Music Streaming Wars: Song Popularity Prediction

Author: E. Berke Tezcan

TABLE OF CONTENTS

Click to jump to matching Markdown Header.

INTRODUCTION

With Apple Music announcing on May 17th that they will be providing lossless audio along with spatial audio by Dolby Atmos for their subscribers and Tidal continuously providing exclusive content from artists, the competition among audio streaming platforms is heating up. Spotify would like to stay competitive by being able to predict which songs are going to be popular ahead of time so that they can curate even better playlists and sign deals with up-and-coming artists to have exclusivity on their content. This would not only help retain the current subscribers but also help market the platform to new subscribers as well.

For this project, we were hired by Spotify to train and test a machine learning model that can accurately predict whether a song is going to be popular or not. In order to achieve this, we will be testing out different machine learning models and will look at what attributes of a song are the most important for determining its popularity.

OBTAIN

We will be using a dataset from Kaggle (https://www.kaggle.com/zaheenhamidani/ultimate-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]:
```
import pandas as pd
```

In [2]:
```
#importing data into a dataframe
df = pd.read_csv('./data/SpotifyFeatures.csv')
```

```
df.head()
```

Out[2]:

	genre	artist_name	track_name	track_id	popularity	acousticness	danceability	du
0	Movie	Henri Salvador	C'est beau de faire un Show	0BRjO6ga9RKCKjfDqeFgWV	0	0.611	0.389	
1	Movie	Martin & les fées	Perdu d'avance (par Gad Elmaleh)	0BjC1NfoEOOusryehmNudP	1	0.246	0.590	
2	Movie	Joseph Williams	Don't Let Me Be Lonely Tonight	0CoSDzoNIKCRs124s9uTVy	3	0.952	0.663	
3	Movie	Henri Salvador	Dis-moi Monsieur Gordon Cooper	0Gc6TVm52BwZD07Ki6tIvf	0	0.703	0.240	
4	Movie	Fabien Nataf	Ouverture	0IusIXpMROHdEPvSI1fTQK	4	0.950	0.331	

In [3]:
```
#looking at the stats of different columns
df.describe()
```

Out[3]:

	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	
count	232725.000000	232725.000000	232725.000000	2.327250e+05	232725.000000	232725.000000	23
mean	41.127502	0.368560	0.554364	2.351223e+05	0.570958	0.148301	
std	18.189948	0.354768	0.185608	1.189359e+05	0.263456	0.302768	
min	0.000000	0.000000	0.056900	1.538700e+04	0.000020	0.000000	
25%	29.000000	0.037600	0.435000	1.828570e+05	0.385000	0.000000	
50%	43.000000	0.232000	0.571000	2.204270e+05	0.605000	0.000044	
75%	55.000000	0.722000	0.692000	2.657680e+05	0.787000	0.035800	
max	100.000000	0.996000	0.989000	5.552917e+06	0.999000	0.999000	

In [4]:
```
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        232725 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]:
```
#looking at different values contained within columns
for col in df.columns:
    print(f"Column: {col}")
    print(df[col].value_counts())
    print("--------------------")
```

```
Column: 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: genre, dtype: int64
--------------------
Column: artist_name
Giuseppe Verdi            1394
Giacomo Puccini           1137
Kimbo Children's Music     971
Nobuo Uematsu              825
Richard Wagner             804
                          ...
```

```
Chrishan                    1
Duke Garwood                1
Joe Tex                     1
Alastair Greene             1
Jonathan Wilson             1
Name: artist_name, Length: 14564, dtype: int64
--------------------
Column: track_name
Home                                                                       100
You                                                                         71
Intro                                                                       69
Stay                                                                        63
Wake Up                                                                     59
                                                                          ...
Planting the Seeds of Insecurity                                             1
Feeling Blue                                                                 1
Sola Bonita                                                                  1
Bhaja Govindam                                                               1
Rigoletto, Act I, Scene 1: Partite? Crudele! (Duca/Contessa Ceprano)         1
Name: track_name, Length: 148615, dtype: int64
--------------------
Column: track_id
6AIte2Iej1QKlaofpjCzW1    8
0UE0RhnRaEYsiYgXpyLoZc    8
3R73Y7X53MIQZWnKloWq5i    8
0wY9rA9fJkuESyYm9uzVK5    8
3uSSjnDMmoyERaAK9KvpJR    8
                         ..
0hIB8O5Ha9x3rDjHtA7XLW    1
4G0uXPTWN9Tzep1MqwmOR7    1
3rnvrqjkkaKFu0rcFbKn6E    1
6BLTxuiS6APPYSz3XtNWsF    1
6TH2QNqd4l7TSerz5j9LpA    1
Name: track_id, Length: 176774, dtype: int64
--------------------
Column: popularity
0      6312
50     5415
53     5414
51     5401
52     5342
       ...
96        8
94        7
99        4
98        3
100       2
Name: popularity, Length: 101, dtype: int64
--------------------
Column: acousticness
0.995000    851
0.994000    701
0.992000    682
0.993000    646
0.991000    597
           ...
0.000005      1
0.000007      1
0.000098      1
0.000083      1
0.000009      1
Name: acousticness, Length: 4734, dtype: int64
--------------------
Column: danceability
0.5970    558
```

```
0.5470    544
0.6100    542
0.5890    542
0.6220    540
         ...
0.0584      1
0.0577      1
0.0570      1
0.0878      1
0.0596      1
Name: danceability, Length: 1295, dtype: int64
--------------------
Column: duration_ms
240000    138
180000    120
192000    115
216000     99
200000     85
         ...
258851      1
238377      1
164064      1
244522      1
262144      1
Name: duration_ms, Length: 70749, dtype: int64
--------------------
Column: energy
0.721000    417
0.675000    403
0.720000    392
0.686000    389
0.738000    389
           ...
0.002230      1
0.000216      1
0.006110      1
0.009910      1
0.007330      1
Name: energy, Length: 2517, dtype: int64
--------------------
Column: instrumentalness
0.00000    79236
0.91200      235
0.91000      230
0.91800      222
0.92300      222
           ...
0.00966        1
0.99900        1
0.00667        1
0.99800        1
0.00888        1
Name: instrumentalness, Length: 5400, dtype: int64
--------------------
Column: 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: key, dtype: int64
--------------------
Column: liveness
0.1110    2860
0.1100    2702
0.1080    2608
0.1090    2537
0.1070    2451
          ...
0.0240       1
0.0185       1
0.0200       1
0.0177       1
0.0143       1
Name: liveness, Length: 1732, dtype: int64
--------------------
Column: loudness
-5.318     57
-5.460     52
-5.131     51
-5.428     51
-6.611     50
           ..
-31.696     1
-38.267     1
-45.192     1
-28.588     1
-1.494      1
Name: loudness, Length: 27923, dtype: int64
--------------------
Column: mode
Major    151744
Minor     80981
Name: mode, dtype: int64
--------------------
Column: speechiness
0.0374    663
0.0332    654
0.0337    652
0.0363    650
0.0343    642
         ...
0.6070      1
0.6880      1
0.6620      1
0.6750      1
0.6670      1
Name: speechiness, Length: 1641, dtype: int64
--------------------
Column: tempo
120.016    61
100.003    60
100.014    60
120.008    59
120.003    59
           ..
82.571      1
94.596      1
62.067      1
91.555      1
110.206     1
Name: tempo, Length: 78512, dtype: int64
--------------------
```

```
Column: time_signature
4/4    200760
3/4     24111
5/4      5238
1/4      2608
0/4         8
Name: time_signature, dtype: int64
--------------------
Column: valence
0.9610    479
0.9620    403
0.9630    368
0.3700    363
0.3580    363
         ...
0.0232      1
0.0209      1
0.9950      1
0.0227      1
0.0180      1
Name: valence, 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.

SCRUB/EXPLORE

Addressing "Children's Music" Character Discrepancy

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

Out[6]:
```
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: genre, 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]:
```
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: genre, dtype: int64
```

Missing Values

In [9]:
```
#checking for missing values
df.isna().sum()
```

Out[9]:
```
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
```

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 [10]:
```
df[df['track_id'].duplicated()]
```

Out[10]:

	genre	artist_name	track_name	track_id	popularity	acousticness	danc
1348	Alternative	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.07160	
1385	Alternative	Frank Ocean	Seigfried	1BViPjTT585XAhkUUrkts0	61	0.97500	
1452	Alternative	Frank Ocean	Bad Religion	2pMPWE7PJH1PizfgGRMnR9	56	0.77900	
1554	Alternative	Steve Lacy	Some	4riDfclV7kPDT9D58FpmHd	58	0.00548	
1634	Alternative	tobi lou	Buff Baby	1F1QmI8TMHir9SUFrooq5F	59	0.19000	
...	...	...	...	...	...	...	
232715	Soul	Emily King	Down	5cA0vB8c9FMOVDWyJHgf26	42	0.55000	
232718	Soul	Muddy Waters	I Just Want To Make Love To You - Electric Mud...	2HFczeynfKGiM9KF2z2K7K	43	0.01360	
232720	Soul	Slave	Son Of Slide	2XGLdVl7lGeq8ksM6Al7jT	39	0.00384	
232722	Soul	Muddy Waters	(I'm Your) Hoochie Coochie Man	2ziWXUmQLrXTiYjCg2fZ2t	47	0.90100	
232723	Soul	R.LUM.R	With My Words	6EFsue2YbIG4Qkq8Zr9Rir	44	0.26200	

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 [11]:
```
df[df['track_id']=='6iOvnACn4ChlAw4lWUU4dd']
```

Out[11]:

	genre	artist_name	track_name	track_id	popularity	acousticness	dance
257	R&B	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	
1348	Alternative	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	
77710	Children's Music	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	
93651	Indie	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	

	genre	artist_name	track_name	track_id	popularity	acousticness	dance
113770	Pop	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	

In [12]:
```
df[df['track_id']=='2XGLdVl7lGeq8ksM6Al7jT']
```

Out[12]:

	genre	artist_name	track_name	track_id	popularity	acousticness	danceability
179212	Jazz	Slave	Son Of Slide	2XGLdVl7lGeq8ksM6Al7jT	39	0.00384	0.687
232720	Soul	Slave	Son Of Slide	2XGLdVl7lGeq8ksM6Al7jT	39	0.00384	0.687

In [13]:
```
df[df['track_id']=='2HFczeynfKGiM9KF2z2K7K']
```

Out[13]:

	genre	artist_name	track_name	track_id	popularity	acousticness	danceability
48555	Blues	Muddy Waters	I Just Want To Make Love To You - Electric Mud...	2HFczeynfKGiM9KF2z2K7K	35	0.0136	0.294
232718	Soul	Muddy Waters	I Just Want To Make Love To You - Electric Mud...	2HFczeynfKGiM9KF2z2K7K	43	0.0136	0.294

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 [14]:
```
#generating a list with the genre names
genre_list = list(df['genre'].unique())
```

In [15]:
```
#creating the genre columns using the genre list
for genre in genre_list:
    df[genre] = (df['genre']==genre).astype('int')
```

In [16]:
```
#grouping by track_id number to get rid of duplicates and keeping the maximum values in
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. If a track's best popularity score was 42 for example, we are keeping the best value by taking the max.

In [17]:
```
#removing redundant genre column
df.drop('genre', axis=1, inplace=True)
```

```python
df.head()
```

Out[17]:

track_id	artist_name	track_name	popularity	acousticness	danceability	duration_ms
00021Wy6AyMbLP2tqij86e	Capcom Sound Team	Zangief's Theme	13	0.234	0.617	169173
000CzNKC8PEt1yC3L8dqwV	Henri Salvador	Coeur Brisé à Prendre - Remastered	5	0.249	0.518	130653
000DfZJww8KiixTKuk9usJ	Mike Love	Earthlings	30	0.366	0.631	357573
000EWWBkYaREzsBplYjUag	Don Philippe	Fewerdolr	39	0.815	0.768	104924
000xQL6tZNLJzIrtIgxqSl	ZAYN	Still Got Time	70	0.131	0.748	188491

5 rows × 42 columns

In [18]:

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

Out[18]:

track_id	artist_name	track_name	popularity	acousticness	danceability	duration_m
6iOvnACn4ChlAw4lWUU4dd	Doja Cat	Go To Town	64	0.0716	0.71	21781

1 rows × 42 columns

We successfully addressed the duplicates of each track by aggregating them to a single row.

