C964 -- Music Popularity Predictor

Setting Up Environment and Importing All Required Libraries

First we import all necessary Python libraries. Don't worry if you don't have one or more libraries installed on your machine. Just install them by typing in the empty cell below:

!pip install "name of the library"

(Don't forget the exclamation mark and type the name of the library without the quotation marks)

In [1]:
```
# empty cell to use !pip install to install libraries in case not present on machine
```

In [2]:
```
# for data manipulations
import numpy as np
import pandas as pd

# for data visualizations
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
plt.style.use('fivethirtyeight')

# for interactivity
import ipywidgets.widgets as widgets
from ipywidgets import interact, interact_manual

import os

small_figsize = (14, 7)
large_figsize = (18, 18)
plt.rcParams['figure.figsize'] = small_figsize

from warnings import filterwarnings as warnings

# import gzip, pickletools
import pickle
# from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import *
from sklearn.metrics import mean_absolute_error, confusion_matrix, accuracy_score, classification_report, balanced_accuracy_score
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier, AdaBoostClassifier, GradientBoostingClassifier
from sklearn.model_selection import GridSearchCV, cross_val_score, train_test_split
```

In [3]:
```
'''Later on, we'll filter the dataframe for songs after a cerain threshold, in this case 2015.
For maintenance purposes we can set the constant here, in case we want to change the threshold
in later iterations of the project.'''
YEAR_THRESHOLD = 2015
```

Next up, loading our dataset using pandas

In [4]:
```
# Load the track data using pandas
# I was using this code when the csv-files were present locally on my machine.
# The next cell will load the same datasets directly from GitHub.
'''track_data = pd.read_csv('../Data Sources/spotify_tracks.csv')
artist_data = pd.read_csv('../Data Sources/spotify_artists.csv')
album_data = pd.read_csv('../Data Sources/spotify_albums.csv')'''
```

Out[4]:
```
"track_data = pd.read_csv('../Data Sources/spotify_tracks.csv')\nartist_data = pd.read_csv('../Data Sources/spotify_artists.csv')\nalbum_data
= pd.read_csv('../Data Sources/spotify_albums.csv')"
```

In [5]:
```
# Load the track data using pandas from a URL
# URL for tracks: https://media.githubusercontent.com/media/jonivanrossum/C964-Data-Sources/main/spotify_tracks.csv
# URL for artist: https://media.githubusercontent.com/media/jonivanrossum/C964-Data-Sources/main/spotify_artists.csv
# URL for albums: https://media.githubusercontent.com/media/jonivanrossum/C964-Data-Sources/main/spotify_albums.csv

track_data = pd.read_csv('https://media.githubusercontent.com/media/jonivanrossum/C964-Data-Sources/main/spotify_tracks.csv')
artist_data = pd.read_csv('https://media.githubusercontent.com/media/jonivanrossum/C964-Data-Sources/main/spotify_artists.csv')
album_data = pd.read_csv('https://media.githubusercontent.com/media/jonivanrossum/C964-Data-Sources/main/spotify_albums.csv')
```

Always check if we have actually loaded the dataset, either by displaying a sample from the dataset (see below) or by typing the name of the datafram followed by .head()

In [6]:
```
# Exploring track data by getting a sample from the dataset
track_data.sample(2)
```

Out[6]:

	Unnamed: 0	acousticness	album_id	analysis_url	artists_id	available_markets	country	danceability	disc_numl
78387	78387	0.698	7eECqp8MX57lOvt5Me1eXM	https://api.spotify.com/v1/audio-analysis/6Abl...	['3hvpB2JNbOGd2NTjdaDMGl']	['AD', 'AE', 'AR', 'AT', 'AU', 'BE', 'BG', 'BH...	AR	0.591	
58078	58078	0.115	6inQEkewkcaA8naYltd1sz	https://api.spotify.com/v1/audio-analysis/0vK5...	['634CNOtyAzRpXriVYTms8A']	['AD', 'AE', 'AR', 'AT', 'AU', 'BE', 'BG', 'BH...	FI	0.762	

2 rows × 32 columns

In [7]:
```
# Exploring artist data by getting a sample from the dataset
artist_data.sample(2)
```

Out[7]:

	Unnamed: 0	artist_popularity	followers	genres	id	name	track_id	track_name_prev	type
37436	37436	16	448	[]	5gSq53ejD3vr07CsruwvRa	Sutherland	4KqCtzseDZlu4iaTFyu12W	track_35	artist
46990	46990	52	9853	['norwegian pop', 'norwegian pop rap', 'swedis...	46XcyK8FnyCJJlvYCUwVZH	Hkeem	7BjX8clU1Ss1SD48St76gE	track_11	artist

In [8]:

```python
# Exploring album data by getting a sample from the dataset
album_data.sample(2)
```

Out[8]:

	Unnamed: 0	album_type	artist_id	available_markets	external_urls	href	
65145	65145	album	60Ygpctc1ZdaRuvvrkiqAU	['AD', 'AE', 'AR', 'AT', 'AU', 'BE', 'BG', 'BH...	{'spotify': 'https://open.spotify.com/album/0N...	https://api.spotify.com/v1/albums/0NprVMRsU2cJ...	0NprVMRsU2
67068	67068	album	0jnsk9HBra6NMjO2oANoPY	['AD', 'AE', 'AR', 'AT', 'BE', 'BG', 'BH', 'BO...	{'spotify': 'https://open.spotify.com/album/2v...	https://api.spotify.com/v1/albums/2vBLKFrl1rZq...	2vBLKFrl1r

All three datasets are loaded successfully! Let's dive in a little deeper and decide on how we are going to prepare the dataset for modeling. Which attributes are important to make a prediction for the target variable (popularity rating)? Which aren't important and can be dropped from the dataset?

Examining and Preparing the Dataset

Let's learn about the data columns and rows present. We'll do this by first checking the column names and shape of each dataframe. Then we'll join all 3 dataframes together in one dataframe. We'll check the data description and data types present next. After that, we'll keep the columns that interest us and we'll drop the ones that do not. Finally, we will check the Target Class Balance.

In [9]:

```python
# check for what columns we have in all dataframes and select which ones we want to use for prediction
print("Columns present in track_data:", track_data.columns, "\n\nShape of track_data:", track_data.shape, "\n")
print("Columns present in artist_data:", artist_data.columns, "\n\nShape of artist_data:", artist_data.shape, "\n")
print("Columns present in album_data:", album_data.columns, "\n\nShape of album_data:", album_data.shape, "\n")
```

```
Columns present in track_data: Index(['Unnamed: 0', 'acousticness', 'album_id', 'analysis_url', 'artists_id',
       'available_markets', 'country', 'danceability', 'disc_number',
       'duration_ms', 'energy', 'href', 'id', 'instrumentalness', 'key',
       'liveness', 'loudness', 'lyrics', 'mode', 'name', 'playlist',
       'popularity', 'preview_url', 'speechiness', 'tempo', 'time_signature',
       'track_href', 'track_name_prev', 'track_number', 'uri', 'valence',
       'type'],
      dtype='object')

Shape of track_data: (101939, 32)

Columns present in artist_data: Index(['Unnamed: 0', 'artist_popularity', 'followers', 'genres', 'id', 'name',
       'track_id', 'track_name_prev', 'type'],
      dtype='object')

Shape of artist_data: (56129, 9)

Columns present in album_data: Index(['Unnamed: 0', 'album_type', 'artist_id', 'available_markets',
       'external_urls', 'href', 'id', 'images', 'name', 'release_date',
       'release_date_precision', 'total_tracks', 'track_id', 'track_name_prev',
       'uri', 'type'],
      dtype='object')

Shape of album_data: (75511, 16)
```

In [10]:

```python
# let's join track data and artist data, but all we need from the artist dataset is 'genres'
song_data = pd.merge(track_data, artist_data[['track_id', 'genres']], left_on='id', right_on='track_id', how='inner')

# drop 'track_id' column, because we don't need it anymore.
song_data.drop('track_id', axis=1, inplace=True)

# check song_data head
song_data.head(3)
```

Out[10]:

	Unnamed: 0	acousticness	album_id	analysis_url	artists_id	available_markets	country	danceability	disc_number
0	1	0.863	1bcqsH5UyTBzmh9YizdsBE	https://api.spotify.com/v1/audio-analysis/3VAX...	['4xWMewm6CYMstu0sPgd9jJ']	['AD', 'AE', 'AR', 'AT', 'AU', 'BE', 'BG', 'BH...	BE	0.719	1.0
1	3	0.763	6FeJF5r8roonnKraJxr4oB	https://api.spotify.com/v1/audio-analysis/6aCe...	['2KQsUB9DRBcJk17JWX1eXD']	['AD', 'AE', 'AR', 'AT', 'AU', 'BE', 'BG', 'BH...	BE	0.719	1.0
2	10	0.101	7noNViHJAYZ3UxlhDNKAt9	https://api.spotify.com/v1/audio-analysis/01zM...	['3FLUBwpAnalllKeaBfsxFe', '5r5Va4lVQ1zjEfbJSr...	['AD', 'AE', 'AR', 'AT', 'AU', 'BE', 'BG', 'BH...	BE	0.748	1.0

3 rows × 33 columns

In [11]:

```python
# now we'll merge song_data with the album data release date
song_data = pd.merge(song_data, album_data[['track_id', 'release_date']], left_on='id', right_on='track_id', how='inner')

# drop 'track_id' column again, because we don't need it anymore.
song_data.drop('track_id', axis=1, inplace=True)
```

```python
# check song_data head
song_data.head()
```

Out[11]:

	Unnamed: 0	acousticness	album_id	analysis_url	artists_id	available_markets	country	danceability	disc_number
0	1	0.8630	1bcqsH5UyTBzmh9YizdsBE	https://api.spotify.com/v1/audio-analysis/3VAX...	['4xWMewm6CYMstu0sPgd9jJ']	['AD', 'AE', 'AR', 'AT', 'AU', 'BE', 'BG', 'BH...	BE	0.719	1.0
1	10	0.1010	7noNViHJAYZ3UxlhDNKAt9	https://api.spotify.com/v1/audio-analysis/01zM...	['3FLUBwpAnallIKeaBfsxFe', '5r5Va4lVQ1zjEfbJSr...	['AD', 'AE', 'AR', 'AT', 'AU', 'BE', 'BG', 'BH...	BE	0.748	1.0
2	25	0.1910	6cflCkqI3e9MHkm7rZlkXA	https://api.spotify.com/v1/audio-analysis/2Dh5...	['06lig2bqY8mv98B1c9lyo8']	['AD', 'AT', 'BE', 'BG', 'CH', 'CY', 'CZ', 'DE...	BE	0.608	1.0
3	28	0.6780	6VVr09AK8qjO6doYUEzrVj	https://api.spotify.com/v1/audio-analysis/2hX9...	['7mdXCgprfvNzxRQsjuUwy8']	['AD', 'AE', 'AR', 'AT', 'AU', 'BE', 'BG', 'BH...	BE	0.679	1.0
4	35	0.0786	6HliYi1SE9uMcnJHFVC0oT	https://api.spotify.com/v1/audio-analysis/58QD...	['04XdCDDrPnnqidaVBTOQjt']	['AD', 'AE', 'AR', 'AT', 'AU', 'BE', 'BG', 'BH...	BE	0.470	1.0

5 rows × 34 columns

```python
# let's check what columns are present now in the dataframe
song_data.columns
```

Out[12]:

```
Index(['Unnamed: 0', 'acousticness', 'album_id', 'analysis_url', 'artists_id',
       'available_markets', 'country', 'danceability', 'disc_number',
       'duration_ms', 'energy', 'href', 'id', 'instrumentalness', 'key',
       'liveness', 'loudness', 'lyrics', 'mode', 'name', 'playlist',
       'popularity', 'preview_url', 'speechiness', 'tempo', 'time_signature',
       'track_href', 'track_name_prev', 'track_number', 'uri', 'valence',
       'type', 'genres', 'release_date'],
      dtype='object')
```

Merging the data was a succes!

