Spotify Recommender and Song Popularity Predictor

By Loizos Konstantinou, Mikaela Spaventa, Tianyang Han

Table of contents

*Please check the table of contents tab to see a breakdown of each section.

Dataset: https://www.kaggle.com/datasets/lehaknarnauli/spotify-datasets?select=tracks.csv (https://www.kaggle.com/datasets/lehaknarnauli/spotify-datasets?select=tracks.csv)

Spotify Audio Features

For every track on their platform, Spotify provides data for thirteen Audio Features.The Spotify Web API developer ('https://developer.spotify.com/') guide defines them as follows:

Danceability: Describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity.

Valence: Describes the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).

Energy: Represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale.

Tempo: The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece, and derives directly from the average beat duration.

Loudness: The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks.

Speechiness: This detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value.

Instrumentalness: Predicts whether a track contains no vocals. “Ooh” and “aah” sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly “vocal”.

Liveness: Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live.

Acousticness: A confidence measure from 0.0 to 1.0 of whether the track is acoustic.

Key: The estimated overall key of the track. Integers map to pitches using standard Pitch Class notation . E.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on. Mode: Indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.

Duration: The duration of the track in milliseconds.

Time Signature: An estimated overall time signature of a track. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure).

Section 1: Motivation

MOTIVATION

Music changes people's mood and connects people and Spotify is the main source that consumers use when it comes to listening to music. Our team will use Spotify Web API to work with a spotify dataset from Kaggle (https://www.kaggle.com/datasets/ektanegi/spotifydata-19212020) that contains more than 586,672 songs from 1921 until 2021. The goal is to build a song recommendor for people based on features that are related with their playlist preferences. In addition, we want to explore music trends throughout the years in order to build a song popularity predictor that features the trening song attributes.

Working with the Spotify dataset to make these predictions will allow us to get a thorough understanding on motivation drivers about people in conjunction with trends and perception about music.

We chose this particular Spotify dataset for a couple reasons: \

1. It consists many songs. Therefore, we have more data to see trends and build models with
2. It consists many features that we were interested in working with. For instance, tbis dataset contains information regarding the genre, artist's popularity, and artist's number of followers on Spotify. It was somewhat difficult to find a dataset with all of these features.
3. This dataset has a usability of ***2.94/10***, making it extremely difficult to work with. Many datasets on Kaggle require very little cleaning if any at all. We wanted to make this project more challenging and realistic (since data scientists spend 80-90% of their time cleaning) by using a dataset like this. Just a couple difficulties with this dataset include as follows:

    - Columns with null values
    - Join incompatibility (many songs had multiple artists making it more difficult to join the track and artist datasets
    - Too many genres to accurately place into a broad genre
    - Some of the same songs were duplicates with various ids.
    - Many song genres were unknown
    - A lot of variance in columns that were important for predicting song_popularity (artist_popularity, year, etc.)

In the end, we were able to work through most of these problems in the dataset using principles we learned in class

Section 2: Imports

Imports

In [3]:
```
import json
import glob
import pandas as pd
import numpy as np
import datetime as dt
import re
import os
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib import cm
from tqdm import tqdm
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import warnings
warnings.filterwarnings('ignore')
import wordcloud
from collections import Counter
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from IPython.display import display
```

In [4]:
```
pd.set_option('display.max_columns', 500)
```

Section 3: Exploring Raw Data and Cleaning

In this section, we will explore our raw data prior to making enhancements and then clean the dataset so it is satisfactory for the project's next steps. Since the usability of this dataset was 2.94, this dataset required many adjustments. This section alone took the longest to implement and required many frequent adjustments.

Exploring Raw Data

In [8]:
```
#Read in csv files
df_artists = pd.read_csv('artists.csv')
df_tracks = pd.read_csv('tracks.csv')
```

In [9]:
```
#Find the size of each dataset (# rows, # columns)
print(df_artists.shape)
print(df_tracks.shape)
```

```
(1104349, 5)
(586672, 20)
```

We see that the datasets are very large. Note, df_artists is about twice as large as df_tracks.

In [10]:
```
#Print first 5 entres of df_artists
df_artists.head(5)
```

Out[10]:

	id	followers	genres	name	popularity
0	0DheY5irMjBUeLybbCUEZ2	0.0	[]	Armid & Amir Zare Pashai feat. Sara Rouzbehani	0
1	0DlhY15l3wsrnlfGio2bjU	5.0	[]	ปูนา ภาวิณี	0
2	0DmRESX2JknGPQyO15yxg7	0.0	[]	Sadaa	0
3	0DmhnbHjm1qw6NCYPeZNgJ	0.0	[]	Tra'gruda	0
4	0Dn11fWM7vHQ3rinvWEl4E	2.0	[]	Ioannis Panoutsopoulos	0

In [11]:
```
#Print first 5 entres of df_tracks
df_tracks.head(5)
```

Out[11]:

	id	name	popularity	duration_ms	explicit	artists	id_artists	release_date	danceability	energy	key	lo
0	35iwgR4jXetI318WEWsa1Q	Carve	6	126903	0	['Uli']	['45tIt06XoI0Iio4LBEVpls']	1922-02-22	0.645	0.4450	0	
1	021ht4sdgPcrDgSk7JTbKY	Capítulo 2.16 - Banquero Anarquista	0	98200	0	['Fernando Pessoa']	['14jtPCOoNZwquk5wd9DxrY']	1922-06-01	0.695	0.2630	0	
2	07A5yehtSnoedViJAZkNnc	Vivo para Quererte - Remasterizado	0	181640	0	['Ignacio Corsini']	['5LiOoJbxVSAMkBS2fUm3X2']	1922-03-21	0.434	0.1770	1	
3	08FmqUhxtyLTn6pAh6bk45	El Prisionero - Remasterizado	0	176907	0	['Ignacio Corsini']	['5LiOoJbxVSAMkBS2fUm3X2']	1922-03-21	0.321	0.0946	7	
4	08y9GfoqCWfOGsKdwojr5e	Lady of the Evening	0	163080	0	['Dick Haymes']	['3BiJGZsyX9sJchTqcSA7Su']	1922	0.402	0.1580	3	

In [12]:
```
#Explore info of df_artists
df_artists.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1104349 entries, 0 to 1104348
Data columns (total 5 columns):
 #   Column      Non-Null Count    Dtype  
---  ------      --------------    -----  
 0   id          1104349 non-null  object 
 1   followers   1104336 non-null  float64
 2   genres      1104349 non-null  object 
 3   name        1104349 non-null  object 
 4   popularity  1104349 non-null  int64  
dtypes: float64(1), int64(1), object(3)
memory usage: 42.1+ MB
```

In [13]:
```
#Explore info of df_tracks
df_tracks.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 586672 entries, 0 to 586671
Data columns (total 20 columns):
 #   Column            Non-Null Count   Dtype  
---  ------            --------------   -----  
 0   id                586672 non-null  object 
 1   name              586601 non-null  object 
 2   popularity        586672 non-null  int64  
 3   duration_ms       586672 non-null  int64  
 4   explicit          586672 non-null  int64  
 5   artists           586672 non-null  object 
 6   id_artists        586672 non-null  object 
 7   release_date      586672 non-null  object 
 8   danceability      586672 non-null  float64
 9   energy            586672 non-null  float64
 10  key               586672 non-null  int64  
 11  loudness          586672 non-null  float64
 12  mode              586672 non-null  int64  
 13  speechiness       586672 non-null  float64
 14  acousticness      586672 non-null  float64
 15  instrumentalness  586672 non-null  float64
 16  liveness          586672 non-null  float64
 17  valence           586672 non-null  float64
 18  tempo             586672 non-null  float64
 19  time_signature    586672 non-null  int64  
dtypes: float64(9), int64(6), object(5)
memory usage: 89.5+ MB
```

In [14]:
```
# Check nulls of df_artists
df_artists.isna().sum()
```

Out[14]:
```
id             0
followers     13
genres         0
name           0
popularity     0
dtype: int64
```

In [15]:
```
#Check nulls of df_tracks
df_tracks.isna().sum()
```

Out[15]:
```
id                   0
name                71
popularity           0
duration_ms          0
explicit             0
artists              0
id_artists           0
release_date         0
danceability         0
energy               0
key                  0
loudness             0
mode                 0
speechiness          0
acousticness         0
instrumentalness     0
liveness             0
valence              0
tempo                0
time_signature       0
dtype: int64
```

Cleaning

Now, let's start cleaning the data.

Transform Columns: genres, artists, and id_artists

First, we wanted to remove the brackets in the artists column in df_tracks, id_artists in df_tracks, and genres in df_artists. We thought this would make it easier to find null values.

In [16]:
```
#Removing brackets from the cells in the artists column
df_tracks["artists"]=df_tracks["artists"].str.replace("[", "")
df_tracks["artists"]=df_tracks["artists"].str.replace("]", "")
df_tracks["artists"]=df_tracks["artists"].str.replace("'", "")

#Replacing blank cells with null so we know to transform them
#Used: https://sparkbyexamples.com/pandas/pandas-replace-blank-values-with-nan/
df_tracks["artists"] = df_tracks["artists"].replace(r'^\s*$', np.nan, regex=True)
```

In [17]:
```
#Removing brackets from the cell in the id_artists column
df_tracks["id_artists"]=df_tracks["id_artists"].str.replace("[", "")
df_tracks["id_artists"]=df_tracks["id_artists"].str.replace("]", "")
df_tracks["id_artists"]=df_tracks["id_artists"].str.replace("'", "")

#Replacing blank cells with null so we know to transform them
#Used: https://sparkbyexamples.com/pandas/pandas-replace-blank-values-with-nan/
df_tracks["id_artists"] = df_tracks["id_artists"].replace(r'^\s*$', np.nan, regex=True)
```

In [18]:
```
#Removing brackets from the cell in the genres column
df_artists["genres"]=df_artists["genres"].str.replace("[", "")
df_artists["genres"]=df_artists["genres"].str.replace("]", "")
df_artists["genres"]=df_artists["genres"].str.replace("'", "")

#Replacing blank cells with null so we know to transform them
#Used: https://sparkbyexamples.com/pandas/pandas-replace-blank-values-with-nan/
df_artists["genres"] = df_artists["genres"].replace(r'^\s*$', 'unknown', regex=True)
```

Handling Null Values

As we saw in the previous section there are 13 null values in df_artists's followers column. Since this is such a low number and artists are not very popular overall (as seen below), we decided to replace the null followers values with the average number of followers of artists of the same popularity in the dataset.

In [19]:
```
#define df_fill - dataframe that consists null values from df_artists
df_fill = df_artists[df_artists.isnull().any(axis=1)]
```

Let's take a look at these rows.

In [20]:
```
df_fill
```

Out[20]:

	id	followers	genres	name	popularity
444199	7F71W80jaXFARK7hBjsDl2	NaN	czech pop	Marcell	36
444200	3MLHJz04KmEVzCTPclzkEm	NaN	czech pop	Niko	21
444797	0cqZsULDZdJTGA4Zqh8Ckv	NaN	unknown	Savzilla	0
444798	0BuknWzKujyc9HfZ1V50Uk	NaN	unknown	Duck Doja	0
446635	6ltU5gIDLmWNYaVNHnll5G	NaN	mexican electronic	Zofa	0
446636	7C9nWRMbRqpPUuKh2OEw9n	NaN	unknown	MHV	3
446637	41c30F8zy5UCTSevbn0WfD	NaN	mexican electronic	Broadband Star	0
468525	1DK979aOesiZ4Vkus8txqu	NaN	unknown	AmorArtis Orchestra & Johannes Somary	2
468526	6jkpqSWWsXSuqtsoeAiMDU	NaN	uk americana	Police Dog Hogan	14
468527	7aMdHPv79qOuqqBD6TnaCp	NaN	unknown	Miles Davis & Charlie "Bird" Parker	0
468528	2lr0R5vHGfl0C489h0r6qV	NaN	dc indie	Black Dog Prowl	16
468529	0xkSOleyeTILNlOZKyFgaP	NaN	unknown	Band of the Fifteenth Field Artillery Regiment...	3
468530	4EqqnE0XMAcreVF84QGYJ0	NaN	unknown	Robert Steven Williams	0

In [21]:
```
#function that finds the average followers of all artists with the same popularity
def avg_followers_by_popularity(df, p):
  avg_followers = df[df['popularity'] == p]['followers'].mean()
  return avg_followers

#apply each row to the function above
df_fill['followers'] = df_fill['popularity'].apply(lambda x: round(avg_followers_by_popularity(df_artists, x)))

#check to see that all values are filled
df_fill.head(13)
```

Out[21]:

	id	followers	genres	name	popularity
444199	7F71W80jaXFARK7hBjsDl2	10034	czech pop	Marcell	36
444200	3MLHJz04KmEVzCTPclzkEm	2056	czech pop	Niko	21
444797	0cqZsULDZdJTGA4Zqh8Ckv	53	unknown	Savzilla	0
444798	0BuknWzKujyc9HfZ1V50Uk	53	unknown	Duck Doja	0
446635	6ltU5gIDLmWNYaVNHnll5G	53	mexican electronic	Zofa	0
446636	7C9nWRMbRqpPUuKh2OEw9n	237	unknown	MHV	3
446637	41c30F8zy5UCTSevbn0WfD	53	mexican electronic	Broadband Star	0
468525	1DK979aOesiZ4Vkus8txqu	176	unknown	AmorArtis Orchestra & Johannes Somary	2
468526	6jkpqSWWsXSuqtsoeAiMDU	1034	uk americana	Police Dog Hogan	14
468527	7aMdHPv79qOuqqBD6TnaCp	53	unknown	Miles Davis & Charlie "Bird" Parker	0
468528	2lr0R5vHGfl0C489h0r6qV	1262	dc indie	Black Dog Prowl	16
468529	0xkSOleyeTILNlOZKyFgaP	237	unknown	Band of the Fifteenth Field Artillery Regiment...	3
468530	4EqqnE0XMAcreVF84QGYJ0	53	unknown	Robert Steven Williams	0

Now let's drop the rows that consist null values in df_artists and replace with the newly filled rows.

In [22]:
```
#drop rows with null values
df_artists = df_artists.dropna()

#concatenate df_artists and df_fill
df_artists = pd.concat([df_artists, df_fill])
```

Next, let's address the null vallues in df_tracks.

In [23]:
```
#prints all rows with null values in the dataset
df_tracks[df_tracks.isnull().any(axis=1)]
```

Out[23]:

	id	name	popularity	duration_ms	explicit	artists	id_artists	release_date	danceability	energy	key	loudness
226336	4iH7negBYMfj2z0wDNmgdx	NaN	28	264973	0	NaN	0LyfQWJT6nXafLPZqxe9Of	1994-01-01	0.512	0.578	0	-12.280
510975	04d5kbLvSAIBt3pGcljdhC	NaN	0	184293	0	NaN	0LyfQWJT6nXafLPZqxe9Of	1922-04-01	0.426	0.285	11	-11.970
510976	05tRkgyxVdwMePGqOXMDYU	NaN	0	191587	0	NaN	0LyfQWJT6nXafLPZqxe9Of	1922-04-01	0.344	0.186	0	-13.495
510978	0YAMRgAQH6tkTh4sWNXr8L	NaN	0	191573	0	NaN	0LyfQWJT6nXafLPZqxe9Of	1922-04-01	0.316	0.257	3	-13.611
510979	1K6MQQxmFpPb66ZnaiIpHX	NaN	0	167602	0	NaN	0LyfQWJT6nXafLPZqxe9Of	1922-04-01	0.558	0.283	1	-12.847
...	...	...	...	...	...	...	...	...	...	...	...	...
517206	6OH9mz9aFbGlbf74cBwYWD	NaN	2	209760	0	NaN	0LyfQWJT6nXafLPZqxe9Of	1962-02-01	0.506	0.598	7	-4.672
517215	15RqFDA86slfzujSQMEX4i	NaN	2	257280	0	NaN	0LyfQWJT6nXafLPZqxe9Of	1962-02-01	0.612	0.615	5	-5.609
520127	0hKA9A2JPtFdg0fiMhyjQD	NaN	6	194081	0	NaN	0LyfQWJT6nXafLPZqxe9Of	1974-12-31	0.471	0.369	4	-12.927
525238	1kR4glb7nGxHPl3D2ifs59	NaN	26	289440	0	NaN	0LyfQWJT6nXafLPZqxe9Of	1998-01-05	0.501	0.583	7	-9.460
525265	4xyBq8t4nkUKH8s28j6ZoD	NaN	24	254000	0	NaN	0LyfQWJT6nXafLPZqxe9Of	1998-01-05	0.390	0.452	7	-9.900

71 rows × 20 columns

It looks like the ID of the artist are all the same. Let's groupby id_artists and check.

In [24]:
```
df_tracks[df_tracks.isnull().any(axis=1)].groupby(by='id_artists')['id'].count()
```

Out[24]:
```
id_artists
0LyfQWJT6nXafLPZqxe9Of    71
Name: id, dtype: int64
```

We looked up the artist with this id at https://open.spotify.com/artist/0LyfQWJT6nXafLPZqxe9Of (https://open.spotify.com/artist/0LyfQWJT6nXafLPZqxe9Of) and it appears the artist is 'Various Artists'. We decided to drop these values from df_tracks, because we want to elimate this ambiguity.

In [25]:
```
#drop rows with null values
df_tracks = df_tracks.dropna()
```

Dropping Unused Rows in df_artists

Since artist_df is twice the size of tracks_df, we expect many artists will not be matched to a track. Therefore, let's only keep artists that are in df_track to make our job a little easier since we'll only be needing the song data.

In [26]:
```
#making a list of all the used ids
ids_list = []

for row in df_tracks['id_artists']:
  row_list = row.split(', ')
  ids_list += row_list
```

In [27]:
```
#see how many ids in df_artists are in df_tracks
#True indicates the id is in df_tracks
df_artists['id'].isin(ids_list).value_counts()
```

Out[27]:
```
False    1022966
True       81383
Name: id, dtype: int64
```

We see that this would make a remarkable difference as we only use 10% of the rows in df_artists. Let's only keep the artists that are in df_tracks.

In [28]:
```
#only takes rows where the id is in ids_list (list of ids in df_tracks)
df_artists = df_artists[df_artists['id'].isin(ids_list)]
```

Adding Genre Feature to df_artists

Next, we'll find the broad features to categorize each artist's primary genre. We decided to classify the broad genre based on keywords in each row's genre list, which consists 2-4 very specific genres. We chose to group the 4250 unique genres into the following broad categories: \

- Rap - hip hop and trap songs
- Pop - any song with pop in the name
- Oldies - songs from the 1940s-1950s that are classified by adult standard or easy listening
- R&B - r&b songs
  - This is commonly grouped with rap but we wanted to make it it's own category since the speed and danceability of rap and r&b songs are very different
- Religion - religious songs, mostly gospel and Christian tunes
- Latin - songs based in Central and Southern America
- Jazz/Blues - jazz, blues, groove, etc.
- Folk - folk.. this is more popular then you think, which is why we made it it's own category.
- Children - lullaby, baby, and children's songs. This is also more popular than you would think.
- Dance - EDM, electronic tunes
- Soundtrack - songs that from movie and tv show soundtracks.. this is also more popular than you would think and it was difficult to place these in order categories
- Country - just country music
- Rock - alternative, classic rock, and other popular genres we found to be a part of the rock umbrella
- Classical - songs with musical instruments
- Unknown - songs we did not have the genre data for
- Other - songs that did not match the categories above
  - We chose to seperate Unknown and Other because Other just doesn't match the popular categories we tried to find. However unknown was what we couldn't find the data for.

This was one of the most challeging aspects of the project for several reasons: \

1. Some artists have songs from multiple categories and some songs have multiple categories. This is why the order of the if statements was important. It was also easier to assume all of their songs fall under the category they're known for.
2. Initially we didn't have many broad categories (just Rap, Pop, Country, Classical, and Rock), but as we explored the data and saw how many songs were uncategorized, we recognized the importance of having more broad genres and using more key words in order to place songs in the correct genres. To do this we would find songs that were find mostly commonly uncategorized genres, do a Google search to, and place a key word in the appropriate category. Therefore, we spent a lot of time tweaking this.

