loc[df['genre']=="Children’s Music",'genre']="Children's Music"
```

In [8]:
```
#verifying that the issue has been resolved
df['genre'].value_counts()
```

Out[8]:
```
genre
Children's Music    14756
Comedy               9681
Soundtrack           9646
Indie                9543
Jazz                 9441
Pop                  9386
Electronic           9377
Folk                 9299
Hip-Hop              9295
Rock                 9272
Alternative          9263
Classical            9256
Rap                  9232
World                9096
Soul                 9089
Blues                9023
R&B                  8992
Anime                8936
Reggaeton            8927
Ska                  8874
Reggae               8771
Dance                8701
Country              8664
Opera                8280
Movie                7806
A Capella             119
Name: count, dtype: int64
```

In [9]:
```
df_no_key = df.drop(columns=['genre','artist_name','track_name','track_id','key','mode','time_signature'])
correlation_matrix = df_no_key.corr()
print(correlation_matrix)
plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix (excluding key column)')
plt.show()
```

```
                  popularity  acousticness  danceability  duration_ms  \
popularity          1.000000     -0.381295      0.256564     0.002348
acousticness       -0.381295      1.000000     -0.364546     0.011203
danceability        0.256564     -0.364546      1.000000    -0.125781
duration_ms         0.002348      0.011203     -0.125781     1.000000
energy              0.248922     -0.725576      0.325807    -0.030550
instrumentalness   -0.210983      0.316154     -0.364941     0.076021
liveness           -0.167995      0.069004     -0.041684     0.023783
loudness            0.363011     -0.690202      0.438668    -0.047618
speechiness        -0.151076      0.150935      0.134560    -0.016171
tempo               0.081039     -0.238247      0.021939    -0.028456
valence             0.060076     -0.325798      0.547154    -0.141811

                    energy  instrumentalness  liveness  loudness  speechiness  \
popularity        0.248922         -0.210983 -0.167995  0.363011    -0.151076
acousticness     -0.725576          0.316154  0.069004 -0.690202     0.150935
danceability      0.325807         -0.364941 -0.041684  0.438668     0.134560
duration_ms      -0.030550          0.076021  0.023783 -0.047618    -0.016171
energy            1.000000         -0.378957  0.192801  0.816088     0.145120
instrumentalness -0.378957          1.000000 -0.134198 -0.506320    -0.177147
liveness          0.192801         -0.134198  1.000000  0.045686     0.510147
loudness          0.816088         -0.506320  0.045686  1.000000    -0.002273
speechiness       0.145120         -0.177147  0.510147 -0.002273     1.000000
tempo             0.228774         -0.104133 -0.051355  0.228364    -0.081541
valence           0.436771         -0.307522  0.011804  0.399901     0.023842

                     tempo   valence
popularity        0.081039  0.060076
acousticness     -0.238247 -0.325798
danceability      0.021939  0.547154
duration_ms      -0.028456 -0.141811
energy            0.228774  0.436771
instrumentalness -0.104133 -0.307522
liveness         -0.051355  0.011804
loudness          0.228364  0.399901
speechiness      -0.081541  0.023842
tempo             1.000000  0.134857
valence           0.134857  1.000000
```


Correlation Matrix (excluding key column)
popularity
acousticness
danceability
duration_ms
energy
instrumentalness
liveness
loudness
speechiness
tempo
valence
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6

Missing Values

In [10]:
```
#checking for missing values
df.isna().sum()
```

Out[10]:
```
genre               0
artist_name         0
track_name          1
track_id            0
popularity          0
acousticness        0
danceability        0
duration_ms         0
energy              0
instrumentalness    0
key                 0
liveness            0
loudness            0
mode                0
speechiness         0
tempo               0
time_signature      0
valence             0
dtype: int64
```

We don't have any missing values in our columns so we will move onto check for duplicated rows.

Addressing Duplicated Tracks

We need to take a look and find all duplicated tracks by using their unique id numbers.

In [11]:
```
df['track_name'].fillna('Unknown', inplace=True)
```

In [12]:
```
df.isna().sum()
```

Out[12]:
```
genre               0
artist_name         0
track_name          0
track_id            0
popularity          0
acousticness        0
danceability        0
duration_ms         0
energy              0
instrumentalness    0
key                 0
liveness            0
loudness            0
mode                0
speechiness         0
tempo               0
time_signature      0
valence             0
dtype: int64
```

In [13]:
```
df[df['track_id'].duplicated()]
```

Out[13]:

	genre	artist_name	track_name	track_id	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechine
1348	Alternative	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.07160	0.710	217813	0.710	0.000001	C	0.2060	-2.474	Major	0.05
1385	Alternative	Frank Ocean	Seigfried	1BViPjTT585XAhkUUrkts0	61	0.97500	0.377	334570	0.255	0.000208	E	0.1020	-11.165	Minor	0.03
1452	Alternative	Frank Ocean	Bad Religion	2pMPWE7PJH1PizfgGRMnR9	56	0.77900	0.276	175453	0.358	0.000003	A	0.0728	-7.684	Major	0.04
1554	Alternative	Steve Lacy	Some	4riDfclV7kPDT9D58FpmHd	58	0.00548	0.784	118393	0.554	0.254000	G	0.0995	-6.417	Major	0.03
1634	Alternative	tobi lou	Buff Baby	1F1QmI8TMHir9SUFrooq5F	59	0.19000	0.736	215385	0.643	0.000000	F	0.1060	-8.636	Major	0.04
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
232715	Soul	Emily King	Down	5cA0vB8c9FMOVDWyJHgf26	42	0.55000	0.394	281853	0.346	0.000002	E	0.1290	-13.617	Major	0.06
232718	Soul	Muddy Waters	I Just Want To Make Love To You - Electric Mud...	2HFczeynfKGiM9KF2z2K7K	43	0.01360	0.294	258267	0.739	0.004820	C	0.1380	-7.167	Major	0.04
232720	Soul	Slave	Son Of Slide	2XGLdVl7IGeq8ksM6AI7jT	39	0.00384	0.687	326240	0.714	0.544000	D	0.0845	-10.626	Major	0.03
232722	Soul	Muddy Waters	(I'm Your) Hoochie Coochie Man	2ziWXUmQLrXTiYjCg2fZ2t	47	0.90100	0.517	166960	0.419	0.000000	D	0.0945	-8.282	Major	0.14
232723	Soul	R.LUM.R	With My Words	6EFsue2YbIG4Qkq8Zr9Rir	44	0.26200	0.745	222442	0.704	0.000000	A	0.3330	-7.137	Major	0.14

55951 rows × 18 columns

We have 55,951 duplicated rows that we need to address. Before we can address these duplications though we need to see what the cause of the duplicates are.

In [14]:
```python
#checking rows for duplicated ids to see differences
df[df['track_id']=='6iOvnACn4ChlAw4lWUU4dd']
```

Out[14]:

	genre	artist_name	track_name	track_id	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness
257	R&B	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	0.71	217813	0.71	0.000001	C	0.206	-2.474	Major	0.0579
1348	Alternative	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	0.71	217813	0.71	0.000001	C	0.206	-2.474	Major	0.0579
77710	Children's Music	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	0.71	217813	0.71	0.000001	C	0.206	-2.474	Major	0.0579
93651	Indie	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	0.71	217813	0.71	0.000001	C	0.206	-2.474	Major	0.0579
113770	Pop	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	0.71	217813	0.71	0.000001	C	0.206	-2.474	Major	0.0579

In [15]:
```python
df[df['track_id']=='2XGLdVl7lGeq8ksM6Al7jT']
```

Out[15]:

	genre	artist_name	track_name	track_id	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	te
179212	Jazz	Slave	Son Of Slide	2XGLdVl7lGeq8ksM6Al7jT	39	0.00384	0.687	326240	0.714	0.544	D	0.0845	-10.626	Major	0.0316	115
232720	Soul	Slave	Son Of Slide	2XGLdVl7lGeq8ksM6Al7jT	39	0.00384	0.687	326240	0.714	0.544	D	0.0845	-10.626	Major	0.0316	115

In [16]:
```python
df[df['track_id']=='2HFczeynfKGiM9KF2z2K7K']
```

Out[16]:

	genre	artist_name	track_name	track_id	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	t
48555	Blues	Muddy Waters	I Just Want To Make Love To You - Electric Mud...	2HFczeynfKGiM9KF2z2K7K	35	0.0136	0.294	258267	0.739	0.00482	C	0.138	-7.167	Major	0.0434	17
232718	Soul	Muddy Waters	I Just Want To Make Love To You - Electric Mud...	2HFczeynfKGiM9KF2z2K7K	43	0.0136	0.294	258267	0.739	0.00482	C	0.138	-7.167	Major	0.0434	17

We see that most of the attributes of the duplicated songs are the same except for 'popularity' and 'genre'. The 'popularity' column can be aggregated since it is a numerical column but the categorical column of 'genre' is a little bit trickier. What makes the most sense in this case would be to create different columns with the genre names and display with binary values whether a song belongs to that genre or not.

In [17]:
```python
#generating a list with the genre names
genre_list = list(df['genre'].unique())
```

In [18]:
```python
#creating the genre columns using the genre list
for genre in genre_list:
    df[genre] = (df['genre']==genre).astype('int')
```

In [19]:
```python
#grouping by track_id number to get rid of duplicates and keeping the maximum values in each column.
```

```python
df=df.groupby(['track_id']).max()
```

Above, we created the genre columns and merged the duplicated values keeping the maximum value in each column. This makes sense since the track that is being listened to is the same one. For example, if a track had popularity scores of 15, 25, 38 and 42 in its duplicated rows, we are keeping the best value of 42 by taking the max.

In [20]:
```python
#removing redundant genre column
df.drop('genre', axis=1, inplace=True)
df.head()
```

Out[20]:

track_id	artist_name	track_name	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	...	Pop	Reggae	Reggaeton	Jazz	Rock	Ska
00021Wy6AyMbLP2tqij86e	Capcom Sound Team	Zangief's Theme	13	0.234	0.617	169173	0.862	0.976000	G	0.1410	...	0	0	0	0	0	0
000CzNKC8PEt1yC3L8dqwV	Henri Salvador	Coeur Brisé à Prendre - Remastered	5	0.249	0.518	130653	0.805	0.000000	F	0.3330	...	0	0	0	0	0	0
000DfZJww8KiixTKuk9usJ	Mike Love	Earthlings	30	0.366	0.631	357573	0.513	0.000004	D	0.1090	...	0	1	0	0	0	0
000EWWBkYaREzsBplYjUag	Don Philippe	Fewerdolr	39	0.815	0.768	104924	0.137	0.922000	C#	0.1130	...	0	0	0	1	0	0
000xQL6tZNLJzIrtIgxqSl	ZAYN	Still Got Time	70	0.131	0.748	188491	0.627	0.000000	G	0.0852	...	1	0	0	0	0	0

5 rows × 42 columns

In [21]:
```python
#verifying that duplicates have been eliminated
df[df.index =='6iOvnACn4ChlAw4lWUU4dd']
```

Out[21]:

track_id	artist_name	track_name	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	...	Pop	Reggae	Reggaeton	Jazz	Rock	Ska
6iOvnACn4ChlAw4lWUU4dd	Doja Cat	Go To Town	64	0.0716	0.71	217813	0.71	0.000001	C	0.206	...	1	0	0	0	0	0

1 rows × 42 columns

In [22]:
```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 176774 entries, 00021Wy6AyMbLP2tqij86e to 7zzbfi8fvHe6hm342GcNYl
Data columns (total 42 columns):
 #   Column            Non-Null Count   Dtype
---  ------            --------------   -----
 0   artist_name       176774 non-null  object
 1   track_name        176774 non-null  object
 2   popularity        176774 non-null  int64
 3   acousticness      176774 non-null  float64
 4   danceability      176774 non-null  float64
 5   duration_ms       176774 non-null  int64
 6   energy            176774 non-null  float64
 7   instrumentalness  176774 non-null  float64
 8   key               176774 non-null  object
 9   liveness          176774 non-null  float64
 10  loudness          176774 non-null  float64
 11  mode              176774 non-null  object
 12  speechiness       176774 non-null  float64
 13  tempo             176774 non-null  float64
 14  time_signature    176774 non-null  object
 15  valence           176774 non-null  float64
 16  Movie             176774 non-null  int32
 17  R&B               176774 non-null  int32
 18  A Capella         176774 non-null  int32
 19  Alternative       176774 non-null  int32
 20  Country           176774 non-null  int32
 21  Dance             176774 non-null  int32
 22  Electronic        176774 non-null  int32
 23  Anime             176774 non-null  int32
 24  Folk              176774 non-null  int32
 25  Blues             176774 non-null  int32
 26  Opera             176774 non-null  int32
 27  Hip-Hop           176774 non-null  int32
 28  Children's Music  176774 non-null  int32
 29  Rap               176774 non-null  int32
 30  Indie             176774 non-null  int32
 31  Classical         176774 non-null  int32
 32  Pop               176774 non-null  int32
 33  Reggae            176774 non-null  int32
 34  Reggaeton         176774 non-null  int32
 35  Jazz              176774 non-null  int32
 36  Rock              176774 non-null  int32
 37  Ska               176774 non-null  int32
 38  Comedy            176774 non-null  int32
 39  Soul              176774 non-null  int32
 40  Soundtrack        176774 non-null  int32
 41  World             176774 non-null  int32
dtypes: float64(9), int32(26), int64(2), object(5)
memory usage: 40.5+ MB
```

In [23]:

```
#verifying that duplicates have been eliminated
df[df.index =='6iOvnACn4ChlAw4lWUU4dd']
```

Out[23]:

	artist_name	track_name	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	...	Pop	Reggae	Reggaeton	Jazz	Rock	Ska
track_id																	
6iOvnACn4ChlAw4lWUU4dd	Doja Cat	Go To Town	64	0.0716	0.71	217813	0.71	0.000001	C	0.206	...	1	0	0	0	0	0

1 rows × 42 columns

In [24]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 176774 entries, 00021Wy6AyMbLP2tqij86e to 7zzbfi8fvHe6hm342GcNYl
Data columns (total 42 columns):
 #   Column            Non-Null Count   Dtype  
---  ------            --------------   -----  
 0   artist_name       176774 non-null  object 
 1   track_name        176774 non-null  object 
 2   popularity        176774 non-null  int64  
 3   acousticness      176774 non-null  float64
 4   danceability      176774 non-null  float64
 5   duration_ms       176774 non-null  int64  
 6   energy            176774 non-null  float64
 7   instrumentalness  176774 non-null  float64
 8   key               176774 non-null  object 
 9   liveness          176774 non-null  float64
 10  loudness          176774 non-null  float64
 11  mode              176774 non-null  object 
 12  speechiness       176774 non-null  float64
 13  tempo             176774 non-null  float64
 14  time_signature    176774 non-null  object 
 15  valence           176774 non-null  float64
 16  Movie             176774 non-null  int32  
 17  R&B               176774 non-null  int32  
 18  A Capella         176774 non-null  int32  
 19  Alternative       176774 non-null  int32  
 20  Country           176774 non-null  int32  
 21  Dance             176774 non-null  int32  
 22  Electronic        176774 non-null  int32  
 23  Anime             176774 non-null  int32  
 24  Folk              176774 non-null  int32  
 25  Blues             176774 non-null  int32  
 26  Opera             176774 non-null  int32  
 27  Hip-Hop           176774 non-null  int32  
 28  Children's Music  176774 non-null  int32  
 29  Rap               176774 non-null  int32  
 30  Indie             176774 non-null  int32  
 31  Classical         176774 non-null  int32  
 32  Pop               176774 non-null  int32  
 33  Reggae            176774 non-null  int32  
 34  Reggaeton         176774 non-null  int32  
 35  Jazz              176774 non-null  int32  
 36  Rock              176774 non-null  int32  
 37  Ska               176774 non-null  int32  
 38  Comedy            176774 non-null  int32  
 39  Soul              176774 non-null  int32  
 40  Soundtrack        176774 non-null  int32  
 41  World             176774 non-null  int32  
dtypes: float64(9), int32(26), int64(2), object(5)
memory usage: 40.5+ MB
```

We now have 176,774 unique tracks in our dataset (down from 232,725).