Let's get a description of the data and the data types:

```python
song_data.describe()
```

Out[13]:

	Unnamed: 0	acousticness	danceability	disc_number	duration_ms	energy	instrumentalness	key	liveness	loudness	moc
count	53247.000000	53247.000000	53247.000000	53247.000000	5.324700e+04	53247.000000	53247.000000	53247.000000	53247.000000	53247.000000	53247.00000
mean	56500.671099	0.352891	0.575549	1.019419	2.551591e+05	0.584284	0.193634	5.267189	0.182994	-9.685067	0.5923
std	29623.267223	0.350893	0.192521	0.234496	1.876593e+05	0.269966	0.332821	3.560689	0.155679	6.463120	0.49140
min	1.000000	0.000000	0.000000	1.000000	4.000000e+03	0.000000	0.000000	0.000000	0.000000	-57.436000	0.00000
25%	31838.000000	0.032500	0.454000	1.000000	1.868595e+05	0.408000	0.000000	2.000000	0.093400	-11.311000	0.00000
50%	59157.000000	0.213000	0.603000	1.000000	2.193330e+05	0.636000	0.000237	5.000000	0.119000	-7.680000	1.00000
75%	83162.000000	0.670000	0.723000	1.000000	2.700490e+05	0.802000	0.233000	8.000000	0.219000	-5.528000	1.00000
max	101937.000000	0.996000	0.984000	14.000000	4.811520e+06	1.000000	1.000000	11.000000	0.999000	1.605000	1.00000

Above are all the columns that contain numerical values (integers and floats). Let's see if there are columns that hold objects:

```python
song_data.describe(include='object')
```

Out[14]:

	album_id	analysis_url	artists_id	available_markets	country	href	
count	53247	53247	53247	53247	53247	53247	
unique	42549	42549	42549	1647	3	42549	
top	4w1gNoYG8Ziudf4gQ0L57V	https://api.spotify.com/v1/audio-analysis/3rPs...	['7ehgiWrLMRMQkrr5tQQB2P', '3wzYUp9ga2NGxFLQyW...	['AD', 'AE', 'AR', 'AT', 'AU', 'BE', 'BG', 'BH...	AR	https://api.spotify.com/v1/tracks/3rPsrmdQkua1...	3rPsr
freq	21	21	21	39124	30562	21	

```python
song_data.dtypes
```

Out[15]:
```
Unnamed: 0            int64
acousticness         float64
album_id              object
analysis_url          object
artists_id            object
available_markets     object
country               object
danceability         float64
disc_number          float64
duration_ms          float64
energy               float64
href                  object
id                    object
instrumentalness     float64
key                  float64
liveness             float64
loudness             float64
lyrics                object
mode                 float64
name                  object
playlist              object
popularity           float64
preview_url           object
speechiness          float64
tempo                float64
time_signature       float64
track_href            object
track_name_prev       object
track_number         float64
uri                   object
valence              float64
type                  object
genres                object
release_date          object
dtype: object
```

Fortunately, we don't need any of the above columns for our predictor, so let's drop them and the numerical columns we don't need as well.

In [16]:
```python
# Let's drop columns we don't need for our predictor
data = song_data.drop(['Unnamed: 0',
                       'album_id',
                       'analysis_url',
                       'artists_id',
                       'available_markets',
                       'country',
                       'disc_number',
                       'href',
                       'id',
                       'lyrics',
                       'name',
                       'playlist',
                       'track_href',
                       'preview_url',
                       'track_href',
                       'track_name_prev',
                       'track_number',
                       'uri',
                       'type',], axis=1)

data.columns
```

Out[16]:
```
Index(['acousticness', 'danceability', 'duration_ms', 'energy',
       'instrumentalness', 'key', 'liveness', 'loudness', 'mode', 'popularity',
       'speechiness', 'tempo', 'time_signature', 'valence', 'genres',
       'release_date'],
      dtype='object')
```

Let's do some Feature Engineering

As can be seen above, we have a column called release_date. Because music trends can definitely change every year, let's extract the year from the release data and put the data in its own column, called "release_year".

In [17]:
```python
# create a new column 'release_year' and drop 'release_date'
data['release_date'] = pd.to_datetime(data['release_date'])
data['release_year'] = data.release_date.dt.year
data.drop('release_date', axis=1, inplace=True)
data.head(2)
```

Out[17]:

	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	popularity	speechiness	tempo	time_signature	valence	genres
0	0.863	0.719	656960.0	0.308	0.000000	6.0	0.2530	-10.340	1.0	31.0	0.9220	115.075	3.0	0.589	[]
1	0.101	0.748	237667.0	0.666	0.000653	6.0	0.0976	-6.094	0.0	47.0	0.0833	114.982	4.0	0.359	['electra']

Interactively Exploring the Data

Let's use ipywidgets and play around with the dataset to get some insights and determine next steps.

In [18]:
```python
@interact
def viz(song_attribute=list(data.select_dtypes('number').columns[:-1])):
    sns.lineplot(x=data['release_year'], y=data[song_attribute])
    plt.title("Trend of selected attribute over the years")
    plt.show()
```

```
interactive(children=(Dropdown(description='song_attribute', options=('acousticness', 'danceability', 'duratio…
```

Playing around with the interactive function above, we can see that, for example, acousticness is at almost 1.0 in the 1920's and almost 0 in 2019. Similarly, energy is low in the 1920's and about 0.65 in 2019. Intuitively this makes sens; songs in the 1920's sound a lot different than songs from the past few years.

Concluding, Song Popularity is very timeperiod sensitive and follows trends. What made a song popular in the 1920's is not always what makes a song popular now. So let's work with a dataset that only has songs starting from 2015. Five years of songs intuitively seems a good number (the dataset only has songs up until 2019). So let's not forget to set the YEAR_THRESHOLD constant to 2015 in the 3rd cell of code in this notebook.

In [19]:
```
# data shape before dropping rows
x = list(data.shape)
print("Data shape before dropping rows", x)

# Select songs from year_treshold and after
df = data[data['release_year'] >= YEAR_THRESHOLD]

# data shape after dropping rows
y = list(df.shape)
print("Data shape after dropping rows", y)

def rows_dropped_report(x, y):
    z = x[0] - y[0]
    return z

print(f"A total of {rows_dropped_report(x, y)} rows were dropped.")
```

```
Data shape before dropping rows [53247, 16]
Data shape after dropping rows [36575, 16]
A total of 16672 rows were dropped.
```

Further Exploring the Data

In [20]:
```
# let's check what datatypes are present in the dataset
df.dtypes
```

Out[20]:
```
acousticness        float64
danceability        float64
duration_ms         float64
energy              float64
instrumentalness    float64
key                 float64
liveness            float64
loudness            float64
mode                float64
popularity          float64
speechiness         float64
tempo               float64
time_signature      float64
valence             float64
genres               object
release_year          int64
dtype: object
```

In [21]:
```
# let's explore what the genres column holds
df['genres'].value_counts()
```

Out[21]:
```
[]                                                                              16764
['focus']                                                                         171
['chillhop', 'lo-fi beats']                                                        95
['lo-fi beats']                                                                    68
['dutch hip hop']                                                                  67
                                                                                  ...
['bass music', 'chillwave', 'grave wave', 'witch house', 'wonky']                   1
['art pop', 'chamber psych', 'escape room', 'fluxwork']                             1
['no wave', 'norwegian indie', 'norwegian pop', 'norwegian pop rap']                1
['boston hardcore', 'chaotic hardcore', 'grindcore', 'mathcore', 'post-doom metal']  1
['atl hip hop', 'hip hop', 'pop', 'pop rap', 'rap', 'southern hip hop', 'trap music']  1
Name: genres, Length: 8051, dtype: int64
```

In our 36000+ records, more than 16000 have an empty list as a genre. This is almost half of the dataset, so filling the empty lists using mode() does not make sense. So let's drop the genre column altogether.

In [22]:
```
df = df.drop('genres', axis=1)

# check if genres column was indeed dropped
df.columns
```

Out[22]:
```
Index(['acousticness', 'danceability', 'duration_ms', 'energy',
       'instrumentalness', 'key', 'liveness', 'loudness', 'mode', 'popularity',
       'speechiness', 'tempo', 'time_signature', 'valence', 'release_year'],
      dtype='object')
```

Genres was dropped indeed.

Let's see if there are songs that have 0 popularity. After all, we are trying to see what makes a song popular, not necessarily what makes it unpopular. This will cut out some noise too.

In [23]:
```
# let's see if there are songs that have 0 popularity
df[df['popularity'] == 0]
```

Out[23]:

	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	popularity	speechiness	tempo	time_signature	valence	rele
598	0.059500	0.000	6500.0	0.477	0.000000	3.0	0.0000	-12.187	1.0	0.0	0.0000	0.000	0.0	0.0000	
937	0.677000	0.000	4750.0	0.366	0.122000	10.0	0.0000	-14.910	0.0	0.0	0.0000	0.000	0.0	0.0000	
1446	0.821000	0.000	6250.0	0.226	0.000000	3.0	0.0000	-9.586	0.0	0.0	0.0000	0.000	0.0	0.0000	
1859	0.186000	0.421	274644.0	0.692	0.000000	2.0	0.2940	-5.721	0.0	0.0	0.0869	116.480	4.0	0.4960	
1860	0.186000	0.421	274644.0	0.692	0.000000	2.0	0.2940	-5.721	0.0	0.0	0.0869	116.480	4.0	0.4960	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
51341	0.000337	0.833	266148.0	0.698	0.346000	10.0	0.1360	-5.266	0.0	0.0	0.0520	122.980	4.0	0.2360	
51968	0.227000	0.404	233860.0	0.427	0.856000	2.0	0.0907	-12.578	1.0	0.0	0.0683	162.668	4.0	0.0388	
52164	0.021100	0.715	254733.0	0.970	0.000019	0.0	0.3350	-2.270	0.0	0.0	0.0467	108.005	4.0	0.8020	
52173	0.007200	0.720	222587.0	0.878	0.000001	7.0	0.2810	-4.930	0.0	0.0	0.1680	125.990	4.0	0.6210	
52506	0.007510	0.382	161613.0	0.931	0.000105	7.0	0.2360	-5.952	1.0	0.0	0.0652	90.618	4.0	0.3580	

140 rows × 15 columns

That query returned 140 rows only. Let's delete them and then categorize the remaining rows by popularity rating.