In the end, many broad genres still could not be determined, but we did the best we could be classify as many artists's broad genres as possible.

We also tried using a word similarity library to simplify this but it wasn't nearly as fast and effective as what we did here.

In [29]:
```
def find_broad_category(category):
 if 'rap' in category or 'hip hop' in category or 'trap' in category or 'freestyle' in category or 'drill' in categ
   return 'rap'
 elif 'pop' in category:
   return 'pop'
 elif 'adult standards' in category or 'vintage' in category or 'easy listening' in category or 'old' in category:
   return 'oldies'
 elif 'r&b' in category:
   return 'r&b'
 elif 'gospel' in category or 'worship' in category or 'ccm' in category or 'judaica' in category or \
 'christian' in category or 'cathedral' in category:
   return 'religion'
 elif 'latin' in category or 'tango' in category or 'reggae' in category or 'corrido' in category or 'trova' in cat
 'salsa' in category or 'cumbia uruguaya' in category or 'ranchera' in category or 'regional mexican' in category o
 'pagode' in category or 'samba' in category or 'musica' in category or 'cumbia' in category or 'mpb' in category o
 'spanish' in category or 'bossa nova' in category or 'brazilian' in category or 'mexican' in category or 'mariachi
 'espanol' in category or 'sertanejo' in category or 'vallenato' in category or 'merengue' in category or 'cuarteto
   return 'latin'
 elif 'jazz' in category or 'blues' in category or 'groove' in category or\
 'harlem renaissance' in category or 'jug band' in category or 'soul' in category or 'big band' in category or \
 'stride' in category or 'swing' in category or 'soul' in category:
   return 'jazz/blues'
 elif 'folk' in category or 'rebetiko' in category or 'arabesk' in category or 'manele' in category or 'fiddle' in
   return 'folk'
 elif 'children' in category or 'barnsagor' in category or 'baby' in category or 'infant' in category or 'lullaby'
 'barnmusik' in category or 'musiikkia lapsille' in category or 'bornesange' in category or 'barnalog' in category
 'barnemusikk' in category or 'detske pesnicky' in category or 'muziek voor kinderen' in category or 'kids' in cate
   return 'children'
 elif 'electric' in category or 'dance' in category or 'edm' in category or 'electro' in category or 'dub' in categ
 'tech' in category or 'house' in category or 'trance' in category or 'disco' in category or 'funk' in category or
   return 'dance'
 elif 'bollywood' in category or 'filmi' in category or 'kollywood' in category or 'movie' in category or 'disney'
 'soundtrack' in category or 'broadway' in category or 'tollywood' in category or 'hollywood' in category:
   return 'soundtrack'
 elif 'country' in category or 'cowboy' in category or 'bluegrass' in category or 'nashville' in category:
   return 'country'
 elif 'rock' in category or 'metal' in category or 'punk' in category or 'indie' in category or \
 'hardcore' in category or 'alternative' in category or 'mellow gold' in category or 'kindermusik' in category or\
 'kayokyoku' in category or 'dansband' in category:
   return 'rock'
 elif 'classical' in category or 'piano' in category or 'romantic' in category or 'violin' in category or\
 'harp' in category or 'hoerspiel' in category or 'ballroom' in category or 'orchestra' in category or\
 'saxophone' in category or 'instrumental' in category or 'flute' in category or 'brass' in category or \
 'opera' in category or 'fado' in category:
   return 'classical'
 elif 'unknown' in category:
   return 'unknown'
 else:
   return 'other'
```

In [30]:
```
df_artists['broad_genres'] = df_artists['genres'].apply(lambda x: find_broad_category(x))
```

In [31]:
```
df_artists['broad_genres'].value_counts()
```

Out[31]:
```
unknown       25219
pop           16854
rap            7659
other          7234
rock           5008
latin          4510
dance          3396
jazz/blues     2830
classical      2240
folk           1838
soundtrack     1269
oldies         1025
religion        915
country         588
children        535
r&b             263
Name: broad_genres, dtype: int64
```

Let's manually fill some of the most popular artists's broad genres.

In [32]:
```
df_artists[df_artists['broad_genres'] == 'unknown'].sort_values(by='popularity', ascending=False)[0:15]
```

Out[32]:

	id	followers	genres	name	popularity	broad_genres
895644	6PvvGcCY2XtUcSRld1Wilr	166844.0	unknown	Silk Sonic	83	unknown
54721	48HORs6F9P7IgdyKrk4MZC	590066.0	unknown	Tarcísio do Acordeon	81	unknown
59396	4L2dV3zY7RmkeiNO035Fi0	2614.0	unknown	Mufasa & Hypeman	80	unknown
126387	3eVa5w3URK5duf6eyVDbu9	639381.0	unknown	ROSÉ	80	unknown
101141	1PKErrAhYFdfrDymGHRQRo	93596.0	unknown	Nathan Evans	80	unknown
146878	1gALaWbNDnwS2ECV09sn2A	31103.0	unknown	Nightcrawlers	80	unknown
59395	3Edve4VIATi0OZngclQlkN	480.0	unknown	Dopamine	80	unknown
167296	4YYxffPVDFe9XoqqbRW6Bq	165447.0	unknown	L-Gante	79	unknown
59398	5PoZtBo8xZKqPWlZrIDq82	1414.0	unknown	Billen Ted	78	unknown
213154	4Euia7UzdRshy1DJOSMTcs	9496.0	unknown	220 KID	78	unknown
145121	2NfSBtmWe7oPw1EmetJVso	224299.0	unknown	Boza	78	unknown
288010	5H4ylnM5zmHqpKloMNAx4r	158890.0	unknown	Central Cee	78	unknown
147766	3z97WMRi731dCvKkIlf2X6	1230807.0	unknown	NEFFEX	77	unknown
114525	1r1uxoy19fzMxunt3ONAkG	642693.0	unknown	Phoebe Bridgers	77	unknown
33342	3HphLd0XiELTvlPYf55dYC	17849.0	unknown	Strange Fruits Music	77	unknown

In [33]:
```
#fill Silk Sonic's cell
df_artists.loc[895644, 'broad_genres'] = 'rap'
#fill Mufasa & Hypeman's cell
df_artists.loc[59396, 'broad_genres'] = 'dance'
#fill Nightcrawlers's cell
df_artists.loc[146878, 'broad_genres'] = 'dance'
#fill Dopamine's cell
df_artists.loc[59395, 'broad_genres'] = 'dance'
#fill L-Gante's cell
df_artists.loc[167296, 'broad_genres'] = 'latin'
#fill Billen Ted's cell
df_artists.loc[59398, 'broad_genres'] = 'latin'
#fill Central cee's cell
df_artists.loc[288010, 'broad_genres'] = 'rap'
```

In [34]:
```
df_artists[df_artists['broad_genres'] == 'unknown'].sort_values(by='followers', ascending=False)[0:15]
```

Out[34]:

	id	followers	genres	name	popularity	broad_genres
145524	2mxe0TnaNL039ysAj51xPQ	2257404.0	unknown	R. Kelly	72	unknown
147766	3z97WMRi731dCvKkIlf2X6	1230807.0	unknown	NEFFEX	77	unknown
62128	2CQZr2RPZmrcvDnaod1ldC	964160.0	unknown	D.O.	55	unknown
128642	2XzXLjXRSeFtsic4ieyLJy	791547.0	unknown	Wilbur Soot	72	unknown
114525	1r1uxoy19fzMxunt3ONAkG	642693.0	unknown	Phoebe Bridgers	77	unknown
126387	3eVa5w3URK5duf6eyVDbu9	639381.0	unknown	ROSÉ	80	unknown
144671	34v5MVKeQnIo0CWYMbbrPf	638402.0	unknown	John Newman	71	unknown
54721	48HORs6F9P7IgdyKrk4MZC	590066.0	unknown	Tarcísio do Acordeon	81	unknown
114564	4FUMTycjZlEY6ZxMgqNjC8	578907.0	unknown	Tierry	75	unknown
141211	6RGxLsQUoGk5PLyMVwb3yE	444464.0	unknown	Sananda Maitreya	56	unknown
91290	0UdoKjWIe3tHyiqc4qT3Oz	430472.0	unknown	OMFG	57	unknown
53990	6TgVQhHx0BWvzdEQxRnr23	409359.0	unknown	POLLO	49	unknown
976526	6Rx1JKzBrSzoKQtmbVmBnM	387723.0	unknown	Hayley Williams	71	unknown
67245	5uUb3J6HqLhBWwzuh84LUZ	356954.0	unknown	Seventeen	52	unknown
126772	6oMRQ5H3A2XA5I3RG3leni	334876.0	unknown	DJ Alan Gomez	72	unknown

In [35]:
```
#fill R. Kelly's cell
df_artists.loc[145524, 'broad_genres'] = 'rap'
#fill NEFFEX's cell
df_artists.loc[147766, 'broad_genres'] = 'dance'
#fill Wilbur Soot's cell
df_artists.loc[128642, 'broad_genres'] = 'pop'
#fill Phoebe Bridgers's cell
df_artists.loc[114525, 'broad_genres'] = 'rock'
#fill ROSÉ's cell
df_artists.loc[126387, 'broad_genres'] = 'pop'
#fill John Newman's cell
df_artists.loc[144671, 'broad_genres'] = 'pop'
#fill DJ Alan Gomez's cell
df_artists.loc[126772, 'broad_genres'] = 'dance'
```

Rename Columns and Merge df_tracks and df_artists

Now in this section, we'll rename some of the columns to make them more precise and finally join df_tracks and df_artists.

In [36]:
```
df_artists.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 81383 entries, 137 to 444199
Data columns (total 6 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   id            81383 non-null  object 
 1   followers     81383 non-null  float64
 2   genres        81383 non-null  object 
 3   name          81383 non-null  object 
 4   popularity    81383 non-null  int64  
 5   broad_genres  81383 non-null  object 
dtypes: float64(1), int64(1), object(4)
memory usage: 6.4+ MB
```

In [37]:
```
df_tracks.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 586601 entries, 0 to 586671
Data columns (total 20 columns):
 #   Column            Non-Null Count   Dtype  
---  ------            --------------   -----  
 0   id                586601 non-null  object 
 1   name              586601 non-null  object 
 2   popularity        586601 non-null  int64  
 3   duration_ms       586601 non-null  int64  
 4   explicit          586601 non-null  int64  
 5   artists           586601 non-null  object 
 6   id_artists        586601 non-null  object 
 7   release_date      586601 non-null  object 
 8   danceability      586601 non-null  float64
 9   energy            586601 non-null  float64
 10  key               586601 non-null  int64  
 11  loudness          586601 non-null  float64
 12  mode              586601 non-null  int64  
 13  speechiness       586601 non-null  float64
 14  acousticness      586601 non-null  float64
 15  instrumentalness  586601 non-null  float64
 16  liveness          586601 non-null  float64
 17  valence           586601 non-null  float64
 18  tempo             586601 non-null  float64
 19  time_signature    586601 non-null  int64  
dtypes: float64(9), int64(6), object(5)
memory usage: 94.0+ MB
```

In [38]:
```
#Rename columns to make names more precise
df_tracks = df_tracks.rename(columns={'name':'song_name', 'popularity':'song_popularity', 'id':'song_id'})
df_artists = df_artists.rename(columns={'id':'id_artists', 'popularity':'artist_popularity', 'name':'artists_name'}
```

In [39]:
```
#Left merge of df_tracks and df_artists
df_merge = df_tracks.merge(df_artists, how='left', on='id_artists')
```

In [40]:
```
#dropping artists_name column because its a duplicate of artists
df_merge = df_merge.drop(columns=['artists_name'])
```

Reordering the dataframe so that it's that the most features are placed on the left.

In [41]:
```
df_merge = df_merge[['song_name', 'artists', 'release_date', 'broad_genres', 'genres', 'song_popularity', 'artist_p
                     'song_id', 'id_artists', 'duration_ms', 'explicit', 'danceability', 'energy', 'key', 'loudness
                     'mode', 'speechiness', 'acousticness', 'instrumentalness', 'liveness', 'valence', 'tempo', \
                     'time_signature']]
```

Now let's check null values of the new dataset.

In [42]:
```
#Finds total number of null values in each column
df_merge.isna().sum()
```

Out[42]:
```
song_name                 0
artists                   0
release_date              0
broad_genres         117018
genres               117018
song_popularity           0
artist_popularity    117018
followers            117018
song_id                   0
id_artists                0
duration_ms               0
explicit                  0
danceability              0
energy                    0
key                       0
loudness                  0
mode                      0
speechiness               0
acousticness              0
instrumentalness          0
liveness                  0
valence                   0
tempo                     0
time_signature            0
dtype: int64
```

From here, we see that many of the null values have multiple artists. This is because we row in df_artists had one id corresponding to one artists and some rows in df_tracks have multiple ids for multiple artists. In the next section, we'll look at how to handle this.

In [43]:
```
#Rows with null values in the dataset grouped by artists
df_merge[df_merge.isnull().any(axis=1)].groupby(by='artists')['song_id'].count()[0:15]
```

Out[43]:
```
artists
"5nizza", dunkelbunt                                  1
"AKA ChaP in 00S"                                     1
"Aidan ORourke", Kit Downes                           1
"Alan Daveys Eclectic Devils"                         1
"Alex DElia", Nihil Young                             1
"Alexander ONeal", Cherrelle                          1
"Alfa"                                                1
"Alibi Montana feat. Diams", "Diams"                  1
"Alix Combelles Hot Four", Alix Combelle              4
"Amy Hanaialii", Willie K                             1
"AnOm", Vayn                                          1
"Anita ODay", Billy May Orchestra                     1
"Anita ODay", Cal Tjader                              1
"Anita ODay", Gene Krupa & His Orchestra              2
"Anita ODay", Gene Krupa & His Orchestra, Roy Eldridge    2
Name: song_id, dtype: int64
```

Handling Nulls From Rows with Multiple Artists

ITo resolve this issue, we will iteratively extract each individual id where there are multiple and attempt to merge it with df_artists. We will start with the first artist's id, assuming the first artist is the most popular, leading to a more accurate portrayal of how popular the song is. If we can't find a match for the first artist, we'll look at the second artist, third artist and so on until we can no longer find matches. We'll try this for up to 15 artists.

In [44]:
```
print('Init null values')
print(df_merge['followers'].isna().sum())

#Get rows with null values
df_nulls = df_merge[df_merge.isnull().any(axis=1)]
#Extract the first artist's id and place it in a column
df_nulls['artist_id_0'] = df_nulls['id_artists'].apply(lambda x: x.split(', ')[0])
#Find the number of artists for every song
df_nulls['num_artists'] = df_nulls['id_artists'].apply(lambda x: len(x.split(', ')))
#Merge with df_artists
df_nulls = df_nulls.merge(df_artists, how='left', left_on='artist_id_0', right_on='id_artists')
#Drop and rename necessary columns
df_nulls = df_nulls.drop(columns=['genres_x', 'artist_popularity_x', 'followers_x', 'artist_id_0', \
                                  'id_artists_y','artists_name', 'broad_genres_x'])
df_nulls = df_nulls.rename(columns={'id_artists_x':'id_artists', 'followers_y': 'followers', 'genres_y':'genres',\
                                    'artist_popularity_y': 'artist_popularity', 'broad_genres_y': 'broad_genres'})
print('Remaining nulls after Round 1:')
print(df_nulls['followers'].isna().sum())
#Drop nulls and call df_concat
df_concat = df_nulls.dropna()

for i in range(1,15):
  #Get rows with null values
  df_nulls = df_nulls[df_nulls.isnull().any(axis=1)]
  #Extract values wear there is more than i artists
  df_nulls = df_nulls[df_nulls['num_artists'] > i]
  #Extract the next artist's id and place it in a column
  df_nulls['artist_id_{}'.format(i)] = df_nulls['id_artists'].apply(lambda x: x.split(', ')[i])
  #Merge with df_artists
  df_nulls = df_nulls.merge(df_artists, how='left', left_on='artist_id_{}'.format(i), right_on='id_artists')
  #Drop and rename necessary columns
  df_nulls = df_nulls.drop(columns=['genres_x', 'artist_popularity_x', 'followers_x', 'artist_id_{}'.format(i), \
                                    'id_artists_y', 'artists_name', 'broad_genres_x'])
  df_nulls = df_nulls.rename(columns={'id_artists_x':'id_artists', 'followers_y': 'followers', 'genres_y':'genres',
                                      'artist_popularity_y': 'artist_popularity', 'broad_genres_y': 'broad_genres'})
  print('Remaining nulls after Round {}:'.format(i+1))
  print(df_nulls['followers'].isna().sum())
  df_updated = df_nulls.dropna()
  df_concat = pd.concat([df_concat, df_updated])
```

```
Init null values
117018
Remaining nulls after Round 1:
12344
Remaining nulls after Round 2:
1001
Remaining nulls after Round 3:
219
Remaining nulls after Round 4:
51
Remaining nulls after Round 5:
29
Remaining nulls after Round 6:
23
Remaining nulls after Round 7:
16
Remaining nulls after Round 8:
1
Remaining nulls after Round 9:
1
Remaining nulls after Round 10:
1
Remaining nulls after Round 11:
1
Remaining nulls after Round 12:
1
Remaining nulls after Round 13:
1
Remaining nulls after Round 14:
0
Remaining nulls after Round 15:
0
```

It looks like we were able to match all 117018 columns in the list after 15 iterations. Now let's remove all the null values from df_merge and concatenate these newly filled rows.

In [45]:
```
#Drop all nulls from df_merge
df_merge = df_merge.dropna()

#rearrange column order so we can concat
df_nulls = df_nulls[list(df_merge.columns)]

df = pd.concat([df_merge, df_nulls])
```

Now, let's check to ensure all nulls were dropped.

In [46]:
```
df.isna().sum()
```

Out[46]:
```
song_name            0
artists              0
release_date         0
broad_genres         0
genres               0
song_popularity      0
artist_popularity    0
followers            0
song_id              0
id_artists           0
duration_ms          0
explicit             0
danceability         0
energy               0
key                  0
loudness             0
mode                 0
speechiness          0
acousticness         0
instrumentalness     0
liveness             0
valence              0
tempo                0
time_signature       0
dtype: int64
```

Dropping Song Duplicates and Other Values

Let's drop the rows where the artist is Workout Music and Various Artists because it is too obscure for our model.

In [47]:
```
df = df[(df['artists'] != 'Workout Music') & (df['artists'] != 'Various Artists')]
```

We noticed that some of the same songs by the same artists had duplicates. For instance:

In [48]:
```
df[df['song_name'] == "Born This Way"]
```

Out[48]:

	song_name	artists	release_date	broad_genres	genres	song_popularity	artist_popularity	followers	song_id	id_
84133	Born This Way	Lady Gaga	2011-05-23	pop	dance pop, pop, post-teen pop	74	89.0	17047864.0	30XU4suKzCeoCK9YFzdufg	1HY2Jd0NmPuamShAr6
303535	Born This Way	Lady Gaga	2011-01-01	pop	dance pop, pop, post-teen pop	69	89.0	17047864.0	6r2BECwMgEoRb5yLfp0Hca	1HY2Jd0NmPuamShAr6
303672	Born This Way	Lady Gaga	2011-01-01	pop	dance pop, pop, post-teen pop	53	89.0	17047864.0	6aDi4gOE2Cfc6ecynvP81R	1HY2Jd0NmPuamShAr6
353017	Born This Way	Lady Gaga	2021-04-13	pop	dance pop, pop, post-teen pop	0	89.0	17047864.0	0n5iwOwzox8e9mjMhG7Rea	1HY2Jd0NmPuamShAr6
353018	Born This Way	Lady Gaga	2021-04-13	pop	dance pop, pop, post-teen pop	0	89.0	17047864.0	1RTbgb0BantdgL67BFFCrT	1HY2Jd0NmPuamShAr6
558798	Born This Way	Lady Gaga	2011-01-01	pop	dance pop, pop, post-teen pop	47	89.0	17047864.0	6lEQQ4PJNJrF2Sp9AjaC67	1HY2Jd0NmPuamShAr6

To resolve this, let's sort the entire dataset in descending order of song popularity and remove all song duplicates (keeping the first instance that is the most popular).