In [19]:

```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
```

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

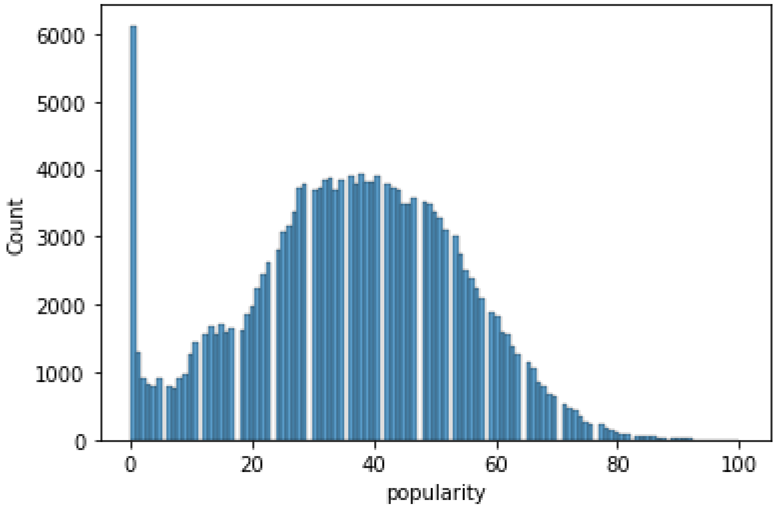
Feature Engineering - is_popular

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 [20]:
```
import matplotlib.pyplot as plt
import seaborn as sns
```

In [21]:
```
#creating a histogram to see distribution of popularity scores in the dataset.
sns.histplot(df['popularity'], bins='auto')
```

Out[21]: `<AxesSubplot: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)

Top 50 Songs - 2019

In [22]:
```
#data from https://www.kaggle.com/leonardopena/top50spotify2019
df_50 = pd.read_csv('data/top50.csv', encoding='latin1', index_col=0)
```

In [23]:
```
df_50.head()
```

Out[23]:

	Track.Name	Artist.Name	Genre	Beats.Per.Minute	Energy	Danceability	Loudness..dB..	Liveness
1	Señorita	Shawn Mendes	canadian pop	117	55	76	-6	8
2	China	Anuel AA	reggaeton flow	105	81	79	-4	8
3	boyfriend (with Social House)	Ariana Grande	dance pop	190	80	40	-4	16
4	Beautiful People (feat. Khalid)	Ed Sheeran	pop	93	65	64	-8	8
5	Goodbyes (Feat. Young Thug)	Post Malone	dfw rap	150	65	58	-4	11

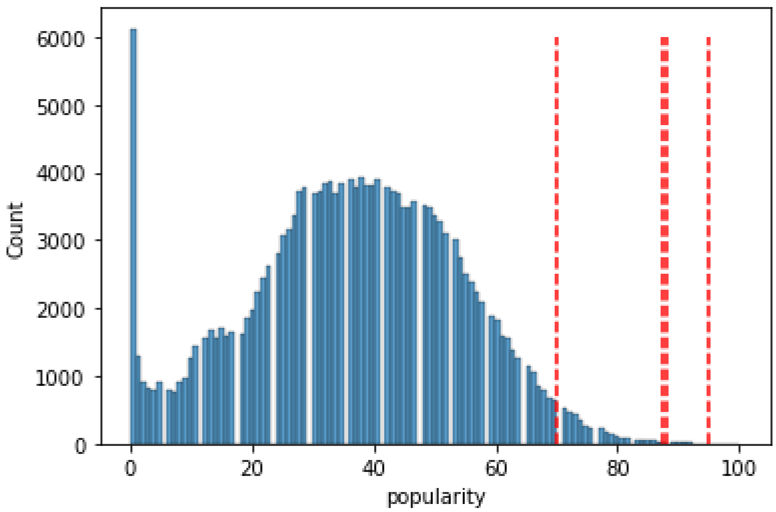
In [24]:
```
#displaying stats information of Top 50 songs
df_50['Popularity'].describe()
```

Out[24]:
```
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
```

Going back to our histogram we can draw vertical lines to see where these values fall into.

In [25]:
```
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='da
```



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. However, before we can establish the cutoff point we need to acknowledge that we are basing it off of only 50 datapoints which is not a lot. It may be good to take a look at Top 100 songs instead of 50 to get a better sample size of popular songs.

Top 100 Songs - 2019

In [26]:
```
#data from https://www.kaggle.com/reach2ashish/top-100-spotify-songs-2019
df_100 = pd.read_csv('data/spotify_top_100_2019.csv')
```

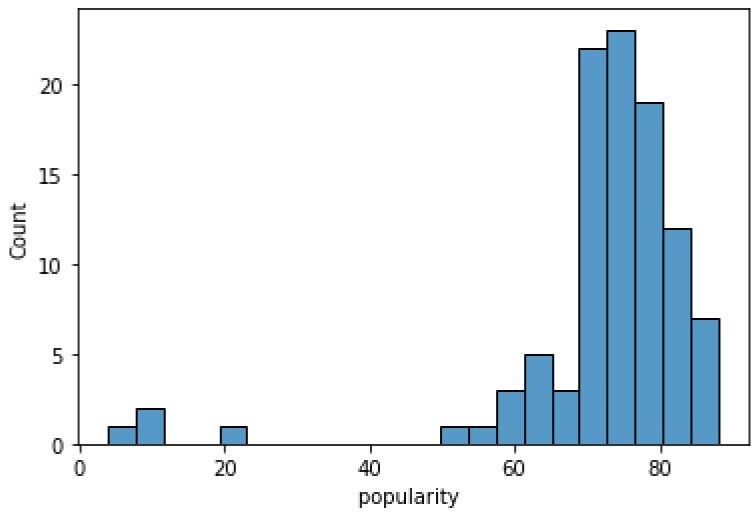
In [27]:
```
df_100['popularity '].describe()
```

Out[27]:
```
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 [28]:
```
fig, ax = plt.subplots()
sns.histplot(df_100['popularity '], bins='auto', ax=ax)
```

Out[28]: <AxesSubplot:xlabel='popularity ', ylabel='Count'>



As we imagined the scores within the range 0-25 seem like outliers. We can remove outliers from this dataset with the IQR method to get a better perspective on the data.

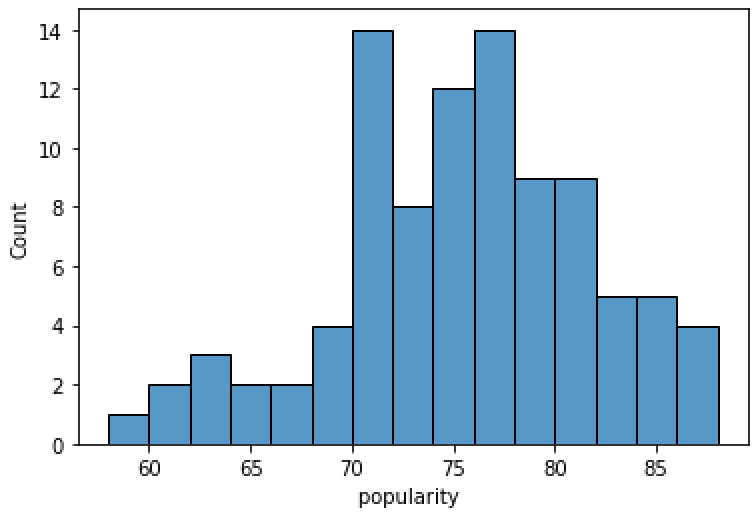
In [29]:
```
#Outlier Removal with the IQR method

def find_outliers_IQR(data):
    """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

    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.

    EXAMPLE USE:
    >> idx_outs = find_outliers_df(df['AdjustedCompensation'])
    >> good_data = df[~idx_outs].copy()

    function snippet from Flatiron School Phase #2 Py Files.
    URL = https://github.com/flatiron-school/Online-DS-FT-022221-Cohort-Notes/blob/mast

    """
    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

    idx_outs = (df_b>upper_limit) | (df_b<lower_limit)

    return idx_outs
```

In [30]:
```
#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
```

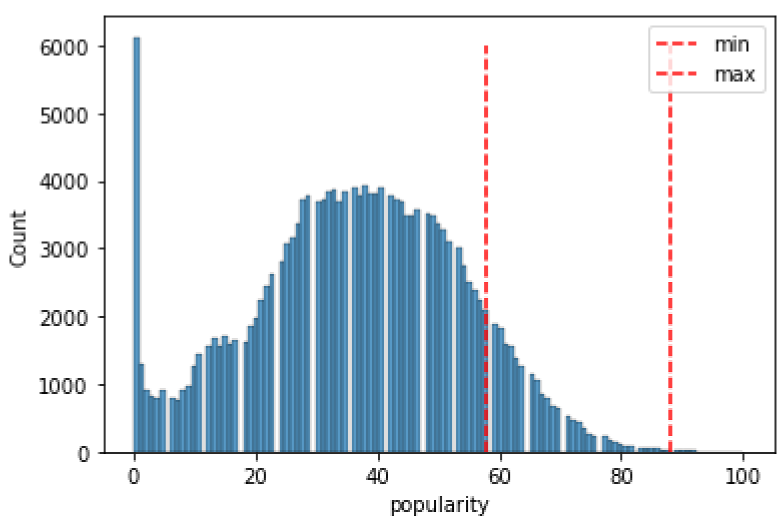
In [31]:
```
fig, ax = plt.subplots()
sns.histplot(df_100['popularity '], bins=15, ax=ax)
```

Out[31]: <AxesSubplot:xlabel='popularity ', ylabel='Count'>



In [32]:
```
#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
ax.vlines(x=df_100['popularity '].max(), ymin=0, ymax=6000, linestyles='dashed', colors
plt.legend()
```

Out[32]: <matplotlib.legend.Legend at 0x2e820827310>



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 [33]:
```
#creating is_popular column with our cutoff point
df['is_popular']=(df['popularity']>=58).astype('int')
df.head()
```

Out[33]:

track_id	artist_name	track_name	popularity	acousticness	danceability	duration_ms
00021Wy6AyMbLP2tqij86e	Capcom Sound Team	Zangief's Theme	13	0.234	0.617	169173
000CzNKC8PEt1yC3L8dqwV	Henri Salvador	Coeur Brisé à Prendre - Remastered	5	0.249	0.518	130653
000DfZJww8KiixTKuk9usJ	Mike Love	Earthlings	30	0.366	0.631	357573
000EWWBkYaREzsBplYjUag	Don Philippe	Fewerdolr	39	0.815	0.768	104924
000xQL6tZNLJzIrtIgxqSl	ZAYN	Still Got Time	70	0.131	0.748	188491

5 rows × 43 columns

In [34]:
```
#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[34]:

track_id	acousticness	danceability	duration_ms	energy	instrumentalness	key	live
00021Wy6AyMbLP2tqij86e	0.234	0.617	169173	0.862	0.976000	G	0.
000CzNKC8PEt1yC3L8dqwV	0.249	0.518	130653	0.805	0.000000	F	0.
000DfZJww8KiixTKuk9usJ	0.366	0.631	357573	0.513	0.000004	D	0.
000EWWBkYaREzsBplYjUag	0.815	0.768	104924	0.137	0.922000	C#	0.
000xQL6tZNLJzIrtIgxqSl	0.131	0.748	188491	0.627	0.000000	G	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.

One Hot Encoding the Categorical Columns

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

In [35]:
```
#Check to see how many more columns we will be creating by OHE the cat_cols.
df.nunique()
```

Out[35]:
```
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
```

We will be creating 2 (mode) + 5 (time_signature) + key (12) - 3 (drop_first) = 16 columns.

In [36]:
```
#define categorical columns
cat_cols = ['key', 'mode', 'time_signature']
```

In [37]:
```
#One hot encoding the dataframe
from sklearn.preprocessing import OneHotEncoder

encoder=OneHotEncoder(sparse=False, drop='first')
data_ohe = encoder.fit_transform(df[cat_cols])
df_ohe = pd.DataFrame(data_ohe, columns=encoder.get_feature_names(cat_cols), index=df.i
```

In [38]:
```
pd.set_option("display.max_columns", None)
df_ohe
```

Out[38]:

	key_A#	key_B	key_C	key_C#	key_D	key_D#	key_E	key_F	key_F#	key

track_id	key_A#	key_B	key_C	key_C#	key_D	key_D#	key_E	key_F	key_F#	key_
track_id										
00021Wy6AyMbLP2tqij86e	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1
000CzNKC8PEt1yC3L8dqwV	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	(
000DfZJww8KiixTKuk9usJ	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	(
000EWWBkYaREzsBplYjUag	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	(
000xQL6tZNLJzIrtIgxqSl	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1
...	...	...	...	...	...	...	...	...	...	
7zz7MbCb9G7KJc1NVl9bL0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	(
7zzFNNxVD0h0ctAT08H0pa	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	(
7zzTeltz93lYI52hlcipm5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1
7zzZmpw8L66ZPjH1M6qmOs	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	(
7zzbfi8fvHe6hm342GcNYl	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	(

176774 rows × 16 columns

In [39]:
```
df_ohe = pd.concat([df.drop(cat_cols, axis=1), df_ohe], axis=1)
df_ohe.head()
```

Out[39]:

	acousticness	danceability	duration_ms	energy	instrumentalness	liveness
track_id						
00021Wy6AyMbLP2tqij86e	0.234	0.617	169173	0.862	0.976000	0.1410
000CzNKC8PEt1yC3L8dqwV	0.249	0.518	130653	0.805	0.000000	0.3330
000DfZJww8KiixTKuk9usJ	0.366	0.631	357573	0.513	0.000004	0.1090
000EWWBkYaREzsBplYjUag	0.815	0.768	104924	0.137	0.922000	0.1130
000xQL6tZNLJzIrtIgxqSl	0.131	0.748	188491	0.627	0.000000	0.0852

With the dataframe scrubbed and one hot encoded we can move onto the modelling process.

MODEL

train_test_split

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

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

The first model we will be generating is a dummy classifier. We will be comparing our models' success to each other but also to this baseline model.

Model #1 - Baseline - Dummy Classifier

In [41]:
```
from sklearn.dummy import DummyClassifier

clf_dummy = DummyClassifier()
clf_dummy.fit(X_train, y_train)
y_pred = clf_dummy.predict(X_test)
```

```
C:\Users\berke\anaconda3\envs\learn-env\lib\site-packages\sklearn\dummy.py:131: FutureWa
rning: The default value of strategy will change from stratified to prior in 0.24.
  warnings.warn("The default value of strategy will change from "
```

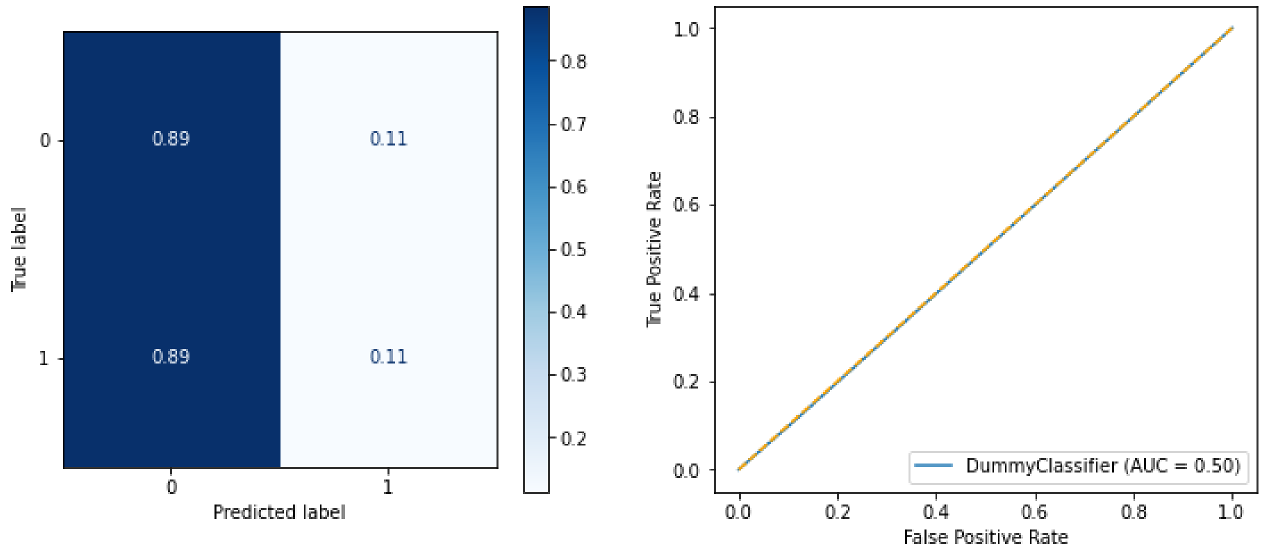
We need a function that will show us the classification report, the confusion matrix as well as the ROC curve to be able to evaluate our models.

In [42]:
```
from sklearn.metrics import classification_report, plot_confusion_matrix, plot_roc_curv

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 per
    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
    plot_confusion_matrix(estimator=clf, X=X, y_true=y_true, cmap='Blues', normalize='t

    #Plotting the ROC curve
    plot_roc_curve(estimator=clf, X=X, y=y_true, ax=ax[1])

    #Plotting the 50-50 guessing plot for reference
    ax[1].plot([0,1], [0,1], ls='--', color='orange')
```

In [43]:
```
classification(y_test, y_pred, X_test, clf_dummy)
```

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

           0       0.89      0.89      0.89     47002
           1       0.11      0.11      0.11      6031

    accuracy                           0.80     53033
   macro avg       0.50      0.50      0.50     53033
```

```
weighted avg       0.80      0.80      0.80     53033
```



In [44]:
```python
#class imbalance percentages
y_train.value_counts(normalize=True)
```

Out[44]:
```
0    0.885503
1    0.114497
Name: is_popular, dtype: float64
```

Our dummy classifier correctly predicted 98% of the unpopular songs as unpopular; however, it correctly predicted only 12% of the popular songs as popular and instead classified 88% of them as unpopular as well. We clearly have a class imbalance problem where approximately 98% of our data is not popular and only about 2% of it is. To address this we can SMOTE the training data and see if training a model with this method would improve our results.

Addressing Class Imbalance with SMOTENC

In [45]:
```python
#looking at column names to extract categorical column indices for SMOTENC
X.columns
```

Out[45]:
```
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 [46]:
```python
#creating a list of categorical column indices
cat_cols = list(range(10, len(X.columns)))
X.columns[cat_cols]
```

Out[46]:
```
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 [47]:
```
from imblearn.over_sampling import SMOTE, SMOTENC

sm = SMOTENC(categorical_features=cat_cols)

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

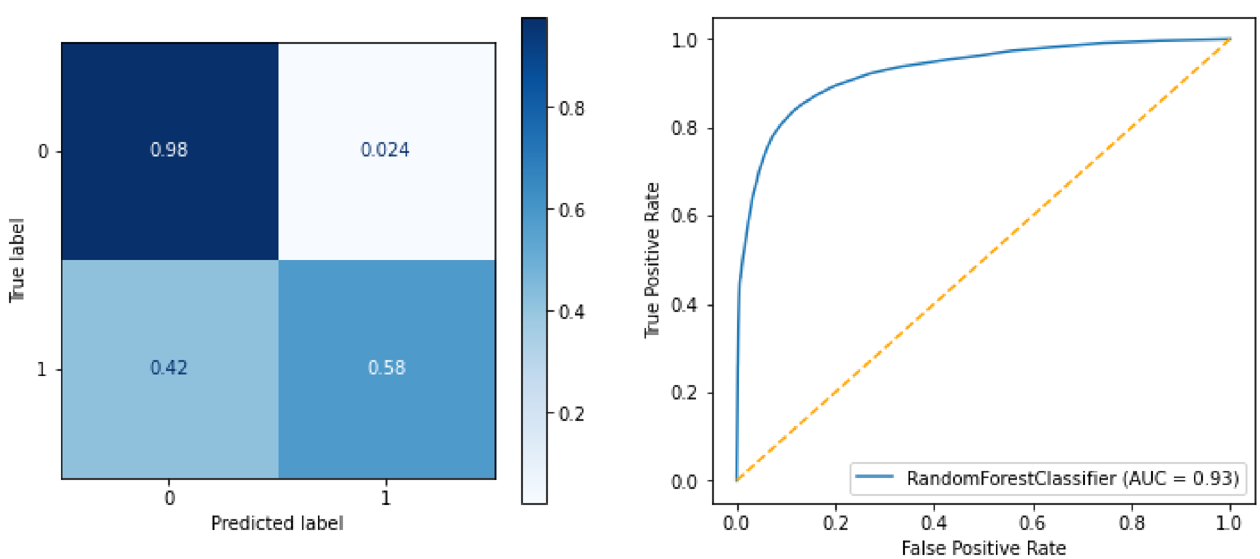
Out[47]:
```
1    0.5
0    0.5
Name: is_popular, dtype: float64
```

Model #2 - Random Forest Classifier

Initial Model

In [48]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

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

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

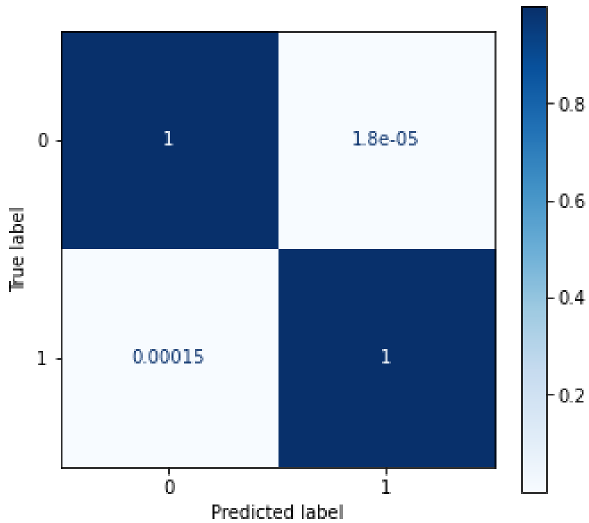
```
CLASSIFICATION REPORT
------------------------------------------
              precision    recall  f1-score   support

           0       0.95      0.98      0.96     47002
           1       0.76      0.58      0.66      6031

    accuracy                           0.93     53033
   macro avg       0.85      0.78      0.81     53033
weighted avg       0.93      0.93      0.93     53033
```

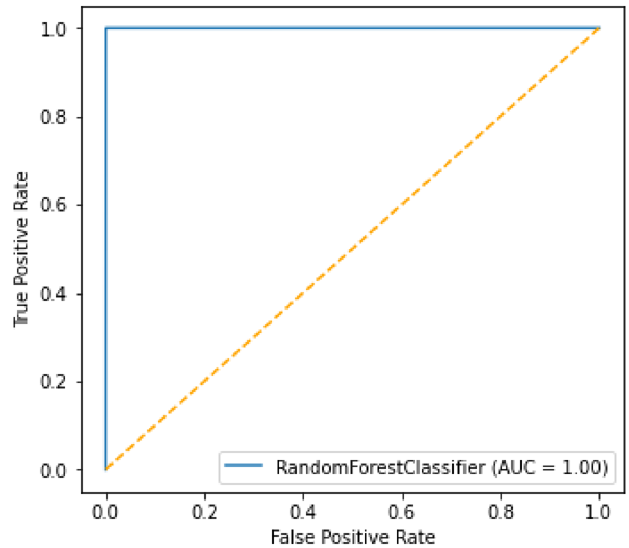


In [49]:
```
#Evaluating the model performance for the training data
y_pred = clf_rf.predict(X_train_sm)
```

```
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. Compared to the baseline dummy classifier model we are performing 47% better in predicting popular songs.

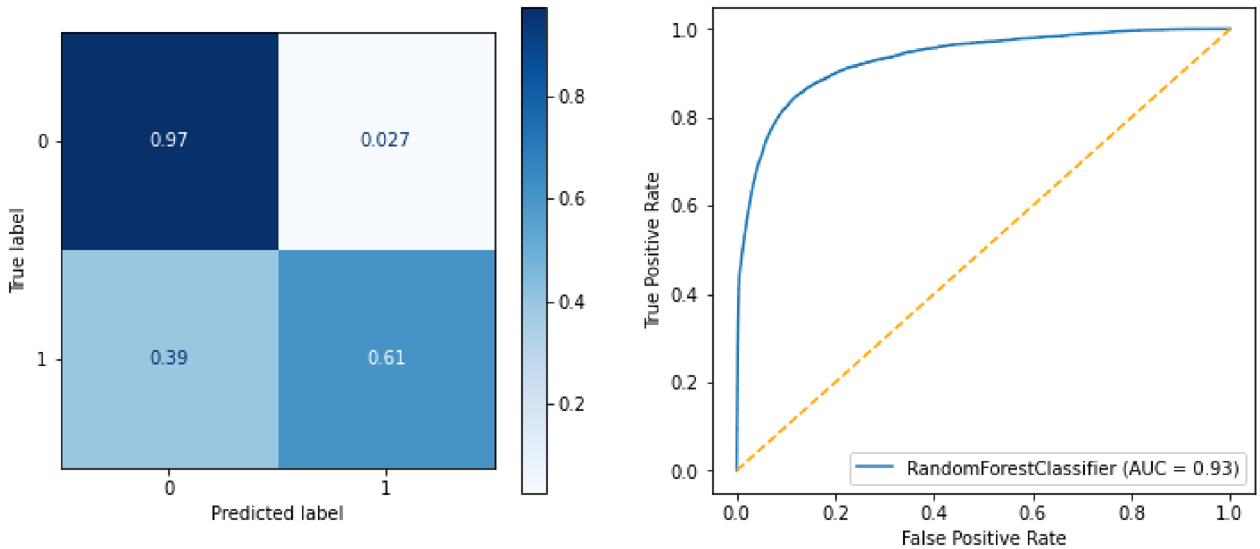
Hyperparameter Tuning

In [50]:
```
# 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 [51]:
```
clf_rf_tuned = RandomForestClassifier(criterion='entropy', max_depth=None, min_samples_
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       0.74      0.61      0.67      6031

    accuracy                           0.93     53033
   macro avg       0.85      0.79      0.82     53033
weighted avg       0.93      0.93      0.93     53033
```



Tuning the hyperparameters of our model unfortunately did not improve the recall score for this training set of data (refer to Future Considerations section for more information). We can move onto cross validating this score to see what happens to the recall score with 5 other splits of the training and testing data and then proceed with trying additional types of models to see if the recall score improves.

Cross-validation Scores

In [52]:
```
from sklearn.model_selection import cross_val_score
#5-fold cross validation
cross_val_scores = cross_val_score(clf_rf_tuned, X_train_sm, y_train_sm, scoring='recal
```

In [53]:
```
import numpy as np
print(np.round(cross_val_scores, 2))
print(f"Mean cross-validation score: {np.round(cross_val_scores.mean(),2)}")
```

```
[0.64 0.99 0.98 0.98 0.98]
Mean cross-validation score: 0.92
```

The cross-validated scores of the model are still lower than we would like them to be so we will proceed with trying a XGBoost model next.

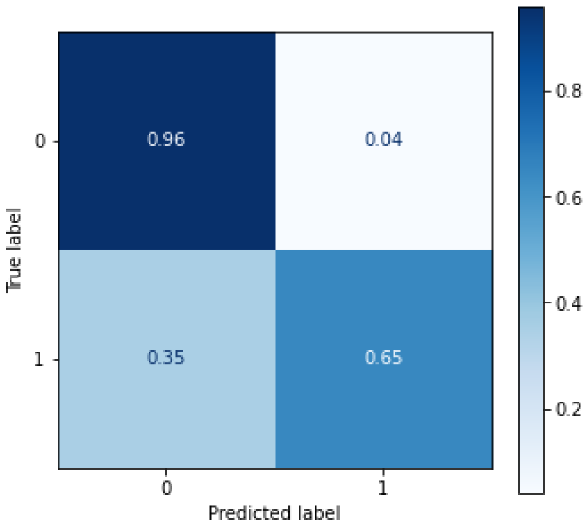
Model #3 - XGBoost

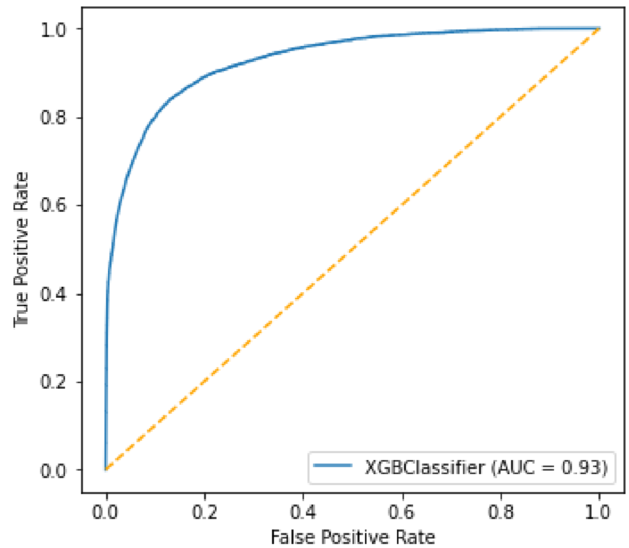
Initial Model

In [54]:
```
from xgboost import XGBClassifier
```

In [55]:
```
clf_xgb = XGBClassifier()
clf_xgb.fit(X_train_sm, y_train_sm)
y_pred = clf_xgb.predict(X_test)
classification(y_test, y_pred, X_test, clf_xgb)
```

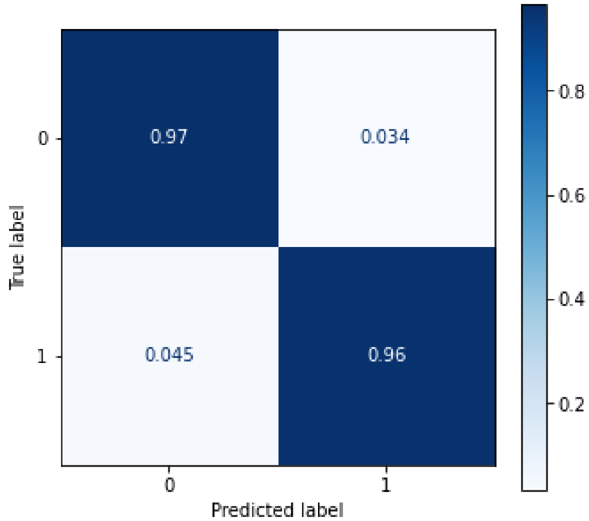
```
CLASSIFICATION REPORT
------------------------------------------
              precision    recall  f1-score   support

           0       0.96      0.96      0.96     47002
           1       0.67      0.65      0.66      6031

    accuracy                           0.92     53033
   macro avg       0.81      0.80      0.81     53033
weighted avg       0.92      0.92      0.92     53033
```

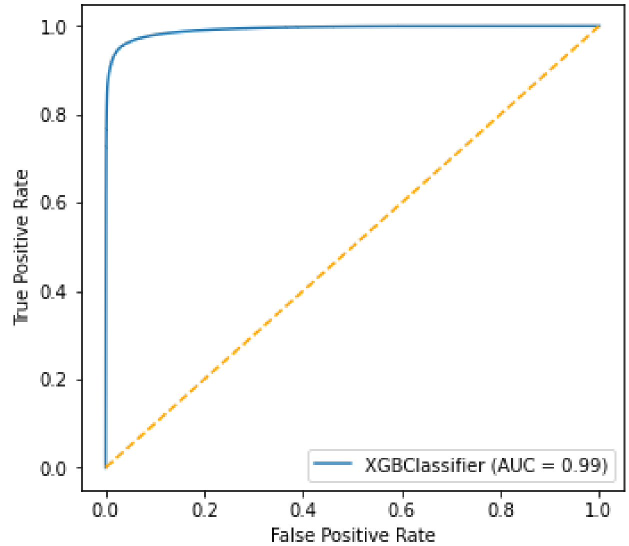




In [56]:
```
#Evaluating the model performance for the training data
y_pred = clf_xgb.predict(X_train_sm)
classification(y_train_sm, y_pred, X_train_sm, clf_xgb)
```

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

           0       0.96      0.97      0.96    109573
           1       0.97      0.96      0.96    109573

    accuracy                           0.96    219146
   macro avg       0.96      0.96      0.96    219146
weighted avg       0.96      0.96      0.96    219146
```





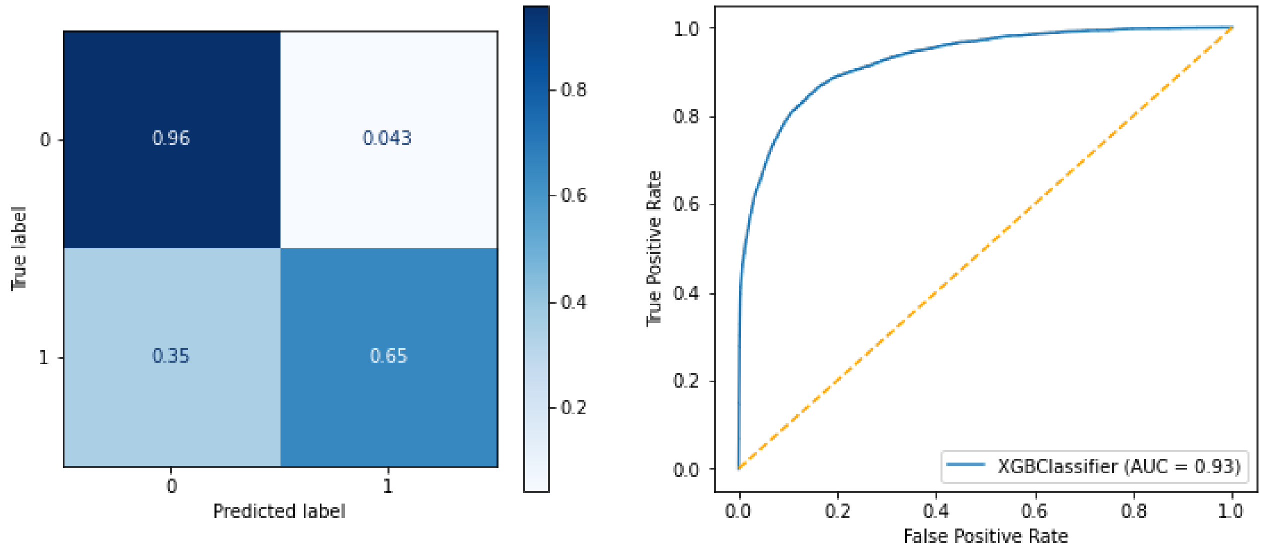
Once again, our model is overfitting the training data. We can run another gridsearch and tune our model to see if the recall score can be improved.

Hyperparameter Tuning

In [57]:
```
# grid = {
#     'learning_rate': [0.01, 0.1, 0.2],
#     'max_depth': [10, 20, None]
#        }
# gridsearch = GridSearchCV(estimator=clf_xgb, param_grid = grid, scoring='recall', n_j

# gridsearch.fit(X_train_sm,  y_train_sm)
# gridsearch.best_params_
# # Results: {'learning_rate': 0.1, 'max_depth': 10}
```

In [58]:
```
clf_xgb_tuned = XGBClassifier(learning_rate=0.1, max_depth=10)
clf_xgb_tuned.fit(X_train_sm, y_train_sm)
y_pred = clf_xgb_tuned.predict(X_test)
classification(y_test, y_pred, X_test, clf_xgb_tuned)
```

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

           0       0.96      0.96      0.96     47002
           1       0.66      0.65      0.65      6031

    accuracy                           0.92     53033
   macro avg       0.81      0.80      0.81     53033
weighted avg       0.92      0.92      0.92     53033
```



Tuning our model has led to an increase of performance in our recall score by 1%, so we are performing 53% better compared to our baseline Dummy Classifier model and 6% better than our tuned Random Forest model.

Cross-validation Scores

In [59]:
```
xgb_cross_val_scores = cross_val_score(clf_xgb_tuned, X_train_sm, y_train_sm, scoring='
```

In [60]:
```
import numpy as np
print(np.round(xgb_cross_val_scores, 2))
print(f"Mean cross-validation score: {np.round(xgb_cross_val_scores.mean(), 2)}")
```

```
[0.61 1.   0.99 0.99 0.99]
Mean cross-validation score: 0.92
```

Next we will be building and evaluating a Logistic Regression model based on the recall score.

Model #4 - LogisticRegressionCV

Since the Logistic Regression models are potentially sensitive to outliers and need scaled data we will need to process our data one more time to remove outliers and scale it.

Removing Outliers

In [61]:
```
#separating out the numerical columns for outlier removal
num_cols = list(X.columns[0:10])
num_cols
```

Out[61]:
```
['acousticness',
 'danceability',
 'duration_ms',
 'energy',
 'instrumentalness',
 'liveness',
 'loudness',
 'speechiness',
 'tempo',
 'valence']
```

In [62]:
```
df_ohe_clean = df_ohe.copy()
```

In [63]:
```
for col in num_cols:
    df_ohe_clean = df_ohe_clean[find_outliers_IQR(df_ohe_clean[col])==False]

df_ohe_clean
```

Out[63]:

	acousticness	danceability	duration_ms	energy	instrumentalness	liveness
track_id						
000CzNKC8PEt1yC3L8dqwV	0.2490	0.518	130653	0.805	0.000000	0.3330
000DfZJww8KiixTKuk9usJ	0.3660	0.631	357573	0.513	0.000004	0.1090
000xQL6tZNLJzIrtIgxqSl	0.1310	0.748	188491	0.627	0.000000	0.0852
001CyR8xqmmpVZFiTZJ5BC	0.3070	0.826	160107	0.679	0.000025	0.1510
001KkOBeRiQ1J7IEJYHODW	0.0697	0.279	300053	0.455	0.000091	0.0974
...	...	...	...	...	...	...
7zywdG4ysfC5XNBzjQAo2o	0.1230	0.443	202760	0.885	0.000031	0.2800
7zz3cHALU9cj7Io5qINt1R	0.8330	0.353	273800	0.383	0.000131	0.1100
7zzTeltz93lYI52hlcipm5	0.1130	0.716	228493	0.806	0.000000	0.1510
7zzZmpw8L66ZPjH1M6qmOs	0.2170	0.664	267960	0.537	0.000003	0.1180
7zzbfi8fvHe6hm342GcNYl	0.0299	0.533	342827	0.547	0.011300	0.0723

97339 rows × 53 columns

train_test_split

In [64]:
```
y=df_ohe_clean['is_popular']
X=df_ohe_clean.drop('is_popular', axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.30, random_state=4
```

Addressing Class Imbalance with SMOTENC

In [65]:
```
y_train.value_counts(normalize=True)
```

Out[65]:
```
0    0.842318
1    0.157682
Name: is_popular, dtype: float64
```

In [66]:
```
X_train.columns
```

Out[66]:
```
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 [67]:
```
cat_cols = list(range(10,len(X_train.columns)))
```

In [68]:
```
from imblearn.over_sampling import SMOTE, SMOTENC
sm = SMOTENC(categorical_features=cat_cols)

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

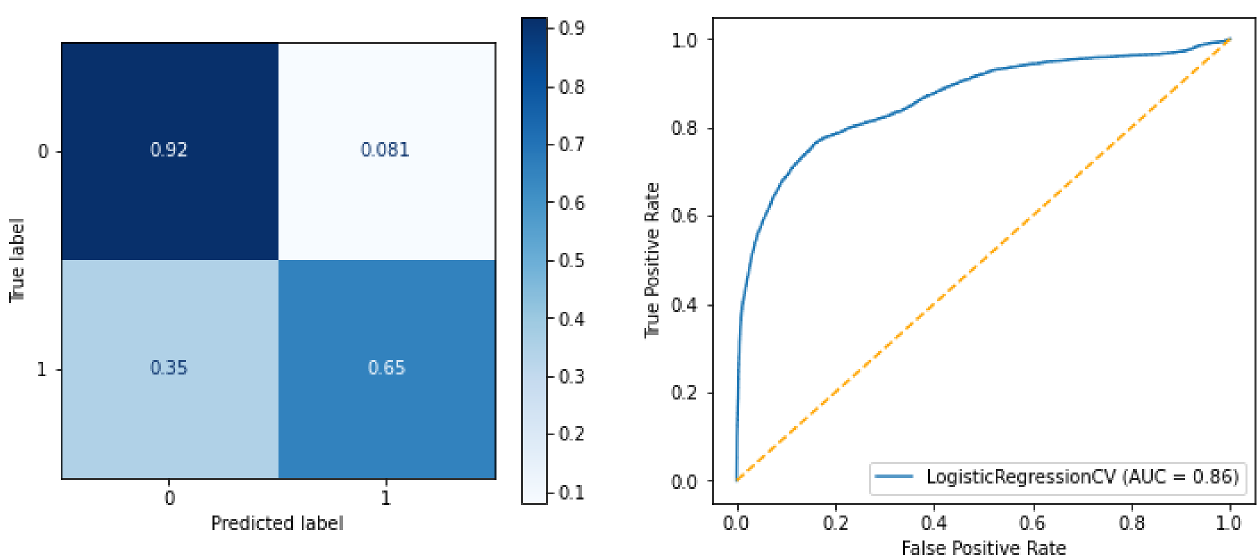
Out[68]:
```
1    0.5
0    0.5
Name: is_popular, dtype: float64
```

Scaling the Data

In [69]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

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

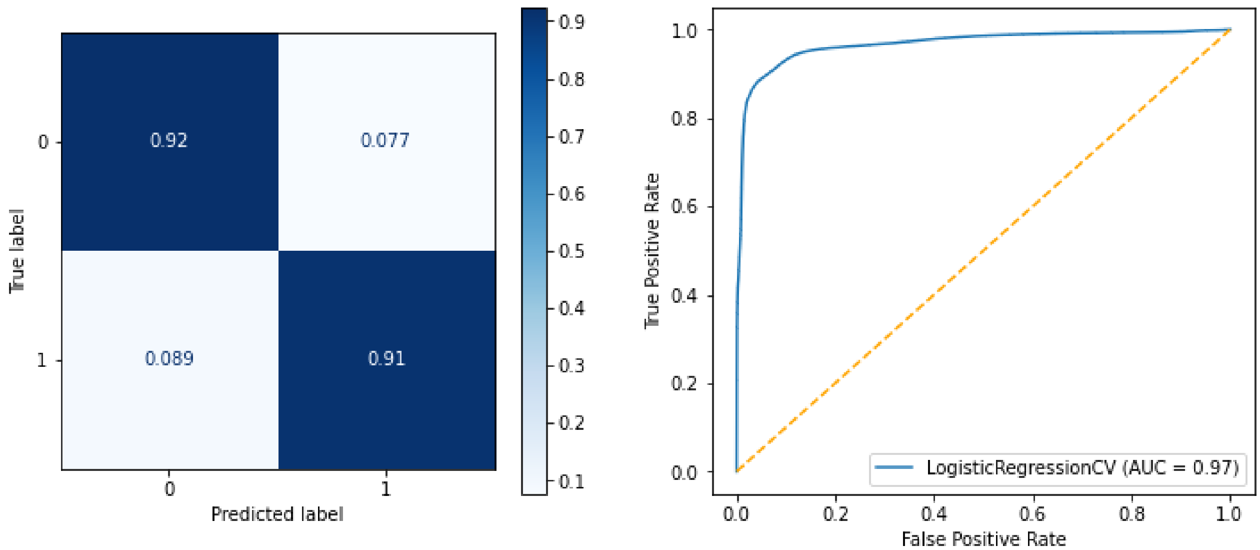
In [70]:
```
from sklearn.linear_model import LogisticRegressionCV
clf_logregcv = LogisticRegressionCV(cv=5)
clf_logregcv.fit(X_train_sm_sc,  y_train_sm)
y_pred = clf_logregcv.predict(X_test_sc)
classification(y_test, y_pred, X_test_sc, clf_logregcv)
```

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

           0       0.93      0.92      0.93     24561
           1       0.60      0.65      0.63      4641

    accuracy                           0.88     29202
   macro avg       0.77      0.79      0.78     29202
weighted avg       0.88      0.88      0.88     29202
```



In [71]:
```
#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     57393
           1       0.92      0.91      0.92     57393

    accuracy                           0.92    114786
   macro avg       0.92      0.92      0.92    114786
weighted avg       0.92      0.92      0.92    114786
```



Our model is once again overfitting to the training data and performing very well on it but the model's performance drops significantly when we test it with the test data. In order to address this, we can once again perform a grid search and try to tune the model.

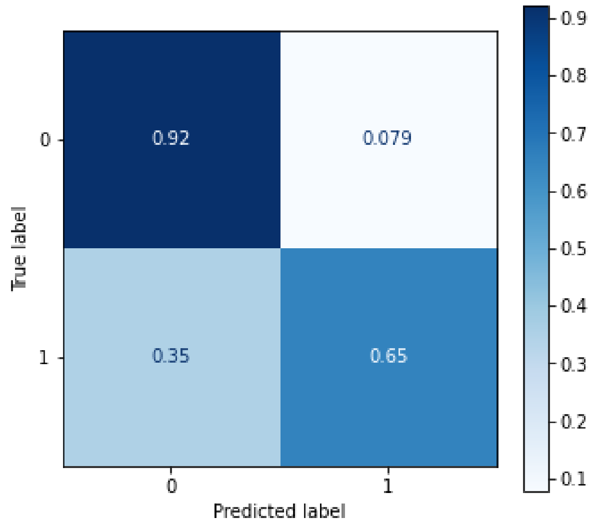
Hyperparameter Tuning

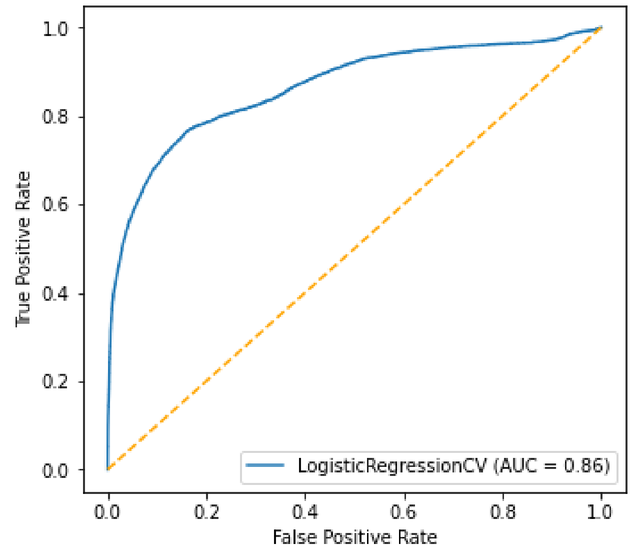
In [72]:
```
# clf = LogisticRegressionCV(cv=5)
# grid = {'penalty': ['l1','l2'],
#          'solver': ['liblinear', 'lbfgs', 'sag', 'saga'],
#          'class_weight': ['balanced', None],
#          'Cs': [1e12, 10, 1, 0.1]
#    }

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

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

In [73]:
```
clf_logregcv_tuned = LogisticRegressionCV(cv=5, class_weight='balanced', Cs=1, penalty=
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     24561
              1       0.61      0.65      0.63      4641

       accuracy                           0.88     29202
      macro avg       0.77      0.79      0.78     29202
   weighted avg       0.88      0.88      0.88     29202
```





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 data points to more accurately predict the popularity of a song.

iNTERPRET

Parsing Feature Importances to Dataframes

Random Forest

In [74]:
```
rf_importances_df = pd.Series(clf_rf_tuned.feature_importances_, index=X.columns).sort_
#parsing the series to a dataframe
rf_importances_df = rf_importances_df.reset_index()
rf_importances_df.columns = ['Attribute', 'Importance']
rf_importances_df
```

Out[74]:

	Attribute	Importance
0	Pop	0.164525
1	acousticness	0.057078
2	loudness	0.036838
3	instrumentalness	0.035286
4	energy	0.031909
5	danceability	0.031781

	Attribute	Importance
6	speechiness	0.026970
7	Reggae	0.026713
8	valence	0.025133
9	Ska	0.024537
10	Electronic	0.023677
11	Reggaeton	0.023457
12	Rock	0.023326
13	duration_ms	0.023224
14	Anime	0.022284
15	Blues	0.021852
16	key_C	0.020863
17	liveness	0.019780
18	key_D	0.019494
19	key_G	0.019102
20	Country	0.018105
21	World	0.017470
22	time_signature_4/4	0.016976
23	tempo	0.016694
24	key_F	0.015880
25	Jazz	0.015826
26	key_C#	0.015623
27	key_B	0.015528
28	Soul	0.015292
29	key_E	0.014387
30	Movie	0.013773
31	key_G#	0.013426
32	key_A#	0.012829
33	Rap	0.012073
34	Folk	0.012057
35	key_F#	0.011485
36	Comedy	0.010366
37	time_signature_3/4	0.010352
38	Children's Music	0.009645

	Attribute	Importance
39	R&B	0.009163
40	Indie	0.006153
41	key_D#	0.006080
42	Hip-Hop	0.006077
43	Alternative	0.005603
44	Soundtrack	0.004633
45	Dance	0.004584
46	Classical	0.004508
47	Opera	0.003727
48	mode_Minor	0.003216
49	time_signature_5/4	0.000432
50	time_signature_1/4	0.000153
51	A Capella	0.000056

XGBoost

In [75]:
```
#parsing feature importances to a series, sorting and displaying top 10
xgb_importances_df = pd.Series(clf_xgb_tuned.feature_importances_, index=X.columns).sor
#parsing the series to a dataframe
xgb_importances_df = xgb_importances_df.reset_index()
#renaming columns
xgb_importances_df.columns=['Attribute', 'Importance']
xgb_importances_df
```

Out[75]:

	Attribute	Importance
0	Pop	0.316489
1	Blues	0.045231
2	Anime	0.040975
3	Ska	0.040372
4	Electronic	0.040091
5	key_F	0.036262
6	Reggae	0.029529
7	World	0.026996
8	Reggaeton	0.026405
9	key_G	0.023537
10	Comedy	0.023419
11	key_E	0.023021
12	key_D	0.022285

	Attribute	Importance
13	key_C#	0.020703
14	time_signature_4/4	0.020695
15	key_B	0.019994
16	Movie	0.019381
17	Country	0.019373
18	key_C	0.018227
19	key_G#	0.017221
20	Jazz	0.017202
21	key_F#	0.016300
22	key_A#	0.016279
23	key_D#	0.011835
24	Soul	0.011235
25	Rock	0.010703
26	Opera	0.009135
27	Folk	0.008910
28	Rap	0.008456
29	Soundtrack	0.006531
30	Classical	0.006520
31	R&B	0.006228
32	Children's Music	0.006204
33	acousticness	0.005566
34	time_signature_1/4	0.003616
35	Hip-Hop	0.003579
36	A Capella	0.003334
37	Indie	0.003131
38	instrumentalness	0.002617
39	Alternative	0.002514
40	loudness	0.001816
41	Dance	0.001791
42	speechiness	0.000861
43	danceability	0.000762
44	duration_ms	0.000711
45	liveness	0.000679

	Attribute	Importance
46	energy	0.000659
47	valence	0.000648
48	time_signature_5/4	0.000621
49	mode_Minor	0.000596
50	time_signature_3/4	0.000412
51	tempo	0.000341

LogisticRegressionCV

In [76]:

```
logregcv_importances_df = pd.Series(clf_logregcv_tuned.coef_[0], index=X.columns).sort_
#parsing the series to a dataframe
logregcv_importances_df = logregcv_importances_df.reset_index()
logregcv_importances_df.columns = ['Attribute', 'Importance']
logregcv_importances_df
```

Out[76]:

	Attribute	Importance
0	Pop	0.601950
1	Rock	0.309054
2	danceability	0.116494
3	loudness	0.108664
4	Rap	0.107316
5	time_signature_4/4	0.084469
6	Dance	0.048944
7	duration_ms	0.026870
8	Hip-Hop	0.014091
9	speechiness	-0.007967
10	tempo	-0.009485
11	Indie	-0.018509
12	time_signature_1/4	-0.025143
13	energy	-0.027512
14	Alternative	-0.028040
15	acousticness	-0.028645
16	A Capella	-0.029475
17	mode_Minor	-0.030655
18	time_signature_5/4	-0.032491
19	liveness	-0.038735
20	instrumentalness	-0.041860

	Attribute	Importance
21	Soundtrack	-0.054250
22	Comedy	-0.073649
23	time_signature_3/4	-0.079753
24	valence	-0.091473
25	Classical	-0.103433
26	R&B	-0.107219
27	Opera	-0.137085
28	Children's Music	-0.143973
29	key_D#	-0.150332
30	Jazz	-0.169850
31	Folk	-0.174009
32	Soul	-0.178986
33	key_F#	-0.191404
34	Electronic	-0.201808
35	key_A#	-0.213272
36	key_G#	-0.222287
37	key_E	-0.228935
38	key_B	-0.232979
39	Movie	-0.241415
40	key_C#	-0.245953
41	World	-0.250408
42	Country	-0.253917
43	key_F	-0.254419
44	Blues	-0.259028
45	Reggaeton	-0.264379
46	key_D	-0.277526
47	key_C	-0.284692
48	key_G	-0.286190
49	Reggae	-0.289582
50	Anime	-0.296689
51	Ska	-0.315194

In [77]:
```
importances_df = pd.concat([xgb_importances_df, logregcv_importances_df, rf_importances
importances_df
```

Out[77]:

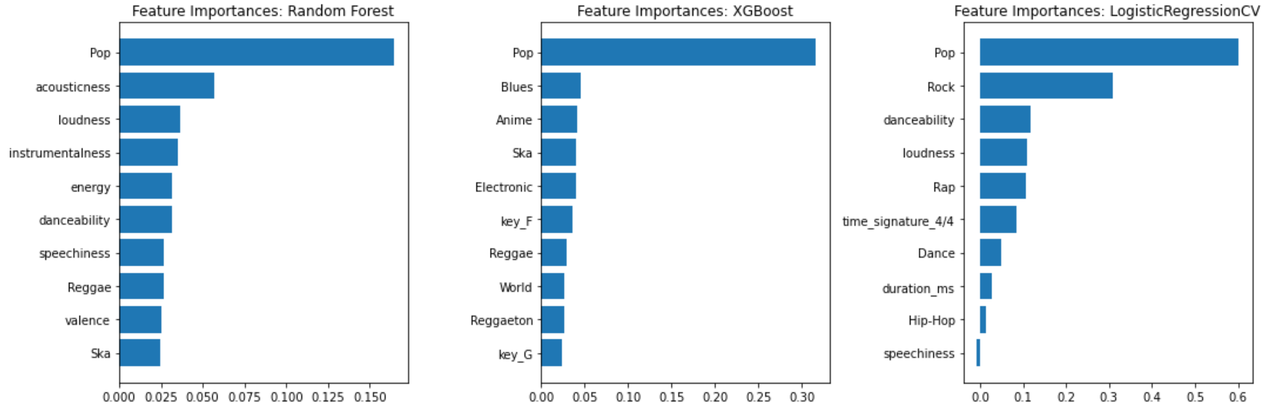
	Attribute	Importance	Attribute	Importance	Attribute	Importance
0	Pop	0.316489	Pop	0.601950	Pop	0.164525
1	Blues	0.045231	Rock	0.309054	acousticness	0.057078
2	Anime	0.040975	danceability	0.116494	loudness	0.036838
3	Ska	0.040372	loudness	0.108664	instrumentalness	0.035286
4	Electronic	0.040091	Rap	0.107316	energy	0.031909
5	key_F	0.036262	time_signature_4/4	0.084469	danceability	0.031781
6	Reggae	0.029529	Dance	0.048944	speechiness	0.026970
7	World	0.026996	duration_ms	0.026870	Reggae	0.026713
8	Reggaeton	0.026405	Hip-Hop	0.014091	valence	0.025133
9	key_G	0.023537	speechiness	-0.007967	Ska	0.024537
10	Comedy	0.023419	tempo	-0.009485	Electronic	0.023677
11	key_E	0.023021	Indie	-0.018509	Reggaeton	0.023457
12	key_D	0.022285	time_signature_1/4	-0.025143	Rock	0.023326
13	key_C#	0.020703	energy	-0.027512	duration_ms	0.023224
14	time_signature_4/4	0.020695	Alternative	-0.028040	Anime	0.022284
15	key_B	0.019994	acousticness	-0.028645	Blues	0.021852
16	Movie	0.019381	A Capella	-0.029475	key_C	0.020863
17	Country	0.019373	mode_Minor	-0.030655	liveness	0.019780
18	key_C	0.018227	time_signature_5/4	-0.032491	key_D	0.019494
19	key_G#	0.017221	liveness	-0.038735	key_G	0.019102
20	Jazz	0.017202	instrumentalness	-0.041860	Country	0.018105
21	key_F#	0.016300	Soundtrack	-0.054250	World	0.017470
22	key_A#	0.016279	Comedy	-0.073649	time_signature_4/4	0.016976
23	key_D#	0.011835	time_signature_3/4	-0.079753	tempo	0.016694
24	Soul	0.011235	valence	-0.091473	key_F	0.015880
25	Rock	0.010703	Classical	-0.103433	Jazz	0.015826
26	Opera	0.009135	R&B	-0.107219	key_C#	0.015623
27	Folk	0.008910	Opera	-0.137085	key_B	0.015528
28	Rap	0.008456	Children's Music	-0.143973	Soul	0.015292
29	Soundtrack	0.006531	key_D#	-0.150332	key_E	0.014387
30	Classical	0.006520	Jazz	-0.169850	Movie	0.013773
31	R&B	0.006228	Folk	-0.174009	key_G#	0.013426
32	Children's Music	0.006204	Soul	-0.178986	key_A#	0.012829

	Attribute	Importance	Attribute	Importance	Attribute	Importance
33	acousticness	0.005566	key_F#	-0.191404	Rap	0.012073
34	time_signature_1/4	0.003616	Electronic	-0.201808	Folk	0.012057
35	Hip-Hop	0.003579	key_A#	-0.213272	key_F#	0.011485
36	A Capella	0.003334	key_G#	-0.222287	Comedy	0.010366
37	Indie	0.003131	key_E	-0.228935	time_signature_3/4	0.010352
38	instrumentalness	0.002617	key_B	-0.232979	Children's Music	0.009645
39	Alternative	0.002514	Movie	-0.241415	R&B	0.009163
40	loudness	0.001816	key_C#	-0.245953	Indie	0.006153
41	Dance	0.001791	World	-0.250408	key_D#	0.006080
42	speechiness	0.000861	Country	-0.253917	Hip-Hop	0.006077
43	danceability	0.000762	key_F	-0.254419	Alternative	0.005603
44	duration_ms	0.000711	Blues	-0.259028	Soundtrack	0.004633
45	liveness	0.000679	Reggaeton	-0.264379	Dance	0.004584
46	energy	0.000659	key_D	-0.277526	Classical	0.004508
47	valence	0.000648	key_C	-0.284692	Opera	0.003727
48	time_signature_5/4	0.000621	key_G	-0.286190	mode_Minor	0.003216
49	mode_Minor	0.000596	Reggae	-0.289582	time_signature_5/4	0.000432
50	time_signature_3/4	0.000412	Anime	-0.296689	time_signature_1/4	0.000153
51	tempo	0.000341	Ska	-0.315194	A Capella	0.000056

Feature Importance Comparison

In [78]:

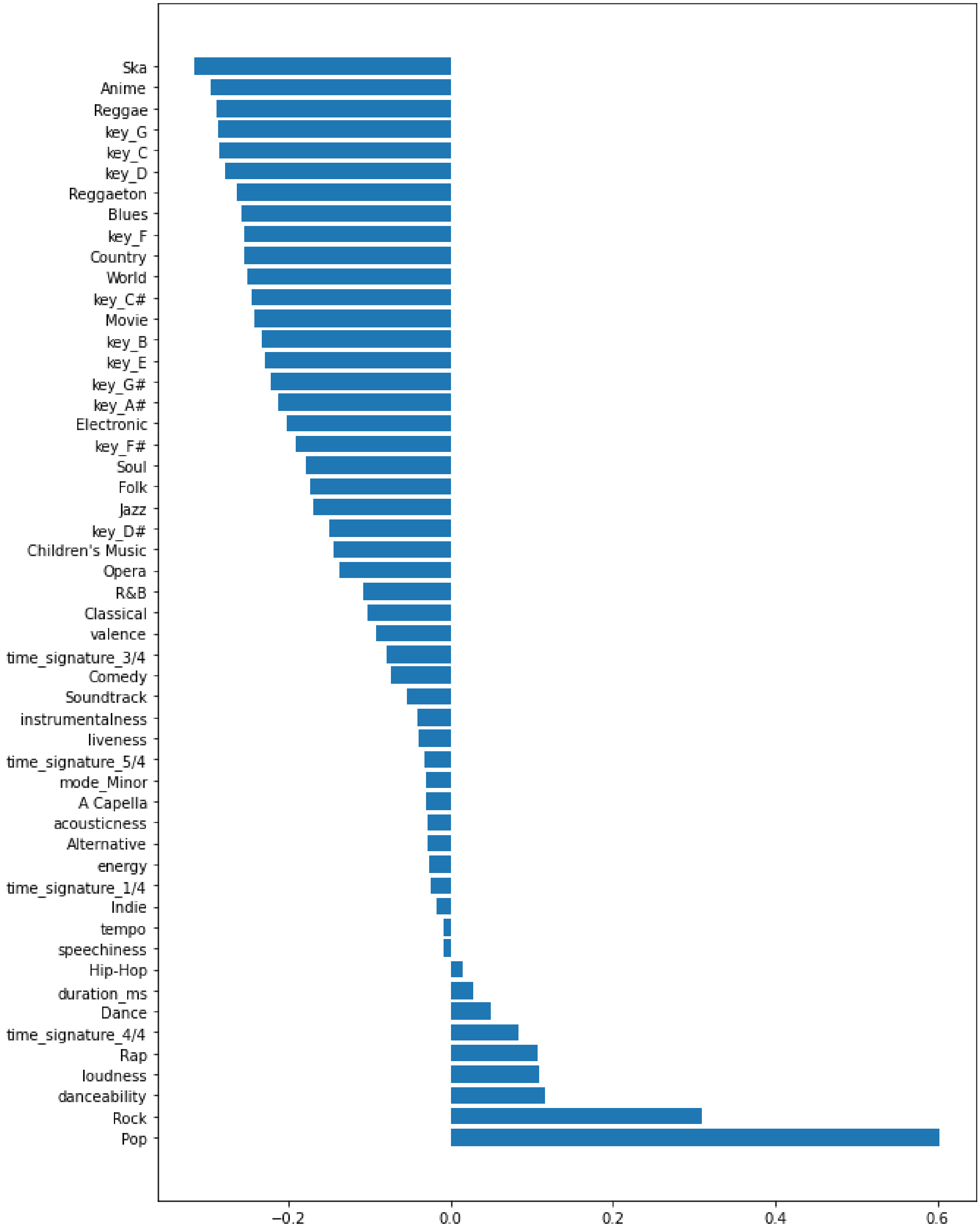
```
#plotting feature importances for all models for comparison

fig, ax = plt.subplots(ncols=3, figsize=(15,5))

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

xgb_importances_df = xgb_importances_df.sort_values(by='Importance', ascending=True).ta
ax[1].barh(xgb_importances_df['Attribute'], xgb_importances_df['Importance'])
ax[1].set_title('Feature Importances: XGBoost')

logregcv_importances_df = logregcv_importances_df.sort_values(by='Importance', ascendin
ax[2].barh(logregcv_importances_df['Attribute'], logregcv_importances_df['Importance'])
ax[2].set_title('Feature Importances: LogisticRegressionCV')
plt.tight_layout()
```



Among the 3 models we built we can see that Genre of a song has the highest effect on the popularity of a song. On all 3 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, it is difficult to reach conclusions as the importance values for the XGBoost and Random Forests don't have directionality to them.

In [79]:
```
logregcv_importances_df = pd.Series(clf_logregcv_tuned.coef_[0], index=X.columns).sort_
#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[79]:
```
<BarContainer object of 52 artists>
```



Data Visualizations

Genre

In [80]:
```
popular_songs_df = df_ohe[df_ohe['is_popular'] == 1]
unpopular_songs_df = df_ohe[df_ohe['is_popular']==0]
```

In [81]:
```
popular_genre_df = popular_songs_df.iloc[:, 10:36].agg('sum').sort_values(ascending=Fal
```

```
popular_genre_df.columns = ['genre', 'count']
popular_genre_df
```

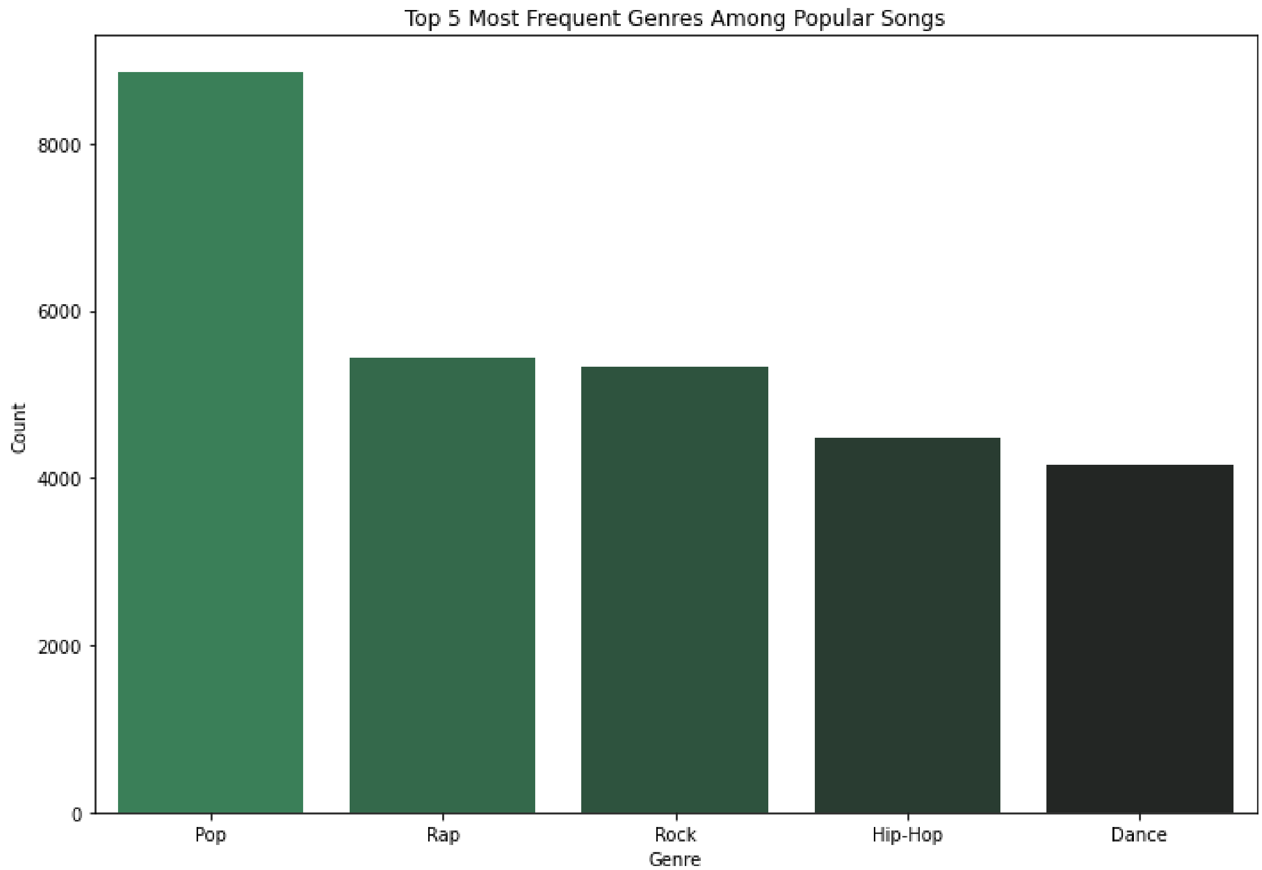
Out[81]:

	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 [82]:

```
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();
# plt.savefig('images/genre-popular.jpg')
```



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. These results make sense and are in-line with a survey conducted by IFPI (https://www.statista.com/chart/15763/most-popular-music-genres-worldwide/).

In [83]:
```
unpopular_genre_df = unpopular_songs_df.iloc[:, 10:36].agg('sum').sort_values(ascending
unpopular_genre_df.columns = ['genre', 'count']
unpopular_genre_df
```

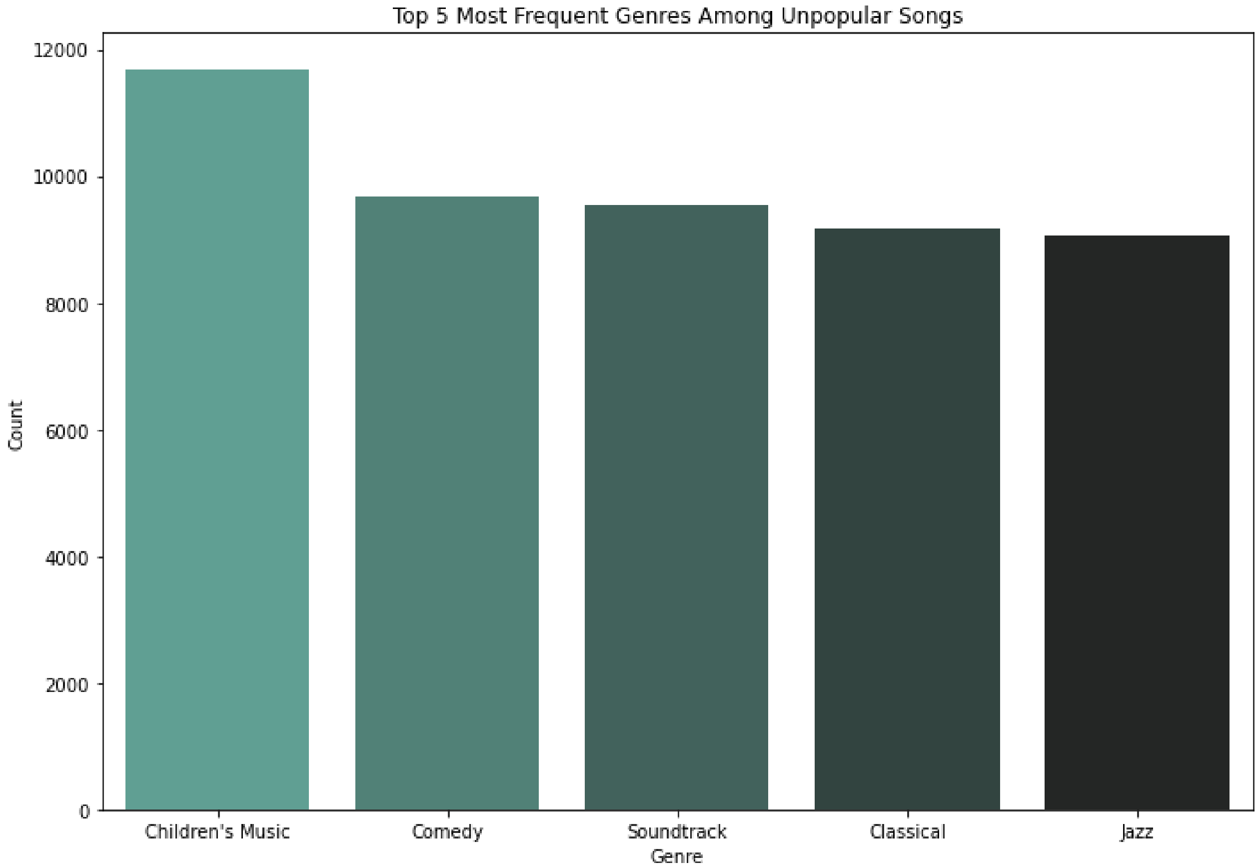
Out[83]:

	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

	genre	count
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 [84]:

```
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();
# plt.savefig('images/genre-unpopular.jpg')
# ax.set_xticklabels(ax.get_xticklabels(), rotation=45,ha='center');
```



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.

Energy

In [85]:

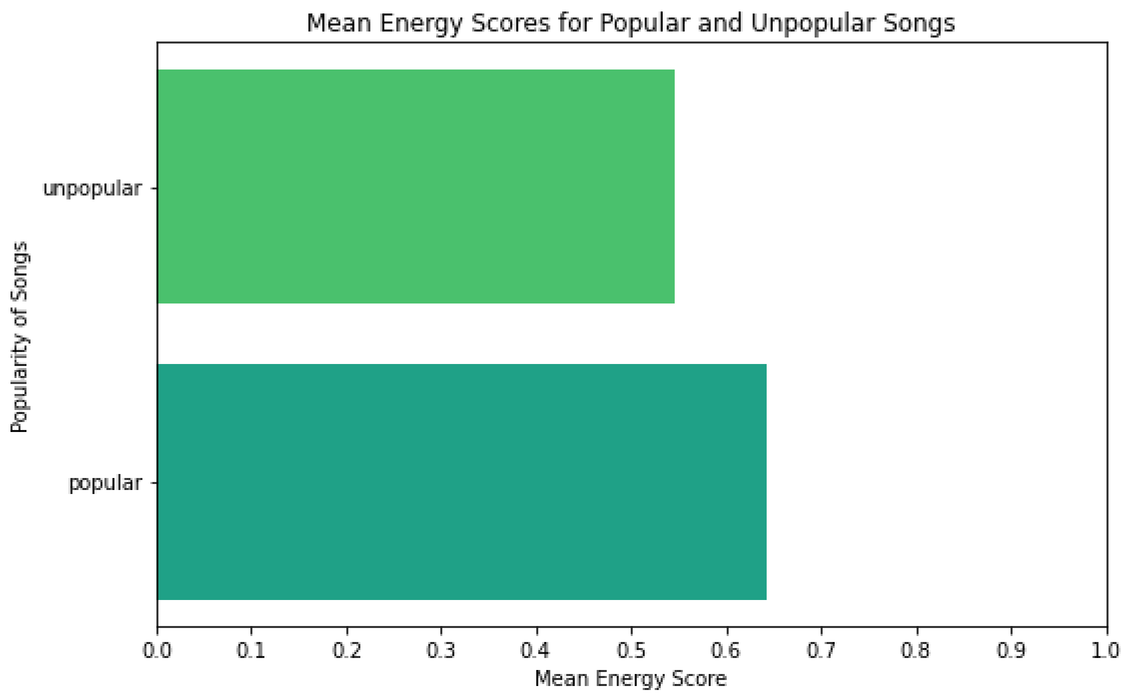
```
popular_energy_clean = popular_songs_df[find_outliers_IQR(popular_songs_df['energy'])==
print(popular_energy_clean['energy'].describe())

unpopular_energy_clean = unpopular_songs_df[find_outliers_IQR(unpopular_songs_df['energ
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 [86]:

```
mean_energy = {'popular': popular_energy_clean['energy'].mean(),
```

```
                    'unpopular': unpopular_energy_clean['energy'].mean()}

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()
plt.savefig('images/energy.jpg')
```



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.

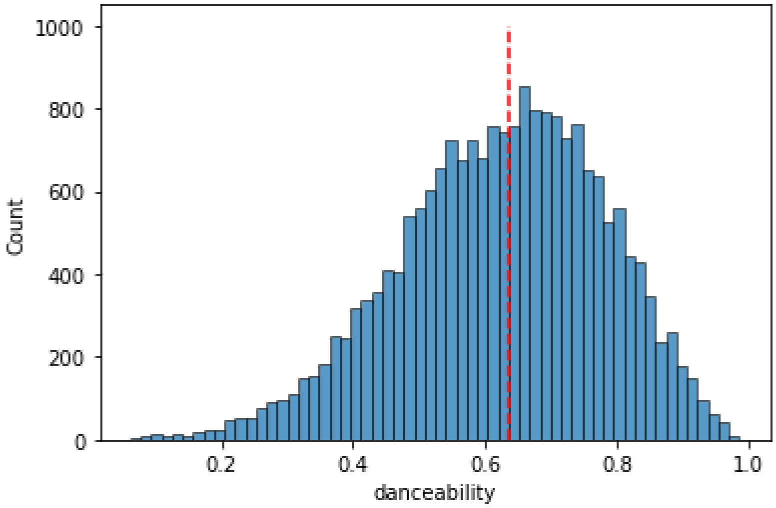
Danceability

In [87]:
```
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
```

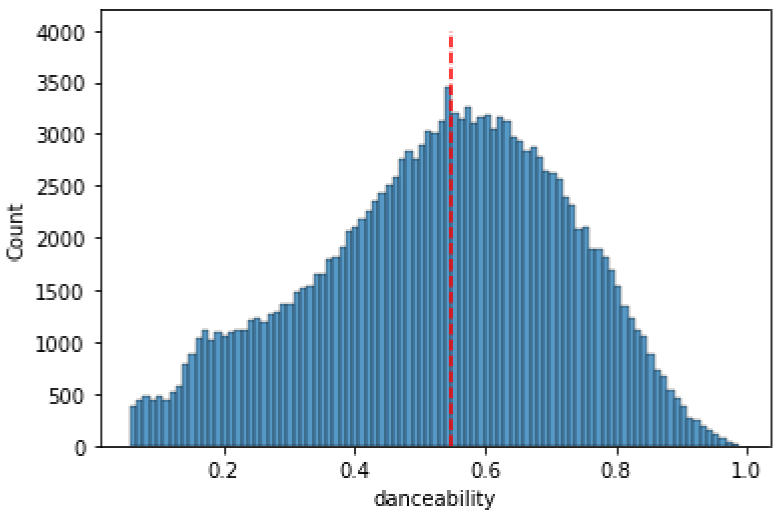
In [88]:
```
sns.histplot(data = popular_songs_df, x='danceability', bins='auto')
plt.vlines(x=popular_songs_df['danceability'].median(), ymin=0, ymax=1000, color='red',
```

Out[88]:
```
<matplotlib.collections.LineCollection at 0x2e825fb4550>
```



In [89]:
```
sns.histplot(data = unpopular_songs_df, x='danceability', bins='auto')
plt.vlines(x=unpopular_songs_df['danceability'].median(), ymin=0, ymax=4000, color='red
```

Out[89]: <matplotlib.collections.LineCollection at 0x2e83120d430>



In [90]:
```
popular_dance_clean = popular_songs_df[find_outliers_IQR(popular_songs_df['danceability
popular_dance_clean['danceability'].describe()
```

Out[90]:
```
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
```

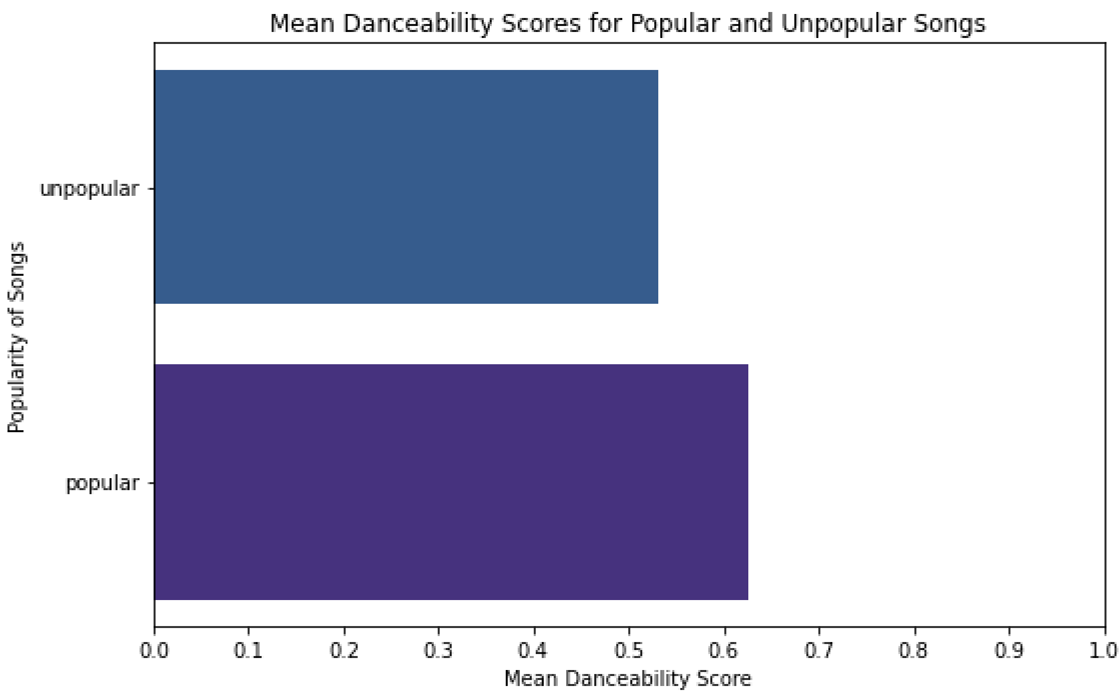
In [91]:
```
unpopular_dance_clean = unpopular_songs_df[find_outliers_IQR(unpopular_songs_df['dancea
unpopular_dance_clean['danceability'].describe()
```

Out[91]:
```
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 [92]:

```
mean_danceability = {'popular': popular_dance_clean['danceability'].mean(),
                     'unpopular': unpopular_dance_clean['danceability'].mean()}

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();
# plt.savefig('images/danceability.jpg')
```



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 [93]:

```
popular_acoustic_clean = popular_songs_df[find_outliers_IQR(popular_songs_df['acousticn
print(popular_acoustic_clean['acousticness'].describe())

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

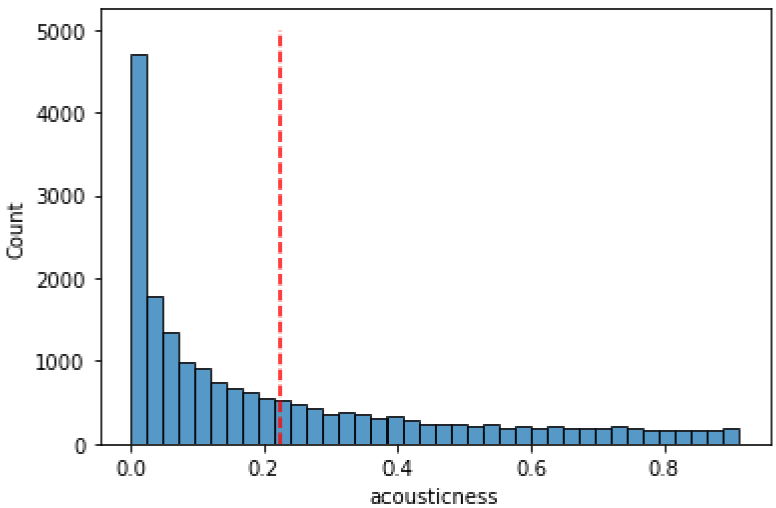
```
count    19715.000000
mean         0.226220
std          0.248585
min          0.000002
25%          0.026400
```

```
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 [94]:

```
sns.histplot(data = popular_acoustic_clean, x='acousticness', bins='auto')
plt.vlines(x=popular_acoustic_clean['acousticness'].mean(), ymin=0, ymax=5000, color='r
```

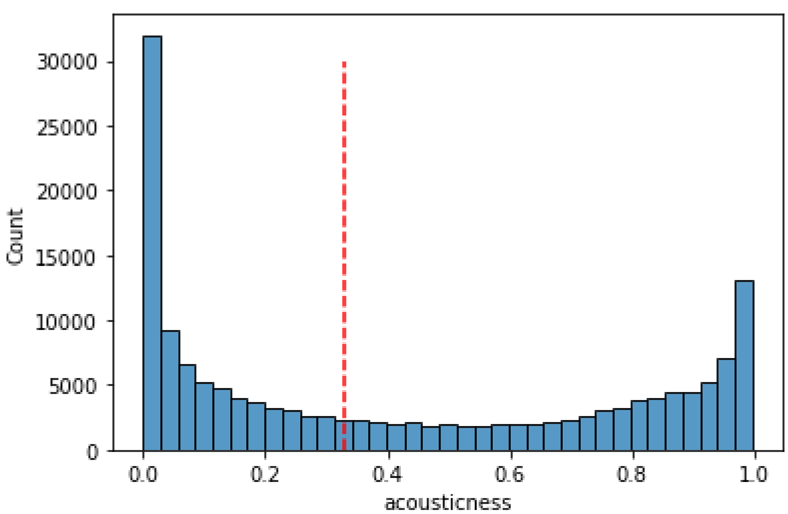
Out[94]: <matplotlib.collections.LineCollection at 0x2e82c161310>



In [95]:

```
sns.histplot(data = unpopular_songs_df, x='acousticness', bins='auto')
plt.vlines(x=unpopular_songs_df['acousticness'].median(), ymin=0, ymax=30000, color='re
```

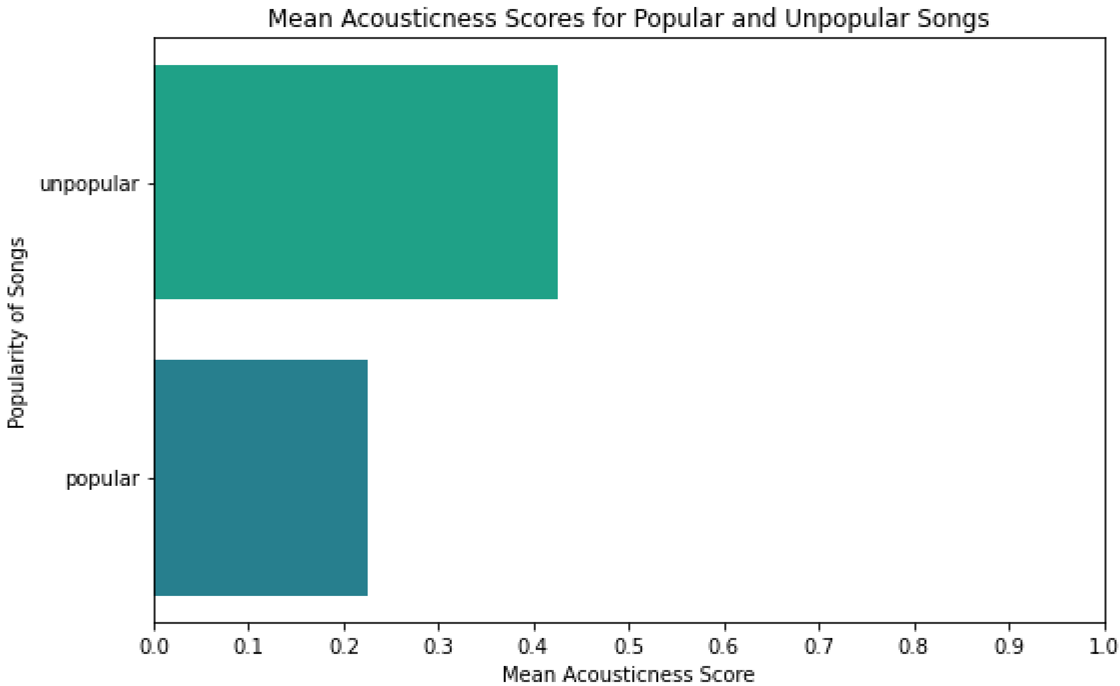
Out[95]: <matplotlib.collections.LineCollection at 0x2e8301f2640>



In [96]:

```
mean_acousticness = {'popular': popular_acoustic_clean['acousticness'].mean(),
                     'unpopular': unpopular_acoustic_clean['acousticness'].mean()}
```

```
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();
plt.savefig('images/acousticness.jpg')
```



Similar to the energy and danceability scores we see that the popular songs tended to have a lower acousticness score. Since acoustic songs are usually lower energy and rarely danceable this follows the same trend we've been observing.

CONCLUSIONS & RECOMMENDATIONS

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.