In [24]:

```
# data shape before dropping rows
x = list(df.shape)
print("Data shape before dropping rows", x)

# Drop rows with 0 popularity
df = df[df['popularity'] > 0]

# data shape after dropping rows
y = list(df.shape)
print("Data shape after dropping rows", y)

print(f"A total of {rows_dropped_report(x, y)} rows were dropped.")
```

```
Data shape before dropping rows [36575, 15]
Data shape after dropping rows [36435, 15]
A total of 140 rows were dropped.
```

Let's split the popularity observations into 3 classes:

- Unpopular
- Popular
- Very Popular

where 50 popularity rating is the magic number/threshold for a song to be considered popular. A rating of 80 and more would make a song very popular, so we'll create a seperate category for that as well.

In [25]:

```
df['pop_rating'] = ''
```

In [26]:

```
pop_ratings = ['Unpopular', 'Popular', 'Very Popular']

for i, row in df.iterrows():
    score = pop_ratings[0]
    if (row.popularity >= 80):
        score = pop_ratings[2]
    elif (row.popularity >= 50 and row.popularity < 80):
        score = pop_ratings[1]
    df.at[i, 'pop_rating'] = score

df[['popularity', 'pop_rating']].head()
```

Out[26]:

	popularity	pop_rating
1	47.0	Unpopular
2	35.0	Unpopular
4	55.0	Popular
6	41.0	Unpopular
7	49.0	Unpopular

In [27]:

```
df.pop_rating.describe()
```

Out[27]:

```
count          36435
unique             3
top        Unpopular
freq           27969
Name: pop_rating, dtype: object
```

Categorizing popularity into 3 categories was a success.

Let's have a look at how song attributes relate to a song's popularity over the years.

In [28]:

```
song_attribute=list(df.select_dtypes('number').columns[:-1])
song_attribute.remove('popularity')

@interact
def viz2(song_attribute=song_attribute):
    sns.barplot(x=df['release_year'], y=df[song_attribute], hue = df['pop_rating'], palette='Reds')
    plt.title("Song Attribute vs. Popularity")
    plt.show()
```

```
interactive(children=(Dropdown(description='song_attribute', options=('acousticness', 'danceability', 'duratio…
```

Using the interactive function below, it is clear that:

- Very popular songs have higher valence compared to the other songs (valence indicates the level of positivity)
- Very popular songs are louder than others.
- Popular songs overall are have a lower instrumentalness than unpopular songs.
- Very popular songs sound more energetic than unpopular songs (except for 2016)
- A shorter song duration seems to indicate higher popularity.

Data Cleaning and Visualization

Let's check for null values first:

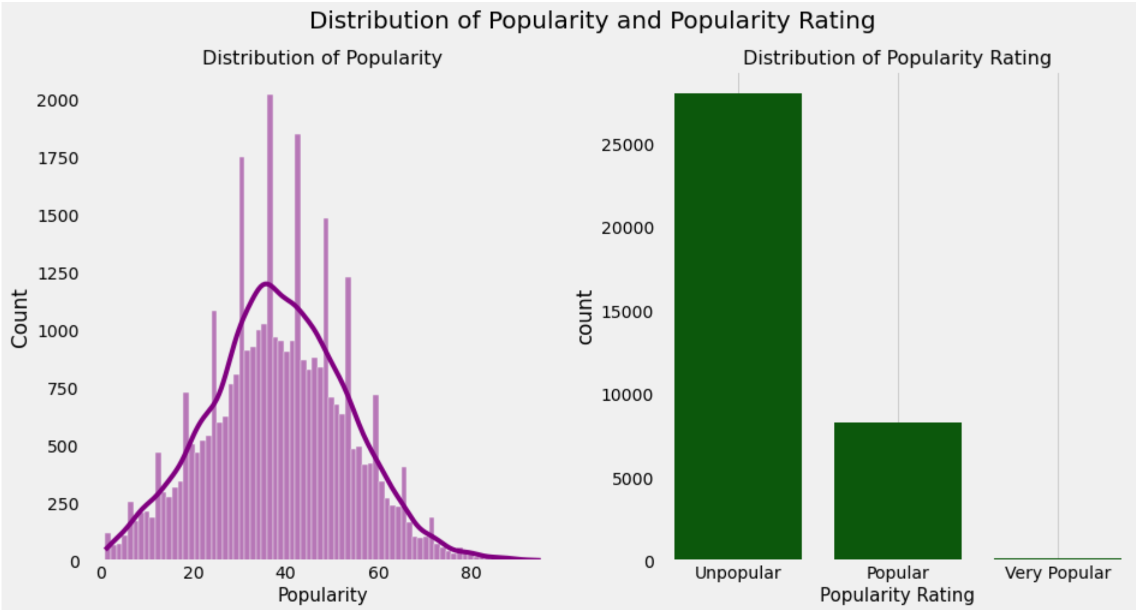
In [29]:
```python
print(df.isnull().sum(), "\n")
print(df.isna().sum())
```

```
acousticness        0
danceability        0
duration_ms         0
energy              0
instrumentalness    0
key                 0
liveness            0
loudness            0
mode                0
popularity          0
speechiness         0
tempo               0
time_signature      0
valence             0
release_year        0
pop_rating          0
dtype: int64

acousticness        0
danceability        0
duration_ms         0
energy              0
instrumentalness    0
key                 0
liveness            0
loudness            0
mode                0
popularity          0
speechiness         0
tempo               0
time_signature      0
valence             0
release_year        0
pop_rating          0
dtype: int64
```

There are no missing values in the dataset.

Since "pop_rating" is the target column, let's check the distribution:

In [30]:
```python
df['pop_rating'] = pd.Categorical(df.pop_rating, pop_ratings)

plt.rcParams['figure.figsize'] = small_figsize

plt.subplot(1, 2, 1)
sns.histplot(x=df.popularity, color = 'purple', kde=True)
plt.xlabel('Popularity', fontsize = 15)
plt.title("Distribution of Popularity", fontsize=16)
plt.grid()

plt.subplot(1, 2, 2)
sns.countplot(x=df.pop_rating, color = 'darkgreen')
plt.xlabel('Popularity Rating', fontsize = 15)
plt.title("Distribution of Popularity Rating", fontsize=16)
plt.grid()

plt.suptitle("Distribution of Popularity and Popularity Rating", fontsize=20)
plt.show()
```



Conclusion: The target column is imbalanced; we will need to use SMOTE to balance out the samples to improve learning.

Descriptive Statistics

In [31]:

```
print("Average Ratio of Acousticness in the Dataset: {0:.2f}".format(df['acousticness'].mean()))
print("Average Ratio of Danceability in the Dataset: {0:.2f}".format(df['danceability'].mean()))
minutes = int((df['duration_ms'].mean()//1000)//60)
seconds = int((df['duration_ms'].mean()//1000)%60)
print("Average Duration of Songs:", minutes, "minutes and", seconds, "seconds")
print("Average Ratio of Energy in the Dataset: {0:.2f}".format(df['energy'].mean()))
print("Average Ratio of Instrumentalness in the Dataset: {0:.2f}".format(df['instrumentalness'].mean()))
print("Average Ratio of Liveness in the Dataset: {0:.2f}".format(df['liveness'].mean()))
print("Average Ratio of Loudness in the Dataset: {0:.2f}".format(df['loudness'].mean()))
print("Average Ratio of Speechiness in the Dataset: {0:.2f}".format(df['speechiness'].mean()))
print("Average Tempo in the Dataset: {0:.2f}".format(df['tempo'].mean()), "bpm")
print("Average Ratio of Valence in the Dataset: {0:.2f}".format(df['valence'].mean()))
print("Average Popularity in the Dataset: {0:.2f}".format(df['popularity'].mean()))
```

```
Average Ratio of Acousticness in the Dataset: 0.29
Average Ratio of Danceability in the Dataset: 0.60
Average Duration of Songs: 3 minutes and 53 seconds
Average Ratio of Energy in the Dataset: 0.62
Average Ratio of Instrumentalness in the Dataset: 0.18
Average Ratio of Liveness in the Dataset: 0.18
Average Ratio of Loudness in the Dataset: -8.42
Average Ratio of Speechiness in the Dataset: 0.10
Average Tempo in the Dataset: 120.22 bpm
Average Ratio of Valence in the Dataset: 0.46
Average Popularity in the Dataset: 38.55
```

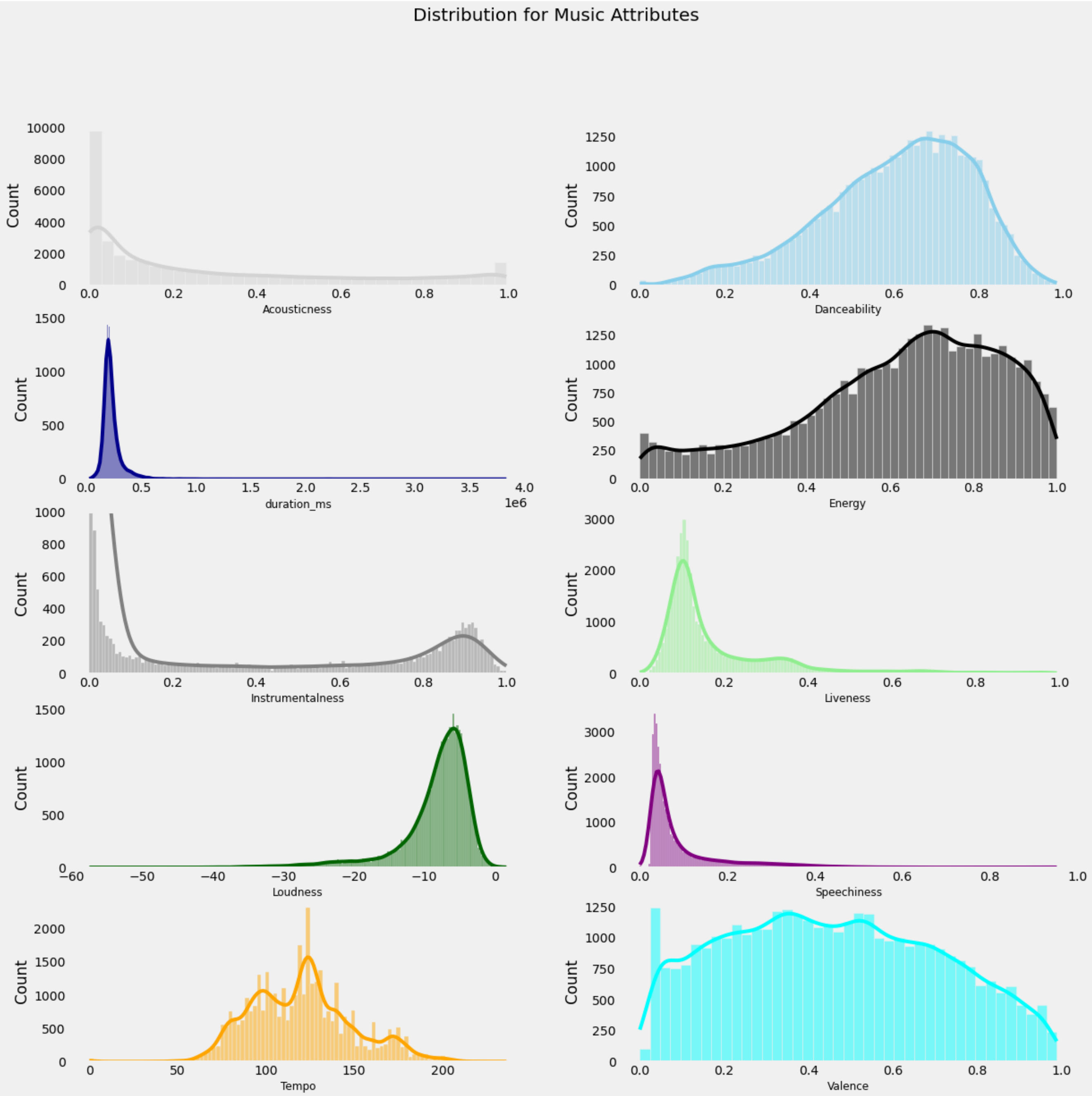
Univariate Analysis

Let's check distributions of all clumns in the dataset . (Descriptive method example)