In [49]:
```
print('Before duplicates dropped', len(df))
```

```
Before duplicates dropped 469583
```

In [50]:
```
df = df.sort_values(by='song_popularity', ascending=False)
#Remove duplicates based on the song and artist
df = df.drop_duplicates(subset=['song_name','artists'], keep='first')
```

In [51]:
```
print('After duplicated dropped', len(df))
```

```
After duplicated dropped 419823
```

Adding New Features

Let's add the feature year and remove release date so we can more easily group songs together.

In [52]:
```
#Adding year column, changing the values from a string to integer, and dropping release date
df['year'] = df['release_date'].apply(lambda x: x[0:4])
df['year'] = df['year'].astype(int)
df = df.drop(columns=['release_date'])
```

We make a copy of df for later use.

In [53]:
```
#copy df for recs
df_copy = df
```

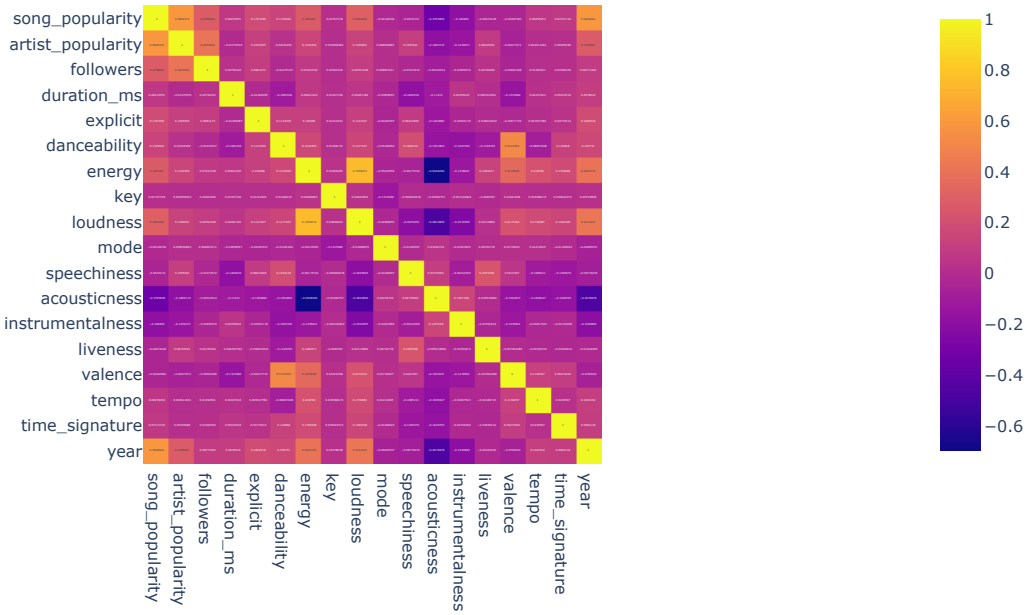
Section 4: Visualizations

In this next section, we'll look at visualizations to see the make up of the dataset and use the data to answer important questions, such as recent trends and an analysis of the top 10 songs from every year since 2010.

Correlation Matrix + Pair plot

Let's see how the features are correlated with eachother.

In [54]:
```
#Plotting the correlation matrix
fig = px.imshow(df.corr(), text_auto=True)
fig.show()
```

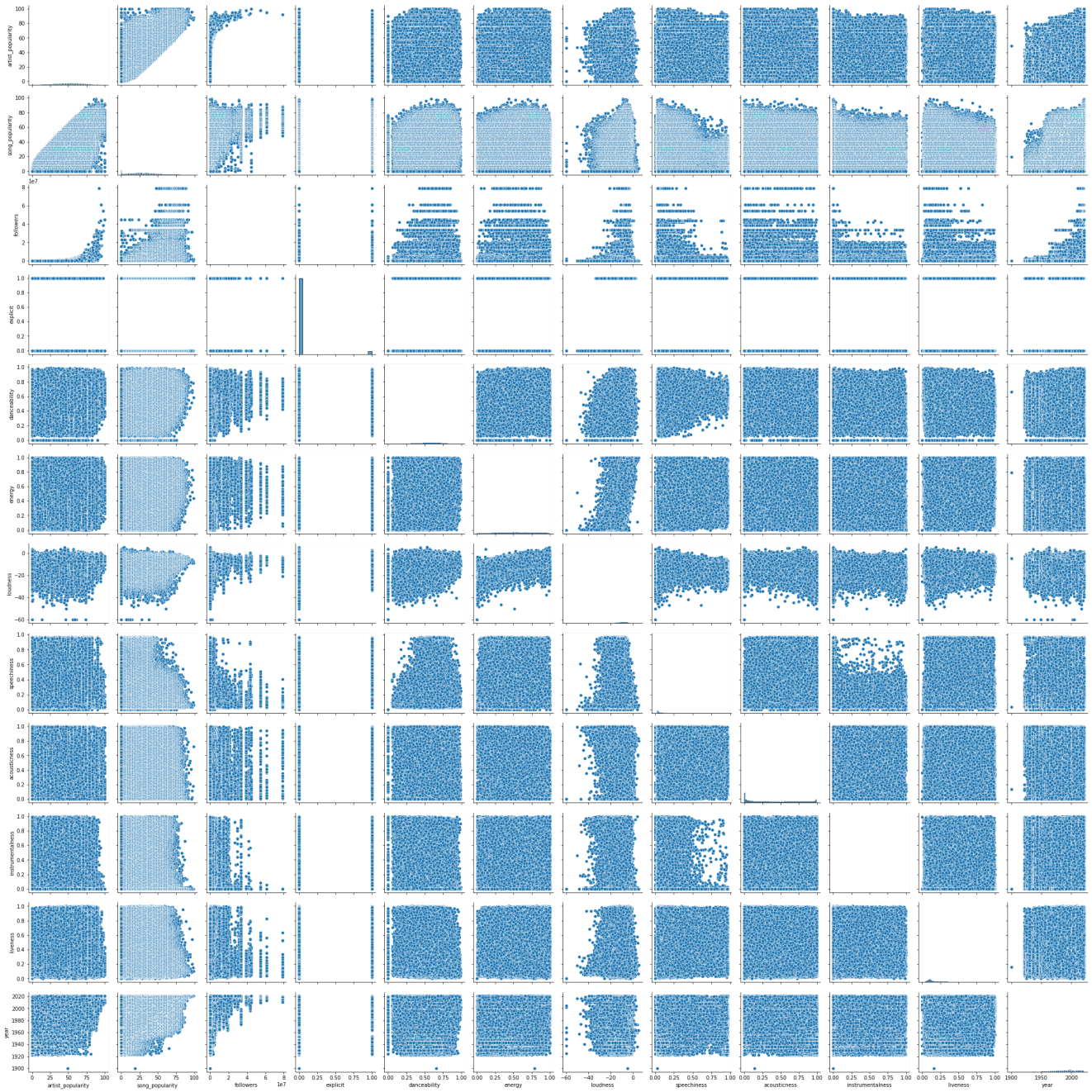


It looks like song_popularity is most positively correlated with the artist's popularity and the year is was released and most negatively correlated with the instrumentalness and acousticness of a song.

Now let's create a pairplot, which will allow us to see the relationship between each variable. Let's focus on seeing the relationship between 'artist_popularity', 'song_popularity', 'followers', 'explicit', 'danceability', 'energy', 'loudness', 'speechiness', 'acousticness', 'instrumentalness', 'liveness', and 'year'. Hopefully we will see these relationships despite the amount of data we have.

In [55]:
```
#Reseting the index of the df and check to make sure all index values are unique
df = df.reset_index(drop=True)
df.index.is_unique

#Creating pairplot
sns.pairplot(df[['artist_popularity', 'song_popularity', 'followers', 'explicit',
                 'danceability', 'energy', 'loudness', 'speechiness', 'acousticness',
                 'instrumentalness', 'liveness', 'year']])
```

Out[55]: <seaborn.axisgrid.PairGrid at 0x28afa1690>



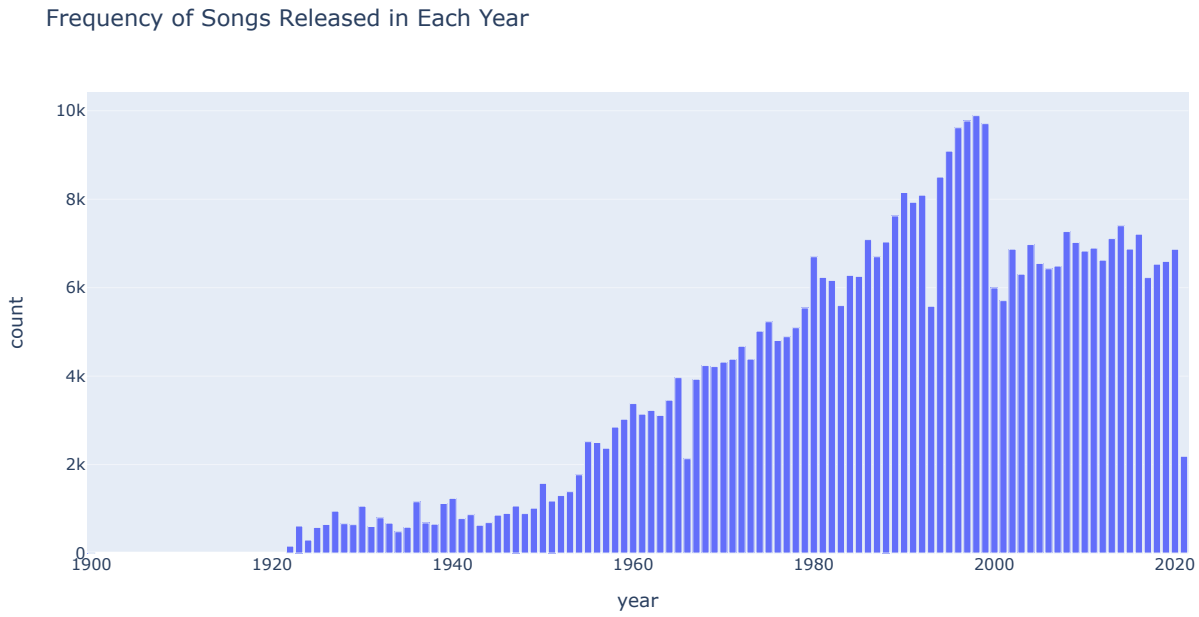
It's a little difficult to see some of these relationships but here are a couple interesting ones to point out:

1. There appears to be a fairly strong relationship between linear relationship between song_popularity and artist_popularity.
2. There is strong relationship between artist_popularity and followers.
3. Loudness and energy of a song seem to be positively related.

Visualizations Relating to Year

Now, let's look at the distribution of the years the songs in the dataset were released.

In [56]:
```
#Creating a df called year_dist that groups songs by year and counts the total number of songs in each year
year_dist = df.groupby(by='year')['song_id'].count().reset_index(name='count')

#Bar plot of year vs song frequency
fig = px.bar(year_dist, x='year', y='count', title='Frequency of Songs Released in Each Year')
fig.show()
```



It looks like most of the songs in our dataset were released in the late 90s. It is kind of surprising to see so many songs in the late 90s in comparison to the early 2000s.

Most Popular Artists and Songs in the Dataset

Now, let's see which artists have the most songs in the dataset.

In [57]:
```
#extract the first artist's id in the case that there are multiple artists featured on a song (assuming here that t
df['first_artist_id'] = df['id_artists'].str.split(',').str[0]

#extract first artist's name
df['first_artist'] = df['artists'].str.split(',').str[0]

#group by first artist's id and count the number of songs by each and sort them in descending order
top_w_features_df = df.groupby(by='first_artist_id')['song_id'].count().reset_index(name='count').sort_values(by='c

#merge the artist's id with their name and just keep the name and count columns
top_df = top_w_features_df.merge(df, on='first_artist_id')[['first_artist', 'count']].drop_duplicates()

#print the df where artists have more than 150 songs in the dataset
top_df[top_df['count'] > 150][0:20]
```

Out[57]:

	first_artist	count
0	Die drei ???	3856
3856	TKKG Retro-Archiv	2006
5862	Benjamin Blümchen	1485
7347	Bibi Blocksberg	1440
8787	Lata Mangeshkar	1086
9873	Bibi und Tina	900
10773	Francisco Canaro	866
11639	Tadeusz Dolega Mostowicz	838
12477	Fünf Freunde	812
13289	Ella Fitzgerald	770
14059	Die Originale	638
14697	Elvis Presley	628
15325	Frank Sinatra	626
15951	Mohammed Rafi	585
16536	Globi	582
17118	Ignacio Corsini	504
17622	Pink Floyd	469
18091	Bob Dylan	455
18546	The Rolling Stones	438
18984	Johnny Cash	434

Here, we see that the artists with the most songs in the dataset include Die drei ???, TKKG Retro-Archiv, Benjamin Blümchen, and Bibi Blocksberg, many artists I have never heard of. However, seeing artists like Bob Dylan, The Rolling Stones, and Pink Floyd isn't very suprising.

Now let's look at the most common artists that released songs in the last 10 years (from 2012 to now)

In [58]:
```
#Extracting songs from the year 2012 and onwards, counting the frequency for each artist and sorting in descending
top_w_features_df = df[df['year'] > 2011].groupby(by='first_artist_id')['song_id'].count().reset_index(name='count'

#Merging the artist's name with their name and dropping duplicates
top_recent_df = top_w_features_df.merge(df, on='first_artist_id')[['first_artist', 'count', 'first_artist_id']].dro

top_recent_df[0:20]
```

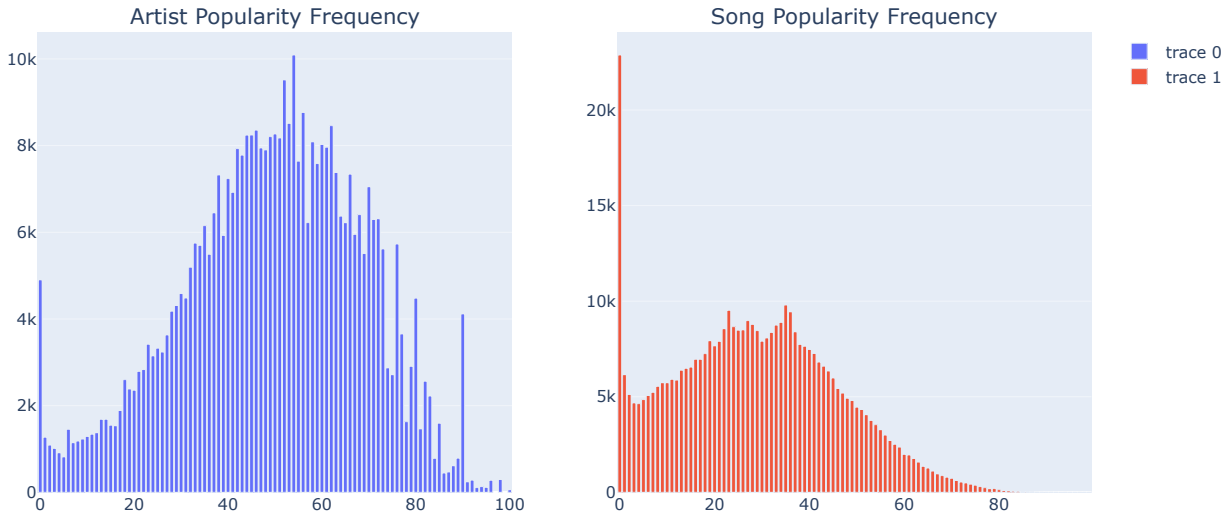
Out[58]:

	first_artist	count	first_artist_id
0	Armin van Buuren	237	0SfsnGyD8FpIN4U4WCkBZ5
257	BTS	173	3Nrfpe0tUJi4K4DXYWgMUX
430	Taylor Swift	114	06HL4z0CvFAxyc27GXpf02
603	Pink Floyd	107	0k17h0D3J5VfsdmQ1iZtE9
1072	Jul	98	3IW7ScrzXmPvZhB27hmfgy
1170	One Direction	91	4AK6F7OLvEQ5QYCBNiQWHq
1267	The Beta Band	81	2LMR8u7DOMF0FBseDpTsRa
1349	Jorge & Mateus	77	1elUiq4X7pxej6FRlrEzjM
1475	Lana Del Rey	75	00FQb4jTyendYWaN8pK0wa
1552	Paul Carrack	75	0FFuvdY7fuiuTmHN9unYoz
1631	Eyal Golan	75	54jZWpivOTllo1afYNSx5U
1789	Wild Stylerz	69	1ZY4523hJlHABsolGRelwr
1858	TWICE	69	7n2Ycct7Beij7Dj7meI4X0
1927	PNL	68	3NH8t45zOTqzlZgBvZRjvB
1995	Taco Hemingway	66	7CJgLPEqilRuneZSolpawQ
2061	The Weeknd	66	1Xyo4u8uXC1ZmMpatF05PJ
2127	Adam Gardner	62	7GbMHSUkqSEF3eeL4x7Nuy
2189	Arijit Singh	60	4YRxDV8wJFPHPTeXepOstw
2249	Drake	60	3TVXtAsR1Inumwj472S9r4
2329	The Cars	60	6DCIj8jNaNpBz8e5oKFPtp

Looking at the top 20 artists with the most hits since 2012, results are not that suprising, especially Drake, One Direction, Taylor Swift, and The Weeknd that have had really long musical careers and have consistently released music since 2012.

Now, let's see the distribution of song and artist popularity in the dataset. It would be really nice to see an even distribution of both but I'm expecting to see most artists and songs with a moderate popularity (between 30 and 70).

In [59]:
```
larity and counting the total songs
f.groupby(by='artist_popularity')['song_id'].count().reset_index(name='count').sort_values(by='count', ascending=False)

rity and counting the total songs
groupby(by='song_popularity')['song_id'].count().reset_index(name='count').sort_values(by='count', ascending=False)

 can see the distribution of artist and song popularity side by side
com/python/subplots/
=1, cols = 2, subplot_titles=['Artist Popularity Frequency', 'Song Popularity Frequency'])

tist_popularity_df['artist_popularity'].to_list(),
tist_popularity_df['count'].to_list()),
1, col=1)

larity_df['song_popularity'].to_list(),
larity_df['count'].to_list()),
2, 1)
```



As a result, we can see the artist popularity is almost distributed as expected since most artists have a popularity between 30 and 80. However, I expected to see more popular songs in this dataset. What was most shocking was to see how to artists and especially songs have a popularity of 0. We will drop the rows with a song_popularity of 0 later on.

Now let's see what the most popular songs are in the dataset.

In [60]:
```
#Sorting by song_popularity and printing the top 15
df.sort_values(by='song_popularity', ascending=False)[0:15][['song_name', 'artists', 'genres', 'song_popularity', '
```

Out[60]:

	song_name	artists	genres	song_popularity	year	artist_popularity
0	drivers license	Olivia Rodrigo	pop, post-teen pop	99	2021	88.0
1	Astronaut In The Ocean	Masked Wolf	australian hip hop	98	2021	85.0
2	Save Your Tears	The Weeknd	canadian contemporary r&b, canadian pop, pop	97	2020	96.0
3	telepatía	Kali Uchis	colombian pop, pop	97	2020	88.0
4	Blinding Lights	The Weeknd	canadian contemporary r&b, canadian pop, pop	96	2020	96.0
5	The Business	Tiësto	big room, brostep, dance pop, dutch edm, edm, ...	95	2020	87.0
6	Heartbreak Anniversary	Giveon	pop, r&b	94	2020	91.0
7	WITHOUT YOU	The Kid LAROI	australian hip hop	94	2020	90.0
8	Streets	Doja Cat	dance pop, pop, pop rap	94	2019	91.0
9	Good Days	SZA	pop, pop rap, r&b	93	2020	88.0
10	positions	Ariana Grande	pop, post-teen pop	92	2020	95.0
11	Hecha Pa' Mi	Boza	unknown	92	2020	78.0
12	Up	Cardi B	pop, pop rap, post-teen pop, rap	92	2021	90.0
13	Hold On	Justin Bieber	canadian pop, pop, post-teen pop	92	2021	100.0
14	Watermelon Sugar	Harry Styles	pop, post-teen pop	92	2019	90.0

I'm not shocked to see these results since these were very popular song in 2021. Notice from these results how all of the songs were released in the past couple of years and the artists are all very popularity (all have popularity score of over 80).