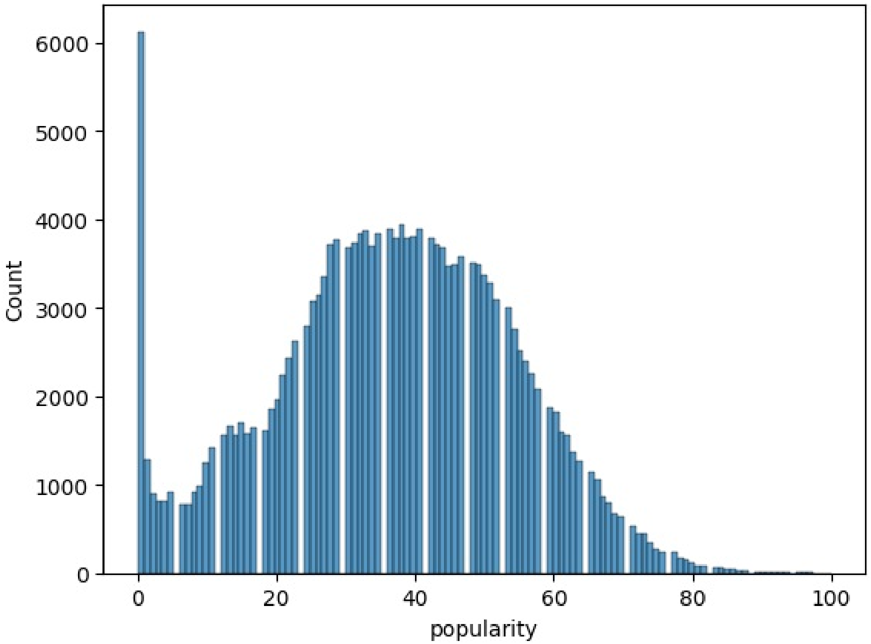
Feature Engineering - is_popular

Since our goal is to be able to identify which tracks will be popular, we need to feature engineer a new column by binarizing the popularity column. To be able to do this, we need to decide on a cut-off point of popularity score which if a song stays above this cut-off point it will be considered "popular" and if it stays below it will be considered "not popular". We can start off by taking a look at the distribution of the popularity score distribution.

In [25]:
```
import matplotlib.pyplot as plt
import seaborn as sns
```

In [26]:
```
#creating a histogram to see distribution of popularity scores in the dataset.
sns.histplot(df['popularity'], bins='auto')
```

Out[26]: <Axes: xlabel='popularity', ylabel='Count'>



From the above histogram we see that we have a bimodal distribution. One of the peaks is at 0, and the other one seems to be around 40. In order to better decide what's popular, we can take a look at the Top 50 songs' popularity scores (this data is also from 2019 similar to our main dataset to keep the analysis consistent.)

Top 50 Songs - 2019

In [27]:
```
df_50 = pd.read_csv("C:\\Users\\tabas\\top50.csv",encoding='latin1')
```

In [28]:
```
df_50.head()
```

Out[28]:

	Unnamed: 0	Track.Name	Artist.Name	Genre	Beats.Per.Minute	Energy	Danceability	Loudness..dB..	Liveness	Valence.	Length.	Acousticness..	Speechiness.	Popularity
0	1	Señorita	Shawn Mendes	canadian pop	117	55	76	-6	8	75	191	4	3	79
1	2	China	Anuel AA	reggaeton flow	105	81	79	-4	8	61	302	8	9	92
2	3	boyfriend (with Social House)	Ariana Grande	dance pop	190	80	40	-4	16	70	186	12	46	85
3	4	Beautiful People (feat. Khalid)	Ed Sheeran	pop	93	65	64	-8	8	55	198	12	19	86
4	5	Goodbyes (Feat. Young Thug)	Post Malone	dfw rap	150	65	58	-4	11	18	175	45	7	94

In [29]:

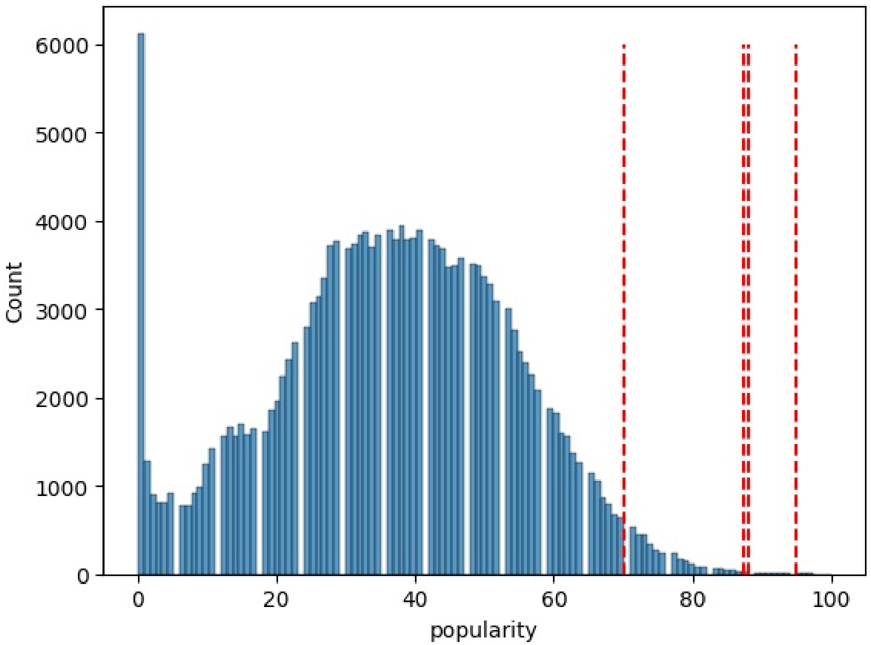
```
#displaying stats information of Top 50 songs
df_50['Popularity'].describe()
```

Out[29]:

```
count    50.000000
mean     87.500000
std       4.491489
min      70.000000
25%      86.000000
50%      88.000000
75%      90.750000
max      95.000000
Name: Popularity, dtype: float64
```

In [30]:

```
fig, ax = plt.subplots()
sns.histplot(df['popularity'], bins='auto', ax=ax)
stats=['mean', '50%', 'min', 'max']
for stat in stats:
    ax.vlines(x=df_50['Popularity'].describe()[stat], ymin=0, ymax=6000, linestyles='dashed', colors='red', label=stat)
```



We can see that there was a range of popularity scores in the Top 50 songs between 70 and 95. Which means that any song that is above a 70 theoretically could be a popular song. It doesn't make sense to use median or mean scores for our cutoff point in this case since then we would be disregarding all the songs that had lower values than 87.5 or 88 as unpopular which is untrue. In a previous iteration of this project, we proceeded modelling with the popularity score of 70 being the cutoff point and our models did not perform well since the cutoff point was based off of only 50 data points. Therefore we proceeded to look at a larger dataset to get a better sample size of popular songs.

Top 100 Songs - 2019

In [31]:
```
df_100 = pd.read_csv("C:\\Users\\tabas\\spotify_top_100_2019.csv",encoding='latin1')
```

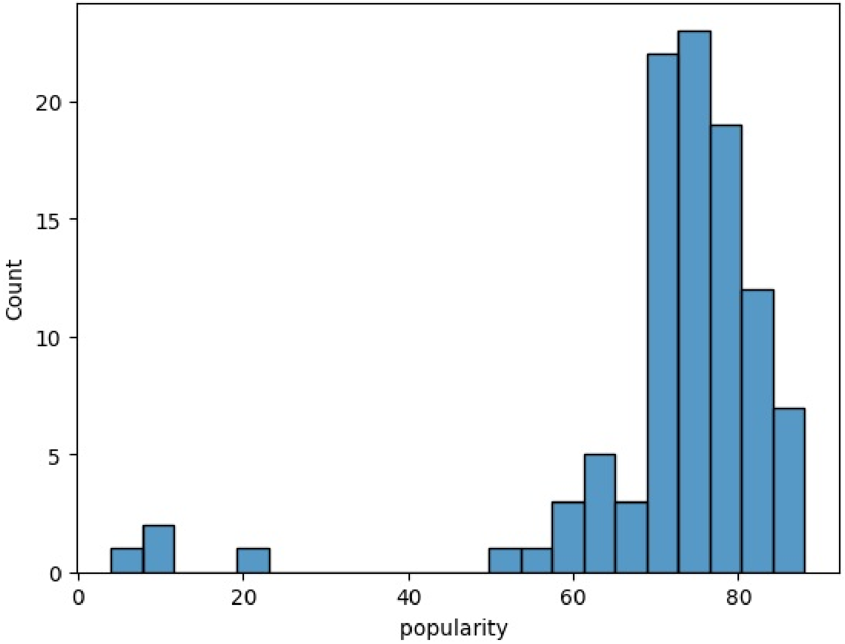
In [32]:
```
df_100['popularity '].describe()
```

Out[32]:
```
count    100.000000
mean      72.020000
std       14.088451
min        4.000000
25%       70.000000
50%       74.500000
75%       79.000000
max       88.000000
Name: popularity , dtype: float64
```

The minimum value of 4 for the popularity score on the Top 100 Songs chart seems like an outlier. Next, we'll visualize the spread of this column to confirm.

In [33]:
```
fig, ax = plt.subplots()
sns.histplot(df_100['popularity '], bins='auto', ax=ax)
```

Out[33]:
```
<Axes: xlabel='popularity ', ylabel='Count'>
```



In [34]:

```
#Outlier Removal with the IQR method

def find_outliers_IQR(data, return_limits = False):
    """Use Tukey's Method of outlier removal AKA InterQuartile-Range Rule
    and return boolean series where True indicates it is an outlier.
    - Calculates the range between the 75% and 25% quartiles
    - Outliers fall outside upper and lower limits, using a treshold of  1.5*IQR the 75% and 25% quartiles.

    IQR Range Calculation:
        res = df.describe()
        IQR = res['75%'] -  res['25%']
        lower_limit = res['25%'] - 1.5*IQR
        upper_limit = res['75%'] + 1.5*IQR

    Args:
        data (Series,or ndarray): data to test for outliers.

    Returns:
        [boolean Series]: A True/False for each row use to slice outliers.

    Adapted from Flatiron School Phase #2 Py Files.
    URL = https://github.com/flatiron-school/Online-DS-FT-022221-Cohort-Notes/blob/master/py_files/functions_SG.py

    """
    df_b=data
    res= df_b.describe()

    IQR = res['75%'] -  res['25%']
    lower_limit = res['25%'] - 1.5*IQR
    upper_limit = res['75%'] + 1.5*IQR
```

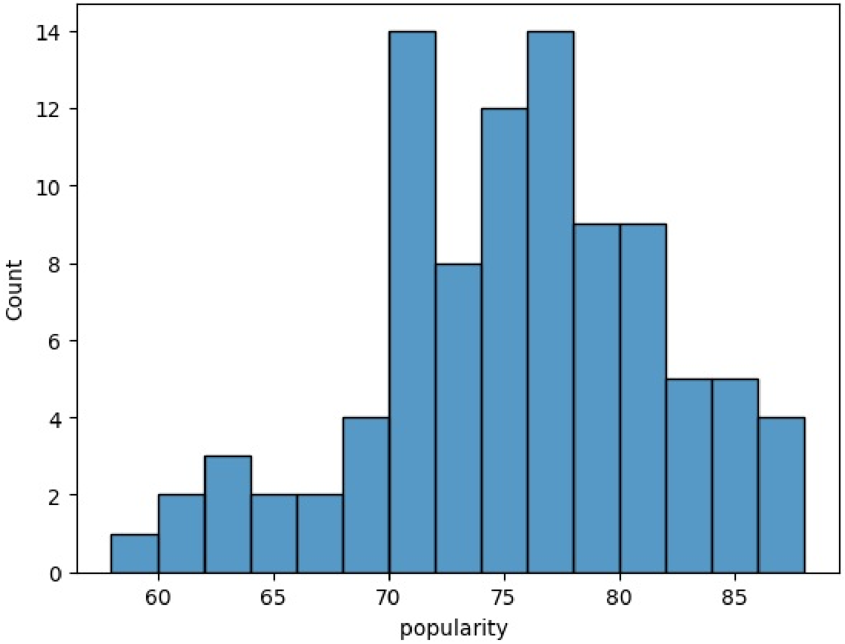
```
    if return_limits:
        return lower_limit, upper_limit

    else:
        idx_outs = (df_b>upper_limit) | (df_b<lower_limit)
        return idx_outs
```

In [35]:
```
#removing outliers from the popularity column
df_100 = df_100[find_outliers_IQR(df_100['popularity '])==False]
#displaying minimum & maxium values in popularity column
print("Minimum:", df_100['popularity '].min())
print("Maximum:", df_100['popularity '].max())
```

```
Minimum: 58
Maximum: 88
```

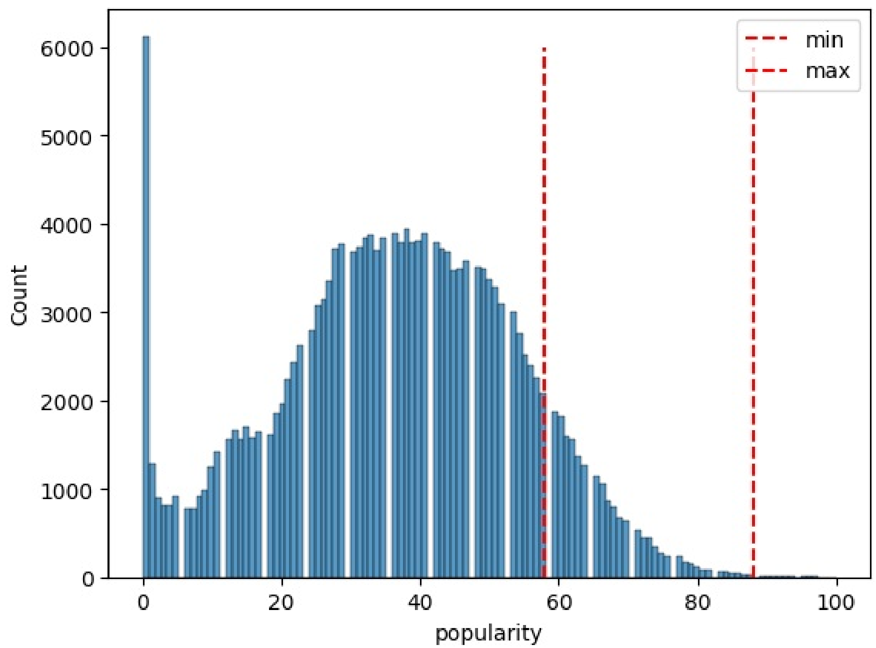
In [36]:
```
fig, ax = plt.subplots()
sns.histplot(df_100['popularity '], bins=15, ax=ax)
```

Out[36]: <Axes: xlabel='popularity ', ylabel='Count'>



In [37]:
```
#visualizing the min and max popularity scores on the overall dataset histogram
fig, ax = plt.subplots()
sns.histplot(df['popularity'], bins='auto', ax=ax)
ax.vlines(x=df_100['popularity '].min(), ymin=0, ymax=6000, linestyles='dashed', colors='red', label='min')
ax.vlines(x=df_100['popularity '].max(), ymin=0, ymax=6000, linestyles='dashed', colors='red', label='max')
plt.legend()
```

Out[37]: <matplotlib.legend.Legend at 0x159a5cd2910>



As we can expect to see, the top 100 songs have a wider range and therefore a lower popularity score threshold compared to the top 50 songs. We will be defining a song being popular as being Top 100 worthy and therefore will establish our cutoff point at 58.

In [38]:
```
#creating is_popular column with our cutoff point
df['is_popular']=(df['popularity']>=58).astype('int')
df.head()
```

Out[38]:

track_id	artist_name	track_name	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	...	Reggae	Reggaeton	Jazz	Rock	Ska	Com
00021Wy6AyMbLP2tqij86e	Capcom Sound Team	Zangief's Theme	13	0.234	0.617	169173	0.862	0.976000	G	0.1410	...	0	0	0	0	0	
000CzNKC8PEt1yC3L8dqwV	Henri Salvador	Coeur Brisé à Prendre - Remastered	5	0.249	0.518	130653	0.805	0.000000	F	0.3330	...	0	0	0	0	0	
000DfZJww8KiixTKuk9usJ	Mike Love	Earthlings	30	0.366	0.631	357573	0.513	0.000004	D	0.1090	...	1	0	0	0	0	
000EWWBkYaREzsBplYjUag	Don Philippe	Fewerdolr	39	0.815	0.768	104924	0.137	0.922000	C#	0.1130	...	0	0	1	0	0	
000xQL6tZNLJzIrtIgxqSI	ZAYN	Still Got Time	70	0.131	0.748	188491	0.627	0.000000	G	0.0852	...	0	0	0	0	0	

5 rows × 43 columns

In [39]:
```
#dropping popularity score column since we will not be using it
df.drop(['popularity', 'artist_name', 'track_name'], axis=1, inplace=True)
df.head()
```

Out[39]:

	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	...	Reggae	Reggaeton	Jazz	Rock	Ska	Comedy
track_id																	
00021Wy6AyMbLP2tqij86e	0.234	0.617	169173	0.862	0.976000	G	0.1410	-12.855	Major	0.0514	...	0	0	0	0	0	0
000CzNKC8PEt1yC3L8dqwV	0.249	0.518	130653	0.805	0.000000	F	0.3330	-6.248	Major	0.0407	...	0	0	0	0	0	0
000DfZJww8KiixTKuk9usJ	0.366	0.631	357573	0.513	0.000004	D	0.1090	-6.376	Major	0.0293	...	1	0	0	0	0	0
000EWWBkYaREzsBplYjUag	0.815	0.768	104924	0.137	0.922000	C#	0.1130	-13.284	Minor	0.0747	...	0	0	1	0	0	0
000xQL6tZNLJzIrtIgxqSI	0.131	0.748	188491	0.627	0.000000	G	0.0852	-6.029	Major	0.0644	...	0	0	0	0	0	0

5 rows × 40 columns

We dropped popularity scores since we already binarized that column, but additionally we are dropping 'artist_name' and 'track_name' since we are looking at the anatomy of a song and not who sings it or what it's called. The goal is to identify songs that will become popular without being affected by the artist's name since we would also like to find songs from up-and-coming artists.

train_test_split

In [40]:
```
#splitting the data to training and test sets in order to be able to measure performance
from sklearn.model_selection import train_test_split
y=df['is_popular']
X=df.drop('is_popular',axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=42)
```

One Hot Encoding the Categorical Columns

We still have categorical columns that need one hot encoding. Namely, these columns are 'key', 'mode' and 'time_signature'.

In [41]:
```
#Check to see how many more columns we will be creating by OHE the cat_cols.
df.nunique()
```

Out[41]:
```
acousticness         4734
danceability         1295
duration_ms         70749
energy               2517
instrumentalness     5400
key                    12
liveness             1732
loudness            27923
mode                    2
speechiness          1641
tempo               78509
time_signature          5
valence              1692
Movie                   2
R&B                     2
A Capella               2
Alternative             2
Country                 2
Dance                   2
Electronic              2
Anime                   2
Folk                    2
Blues                   2
Opera                   2
Hip-Hop                 2
Children's Music        2
Rap                     2
Indie                   2
Classical               2
Pop                     2
Reggae                  2
Reggaeton               2
Jazz                    2
Rock                    2
Ska                     2
Comedy                  2
Soul                    2
Soundtrack              2
World                   2
is_popular              2
dtype: int64
```

In [42]:
```
#define categorical columns
cat_cols = ['key', 'mode', 'time_signature']
```

In [43]:
```
from sklearn.preprocessing import OneHotEncoder
import pandas as pd

# Define the OneHotEncoder with desired parameters
encoder = OneHotEncoder(sparse_output=False, drop='first')

# Training set
data_ohe_train = encoder.fit_transform(X_train[cat_cols])
df_ohe_train = pd.DataFrame(data_ohe_train, columns=encoder.get_feature_names_out(cat_cols), index=X_train.index)

# Testing set
data_ohe_test = encoder.transform(X_test[cat_cols])
df_ohe_test = pd.DataFrame(data_ohe_test, columns=encoder.get_feature_names_out(cat_cols), index=X_test.index)
```

In [44]:
```python
pd.set_option("display.max_columns", None)
df_ohe_train
```

Out[44]:

track_id	key_A#	key_B	key_C	key_C#	key_D	key_D#	key_E	key_F	key_F#	key_G	key_G#	mode_Minor	time_signature_1/4	time_signature_3/4	time_signature_4/4	time
5SbN8IXhPno4BRrFb9yqkF	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	
34n3eoeqVaXAgtMqy8Ncyz	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	
3QbxHo2OTwBVDZbaJaMniP	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	
6Y8aA0SWBMB5XTZIXIDpYv	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	
16h3GCdEJ9lgiOyox4LJQA	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
5H23I3K3TUXQMsLg2FzCiY	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	
4YnYtYWBmDM8YjfMMK0cqs	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	
5o5xTG5Mh3JAm2BZv4nOI7	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	
6T1oL7ed1wUEqICR1iCpIR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	
5MnEYPkZ5HC7BQ988kBKqp	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	

123741 rows × 16 columns

In [45]:
```python
#merging OHE columns with numerical columns
X_train = pd.concat([X_train.drop(cat_cols, axis=1), df_ohe_train], axis=1)
X_test = pd.concat([X_test.drop(cat_cols, axis=1), df_ohe_test], axis=1)
X_train.tail()
```

Out[45]:

track_id	acousticness	danceability	duration_ms	energy	instrumentalness	liveness	loudness	speechiness	tempo	valence	Movie	R&B	A Capella	Alternative	Country	Da
5H23I3K3TUXQMsLg2FzCiY	0.23100	0.461	326680	0.252	0.000084	0.106	-17.082	0.2740	74.768	0.481	0	0	0	0	0	
4YnYtYWBmDM8YjfMMK0cqs	0.00571	0.309	201947	0.820	0.000368	0.159	-4.844	0.0636	174.762	0.399	0	0	0	0	0	
5o5xTG5Mh3JAm2BZv4nOI7	0.01210	0.691	265587	0.834	0.001480	0.115	-4.378	0.0432	129.982	0.330	0	0	0	0	0	
6T1oL7ed1wUEqICR1iCpIR	0.35500	0.364	224387	0.733	0.000007	0.981	-5.071	0.0299	145.394	0.244	0	0	0	0	1	
5MnEYPkZ5HC7BQ988kBKqp	0.01190	0.478	211800	0.695	0.000000	0.119	-5.923	0.0529	89.801	0.297	0	1	0	0	0	

In [46]:
```python
#concatenating all parts of our data for future reference (see Data Visualizations section)
df_ohe_x = pd.concat([X_train, X_test])
df_ohe_y = pd.concat([y_train, y_test])
df_ohe = pd.concat([df_ohe_x, df_ohe_y], axis=1)
```

MODELS

We need a function that will show us the classification report, the confusion matrix as well as the ROC curve to be able to evaluate our models.

In [47]:
```
from sklearn.metrics import classification_report
from sklearn.metrics import ConfusionMatrixDisplay, RocCurveDisplay
```

In [48]:
```
from sklearn.metrics import classification_report
from sklearn.metrics import ConfusionMatrixDisplay, RocCurveDisplay
import matplotlib.pyplot as plt

def classification(y_true, y_pred, X, clf):
    """This function shows the classification report,
    the confusion matrix as well as the ROC curve for evaluation of model quality.

    y_true: Correct y values, typically y_test that comes from the train_test_split performed at the beginning of model development.
    y_pred: Predicted y values by the model.
    clf: classifier model that was fit to training data.
    X: X_test values"""

    # Classification report
    print("CLASSIFICATION REPORT")
    print("------------------------------------------")
    print(classification_report(y_true=y_true, y_pred=y_pred))

    # Creating a figure/axes for confusion matrix and ROC curve
    fig, ax = plt.subplots(ncols=2, figsize=(12, 5))

    # Plotting the normalized confusion matrix
    ConfusionMatrixDisplay.from_estimator(estimator=clf, X=X, y=y_true, cmap='Blues', normalize='true', ax=ax[0])
    ax[0].set_title("Normalized Confusion Matrix")

    # Plotting the ROC curve
    RocCurveDisplay.from_estimator(estimator=clf, X=X, y=y_true, ax=ax[1])
    ax[1].plot([0,1], [0,1], ls='--', color='orange', label='Random Chance')
    ax[1].set_title("ROC Curve")
    ax[1].legend(loc="lower right")

    plt.tight_layout()
    plt.show()
```

In [49]:
```
#class imbalance percentages
y_train.value_counts(normalize=True)
```

Out[49]:
```
is_popular
0    0.885503
1    0.114497
Name: proportion, dtype: float64
```

Addressing Class Imbalance with SMOTENC

In [50]:
```
#looking at column names to extract categorical column indices for SMOTENC
X_train.columns
```

Out[50]:
```
Index(['acousticness', 'danceability', 'duration_ms', 'energy',
       'instrumentalness', 'liveness', 'loudness', 'speechiness', 'tempo',
       'valence', 'Movie', 'R&B', 'A Capella', 'Alternative', 'Country',
       'Dance', 'Electronic', 'Anime', 'Folk', 'Blues', 'Opera', 'Hip-Hop',
       'Children's Music', 'Rap', 'Indie', 'Classical', 'Pop', 'Reggae',
       'Reggaeton', 'Jazz', 'Rock', 'Ska', 'Comedy', 'Soul', 'Soundtrack',
       'World', 'key_A#', 'key_B', 'key_C', 'key_C#', 'key_D', 'key_D#',
       'key_E', 'key_F', 'key_F#', 'key_G', 'key_G#', 'mode_Minor',
       'time_signature_1/4', 'time_signature_3/4', 'time_signature_4/4',
       'time_signature_5/4'],
      dtype='object')
```

In [51]:
```
#creating a list of categorical column indices
cat_cols = list(range(10, len(X_train.columns)))
X_train.columns[cat_cols]
```

Out[51]:
```
Index(['Movie', 'R&B', 'A Capella', 'Alternative', 'Country', 'Dance',
       'Electronic', 'Anime', 'Folk', 'Blues', 'Opera', 'Hip-Hop',
       'Children's Music', 'Rap', 'Indie', 'Classical', 'Pop', 'Reggae',
       'Reggaeton', 'Jazz', 'Rock', 'Ska', 'Comedy', 'Soul', 'Soundtrack',
       'World', 'key_A#', 'key_B', 'key_C', 'key_C#', 'key_D', 'key_D#',
       'key_E', 'key_F', 'key_F#', 'key_G', 'key_G#', 'mode_Minor',
       'time_signature_1/4', 'time_signature_3/4', 'time_signature_4/4',
       'time_signature_5/4'],
      dtype='object')
```

In [52]:
```
#Using SMOTENC to address class imbalance. We are not using SMOTE since we have categorical columns.
from imblearn.over_sampling import SMOTE, SMOTENC

sm = SMOTENC(categorical_features=cat_cols, random_state=42)

X_train_sm, y_train_sm = sm.fit_resample(X_train, y_train)
y_train_sm.value_counts(normalize=True)
```

Out[52]:
```
is_popular
0    0.5
1    0.5
Name: proportion, dtype: float64
```

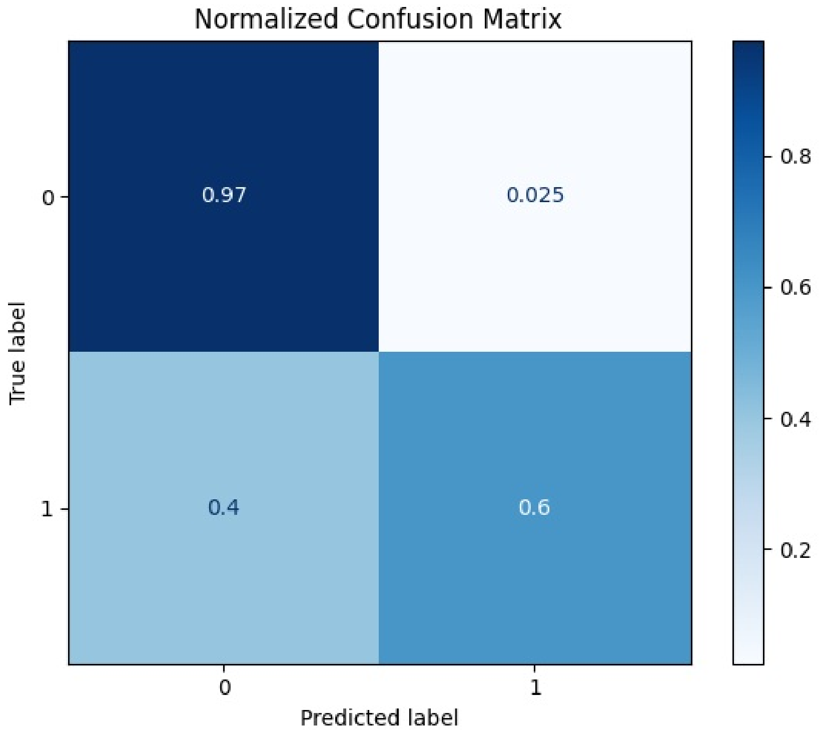
In [53]:
```
import pandas as pd
from sklearn.metrics import recall_score

def add_results(model_name, df, y_test, y_pred):
    # Create a new DataFrame with the results
    new_row = pd.DataFrame({
        'Model Name': [model_name],
        'Recall Score': [round(recall_score(y_test, y_pred), 2)]
    })

    # Append the new row to the existing results DataFrame using concat
    df = pd.concat([df, new_row], ignore_index=True)

    return df
```

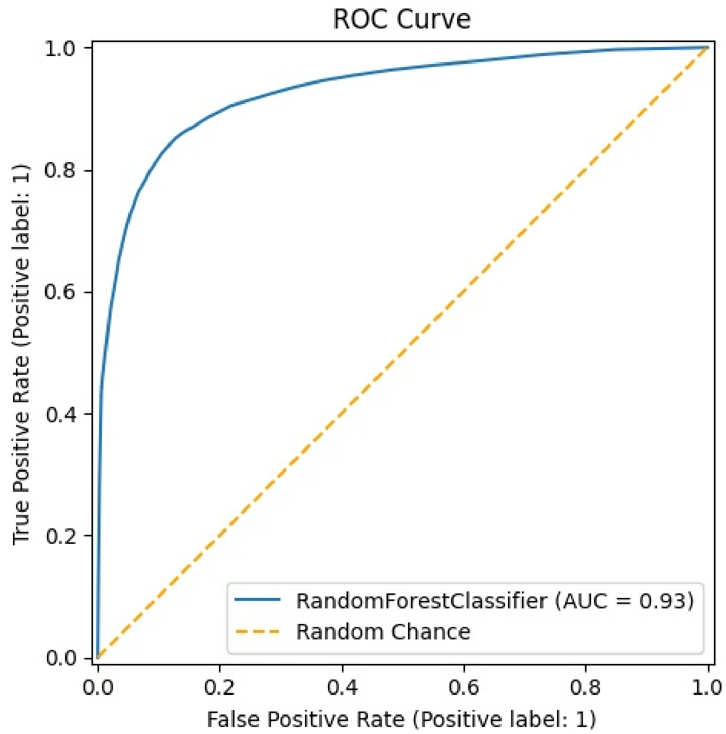
Model #1 - Random Forest Classifier

Initial Model

In [54]:
```
#Fitting RF Classifier to SMOTE'd data
from sklearn.ensemble import RandomForestClassifier

clf_rf = RandomForestClassifier(random_state=42)
clf_rf.fit(X_train_sm, y_train_sm)

#Making predictions and evaluation.
y_pred = clf_rf.predict(X_test)
classification(y_test, y_pred, X_test, clf_rf)
```

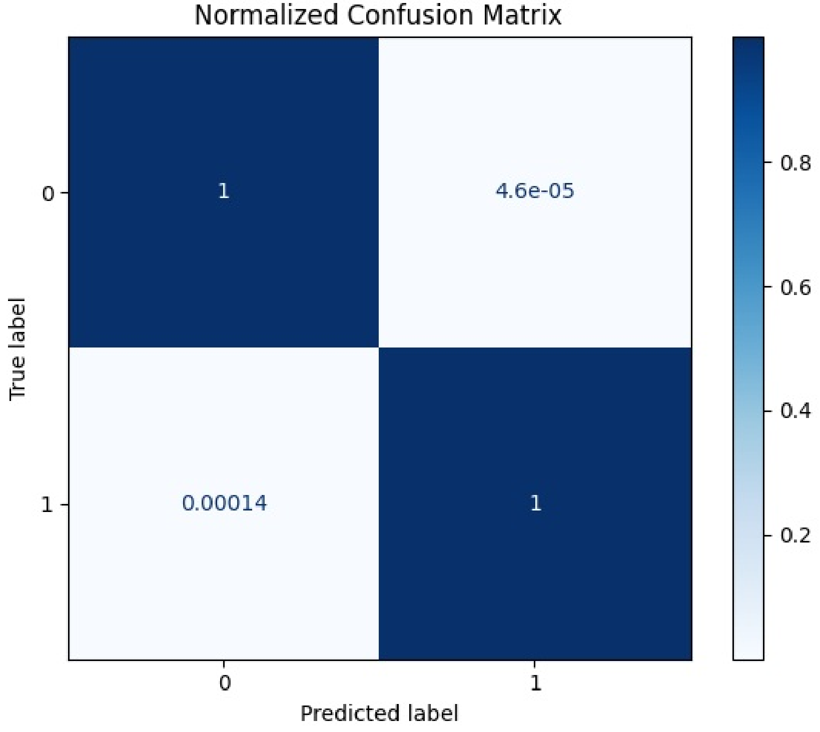
```
CLASSIFICATION REPORT
------------------------------------------
              precision    recall  f1-score   support

           0       0.95      0.97      0.96     47002
           1       0.75      0.60      0.67      6031

    accuracy                           0.93     53033
   macro avg       0.85      0.79      0.81     53033
weighted avg       0.93      0.93      0.93     53033
```

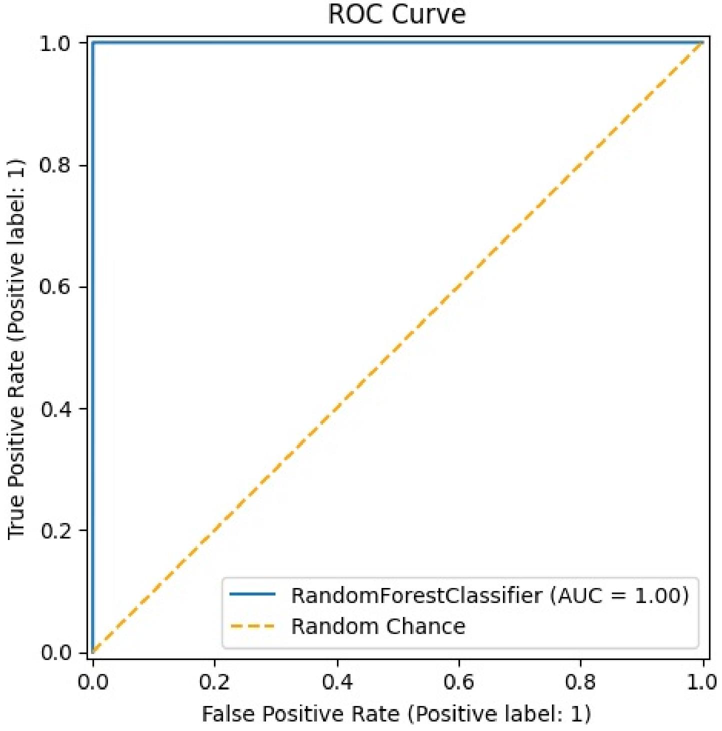




In [55]:
```
#Evaluating the model performance for the training data
y_pred = clf_rf.predict(X_train_sm)
classification(y_train_sm, y_pred, X_train_sm, clf_rf)
```

```
CLASSIFICATION REPORT
------------------------------------------
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    109573
           1       1.00      1.00      1.00    109573

    accuracy                           1.00    219146
   macro avg       1.00      1.00      1.00    219146
weighted avg       1.00      1.00      1.00    219146
```





Our model is performing perfectly on the training data but not so much on the test data since it is overfitting to the training set. We need to tune our model to get more accurate results on unseen data. We will be using a grid search to optimize for the recall score. We are optimizing recall instead of other scores since we primarily care about correctly identifying a song that will be popular and we don't mind it if we pick a few songs that don't end up becoming popular

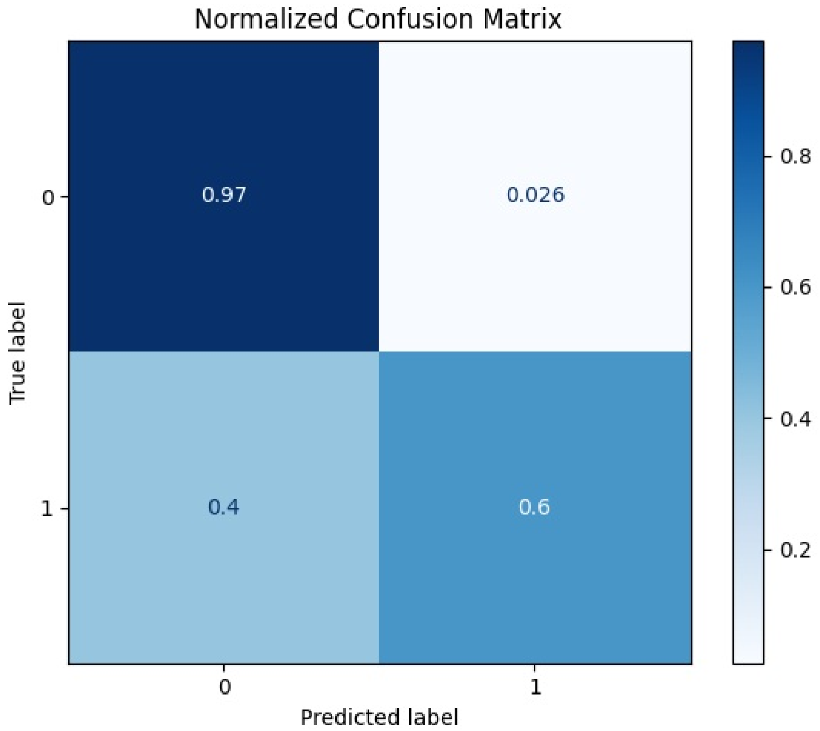
Hyperparameter Tuning

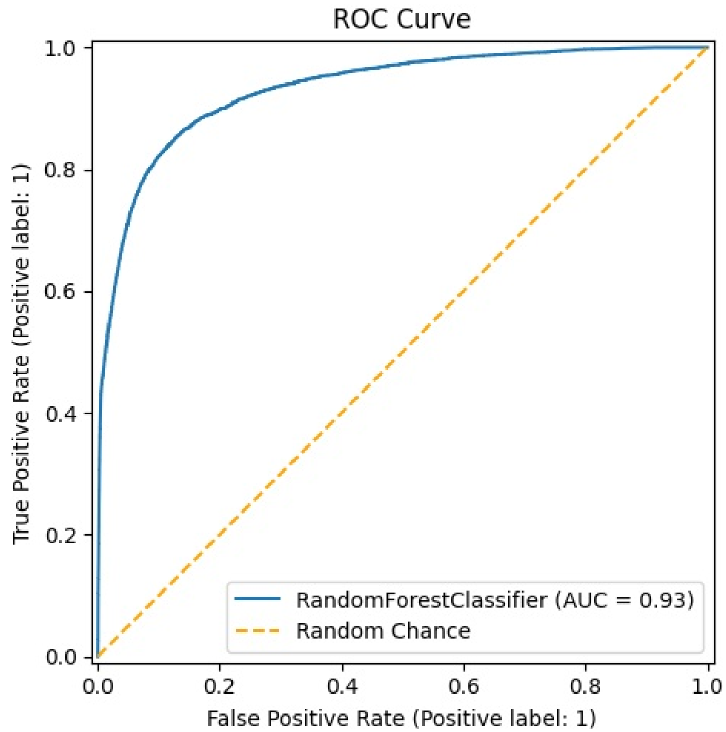
In [56]:
```
# from sklearn.model_selection import GridSearchCV

# clf = RandomForestClassifier()
# grid = {'criterion': ['gini', 'entropy'],
#         'max_depth': [10, 20, None],
#         'min_samples_leaf': [1, 2, 3]
#        }

# gridsearch = GridSearchCV(estimator=clf, param_grid = grid, scoring='recall')
```

```
# gridsearch.fit(X_train_sm,  y_train_sm)
# gridsearch.best_params_
# #Results: {'criterion': 'entropy', 'max_depth': None, 'min_samples_leaf': 2}
```

In [57]:
```
clf_rf_tuned = RandomForestClassifier(criterion='entropy', max_depth=None,
                                      min_samples_leaf=2, class_weight='balanced',
                                      random_state=42)
clf_rf_tuned.fit(X_train_sm, y_train_sm)

y_pred = clf_rf_tuned.predict(X_test)
classification(y_test, y_pred, X_test, clf_rf_tuned)
```

```
CLASSIFICATION REPORT
------------------------------------------
              precision    recall  f1-score   support

           0       0.95      0.97      0.96     47002
           1       0.75      0.60      0.66      6031

    accuracy                           0.93     53033
   macro avg       0.85      0.79      0.81     53033
weighted avg       0.93      0.93      0.93     53033
```





Tuning the hyperparameters of our model improved the recall score for predicting popular songs by 1%. We can proceed with trying additional types of models to see if the recall score improves.

Model #2 - LogisticRegressionCV

Removing Outliers since LR model is sensitive to outliers and we need scaled data so we'll process our data once more and scale it

In [58]:
```
#separating out the numerical columns for outlier removal
num_cols = ['acousticness', 'danceability', 'duration_ms', 'energy', 'instrumentalness',
            'liveness', 'loudness', 'speechiness', 'tempo', 'valence']
num_cols
```

Out[58]:
```
['acousticness',
 'danceability',
 'duration_ms',
 'energy',
 'instrumentalness',
 'liveness',
 'loudness',
 'speechiness',
 'tempo',
 'valence']
```

In [59]:
```
#Concatenating the training and testing sets together for outlier removal
df_train = pd.concat([X_train, y_train], axis=1)
df_test = pd.concat([X_test, y_test], axis=1)
```

In [60]:
```
#finding and removing outliers based on X_train (df_train) to avoid data leakage

original_length_train = len(df_train)
original_length_test = len(df_test)

for col in num_cols:

    lower_limit, upper_limit = find_outliers_IQR(df_train[col], return_limits=True)

    df_train = df_train[(df_train[col]>lower_limit) & (df_train[col]<upper_limit)]
    df_test = df_test[(df_test[col]>lower_limit) & (df_test[col]<upper_limit)]

print(f'{original_length_train - len(df_train)} outliers removed from training set')
print(f'{original_length_test - len(df_test)} outliers removed from test set')
```

```
55567 outliers removed from training set
23796 outliers removed from test set
```

In [61]:
```
#Separating out the X and y values for training and test sets

y_train = df_train['is_popular']
X_train = df_train.drop('is_popular', axis=1)

y_test = df_test['is_popular']
X_test = df_test.drop('is_popular', axis=1)
```

In [62]:
```
y_train.value_counts(normalize=True)
```

Out[62]:
```
is_popular
0    0.842345
1    0.157655
Name: proportion, dtype: float64
```

Addressing Class Imbalance with SMOTENC

In [63]:
```
X_train.columns
```

Out[63]:
```
Index(['acousticness', 'danceability', 'duration_ms', 'energy',
       'instrumentalness', 'liveness', 'loudness', 'speechiness', 'tempo',
       'valence', 'Movie', 'R&B', 'A Capella', 'Alternative', 'Country',
       'Dance', 'Electronic', 'Anime', 'Folk', 'Blues', 'Opera', 'Hip-Hop',
       'Children's Music', 'Rap', 'Indie', 'Classical', 'Pop', 'Reggae',
       'Reggaeton', 'Jazz', 'Rock', 'Ska', 'Comedy', 'Soul', 'Soundtrack',
       'World', 'key_A#', 'key_B', 'key_C', 'key_C#', 'key_D', 'key_D#',
       'key_E', 'key_F', 'key_F#', 'key_G', 'key_G#', 'mode_Minor',
       'time_signature_1/4', 'time_signature_3/4', 'time_signature_4/4',
       'time_signature_5/4'],
      dtype='object')
```

In [64]:
```
cat_cols = list(range(10,len(X_train.columns)))
cat_cols
```

Out[64]:
```
[10,
 11,
 12,
 13,
 14,
 15,
 16,
 17,
 18,
 19,
 20,
 21,
 22,
 23,
 24,
 25,
 26,
 27,
 28,
 29,
 30,
 31,
 32,
 33,
 34,
 35,
 36,
 37,
 38,
 39,
 40,
 41,
 42,
 43,
 44,
 45,
 46,
 47,
 48,
 49,
 50,
 51]
```

In [65]:
```
sm = SMOTENC(categorical_features=cat_cols, random_state=42)

X_train_sm, y_train_sm = sm.fit_resample(X_train, y_train)
y_train_sm.value_counts(normalize=True)
```

Out[65]:
```
is_popular
0    0.5
1    0.5
Name: proportion, dtype: float64
```

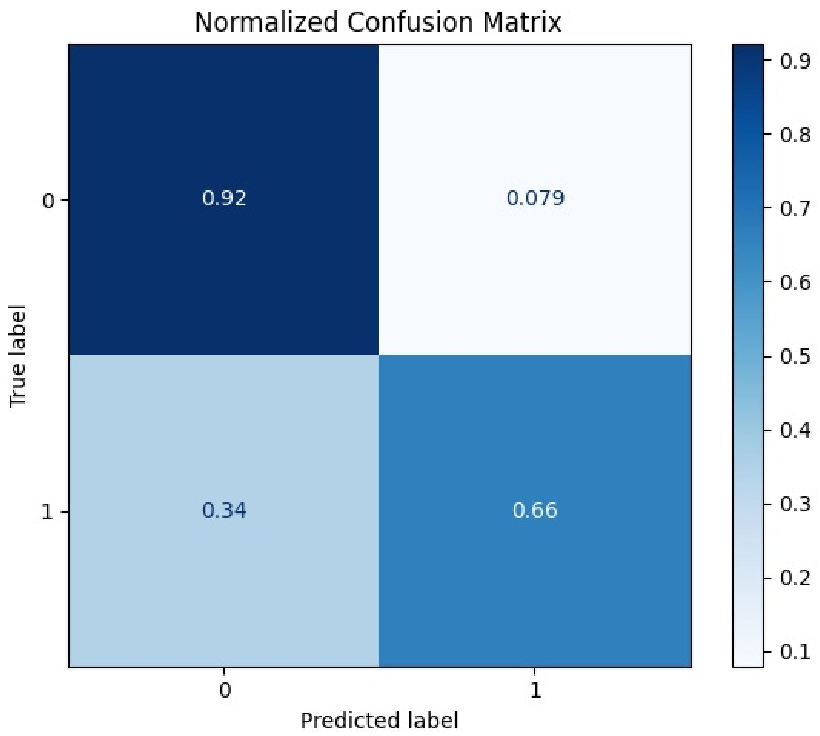
Scaling the data

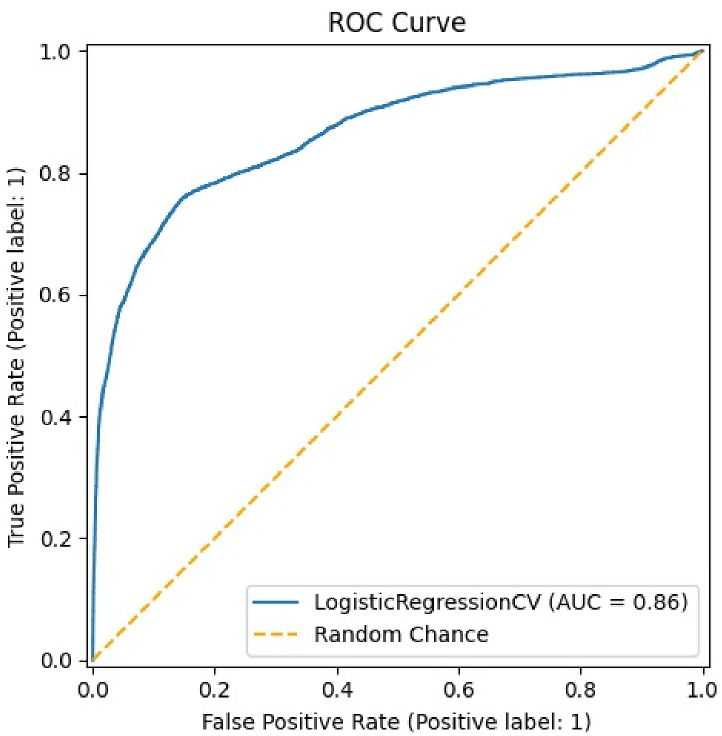
In [66]:
```
#Using Standard Scaler to scale the smote'd data
from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler()

X_train_sm_sc = scaler.fit_transform(X_train_sm)
X_test_sc = scaler.transform(X_test)
```

In [67]:
```
from sklearn.linear_model import LogisticRegressionCV
clf_logregcv = LogisticRegressionCV(cv=5, random_state=42)
clf_logregcv.fit(X_train_sm_sc,  y_train_sm)
y_pred = clf_logregcv.predict(X_test_sc)
classification(y_test, y_pred, X_test_sc, clf_logregcv)
```

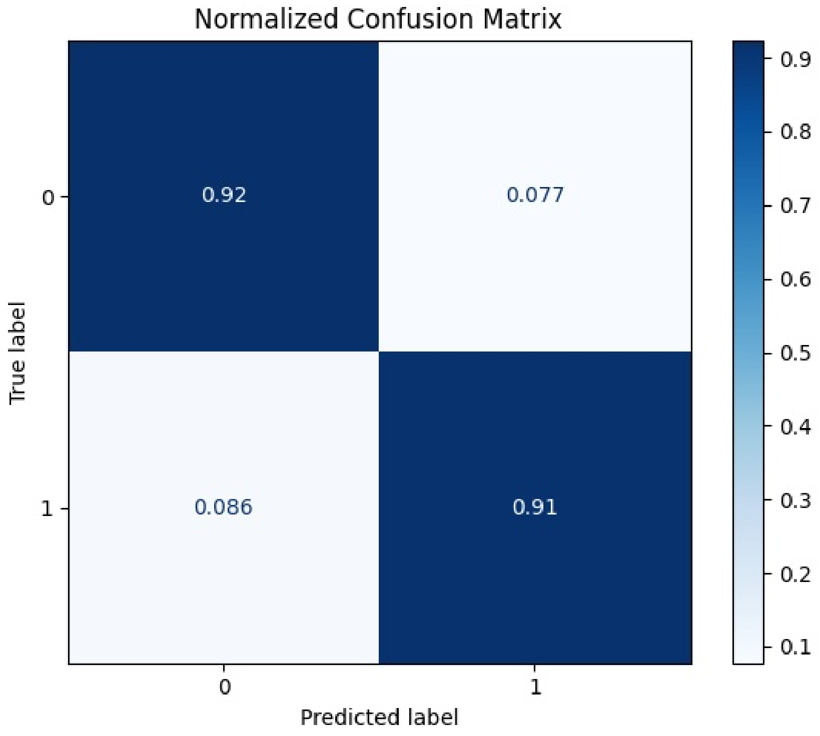
```
CLASSIFICATION REPORT
------------------------------------------
              precision    recall  f1-score   support

           0       0.93      0.92      0.93     24588
           1       0.61      0.66      0.63      4649

    accuracy                           0.88     29237
   macro avg       0.77      0.79      0.78     29237
weighted avg       0.88      0.88      0.88     29237
```

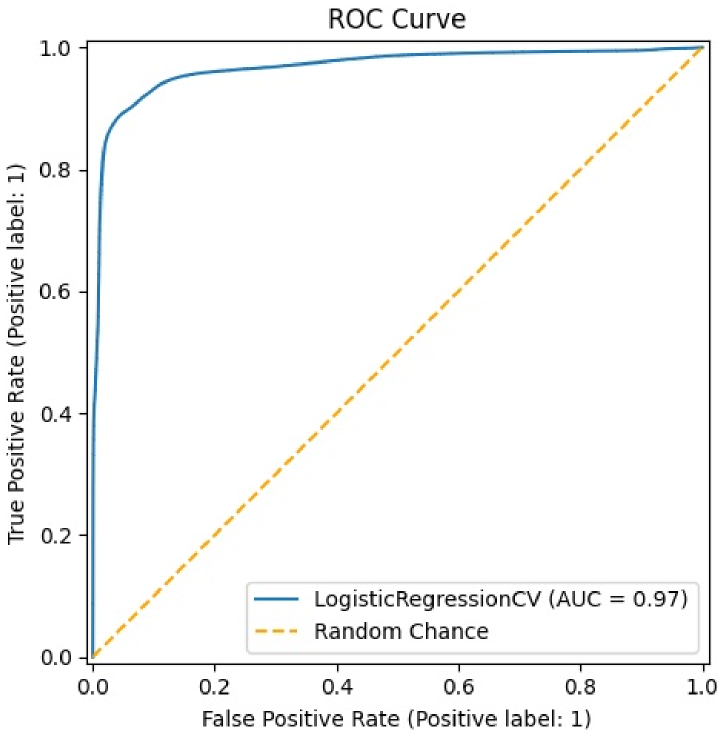




In [68]:
```
#Evaluating the model performance for the training data
y_pred = clf_logregcv.predict(X_train_sm_sc)
classification(y_train_sm, y_pred, X_train_sm_sc, clf_logregcv)
```

```
CLASSIFICATION REPORT
------------------------------------------
              precision    recall  f1-score   support

           0       0.91      0.92      0.92     57426
           1       0.92      0.91      0.92     57426

    accuracy                           0.92    114852
   macro avg       0.92      0.92      0.92    114852
weighted avg       0.92      0.92      0.92    114852
```





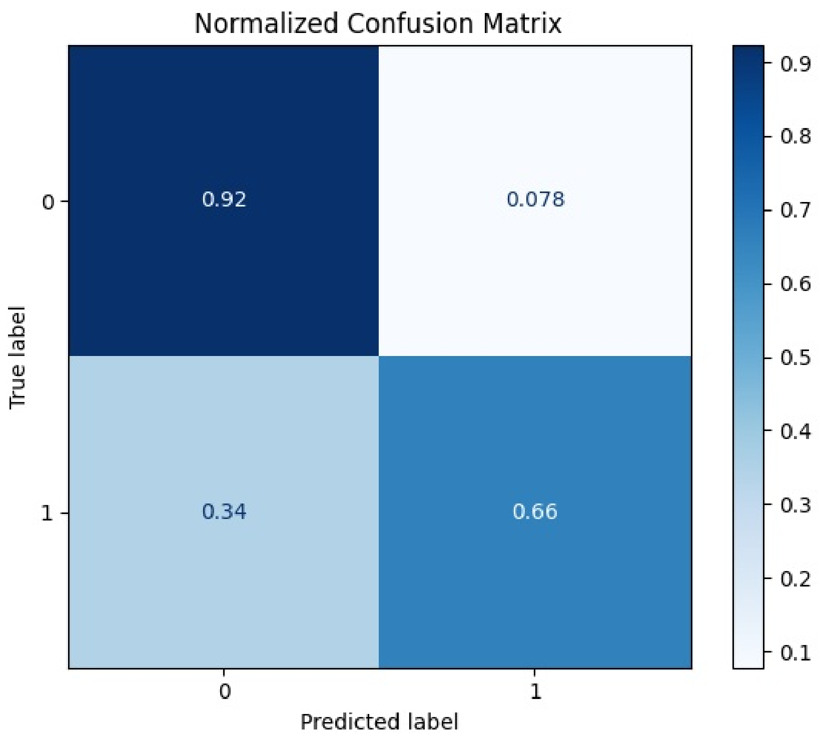
Hyperparameter Tuning

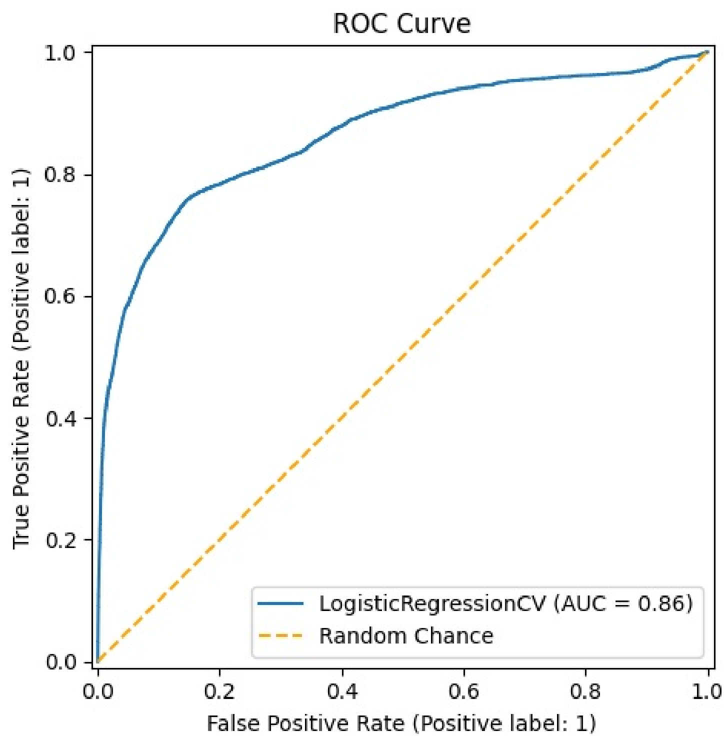
In [69]:
```
# clf = LogisticRegressionCV(cv=5)
# grid = {'penalty': ['l1','l2'],
#         'solver': ['liblinear', 'lbfgs', 'sag', 'saga'],
#         'class_weight': ['balanced', None],
#         'Cs': [1e12, 10, 1, 0.1]
#    }

# gridsearch = GridSearchCV(estimator=clf, param_grid = grid, scoring='recall', n_jobs=-1)

# gridsearch.fit(X_train_sm_sc,  y_train_sm)
# gridsearch.best_params_
# # {'Cs': 1, 'class_weight': 'balanced', 'penalty': 'l2', 'solver': 'liblinear'}
```

In [70]:
```
clf_logregcv_tuned = LogisticRegressionCV(cv=5, class_weight='balanced', Cs=1,
                                          penalty='l2', solver='liblinear',
```

```
                                                   random_state=42)
clf_logregcv_tuned.fit(X_train_sm_sc,  y_train_sm)
y_pred = clf_logregcv_tuned.predict(X_test_sc)
classification(y_test, y_pred, X_test_sc, clf_logregcv_tuned)
```

```
CLASSIFICATION REPORT
------------------------------------------
              precision    recall  f1-score   support

           0       0.93      0.92      0.93     24588
           1       0.61      0.66      0.63      4649

    accuracy                           0.88     29237
   macro avg       0.77      0.79      0.78     29237
weighted avg       0.88      0.88      0.88     29237
```





Unfortunately, the parameters returned by our grid search did not seem to improve the recall score. This can potentially be due to the limitation of the model itself or more likely is the limitations of our dataset. We simply may not have enough information in the data to more accurately predict the popularity of a song.

INTERPRETATION

Parsing Feature Importances to Dataframes

In [71]:
```
#accessing feature importance values of the tuned random forest model and sorting them
rf_importances_df = pd.Series(clf_rf_tuned.feature_importances_, index=X_train.columns).sort_values(ascending=False)
#parsing the series to a dataframe
```

```python
rf_importances_df = rf_importances_df.reset_index()
rf_importances_df.columns = ['RF-Attribute', 'RF-Importance']
rf_importances_df
```

Out[71]:

	RF-Attribute	RF-Importance
0	Pop	0.172521
1	acousticness	0.058269
2	loudness	0.042163
3	instrumentalness	0.034944
4	energy	0.030286
5	speechiness	0.027851
6	Reggae	0.025021
7	Ska	0.025001
8	danceability	0.024611
9	valence	0.023468
10	Rock	0.023411
11	duration_ms	0.022824
12	Anime	0.022618
13	key_C	0.022361
14	Electronic	0.021811
15	Reggaeton	0.021767
16	key_G	0.021063
17	key_D	0.021042
18	liveness	0.020094
19	Blues	0.020038
20	key_C#	0.017728
21	time_signature_4/4	0.017353
22	tempo	0.016592
23	Country	0.016454
24	key_E	0.016420
25	key_F	0.016397
26	World	0.016354
27	key_B	0.015852
28	Jazz	0.014745
29	Soul	0.014088
30	key_A#	0.013939
31	key_G#	0.013914

32	Rap	0.013426
33	key_F#	0.012316
34	Movie	0.012139
35	Folk	0.011197
36	Comedy	0.009446
37	R&B	0.008796
38	Children's Music	0.008679
39	time_signature_3/4	0.008303
40	key_D#	0.006153
41	Indie	0.006087
42	Hip-Hop	0.006072
43	Alternative	0.005167
44	Soundtrack	0.004553
45	Dance	0.004519
46	Classical	0.004364
47	Opera	0.004043
48	mode_Minor	0.003190
49	time_signature_5/4	0.000376
50	time_signature_1/4	0.000122
51	A Capella	0.000049

In [72]:

```
#accessing feature importance values of the tuned logistic regression model and sorting them
logregcv_importances_df = pd.Series(clf_logregcv_tuned.coef_[0], index=X_train.columns).sort_values(ascending=False)
#parsing the series to a dataframe
logregcv_importances_df = logregcv_importances_df.reset_index()
logregcv_importances_df.columns = ['LogReg-Attribute', 'LogReg-Importance']
logregcv_importances_df
```

Out[72]:

	LogReg-Attribute	LogReg-Importance
0	Pop	0.602771
1	Rock	0.309625
2	danceability	0.117188
3	loudness	0.102003
4	Rap	0.097956
5	time_signature_4/4	0.088190
6	Dance	0.066666
7	duration_ms	0.023603
8	Hip-Hop	0.014995
9	speechiness	-0.005824

10	tempo	-0.011091
11	time_signature_1/4	-0.023202
12	Indie	-0.023908
13	acousticness	-0.026374
14	Alternative	-0.029339
15	time_signature_5/4	-0.030763
16	energy	-0.031201
17	A Capella	-0.034542
18	mode_Minor	-0.037863
19	liveness	-0.040335
20	instrumentalness	-0.042672
21	Soundtrack	-0.056094
22	Comedy	-0.075320
23	time_signature_3/4	-0.082947
24	valence	-0.097692
25	Classical	-0.103310
26	R&B	-0.107485
27	Opera	-0.140447
28	key_D#	-0.146707
29	Children's Music	-0.150631
30	Jazz	-0.165628
31	Soul	-0.177036
32	Folk	-0.178010
33	key_F#	-0.198850
34	Electronic	-0.203208
35	key_A#	-0.214185
36	key_G#	-0.217818
37	key_B	-0.232801
38	key_E	-0.235656
39	Movie	-0.240680
40	key_F	-0.245328
41	key_C#	-0.245346
42	World	-0.247186
43	Country	-0.253768
44	Blues	-0.259086

45	Reggaeton	-0.265264
46	key_G	-0.279558
47	key_D	-0.280967
48	Anime	-0.291578
49	Reggae	-0.296106
50	key_C	-0.296985
51	Ska	-0.313558

In [73]:
```
#Concatenating feature importances into a single dataframe
importances_df = pd.concat([rf_importances_df, logregcv_importances_df], axis=1)
importances_df
```

Out[73]:

	RF-Attribute	RF-Importance	LogReg-Attribute	LogReg-Importance
0	Pop	0.172521	Pop	0.602771
1	acousticness	0.058269	Rock	0.309625
2	loudness	0.042163	danceability	0.117188
3	instrumentalness	0.034944	loudness	0.102003
4	energy	0.030286	Rap	0.097956
5	speechiness	0.027851	time_signature_4/4	0.088190
6	Reggae	0.025021	Dance	0.066666
7	Ska	0.025001	duration_ms	0.023603
8	danceability	0.024611	Hip-Hop	0.014995
9	valence	0.023468	speechiness	-0.005824
10	Rock	0.023411	tempo	-0.011091
11	duration_ms	0.022824	time_signature_1/4	-0.023202
12	Anime	0.022618	Indie	-0.023908
13	key_C	0.022361	acousticness	-0.026374
14	Electronic	0.021811	Alternative	-0.029339
15	Reggaeton	0.021767	time_signature_5/4	-0.030763
16	key_G	0.021063	energy	-0.031201
17	key_D	0.021042	A Capella	-0.034542
18	liveness	0.020094	mode_Minor	-0.037863
19	Blues	0.020038	liveness	-0.040335
20	key_C#	0.017728	instrumentalness	-0.042672
21	time_signature_4/4	0.017353	Soundtrack	-0.056094
22	tempo	0.016592	Comedy	-0.075320
23	Country	0.016454	time_signature_3/4	-0.082947

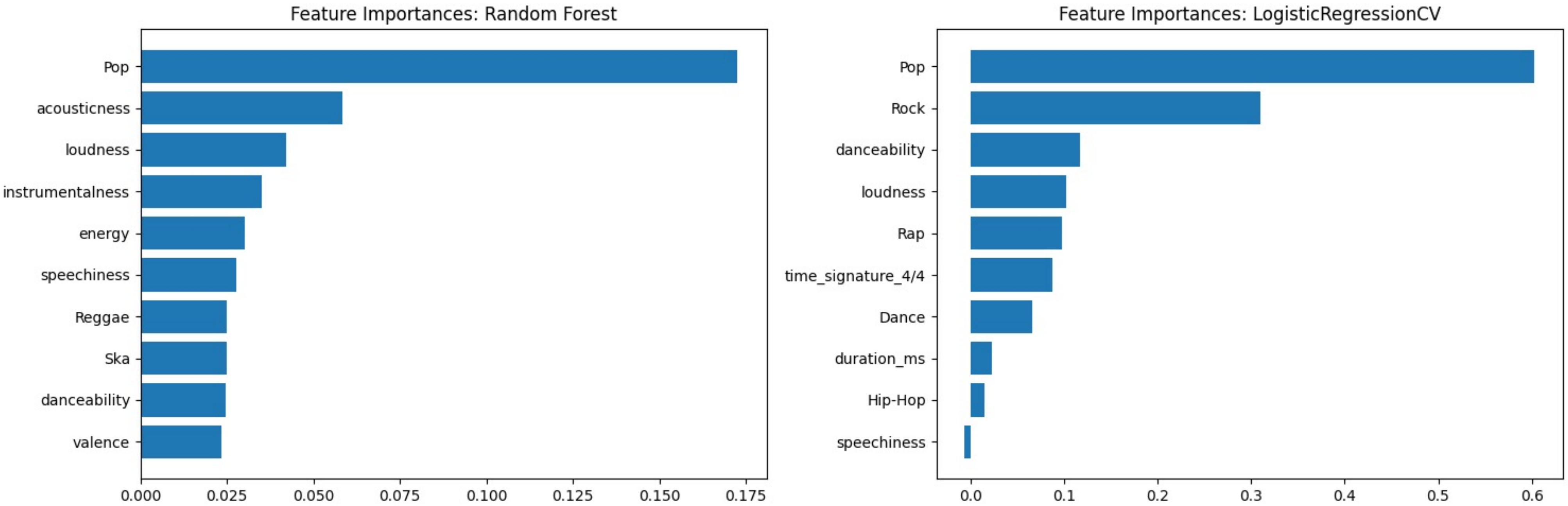
24	key_E	0.016420	valence	-0.097692
25	key_F	0.016397	Classical	-0.103310
26	World	0.016354	R&B	-0.107485
27	key_B	0.015852	Opera	-0.140447
28	Jazz	0.014745	key_D#	-0.146707
29	Soul	0.014088	Children's Music	-0.150631
30	key_A#	0.013939	Jazz	-0.165628
31	key_G#	0.013914	Soul	-0.177036
32	Rap	0.013426	Folk	-0.178010
33	key_F#	0.012316	key_F#	-0.198850
34	Movie	0.012139	Electronic	-0.203208
35	Folk	0.011197	key_A#	-0.214185
36	Comedy	0.009446	key_G#	-0.217818
37	R&B	0.008796	key_B	-0.232801
38	Children's Music	0.008679	key_E	-0.235656
39	time_signature_3/4	0.008303	Movie	-0.240680
40	key_D#	0.006153	key_F	-0.245328
41	Indie	0.006087	key_C#	-0.245346
42	Hip-Hop	0.006072	World	-0.247186
43	Alternative	0.005167	Country	-0.253768
44	Soundtrack	0.004553	Blues	-0.259086
45	Dance	0.004519	Reggaeton	-0.265264
46	Classical	0.004364	key_G	-0.279558
47	Opera	0.004043	key_D	-0.280967
48	mode_Minor	0.003190	Anime	-0.291578
49	time_signature_5/4	0.000376	Reggae	-0.296106
50	time_signature_1/4	0.000122	key_C	-0.296985
51	A Capella	0.000049	Ska	-0.313558

Feature Importance Comparison

In [74]:

```
#plotting feature importances for all models for comparison

fig, ax = plt.subplots(ncols=2, figsize=(15,5))

rf_importances_df = rf_importances_df.sort_values(by='RF-Importance', ascending=True).tail(10)
ax[0].barh(rf_importances_df['RF-Attribute'], rf_importances_df['RF-Importance'])
ax[0].set_title('Feature Importances: Random Forest')
```

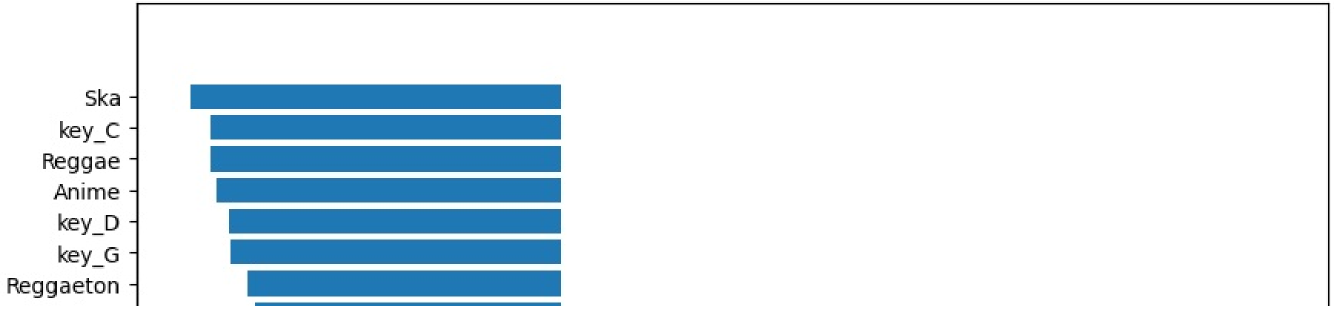
```
logregcv_importances_df = logregcv_importances_df.sort_values(by='LogReg-Importance', ascending=True).tail(10)
ax[1].barh(logregcv_importances_df['LogReg-Attribute'], logregcv_importances_df['LogReg-Importance'])
ax[1].set_title('Feature Importances: LogisticRegressionCV')
plt.tight_layout()
```

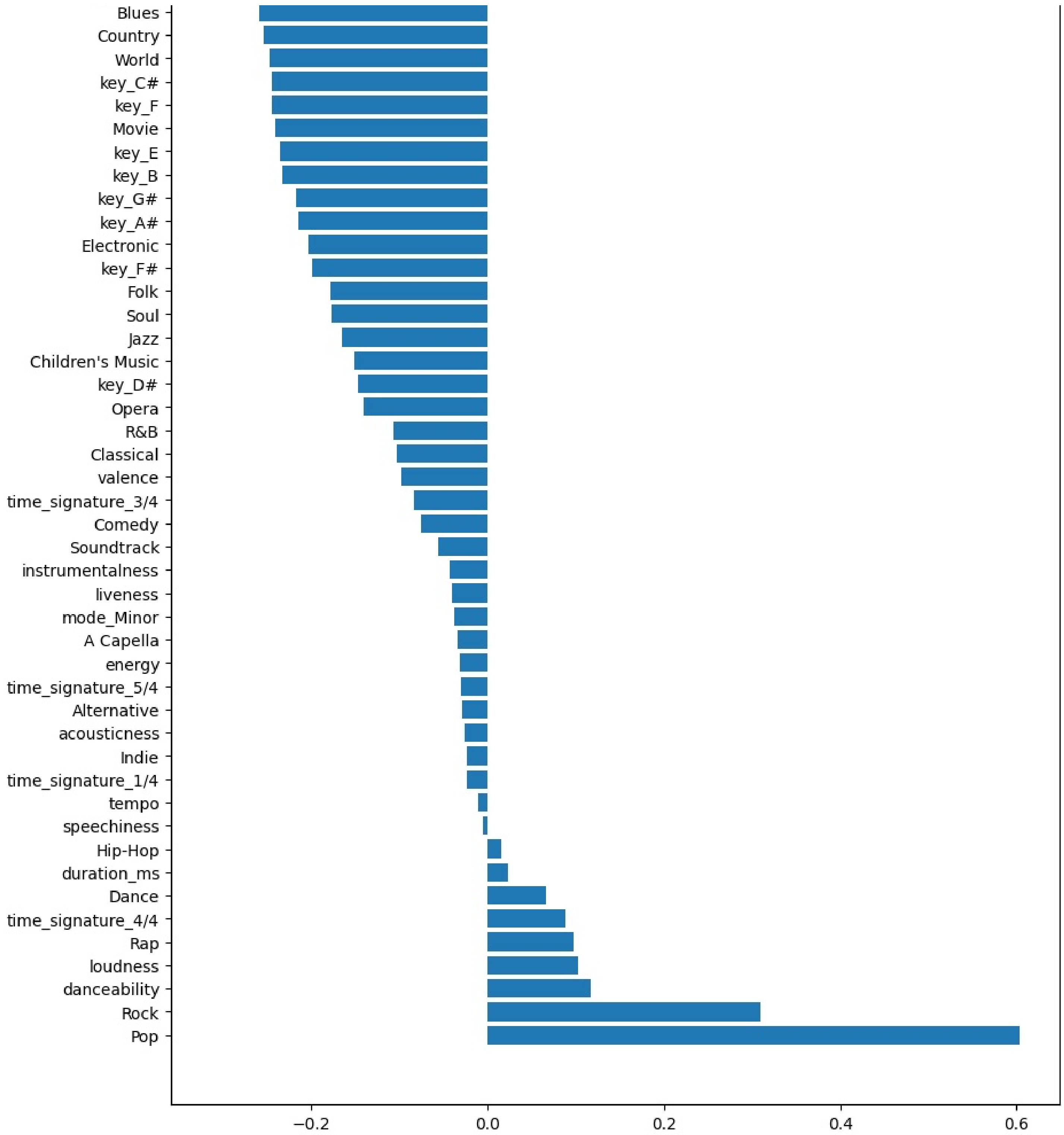


Among the 2 models we built we can see that Genre of a song has the highest effect on the popularity of a song. On THE 2 models, a song having Pop as its genre had the most impact on its popularity. This makes sense since Pop songs by nature are considered popular. Among the rest of the features shown above, different attribute scores such as danceability, energy, different genres and acousticness play a major role. Next, we can inspect the full gamut of the feature importances for logistic regression for reference.

In [75]:
```
logregcv_importances_df = pd.Series(clf_logregcv_tuned.coef_[0], index=X_train.columns).sort_values(ascending=False)
#parsing the series to a dataframe
logregcv_importances_df = logregcv_importances_df.reset_index()
logregcv_importances_df.columns = ['Attribute', 'Importance']

fig, ax = plt.subplots(figsize=(10,15))
ax.barh(logregcv_importances_df['Attribute'], logregcv_importances_df['Importance'])
```

Out[75]: <BarContainer object of 52 artists>




Blues
Country
World
key_C#
key_F
Movie
key_E
key_B
key_G#
key_A#
Electronic
key_F#
Folk
Soul
Jazz
Children's Music
key_D#
Opera
R&B
Classical
valence
time_signature_3/4
Comedy
Soundtrack
instrumentalness
liveness
mode_Minor
A Capella
energy
time_signature_5/4
Alternative
acousticness
Indie
time_signature_1/4
tempo
speechiness
Hip-Hop
duration_ms
Dance
time_signature_4/4
Rap
loudness
danceability
Rock
Pop
−0.2
0.0
0.2
0.4
0.6

We can see here that while certain features like 'Pop', 'Rock' and 'danceability' positively affected the prediction, other features such as 'Ska', 'Anime' and 'key_G' negatively affected it. Next we can dive into our processed dataframe and explore some of these attributes for popular and unpopular songs to come to conclusions.

Data Visualizations

Genre

In [76]:

```
#separating popular and unpopular songs to two dfs
popular_songs_df = df_ohe[df_ohe['is_popular'] == 1]
unpopular_songs_df = df_ohe[df_ohe['is_popular']==0]
```

In [77]:

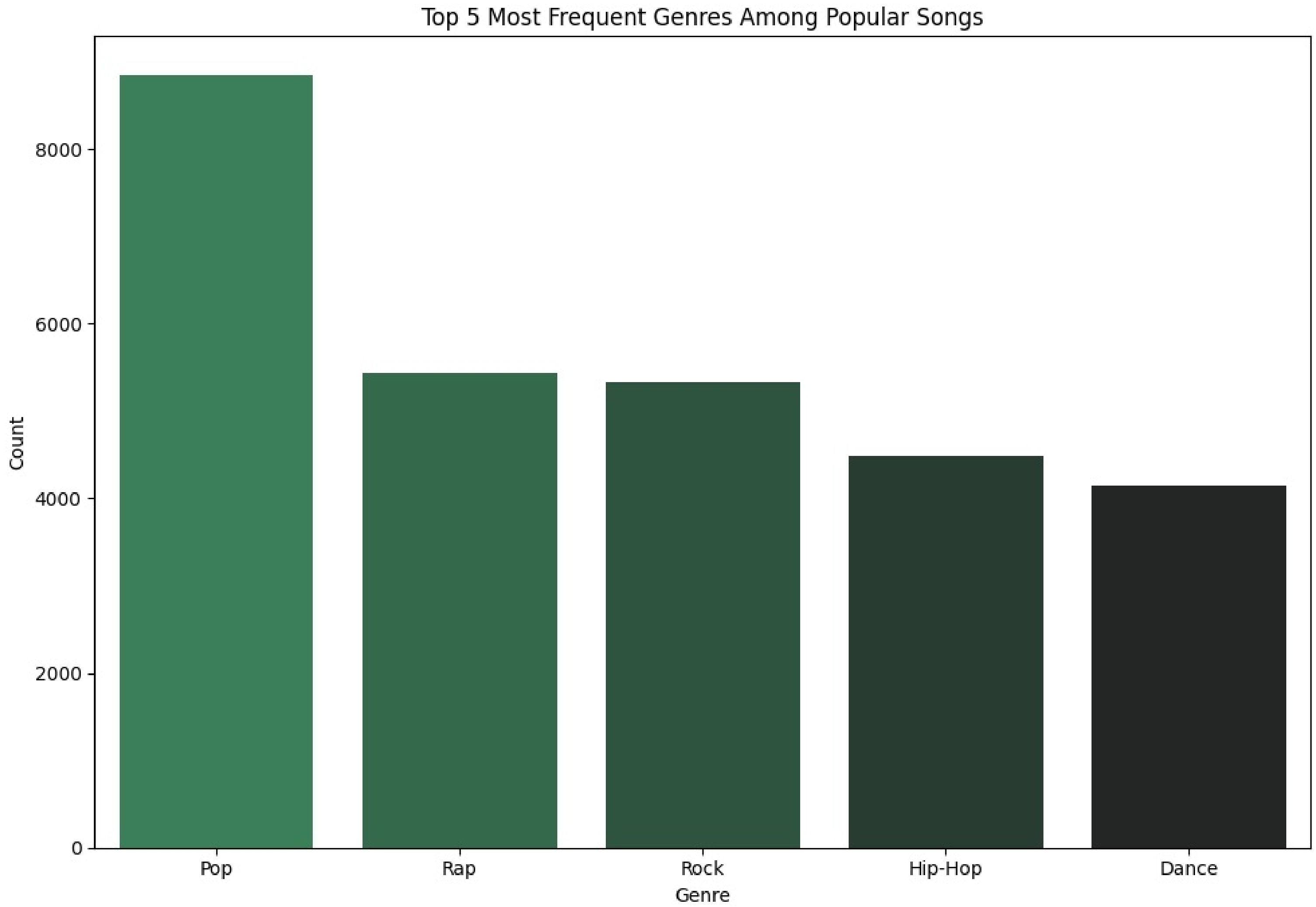
```
#checking for genre occurence counts for popular songs
popular_genre_df = popular_songs_df.iloc[:, 10:36].agg('sum').sort_values(ascending=False).reset_index()
popular_genre_df.columns = ['genre', 'count']
popular_genre_df
```

Out[77]:

	genre	count
0	Pop	8845
1	Rap	5440
2	Rock	5332
3	Hip-Hop	4483
4	Dance	4151
5	Indie	3096
6	Children's Music	3079
7	Alternative	2713
8	R&B	2347
9	Folk	1658
10	Soul	1205
11	Country	1088
12	Reggaeton	841
13	Blues	398
14	Jazz	368
15	Electronic	333
16	Reggae	301
17	World	221
18	Ska	120
19	Soundtrack	102
20	Classical	87
21	Movie	69
22	Anime	35
23	Opera	3
24	Comedy	1
25	A Capella	0

In [78]:

```
fig, ax = plt.subplots(figsize=(10, 7))
sns.barplot(x=popular_genre_df['genre'].head(5), y=popular_genre_df['count'].head(5),
           palette='dark:seagreen_r')

ax=plt.gca()
ax.set_xlabel('Genre')
ax.set_ylabel('Count')
ax.set_title('Top 5 Most Frequent Genres Among Popular Songs')
plt.tight_layout()
```



Above bar graph shows us the most frequent genres among popular songs. As we discussed above, most popular songs have Pop as their genre followed by Rap, Rock, Hip-Hop and Dance.

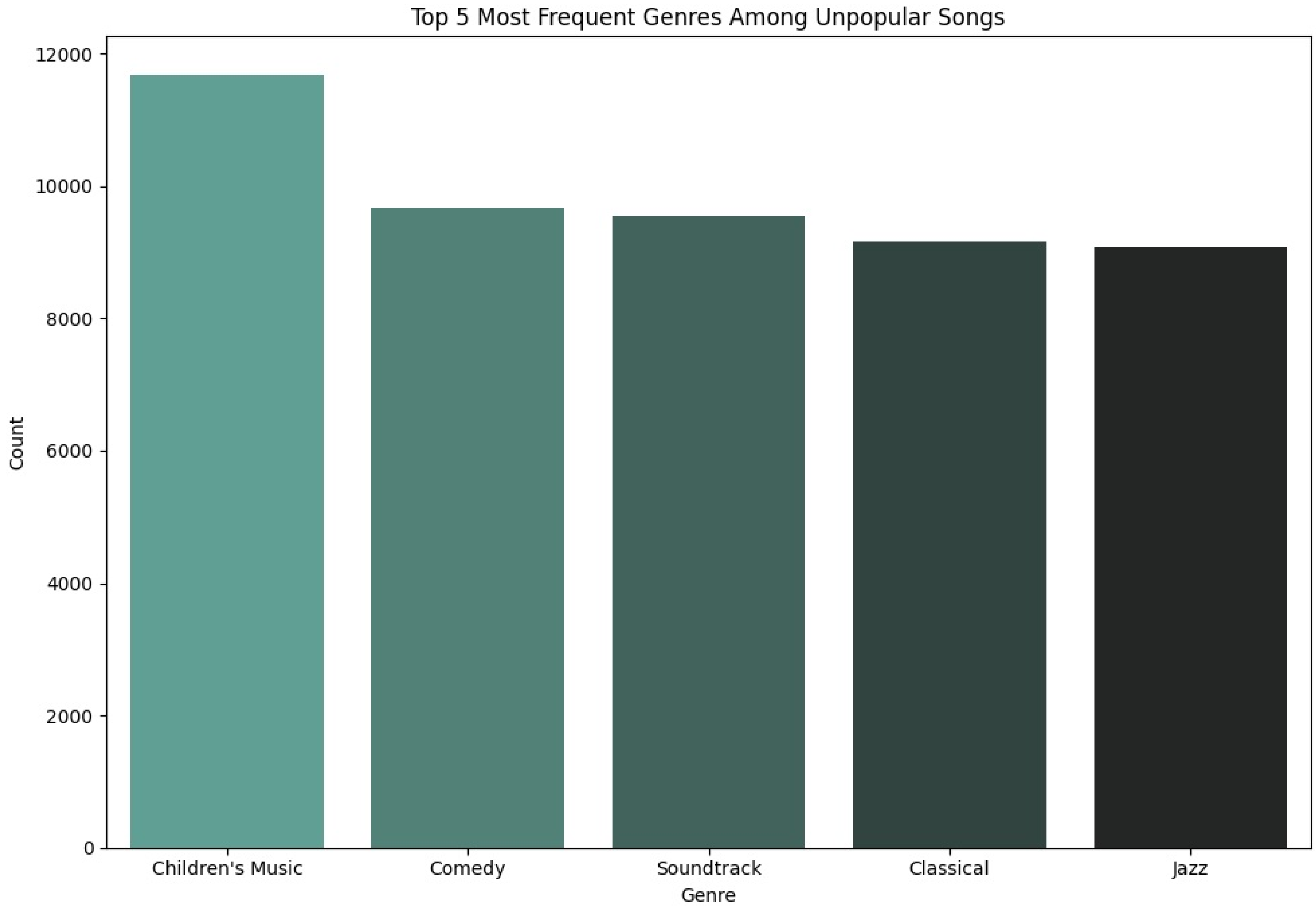
In [79]:
```
#checking for genre occurence counts for unpopular songs
unpopular_genre_df = unpopular_songs_df.iloc[:, 10:36].agg('sum').sort_values(ascending=False).reset_index()
unpopular_genre_df.columns = ['genre', 'count']
unpopular_genre_df
```

Out[79]:

	genre	count
0	Children's Music	11677
1	Comedy	9680
2	Soundtrack	9544
3	Classical	9169
4	Jazz	9073
5	Electronic	9044
6	Anime	8901
7	World	8875
8	Ska	8754
9	Blues	8625
10	Reggae	8470
11	Opera	8277
12	Reggaeton	8086
13	Soul	7884
14	Movie	7737
15	Folk	7641
16	Country	7576
17	R&B	6645
18	Alternative	6550
19	Indie	6447
20	Hip-Hop	4812
21	Dance	4550
22	Rock	3940
23	Rap	3792
24	Pop	541
25	A Capella	119

In [80]:

```
fig, ax = plt.subplots(figsize=(10,7))
sns.barplot(x=unpopular_genre_df['genre'].head(5), y=unpopular_genre_df['count'].head(5),
           palette='dark:#5A9_r')

ax=plt.gca()
ax.set_xlabel('Genre')
ax.set_ylabel('Count')
ax.set_title('Top 5 Most Frequent Genres Among Unpopular Songs')
plt.tight_layout();
```



The most frequent genres of unpopular songs can be seen above. The results make sense as these genres tend to have a more niche fanbase or as in the case of "Children's Music" are listened to infrequently.

In [81]:
```
#displaying percentages for each genre
popular_genre_df['count']=popular_genre_df['count']/popular_genre_df['count'].sum()
popular_genre_df
```

Out[81]:

	genre	count
0	Pop	0.190971
1	Rap	0.117454
2	Rock	0.115122
3	Hip-Hop	0.096792
4	Dance	0.089623
5	Indie	0.066845
6	Children's Music	0.066478
7	Alternative	0.058576
8	R&B	0.050674
9	Folk	0.035798
10	Soul	0.026017
11	Country	0.023491
12	Reggaeton	0.018158
13	Blues	0.008593
14	Jazz	0.007945
15	Electronic	0.007190
16	Reggae	0.006499
17	World	0.004772
18	Ska	0.002591
19	Soundtrack	0.002202
20	Classical	0.001878
21	Movie	0.001490
22	Anime	0.000756
23	Opera	0.000065
24	Comedy	0.000022
25	A Capella	0.000000

In [82]:

```
#displaying percentages for each genre
unpopular_genre_df['count']=unpopular_genre_df['count']/unpopular_genre_df['count'].sum()
unpopular_genre_df
```

Out[82]:

	genre	count
0	Children's Music	0.062642
1	Comedy	0.051929
2	Soundtrack	0.051199
3	Classical	0.049188
4	Jazz	0.048673
5	Electronic	0.048517
6	Anime	0.047750
7	World	0.047610
8	Ska	0.046961
9	Blues	0.046269
10	Reggae	0.045438
11	Opera	0.044402
12	Reggaeton	0.043378
13	Soul	0.042294
14	Movie	0.041506
15	Folk	0.040991
16	Country	0.040642
17	R&B	0.035647
18	Alternative	0.035138
19	Indie	0.034585
20	Hip-Hop	0.025814
21	Dance	0.024409
22	Rock	0.021136
23	Rap	0.020342
24	Pop	0.002902
25	A Capella	0.000638

Energy

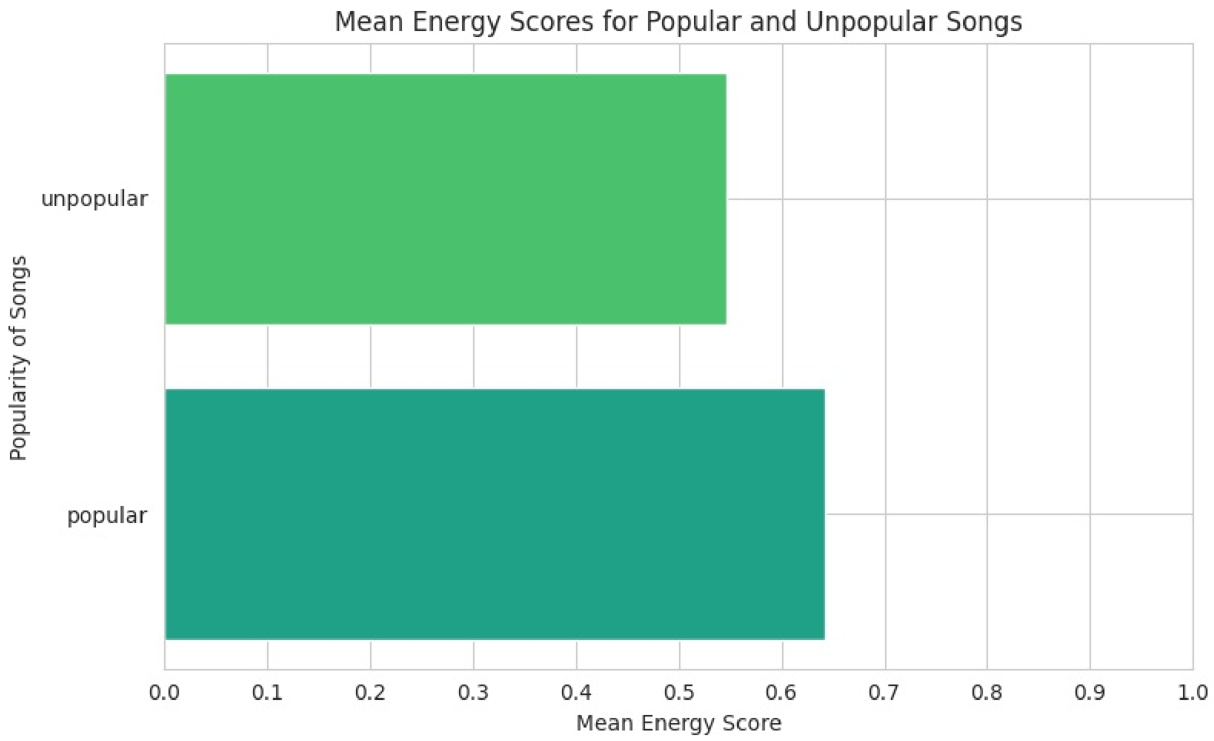
In [83]:

```
#removing outliers from energy scores and separating them to Series for popular and unpopular songs
popular_energy_clean = popular_songs_df[find_outliers_IQR(popular_songs_df['energy'])==False]
print(popular_energy_clean['energy'].describe())

unpopular_energy_clean = unpopular_songs_df[find_outliers_IQR(unpopular_songs_df['energy'])==False]
print(unpopular_energy_clean['energy'].describe())
```

```
count    20040.000000
mean         0.642509
std          0.195809
min          0.074000
25%          0.511000
50%          0.662000
75%          0.796000
max          0.999000
Name: energy, dtype: float64
count    156575.000000
mean          0.546617
std           0.282264
min           0.000020
25%           0.318000
50%           0.578000
75%           0.788000
max           0.999000
Name: energy, dtype: float64
```

In [84]:

```
import numpy as np

#storing mean energy scores in dict
mean_energy = {'popular': popular_energy_clean['energy'].mean(),
                     'unpopular': unpopular_energy_clean['energy'].mean()}
#visualizing mean scores
with sns.axes_style("whitegrid"):
    fig, ax = plt.subplots(figsize=(8,5))
    ax.barh(y=list(mean_energy.keys()),
            width=list(mean_energy.values()),
            color=[sns.color_palette('viridis')[3],sns.color_palette('viridis')[4]])
    ax.set_xlim(0, 1)
    ax.set_xticks(np.arange(0,1.1,0.1))
    ax.set_ylabel('Popularity of Songs')
    ax.set_xlabel('Mean Energy Score')
    ax.set_title('Mean Energy Scores for Popular and Unpopular Songs')
    plt.tight_layout()
```



As we can see above, popular songs tended to be more energetic compared to unpopular songs. This makes sense since the most frequent genres we explored tend to also be energetic genres.

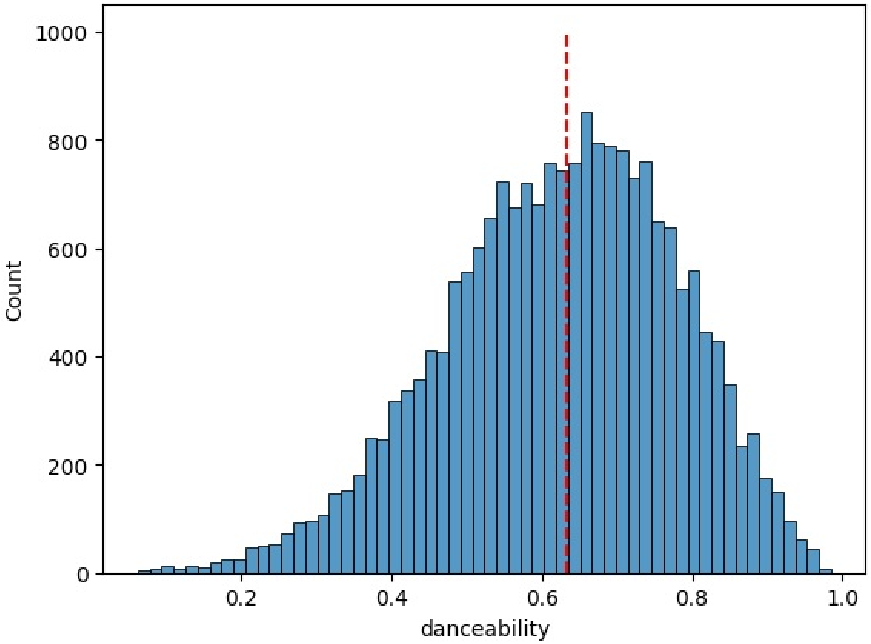
Danceability

In [85]:
```
print('Median Danceability Scores')
print('-------------------')
print(f"Unpopular Songs: {round(unpopular_songs_df['danceability'].median(),2)}")
print(f"Popular Songs: {round(popular_songs_df['danceability'].median(),2)}")
```

```
Median Danceability Scores
-------------------
Unpopular Songs: 0.55
Popular Songs: 0.63
```

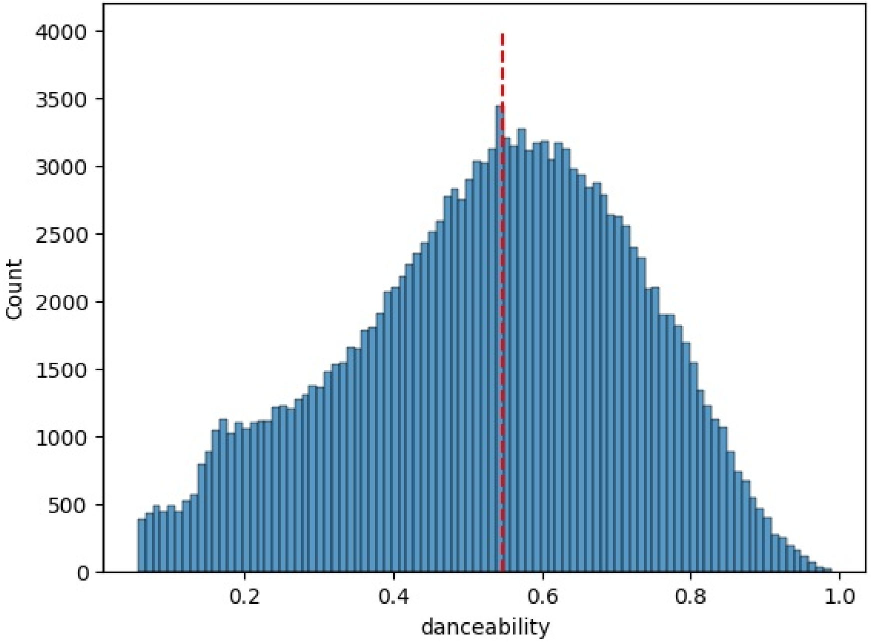
In [86]:
```
sns.histplot(data = popular_songs_df, x='danceability', bins='auto')
plt.vlines(x=popular_songs_df['danceability'].median(), ymin=0, ymax=1000, color='red', ls='--')
```

Out[86]:
```
<matplotlib.collections.LineCollection at 0x159defe2c90>
```



In [87]:
```
sns.histplot(data = unpopular_songs_df, x='danceability', bins='auto')
plt.vlines(x=unpopular_songs_df['danceability'].median(), ymin=0, ymax=4000, color='red', ls='--')
```

Out[87]:
```
<matplotlib.collections.LineCollection at 0x159ccc96010>
```

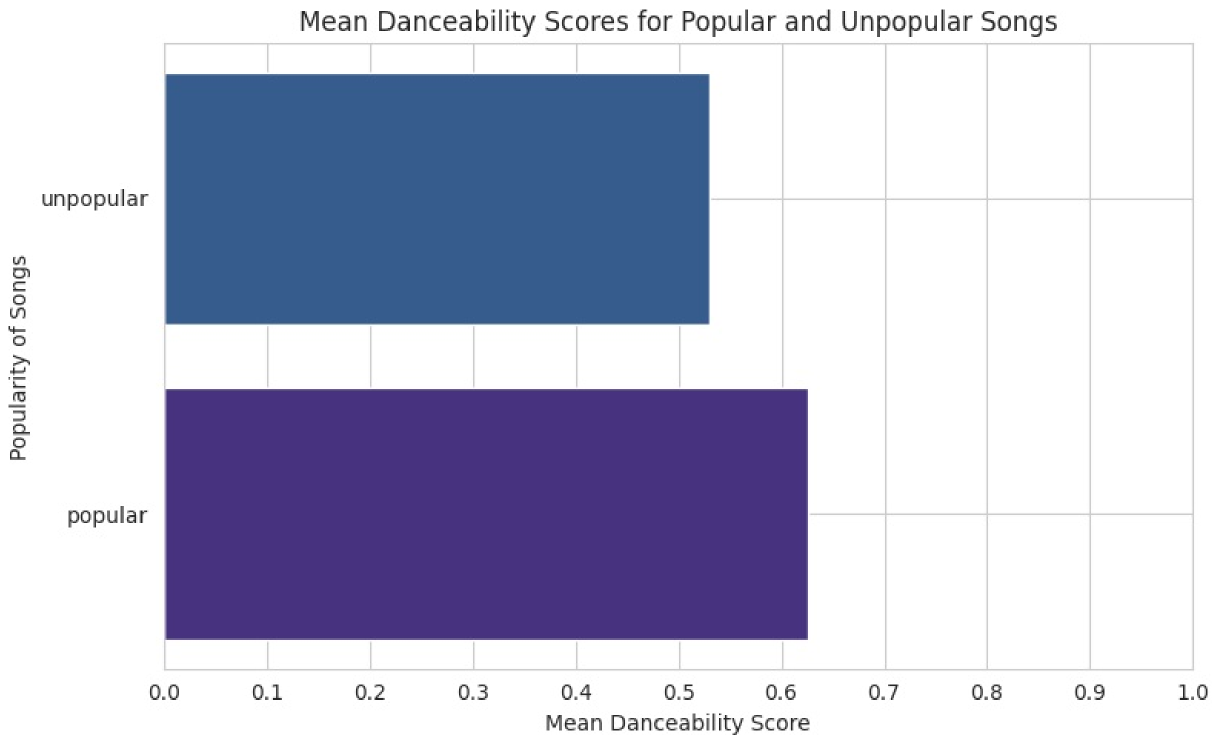


In [88]:
```
#removing outliers from danceability scores and separating them to Series for popular and unpopular songs
popular_dance_clean = popular_songs_df[find_outliers_IQR(popular_songs_df['danceability'])==False]
print(popular_dance_clean['danceability'].describe())

unpopular_dance_clean = unpopular_songs_df[find_outliers_IQR(unpopular_songs_df['danceability'])==False]
print(unpopular_dance_clean['danceability'].describe())
```

```
count    20094.000000
mean         0.625974
std          0.151130
min          0.196000
25%          0.523000
50%          0.636000
75%          0.738000
max          0.985000
Name: danceability, dtype: float64
count    156575.000000
mean          0.530440
std           0.191956
min           0.056900
25%           0.401000
50%           0.547000
75%           0.674000
max           0.989000
Name: danceability, dtype: float64
```

In [89]:

```python
#storing mean danceability scores in dict
mean_danceability = {'popular': popular_dance_clean['danceability'].mean(),
                     'unpopular': unpopular_dance_clean['danceability'].mean()}

#visualizing mean scores
with sns.axes_style("whitegrid"):
    fig, ax = plt.subplots(figsize=(8,5))
    ax.barh(y=list(mean_danceability.keys()),
            width=list(mean_danceability.values()),
            color=[sns.color_palette('viridis')[0],sns.color_palette('viridis')[1]])
    ax.set_xlim(0, 1)
    ax.set_xticks(np.arange(0,1.1,0.1))
    ax.set_ylabel('Popularity of Songs')
    ax.set_xlabel('Mean Danceability Score')
    ax.set_title('Mean Danceability Scores for Popular and Unpopular Songs')
    plt.tight_layout()
```



Above, it is clear that the popular songs tended to have a higher danceability score compared to unpopular songs. This follows the same trend as the energy scores where majority of the popular songs are high energy and danceable (refer to Appendix A for definition of "danceability": high tempo, high beat strength etc.)

Acousticness

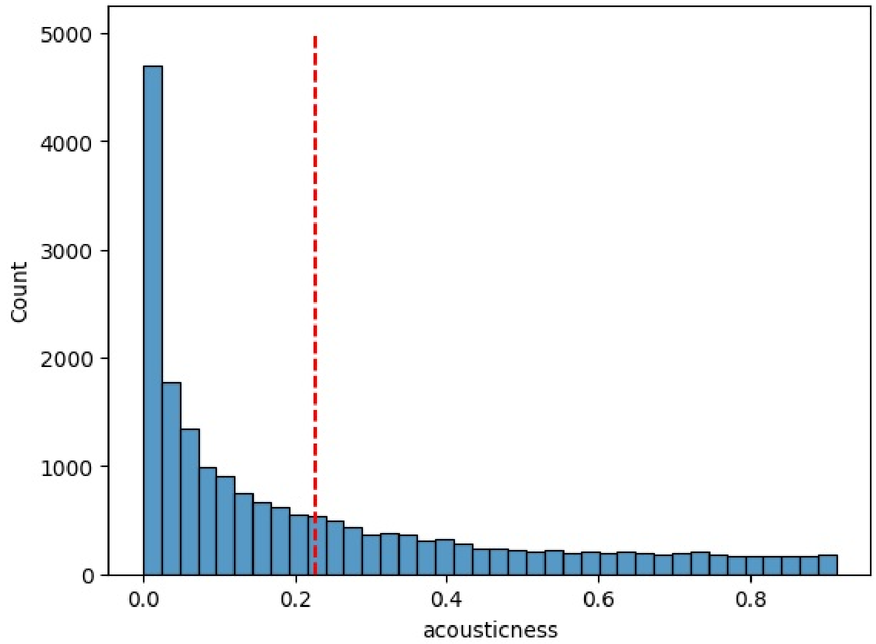
In [90]:

```
#removing outliers from danceability scores and separating them to Series for popular and unpopular songs
popular_acoustic_clean = popular_songs_df[find_outliers_IQR(popular_songs_df['acousticness'])==False]
print(popular_acoustic_clean['acousticness'].describe())

unpopular_acoustic_clean = unpopular_songs_df[find_outliers_IQR(unpopular_songs_df['acousticness'])==False]
print(unpopular_acoustic_clean['acousticness'].describe())
```

```
count    19715.000000
mean         0.226220
std          0.248585
min          0.000002
25%          0.026400
50%          0.125000
75%          0.355000
max          0.913000
Name: acousticness, dtype: float64
count    156575.000000
mean          0.424829
std           0.371949
min           0.000000
25%           0.049800
50%           0.329000
75%           0.819000
max           0.996000
Name: acousticness, dtype: float64
```

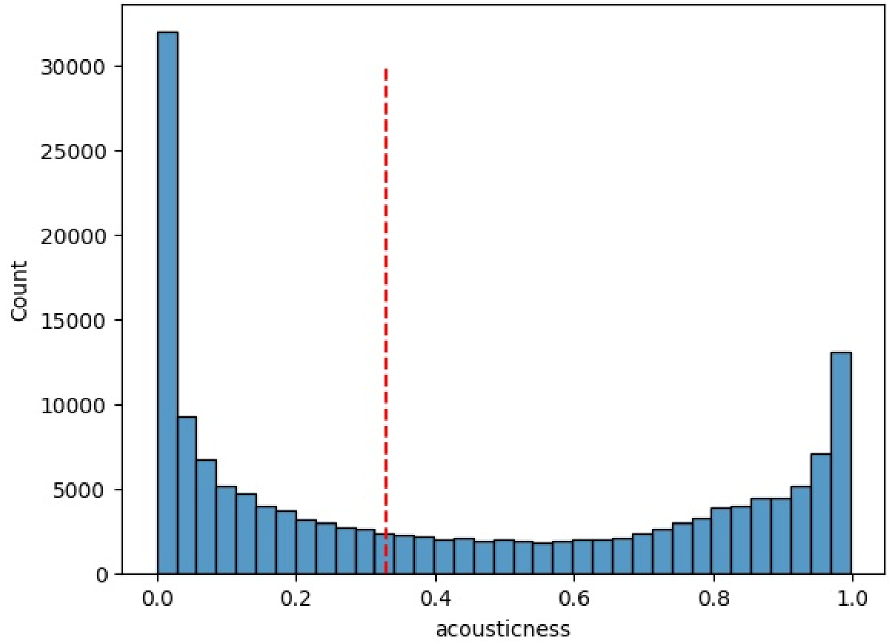
In [91]:
```
sns.histplot(data = popular_acoustic_clean, x='acousticness', bins='auto')
plt.vlines(x=popular_acoustic_clean['acousticness'].mean(), ymin=0, ymax=5000, color='red', ls='--')
```

Out[91]: <matplotlib.collections.LineCollection at 0x159cc9fb950>

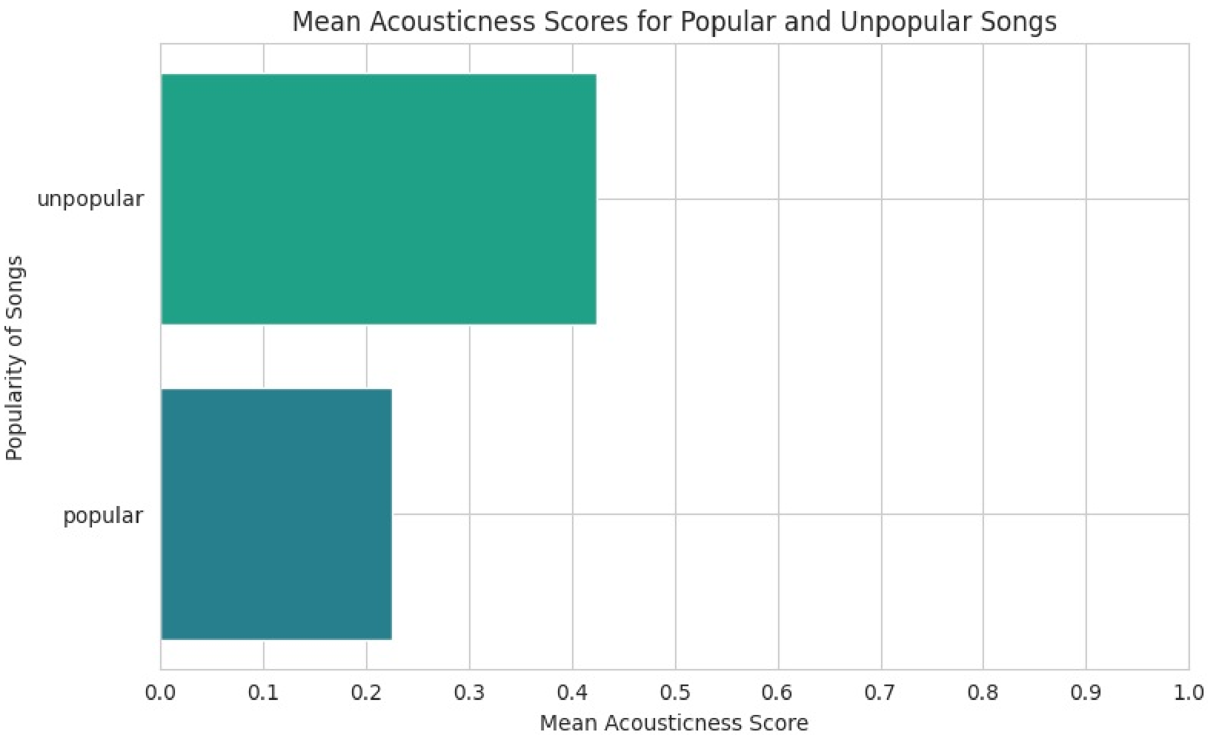


In [92]:
```
sns.histplot(data = unpopular_songs_df, x='acousticness', bins='auto')
plt.vlines(x=unpopular_songs_df['acousticness'].median(), ymin=0, ymax=30000, color='red', ls='--')
```

Out[92]: <matplotlib.collections.LineCollection at 0x159c7a7b210>



In [93]:
```
#storing mean acousticness scores in dict
mean_acousticness = {'popular': popular_acoustic_clean['acousticness'].mean(),
                     'unpopular': unpopular_acoustic_clean['acousticness'].mean()}

#visualizing mean scores
with sns.axes_style("whitegrid"):
    fig, ax = plt.subplots(figsize=(8,5))
    ax.barh(y=list(mean_acousticness.keys()),
            width=list(mean_acousticness.values()),
            color=[sns.color_palette('viridis')[2],sns.color_palette('viridis')[3]])
    ax.set_xlim(0, 1)
    ax.set_xticks(np.arange(0,1.1,0.1))
    ax.set_ylabel('Popularity of Songs')
    ax.set_xlabel('Mean Acousticness Score')
    ax.set_title('Mean Acousticness Scores for Popular and Unpopular Songs')
    plt.tight_layout()
```



In a competitive environment like the music streaming market, it is vital to retain current subscribers and add new subscribers over time. By accurately predicting which song will be popular next, companies like Spotify can leverage this information to create better playlists and find and sign exclusivity deals with established and up-and-coming artists more easily. To sum up, our analysis of approximately 176,000 songs from 2019 showed the following:

Popular songs tend to have Pop, Rap, Rock, Hip-Hop and Dance as their genres. More niche genres such as Children's Music, Comedy, Soundtracks, Classical and Jazz tend to be unpopular. Generally, popular songs are higher energy, danceable, and therefore less acoustic.

Our recommendations to Spotify for leveraging this information would be the following:

By identifying the next popular songs, Spotify can reach out to these artists and sign exclusivity deals with them to make their soon-to-be popular music available only on Spotify's platform. This would also help in identifying up-and-coming artists and may provide additional opportunities in the future.

Furthermore, Spotify can work with these artists on additional exclusive content such as song commentary or behind the scenes recordings.

Spotify can also curate even better playlists for their current subscribers by finding "fresh hits" ahead of the competition and use this to market the platform to new subscribers.

We think that by utilizing our model and the insights we've highlighted, Spotify will stay competitive in the music streaming market for years to come.