In [32]:

```
# check distribution of different categories in the dataset

plt.rcParams['figure.figsize'] = large_figsize

plt.subplot(5, 2, 1)
sns.histplot(df['acousticness'], color = 'lightgrey', kde=True)
plt.xlabel('Acousticness', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 2)
sns.histplot(df['danceability'], color = 'skyblue', kde=True)
plt.xlabel('Danceability', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 3)
sns.histplot(df['duration_ms'], color ='darkblue', kde=True)
plt.xlabel('duration_ms', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 4)
sns.histplot(df['energy'], color = 'black', kde=True)
plt.xlabel('Energy', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 5)
sns.histplot(df['instrumentalness'], color = 'grey', kde=True)
plt.xlabel('Instrumentalness', fontsize = 12)
plt.ylim(0, 1000)
plt.grid()

plt.subplot(5, 2, 6)
```

```
sns.histplot(df['liveness'], color = 'lightgreen', kde=True)
plt.xlabel('Liveness', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 7)
sns.histplot(df['loudness'], color = 'darkgreen', kde=True)
plt.xlabel('Loudness', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 8)
sns.histplot(df['speechiness'], color = 'purple', kde=True)
plt.xlabel('Speechiness', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 9)
sns.histplot(df['tempo'], color = 'orange', kde=True)
plt.xlabel('Tempo', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 10)
sns.histplot(df['valence'], color = 'cyan', kde=True)
plt.xlabel('Valence', fontsize = 12)
plt.grid()

plt.suptitle('Distribution for Music Attributes', fontsize = 20)
plt.show()
```

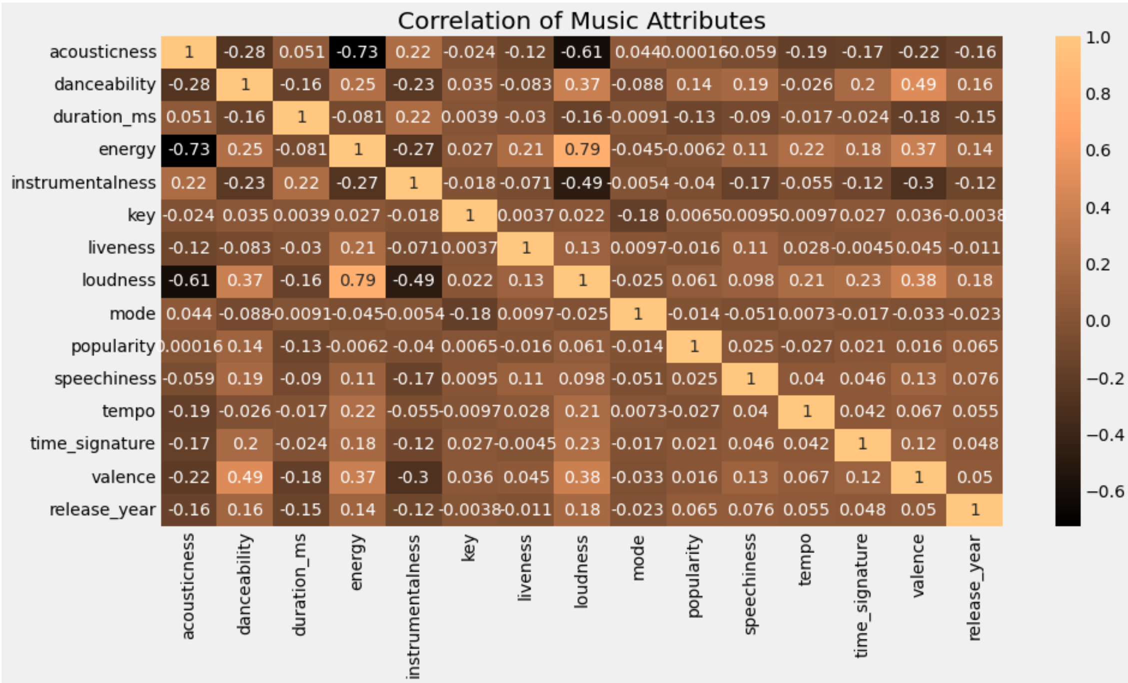


When the model turns out to underperform, we can come back to this and try performing transformations on the columns to fix skewed distributions, such as 'speechiness', 'loudness', 'liveness', etc.

Bivariate Analysis

When checking for correlation, a heatmap is useful:

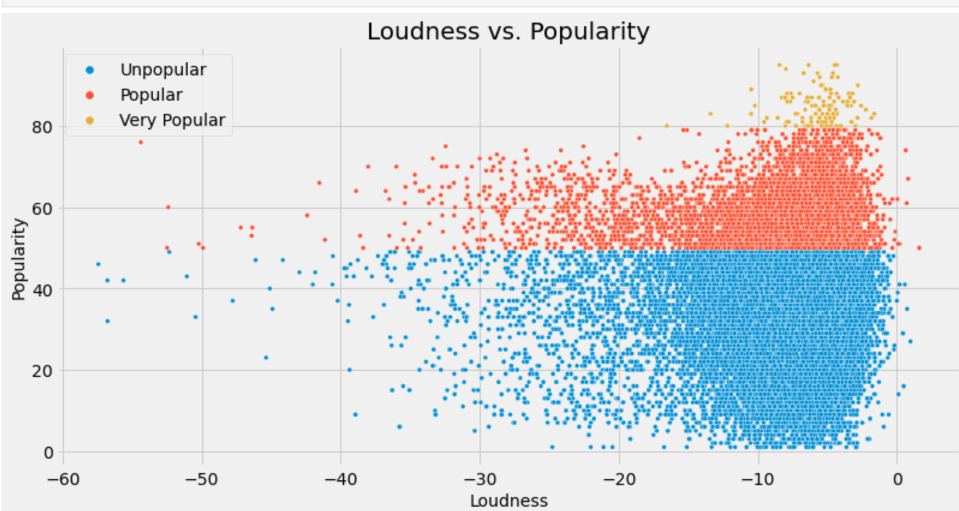
In [33]:
```
plt.rcParams['figure.figsize'] = small_figsize
sns.heatmap(df.corr(), annot=True, cmap='copper')
plt.title("Correlation of Music Attributes")
plt.show()
```



No attribute strongly relates to Popularity, but loudness seems to correlate to energy, and there is some correlation between ergy, danceability, and valence as well.

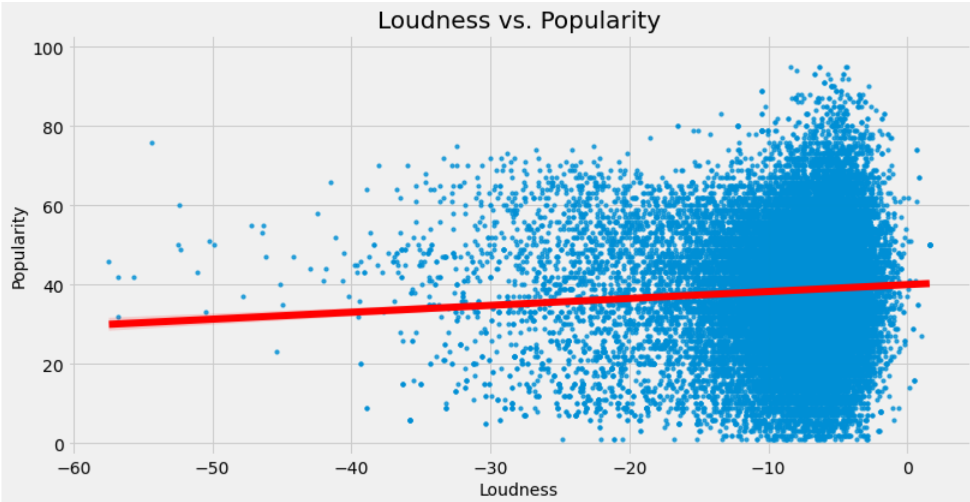
In [34]:
```
# Define a function to help plot a scatterplot using given inputs
def scat_plot(x, y, hue=None, xlabel='', ylabel='', title=''):
    fig, ax = plt.subplots(figsize=(12, 6))
    _ = sns.scatterplot(x=x, y=y, hue=hue, s=14)
    _ = plt.xlabel(xlabel, fontsize=14)
    _ = plt.ylabel(ylabel, fontsize=14)
    _ = plt.title(title, fontsize=20)
    _ = plt.legend(fontsize=14)
    plt.show()

# Define a function to help plot a scatterplot with a regression line using given inputs
def regress_plot(x='', y='', data=None, xlabel='', ylabel='', title=''):
    fig, ax = plt.subplots(figsize=(12, 6))
    _ = sns.regplot(x=x, y=y, data=data, scatter_kws={"s": 10}, line_kws={'color':'r'})
    _ = plt.xlabel(xlabel, fontsize=14)
    _ = plt.ylabel(ylabel, fontsize=14)
    _ = plt.title(title, fontsize=20)
    _ = plt.ylim(-3, 103)
    plt.show()
```

In [35]:
```
scat_plot(df.loudness, df.popularity, hue=df.pop_rating, ylabel="Popularity",
          xlabel="Loudness", title="Loudness vs. Popularity")
```



This scatterplot is more dense toward higher levels of loudness. Loudness may not guarantee popularity, but it seems the odds are greater for loud songs than songs that have low levels of loudness. Let's use a regression line to see if that's true (Non-Descriptive method example):

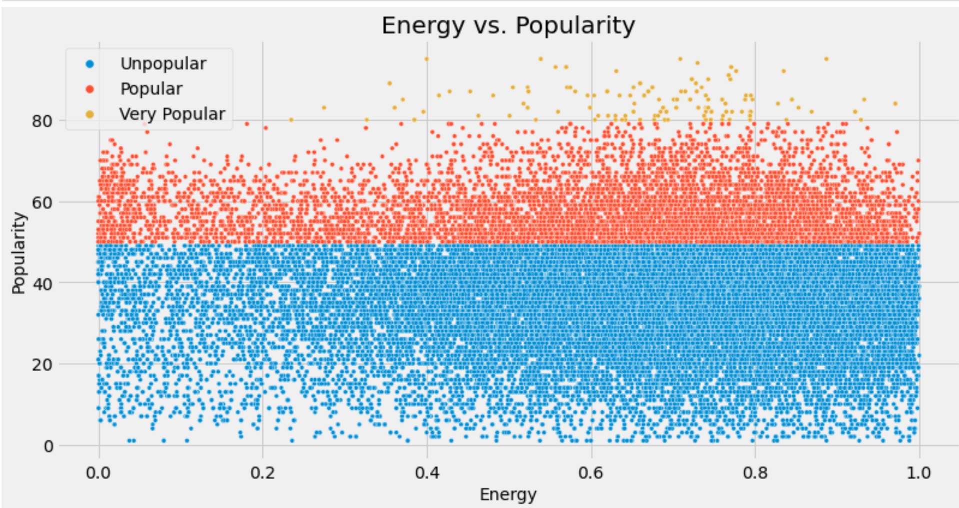
In [36]:
```
regress_plot(df.loudness, df.popularity, data=df, ylabel="Popularity",
         xlabel="Loudness", title="Loudness vs. Popularity")
```



The regressor shows there is somewhat of a positive correlation, but it's not a very strong fit.