Now most popular artists by the dataset's artist_popularity.

In [61]:
```
#Groupby artists, take mean popularity, and sort in descending order
df.groupby(by='first_artist')[['artist_popularity']].mean().reset_index().sort_values(by='artist_popularity', ascen
```

Out[61]:

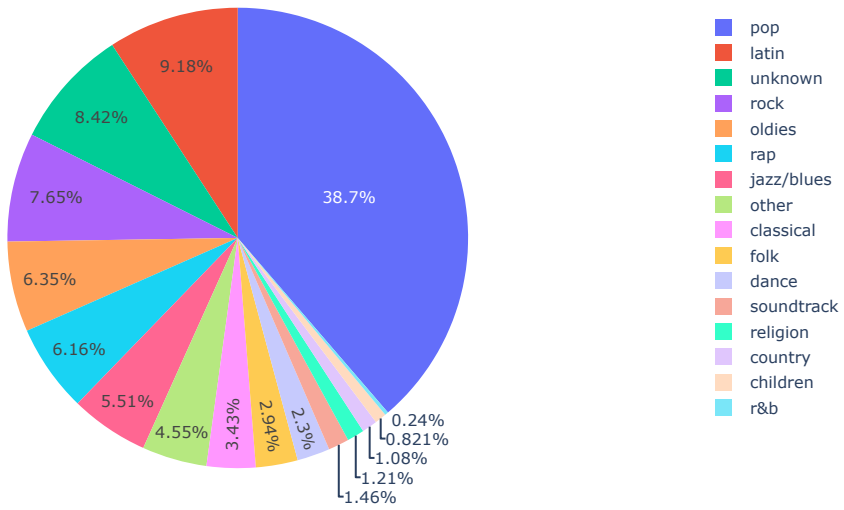
	first_artist	artist_popularity
23806	Justin Bieber	100.0
12854	Drake	98.0
45643	Taylor Swift	98.0
4849	Bad Bunny	98.0
47746	The Weeknd	96.0
23567	Juice WRLD	96.0
4728	BTS	96.0
12932	Dua Lipa	95.0
3814	Ariana Grande	95.0
21084	J Balvin	95.0
33423	Myke Towers	95.0
48847	Travis Scott	94.0
14320	Eminem	94.0
10615	DaBaby	93.0
35926	Ozuna	93.0

These results are not shocking all, considering these were the most popular artists in 2021. I am a little surprised to see how popular Juice WRLD still is since he passed away in 2019. I wonder if this influenced his popularity in any way?

Genre Visualizations

Now let's look at the most popular categories. First, let's look at broad categories using a pie chart.

In [62]:
```
#Making a list of all broad categories
broad_genres = []
for row in df['broad_genres']:
  row_list = row.split(', ')
  broad_genres += row_list

#Count the occurence of each genre and create a df showing each genre and its frequency
top_broad_genres = Counter(broad_genres).most_common(50)
df_top_broad_genres = pd.DataFrame(top_broad_genres, columns=['genre', 'frequency'])

fig = go.Figure(data=[go.Pie(labels=df_top_broad_genres['genre'], values=df_top_broad_genres['frequency'])])
fig.show()
```



It looks like our most popular category (making up 38.8% of the dataset) is pop. Songs classified as other are also popular, along with rock, unknown, latin, and classical. Let me note again here that the difference between songs that are unknown and other is that songs that are 'unknown' did not consist any information about the genre in the raw dataset. Songs that are classified as 'other' had information in the dataset but we were unable to place them in a broad category. We decided to keep these two categories separate because we didn't want to group together songs that could potentially be classified in other categories with songs that clearly could not be.

Now, see what the most frequently occuring (non-broad) categories are. Since most songs fall under multiple categories, we'll use a word map to show which ones appear the most frequently.

In [63]:
```
%%time
#Create a list of all the genres
genres = []
for row in df['genres']:
  row_list = row.split(', ')
  genres += row_list

#Join the genres as a string and create the word cloud
genres_str = (" ").join(genres)
genres_wordcloud = wordcloud.WordCloud(width = 1000, height = 800, background_color='white').generate(genres_str)

plt.figure(figsize=(20,8))
plt.imshow(genres_wordcloud)
```

```
CPU times: user 5.13 s, sys: 308 ms, total: 5.43 s
Wall time: 5.44 s
```

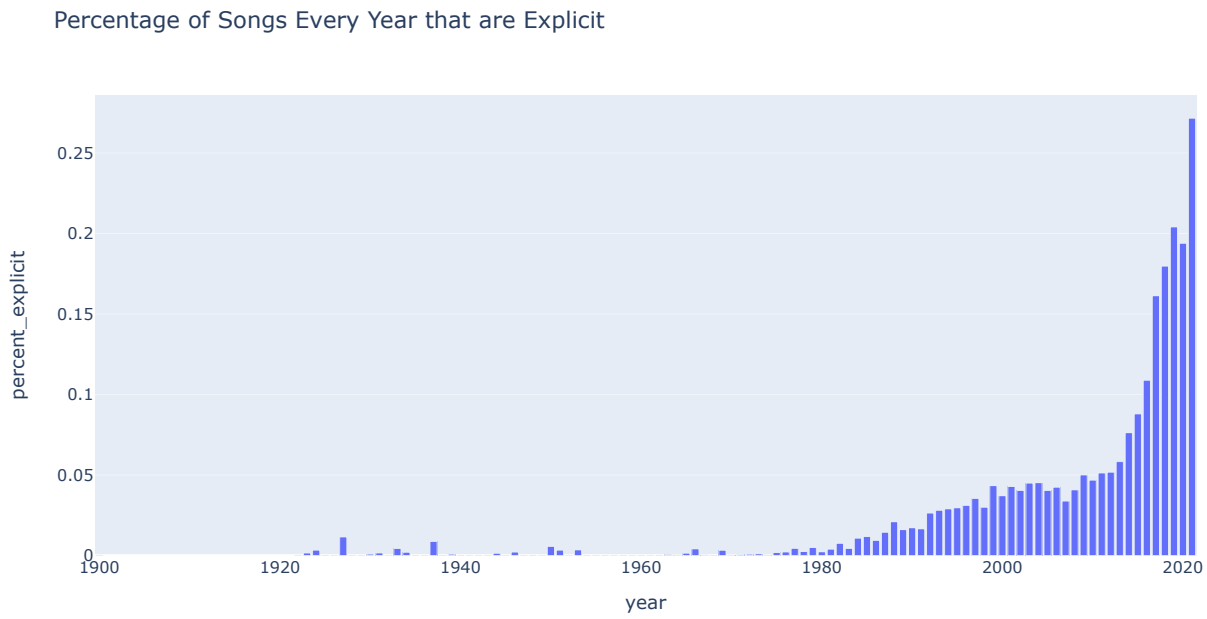
Out[63]: <matplotlib.image.AxesImage at 0x2e5cf5e10>

It looks like hip-hop, classic rock, adult standards, and pop are very frequently occuring.

Have Explicit Lyrics Become More Common Over Time?

Our next visualization will plot the proportion of songs in the dataset are 'explicit' every year. We expect this to steadily increase.

In [64]:
```
# songs that are explicit and not explicit, group them by year, and count their frequency
df = df[df['explicit'] == 1].groupby('year')['song_id'].count().reset_index().rename(columns={'song_id':'explicit'})
non_explicit_df = df[df['explicit'] == 0].groupby('year')['song_id'].count().reset_index().rename(columns={'song_id':'non-explicit'})

# together to make it convenient for plotting
explicit_df.merge(non_explicit_df, how='outer', on='year')

# 80
exp_join.fillna(0)

# 'percent_explicit' that is the proportion of explicit vs. non-explicit songs
exp_join['percent_explicit'] = round(exp_join['explicit'] / (exp_join['explicit'] + exp_join['non-explicit']), 5)

# ion of explicit songs for every year
px.line(exp_join, x="year", y="percent_explicit",
itle='Percentage of Songs Every Year that are Explicit')
```

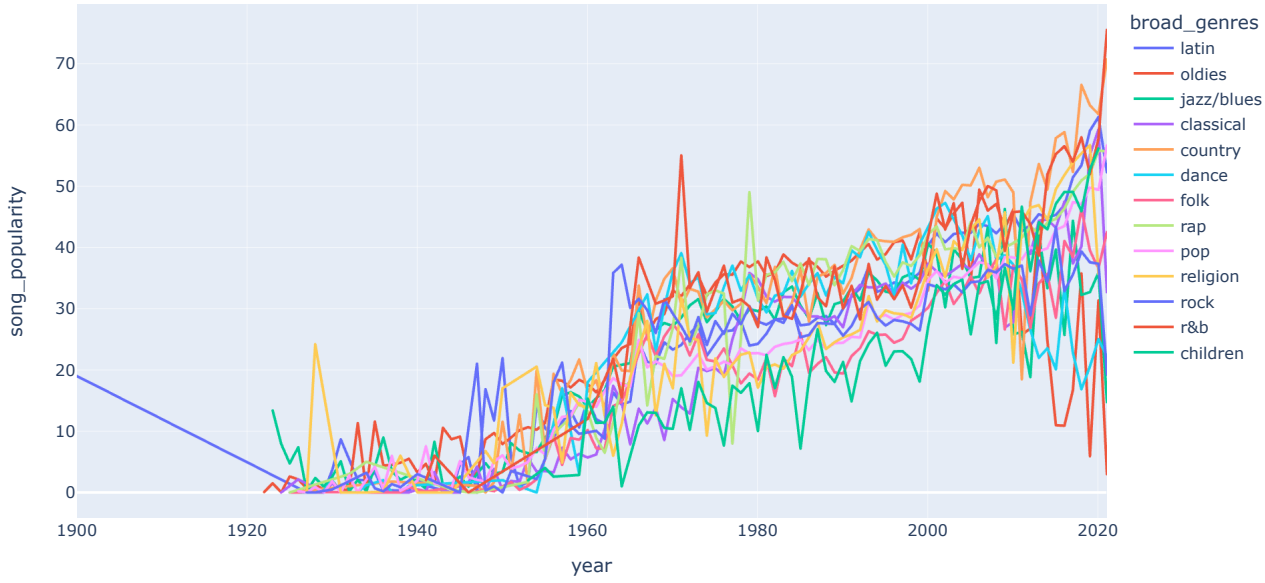


Wow. It appears the proportion of explicit vs non-explicit songs steadily increased starting from the 80's to the early 2010's but then increased dramatically after 2014/2015. I find it very shocking that more than a quarter of the songs released in 2021 were explicit. One possible explanation for this could be maybe that certain genres started becoming more popular or songs with more abrusive language became more liked (or possibly a combination of both).

What Genres Have Become More Popular Over the Years?

Let's see what genres have become more popular over the years first looking at the mean song_popularity of every genre and then looking at the frequency of just pop, rock, rap, and country songs.

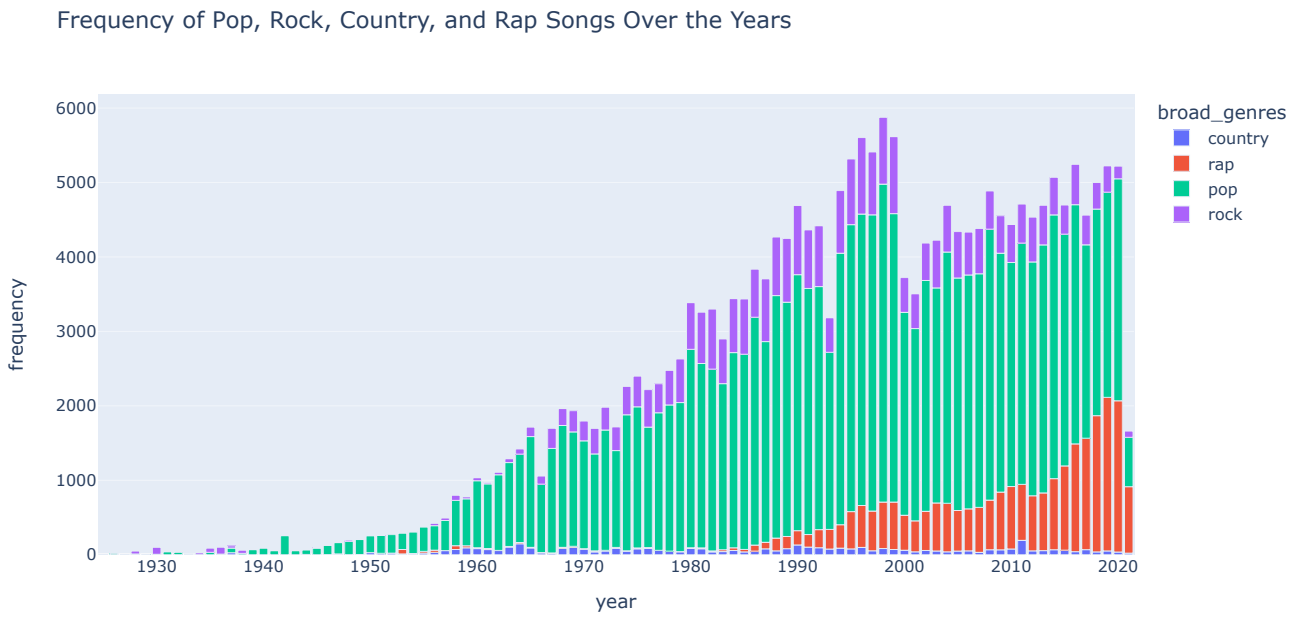
In [65]:
```
#Grouping by year and genre and taking the average song_popularity
average_popularity_df = df.groupby(['year', 'broad_genres'])['song_popularity'].mean().reset_index()

#Excluding the categories other, unknown, and soundtrack
average_popularity_df = average_popularity_df[(average_popularity_df['broad_genres'] != 'other') & \
                                              (average_popularity_df['broad_genres'] != 'unknown') & \
                                              (average_popularity_df['broad_genres'] != 'soundtrack')
                                              ]

#Plot average song popular for each genre in each year
fig = px.line(average_popularity_df, x="year", y="song_popularity", color='broad_genres')
fig.show()
```



It looks like country, latin, r&b, and rap have the highest mean song_popularity. We're very surprised to see country, but I'm not as suprised by the others.

Now, let's focus on just comparing the frequencies of songs in pop, rock, country, and rap over the years. I have a feeling we'll see that rap songs have become more frequent and rock songs have decreased.

In [66]:
```
#Filter only pop, rock, country, and rap songs and then groupby category and year and rename the frequency column
frequency_df = df[(df['broad_genres'] == 'pop') | (df['broad_genres'] == 'rock') | (df['broad_genres'] == 'country'
frequency_df = frequency_df.rename(columns={'song_id':'frequency'})

#Plot each year's number of the songs and how many belong to each genre. Hover over the plot to see more details ab
fig = px.bar(frequency_df, x="year", y='frequency', color='broad_genres', title="Frequency of Pop, Rock, Country, a
fig.show()
```



Notice much the frequency of rap songs increased over the years since the 1990's in comparison to the other genres. Also, look at how the frequency of rock songs steadily decreased. My predictions were correct although a expected more of a drastic increase in rap and decrease in rop. It appears pop hasn't changed very much and if anything has become more steadily popular. I'm also a little suprised to see that country songs are hardly popular in total makeup given their mean song popularity in the previous chart. It must be that country songs aren't released as often as the other categories, but when they are released they're usually very popular.

Analysis of Top 10 Most Popular Songs of Every Year Since 2010

Let's take a look at the most popular songs since 2010!

In [67]:
```
#Extracting songs that were released after 2009
recent_df = df[df['year'] > 2009]

#Making a list of every year after 2009
years_list = recent_df.year.unique().tolist()

#Removing 2010 from the list
years_list.remove(2010)

#Filter songs only released in 2010 and take the top 10 most popular after sorting them in order of popularity
concat_df = recent_df[recent_df['year'] == 2010].sort_values(by='song_popularity', ascending=False).head(10)

#Do this for all other years and concatenate the df from each year with those from the previous years
for year in years_list:
  temp_df = recent_df[recent_df['year'] == year].sort_values(by='song_popularity', ascending=False).head(10)
  concat_df = pd.concat([temp_df, concat_df], axis=0)

#Rearranging the columns so they're easier to visualize
concat_df = concat_df[['year', 'song_name', 'artists', 'broad_genres', 'song_popularity', 'artist_popularity']]
```

Now, let's see the most top 10 most popular songs every year since 2010. We used the display function from the iPython library to make to make the tables were easier to visualize. I'm getting nostalgic just seeing some of these songs ...

