dsc-phase-3-project (/github/ebtezcan/dsc-phase-3-project/tree/master)
/ final_notebook.ipynb (/github/ebtezcan/dsc-phase-3-project/tree/master/final_notebook.ipynb)

Music Streaming Wars: Song Popularity Prediction

Author: E. Berke Tezcan

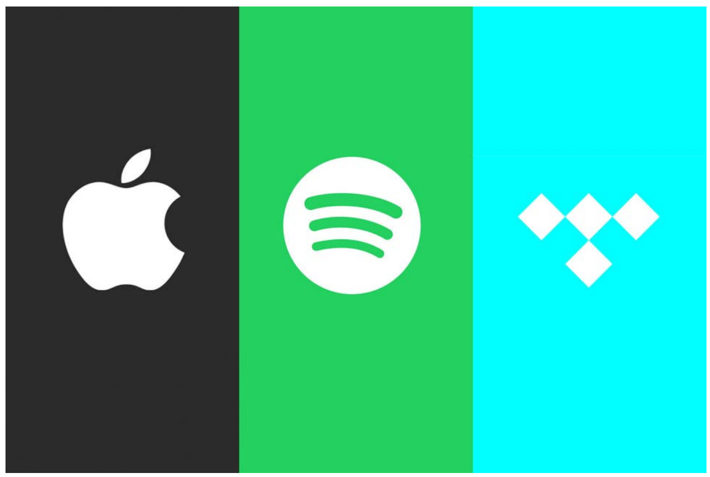


TABLE OF CONTENTS

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 develop 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 evaluating 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 (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	dance
0	Movie	Henri Salvador	C'est beau de faire un Show	0BRjO6ga9RKCKjfDqeFgWV	0	0.611	
1	Movie	Martin & les fées	Perdu d'avance (par Gad Elmaleh)	0BjC1NfoEOOusryehmNudP	1	0.246	
2	Movie	Joseph Williams	Don't Let Me Be Lonely Tonight	0CoSDzoNIKCRs124s9uTVy	3	0.952	
3	Movie	Henri Salvador	Dis-moi Monsieur Gordon Cooper	0Gc6TVm52BwZD07Ki6tIvf	0	0.703	
4	Movie	Fabien Nataf	Ouverture	0IuslXpMROHdEPvSl1fTQK	4	0.950	

In [3]:

```
#Looking at the stats of different columns
df.describe()
```

Out[3]:

	popularity	acousticness	danceability	duration_ms	energy	instrumenta
count	232725.000000	232725.000000	232725.000000	2.327250e+05	232725.000000	232725.0
mean	41.127502	0.368560	0.554364	2.351223e+05	0.570958	0.1
std	18.189948	0.354768	0.185608	1.189359e+05	0.263456	0.3
min	0.000000	0.000000	0.056900	1.538700e+04	0.000020	0.0
25%	29.000000	0.037600	0.435000	1.828570e+05	0.385000	0.0
50%	43.000000	0.232000	0.571000	2.204270e+05	0.605000	0.0
75%	55.000000	0.722000	0.692000	2.657680e+05	0.787000	0.0
max	100.000000	0.996000	0.989000	5.552917e+06	0.999000	0.9

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
                          ... 
REYNA                        1
Your Smith                   1
Juliana Aquino               1
Changmo                      1
Cartel                       1
Name: artist_name, Length: 14564, dtype: int64
--------------------
Column: track_name
Home                                                     100
You                                                       71
Intro                                                     69
Stay                                                      63
Wake Up                                                   59
```

```
                                                    ...
Sweet Little Angel - Live                              1
Organ Donor Programs                                   1
Bone People                                            1
Tú Mereces                                             1
Hope She Cheats On You (With A Basketball Player)      1
Name: track_name, Length: 148615, dtype: int64
--------------------
Column: track_id
0UE0RhnRaEYsiYgXpyLoZc    8
6sVQNUvcVFTXvlk3ec0ngd    8
6AIte2Iej1QKlaofpjCzW1    8
3uSSjnDMmoyERaAK9KvpJR    8
0wY9rA9fJkuESyYm9uzVK5    8
                         ..
2ENWi9dhKyuclKul6QaeLl    1
0hORgg6aV6GgDn5VQfZpcl    1
4UYkjmUTy0YCOotrNcz6uI    1
3Z6ZB2rAjuqsErN2gWzojW    1
64i7jAh24blhIrRuxMWK1U    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	acousticn
1348	Alternative	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.07
1385	Alternative	Frank Ocean	Seigfried	1BViPjTT585XAhkUUrkts0	61	0.97
1452	Alternative	Frank Ocean	Bad Religion	2pMPWE7PJH1PizfgGRMnR9	56	0.77
1554	Alternative	Steve Lacy	Some	4riDfclV7kPDT9D58FpmHd	58	0.00
1634	Alternative	tobi lou	Buff Baby	1F1QmI8TMHir9SUFrooq5F	59	0.19
...	...	...	...	...	...	
232715	Soul	Emily King	Down	5cA0vB8c9FMOVDWyJHgf26	42	0.55
232718	Soul	Muddy Waters	I Just Want To Make Love To You - Electric Mud...	2HFczeynfKGiM9KF2z2K7K	43	0.01
232720	Soul	Slave	Son Of Slide	2XGLdVl7lGeq8ksM6Al7jT	39	0.00
232722	Soul	Muddy Waters	(I'm Your) Hoochie Coochie Man	2ziWXUmQLrXTiYjCg2fZ2t	47	0.90
232723	Soul	R.LUM.R	With My Words	6EFsue2YbIG4Qkq8Zr9Rir	44	0.26

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

Out[11]:

	genre	artist_name	track_name	track_id	popularity	acousticnes
257	R&B	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.07
1348	Alternative	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.07
77710	Children's Music	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.07
93651	Indie	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.07
113770	Pop	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.07

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

Out[12]:

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

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

Out[13]:

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

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 maximu
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 [17]:

```
#removing redundant genre column
df.drop('genre', axis=1, inplace=True)
df.head()
```

Out[17]:

	artist_name	track_name	popularity	acousticness	danceability	d
track_id						
00021Wy6AyMbLP2tqij86e	Capcom Sound Team	Zangief's Theme	13	0.234	0.617	
000CzNKC8PEt1yC3L8dqwV	Henri Salvador	Coeur Brisé à Prendre - Remastered	5	0.249	0.518	
000DfZJww8KiixTKuk9usJ	Mike Love	Earthlings	30	0.366	0.631	
000EWWBkYaREzsBplYjUag	Don Philippe	Fewerdolr	39	0.815	0.768	
000xQL6tZNLJzIrtIgxqSI	ZAYN	Still Got Time	70	0.131	0.748	

5 rows × 42 columns

In [18]:

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

Out[18]:

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

1 rows × 42 columns

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

In [19]:

```
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]:

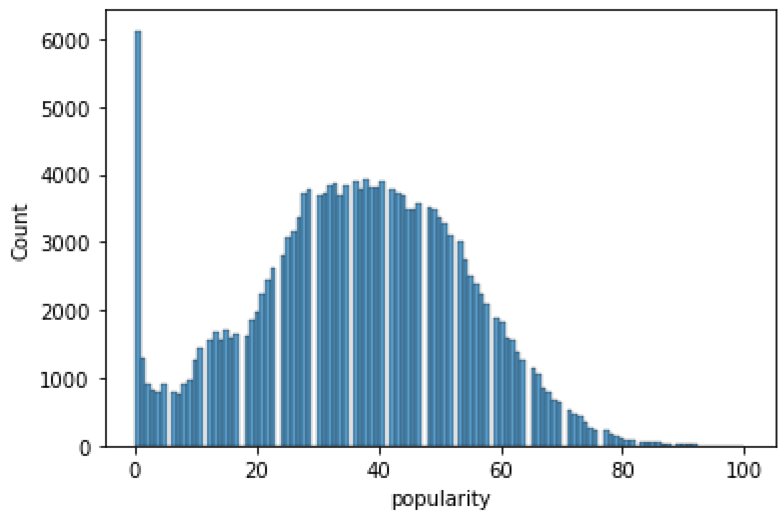
```
import matplotlib.pyplot as plt
import seaborn as sns
```

In [21]:

```
#creating a histogram to see distribution of popularity scores in the datase
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 to keep the analysis consistent.)

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

In [24]:

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

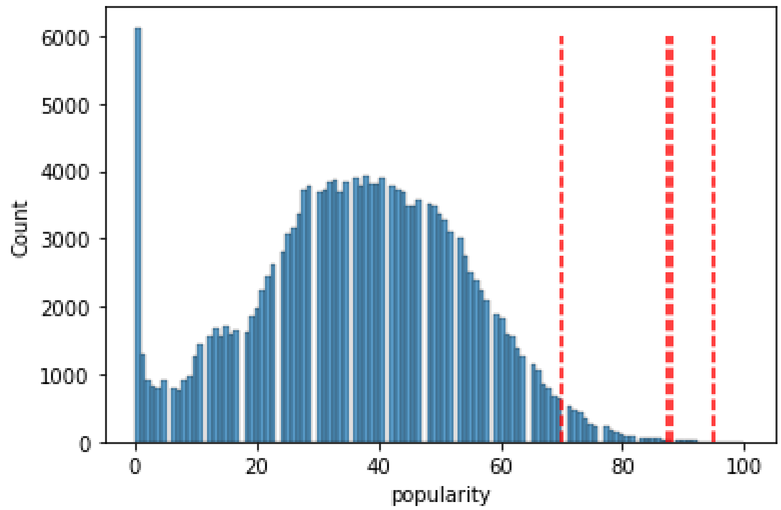
Out[24]:

```
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, lir
```



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 [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()
```

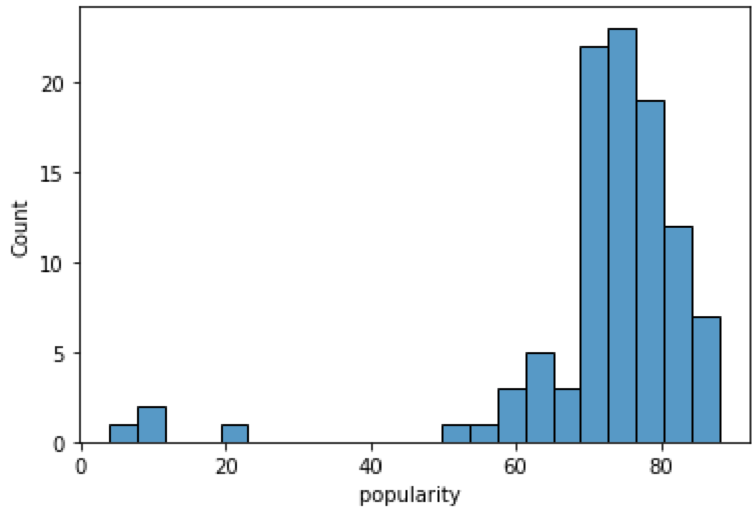
Out[27]:

```
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, 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 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-Note

    """
    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
```

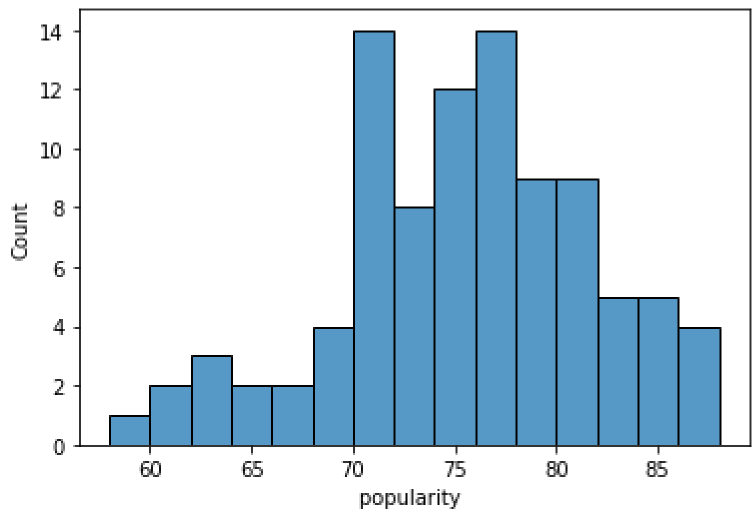
In [30]:

```
#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
```

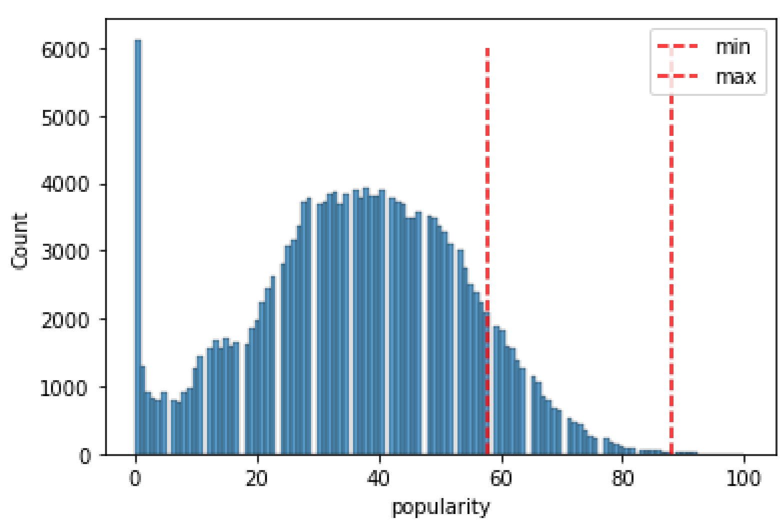
In [31]:
```
fig, ax = plt.subplots()
sns.histplot(df_100['popularity '], bins=15, ax=ax)
```

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



In [32]:
```
#visualizing the min and max popularity scores on the overall dataset histog
fig, ax = plt.subplots()
sns.histplot(df['popularity'], bins='auto', ax=ax)
ax.vlines(x=df_100['popularity '].min(), ymin=0, ymax=6000, linestyles='dash
ax.vlines(x=df_100['popularity '].max(), ymin=0, ymax=6000, linestyles='dash
plt.legend()
```

Out[32]:
```
<matplotlib.legend.Legend at 0x1ddd0f792b0>
```



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]:

	artist_name	track_name	popularity	acousticness	danceability	d
track_id						
00021Wy6AyMbLP2tqij86e	Capcom Sound Team	Zangief's Theme	13	0.234	0.617	
000CzNKC8PEt1yC3L8dqwV	Henri Salvador	Coeur Brisé à Prendre - Remastered	5	0.249	0.518	
000DfZJww8KiixTKuk9usJ	Mike Love	Earthlings	30	0.366	0.631	
000EWWBkYaREzsBplYjUag	Don Philippe	Fewerdolr	39	0.815	0.768	
000xQL6tZNLJzIrtIgxqSI	ZAYN	Still Got Time	70	0.131	0.748	

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]:

	acousticness	danceability	duration_ms	energy	instrumentalness
track_id					
00021Wy6AyMbLP2tqij86e	0.234	0.617	169173	0.862	0.976000
000CzNKC8PEt1yC3L8dqwV	0.249	0.518	130653	0.805	0.000000
000DfZJww8KiixTKuk9usJ	0.366	0.631	357573	0.513	0.000004
000EWWBkYaREzsBplYjUag	0.815	0.768	104924	0.137	0.922000
000xQL6tZNLJzIrtIgxqSI	0.131	0.748	188491	0.627	0.000000

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 [35]:
```
#splitting the data to training and test sets in order to be able to measure
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, ra
```

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 [36]:

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

Out[36]:

```
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 [37]:

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

In [38]:

```
#One hot encoding the dataframes
from sklearn.preprocessing import OneHotEncoder

encoder=OneHotEncoder(sparse=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_name

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

In [39]:

```
pd.set_option("display.max_columns", None)
df_ohe_train
```

Out[39]:

	key_A#	key_B	key_C	key_C#	key_D	key_D#	key_E	key_F
track_id								
5SbN8lXhPno4BRrFb9yqkF	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
34n3eoeqVaXAgtMqy8Ncyz	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
3QbxHo2OTwBVDZbaJaMniP	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
6Y8aA0SWBMB5XTZlXlDpYv	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
16h3GCdEJ9lgiOyox4LJQA	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...
5H23l3K3TUXQMsLg2FzCiY	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
4YnYtYWBmDM8YjfMMK0cqs	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
5o5xTG5Mh3JAm2BZv4nOl7	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
6T1oL7ed1wUEqlCR1iCplR	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5MnEYPkZ5HC7BQ988kBKqp	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

123741 rows × 16 columns

In [40]:

```
#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[40]:

	acousticness	danceability	duration_ms	energy	instrumentalnes
track_id					
5H23I3K3TUXQMsLg2FzCiY	0.23100	0.461	326680	0.252	0.00008
4YnYtYWBmDM8YjfMMK0cqs	0.00571	0.309	201947	0.820	0.00036
5o5xTG5Mh3JAm2BZv4nOI7	0.01210	0.691	265587	0.834	0.00148
6T1oL7ed1wUEqICR1iCpIR	0.35500	0.364	224387	0.733	0.00000
5MnEYPkZ5HC7BQ988kBKqp	0.01190	0.478	211800	0.695	0.00000

In [41]:

```
#concatenating all parts of our data for future reference (see Data Visualiz
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)
```

With both the X_train and X_test dataframes scrubbed and one hot encoded we can move onto the modelling process.

MODEL

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 [42]:

```
from sklearn.dummy import DummyClassifier

clf_dummy = DummyClassifier(random_state=42)
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
  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.

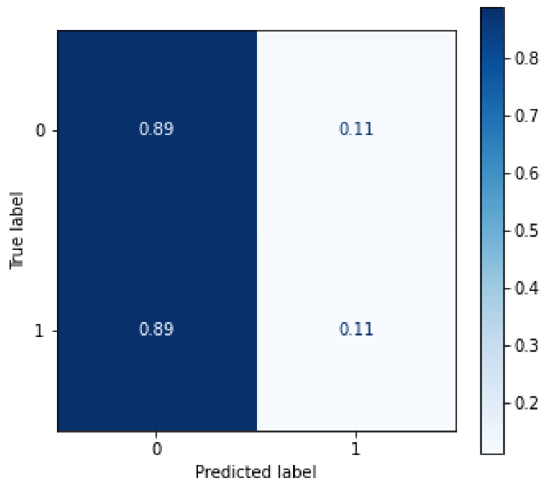
In [43]:

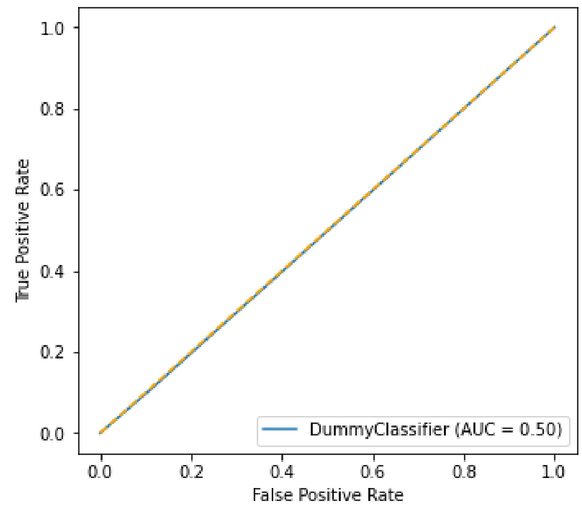
```
from sklearn.metrics import classification_report, plot_confusion_matrix, pl

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 qu

    y_true: Correct y values, typically y_test that comes from the train_tes
    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', r

    #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 [44]:

```
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 [45]:

```
#class imbalance percentages
y_train.value_counts(normalize=True)
```

Out[45]:

```
0    0.885503
1    0.114497
Name: is_popular, dtype: float64
```

Our dummy classifier correctly predicted 89% of the unpopular songs as unpopular; however, it correctly predicted only 11% of the popular songs as popular and instead classified 89% of them as unpopular as well. We clearly have a class imbalance problem where approximately 89% of our data is not popular and only about 11% 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 [46]:

```
#Looking at column names to extract categorical column indices for SMOTENC
X_train.columns
```

Out[46]:

```
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 [47]:

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

Out[47]:

```
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 [48]:

```
#Using SMOTENC to address class imbalance. We are not using SMOTE since we h
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[48]:

```
1    0.5
0    0.5
Name: is_popular, dtype: float64
```

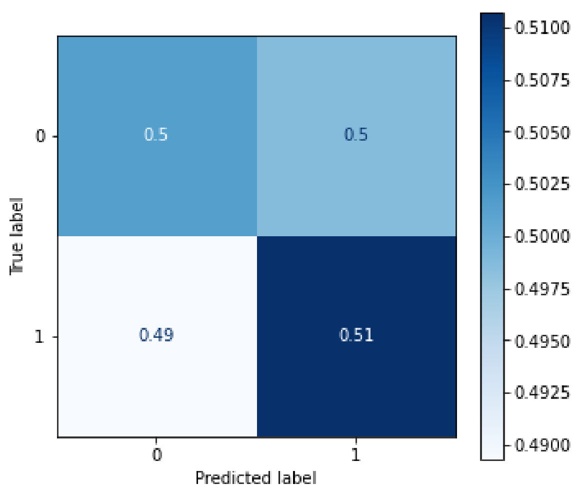
Now that we addressed our class imbalance problem, we can look at the performance of the dummy classifier model once again to use as our baseline.

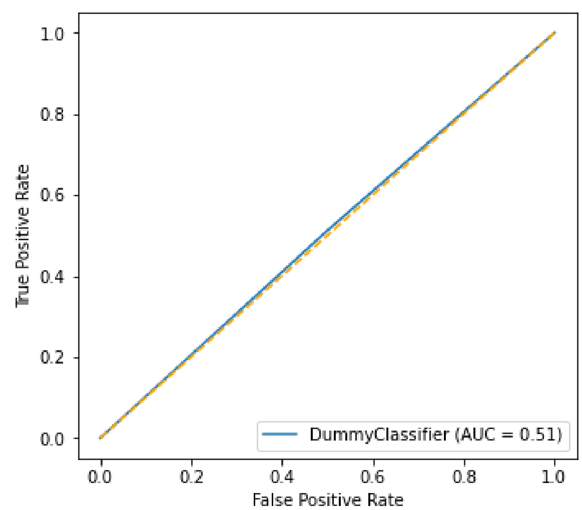
In [49]:

```
#fitting Dummy Classifier to data without the class imbalance problem to ser
clf_dummy_sm = DummyClassifier(random_state=42)
clf_dummy_sm.fit(X_train_sm, y_train_sm)
y_pred = clf_dummy_sm.predict(X_test)
classification(y_test, y_pred, X_test, clf_dummy_sm)
```

```
C:\Users\berke\anaconda3\envs\learn-env\lib\site-packages\sklearn\dummy.py:1
  warnings.warn("The default value of strategy will change from "
```

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

           0       0.89      0.50      0.64     47002
           1       0.12      0.51      0.19      6031

    accuracy                           0.50     53033
   macro avg       0.50      0.51      0.42     53033
weighted avg       0.80      0.50      0.59     53033
```





We see here that the dummy classifier is essentially flipping a coin and guessing whether a song is popular or not which is not very useful. However, this serves as a great baseline for our other models to be evaluated against. We can now initialize a results dataframe and keep track of the recall scores of our models for comparison later.

In [50]:

```
from sklearn.metrics import recall_score

df_results = pd.DataFrame(columns=['Model Name', 'Recall Score'])

def add_results(model_name, df):
    df = df.append({'Model Name': model_name,
                                    'Recall Score': round(recall_score(y_tes
                                   ignore_index=True)
    return df
```

In [51]:

```
df_results = add_results('Dummy Classifier', df_results)
df_results.head()
```

Out[51]:

	Model Name	Recall Score
0	Dummy Classifier	0.51

Model #2 - Random Forest Classifier

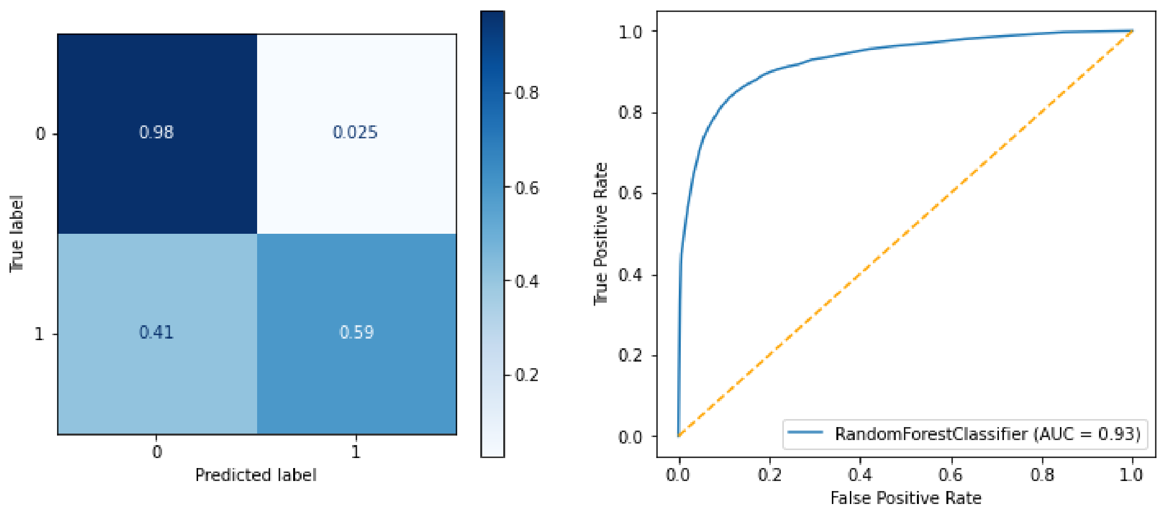
The first model we will be developing is the Random Forest classifier.

Initial Model

In [52]:

```
#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      0.98      0.96     47002
           1       0.75      0.59      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
```

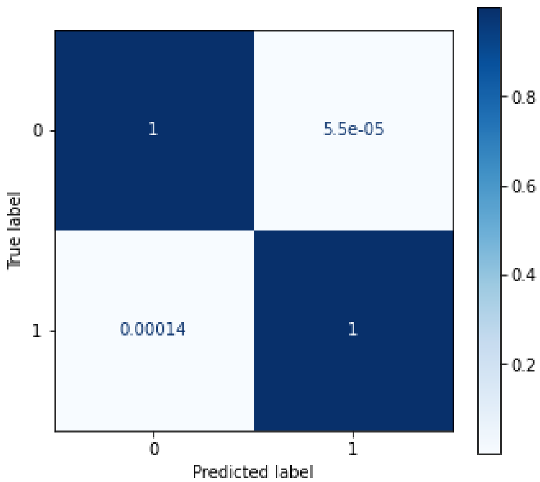


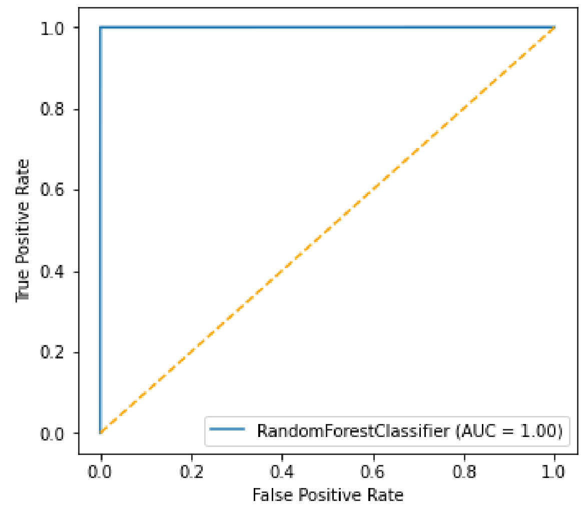
Our Random Forest model performs 48% better than the baseline classifier in predicting unpopular songs correctly and 8% better in predicting popular songs. The model may be overfitting, so to confirm we will look at the performance of the model with the training data.

In [53]:

```
#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

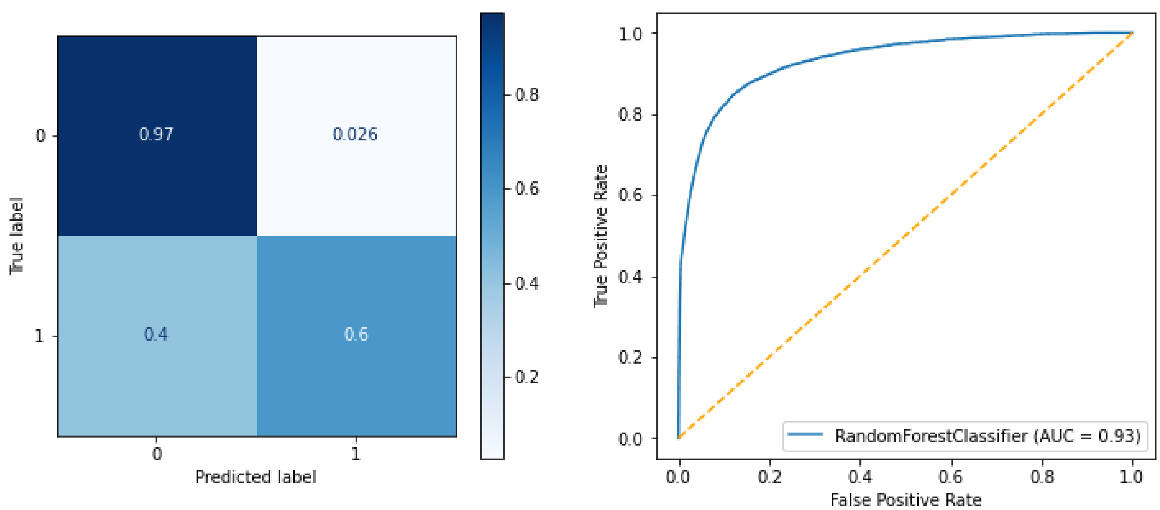
In [54]:

```
# 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='recal

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

In [55]:

```
clf_rf_tuned = RandomForestClassifier(criterion='entropy', max_depth=None,
                                      min_samples_leaf=2, class_weight='bala
                                      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.

In [56]:

```
#appending the recall score to the results dataframe
df_results = add_results('Random Forest', df_results)
df_results.head()
```

Out[56]:

	Model Name	Recall Score
0	Dummy Classifier	0.51
1	Random Forest	0.60

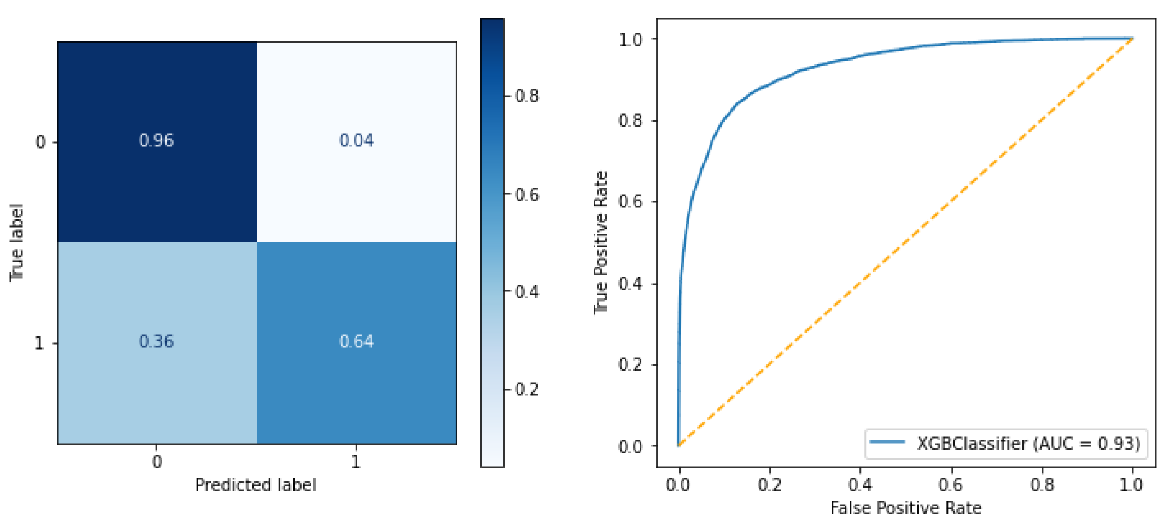
Model #3 - XGBoost

Initial Model

In [57]:

```
#Fitting XGBoost classifier to training data and evaluating results
from xgboost import XGBClassifier

clf_xgb = XGBClassifier(random_state=42)
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)
```

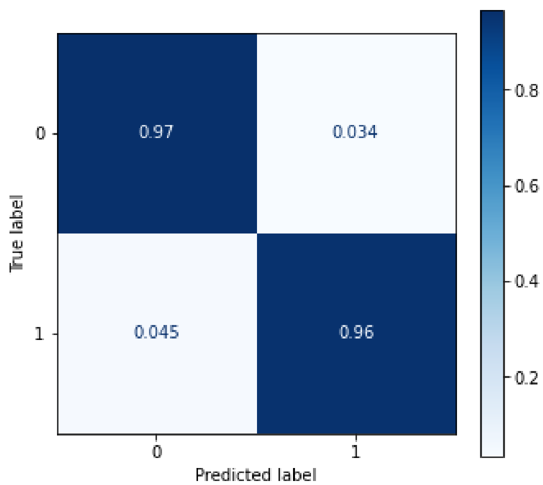
```
CLASSIFICATION REPORT
------------------------------------------
              precision    recall  f1-score   support

           0       0.95      0.96      0.96     47002
           1       0.67      0.64      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
```

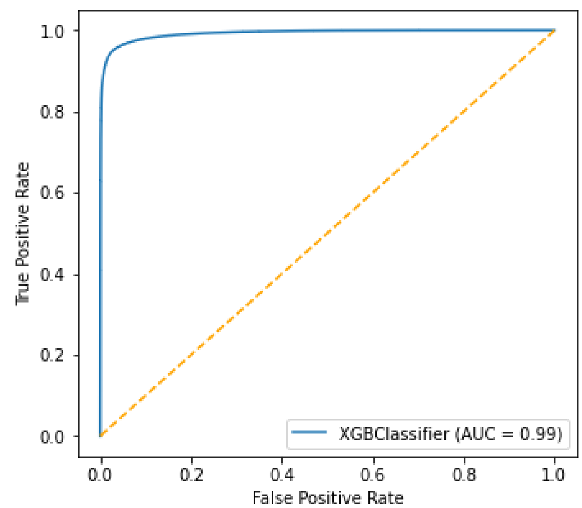


The XGBoost model performed 13% better than the baseline model and 4% better than the random forest model in predicting popular songs right out of the box. We can see how it performs on the training data to see whether it is overfitting and try to tune it if it is.

In [58]:
```
#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, we see here that 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.

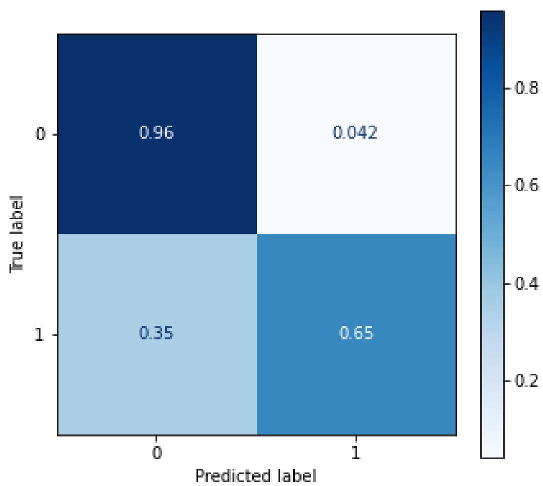
Hyperparameter Tuning

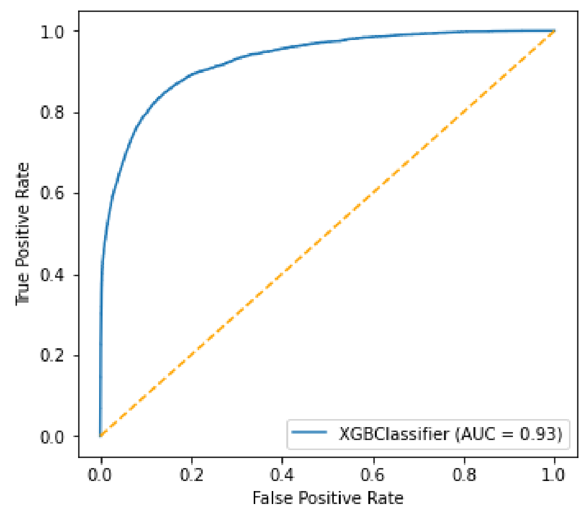
In [59]:
```
# grid = {
#     'learning_rate': [0.01, 0.1, 0.2],
#     'max_depth': [10, 20, None]
#         }
# gridsearch = GridSearchCV(estimator=clf_xgb, param_grid = grid, scoring='r

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

In [60]:

```
clf_xgb_tuned = XGBClassifier(learning_rate=0.1, max_depth=10,
                              random_state=42)
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      0.66      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 14% better compared to our baseline Dummy Classifier model and 5% better than our tuned Random Forest model. Next we will try a logistic regression model.

In [61]:

```
#appending the recall score to the results dataframe
df_results = add_results('XGBoost', df_results)
df_results.head()
```

Out[61]:

	Model Name	Recall Score
0	Dummy Classifier	0.51
1	Random Forest	0.60
2	XGBoost	0.65

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 [62]:

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

Out[62]:

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

In [63]:

```
#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 [64]:

```
#finding and removing outliers based on X_train (df_train) to avoid data lea

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_limit

    df_train = df_train[(df_train[col]>lower_limit) & (df_train[col]<upper_l
    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 traini
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 [65]:

```
#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)
```

Addressing Class Imbalance with SMOTENC

In [66]:

```
y_train.value_counts(normalize=True)
```

Out[66]:

```
0    0.842345
1    0.157655
Name: is_popular, dtype: float64
```

Once again our data has a class imbalance issue so we will be using SMOTENC to address this.

In [67]:

```
X_train.columns
```

Out[67]:

```
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 [68]:

```
cat_cols = list(range(10,len(X_train.columns)))
cat_cols
```

Out[68]:

```
[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 [69]:

```
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[69]:

```
1    0.5
0    0.5
Name: is_popular, dtype: float64
```

Scaling the Data

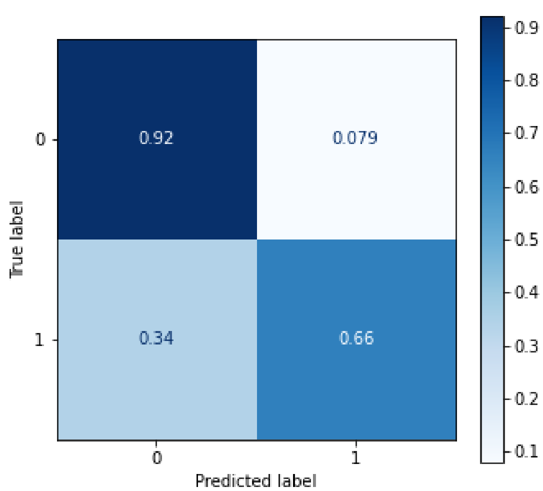
In [70]:

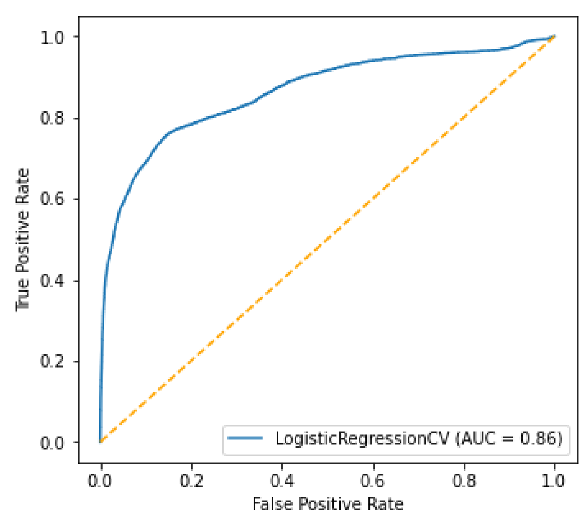
```
#Using Standard Scaler to scale the smote'd data
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

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

In [71]:

```
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
```

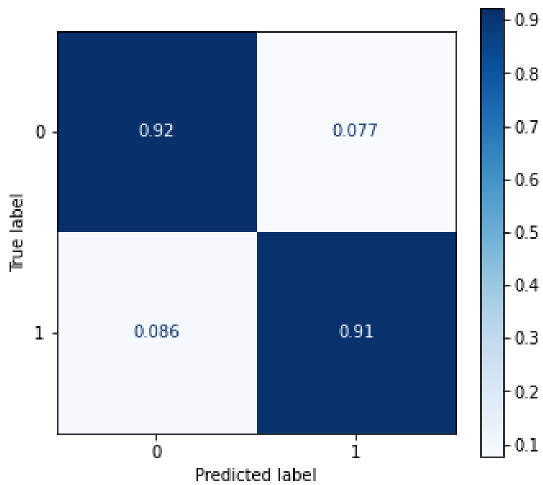


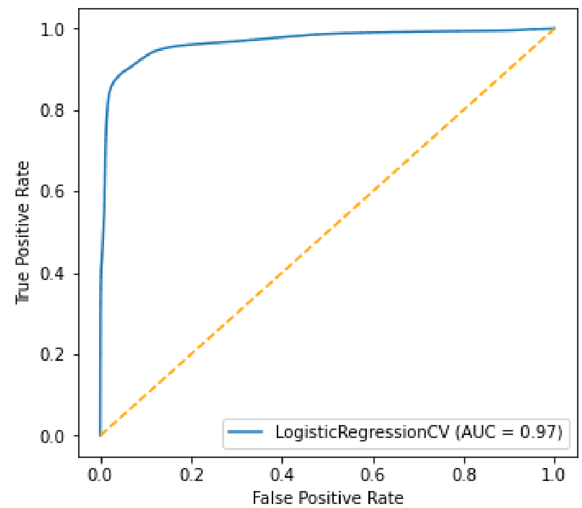


The logistic regression model has 1% better performance compared to our tuned XGBoost model for predicting popular songs while the recall score for predicting unpopular songs is 4% lower. Once again, we will check to see if the model is overfitting and tune the model if it is.

In [72]:
```
#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
```





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 [73]:

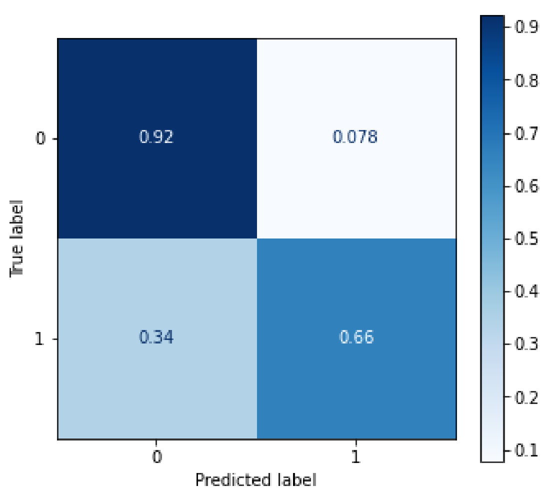
```
# clf = LogisticRegressionCV(cv=5)
# grid = {'penalty': ['l1','l2'],
#         'solver': ['liblinear', 'lbfgs', 'sag', 'saga'],
#         'class_weight': ['balanced', None],
#         'Cs': [1e12, 10, 1, 0.1]
#    }

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

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

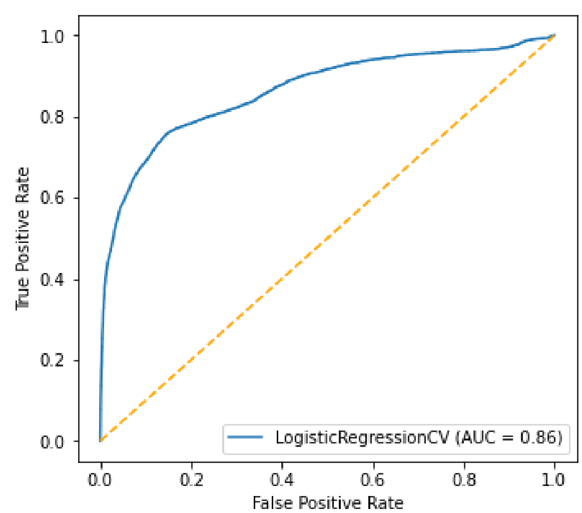
The grid search returned 'l2' as the regularization method which is the Ridge regularization as well as a C value of 1. We will use these parameters on a new model to see if the recall score improves.

In [74]:

```
clf_logregcv_tuned = LogisticRegressionCV(cv=5, class_weight='balanced', Cs=
                                          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.

In [75]:

```
#appending the recall score to the results dataframe
df_results = add_results('Logistic Regression', df_results)
df_results.head()
```

Out[75]:

	Model Name	Recall Score
0	Dummy Classifier	0.51
1	Random Forest	0.60
2	XGBoost	0.65
3	Logistic Regression	0.66

iNTERPRET

Now that we have 3 tuned models, we can analyze which attributes they used in predicting whether a song was going to be popular or not and interpret these values. For this we will be looking at feature importances of each model and comparing them against each other to see if we can see any common threads between the models.

Parsing Feature Importances to Dataframes

Random Forest

In [76]:

```
#accessing feature importance values of the tuned random forest model and sc
rf_importances_df = pd.Series(clf_rf_tuned.feature_importances_, index=X_tra
#parsing the series to a dataframe
rf_importances_df = rf_importances_df.reset_index()
rf_importances_df.columns = ['RF-Attribute', 'RF-Importance']
rf_importances_df
```

Out[76]:

	RF-Attribute	RF-Importance
0	Pop	0.170450
1	acousticness	0.058465
2	loudness	0.042098
3	instrumentalness	0.035188
4	energy	0.030445
5	speechiness	0.027606
6	Reggae	0.025547
7	Ska	0.025070
8	danceability	0.024540
9	valence	0.023421
10	Rock	0.023300
11	duration_ms	0.022952
12	Anime	0.022609
13	key_C	0.022232
14	Electronic	0.022210
15	Reggaeton	0.022066
16	key_G	0.020926
17	key_D	0.020827
18	liveness	0.020508
19	Blues	0.020394
20	key_C#	0.017425
21	time_signature_4/4	0.017337
22	Country	0.016717
23	World	0.016677
24	tempo	0.016582
25	key_F	0.016250
26	key_E	0.016020
27	key_B	0.015608
28	Jazz	0.014793
29	Soul	0.014269

	RF-Attribute	RF-Importance
30	key_A#	0.013765
31	key_G#	0.013732
32	Rap	0.013512
33	Movie	0.012420
34	key_F#	0.012048
35	Folk	0.011505
36	Comedy	0.009459
37	R&B	0.009058
38	Children's Music	0.008871
39	time_signature_3/4	0.008178
40	Hip-Hop	0.006198
41	Indie	0.006080
42	key_D#	0.006018
43	Alternative	0.005233
44	Dance	0.004591
45	Soundtrack	0.004552
46	Classical	0.004386
47	Opera	0.004076
48	mode_Minor	0.003241
49	time_signature_5/4	0.000374
50	time_signature_1/4	0.000124
51	A Capella	0.000049

XGBoost

In [77]:

```
#parsing feature importances to a series and sorting
xgb_importances_df = pd.Series(clf_xgb_tuned.feature_importances_, index=X_t
#parsing the series to a dataframe
xgb_importances_df = xgb_importances_df.reset_index()
xgb_importances_df.columns=['XGB-Attribute', 'XGB-Importance']
xgb_importances_df
```

Out[77]:

	XGB-Attribute	XGB-Importance
0	Pop	0.338478
1	Blues	0.044411
2	Ska	0.041425
3	Anime	0.038448
4	Electronic	0.035823
5	key_F	0.030206
6	Reggae	0.028680
7	Reggaeton	0.028236
8	World	0.026180
9	Comedy	0.022677
10	time_signature_4/4	0.022424
11	key_G	0.021797
12	key_D	0.021443
13	key_E	0.020443
14	key_B	0.019590
15	Movie	0.019176
16	key_C	0.019146
17	key_C#	0.018567
18	key_A#	0.018249
19	Country	0.017618
20	Jazz	0.017153
21	key_G#	0.017010
22	key_F#	0.015459
23	key_D#	0.011452
24	Rock	0.011072
25	Soul	0.010851
26	Opera	0.008906
27	Folk	0.008619
28	Rap	0.008345
29	Classical	0.007541

	XGB-Attribute	XGB-Importance
30	R&B	0.006807
31	Soundtrack	0.006583
32	Children's Music	0.006214
33	acousticness	0.005366
34	Hip-Hop	0.003538
35	A Capella	0.003390
36	Indie	0.002931
37	Alternative	0.002512
38	instrumentalness	0.002380
39	loudness	0.001734
40	Dance	0.001682
41	time_signature_1/4	0.001456
42	speechiness	0.000771
43	energy	0.000751
44	danceability	0.000741
45	duration_ms	0.000706
46	liveness	0.000661
47	valence	0.000645
48	time_signature_5/4	0.000507
49	mode_Minor	0.000457
50	time_signature_3/4	0.000401
51	tempo	0.000339

LogisticRegressionCV

In [78]:

```
#accessing feature importance values of the tuned logistic regression model
logregcv_importances_df = pd.Series(clf_logregcv_tuned.coef_[0], index=X_tra
#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[78]:

	LogReg-Attribute	LogReg-Importance
0	Pop	0.602770
1	Rock	0.309627
2	danceability	0.117183
3	loudness	0.102000
4	Rap	0.097957
5	time_signature_4/4	0.088191
6	Dance	0.066667
7	duration_ms	0.023599
8	Hip-Hop	0.014992
9	speechiness	-0.005829
10	tempo	-0.011091
11	time_signature_1/4	-0.023202
12	Indie	-0.023943
13	acousticness	-0.026369
14	Alternative	-0.029341
15	time_signature_5/4	-0.030763
16	energy	-0.031187
17	A Capella	-0.034542
18	mode_Minor	-0.037862
19	liveness	-0.040334
20	instrumentalness	-0.042670
21	Soundtrack	-0.056095
22	Comedy	-0.075320
23	time_signature_3/4	-0.082948
24	valence	-0.097720
25	Classical	-0.103312
26	R&B	-0.107484
27	Opera	-0.140450
28	key_D#	-0.146706
29	Children's Music	-0.150627

	LogReg-Attribute	LogReg-Importance
30	Jazz	-0.165628
31	Soul	-0.177035
32	Folk	-0.178005
33	key_F#	-0.198849
34	Electronic	-0.203210
35	key_A#	-0.214184
36	key_G#	-0.217816
37	key_B	-0.232799
38	key_E	-0.235655
39	Movie	-0.240681
40	key_F	-0.245327
41	key_C#	-0.245345
42	World	-0.247189
43	Country	-0.253770
44	Blues	-0.259086
45	Reggaeton	-0.265262
46	key_G	-0.279556
47	key_D	-0.280966
48	Anime	-0.291580
49	Reggae	-0.296104
50	key_C	-0.296984
51	Ska	-0.313558

In [79]:

```
#Concatenating feature importances into a single dataframe
importances_df = pd.concat([rf_importances_df, xgb_importances_df, logregcv_
importances_df
```

Out[79]:

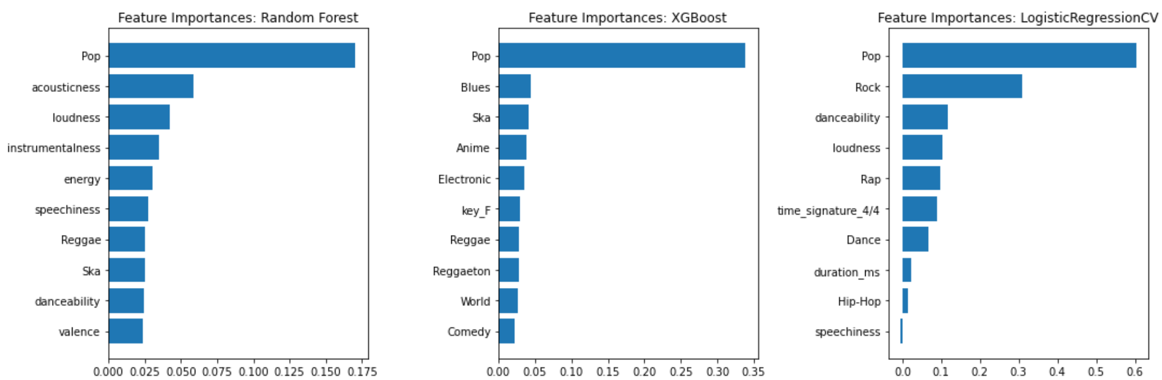
	RF-Attribute	RF-Importance	XGB-Attribute	XGB-Importance	LogReg-Attribute	LogReg Importanc
0	Pop	0.170450	Pop	0.338478	Pop	0.60277
1	acousticness	0.058465	Blues	0.044411	Rock	0.30962
2	loudness	0.042098	Ska	0.041425	danceability	0.11718
3	instrumentalness	0.035188	Anime	0.038448	loudness	0.10200
4	energy	0.030445	Electronic	0.035823	Rap	0.09795
5	speechiness	0.027606	key_F	0.030206	time_signature_4/4	0.08819
6	Reggae	0.025547	Reggae	0.028680	Dance	0.06666
7	Ska	0.025070	Reggaeton	0.028236	duration_ms	0.02359
8	danceability	0.024540	World	0.026180	Hip-Hop	0.01499
9	valence	0.023421	Comedy	0.022677	speechiness	-0.00582
10	Rock	0.023300	time_signature_4/4	0.022424	tempo	-0.01109
11	duration_ms	0.022952	key_G	0.021797	time_signature_1/4	-0.02320
12	Anime	0.022609	key_D	0.021443	Indie	-0.02394
13	key_C	0.022232	key_E	0.020443	acousticness	-0.02636
14	Electronic	0.022210	key_B	0.019590	Alternative	-0.02934
15	Reggaeton	0.022066	Movie	0.019176	time_signature_5/4	-0.03076
16	key_G	0.020926	key_C	0.019146	energy	-0.03118
17	key_D	0.020827	key_C#	0.018567	A Capella	-0.03454
18	liveness	0.020508	key_A#	0.018249	mode_Minor	-0.03786
19	Blues	0.020394	Country	0.017618	liveness	-0.04033
20	key_C#	0.017425	Jazz	0.017153	instrumentalness	-0.04267
21	time_signature_4/4	0.017337	key_G#	0.017010	Soundtrack	-0.05609
22	Country	0.016717	key_F#	0.015459	Comedy	-0.07532
23	World	0.016677	key_D#	0.011452	time_signature_3/4	-0.08294
24	tempo	0.016582	Rock	0.011072	valence	-0.09772
25	key_F	0.016250	Soul	0.010851	Classical	-0.10331
26	key_E	0.016020	Opera	0.008906	R&B	-0.10748
27	key_B	0.015608	Folk	0.008619	Opera	-0.14045
28	Jazz	0.014793	Rap	0.008345	key_D#	-0.14670
29	Soul	0.014269	Classical	0.007541	Children's Music	-0.15062
30	key_A#	0.013765	R&B	0.006807	Jazz	-0.16562
31	key_G#	0.013732	Soundtrack	0.006583	Soul	-0.17703

	RF-Attribute	RF-Importance	XGB-Attribute	XGB-Importance	LogReg-Attribute	LogReg Importanc
32	Rap	0.013512	Children's Music	0.006214	Folk	-0.17800
33	Movie	0.012420	acousticness	0.005366	key_F#	-0.19884
34	key_F#	0.012048	Hip-Hop	0.003538	Electronic	-0.20321
35	Folk	0.011505	A Capella	0.003390	key_A#	-0.21418
36	Comedy	0.009459	Indie	0.002931	key_G#	-0.21781
37	R&B	0.009058	Alternative	0.002512	key_B	-0.23279
38	Children's Music	0.008871	instrumentalness	0.002380	key_E	-0.23565
39	time_signature_3/4	0.008178	loudness	0.001734	Movie	-0.24068
40	Hip-Hop	0.006198	Dance	0.001682	key_F	-0.24532
41	Indie	0.006080	time_signature_1/4	0.001456	key_C#	-0.24534
42	key_D#	0.006018	speechiness	0.000771	World	-0.24718
43	Alternative	0.005233	energy	0.000751	Country	-0.25377
44	Dance	0.004591	danceability	0.000741	Blues	-0.25908
45	Soundtrack	0.004552	duration_ms	0.000706	Reggaeton	-0.26526
46	Classical	0.004386	liveness	0.000661	key_G	-0.27955
47	Opera	0.004076	valence	0.000645	key_D	-0.28096
48	mode_Minor	0.003241	time_signature_5/4	0.000507	Anime	-0.29158
49	time_signature_5/4	0.000374	mode_Minor	0.000457	Reggae	-0.29610
50	time_signature_1/4	0.000124	time_signature_3/4	0.000401	key_C	-0.29698
51	A Capella	0.000049	tempo	0.000339	Ska	-0.31355

Feature Importance Comparison

In [80]:

```
#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='RF-Importance', ascenc
ax[0].barh(rf_importances_df['RF-Attribute'], rf_importances_df['RF-Importar
ax[0].set_title('Feature Importances: Random Forest')

xgb_importances_df = xgb_importances_df.sort_values(by='XGB-Importance', asc
ax[1].barh(xgb_importances_df['XGB-Attribute'], xgb_importances_df['XGB-Impc
ax[1].set_title('Feature Importances: XGBoost')

logregcv_importances_df = logregcv_importances_df.sort_values(by='LogReg-Imp
ax[2].barh(logregcv_importances_df['LogReg-Attribute'], logregcv_importances
ax[2].set_title('Feature Importances: LogisticRegressionCV')
plt.tight_layout()
```

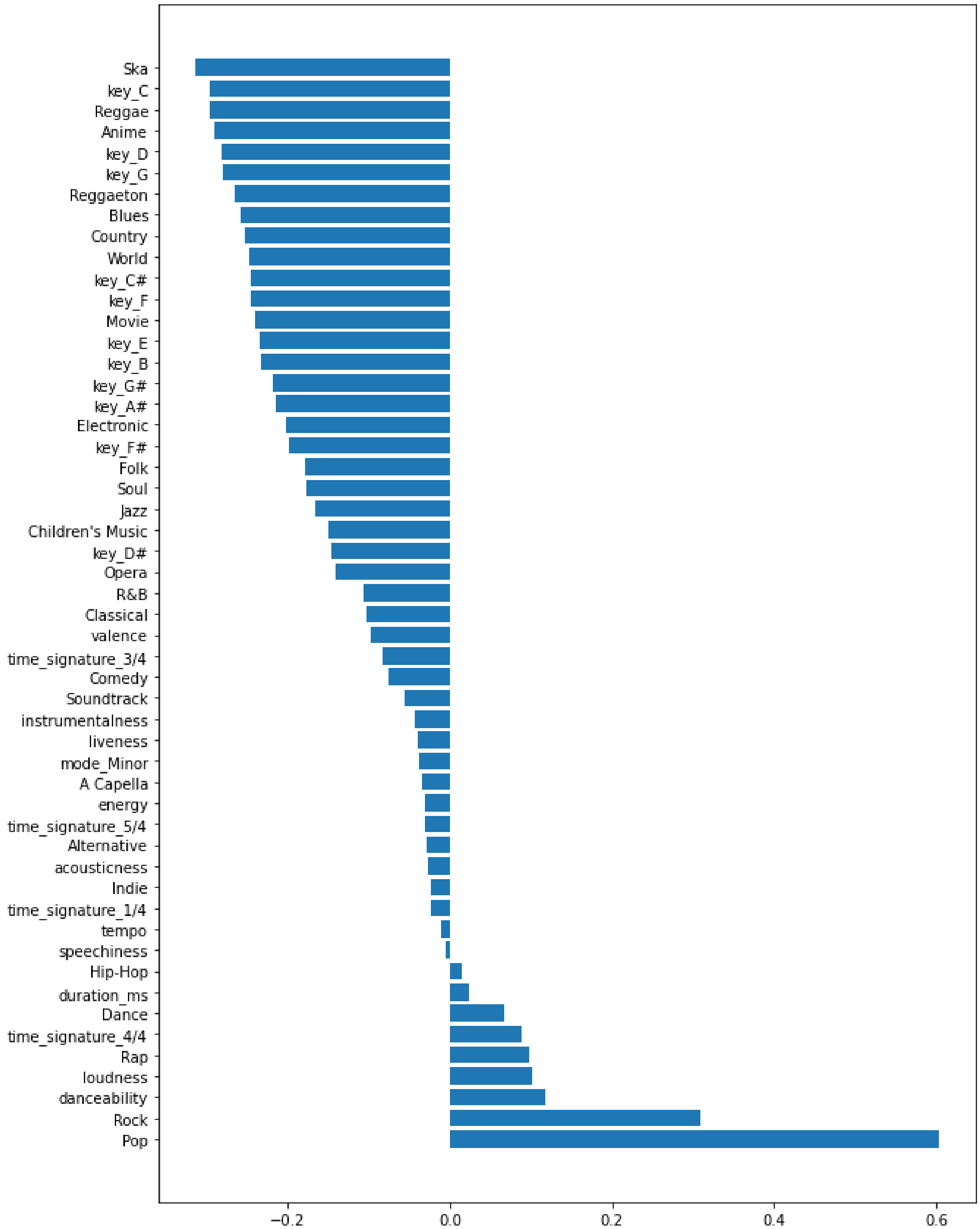


Among the 3 models we built we can see that Genre of a song has the highest effect on the popularity of a song. On all 3 models, a song having Pop as its genre had the most impact on its popularity. This makes sense since Pop songs by nature are considered popular. Among the rest of the features shown above, 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 [81]:

```
logregcv_importances_df = pd.Series(clf_logregcv_tuned.coef_[0], index=X_tra
#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['Impor
```

Out[81]: <BarContainer object of 52 artists>



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 [82]:
```
#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]
```

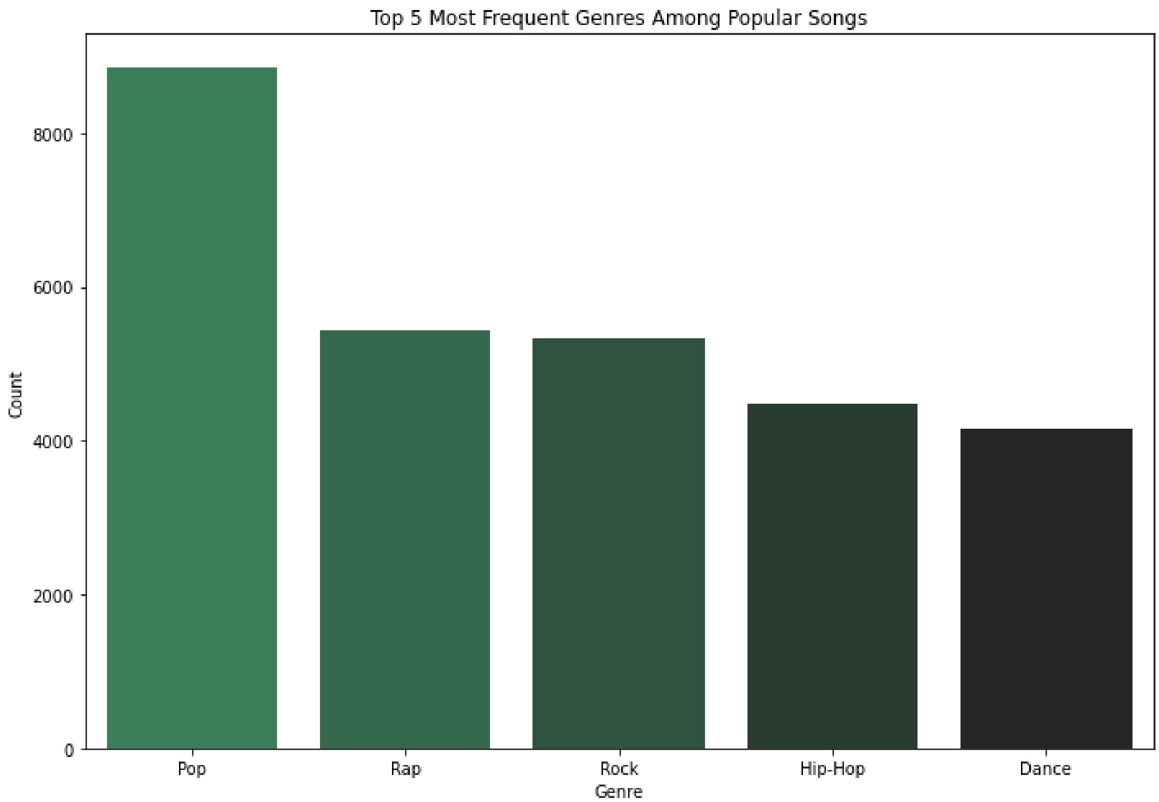
In [83]:

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

Out[83]:

	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 [84]:
```
fig, ax = plt.subplots(figsize=(10, 7))
sns.barplot(x=popular_genre_df['genre'].head(5), y=popular_genre_df['count']
           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/ (https://www.statista.com/chart/15763/most-popular-music-genres-worldwide/)).

In [85]:

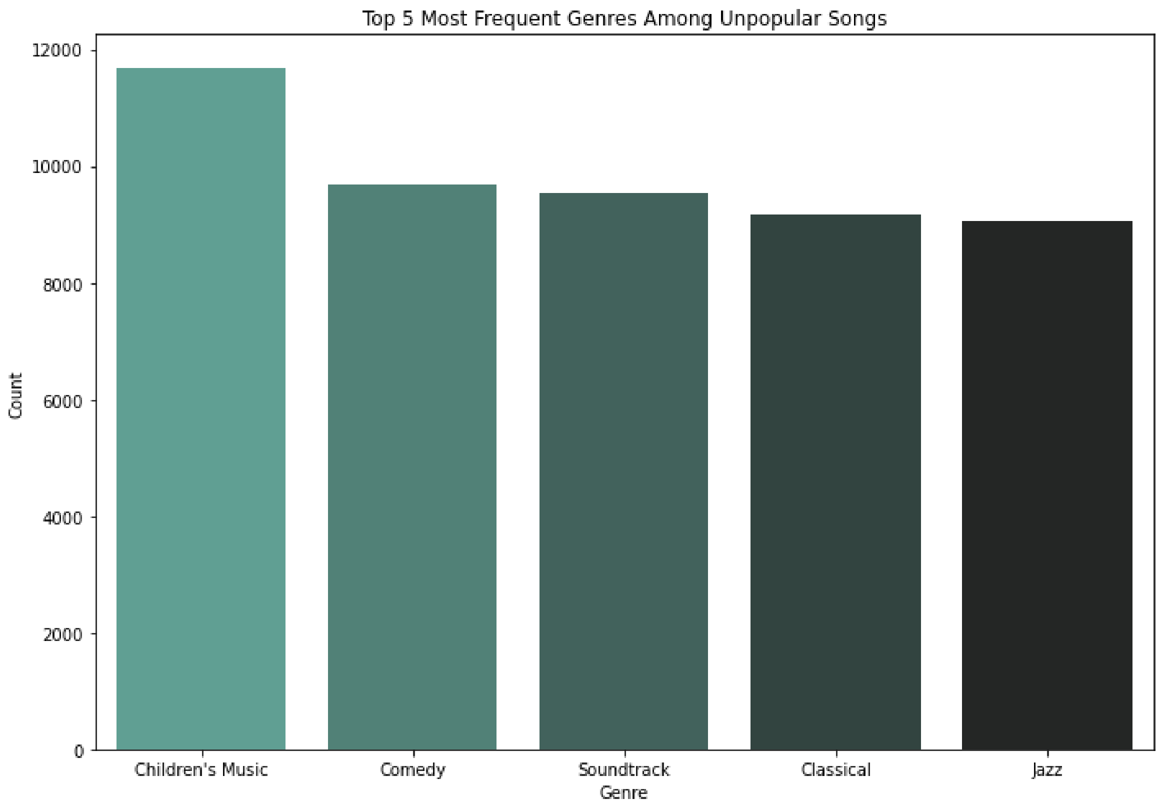
```
#checking for genre occurence counts for unpopular songs
unpopular_genre_df = unpopular_songs_df.iloc[:, 10:36].agg('sum').sort_value
unpopular_genre_df.columns = ['genre', 'count']
unpopular_genre_df
```

Out[85]:

	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 [86]:

```
fig, ax = plt.subplots(figsize=(10,7))
sns.barplot(x=unpopular_genre_df['genre'].head(5), y=unpopular_genre_df['cou
           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.

In [87]:

```
#displaying percentages for each genre
popular_genre_df['count']=popular_genre_df['count']/popular_genre_df['count'
popular_genre_df
```

Out[87]:

	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 [88]:

```
#displaying percentages for each genre
unpopular_genre_df['count']=unpopular_genre_df['count']/unpopular_genre_df['
unpopular_genre_df
```

Out[88]:

	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

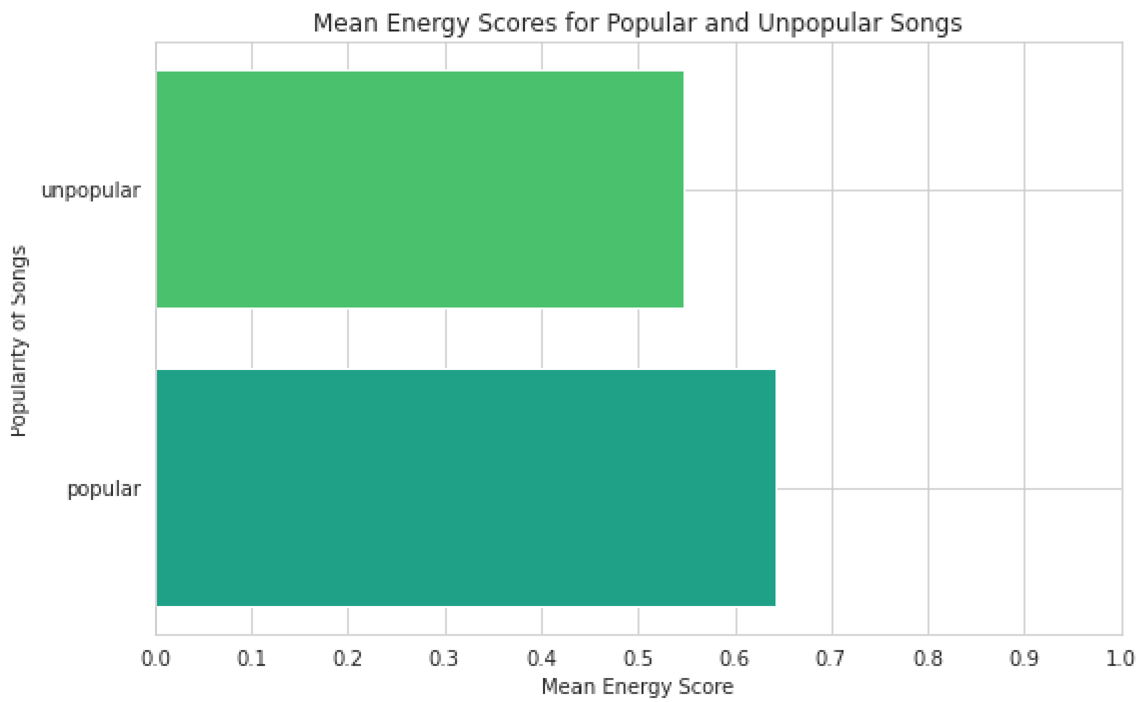
In [89]:

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

unpopular_energy_clean = unpopular_songs_df[find_outliers_IQR(unpopular_song
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 [90]:

```
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('viridi
    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 [91]:

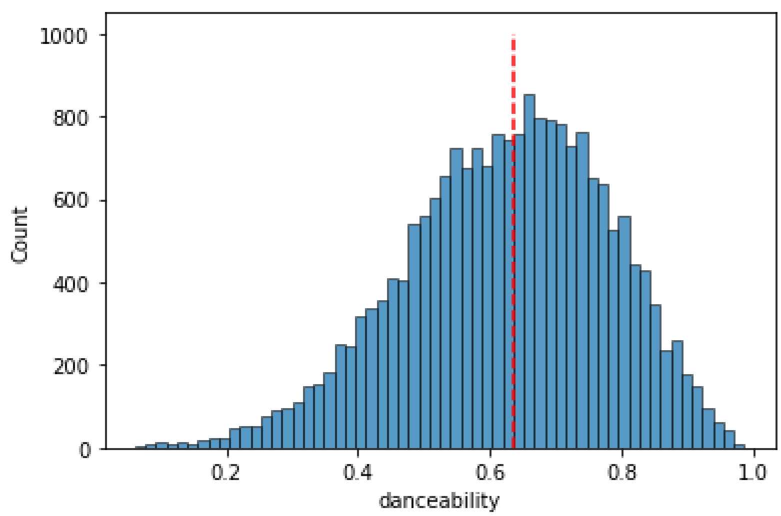
```
print('Median Danceability Scores')
print('-------------------')
print(f"Unpopular Songs: {round(unpopular_songs_df['danceability'].median(),
print(f"Popular Songs: {round(popular_songs_df['danceability'].median(),2)}'
```

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

In [92]:

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

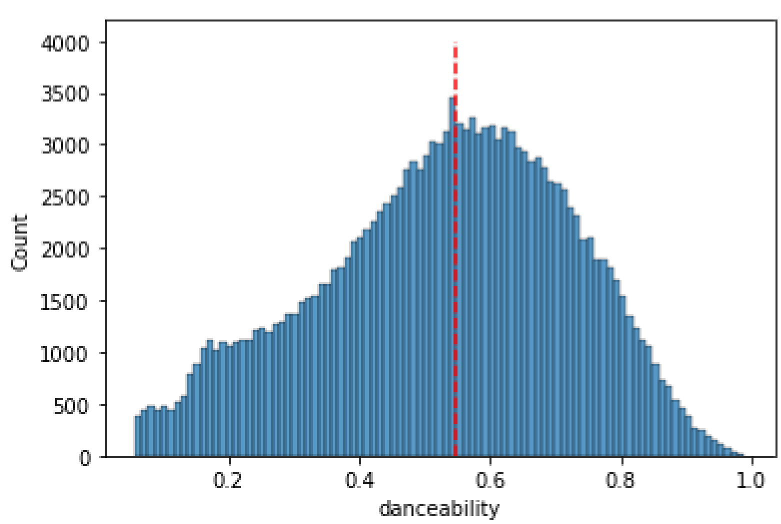
Out[92]: <matplotlib.collections.LineCollection at 0x1ddd82e5d30>



In [93]:

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

Out[93]: <matplotlib.collections.LineCollection at 0x1ddd6880880>



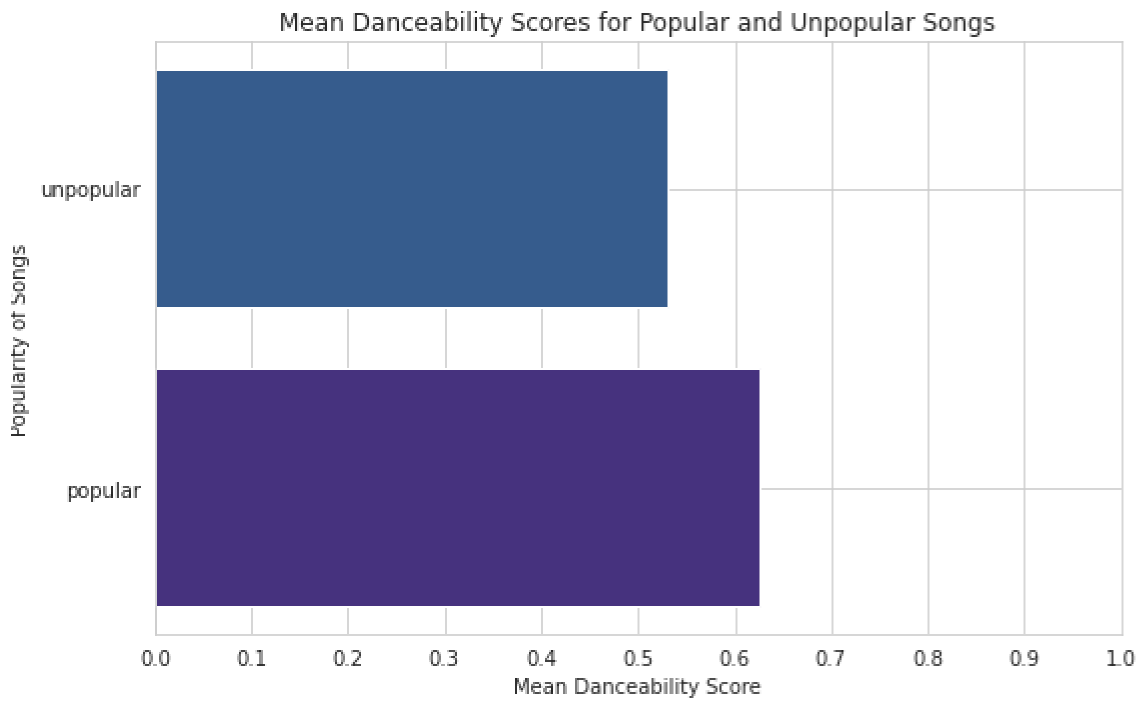
In [94]:

```
#removing outliers from danceability scores and separating them to Series fc
popular_dance_clean = popular_songs_df[find_outliers_IQR(popular_songs_df['c
print(popular_dance_clean['danceability'].describe())

unpopular_dance_clean = unpopular_songs_df[find_outliers_IQR(unpopular_songs
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 [95]:

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

#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('viridi
    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 [96]:

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

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

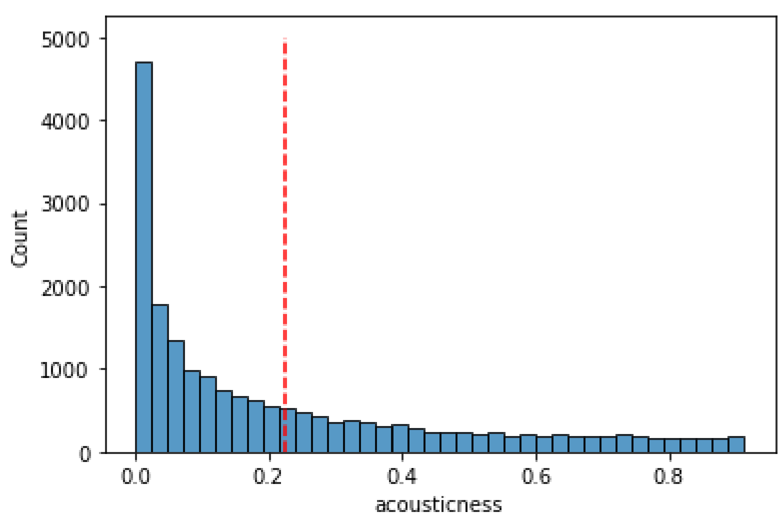
```
count    19715.000000
mean         0.226220
std          0.248585
min          0.000002
25%          0.026400
50%          0.125000
75%          0.355000
max          0.913000
Name: acousticness, dtype: float64
count    156575.000000
mean          0.424829
std           0.371949
min           0.000000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

In [97]:

```
sns.histplot(data = popular_acoustic_clean, x='acousticness', bins='auto')
plt.vlines(x=popular_acoustic_clean['acousticness'].mean(), ymin=0, ymax=50€
```

Out[97]:

```
<matplotlib.collections.LineCollection at 0x1ddd82e54c0>
```

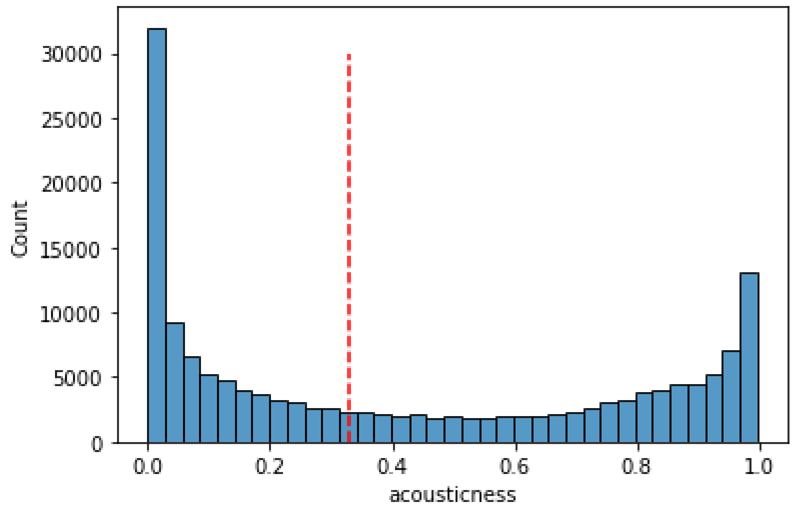


In [98]:

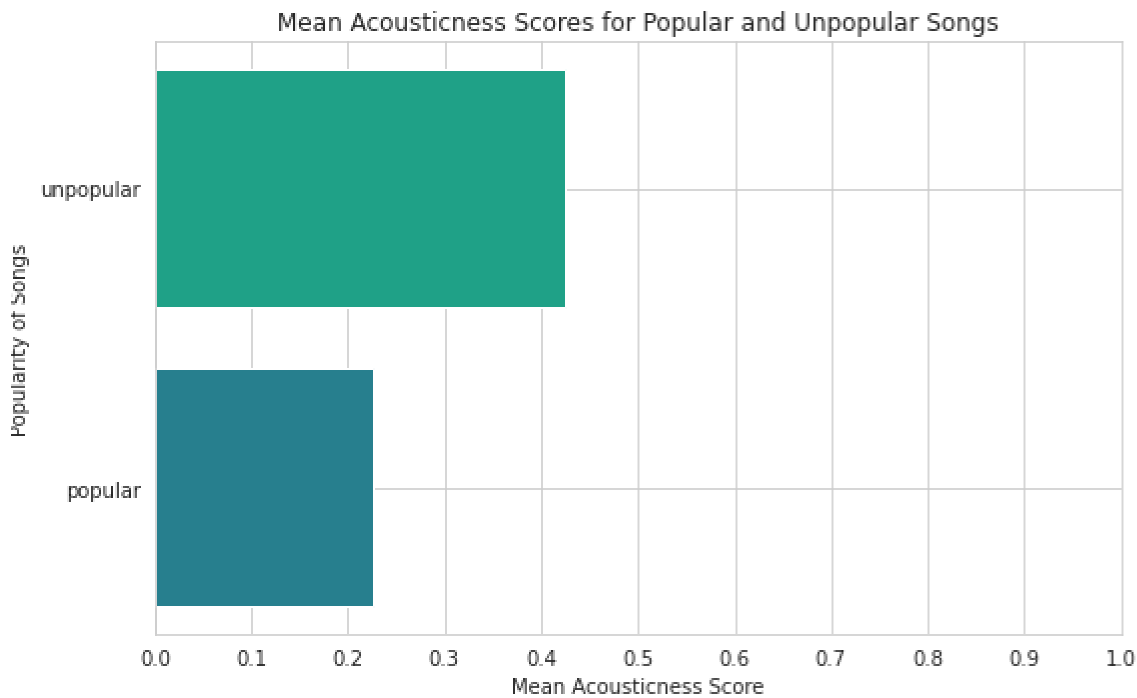
```
sns.histplot(data = unpopular_songs_df, x='acousticness', bins='auto')
plt.vlines(x=unpopular_songs_df['acousticness'].median(), ymin=0, ymax=3000(
```

Out[98]:

```
<matplotlib.collections.LineCollection at 0x1ddd82e5430>
```



In [99]:
```
#storing mean acousticness scores in dict
mean_acousticness = {'popular': popular_acoustic_clean['acousticness'].mean(
                     'unpopular': unpopular_acoustic_clean['acousticness'].m

#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('viridi
    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

Best Model Results

In [106]:

```
df_results.sort_values(by='Recall Score', ascending=False)
```

Out[106]:

	Model Name	Recall Score
3	Logistic Regression	0.66
2	XGBoost	0.65
1	Random Forest	0.60
0	Dummy Classifier	0.51

Out of the 3 models, the Logistic Regression model was the best one in identifying popular songs. It had a 66% recall score for identifying popular songs compared to 51% by the baseline Dummy Classifier model. The closest rival was the XGBoost model at 65%. Even though XGBoost fell short in this regard, we consider the XGBoost as the best overall model since it had 4% higher recall score in identifying unpopular songs compared to the Logistic Regression model. Since this project was focused on identifying popular songs, Logistic Regression wins.

Takeaways

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.

Recommendations

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.