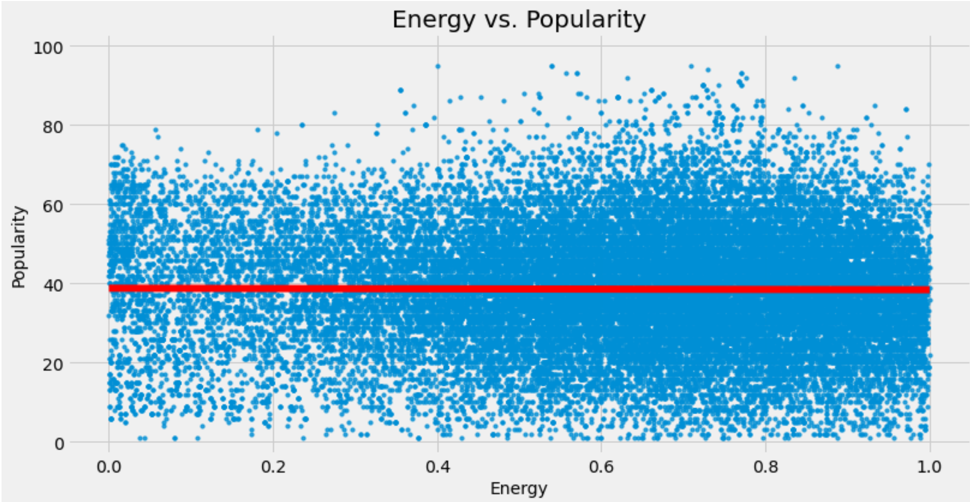
According to the heatmap, Popularity is stronger correlated to Energy than Loudness, so let's check what that is about:

In [37]:
```
scat_plot(df.energy, df.popularity, hue=df.pop_rating,
          xlabel="Energy",
          ylabel="Popularity",
          title="Energy vs. Popularity")
```



Most Very Popular Songs definitely are in the higher Energy range.

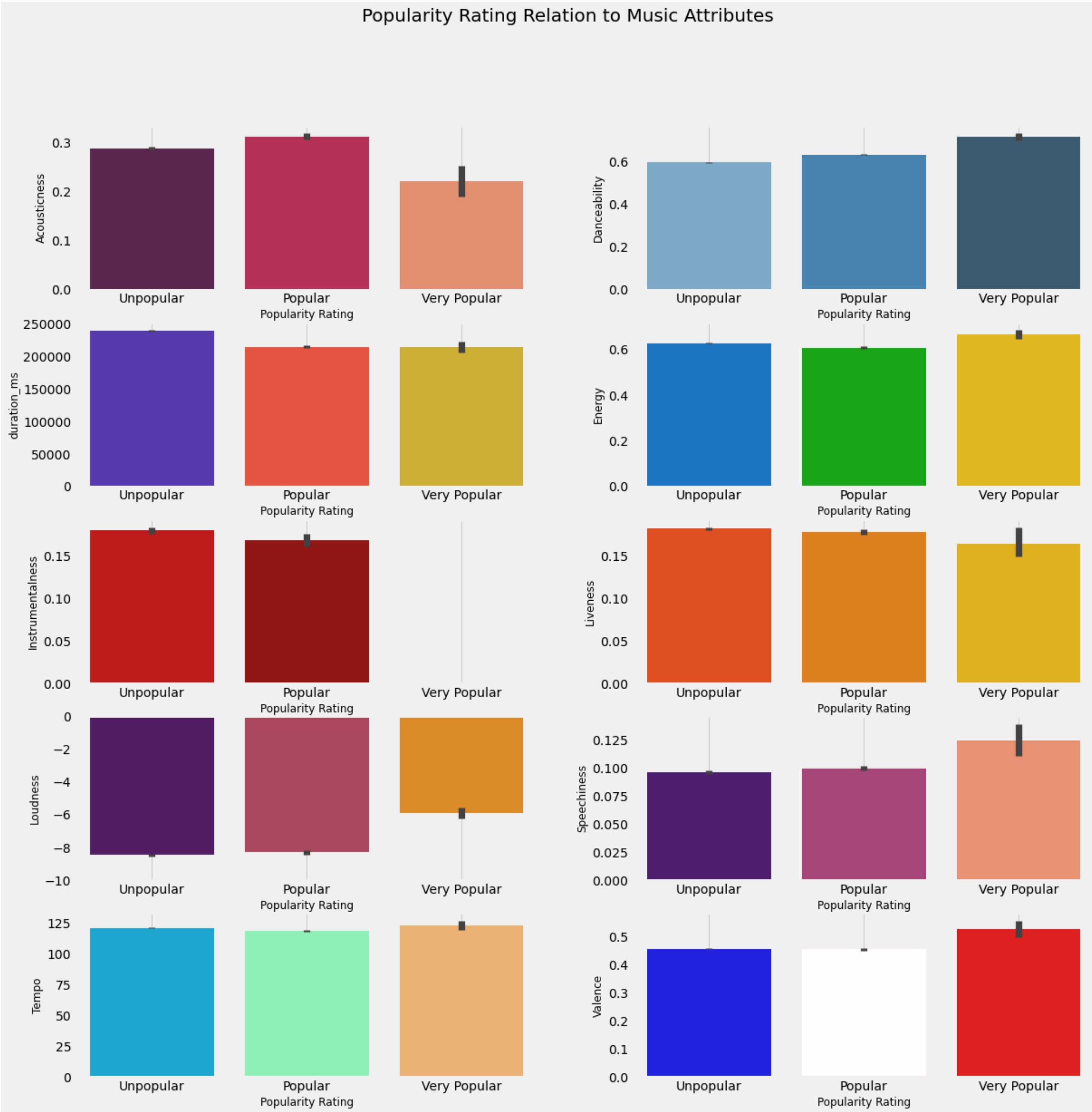
In [38]:
```
regress_plot(df.energy, df.popularity, data=df,
          xlabel="Energy",
          ylabel="Popularity",
          title="Energy vs. Popularity")
```



Even though the regressor trends downwards, it seems that the most popular songs are high energy.

We already looked at how song attributes relate to popularity rating over the years, but let's do the same taking the dataset as a whole into consideration:

In [39]:
```
# Categorical vs. Numerical
plt.rcParams['figure.figsize'] = large_figsize

plt.subplot(5, 2, 1)
sns.barplot(x=df.pop_rating, y=df.acousticness, palette = 'rocket')
plt.xlabel('Popularity Rating', fontsize = 12)
plt.ylabel('Acousticness', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 2)
sns.barplot(x=df.pop_rating, y=df['danceability'], palette='Blues_d')
plt.xlabel('Popularity Rating', fontsize=12)
plt.ylabel('Danceability', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 3)
sns.barplot(x=df.pop_rating, y=df['duration_ms'], palette='CMRmap')
plt.xlabel("Popularity Rating", fontsize=12)
plt.ylabel('duration_ms', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 4)
sns.barplot(x=df.pop_rating, y=df['energy'], palette='nipy_spectral')
plt.xlabel("Popularity Rating", fontsize=12)
plt.ylabel('Energy', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 5)
sns.barplot(x=df.pop_rating, y=df['instrumentalness'], palette='flag')
plt.xlabel("Popularity Rating", fontsize=12)
plt.ylabel('Instrumentalness', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 6)
sns.barplot(x=df.pop_rating, y=df['liveness'], palette='autumn')
plt.xlabel("Popularity Rating", fontsize=12)
plt.xlabel("Popularity Rating", fontsize=12)
plt.ylabel('Liveness', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 7)
sns.barplot(x=df.pop_rating, y=df['loudness'], palette='inferno')
plt.xlabel("Popularity Rating", fontsize=12)
plt.ylabel('Loudness', fontsize = 12)
plt.ylim(-10, 0)
plt.grid()

plt.subplot(5, 2, 8)
sns.barplot(x=df.pop_rating, y=df['speechiness'], palette='magma')
plt.xlabel("Popularity Rating", fontsize=12)
plt.ylabel('Speechiness', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 9)
sns.barplot(x=df.pop_rating, y=df['tempo'], palette='rainbow')
plt.xlabel("Popularity Rating", fontsize=12)
plt.ylabel('Tempo', fontsize = 12)
plt.grid()

plt.subplot(5, 2, 10)
sns.barplot(x=df.pop_rating, y=df['valence'], palette='seismic')
plt.xlabel("Popularity Rating", fontsize=12)
plt.ylabel('Valence', fontsize = 12)
plt.grid()

plt.suptitle('Popularity Rating Relation to Music Attributes', fontsize = 20)
plt.show()
```



This confirms what we already found:

- popular songs have lower acousticness than unpopular songs
- Popular songs are louder
- Popular songs have more speechiness
- Popular songs are moe positive in nature
- Popular songs are more energetic
- Very Popular songs are not instrumental at all

Data Preprocessing

Even though we can logically infer that logistic regression will probably not work well in this situation (too much noise in the scatterplots), we still want to check it out and confirm that it's not a great solution to our problem.

col = df.columns.to_list()# This is left over code from ideas that were later abandoned. Here I was trying to improve model's performance by using One Hot Encoder on some columns: # One Hot Encode the following features onehot_enc = ['mode', 'key', 'time_signature', 'pop_rating']continuous_data = [i for i in col if i not in onehot_enc]

First, we're going to try without the categorical pop_rating column

In [40]:
```
df_cont = df.copy().reset_index(drop=True)
df_cont.head()
```

Out[40]:

	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	popularity	speechiness	tempo	time_signature	valence	release_y
0	0.1010	0.748	237667.0	0.666	0.000653	6.0	0.0976	-6.094	0.0	47.0	0.0833	114.982	4.0	0.359	2
1	0.1910	0.608	243667.0	0.664	0.042700	5.0	0.1200	-8.261	0.0	35.0	0.0435	100.011	4.0	0.513	2
2	0.0786	0.470	157500.0	0.828	0.000000	9.0	0.1780	-6.280	1.0	55.0	0.0700	96.149	4.0	0.856	2
3	0.3160	0.336	207354.0	0.861	0.000107	11.0	0.2160	-3.274	1.0	41.0	0.1020	179.142	4.0	0.789	2
4	0.1480	0.790	174189.0	0.722	0.000000	1.0	0.1720	-4.149	0.0	49.0	0.0678	98.109	4.0	0.948	2

In [41]:

```
# Drop release year as we don't want it to be a factor in the prediction
index = ['acousticness', 'danceability', 'duration_ms', 'energy', 'instrumentalness',
         'key', 'liveness', 'loudness', 'mode', 'speechiness', 'tempo',
         'time_signature', 'valence', 'popularity']

# drop pop_rating and reindex
df_cont.drop('pop_rating', inplace=True, axis=1)
df_cont = df_cont.reindex(columns=index)
df_cont.head(3)
```