In [68]:
```
#Year by year print the df with the songs sorted in order to song popularity
for year in sorted(recent_df.year.unique().tolist()):
  print("Year: ", year)
  temp = concat_df[concat_df.year == year].sort_values(by='song_popularity', ascending=False).head(10)
  display(temp)
  print( )
  print( )
```

```
Year:  2010
```

	year	song_name	artists	broad_genres	song_popularity	artist_popularity
137	2010	Hey, Soul Sister	Train	pop	84	78.0
253	2010	Just the Way You Are	Bruno Mars	pop	83	93.0
257	2010	Talking to the Moon	Bruno Mars	pop	83	93.0
252	2010	TiK ToK	Kesha	pop	83	81.0
283	2010	I Can't Handle Change	Roar	unknown	82	69.0
289	2010	Please Don't Go	Mike Posner	rap	82	77.0
465	2010	Not Afraid	Eminem	rap	80	94.0
476	2010	Dynamite	Taio Cruz	rap	80	75.0
475	2010	The Spins	Mac Miller	rap	80	87.0
455	2010	Super Bass	Nicki Minaj	rap	80	90.0

```
Year:  2011
```

	year	song_name	artists	broad_genres	song_popularity	artist_popularity
113	2011	Pumped Up Kicks	Foster The People	pop	85	78.0
232	2011	Tongue Tied	Grouplove	pop	83	73.0
371	2011	A Thousand Years	Christina Perri	pop	81	75.0
374	2011	Work Out	J. Cole	rap	81	87.0
375	2011	Paradise	Coldplay	pop	81	89.0
443	2011	Someone Like You	Adele	pop	80	85.0
721	2011	Love You Like A Love Song	Selena Gomez & The Scene	rap	79	71.0
912	2011	Rolling in the Deep	Adele	pop	78	85.0
934	2011	Levels - Radio Edit	Avicii	pop	78	85.0
967	2011	What the Hell	Avril Lavigne	pop	77	80.0

```
Year:  2012
```

	year	song_name	artists	broad_genres	song_popularity	artist_popularity
114	2012	Can't Hold Us - feat. Ray Dalton	Macklemore & Ryan Lewis	rap	85	78.0
130	2012	Locked out of Heaven	Bruno Mars	pop	85	93.0
115	2012	When I Was Your Man	Bruno Mars	pop	85	93.0
169	2012	What Makes You Beautiful	One Direction	pop	84	90.0
287	2012	Classic	MKTO	rap	82	73.0
296	2012	Bubblegum Bitch	MARINA	pop	82	78.0
310	2012	Demons	Imagine Dragons	rock	82	89.0
317	2012	Everybody Talks	Neon Trees	pop	82	73.0
324	2012	Feel So Close - Radio Edit	Calvin Harris	pop	82	87.0
469	2012	Whistle	Flo Rida	rap	80	83.0

```
Year:  2013
```

	year	song_name	artists	broad_genres	song_popularity	artist_popularity
24	2013	Sweater Weather	The Neighbourhood	pop	90	86.0
74	2013	Do I Wanna Know?	Arctic Monkeys	dance	87	87.0
72	2013	All of Me	John Legend	pop	87	83.0
81	2013	Why'd You Only Call Me When You're High?	Arctic Monkeys	dance	86	87.0
119	2013	Wake Me Up	Avicii	pop	85	85.0
160	2013	All I Want	Kodaline	pop	84	76.0
171	2013	Counting Stars	OneRepublic	pop	84	84.0
176	2013	Story of My Life	One Direction	pop	84	90.0
234	2013	Another Love	Tom Odell	pop	83	75.0
295	2013	Safe And Sound	Capital Cities	pop	82	69.0

```
Year:  2014
```

	year	song_name	artists	broad_genres	song_popularity	artist_popularity
98	2014	The Nights	Avicii	pop	86	85.0
108	2014	No Role Modelz	J. Cole	rap	85	87.0
107	2014	Photograph	Ed Sheeran	pop	85	92.0
172	2014	Thinking out Loud	Ed Sheeran	pop	84	92.0
173	2014	Take Me To Church	Hozier	pop	84	82.0
197	2014	Night Changes	One Direction	pop	83	90.0
202	2014	Summer	Calvin Harris	pop	83	87.0
203	2014	Stay With Me	Sam Smith	pop	83	88.0
273	2014	Shower	Becky G	rap	82	82.0
281	2014	Sugar	Maroon 5	pop	82	91.0

```
Year:  2015
```

	year	song_name	artists	broad_genres	song_popularity	artist_popularity
60	2015	Daddy Issues	The Neighbourhood	pop	87	86.0
73	2015	The Hills	The Weeknd	pop	87	96.0
84	2015	The Less I Know The Better	Tame Impala	other	86	83.0
88	2015	Stressed Out	Twenty One Pilots	rock	86	86.0
89	2015	Electric Love	BØRNS	pop	86	75.0
178	2015	Love Yourself	Justin Bieber	pop	84	100.0
187	2015	Play Date	Melanie Martinez	pop	84	82.0
260	2015	Can't Feel My Face	The Weeknd	pop	82	96.0
264	2015	Ride	Twenty One Pilots	rock	82	86.0
265	2015	Wait a Minute!	WILLOW	pop	82	75.0

```
Year:  2016
```

	year	song_name	artists	broad_genres	song_popularity	artist_popularity
39	2016	goosebumps	Travis Scott	rap	88	94.0
71	2016	Say You Won't Let Go	James Arthur	pop	87	86.0
62	2016	Train Wreck	James Arthur	pop	87	86.0
83	2016	Devil Eyes	Hippie Sabotage	rap	86	77.0
133	2016	Runaway	AURORA	pop	84	79.0
134	2016	That's What I Like	Bruno Mars	pop	84	93.0
135	2016	Line Without a Hook	Ricky Montgomery	pop	84	76.0
136	2016	Redbone	Childish Gambino	rap	84	81.0
240	2016	Heathens	Twenty One Pilots	rock	83	86.0
251	2016	Mr Loverman	Ricky Montgomery	pop	83	76.0

Year: 2017

	year	song_name	artists	broad_genres	song_popularity	artist_popularity
50	2017	Believer	Imagine Dragons	rock	88	89.0
66	2017	Perfect	Ed Sheeran	pop	87	92.0
67	2017	Jocelyn Flores	XXXTENTACION	rap	87	92.0
59	2017	Shape of You	Ed Sheeran	pop	87	92.0
120	2017	Thunder	Imagine Dragons	rock	85	89.0
125	2017	Everybody Dies In Their Nightmares	XXXTENTACION	rap	85	92.0
138	2017	XO Tour Llif3	Lil Uzi Vert	rap	84	91.0
175	2017	Sign of the Times	Harry Styles	pop	84	90.0
228	2017	Look At Me!	XXXTENTACION	rap	83	92.0
231	2017	20 Min	Lil Uzi Vert	rap	83	91.0

Year: 2018

	year	song_name	artists	broad_genres	song_popularity	artist_popularity
40	2018	Lucid Dreams	Juice WRLD	rap	88	96.0
69	2018	SICKO MODE	Travis Scott	rap	87	94.0
75	2018	SAD!	XXXTENTACION	rap	87	92.0
68	2018	All Girls Are The Same	Juice WRLD	rap	87	96.0
95	2018	Hope	XXXTENTACION	rap	86	92.0
96	2018	Snowman	Sia	pop	86	90.0
112	2018	Moonlight	XXXTENTACION	rap	85	92.0
121	2018	Falling	Trevor Daniel	rap	85	77.0
128	2018	Let Me Down Slowly	Alec Benjamin	pop	85	80.0
129	2018	SLOW DANCING IN THE DARK	Joji	pop	85	85.0

Year: 2019

	year	song_name	artists	broad_genres	song_popularity	artist_popularity
8	2019	Streets	Doja Cat	rap	94	91.0
14	2019	Watermelon Sugar	Harry Styles	pop	92	90.0
25	2019	Someone You Loved	Lewis Capaldi	pop	90	86.0
29	2019	Circles	Post Malone	rap	89	93.0
30	2019	Arcade	Duncan Laurence	pop	89	80.0
42	2019	Adore You	Harry Styles	pop	88	90.0
43	2019	Dance Monkey	Tones And I	pop	88	81.0
44	2019	bad guy	Billie Eilish	pop	88	92.0
45	2019	The Box	Roddy Ricch	rap	88	88.0
46	2019	Memories	Maroon 5	pop	88	91.0

```
Year:  2020
```

	year	song_name	artists	broad_genres	song_popularity	artist_popularity
2	2020	Save Your Tears	The Weeknd	pop	97	96.0
3	2020	telepatía	Kali Uchis	pop	97	88.0
4	2020	Blinding Lights	The Weeknd	pop	96	96.0
5	2020	The Business	Tiësto	pop	95	87.0
6	2020	Heartbreak Anniversary	Giveon	pop	94	91.0
7	2020	WITHOUT YOU	The Kid LAROI	rap	94	90.0
9	2020	Good Days	SZA	rap	93	88.0
10	2020	positions	Ariana Grande	pop	92	95.0
11	2020	Hecha Pa' Mi	Boza	unknown	92	78.0
20	2020	you broke me first	Tate McRae	pop	91	83.0

```
Year:  2021
```

	year	song_name	artists	broad_genres	song_popularity	artist_popularity
0	2021	drivers license	Olivia Rodrigo	pop	99	88.0
1	2021	Astronaut In The Ocean	Masked Wolf	rap	98	85.0
12	2021	Up	Cardi B	rap	92	90.0
13	2021	Hold On	Justin Bieber	pop	92	100.0
15	2021	What's Next	Drake	rap	91	98.0
16	2021	We're Good	Dua Lipa	pop	91	95.0
22	2021	911	Sech	rap	91	89.0
23	2021	Anyone	Justin Bieber	pop	90	100.0
26	2021	deja vu	Olivia Rodrigo	pop	90	88.0
31	2021	Batom de Cereja - Ao Vivo	Israel & Rodolffo	pop	89	80.0

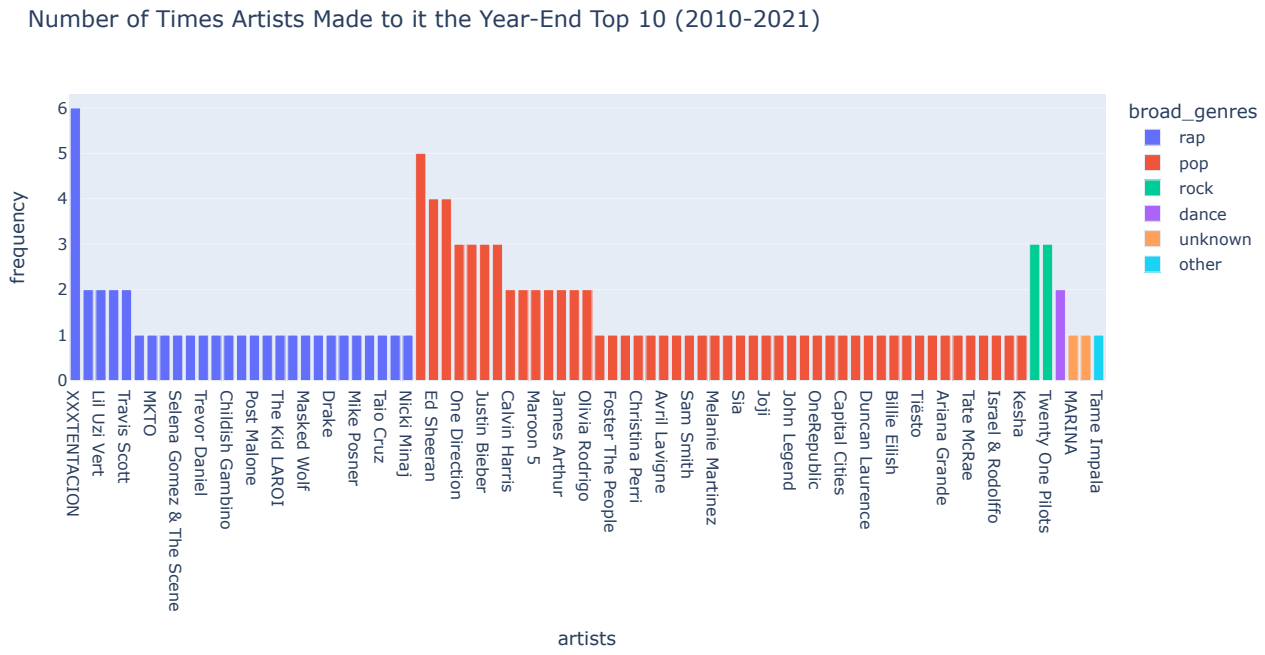
Looking at the top 10 songs from every year, we notice several things:

1. Most of the songs are in English.
2. Many of the most popular songs are by the same artists.
3. It appears Bruno Mars and One Direction dominated the early 2010s and Travis Scott, the Weeknd, XXXTENTACION, and Juice WLRD were more popular in the later 2010s/early 2020s. \
4. The most popular songs from the early 2010s are mostly pop songs but more rap songs make the list in the second half of the decade.

Now, let's create a list of all the artists that make up this list so we can visualize which ones occur the most frequently.

In [69]:
```
#Split on ',' so we can split up the featured artists in multi-artist songs
all_names = []
for name in concat_df.artists.tolist():
  all_names += name.split(',')

#Count the number of time each artist appears on the list
all_names = pd.DataFrame(Counter(all_names).most_common(100), columns=['artists', 'frequency'])

#Join with df_artists on the names so we can obtain the genres column in df_artists
all_names = all_names.merge(df_artists, left_on='artists', right_on='artists_name')

#Dropping duplicated columns
all_names = all_names.drop_duplicates(subset='artists', keep="first")

#Finding the broad category of the artists using the find_broad_category function
all_names['broad_genres'] = all_names['genres'].apply(lambda x: find_broad_category(x))
```

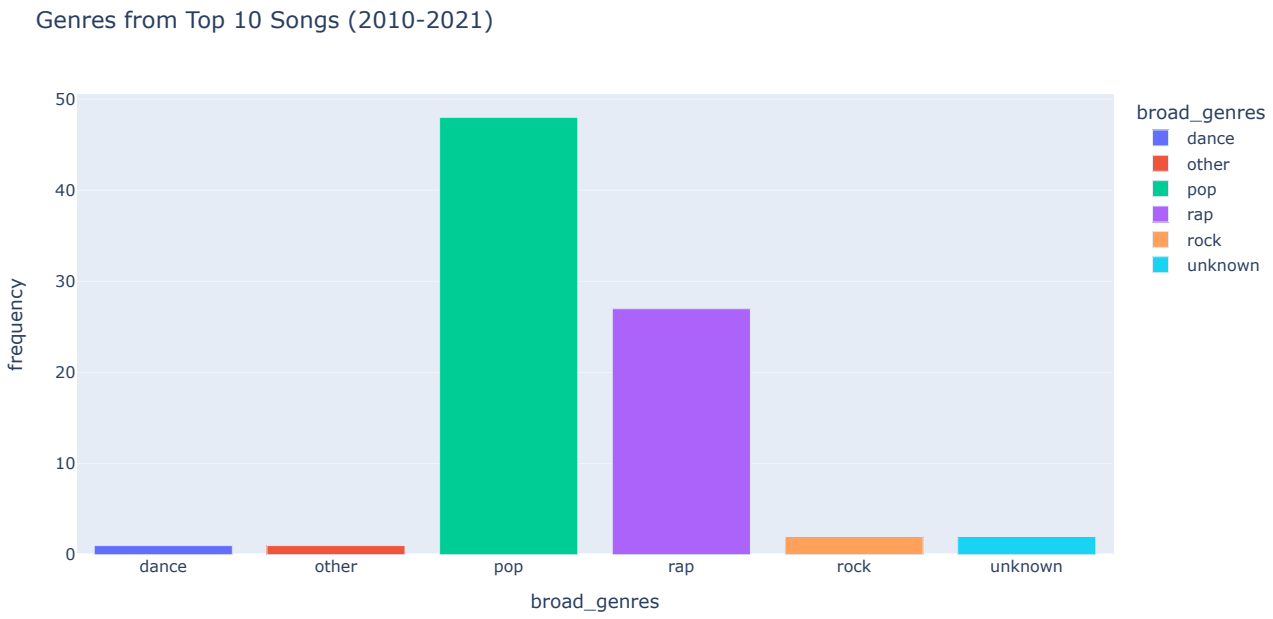
In [70]:
```
#Making a bar plot showing each artist and their frequency making the top 10 list. We grouped each genre together as well.
px.bar(all_names, x="artists", y="frequency", color='broad_genres', \
 title="Number of Times Artists Made to it the Year-End Top 10 (2010-2021)")
fig.show()
```



As a result of the above chart, it appears Bruno Mars, Ed Sheeran, Justin Bieber, The Weeknd, XXXTENTACION, Travis Scott, One Direction, Imagine Dragons, Twenty One Pilots, Avicii, and Arctic Monkeys appear most frequently.

Now, let's look at the frequency of each genre over the decade.

In [71]:
```
#Group by genre and count each instance
recent_top_genre = all_names.groupby('broad_genres')['id_artists'].count().reset_index().rename(columns={'id_artist

#Plot each genre against its frequency
fig = px.bar(recent_top_genre, x="broad_genres", y="frequency", color='broad_genres', \
      title="Genres from Top 10 Songs (2010-2021)")
fig.show()
```



As we can see, an overwhelming majority of the most popular songs in the last decade are pop songs, but rap songs are quite popular as well.

Section 5: Model 1 - Regression : Preprocessing

In this next section, we will process the data and prepare it for training. This involves selecting the features we want to include in the model, splitting training and testing data, one hot encoding categorical features, and standardizing numerical features. This preprocessing will be for our regression models. We'll have a separate processing section for our Classification model since this will involve different transformations

Data Transformation

First, let's select our features from our dataframe and remove rows where the song popularity is 0.

In [72]:
```
df = df[df['song_popularity'] > 0]
```

In [73]:
```
#extracting features from df
X = df[['artist_popularity', 'explicit', 'followers', 'duration_ms', 'danceability', \
        'energy', 'loudness', 'key', 'mode', 'speechiness', 'acousticness', 'instrumentalness', 'liveness', \
        'valence', 'tempo', 'time_signature', 'year', 'broad_genres'
        ]]

y = df[['song_popularity']]
```

Now let's split the data into training and testing sets.

In [74]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

One-hot encode the broad genres for both the training and test sets.

In [75]:
```
#Create dummies
one_hot_encoded_train_genre = pd.get_dummies(X_train['broad_genres'], prefix=None, prefix_sep='_')
one_hot_encoded_test_genre = pd.get_dummies(X_test['broad_genres'], prefix=None, prefix_sep='_')

#Drop non-encoded column
X_train = X_train.drop(columns=['broad_genres'])
X_test = X_test.drop(columns=['broad_genres'])

#Add one-hot-encoded values to X_train and X_test
X_train[list(one_hot_encoded_train_genre.columns)] = one_hot_encoded_train_genre
X_test[list(one_hot_encoded_test_genre.columns)] = one_hot_encoded_test_genre
```

Now let's separate the numerical and categorical training data so we can standardize the numerical data.

In [76]:
```
numerical_train_df = X_train[['artist_popularity', 'duration_ms', 'followers', 'key', 'danceability', \
                  'energy', 'loudness', 'speechiness', 'acousticness', 'instrumentalness', 'liveness', \
                  'valence', 'tempo', 'time_signature', 'year'
                  ]]

numerical_test_df = X_test[['artist_popularity', 'duration_ms', 'followers', 'key', 'danceability', \
                  'energy', 'loudness', 'speechiness', 'acousticness', 'instrumentalness', 'liveness', \
                  'valence', 'tempo', 'time_signature', 'year'
                  ]]

categorical_train_df = X_train[['explicit', 'mode'] + list(one_hot_encoded_train_genre.columns)]

categorical_test_df = X_test[['explicit', 'mode'] + list(one_hot_encoded_test_genre.columns)]
```

Now, let's standardize the numerical data using StandardScalar. We chose to use StandardScalar because it's the preferred scaling method for most regression and classification problems, especially the models we'll use here.

In [77]:
```
scale = StandardScaler()
scale.fit(numerical_train_df)

X_train = scale.transform(numerical_train_df)
X_test = scale.transform(numerical_test_df)
```

Then, let's concatenate the categorical columns and numerical columns.

In [78]:
```
X_train = np.concatenate((X_train, categorical_train_df.to_numpy()), axis=1)
X_test = np.concatenate((X_test, categorical_test_df.to_numpy()), axis=1)
```

Section 6: Model 1 - Regression: Predicting Song Popularity + Evaluation

In this section, we'll train our first step of models. These are our continuous models that aim to predict song_popularity based on the features we selected in the previous section.

Linear Regression - Baseline Model

First, we'll use basic linear regression with no regularization for our baseline model. A baseline model is essentially a simple model that acts as a reference in a machine learning project. Its main function is to contextualize the results of trained models. While baseline models usually lack complexity and may have little predictive power, it is a good opener for our analysis. Linear Regression works by finding the weights (or coefficients) that are used in the linear equation that best explains the correlation between y and all the independent variables

In [93]:
```
from sklearn.linear_model import LinearRegression

linreg = LinearRegression().fit(X_train, y_train)
y_pred = linreg.predict(X_test)
r_squared = linreg.score(X_test, y_test)
```

We can evaluate the Linear Regression model using its r-squared, a goodness-of-fit measure that explains how much variation of song_popularity is explained its features. An r-squared closer to 1 is a better fit and an r-squared closer to 0 is a worse fit.

In [94]:
```
r_squared
```

Out[94]:
```
0.5268546122009392
```

An r-squared of .5268 means that the model explains 52.68% of the variation in song_popularity can be explained by the features we selected. This isn't great but it's a decent starting point.

Let's look at a couple other metrics including:

1. Mean Absolute Error (MAE) - which calculates the absolute difference between the actual and predicted values.

2. Mean Squared Error (MSE) - the squared difference between the actual and predicted values
3. Adjusted R^2 - version of R^2 that shows whether or not adding extra features improves the regression model (something R^2 fails to explain)

In [95]:
```
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error

print("MAE",mean_absolute_error(y_test, y_pred))
print("MSE",mean_squared_error(y_test,y_pred))
n = X_test.shape[0]
p = X_test.shape[1]
adjusted_r_squared = 1 - ((1-r_squared)*((n - 1)/(n-p-1)))
print('Adjusted R^2', adjusted_r_squared)
```

```
MAE 8.829567242236902
MSE 128.9742628230686
Adjusted R^2 0.5266578533603196
```

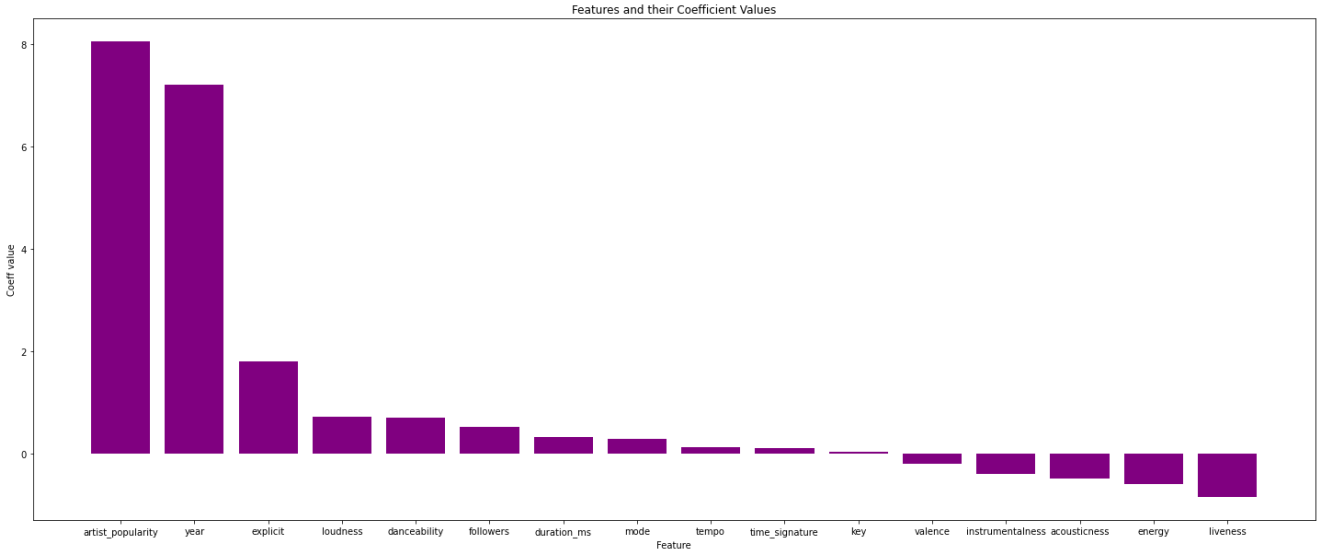
Our MAE and MSE are somewhat low and the adjust R^2 is not very different from regular R^2, showing that are our additional features are not contributing to misleading R^2.

Now let's take a look at each feature's coefficient.

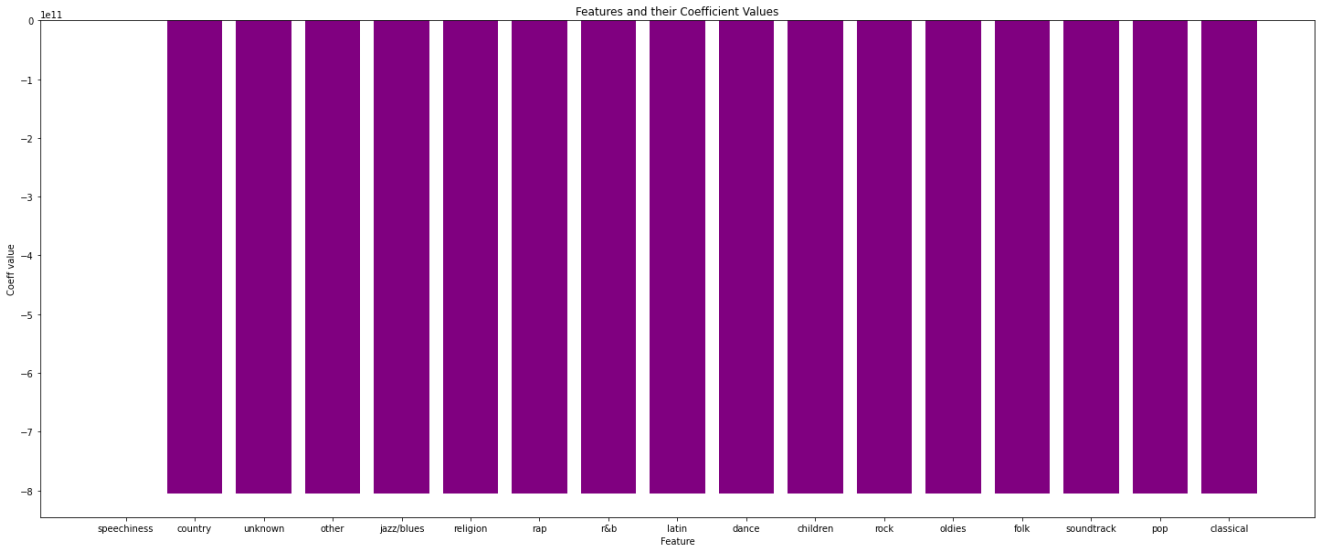
In [96]:
```
coefficients = pd.DataFrame(data={
    'Attribute': ['artist_popularity', 'duration_ms', 'followers', 'key', 'danceability', \
                  'energy', 'loudness', 'speechiness', 'acousticness', 'instrumentalness', 'liveness', \
                  'valence', 'tempo', 'time_signature', 'year'
                  ] + ['explicit', 'mode'] + list(one_hot_encoded_train_genre.columns),
    'Importance': linreg.coef_[0]
})
coefficients = coefficients.sort_values(by='Importance', ascending=False)
```

We split this into 2 graphs so it's easier to see.

In [97]:
```
plt.rcParams["figure.figsize"] = (25,10)
plt.bar(coefficients['Attribute'][0:16], coefficients['Importance'][0:16], color='purple')
plt.ylabel('Coeff value') #fontSize=15)
plt.xlabel('Feature') #fontSize=15)
plt.title('Features and their Coefficient Values') #fontize=18)
plt.show()
```



In [99]:
```
plt.rcParams["figure.figsize"] = (25,10)
plt.bar(coefficients['Attribute'][16:33], coefficients['Importance'][16:33], color='purple')
plt.ylabel('Coeff value')
plt.xlabel('Feature')
plt.title('Features and their Coefficient Values')
plt.show()
```



It appears what features influence the model's output the most are artist_popularity, year, country, folk, pop, and classical.

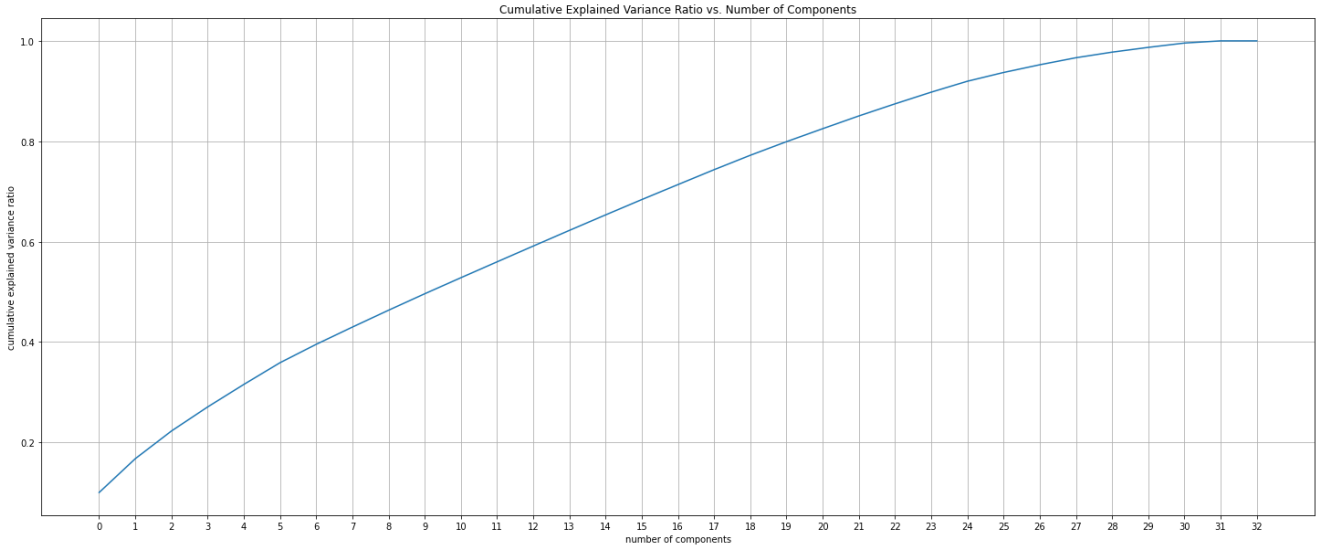
Linear Regression with PCA

Now let's perform PCA to see if this helps reduce the complexity and noise in our model. PCA in linear regression has been used to serve two basic goals. The first one is performed on datasets where the number of predictor variables is too high. It has been a method of dimensionality reduction along with Partial Least Squares Regression.

In our case PCA is useful because we can select the features that are most important for the model, which could boost our R^2. Once we fit PCA on our training data, we can find the explained variance ratio, which is the percentage of variance that is explained by each feature. We can then find the cumulative sum of each and plot these. We should take the number of features where the line falls flat.

In [100]:
```
from sklearn.decomposition import PCA

#fit PCA over X_train
X = StandardScaler().fit_transform(X_train)

#Fit on X
pca = PCA().fit(X)

#Derive the explained variance ratios
explained_variance_ratios = pca.explained_variance_ratio_
cum_evr = explained_variance_ratios.cumsum()
```

In [101]:
```
plt.plot(np.arange(0,X.shape[1]), cum_evr)

plt.ylabel('cumulative explained variance ratio')
plt.xlabel('number of components')
plt.title('Cumulative Explained Variance Ratio vs. Number of Components')
plt.xticks(np.arange(0, X.shape[1], 1))
plt.grid()

plt.show()
```



Here, we see that the line falls flat at 31, so we'll reduce our data to 31 features and train another linear regression model.

In [102]:
```
#Fit and transform X_test and X_train with PCA
pca = PCA(n_components=31)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

#Re-train and re-fit linear regression with X_train_pca and X_test_pca
linreg = LinearRegression().fit(X_train_pca, y_train)
y_pred = linreg.predict(X_train_pca)
r_squared = linreg.score(X_test_pca, y_test)
```

In [103]:
```
r_squared
```

Out[103]:
```
0.5268631776700148
```

We see that this didn't improve our r^2 they way we hoped. Now we'll consider other models other than linear regression.

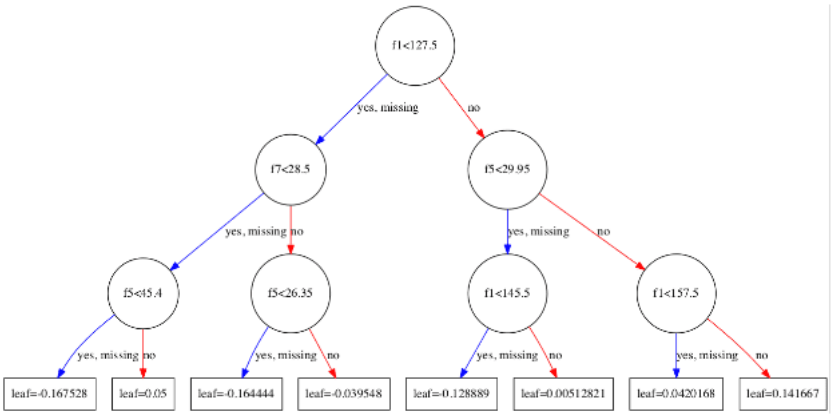
XGBoost Regressor

We decided to next try XGBoost to solve our regression problem.

Extreme Gradient Boosting, or XGBoost for short, is an efficient open-source implementation of the gradient boosting algorithm. Gradient boosting refers to a class of ensemble machine learning algorithms that can be used for classification or regression predictive modeling problems.

We chose XGBoost because it is known to yield very high r^2 scores. XGBoost is a type of decision tree algorithm that uses Gradient Boosting, which sequentially builds stronger models from weaker models. While it is considered more accurate than linear baseline model, its limitation is that it is more prone to overfitting.

Source: https://machinelearningmastery.com/visualize-gradient-boosting-decision-trees-xgboost-python/ (https://machinelearningmastery.com/visualize-gradient-boosting-decision-trees-xgboost-python/)



In [106]:
```
%%time
import xgboost as xgb

xg_reg = xgb.XGBRegressor()
xg_reg.fit(X_train,y_train)
```

```
CPU times: user 1min 18s, sys: 15.5 s, total: 1min 34s
Wall time: 13.8 s
```

Out[106]:
```
XGBRegressor(base_score=0.5, booster='gbtree', callbacks=None,
             colsample_bylevel=1, colsample_bynode=1, colsample_bytree=1,
             early_stopping_rounds=None, enable_categorical=False,
             eval_metric=None, feature_types=None, gamma=0, gpu_id=-1,
             grow_policy='depthwise', importance_type=None,
             interaction_constraints='', learning_rate=0.300000012, max_bin=256,
             max_cat_threshold=64, max_cat_to_onehot=4, max_delta_step=0,
             max_depth=6, max_leaves=0, min_child_weight=1, missing=nan,
             monotone_constraints='()', n_estimators=100, n_jobs=0,
             num_parallel_tree=1, predictor='auto', random_state=0, ...)
```

In [107]:
```
y_pred = xg_reg.predict(X_test)
```

Now, let's look at the metrics.

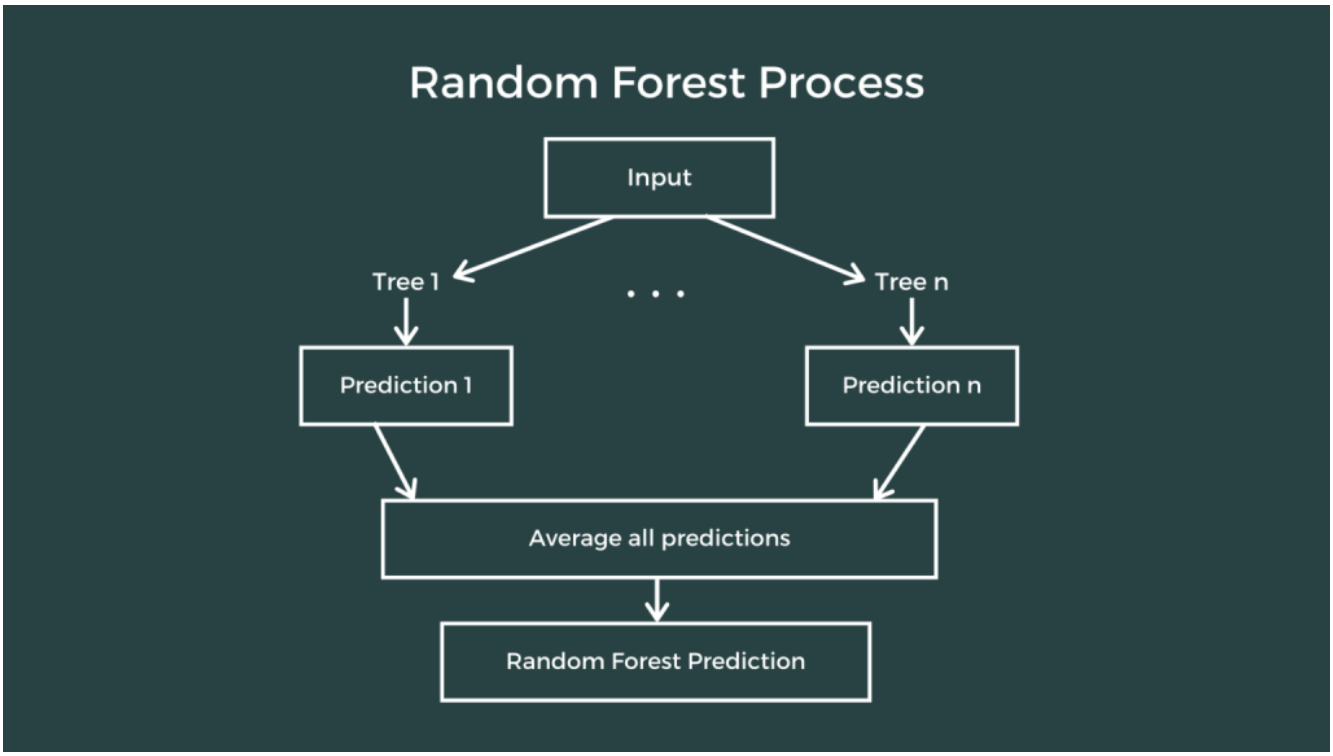
In [108]:
```
print('r^2', xg_reg.score(X_test, y_test))
print("MAE",mean_absolute_error(y_test, y_pred))
print("MSE",mean_squared_error(y_test,y_pred))
```

```
r^2 0.6301861076766799
MAE 7.703353122881975
MSE 100.80722622279053
```

Notice in this model, the r^2, MAE, and MSE are all a bit higher. We're a bit surprised that XGBoost didn't have remarkable better results compared to linear regression.

Random Forest Regressor

Last, we wanted to try the Random Forest Regressor.



Source: https://medium.com/@theclickreader/random-forest-regression-explained-with-implementation-in-python-3dad88caf165 (https://medium.com/@theclickreader/random-forest-regression-explained-with-implementation-in-python-3dad88caf165)

A random forest is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

Since overfitting could be a limitation of XGBoost regressor Random Forest seems like a good alternative. Random Forest uses bagging techniques to reduce variance by leaving some features out. At the same time, it also creates randomness by repeating some observations. Random Forest works by growing multiple decision trees based on our model features and takes the average when making a decision.

In [109]:
```
%%time
from sklearn.ensemble import RandomForestRegressor

rf = RandomForestRegressor()
rf.fit(X_train, y_train)
```

```
CPU times: user 6min 48s, sys: 1.76 s, total: 6min 49s
Wall time: 6min 49s
```

Out[109]:
```
RandomForestRegressor()
```

In [110]:
```
y_pred = rf.predict(X_test)
```

In [111]:
```
print('r^2', rf.score(X_test, y_test))
print("MAE",mean_absolute_error(y_test, y_pred))
print("MSE",mean_squared_error(y_test,y_pred))
```

```
r^2 0.66843692711886
MAE 7.205037520541308
MSE 90.38047079591816
```

As a result, Random Forest is by far our best model for predicting song_popularity with highest r^2 and lowest MAE and MSE.

Now, we would like to convert this into a classification problem. We realize it's very difficult to predict precise song_popularity, so in the next section, we'll put our data into 10 different groups by song_popularity and try to accurately predict the class each song.

Section 7: Model 1 - Classification: Preprocessing

In this section, we'll prepare the data for our classification model. This will be similar to our preprocessing for our regression model but we will pick different features, find our classes, address class imbalance, and remove outliers for each class. First, let's find our classes. We'll have 10 classes. Each one will be based on song_popularity.

In [112]:
```
def determine_class(x):
  if 0 <= x <= 10:
    return 'group 1'
  elif 10 < x <= 20:
    return 'group 2'
  elif 20 < x <= 30:
    return 'group 3'
  elif 30 < x <= 40:
    return 'group 4'
  elif 40 < x <= 50:
    return 'group 5'
  elif 50 < x <= 60:
    return 'group 6'
  elif 60 < x <= 70:
    return 'group 7'
  elif 70 < x <= 80:
    return 'group 8'
  elif 80 < x <= 90:
    return 'group 9'
  elif 90 < x <= 100:
    return 'group 10'
```

In [113]:
```
#creating column called class based on a song's popularity
df['class'] = df['song_popularity'].apply(lambda x: determine_class(x))
```

Let's see the distribution of our classes.

In [114]:
```
df['class'].value_counts()
```

Out[114]:
```
group 3     85779
group 4     84595
group 2     67997
group 5     57833
group 1     52767
group 6     31559
group 7     12436
group 8      3539
group 9       413
group 10       23
Name: class, dtype: int64
```

It looks like there's a lot of class imbalance. Let's first look at removing outliers in each group (especially the ones that consist the most data points) and then we'll address these imbalances.