Out[41]:

	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo	time_signature	valence	popularity
0	0.1010	0.748	237667.0	0.666	0.000653	6.0	0.0976	-6.094	0.0	0.0833	114.982	4.0	0.359	47.0
1	0.1910	0.608	243667.0	0.664	0.042700	5.0	0.1200	-8.261	0.0	0.0435	100.011	4.0	0.513	35.0
2	0.0786	0.470	157500.0	0.828	0.000000	9.0	0.1780	-6.280	1.0	0.0700	96.149	4.0	0.856	55.0

In [42]:

```
x = pd.get_dummies(df_cont['mode'])
```

In [43]:

```
df_cont = pd.concat([df_cont, x], axis=1)
df_cont.head(3)
```

Out[43]:

	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo	time_signature	valence	popularity	0.0	1.0
0	0.1010	0.748	237667.0	0.666	0.000653	6.0	0.0976	-6.094	0.0	0.0833	114.982	4.0	0.359	47.0	1	0
1	0.1910	0.608	243667.0	0.664	0.042700	5.0	0.1200	-8.261	0.0	0.0435	100.011	4.0	0.513	35.0	1	0
2	0.0786	0.470	157500.0	0.828	0.000000	9.0	0.1780	-6.280	1.0	0.0700	96.149	4.0	0.856	55.0	0	1

In [44]:

```
df_cont.rename(columns = {0.0: 'major key', 1.0: 'minor key'}, inplace=True)
df_cont.drop('mode', inplace=True, axis=1)
```

In [45]:

```
index = ['acousticness', 'danceability', 'duration_ms', 'energy', 'instrumentalness',
         'key', 'liveness', 'loudness', 'mode', 'speechiness', 'tempo',
         'time_signature', 'valence', 'major key',
         'minor key','popularity']
df_cont = df_cont.reindex(columns=index)
df_cont.drop('mode', inplace=True, axis=1)
df_cont.head(3)
```

Out[45]:

	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	speechiness	tempo	time_signature	valence	major key	minor key	popularity
0	0.1010	0.748	237667.0	0.666	0.000653	6.0	0.0976	-6.094	0.0833	114.982	4.0	0.359	1	0	47.0
1	0.1910	0.608	243667.0	0.664	0.042700	5.0	0.1200	-8.261	0.0435	100.011	4.0	0.513	1	0	35.0
2	0.0786	0.470	157500.0	0.828	0.000000	9.0	0.1780	-6.280	0.0700	96.149	4.0	0.856	0	1	55.0

Splitting the Data

In [46]:

```
# seperate target data from training data
Y = df_cont['popularity']
X = df_cont.iloc[:, 0:-1]
print("The shape of X:", X.shape)
print("The shape of Y:", Y.shape)
```

```
The shape of X: (36435, 14)
The shape of Y: (36435,)
```

In [47]:

```
# split training data in train and test data
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.2, random_state=0)

print("The shape of x_train:", x_train.shape)
print("The shape of x_test:", x_test.shape)
print("The shape of y_train:", y_train.shape)
print("The shape of y_test:", y_test.shape)
```

```
The shape of x_train: (29148, 14)
The shape of x_test: (7287, 14)
The shape of y_train: (29148,)
The shape of y_test: (7287,)
```

Let's fit the training data and calculate the R2 score. This process, and all the similar ones to follow, is timed to get an understanding of the computational requirements for each model.

Let's also create a custom loss function to give a more complete idea of the model performance. We'll call this loss_function and it will calculate counts for different levels of error between the predicted values and the actual test values, and give an overall average of the differences between predicted and actual as well.

In [48]:

```
# This function only works with the continuous models, not with classifiers
def loss_function(predicts, truths):
    zipped = zip(predicts, truths)
    zipped_list = list(zipped)
```

```
    differences = {'Less than 5': 0, '5 - 10': 0, 'More than 10': 0, 'Average Error': 0}
    sum = 0

    for pair in zipped_list:
        sum += abs(pair[0] - pair[1])
        if abs(pair[0] - pair[1]) < 5:
            differences['Less than 5'] += 1
        elif 5 <= abs(pair[0] - pair[1]) < 10:
            differences['5 - 10'] += 1
        else:
            differences['More than 10'] += 1

    differences['Average Error'] = sum / len(zipped_list)

    return differences
```

Let's create a Linear Regression model.

In [49]:
```
%%time

lr = LinearRegression()
lr.fit(x_train, y_train)

y_pred = lr.predict(x_test)

print(lr.score(x_test, y_test))
```

```
0.04190065431881562
CPU times: user 36.3 ms, sys: 7.85 ms, total: 44.2 ms
Wall time: 41.2 ms
```

In [50]:
```
loss_function(y_pred, y_test)
```

Out[50]:
```
{'Less than 5': 1987,
 '5 - 10': 1749,
 'More than 10': 3551,
 'Average Error': 11.672639816814572}
```

These scores are not very good, as expected. The custom loss function makes us feel a little better about it though, as it isn't as bad as the score would lead us to believe. Over 7000 predictions maye be well off the mark, but that means around 29,000 were reasonably accurate, and the average error difference is not completely terrible.

However, classifying algorithms will probably work better to solve our problem. So let's get started.

Preprocessing

In [51]:
```
# create a dataframe copy to work with
df_cat = df.copy().reset_index()
df_cat.drop(['index', 'popularity', 'release_year'], inplace=True, axis=1)
df_cat.head(3)
```

Out[51]:

	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo	time_signature	valence	pop_rating
0	0.1010	0.748	237667.0	0.666	0.000653	6.0	0.0976	-6.094	0.0	0.0833	114.982	4.0	0.359	Unpopular
1	0.1910	0.608	243667.0	0.664	0.042700	5.0	0.1200	-8.261	0.0	0.0435	100.011	4.0	0.513	Unpopular
2	0.0786	0.470	157500.0	0.828	0.000000	9.0	0.1780	-6.280	1.0	0.0700	96.149	4.0	0.856	Popular

This is left over code from ideas that were later abandoned. Here I was trying to improve model's performance by using One Hot Encoder on some columns: ''''col_to_drop = list(df_cat.iloc[:, :5].columns.values) col_to_drop += ['liveness', 'loudness', 'speechiness', 'tempo', 'valence', 'pop_rating', 'time_signature'] df_cat_toencode = df_cat.drop(col_to_drop, axis=1) #list_to_encode = list(df_cat_toencode.columns.values) #list_to_encode''''''x = pd.get_dummies(df_cat_toencode, columns=df_cat_toencode.columns.values) x.sample()''''''df_cat = pd.concat([df_cat, x], axis=1) df_cat.drop(['key', 'mode', 'time_signature'], axis=1, inplace=True) df_cat.columns''''''cols = ['key', 'mode', 'time_signature', 'pop_rating'] df_cat[cols] = df_cat[cols].apply(lambda x: x.astype('category'))'''

Let's replace the ordinal values in our target column by assigning 3 to Very Popular, 2 to Popular, and 1 to Unpopular uing the replace function.

In [52]:
```
# Replace Ordinal Values
df_cat['pop_rating'] = df_cat['pop_rating'].replace(('Very Popular', 'Popular', 'Unpopular'), (3, 2, 1))
df_cat['pop_rating'].value_counts()
```

Out[52]:
```
1    27969
2     8278
3      188
Name: pop_rating, dtype: int64
```

Splitting and Normalizing the data

All attribute columns range between 0 and 1, except for duration_ms, loudness, tempo, and release_year. Let's normalize them.

In [53]:
```
YY = df_cat['pop_rating']
XX = df_cat.drop('pop_rating', axis=1)

print(XX.shape)
print(YY.shape)
```

```
(36435, 13)
(36435,)
```

In [54]:
```
XX.head(3)
```

Out[54]:

	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo	time_signature	valence
0	0.1010	0.748	237667.0	0.666	0.000653	6.0	0.0976	-6.094	0.0	0.0833	114.982	4.0	0.359
1	0.1910	0.608	243667.0	0.664	0.042700	5.0	0.1200	-8.261	0.0	0.0435	100.011	4.0	0.513
2	0.0786	0.470	157500.0	0.828	0.000000	9.0	0.1780	-6.280	1.0	0.0700	96.149	4.0	0.856

In [55]:

```
YY.head()
```

Out[55]:

```
0    1
1    1
2    2
3    1
4    1
Name: pop_rating, dtype: category
Categories (3, int64): [1, 2, 3]
```

The target data is imbalanced as e concluded earlier, so we'll implement SMOTE.

In [56]:

```
from imblearn.over_sampling import SMOTE

sn = SMOTE(random_state=0)

sn.fit(XX, YY)

x_resampled, y_resampled = sn.fit_resample(XX, YY)

print(x_resampled.shape)
print(y_resampled.shape)
```

```
(83907, 13)
(83907,)
```

Our samples are now balanced.

In [57]:

```
# Splitting the train data in a test set and train set
x_train, x_test, y_train, y_test = train_test_split(x_resampled, y_resampled, test_size=0.25, random_state=0)

print("The shape of x_train:", x_train.shape)
print("The shape of x_test:", x_test.shape)
print("The shape of y_train:", y_train.shape)
print("The shape of y_test:", y_test.shape)
print()
print(y_train.value_counts())
print()
print(y_test.value_counts())
```

```
The shape of x_train: (62930, 13)
The shape of x_test: (20977, 13)
The shape of y_train: (62930,)
The shape of y_test: (20977,)

1    21035
2    20972
3    20923
Name: pop_rating, dtype: int64

3    7046
2    6997
1    6934
Name: pop_rating, dtype: int64
```

The balance of samples looks good, as shown above.

In [58]:

```
# confirm no values are missing
x_test.isna().sum().sum()
```

Out[58]:

```
0
```

Just for the fun of it, let's see if linear Regression works better now:

In [59]:

```
lr = LinearRegression()
lr.fit(x_train, y_train)

y_pred = lr.predict(x_test)

print(lr.score(x_test, y_test))
```

```
0.15891425942788384
```

In [60]:

```
cvals = cross_val_score(lr, XX, YY, cv=6)

# check the results
print(cvals)
print('The mean cross-validation score is: {num:.{dig}f}'.format\
      (num=np.mean(cvals), dig=4))
```

```
[-0.15134531 -0.08163616 -0.03986326  0.00880856 -0.01406349 -0.13214603]
The mean cross-validation score is: -0.0684
```

The scores are definitely a little better, however the results are unacceptable for us in the real world. So let's move on from it.

Build Classification Models

First, let's see how a basic decision tree performs:

In [61]:
```
# Basic decision tree
model1 = DecisionTreeClassifier(max_depth=20, random_state=0)
model1.fit(x_train, y_train)

y_pred = model1.predict(x_test)

print(accuracy_score(y_test, y_pred))
```

```
0.8315774419602422
```

That performs better right away! Let's perform a GridSearch to get the best paramaters.

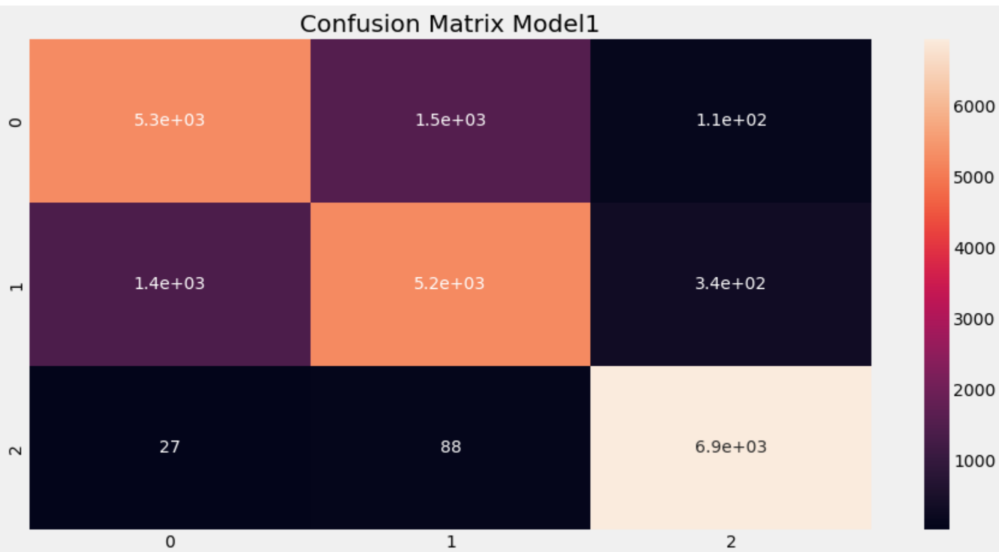
In [62]:
```
# perform GridSearch
params = {'max_depth': [2, 10, 20, 40, 50],
          'min_samples_leaf': np.arange(1, 10, 2),}

model1 = DecisionTreeClassifier(random_state=0)

model1_cv = GridSearchCV(model1, params, cv=6)

model1_cv.fit(x_train, y_train)

print(model1_cv.best_params_)
print('The average runtime is: ', np.mean(model1_cv.cv_results_['mean_fit_time']))
print('The best score is: ', model1_cv.best_score_)
```

```
{'max_depth': 40, 'min_samples_leaf': 1}
The average runtime is:  0.5069964838027955
The best score is:  0.8316064976307191
```

In [63]:
```
# get and print best parameters
p = model1_cv.best_estimator_.predict(x_test)

print(accuracy_score(y_test, p))
```

```
0.8424464890117748
```

In [64]:
```
# show confustion matrix
cm = confusion_matrix(y_test, y_pred)
plt.rcParams['figure.figsize'] = small_figsize
sns.heatmap(cm, annot=True)
plt.title("Confusion Matrix Model1")
plt.show()
```



In [65]:
```
# Accuracy and Classification Report
print(accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
0.8315774419602422
              precision    recall  f1-score   support

           1       0.78      0.76      0.77      6934
           2       0.76      0.75      0.76      6997
           3       0.94      0.98      0.96      7046

    accuracy                           0.83     20977
   macro avg       0.83      0.83      0.83     20977
weighted avg       0.83      0.83      0.83     20977
```

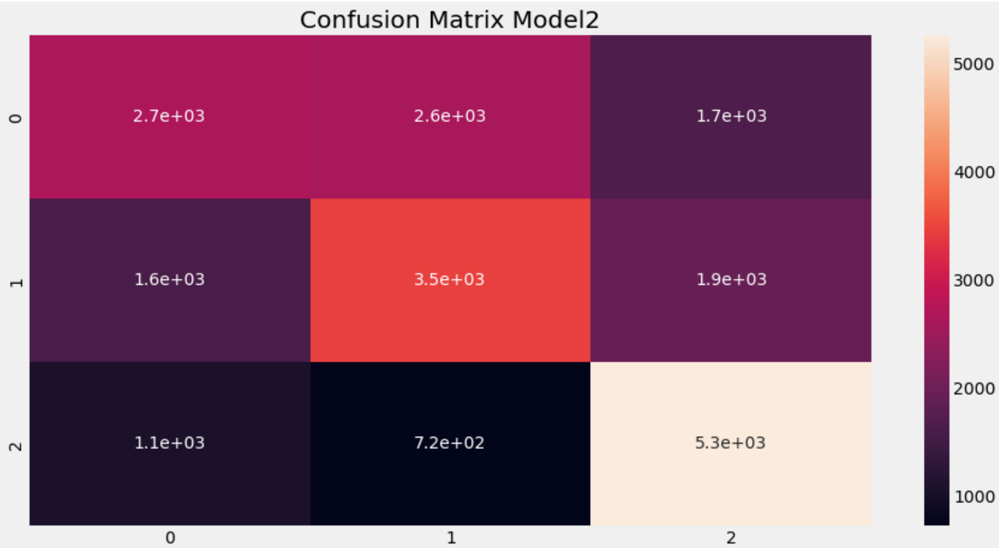
This model is good at predicting Very Popular songs with an f1-score of 96%. Let's see if we can get other models to perform better. Starting with Bagging.

Bagging

In [66]:
```
%%time

from sklearn.ensemble import BaggingClassifier

tree = DecisionTreeClassifier(max_depth=2, random_state=0)
model2 = BaggingClassifier(base_estimator=tree, n_estimators=100, random_state=0)
```

```
# perform cross validation
bag_cv = cross_val_score(model2, x_train, y_train, cv=6)
model2.fit(x_train, y_train)
y_pred = model2.predict(x_test)

print(bag_cv)
print('The mean cross-validation score is: {num:.{dig}f}'.format\
      (num=np.mean(bag_cv), dig=4))

#show confusion matrix
cm = confusion_matrix(y_test, y_pred)
plt.rcParams['figure.figsize'] = small_figsize
sns.heatmap(cm, annot=True)
plt.title("Confusion Matrix Model2")
plt.show()
```

```
[0.53989894 0.534274   0.53499237 0.548627   0.53442029 0.53308543]
The mean cross-validation score is: 0.5375
```




```
CPU times: user 48.5 s, sys: 281 ms, total: 48.8 s
Wall time: 49.2 s
```

In [67]:
```
# Accuracy and Classification Report
print(accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
0.5430709825046479
              precision    recall  f1-score   support

           1       0.50      0.38      0.43      6934
           2       0.51      0.50      0.50      6997
           3       0.59      0.75      0.66      7046

    accuracy                           0.54     20977
   macro avg       0.53      0.54      0.53     20977
weighted avg       0.54      0.54      0.53     20977
```

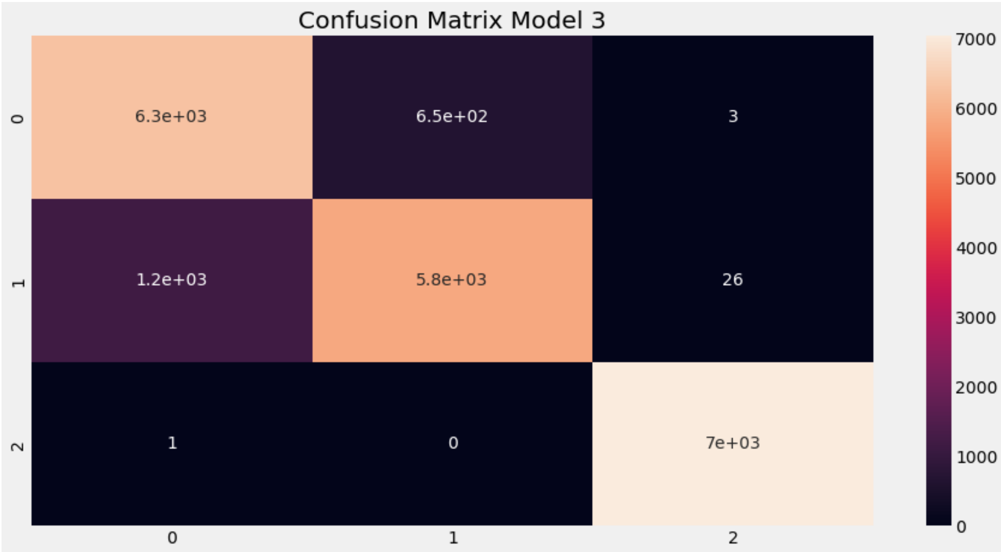
This model2 performs worse than model1. Let's move on to trying out the Random Forest.

Random Forest

In [68]:
```
%%time

# train a first version of the model
model3 = RandomForestClassifier(n_estimators=100, random_state=0, max_depth=50)
model3.fit(x_train, y_train)

# get predictions
y_pred = model3.predict(x_test)

# print accuracy score
print(accuracy_score(y_test, y_pred))
```

```
0.9121418696667779
CPU times: user 14.5 s, sys: 92.3 ms, total: 14.6 s
Wall time: 14.8 s
```

In [69]:
```
%%time

cm = confusion_matrix(y_test, y_pred)
plt.rcParams['figure.figsize'] = small_figsize
sns.heatmap(cm, annot=True)
plt.title("Confusion Matrix Model 3")
plt.show()
```




```
CPU times: user 152 ms, sys: 6.16 ms, total: 158 ms
Wall time: 165 ms
```

In [70]:
```
# Accuracy and Classification Report
print(accuracy_score(y_test, y_pred))
print()
print(classification_report(y_test, y_pred))
```

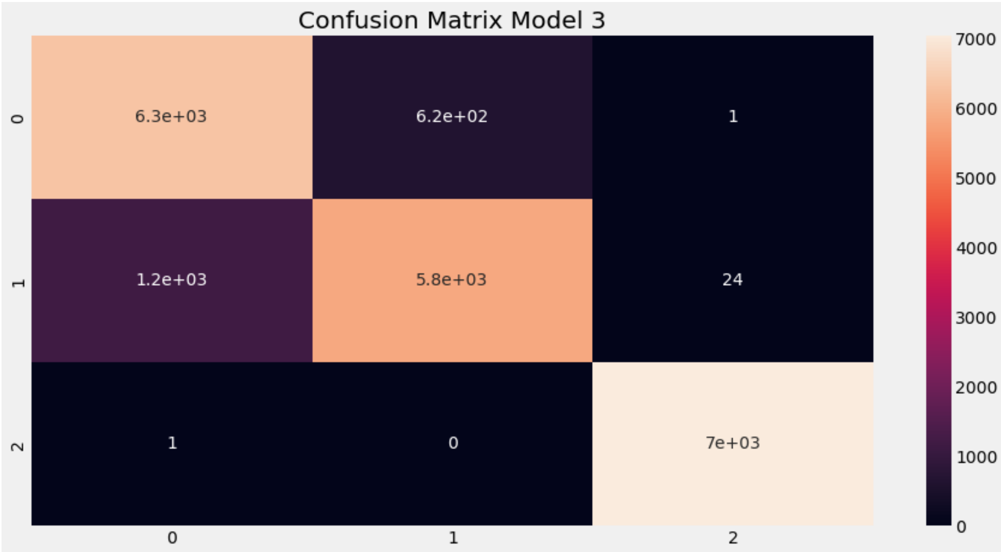
```
0.9121418696667779

              precision    recall  f1-score   support

           1       0.84      0.91      0.87      6934
           2       0.90      0.83      0.86      6997
           3       1.00      1.00      1.00      7046

    accuracy                           0.91     20977
   macro avg       0.91      0.91      0.91     20977
weighted avg       0.91      0.91      0.91     20977
```

CPU times are acceptable and accuracy score is very promising. It's higher than the required threshold of 80%, so this is a good thing. Let's perform a GridSearch again to find best parameters.