Since our last model wasn't super accurate, let's attempt to remove outliers for each group. We can look at the variance of our most numerical important features for predicting song_popularity - artist_popularity and year

In [115]:
```
for group in ['group 1', 'group 2', 'group 3', 'group 4', 'group 5', 'group 6', 'group 7', 'group 8', 'group 9', 'group 10']:
    print('{}'.format(group))
    print('var artist_popularity', df[df['class'] == group]['artist_popularity'].var())
    print('var year', df[df['class'] == group]['year'].var())
    print()
```

```
group 1
var artist_popularity 328.2990483585647
var year 455.10139890488557

group 2
var artist_popularity 246.9599522624908
var year 239.19880756964045

group 3
var artist_popularity 253.57915349342395
var year 215.77536529149586

group 4
var artist_popularity 268.5004732135873
var year 205.01862803384154

group 5
var artist_popularity 171.09237405357146
var year 200.9021159288744

group 6
var artist_popularity 134.88929988218698
var year 196.3460569743654

group 7
var artist_popularity 108.52941945915481
var year 209.04739669116327

group 8
var artist_popularity 97.62185537843597
var year 209.70896068694933

group 9
var artist_popularity 59.36987235242953
var year 123.75587108300618

group 10
var artist_popularity 26.209486166007903
var year 0.35968379446640303
```

It looks like our classes with the most data points (groups 1-8) have the most variation in their artist_popularity and year. Let's look at removing the outliers for these columns. Since these are somewhat extreme, let's remove the bottom 20 percentile and top 80 percentile values for classes 1-8.

In [116]:
```
def remove_outliers(df, group):
  df = df[(df['class'] == group)]
  return df[(df['artist_popularity'] > df['artist_popularity'].quantile(.20)) & \
   (df['artist_popularity'] < df['artist_popularity'].quantile(.80)) & (df['year'] < df['year'].quantile(.80)) & \
   (df['year'] > df['year'].quantile(.20))]
```

In [117]:
```
df_g1 = remove_outliers(df, 'group 1')
df_g2 = remove_outliers(df, 'group 2')
df_g3 = remove_outliers(df, 'group 3')
df_g4 = remove_outliers(df, 'group 4')
df_g5 = remove_outliers(df, 'group 5')
df_g6 = remove_outliers(df, 'group 6')
df_g7 = remove_outliers(df, 'group 7')
df_g8 = remove_outliers(df, 'group 8')
df_outliers_removed = pd.concat([df_g1, df_g2, df_g3, df_g4, df_g5, df_g6, df_g7, df_g8])
```

In [118]:
```
df = pd.concat([df_outliers_removed,
                df[df['class'] == 'group 9'],
                df[df['class'] == 'group 10']
                ])
```

In [119]:
```
df['class'].value_counts()
```

Out[119]:
```
group 3     29047
group 4     29042
group 2     23561
group 5     19664
group 1     18583
group 6      9723
group 7      3817
group 8      1152
group 9       413
group 10       23
Name: class, dtype: int64
```

Let's our data is still imbalanced but atleast we were able to get rid of a lot of outliers. Now let's check our variances.

In [120]:
```
for group in ['group 1', 'group 2', 'group 3', 'group 4', 'group 5', 'group 6', 'group 7', 'group 8', 'group 9', 'g
  print('{}'.format(group))
  print('var artist_popularity', df[df['class'] == group]['artist_popularity'].var())
  print('var year', df[df['class'] == group]['year'].var())
  print( )
```

```
group 1
var artist_popularity 82.17259818650818
var year 107.58491417546682

group 2
var artist_popularity 51.27685065079419
var year 57.881345553992745

group 3
var artist_popularity 56.5995733465005
var year 50.70576320662505

group 4
var artist_popularity 60.15418726485496
var year 48.29681584559189

group 5
var artist_popularity 42.888704985660134
var year 40.39120063788836

group 6
var artist_popularity 30.949916048330152
var year 34.67312435559421

group 7
var artist_popularity 21.096547004491107
var year 36.22080752607913

group 8
var artist_popularity 24.197723597837626
var year 32.41895634231103

group 9
var artist_popularity 59.36987235242953
var year 123.75587108300618

group 10
var artist_popularity 26.209486166007903
var year 0.35968379446640303
```

We still have quite a bit of variance in group 1 but this still looks a lot better now. Now, let's extract our columns.

In [121]:
```
from sklearn.preprocessing import LabelEncoder

X = df[['artist_popularity', 'explicit', 'followers', 'duration_ms', 'danceability', \
        'energy', 'loudness', 'key', 'mode', 'speechiness', 'acousticness', 'instrumentalness', 'liveness', \
        'valence', 'tempo', 'time_signature', 'year', 'broad_genres'
        ]]

y = df[['class']]
y = LabelEncoder().fit_transform(y)
```

Then, one hot encode our categorical variables.

In [122]:
```
one_hot_encoded_genre = pd.get_dummies(X['broad_genres'], prefix=None, prefix_sep='_',
     dummy_na=False, columns=None,
     sparse=False, drop_first=False,
     dtype=None)
X = X.drop(columns=['broad_genres'])
X[list(one_hot_encoded_genre.columns)] = one_hot_encoded_genre
```

Now we're going to use SMOTE (Synthetic Minority Oversampling Technique), which increases the number of samples in every class so that all classes have the same number of data points as the class with the greatest number of samples. It works by creating new samples in every class that are similar to those that already exist. It does this by drawing a line between a data points and another one close to it and then creating a new data point along that line.

Source: https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/ (https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/)

In [126]:
```
%%time
from imblearn.over_sampling import SMOTE

oversample = SMOTE()
X_resampled, y_resampled = oversample.fit_resample(X, y)
```

```
CPU times: user 1min 3s, sys: 23.6 s, total: 1min 27s
Wall time: 33.5 s
```

In [127]:
```
pd.DataFrame(y_resampled).value_counts()
```

Out[127]:
```
0    29047
1    29047
2    29047
3    29047
4    29047
5    29047
6    29047
7    29047
8    29047
9    29047
dtype: int64
```

Now we see that every class has 29,047 data points. Let's now split our data into testing and training sets exactly as we've done in the previous section.

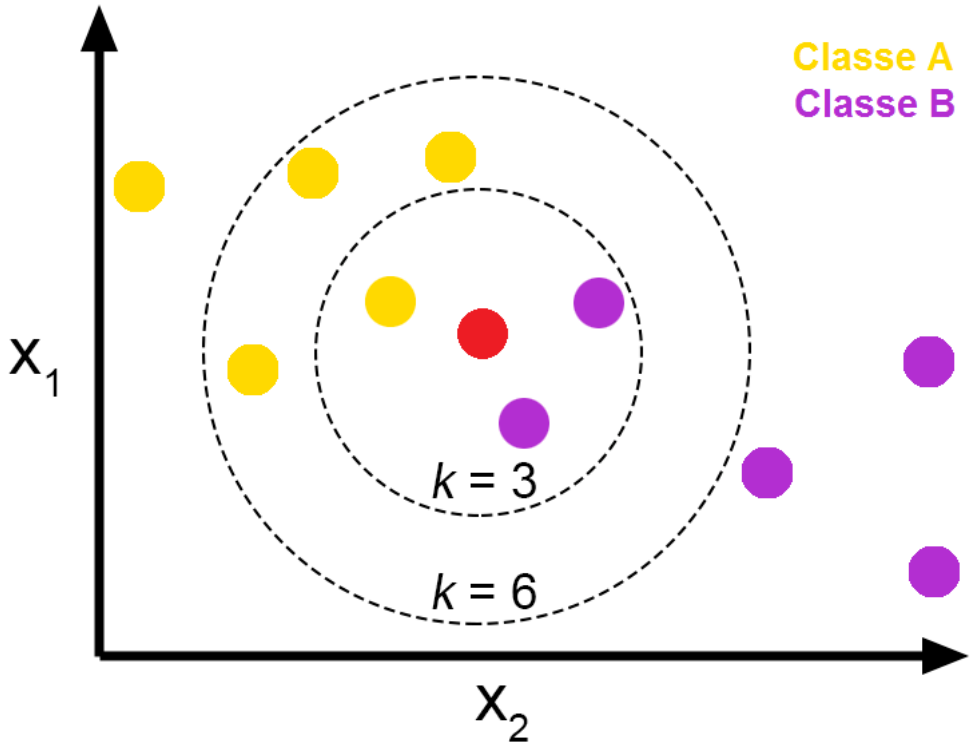
In [128]:
```
X_train, X_test, y_train, y_test = train_test_split(X_resampled, y_resampled, test_size=0.2, random_state=42)

numerical_train_df = X_train[['artist_popularity', 'duration_ms', 'followers', 'key', 'danceability', \
                  'energy', 'loudness', 'speechiness', 'acousticness', 'instrumentalness', 'liveness', \
                  'valence', 'tempo', 'time_signature', 'year'
                  ]]

numerical_test_df = X_test[['artist_popularity', 'duration_ms', 'followers', 'key', 'danceability', \
                  'energy', 'loudness', 'speechiness', 'acousticness', 'instrumentalness', 'liveness', \
                  'valence', 'tempo', 'time_signature', 'year'
                  ]]


categorical_train_df = X_train[['explicit', 'mode'] + list(one_hot_encoded_genre.columns)]

categorical_test_df = X_test[['explicit', 'mode'] + list(one_hot_encoded_genre.columns)]

scale = StandardScaler()
scale.fit(numerical_train_df)

X_train = scale.transform(numerical_train_df)
X_test = scale.transform(numerical_test_df)

X_train = np.concatenate((X_train, categorical_train_df.to_numpy()), axis=1)
X_test = np.concatenate((X_test, categorical_test_df.to_numpy()), axis=1)
```

Section 8: Model 1 - Classification: Predicting Song Popularity + Evaluation

KNN

Let's try to use the K-Nearest Neighbors Algorithm. This algorithm works by placing all training points on a graph. Then, for each testing point we find the k closest training points and classify the testing point as the most frequently occurring class out those training points. We wanted to try KNN because we haven't tried a clustering algorithm yet and KNN usually works very well in classification. While we think this could be a descent model, we don't think it will be the best model because KNN doesn't work very well when there are a lot of features in the data. It also took about 5 minutes for the algorithm to load. Here's an example visualization to illustrate how KNN works:

Source: https://towardsdatascience.com/knn-k-nearest-neighbors-1-a4707b24bd1d (https://towardsdatascience.com/knn-k-nearest-neighbors-1-a4707b24bd1d)



In [129]:
```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
```

Out[129]:
```
KNeighborsClassifier()
```

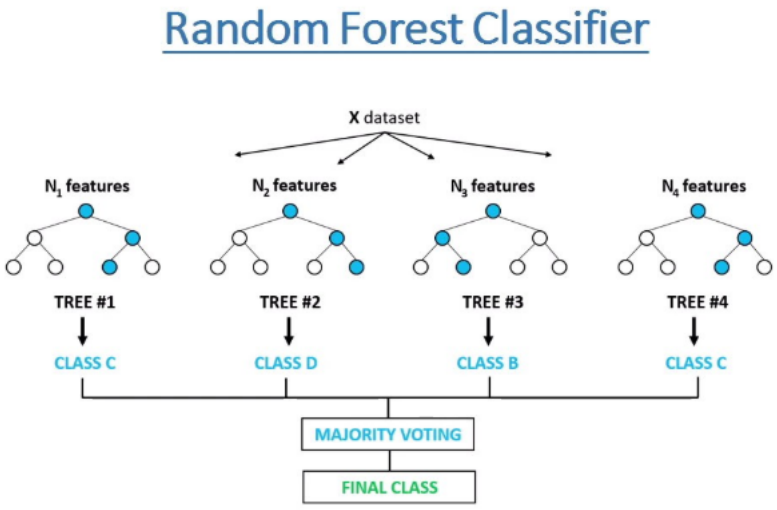
In [130]:
```
knn.score(X_test, y_test)
```

Out[130]:
```
0.6686749061865253
```

We got an accuracy of 66.87%, which is decent, but I still think we can do better than this. We can definitely find another classification model that is faster and more accurate than this.

Random Forest Classifier

Next, we'll look at Random Forest Classifier.



Source: https://medium.com/analytics-vidhya/random-forest-classifier-and-its-hyperparameters-8467bec755f6 (https://medium.com/analytics-vidhya/random-forest-classifier-and-its-hyperparameters-8467bec755f6)

Since this model worked best for our regression problem, we were curious to see how it performed in our classification problem. We also used Random Forest because it's quicker than KNN, which in our use case would be very slow since we have a somewhat large number of features. Random Forest uses bagging techniques to reduce variance by leaving some features out. At the same time, it also creates randomness by repeating some observations. The algorithm essentially creates a 'forest' a decision trees and then picks a class based on majority vote.

In [131]:
```
%%time
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier()
clf.fit(X_train, y_train)
```

```
CPU times: user 58.6 s, sys: 385 ms, total: 59 s
Wall time: 59 s
```

Out[131]:
```
RandomForestClassifier()
```

In [132]:
```
y_pred = clf.predict(X_test)
```

In [133]:
```
clf.score(X_test, y_test)
```

Out[133]:
```
0.7947636588976487
```

Hooray, we have an accuracy of 79.7%! This is our best result! Now let's check the precision, recall, and f1 score of our model. I set average='macro' to calculate the metrics for each of the 10 classes and not take label imbalance into account because our classes are perfectly balanced.

In [134]:
```
from sklearn.metrics import f1_score
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score

print('f1 score', f1_score(y_test, y_pred, average='macro'))
print('precision', precision_score(y_test, y_pred, average='macro'))
print('recall', recall_score(y_test, y_pred, average='macro'))
```

```
f1 score 0.7917513166538099
precision 0.7931849527869995
recall 0.7943592279704366
```

As we see, all three values are very close to the accuracy. Note, recall is the highest out of the three, meaning our model is good at correctly predicting the true labels. Let's define the other metrics. Precision finds the ratio between true positives all all other positives. F-1 score is the harmonic mean of the precision and recall.

Now let's try to tune the hyperparameters and see if we can get a better model. Note, when we tried using GridSearchCV our RAM session crashed, so were going to manually change a couple hyperparameters and see what happens. First, let's see what the default parameters we used in our initial model are.

In [135]:
```
clf.get_params()
```

Out[135]:
```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 100,
 'n_jobs': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 0,
 'warm_start': False}
```

Let's change n_estimators to 250 and the criterion to entropy.

In [139]:
```
%%time

clf_updated = RandomForestClassifier(criterion='entropy', n_estimators=250)
clf_updated.fit(X_train, y_train)
```

```
CPU times: user 3min 5s, sys: 1.02 s, total: 3min 6s
Wall time: 3min 6s
```

Out[139]:
```
RandomForestClassifier(criterion='entropy', n_estimators=250)
```

In [140]:
```
y_pred = clf_updated.predict(X_test)
```

In [141]:
```
clf_updated.score(X_test, y_test)
```

Out[141]:
```
0.8009777257548112
```

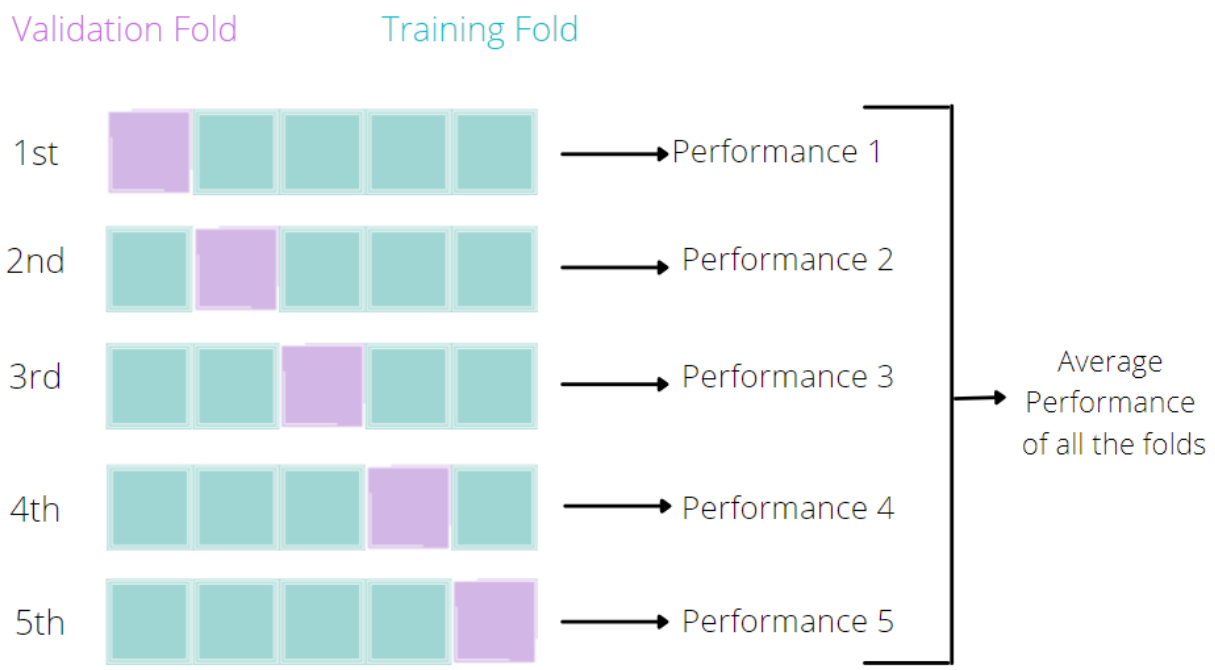
Our accuracy improved by 1 percent. Let's check the other metrics

In [142]:
```
print('f1 score', f1_score(y_test, y_pred, average='macro'))
print('precision', precision_score(y_test, y_pred, average='macro'))
print('recall', recall_score(y_test, y_pred, average='macro'))
```

```
f1 score 0.7974628067178796
precision 0.7990411496812423
recall 0.800579277952896
```

It took almost 8 times longer to run the model with the enhanced hyperparameters and only increased in accuracy by one percent. We want to try k-Fold Cross Validation now but we will use the old model since it's much faster. We'll use 5 folds because this will be equivalent to how we split the data in the previous sections (training on 4/5 of the data and testing on 1/5). Cross Validation allows us to iteratively train and test on different parts of the dataset. This allows us to ensure that our model isn't overfitting and training multiple times on unseen data. Here is a visual to show how it works.

Source: https://pub.towardsai.net/k-fold-cross-validation-for-machine-learning-models-918f6ccfd6d (https://pub.towardsai.net/k-fold-cross-validation-for-machine-learning-models-918f6ccfd6d)



In [143]:
```
%%time
from sklearn.model_selection import cross_val_score

clf = RandomForestClassifier()
print(cross_val_score(clf, X_resampled, y_resampled, cv=5, scoring='accuracy'))
```

```
[0.74365683 0.7810273  0.79917031 0.80791476 0.80498847]
CPU times: user 4min 51s, sys: 2.76 s, total: 4min 53s
Wall time: 4min 54s
```

As we can see our model doesn't overfit since the accuracy scores are all very close to each other.

Now let's create a confusion matrix. A confusion matrix is a table that is used to define the performance of a classification algorithm. A confusion matrix visualizes and summarizes the performance of a classification algorithm. In our case, will use the confusion matrix to see what classes Random Forest was good at predicting and not so good at predicting.

In [145]:
```
from sklearn.metrics import plot_confusion_matrix

classes = ['group 1', 'group 2', 'group 3', 'group 4', 'group 5', 'group 6', 'group 7', 'group 8', 'group 9', 'grou

disp = plot_confusion_matrix(clf_updated, X_test, y_test,
                                 display_labels=classes,
                                 cmap=plt.cm.Purples,
                                 values_format = '.5g')
```

As we can see, the model had the easiest time predicting data points that fell in group 10 and group 2 and had the most difficulty with groups 4 to 6. It appears pretty common for predicted classes to be one before or after its true class.

Section 9: Model 2: Song Recommendation

In this section, we'll build our second model - the Recommendation System. First, we'll preprocess the dataset by one-hot-encoding the broad_genres column and then we'll normalize the data and load it into the model. This model with use Euclidean distance to find the distance between the float and integer features. Data points that are closest in distance to the inputted song will be printed.

Updating df

In [146]:
```
# getting df from right after the clearning process
# because the two models we have have different requirements for df.
df = df_copy
```

In [147]:
```
# add genres into consideration, similar to section 6
one_hot_encoded_genre = pd.get_dummies(df['broad_genres'], prefix=None, prefix_sep='_')
df[list(one_hot_encoded_genre.columns)] = one_hot_encoded_genre
```

In [148]:
```
# check df
df.head(5)
```

Out[148]:

	song_name	artists	broad_genres	genres	song_popularity	artist_popularity	followers	song_id	id_artists	d
93803	drivers license	Olivia Rodrigo	pop	pop, post-teen pop	99	88.0	1444702.0	7lPN2DXiMsVn7XUKtOW1CS	1McMsnEElThX1knmY4oliG	
93804	Astronaut In The Ocean	Masked Wolf	rap	australian hip hop	98	85.0	177401.0	3Ofmpyhv5UAQ70mENzB277	1uU7g3DNSbsu0QjSEqZtEd	
92810	Save Your Tears	The Weeknd	pop	canadian contemporary r&b, canadian pop, pop	97	96.0	31308207.0	5QO79kh1waicV47BqGRL3g	1Xyo4u8uXC1ZmMpatF05PJ	
92811	telepatía	Kali Uchis	pop	colombian pop, pop	97	88.0	1698014.0	6tDDoYIxWvMLTdKpjFkc1B	1U1el3k54VvEUzo3ybLPlM	
92813	Blinding Lights	The Weeknd	pop	canadian contemporary r&b, canadian pop, pop	96	96.0	31308207.0	0VjIjW4GlUZAMYd2vXMi3b	1Xyo4u8uXC1ZmMpatF05PJ	

Normalize Dataset

Before building the model, first we normalize or scale the dataset. For scaling it we are using MinMaxScaler of Scikit-learn library. For that, I selected all the columns with int and float datatypes.

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

In [149]:
```
# import library
from sklearn.preprocessing import MinMaxScaler
# select numerical data
datatypes = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
normalization = df.select_dtypes(include=datatypes)
# apply minmax scaler
for col in normalization.columns:
    MinMaxScaler(col)
```

In [150]:
```
#check normarization
normalization
```

Out[150]:

	song_popularity	artist_popularity	followers	duration_ms	explicit	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumental
93803	99	88.0	1444702.0	242014	1	0.585	0.4360	10	-8.761	1	0.0601	0.72100	0.00
93804	98	85.0	177401.0	132780	0	0.778	0.6950	4	-6.865	0	0.0913	0.17500	0.00
92810	97	96.0	31308207.0	215627	1	0.680	0.8260	0	-5.487	1	0.0309	0.02120	0.00
92811	97	88.0	1698014.0	160191	0	0.653	0.5240	11	-9.016	0	0.0502	0.11200	0.00
92813	96	96.0	31308207.0	200040	0	0.514	0.7300	1	-5.934	1	0.0598	0.00146	0.00
...	...	...	...	...	...	...	...	...	...	...	...	...	
121824	0	18.0	30.0	146900	0	0.714	0.2390	0	-20.573	1	0.9520	0.50100	0.00
121823	0	18.0	30.0	127800	0	0.696	0.1990	6	-25.164	1	0.9610	0.64200	0.00
17218	0	35.0	190912.0	178733	0	0.293	0.0264	10	-21.316	1	0.0461	0.99500	0.54
17220	0	0.0	2.0	193567	0	0.797	0.1740	3	-14.661	1	0.1030	0.97900	0.05
17222	0	47.0	458624.0	196160	0	0.308	0.2400	10	-15.955	1	0.0400	0.99500	0.27

419823 rows × 18 columns

In [151]:
```
# check df
df
```

Out[151]:

	song_name	artists	broad_genres	genres	song_popularity	artist_popularity	followers	song_id	id_artist
93803	drivers license	Olivia Rodrigo	pop	pop, post-teen pop	99	88.0	1444702.0	7lPN2DXiMsVn7XUKtOW1CS	1McMsnEElThX1knmY4oli
93804	Astronaut In The Ocean	Masked Wolf	rap	australian hip hop	98	85.0	177401.0	3Ofmpyhv5UAQ70mENzB277	1uU7g3DNSbsu0QjSEqZtE
92810	Save Your Tears	The Weeknd	pop	canadian contemporary r&b, canadian pop, pop	97	96.0	31308207.0	5QO79kh1waicV47BqGRL3g	1Xyo4u8uXC1ZmMpatF05P
92811	telepatía	Kali Uchis	pop	colombian pop, pop	97	88.0	1698014.0	6tDDoYIxWvMLTdKpjFkc1B	1U1el3k54VvEUzo3ybLPl
92813	Blinding Lights	The Weeknd	pop	canadian contemporary r&b, canadian pop, pop	96	96.0	31308207.0	0VjIjW4GlUZAMYd2vXMi3b	1Xyo4u8uXC1ZmMpatF05P
...	...	...	...	...	...	...	...	...	
121824	Chapter 13.3 - Drugie życie doktora Murka	Tadeusz Dolega Mostowicz	unknown	unknown	0	18.0	30.0	2ptjeVZfBdS8d6lC3TSMJY	4eeMulNeqpZGBxybCxZOd
121823	Chapter 2.11 - Drugie życie doktora Murka	Tadeusz Dolega Mostowicz	unknown	unknown	0	18.0	30.0	2pfcWjwj9wqLeKm1rR3Cxy	4eeMulNeqpZGBxybCxZOd
17218	Jinhen Karna Tha Dil Abaad (From "Naukar")	Noor Jehan	pop	classic bollywood, classic pakistani pop, film...	0	35.0	190912.0	2xPgD421w3ZtgCBrP2P1Ia	0LruguA5aIP6yvLUIkxAN
17220	Blue Lou - 1988 Remaster	Børge Roger Henrichsen og hans kvintet	unknown	unknown	0	0.0	2.0	2yLLE8IH40gZMPXojOB5jJ	1ypsg7Hx1oHZ0JAJ5w61y
17222	Mor Maramer Batayantale	Hemant Kumar	pop	classic bollywood, classic pakistani pop, film...	0	47.0	458624.0	30vyZgP8qTabLafaLYOsrB	02Um2HIOrUdsy3wqPBZw

419823 rows × 40 columns

Recommndation Function

In [152]:

```
# here we are allowing recommendations from the same artist,
# and also having a function not allowing recommendations from the same artist.

class Spotify_Recommendation():
    def __init__(self, dataset):
        self.dataset = dataset
    # allowing same artists
    def recommend_allow_same(self, input_song, artist, amount=1):
        distance = []
        # finding the input song in df
        song = self.dataset[((self.dataset.song_name.str.lower() == input_song.lower()) & (self.dataset.artists.str
        # finding all possible recs in df
        rec = self.dataset[(self.dataset.song_name.str.lower() != input_song.lower())]
        # appying tqdm to create a progress bar
        for input_song in tqdm(rec.values):
            d = 0
            for col in np.arange(len(rec.columns)):
                # excluding all the string columns,
                # additionally leaving out year, time signature
                if not col in [0, 1, 2, 3, 7, 8, 22, 23]:
                  # here we're doing squared difference
                  # not using cosine similarity because
                  # everytime we call cosine_similarity(normalization)
                  # the runtime crashes
                  d = d + (float(song[col]) - float(input_song[col]))**2
            distance.append(d)
        # add the column distance to the rec df
        rec['distance'] = distance
        # sort songs by distance from original song
        rec = rec.sort_values('distance')
        # select columns
        columns = ['artists', 'song_name']
        return rec[columns][:amount]

    # not allowing songs from the same artist
    def recommend_dont_allow_same(self, input_song, artist, amount=1):
        distance = []
        # finding the input song in df
        song = self.dataset[((self.dataset.song_name.str.lower() == input_song.lower()) & (self.dataset.artists.str
        # finding all possible recs in df
        rec = self.dataset[(self.dataset.song_name.str.lower() != input_song.lower()) & ((self.dataset.artists.str.
        # applying tqdm to create a progress bar
        for input_song in tqdm(rec.values):
            d = 0
            for col in np.arange(len(rec.columns)):
                # excluding all the string columns,
                # additionally leaving out year, time signature
                if not col in [0, 1, 2, 3, 7, 8, 22, 23]:
                  # here we're doing squared difference,
                  # not using cosine similarity because
                  # everytime we call cosine_similarity(normalization)
                  # the runtime crashes
                  d = d + (float(song[col]) - float(input_song[col]))**2
            distance.append(d)
        # add the column distance to the rec df
        rec['distance'] = distance
        # sort songs by distance from original song
        rec = rec.sort_values('distance')
        # select columns
        columns = ['artists', 'song_name']
        # remove duplicates
        rec = rec.drop_duplicates(subset = 'artists', keep = 'first')
        return rec[columns][:amount]
```

Examples

SICKO MODE by Travis Scott

In [153]:
```
# allowing songs from the same artist to pop up
Spotify_Recommendation(df).recommend_allow_same("SICKO MODE", "Travis Scott", 10)
```

```
100%|██████████████████████████████| 419822/419822 [00:16<00:00, 25537.65it/s]
```

Out[153]:

	artists	song_name
114875	Travis Scott	STOP TRYING TO BE GOD
89370	Travis Scott	through the late night
504785	Sam Smith	Out of Our Heads - Tom Bruckner Remix
89615	Travis Scott	way back
88637	Sam Smith	Writing's On The Wall - From "Spectre" Soundtrack
91073	Travis Scott	STARGAZING
88051	Travis Scott	Antidote
218758	Sam Smith	I Feel Love
87454	Sam Smith	Lay Me Down
88938	Travis Scott	goosebumps

In [154]:
```
# only allowing songs from different artists
Spotify_Recommendation(df).recommend_dont_allow_same("SICKO MODE", "Travis Scott", 10)
```

```
100%|██████████████████████████████| 419790/419790 [00:16<00:00, 25665.22it/s]
```

Out[154]:

	artists	song_name
504785	Sam Smith	Out of Our Heads - Tom Bruckner Remix
210908	KAROL G	Casi Nada
75942	Red Hot Chili Peppers	Dosed
91169	Cardi B	Bodak Yellow
406012	ZAYN	iT's YoU
92016	Juice WRLD	Empty
436693	Lady Gaga	Hair
303708	Kendrick Lamar	Swimming Pools (Drank) - Extended Version
158194	Anuel AA	Na' Nuevo
481279	Sebastian Yatra	Love You Forever

Die For You by The Weeknd

In [155]:
```
# allowing songs from the same artist to pop up
Spotify_Recommendation(df).recommend_allow_same("Die For You", "The Weeknd", 10)
```

```
100%|██████████████████████████████| 419821/419821 [00:16<00:00, 25635.25it/s]
```

Out[155]:

	artists	song_name
357221	The Weeknd	Outside
357210	The Weeknd	Initiation
85979	The Weeknd	Twenty Eight
89296	The Weeknd	Secrets
88553	The Weeknd	Shameless
131570	The Weeknd	Love In The Sky
87056	The Weeknd	Earned It (Fifty Shades Of Grey) - From The "F...
294055	The Weeknd	Montreal
89203	The Weeknd	Party Monster
87985	The Weeknd	Often

In [156]:
```
# only allowing songs from different artists
Spotify_Recommendation(df).recommend_dont_allow_same("Die For You", "The Weeknd", 10)
```

```
100%|██████████████████████████████| 419756/419756 [00:16<00:00, 25655.21it/s]
```

Out[156]:

	artists	song_name
86756	BTS	Coffee
88990	Post Malone	Feeling Whitney
89126	Bad Bunny	Soy Peor
75441	Maroon 5	She Will Be Loved - Radio Mix
89096	Marshmello	Alone
405907	Shawn Mendes	Running Low
89053	Bruno Mars	Versace on the Floor
456409	Coldplay	Everyday Life
129179	Queen	Thank God It's Christmas
90959	Imagine Dragons	Bad Liar

Young and Beautiful by Lana Del Rey

In [157]:
```
# allowing songs from the same artist to pop up
Spotify_Recommendation(df).recommend_allow_same("Young And Beautiful", "Lana Del Rey", 10)
```

```
100%|██████████████████████████████| 419820/419820 [00:16<00:00, 25541.42it/s]
```

Out[157]:

	artists	song_name
118567	Lana Del Rey	Dark But Just A Game
293994	Lana Del Rey	Bel Air
156876	Lana Del Rey	Gods & Monsters
87546	Lana Del Rey	Pretty When You Cry
245087	Lana Del Rey	This Is What Makes Us Girls
263764	Lana Del Rey	Body Electric
294951	Lana Del Rey	In My Feelings
245082	Lana Del Rey	Million Dollar Man
131442	Lana Del Rey	Burning Desire
487459	Lana Del Rey	Video Games - Radio Edit

In [158]:
```
# only allowing songs from different artists
Spotify_Recommendation(df).recommend_dont_allow_same("Young And Beautiful", "Lana Del Rey", 10)
```

```
100%|██████████████████████████████| 419744/419744 [00:16<00:00, 25619.21it/s]
```

Out[158]:

	artists	song_name
437255	Henrique & Juliano	Não To Valendo Nada (Ao Vivo)
90963	Charlie Puth	Attention
85761	Romeo Santos	La Bella Y La Bestia
384639	Nirvana	Something In The Way - Devonshire Mix
114172	Fifth Harmony	Sledgehammer
210989	Paulo Londra	Relax
84082	J. Cole	Work Out
530993	Badshah	She Move It Like
112248	Chris Brown	Crawl
334339	Green Day	Last Night on Earth

Hey Jude by The Beatles

In [159]:
```
# allowing songs from the same artist to pop up
Spotify_Recommendation(df).recommend_allow_same("Hey Jude", "The Beatles", 10)
```

```
100%|██████████| 419814/419814 [00:16<00:00, 25574.58it/s]
```

Out[159]:

	artists	song_name
117070	The Beatles	Lucy In The Sky With Diamonds - Take 1
41629	The Beatles	Yer Blues - Remastered 2009
41540	The Beatles	Dear Prudence - Remastered 2009
40667	The Beatles	Blue Jay Way - Remastered 2009
43718	The Beatles	Dig A Pony - Remastered 2009
42564	The Beatles	You Never Give Me Your Money - Remastered 2009
43491	The Beatles	Let It Be - Remastered 2009
70109	The Beatles	Real Love - Anthology 2 Version
41645	The Beatles	Don't Pass Me By - Remastered 2009
73852	The Beatles	Let It Be - Remastered 2015

In [160]:
```
# only allowing songs from different artists
Spotify_Recommendation(df).recommend_dont_allow_same("Hey Jude", "The Beatles", 10)
```

```
100%|██████████| 419483/419483 [00:16<00:00, 25564.31it/s]
```

Out[160]:

	artists	song_name
229995	Sia	Clap Your Hands
583710	Avicii	Liar Liar
114750	Demi Lovato	Tell Me You Love Me
437802	Marília Mendonça	Esse Cara Aqui do Lado - Ao Vivo
53592	AC/DC	Have a Drink on Me
264091	Linkin Park	Until It's Gone
86306	Twenty One Pilots	Migraine
299669	"Guns N Roses"	Mama Kin - Live Version
55503	Michael Jackson	P.Y.T. (Pretty Young Thing)
531792	Neha Kakkar	Khyaal Rakhya Kar

White Ferrari by Frank Ocean

In [161]:
```
# allowing songs from the same artist to pop up
Spotify_Recommendation(df).recommend_allow_same("White Ferrari", "Frank Ocean", 10)
```

```
100%|██████████| 419822/419822 [00:23<00:00, 17932.19it/s]
```

Out[161]:

	artists	song_name
88996	Frank Ocean	Ivy
89102	Frank Ocean	Self Control
415039	Kelly Clarkson	Catch My Breath
114774	Frank Ocean	Provider
84432	Frank Ocean	Swim Good
89500	Frank Ocean	Solo
77630	Kelly Clarkson	Breakaway
303879	Frank Ocean	Sweet Life
85149	Frank Ocean	Lost
186984	Kelly Clarkson	Mr. Know It All

In [162]:
```
# only allowing songs from different artists
Spotify_Recommendation(df).recommend_dont_allow_same("White Ferrari", "Frank Ocean", 10)
```

```
100%|██████████████████████████████| 419793/419793 [00:16<00:00, 25703.31it/s]
```

Out[162]:

	artists	song_name
415039	Kelly Clarkson	Catch My Breath
210838	CNCO	Volverte a Ver
85033	Cartel De Santa	El Hornazo
344799	Lewis Capaldi	when the party's over - Recorded at Spotify St...
313243	Anne-Marie	Perfect To Me - Acoustic
575625	Liam Payne	Strip That Down - Acoustic
286050	MC Kevinho	O Grave Bater
71067	Daft Punk	Fresh
335455	EXO	My Turn to Cry
86610	Lorde	Buzzcut Season

Analysis

It seems like the recommender provides mostly songs from the same artist when allowed to do so. It usually returns songs of the artist we provided when we allow one artist to appear multiple times in the recommendations.

When it comes to recommending songs from different artists(excluding the input artist), the recommender generally provide songs of similar style, as illustrated above.

Section 10: Conclusion

Conclusion

In this project, we examined a spotify dataset with 586.672 songs from 1921 to 2021. We first imported and cleaned the data. Second, we conducted exploratory data analysis and visualized some inferences we made such as:

1. **Correlation Matrix** - We found that song_popularity is most positively correlated with the artist's popularity and the year is was released and most negatively correlated with the instrumentalness and acousticness of a song.
2. **Distribution of songs by release year** - It looks like most of the songs in our dataset were released in the late 90s. I'm a little surprisd to see so many songs in the late 90s in comparison to the early 2000s.
3. **Song and Artist popularity distribution** - The artist popularity is a normal distribution with most artists having a popularity between 30 and 80. Songs popularity not that much as most of them have 0 popularity and the higher you go to the popularity scale, the less songs you find.
4. **Song genre pie chart** - the most popular category is pop by far (38.7%), followed by latin (9.18%).
5. **Explicit songs distribution** - Explicit songs were very rare before the 90s with less than 2% of total songs out there. Then they gradually increased. The most notable increase happened in 2010s onwards and today they consist of 27% of all songs.
6. **Frequencies of songs in pop, rock, and rap over the years** - Pop seems that it was always a big category. Rock follows a normal distribution starting from mid 60s, peaking in the 90s, and falling again in 2010s. Rap was introduced in the 90s and since then was increasing among preferences. This can be correlated with increased explicitness in songs over the years.

Third, we preprocessed our data and prepared them for regressions to predict song popularity. We transformed the data and then we conducted linear regration with PCA, XGBoost Regressor and Random Forest Regressor which was by far our best model as it has the highest r^2 and lowest MAE and MSE. Lastly, we prepared the data for our classification model. While similar to our regression preprocessing, here we picked different features, found our classes, addressed class imbalance, and removed outliers for each class.

Overall, this project achieved our objective as the inferences allowed us to get a thorough understanding on motivation drivers about people in conjunction with trends and perception about music.

Things we would have done if we had more time and/or compute power: \

1. Webscraping unknown genres by looking up the artist on Wikipedia (We attempted this but it took almost an hour just to receive a very small percentage of the genres.
2. Finding a more effective way to classify broad genres. This was by far one of the most tedious parts of the project.
3. Webscraping lyrics for every song and using them as a feature in our recommendation model.
4. Finding the country each song was produced in. We were curious to see the distribution of songs all over the world.
5. Using AWS to increase the amount of compute power we could use. We didn't have a problem with this in the beginning phases of the project but towards the end (especially when we used RandomSearchCV) our RAM crashed.
6. Trying to find a dataset with more recent songs (from later 2021 and early 2022) to add to our dataset.