In [95]:
```
# perform GridSearch
model3_params = {'n_estimators': [100, 250, 350],
            'max_depth': [2, 50, 100],
            'min_samples_leaf': [1, 2]}

model3_new = RandomForestClassifier(random_state=0)

model3_cv = GridSearchCV(model3_new, model3_params, n_jobs=-1, cv=5)

model3_cv.fit(x_train, y_train)

print(model3_cv.best_params_)
```

```
{'max_depth': 50, 'min_samples_leaf': 1, 'n_estimators': 350}
```

In [96]:
```
%%time

# train the model again, this time using the best parameters
model3 = RandomForestClassifier(max_depth=50, n_estimators=350, min_samples_leaf=1, random_state=0)

model3.fit(x_train, y_train)

y_pred = model3.predict(x_test)

cm = confusion_matrix(y_test, y_pred)
plt.rcParams['figure.figsize'] = small_figsize
sns.heatmap(cm, annot=True)
plt.title("Confusion Matrix Model 3")
plt.show()
```




```
CPU times: user 50.5 s, sys: 323 ms, total: 50.8 s
Wall time: 51.2 s
```

In [97]:

```
# Accuracy and Classification Report
print(accuracy_score(y_test, y_pred))
print()
print(classification_report(y_test, y_pred))
```

```
0.9140963912856939

              precision    recall  f1-score   support

           1       0.85      0.91      0.88      6934
           2       0.90      0.83      0.87      6997
           3       1.00      1.00      1.00      7046

    accuracy                           0.91     20977
   macro avg       0.92      0.91      0.91     20977
weighted avg       0.92      0.91      0.91     20977
```

The confusion matrix and classification reports look even better than the basic Desicion Tree. With an f1-score of 88% for unpopular songs, 87% for popular songs, and 100% for very popular songs. That is well above our goal of predicting song popularity with at least 80% accuracy.

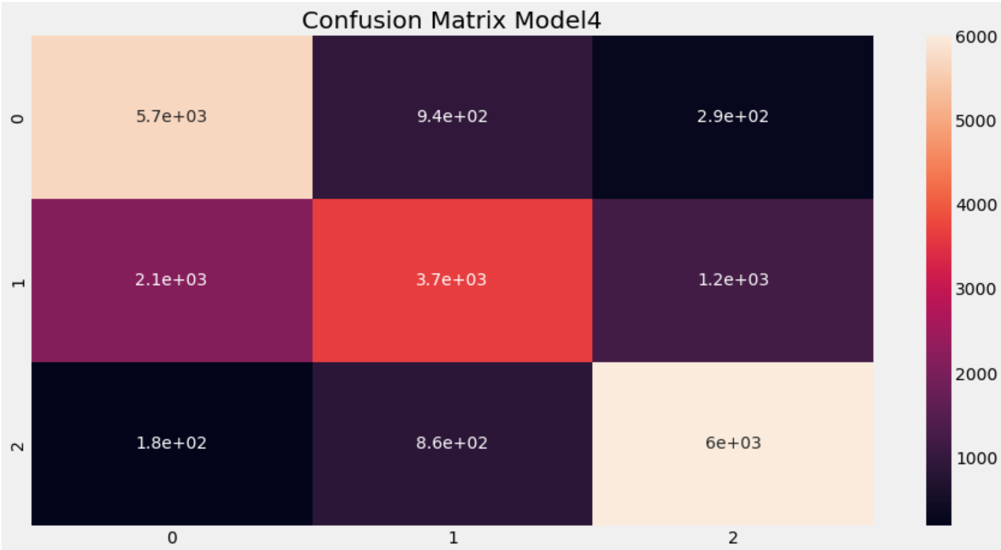
Boosting Models

Next, we want to explore if the model improves when using AdaBoost and Gradient Boosting.

AdaBoost

In [71]:

```
%%time

dtc = DecisionTreeClassifier(max_depth=1, random_state=0)

model4 = AdaBoostClassifier(base_estimator=dtc, n_estimators=350)

model4.fit(x_train, y_train)

y_pred = model4.predict(x_test)

# print confusion matrix
cm = confusion_matrix(y_test, y_pred)
plt.rcParams['figure.figsize'] = small_figsize
sns.heatmap(cm, annot=True)
plt.title("Confusion Matrix Model4")
plt.show()
```



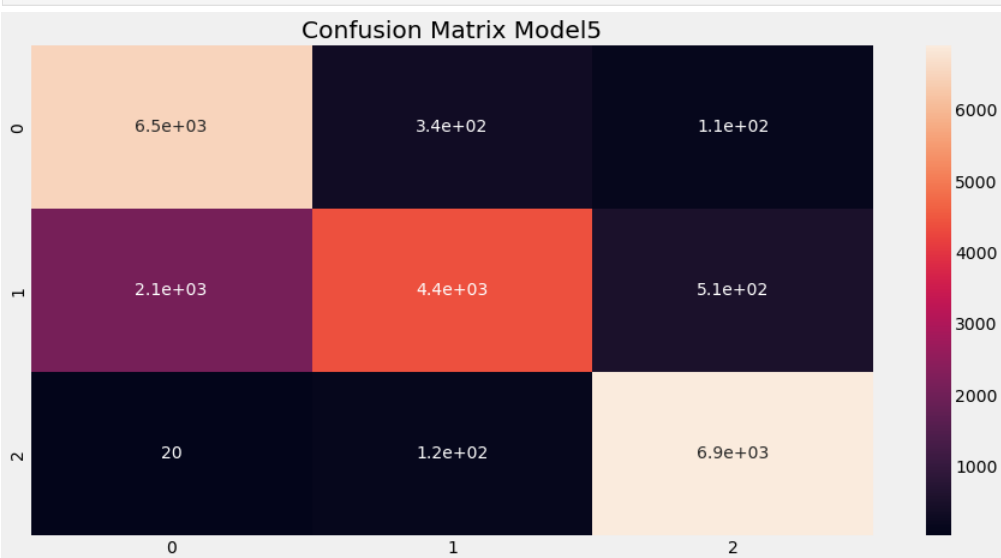

```
CPU times: user 27.4 s, sys: 485 ms, total: 27.9 s
Wall time: 31.4 s
```

In [72]:
```python
# Accuracy and Classification Report
print(accuracy_score(y_test, y_pred))
print()
print(classification_report(y_test, y_pred))
```

```
0.732278209467512

              precision    recall  f1-score   support

           1       0.71      0.82      0.76      6934
           2       0.67      0.52      0.59      6997
           3       0.80      0.85      0.82      7046

    accuracy                           0.73     20977
   macro avg       0.73      0.73      0.72     20977
weighted avg       0.73      0.73      0.73     20977
```

Adaboost performs average compared to Random Forest. Let's try GrandientBoosting next.

In [73]:
```python
%%time

model5 = GradientBoostingClassifier(max_depth=3, n_estimators=350, random_state=0)

model5.fit(x_train, y_train)

y_pred = model5.predict(x_test)

# print confusion matrix
cm = confusion_matrix(y_test, y_pred)
plt.rcParams['figure.figsize'] = small_figsize
sns.heatmap(cm, annot=True)
plt.title("Confusion Matrix Model5")
plt.show()
```




```
CPU times: user 3min 3s, sys: 3.63 s, total: 3min 7s
Wall time: 3min 15s
```

In [74]:
```python
# Accuracy and Classification Report
print(accuracy_score(y_test, y_pred))
```

```
print()
print(classification_report(y_test, y_pred))
```

```
0.8477856700195452

              precision    recall  f1-score   support

           1       0.75      0.94      0.83      6934
           2       0.91      0.63      0.74      6997
           3       0.92      0.98      0.95      7046

    accuracy                           0.85     20977
   macro avg       0.86      0.85      0.84     20977
weighted avg       0.86      0.85      0.84     20977
```

For this dataset the Random Forest Model seems to work best. So we'll be going with that one.

In [75]:
```
# Save the Random Forest model in a pickle file
# Don't run this cell for evaluation purposes,
# as it'll save a pickle file to your hard drive, and you might not want that.
filepath = 'music_pop_predictor.pkl'
with open(filepath, 'wb') as f:
    pickle.dump(model3, f)

# code underneath represents ideas that were later abandoned.
# The deployment through Streamlit failed because Streamlit didn't like unzipping compressed pickle files
'''with gzip.open(filepath, 'wb') as f:
    pickled = pickle.dumps(model3)
    optimized_pickle = pickletools.optimize(pickled)
    f.write(optimized_pickle)'''
```

Out[75]:
```
"with gzip.open(filepath, 'wb') as f:\n    pickled = pickle.dumps(model3)\n    optimized_pickle = pickletools.optimize(pickled)\n    f.write(o
ptimized_pickle)"
```

Deployment in Streamlit

Finally, we're ready for deployment. We wrote a file called webapp,py and will point Streamlit to this file in our repository on GitHub. Webapp.py will unpickle the pickled model file. We are ready to make some real time predictions!

In [322…
```
# redundant code used for testing
col_list = x_test.columns.tolist()
col_list
```

Out[322]:
```
['acousticness',
 'danceability',
 'duration_ms',
 'energy',
 'instrumentalness',
 'key',
 'liveness',
 'loudness',
 'mode',
 'speechiness',
 'tempo',
 'time_signature',
 'valence']
```

In [323…
```
# redundant code used for testing
filepath = 'music_pop_predictor.pkl'
with open(filepath, 'rb') as f:
    model = pickle.load(f)

y_pred = model.predict(x_test)
```

In [104…
```
df[df['pop_rating'] == 'Very Popular']
```

Out[104]:

	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	popularity	speechiness	tempo	time_signature	valence	rel
21460	0.0323	0.731	220454.0	0.794	0.000026	0.0	0.1120	-5.126	0.0	80.0	0.0522	139.994	4.0	0.356	
26499	0.5710	0.765	129600.0	0.327	0.047900	6.0	0.0787	-8.029	1.0	80.0	0.2770	74.931	4.0	0.292	
31138	0.0175	0.657	341574.0	0.746	0.000000	1.0	0.1070	-6.277	1.0	80.0	0.0464	136.887	4.0	0.467	
31139	0.0175	0.657	341574.0	0.746	0.000000	1.0	0.1070	-6.277	1.0	80.0	0.0464	136.887	4.0	0.467	
31140	0.0175	0.657	341574.0	0.746	0.000000	1.0	0.1070	-6.277	1.0	80.0	0.0464	136.887	4.0	0.467	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
53211	0.2690	0.737	192453.0	0.747	0.000000	10.0	0.2190	-4.818	1.0	82.0	0.0323	105.943	4.0	0.447	
53212	0.2690	0.737	192453.0	0.747	0.000000	10.0	0.2190	-4.818	1.0	82.0	0.0323	105.943	4.0	0.447	
53218	0.3870	0.836	240082.0	0.621	0.000092	1.0	0.1040	-4.685	0.0	81.0	0.0892	101.995	4.0	0.762	
53219	0.3870	0.836	240082.0	0.621	0.000092	1.0	0.1040	-4.685	0.0	81.0	0.0892	101.995	4.0	0.762	
53221	0.0135	0.421	175424.0	0.653	0.000000	9.0	0.1140	-4.850	0.0	84.0	0.3980	78.600	4.0	0.733	

188 rows × 16 columns

In [ ]: