Final Project – Music Genre Classification

Name 1: Oscar Giller Name 2: Robbie Cammarano Class: CSCI 349 - Intro to Data Mining Semester: Spring 2023 Instructor: Brian King

Data for this project is sourced from: https://www.kaggle.com/datasets/purumalgi/music-genre-classification?select=submission.csv

The Problem

With the Music Genre Classification data set, we are attempting to be able to classify the genre of different songs based off of metrics such as danceability, loudness, acousticness, tempo, and many more. Similar to how applications like Spotify are able to recommend you songs based off the ones you listen to, we believe that we can also leverage this type of model to help predict similar songs to then recommend songs that the user may like and are most similar to songs they listen to. Our goal first is to create a model that is able to accurately predict the genre of songs, and then use that model to explore the different metrics that we are provided with to be able to find songs that a user may like based on the songs they are listening to.

Data Explanation

The dataset we've chosen has 17,996 rows and 17 columns. Each row represents a song and each column tells us a different thing about that song. The columns are as follows:

- Artist Name
- Track Name
- Popularity - A measure of the song's popularity (values 0-100)
- Danceability - A measure of the song's danceability (values 0.0-1.0)
- Energy - A measure of the song's energy (values 0.0-1.0)
- Key - The key the song is played in (values 0-11)
- Loudness - A measure of the song's loudness (values -40 to 1.35)
- Mode - The song's mode (binary values)
- Speechiness - A measure of the song's speechiness (values 0.0-1.0)
- Acousticness - A measure of the song's acousticness (values 0.0-1.0)
- Instrumentalness - A measure of the song's instrumentalness (values 0.0-1.0)
- Liveness - A measure of the song's liveness (values 0.0-1.0)
- Valence - A measure of the song's valence (values 0.0-1.0)
- Tempo - A measure of the song's tempo (values 30.6-217)
- Duration in Minutes/Milliseconds - How long the song takes to play. This column will need to be consolidated to use only on measurement insgtead of both milliseconds and minutes.

- Time signature - the indication of the rhythm of the song (values 1-5)
- Class (genre) - The genre of the song. This will be used as our Y column for classification.

In [157…
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
import seaborn as sns
import plotly.express as px
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
```

In [96]:
```
test = pd.read_csv('../finalProject/data/test.csv', encoding='utf-8')
train = pd.read_csv('../finalProject/data/train.csv', encoding='utf-8')
target = pd.read_csv('../finalProject/data/submission.csv', encoding='utf-8')
```

First we're going to combine our dataset from the start for consolidation. We started this by not doing that and decided it caused to many problems not to just re-asemble it from the start.

In [97]:
```
#assisted by chatGPT
def unbinarize_df(df):
    categories = list(df.columns)
    category_index = [int(c.split('_')[-1]) for c in categories]
    unbinarized_df = pd.concat([
        pd.DataFrame({'Class': [category_index[row.values.argmax()]]})
        for _, row in df.iterrows()], ignore_index=True)
    return unbinarized_df

# Use the function to un-binarize the dataframe
unbinarized_target = unbinarize_df(target)

# Print the unbinarized dataframe
unbinarized_target
```

Out[97]:

	Class
0	0
1	1
2	2
3	3
4	4
...	...
7708	0
7709	0
7710	0
7711	0
7712	0

7713 rows × 1 columns

In [98]:
```
temp = pd.concat([test, unbinarized_target], axis=1)
df_music = pd.concat([train, temp])
df_music
```

Out[98]:

	Artist Name	Track Name	Popularity	danceability	energy	key	loudness	mode	speechin
0	Bruno Mars	That's What I Like (feat. Gucci Mane)	60.0	0.854	0.564	1.0	-4.964	1	0.0
1	Boston	Hitch a Ride	54.0	0.382	0.814	3.0	-7.230	1	0.0
2	The Raincoats	No Side to Fall In	35.0	0.434	0.614	6.0	-8.334	1	0.0
3	Deno	Lingo (feat. J.I & Chunkz)	66.0	0.853	0.597	10.0	-6.528	0	0.0
4	Red Hot Chili Peppers	Nobody Weird Like Me - Remastered	53.0	0.167	0.975	2.0	-4.279	1	0.2
...	...	...	...	...	...	...	...	...	
7708	Dudu Aharon	◊ë◊ó◊ï◊® ◊®◊í◊ô◊©	28.0	0.816	0.927	7.0	-1.581	1	0.0
7709	Elephant Tree	Echoes	45.0	0.429	0.599	7.0	-7.236	0	0.0
7710	Shankar Mahadevan	Man Mohini	38.0	0.805	0.905	6.0	-7.222	0	0.2
7711	Talking Heads	Life During Wartime - 2005 Remaster	51.0	0.801	0.930	9.0	-7.365	1	0.0
7712	Marlon Craft	Lot To Give	45.0	0.630	0.867	10.0	-4.393	0	0.3

25709 rows × 17 columns

In [99]:
```
df_music.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 25709 entries, 0 to 7712
Data columns (total 17 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   Artist Name          25709 non-null  object 
 1   Track Name           25709 non-null  object 
 2   Popularity           25054 non-null  float64
 3   danceability         25709 non-null  float64
 4   energy               25709 non-null  float64
 5   key                  22887 non-null  float64
 6   loudness             25709 non-null  float64
 7   mode                 25709 non-null  int64  
 8   speechiness          25709 non-null  float64
 9   acousticness         25709 non-null  float64
 10  instrumentalness     19423 non-null  float64
 11  liveness             25709 non-null  float64
 12  valence              25709 non-null  float64
 13  tempo                25709 non-null  float64
 14  duration_in min/ms   25709 non-null  float64
 15  time_signature       25709 non-null  int64  
 16  Class                25709 non-null  int64  
dtypes: float64(12), int64(3), object(2)
memory usage: 3.5+ MB
```

In [100…

```
df_music["Popularity"] = pd.to_numeric(df_music["Popularity"], downcast = 'floa
df_music["danceability"] = pd.to_numeric(df_music["danceability"], downcast = '
df_music["energy"] = pd.to_numeric(df_music["energy"], downcast = 'float')
df_music["key"] = pd.to_numeric(df_music["key"], downcast = 'float')
df_music["loudness"] = pd.to_numeric(df_music["loudness"], downcast = 'float')
df_music["mode"] = df_music['mode'].astype(bool)
df_music["speechiness"] = pd.to_numeric(df_music["speechiness"], downcast = 'fl
df_music["acousticness"] = pd.to_numeric(df_music["acousticness"], downcast = '
df_music["instrumentalness"] = pd.to_numeric(df_music["instrumentalness"], dowr
df_music["liveness"] = pd.to_numeric(df_music["liveness"], downcast = 'float')
df_music["valence"] = pd.to_numeric(df_music["valence"], downcast = 'float')
df_music["tempo"] = pd.to_numeric(df_music["tempo"], downcast = 'float')
df_music["duration_in min/ms"] = pd.to_numeric(df_music["duration_in min/ms"],
df_music["time_signature"] = pd.to_numeric(df_music["time_signature"], downcast
df_music["Class"] = pd.to_numeric(df_music["Class"], downcast = 'integer')
```

In [101…

```
df_music.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 25709 entries, 0 to 7712
Data columns (total 17 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   Artist Name         25709 non-null  object 
 1   Track Name          25709 non-null  object 
 2   Popularity          25054 non-null  float32
 3   danceability        25709 non-null  float32
 4   energy              25709 non-null  float32
 5   key                 22887 non-null  float32
 6   loudness            25709 non-null  float32
 7   mode                25709 non-null  bool   
 8   speechiness         25709 non-null  float32
 9   acousticness        25709 non-null  float32
 10  instrumentalness    19423 non-null  float32
 11  liveness            25709 non-null  float32
 12  valence             25709 non-null  float32
 13  tempo               25709 non-null  float32
 14  duration_in min/ms  25709 non-null  float32
 15  time_signature      25709 non-null  int8   
 16  Class               25709 non-null  int8   
dtypes: bool(1), float32(12), int8(2), object(2)
memory usage: 1.8+ MB
```

In [102…
```
min_songs = df_music[df_music["duration_in min/ms"] < 30]
```

In [103…
```
ms_songs = df_music[df_music["duration_in min/ms"] > 30]
```

In [104…
```
min_songs = df_music[df_music["duration_in min/ms"] < 1000]
min_songs
```

Out[104]:

	Artist Name	Track Name	Popularity	danceability	energy	key	loudness	mo
7	Randy Travis	On the Other Hand	55.0	0.657	0.4150	5.0	-9.915000	Tr
10	Mohammed Rafi	Meri Dosti Mera Pyar	11.0	0.491	0.5630	7.0	-8.588000	Fal
13	Harald Lassen, Bram de Looze	How it feels pt. 2	34.0	0.462	0.3740	11.0	-12.069000	Fal
25	IndianRaga, Akshay Anantapadmanabhan, Madhu Iy...	Swagatham Krishna - Mohanam - Adi	NaN	0.548	0.7110	10.0	-8.440000	Tr
34	The Ayoub Sisters	Mother's Pride	16.0	0.297	0.0958	NaN	-20.316000	Tr
...	...	...	...	...	...	...	...	
7688	Anuradha Sriram, Sujatha, A.R. Rahman, Sonu Nigam	Ishq Bina	50.0	0.669	0.3570	7.0	-16.268999	Tr
7697	Kishore Kumar, Ameen Sayani	Hay Hay Yeh Nigahen	4.0	0.530	0.4630	2.0	-10.735000	Tr
7700	Au/Ra	Concrete Jungle - Acoustic	NaN	0.589	0.4670	6.0	-8.179000	Tr
7701	Joel Lyssarides, Niklas Fernqvist, Rasmus Sven...	Meditation	47.0	0.338	0.0881	4.0	-24.308001	Fal
7710	Shankar Mahadevan	Man Mohini	38.0	0.805	0.9050	6.0	-7.222000	Fal

3731 rows × 17 columns

In [105…
```
df_music["duration_in min/ms"] = df_music["duration_in min/ms"].apply(lambda x:
df_music.rename(columns={"duration_in min/ms" : 'duration_in_ms'}, inplace=True
```

In [106…
```
df_music.describe()
```

Out[106]:

	Popularity	danceability	energy	key	loudness	speechines
count	25054.000000	25709.000000	25709.000000	22887.000000	25709.000000	25709.00000
mean	44.648361	0.544836	0.662983	5.944073	-7.889735	0.07981
std	17.420223	0.165872	0.235204	3.209554	4.029282	0.08377
min	1.000000	0.000000	0.000020	1.000000	-39.952000	0.00000
25%	33.000000	0.434000	0.509000	3.000000	-9.532000	0.03480
50%	44.000000	0.546000	0.700000	6.000000	-6.988000	0.04730
75%	56.000000	0.660000	0.861000	9.000000	-5.188000	0.08310
max	100.000000	0.989000	1.000000	11.000000	1.355000	0.96000

Here we find a remove any rows with missing data

In [107…
```
mask = df_music.isna().any(axis=1)
missing_data = df_music[mask]
missing_data
```

Out[107]:

	Artist Name	Track Name	Popularity	danceability	energy	key	loudness	mode	speechin
0	Bruno Mars	That's What I Like (feat. Gucci Mane)	60.0	0.854	0.564	1.0	-4.964	True	0.0
3	Deno	Lingo (feat. J.I & Chunkz)	66.0	0.853	0.597	10.0	-6.528	False	0.0
9	Dudu Aharon	◊ì◊ï◊ì◊ï, ◊ô◊ê◊ú◊ú◊î ◊ô◊ê◊ú◊ú◊î	14.0	0.716	0.885	1.0	-4.348	False	0.0
11	Arctic Monkeys	The View From The Afternoon	59.0	0.387	0.922	9.0	-5.192	False	0.0
12	Eyal Golan	◊ß◊®◊ï◊ë◊î ◊ê◊ú◊ô◊ô◊	34.0	0.585	0.381	1.0	-7.622	False	0.0
...	...	...	...	...	...	...	...	...	
7696	Fabolous	Breathe	7.0	0.712	0.910	10.0	-5.586	False	0.2
7700	Au/Ra	Concrete Jungle - Acoustic	NaN	0.589	0.467	6.0	-8.179	True	0.0
7704	Powerwolf	We Drink Your Blood	61.0	0.344	0.920	1.0	-5.122	False	0.0
7708	Dudu Aharon	◊ë◊ó◊ï◊® ◊®◊í◊ô◊©	28.0	0.816	0.927	7.0	-1.581	True	0.0
7712	Marlon Craft	Lot To Give	45.0	0.630	0.867	10.0	-4.393	False	0.3

8833 rows × 17 columns

In [108…
```
df_music.dropna(inplace=True)
df_music.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 16876 entries, 1 to 7711
Data columns (total 17 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   Artist Name       16876 non-null  object 
 1   Track Name        16876 non-null  object 
 2   Popularity        16876 non-null  float32
 3   danceability      16876 non-null  float32
 4   energy            16876 non-null  float32
 5   key               16876 non-null  float32
 6   loudness          16876 non-null  float32
 7   mode              16876 non-null  bool   
 8   speechiness       16876 non-null  float32
 9   acousticness      16876 non-null  float32
 10  instrumentalness  16876 non-null  float32
 11  liveness          16876 non-null  float32
 12  valence           16876 non-null  float32
 13  tempo             16876 non-null  float32
 14  duration_in_ms    16876 non-null  float64
 15  time_signature    16876 non-null  int8   
 16  Class             16876 non-null  int8   
dtypes: bool(1), float32(11), float64(1), int8(2), object(2)
memory usage: 1.3+ MB
```

Now lets change the scale of some of our variables in order to standardize the metrics throughout the dataset

We are first changing the scale for the popularity vairable from 1-100 to be a 0-1 scale

In [109…
```
df_music['Popularity'] = df_music['Popularity'] / 100
df_music['Popularity'].describe()
```

Out[109]:
```
count    16876.000000
mean         0.432368
std          0.165912
min          0.010000
25%          0.320000
50%          0.430000
75%          0.540000
max          0.980000
Name: Popularity, dtype: float64
```

Now we rescale the loudness

In [110…
```
scaler = MinMaxScaler()

df_music['loudness'] = scaler.fit_transform(df_music[['loudness']])
df_music['loudness'].describe()
```

Out[110]:
```
count    16876.000000
mean         0.748063
std          0.111266
min          0.000000
25%          0.701311
50%          0.772290
75%          0.823663
max          1.000000
Name: loudness, dtype: float64
```

In [111…
```
scaler2 = MinMaxScaler()

df_music['instrumentalness'] = scaler2.fit_transform(df_music[['instrumentalnes
df_music['instrumentalness'].describe()
```

Out[111]:
```
count    16876.000000
mean         0.176897
std          0.302957
min          0.000000
25%          0.000095
50%          0.004276
75%          0.195782
max          1.000000
Name: instrumentalness, dtype: float64
```

In [112…
```
scaler3 = MinMaxScaler()

df_music['tempo'] = scaler3.fit_transform(df_music[['tempo']])
df_music['tempo'].describe()
```

Out[112]:
```
count    16876.000000
mean         0.422503
std          0.134605
min          0.000000
25%          0.317245
50%          0.409727
75%          0.508516
max          1.000000
Name: tempo, dtype: float64
```

Data Visualization and Analysis

We're going to start visualizing the data to see what we are working with. We will start with showing the distribution of the genres and from there try to show how the different columns might be distributed in each genre. We are also going to make a smaller data set of only the largest 3 genres which are: Acoustic/Folk, Indie Alt, and Rock.

In [113…
```
classes = [1, 2, 8, 9]

df_music_small = df_music[df_music['Class'].isin(classes)]

df_music_small
```

Out[113]:

	Artist Name	Track Name	Popularity	danceability	energy	key	loudness	mode	s
6	Solomon Burke	None Of Us Are Free	0.48	0.674	0.658	5.0	0.709702	False	
8	Professional Murder Music	Slow	0.29	0.431	0.776	10.0	0.821684	True	
15	Elmore James	Madison Blues	0.37	0.431	0.852	2.0	0.792158	True	
16	Dudu Aharon	◊ú◊ê ◊ô◊õ◊ï◊ú ◊ë◊ú◊¢◊ì◊ô◊ô◊ö	0.14	0.713	0.939	5.0	0.869020	False	
18	Eden Ben Zaken	◊û◊†◊í◊ô◊†◊î	0.29	0.570	0.458	9.0	0.807647	False	
...	...	...	...	...	...	...	...	...	...
17993	Smash Hit Combo	Peine perdue	0.34	0.558	0.981	4.0	0.840682	False	
17994	Beherit	Salomon's Gate	0.29	0.215	0.805	6.0	0.627642	False	
1	Crimson Sun	Essence of Creation	0.34	0.511	0.955	1.0	0.830761	True	
8	pizzagirl	car freshener aftershave	0.36	0.603	0.724	7.0	0.783688	True	
9	Dudu Tassa	◊ë◊õ◊ô◊†◊ï ◊ò◊¢◊ô◊†◊ï	0.20	0.403	0.742	7.0	0.769519	False	

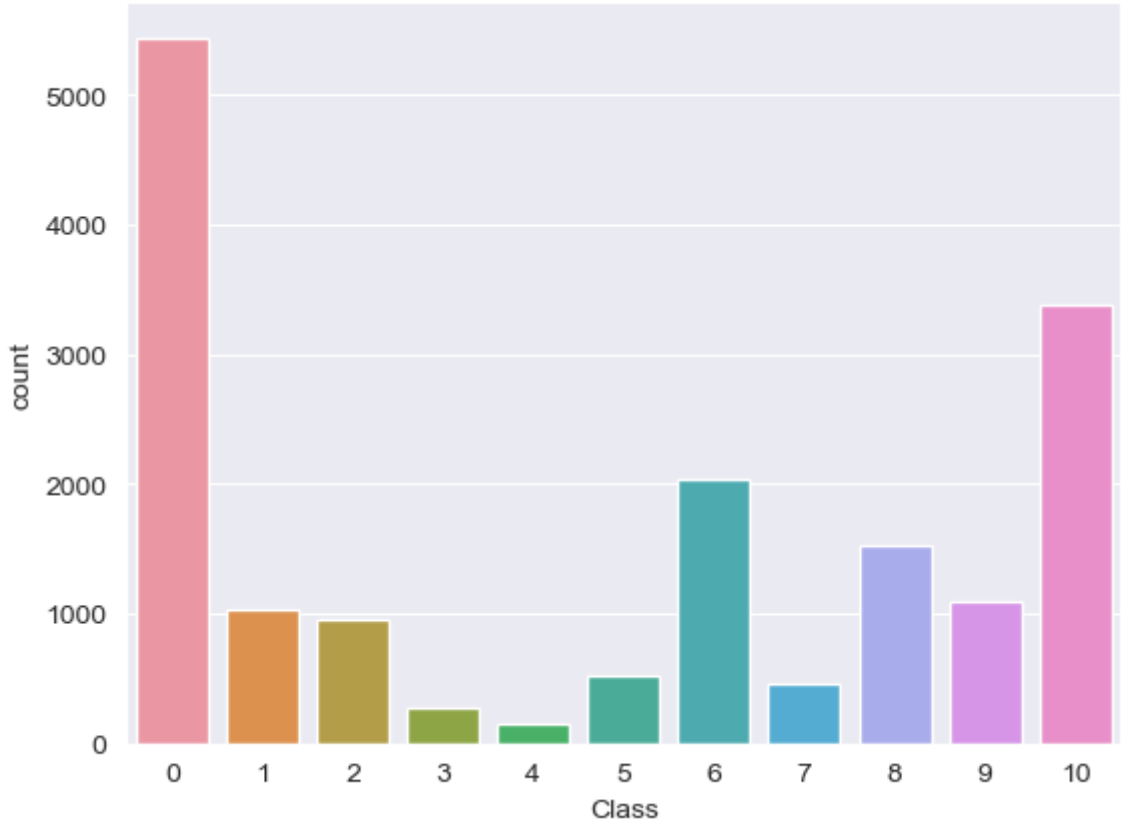
4618 rows × 17 columns

In [114…

```
sns.countplot(x="Class", data=df_music)
```

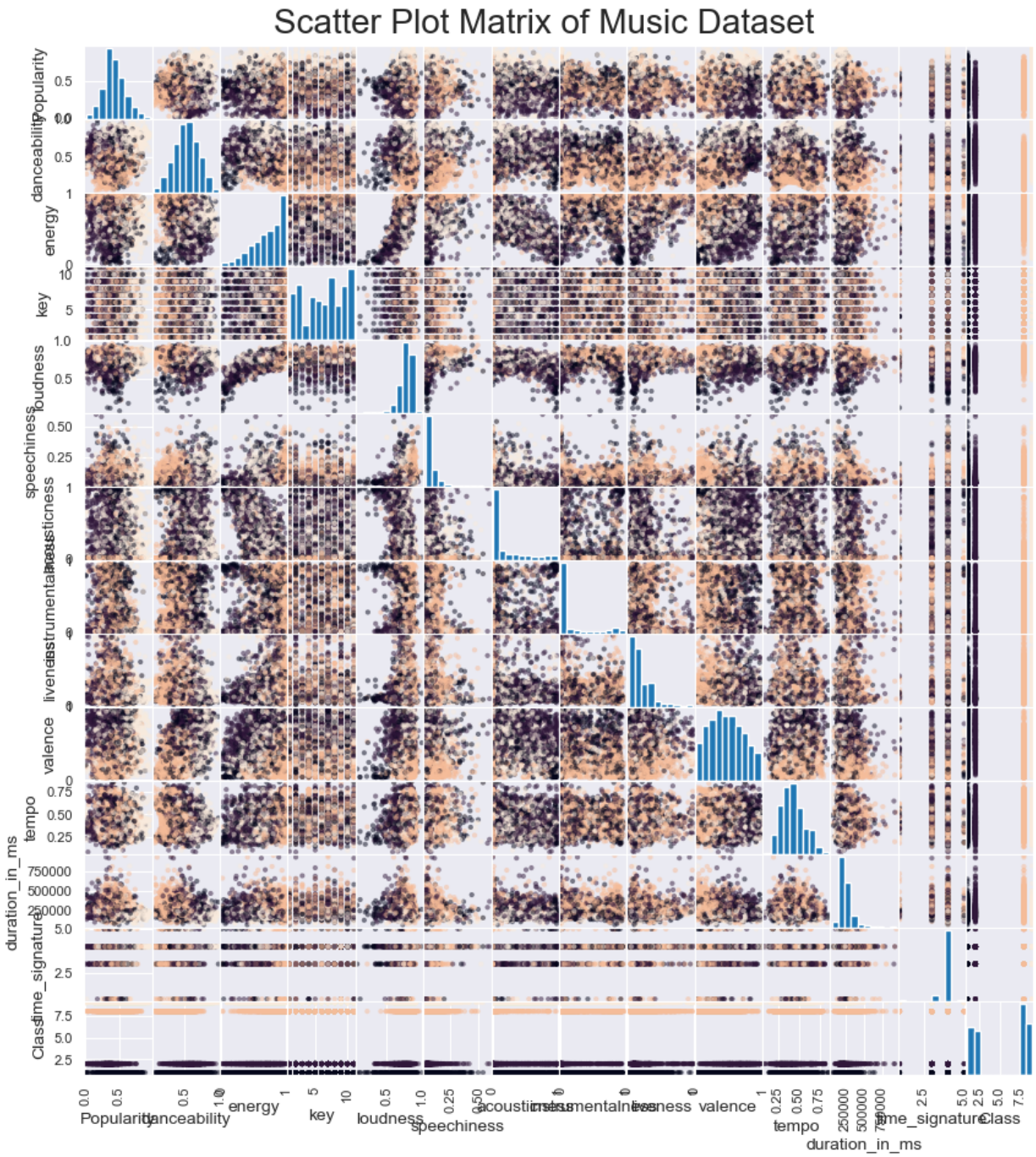
Out[114]:

```
<Axes: xlabel='Class', ylabel='count'>
```



In [115…

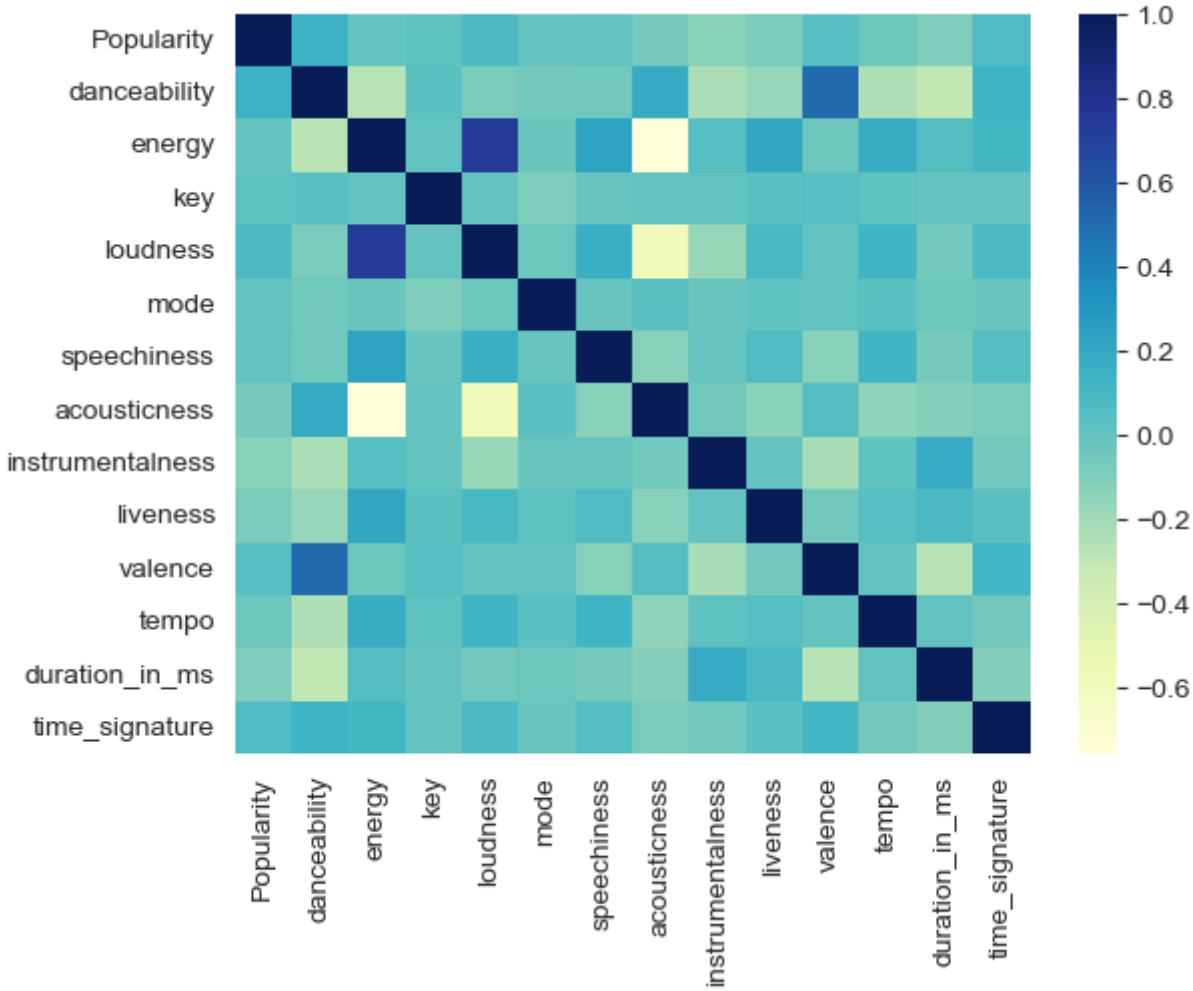
```
#colors = {'setosa': 'red', 'versicolor': 'blue', 'virginica': 'green'}
#pd.plotting.scatter_matrix(df_iris, c=df_iris['species'].apply(lambda x: colo
df_music_scatter = df_music_small.drop("mode", axis=1)
pd.plotting.scatter_matrix(df_music_scatter, c=df_music_scatter['Class'], figsi

# Set the plot labels
plt.suptitle('Scatter Plot Matrix of Music Dataset', size=20)
plt.subplots_adjust(top=0.95)
plt.show()
```



In [116…

```
df_heat = df_music_small.drop(columns=['Artist Name', 'Track Name', 'Class'])

# create the correlation matrix
corr_matrix = df_heat.corr()

# plot the correlation matrix using seaborn
sns.heatmap(corr_matrix, annot=True,  fmt="d", cmap="YlGnBu", annot_kws={"fonts

plt.show()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[116], line 7
      4 corr_matrix = df_heat.corr()
      6 # plot the correlation matrix using seaborn
----> 7 sns.heatmap(corr_matrix, annot=True,  fmt="d", cmap="YlGnBu", annot_kw
s={"fontsize": 8})
      9 plt.show()

File ~/opt/anaconda3/envs/dm/lib/python3.9/site-packages/seaborn/matrix.py:45
9, in heatmap(data, vmin, vmax, cmap, center, robust, annot, fmt, annot_kws, l
inewidths, linecolor, cbar, cbar_kws, cbar_ax, square, xticklabels, yticklabel
s, mask, ax, **kwargs)
    457 if square:
    458     ax.set_aspect("equal")
--> 459 plotter.plot(ax, cbar_ax, kwargs)
    460 return ax

File ~/opt/anaconda3/envs/dm/lib/python3.9/site-packages/seaborn/matrix.py:35
2, in _HeatMapper.plot(self, ax, cax, kws)
    350 # Annotate the cells with the formatted values
    351 if self.annot:
--> 352     self._annotate_heatmap(ax, mesh)

File ~/opt/anaconda3/envs/dm/lib/python3.9/site-packages/seaborn/matrix.py:26
0, in _HeatMapper._annotate_heatmap(self, ax, mesh)
    258 lum = relative_luminance(color)
    259 text_color = ".15" if lum > .408 else "w"
--> 260 annotation = ("{:" + self.fmt + "}").format(val)
    261 text_kwargs = dict(color=text_color, ha="center", va="center")
    262 text_kwargs.update(self.annot_kws)

ValueError: Unknown format code 'd' for object of type 'float'
```



In [117…

```
df_corr = df_music_small.corr()

df_corr
```

```
/var/folders/qg/8dyp06b13yxfb8rgzhdrsmjw0000gn/T/ipykernel_54099/2538898077.p
y:1: FutureWarning:

The default value of numeric_only in DataFrame.corr is deprecated. In a future
version, it will default to False. Select only valid columns or specify the va
lue of numeric_only to silence this warning.
```

Out[117]:

	Popularity	danceability	energy	key	loudness	mode	speec
Popularity	1.000000	0.154175	0.000902	0.014646	0.083562	-0.008813	0.0
danceability	0.154175	1.000000	-0.275063	0.020829	-0.082534	-0.044703	-0.0
energy	0.000902	-0.275063	1.000000	0.010108	0.743059	-0.020203	0.2
key	0.014646	0.020829	0.010108	1.000000	-0.004947	-0.094806	-0.(
loudness	0.083562	-0.082534	0.743059	-0.004947	1.000000	-0.035595	0.1
mode	-0.008813	-0.044703	-0.020203	-0.094806	-0.035595	1.000000	-0.(
speechiness	0.006569	-0.048386	0.234787	-0.011399	0.166232	-0.015989	1.0
acousticness	-0.057850	0.196118	-0.757456	0.003973	-0.582682	0.024877	-0.1
instrumentalness	-0.124956	-0.228960	0.044857	0.004413	-0.172405	-0.011507	-0.(
liveness	-0.084028	-0.166588	0.203714	0.024430	0.098847	0.013462	0.0
valence	0.039184	0.504641	-0.029803	0.043752	0.005482	0.005695	-0.1
tempo	-0.023059	-0.239084	0.175689	0.014087	0.122342	0.026053	0.1
duration_in_ms	-0.094807	-0.298483	0.057077	-0.005970	-0.052817	-0.025567	-0.0
time_signature	0.071825	0.128860	0.107823	-0.007376	0.082678	-0.020862	0.0
Class	0.171700	-0.056963	0.242292	-0.022659	0.253846	-0.073367	0.

From this heatmap showing all the different correlations the different categories have with each other we can see that there are a few categories that are correlated with each other. For example both the `energy` and `loudness` categories have a relatively high correlation between each other. Another pair of variables with high correlation is `valence` and `danceability`. This can help us in the future to try and relate different categories to both predict the genre of a song and also relate other songs together to eventually create a recommendation algorithm based on previous songs people have listened to.

In [118…

```
# get unique values of 'Class' column
classes = df_music_small['Class'].unique()

# generate color map using Plotly color scale
color_map = dict(zip(classes, px.colors.qualitative.Plotly))

fig = px.density_contour(df_music, x="energy", y="loudness", marginal_x="histoc

fig.show()
```

We can see that for energy and loudness that the different genres appear to be relatively clustered near each other, this is a good sign telling us that in fact the correlation between `energy` and `loudness` in terms of genre is relatively strong. However, it still may be difficult to determine the genre from only this as many of the points overlap with each other. Though there do appear to be some outliers within a few of these different genres, this is normal and we can account for those when we get to the modeling part.

Lets take a look now at two variables that are not as correlated just to understand what the distribution of the points look.

In [119…

```
fig = px.density_contour(df_music_small, x="energy", y="acousticness", marginal

fig.show()
```

From this graph we can see a much larger distribution throughout each genre category relating to `energy` and `acousticness`. The categories are either extremely clustered together or are spread out throughout the entire graph space. This shows comparing these two variables together may result in unreliable results and is not recommended to use these variables to create a model with.

Lastly, lets look at another set of variables, `valence` and `danceability`.

In [120…

```
fig = px.density_contour(df_music_small, x="valence", y="danceability", margina

fig.show()
```

We can see that despite these two variables being highly correlated with each other as we saw in the heatmap above, their distributions for these select classes are quite spread out. This can indicate that we may have to more carefully consider which columns we use when attempting to classify different genres of music.

More about the problem:

When looking at the problem that we are trying to solve, at its simplest this is a classification problem. There are a few different solutions that we can use to try and accurately determine the genre of a song. Of course since we are trying to classify the genre we could try things like multi-class classification, or multi-label classification. Additionally, we can use an alternate method such as logistic regression to help differentiate between the genres non-linear relationships.

To justify the use of a classification model, this type of modeling gives us the ability to work with imbalanced datasets, and for the case of genre classification, there are some genres in this dataset that contain more songs than others. Logistic regression is able to offset that issue by assigning different weights to the minority classes throughout training. Additionally, logistic regression can handle high-dimensional data which we are dealing with in the case of genre classification.

Another type of modeling that we can use to classify different genres is gradient boosting classification. Similar to logistic regression gradient boosting is able to handle large feature sets, and deal with different sized classes throughout training the data, and handle complex relations between different features.

Speaking in terms of our overall goal of the project, we are hoping to learn more about the classification of genres and which variables are relevant when deciding what genre a song belongs to. Additionally, we want to be able to find similar songs to each other based on either our logistic regression or gradient boosting classifier models. Where we foresee problems happening will likely be deciding which features to utilize to predict genres as well as if we will be able to predict all the genres or not with an acceptable accuracy score. Our hope is to first be able to classify genres as accurately as we can then if time permits, to be able to determine similar songs to each other.

Milestones:

1. Finish Data Prep EDA
2. Begin creating classification & k-means models
3. Train model to have adequate accuracy for genre classification
4. Be able to recommend a similar song to the user
5. Create Final Report for project

Modeling

Let's start by splitting our data and trying a Random Forrest Classifier

In [121…
```
df_music_data = df_music.iloc[:,2:]
X = df_music_data.drop('Class', axis=1)
y = df_music_data["Class"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random
```

In [122…
```
rfc = RandomForestClassifier(n_estimators=500, random_state=42)
rfc.fit(X_train, y_train)
```

Out[122]:
```
RandomForestClassifier
RandomForestClassifier(n_estimators=500, random_state=42)
```

In [123…
```
y_pred = rfc.predict(X_test)
accuracy_score(y_test, y_pred)
```

Out[123]:
```
0.333399170452301
```

Feature Importance

That accuracy wasn't very good so we're going to look at the Random Forest's estimation of how important each feature is in the data. Using this we will choose which columns we want to keep using in our next pass at using a Random Forrest Classifier

In [124…
```
rfc.feature_importances_
```

Out[124]:
```
array([0.08259221, 0.08700982, 0.08826189, 0.0485753 , 0.08156273,
       0.01468362, 0.08402825, 0.09883388, 0.08770344, 0.07494867,
       0.08499457, 0.07569619, 0.0831482 , 0.00796123])
```

After sorting, we can see that columns 11, 9, 3, 5, and 13 are all less important than the others. Let's remove them and see what our accuracy looks like afterwards.

In [125…
```
df_music_trimmed = df_music_data.drop(['time_signature', 'tempo', 'liveness', '
df_music_trimmed.head()
```

Out[125]:

	Popularity	danceability	energy	loudness	speechiness	acousticness	instrumentalness	va
1	0.54	0.382	0.814	0.773477	0.0406	0.001100	4.025104e-03	0
2	0.35	0.434	0.614	0.744347	0.0525	0.486000	1.957833e-04	0
4	0.53	0.167	0.975	0.851342	0.2160	0.000169	1.616367e-02	0
5	0.53	0.235	0.977	0.987414	0.1070	0.003530	6.063259e-03	0
6	0.48	0.674	0.658	0.709702	0.1040	0.404000	3.413659e-07	0

In [126…
```
X = df_music_trimmed.drop('Class', axis=1)
y = df_music_trimmed["Class"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random
```

In [127…
```
rfc = RandomForestClassifier(n_estimators=500, random_state=42)
rfc.fit(X_train, y_train)
y_pred = rfc.predict(X_test)
accuracy_score(y_test, y_pred)
```

Out[127]:
```
0.33833695437487654
```

In [128…
```
rfc.feature_importances_
```

Out[128]:
```
array([0.10549451, 0.11106557, 0.11267652, 0.10483141, 0.10904888,
       0.12415397, 0.11362805, 0.11035907, 0.10874202])
```

As we can see this didn't help performance very much. This is probably due to the way the difference is class sizes we saw during the EDA section.

Equalizing the data

Because of the skewed class sizes we are dealing with, we are going to use a subset of that classes to try and get better results.

In [129…
```
classes = [1, 2, 8, 9]

df_music_small = df_music[df_music['Class'].isin(classes)]
```

```
df_music_small.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4618 entries, 6 to 9
Data columns (total 17 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   Artist Name       4618 non-null   object 
 1   Track Name        4618 non-null   object 
 2   Popularity        4618 non-null   float32
 3   danceability      4618 non-null   float32
 4   energy            4618 non-null   float32
 5   key               4618 non-null   float32
 6   loudness          4618 non-null   float32
 7   mode              4618 non-null   bool   
 8   speechiness       4618 non-null   float32
 9   acousticness      4618 non-null   float32
 10  instrumentalness  4618 non-null   float32
 11  liveness          4618 non-null   float32
 12  valence           4618 non-null   float32
 13  tempo             4618 non-null   float32
 14  duration_in_ms    4618 non-null   float64
 15  time_signature    4618 non-null   int8   
 16  Class             4618 non-null   int8   
dtypes: bool(1), float32(11), float64(1), int8(2), object(2)
memory usage: 356.3+ KB
```

In [130…

```
df_music_data_sm = df_music_small.iloc[:,2:]
#, 'duration_in_ms', 'key', 'mode'
X = df_music_data_sm.drop(['Class'], axis=1)
Y = df_music_data_sm['Class']

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2)
```

Now that we've got the data subset, lets run some models on it

Gradient Boosting Classifier

In [131…

```
gb = GradientBoostingClassifier(n_estimators=50, max_depth=5, learning_rate=0.1
gb.fit(X_train, y_train)

y_pred = gb.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
```

```
Accuracy: 0.6699134199134199
```

Decision Trees

In [132…

```
dtc = DecisionTreeClassifier(max_depth=4)
dtc.fit(X_train, y_train)

y_pred = dtc.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
```

```
Accuracy: 0.5497835497835498
```

Logisitic Regression

In [133…
```
lr = LogisticRegression(max_iter=50000)
lr.fit(X_train, y_train)

y_pred = lr.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
```

```
Accuracy: 0.35064935064935066
```

Random Forests Classifier

In [134…
```
rfc = RandomForestClassifier(n_estimators=500)
rfc.fit(X_train, y_train)
y_pred = rfc.predict(X_test)
accuracy_score(y_test, y_pred)
```

Out[134]:
```
0.6807359307359307
```

These results are much better! Since Random Forests had a good score, we're going to do a feature importance test again and can remove the columns that don't affect much.

In [135…
```
rfc.feature_importances_
```

Out[135]:
```
array([0.09533567, 0.10462295, 0.10946438, 0.0320086 , 0.06686981,
       0.00996022, 0.06279717, 0.15502488, 0.07516229, 0.0511081 ,
       0.10230425, 0.05316834, 0.07478931, 0.00738402])
```

Let's cut everything below 5% signifigance and see how that changes the random forrest and the gradient boosting classifier:

In [136…
```
X.head()
```

Out[136]:

	Popularity	danceability	energy	key	loudness	mode	speechiness	acousticness	instrur
6	0.48	0.674	0.658	5.0	0.709702	False	0.1040	0.404000	3.4
8	0.29	0.431	0.776	10.0	0.821684	True	0.0527	0.000022	1.3
15	0.37	0.431	0.852	2.0	0.792158	True	0.0431	0.564000	2.0
16	0.14	0.713	0.939	5.0	0.869020	False	0.0372	0.110000	4.8
18	0.29	0.570	0.458	9.0	0.807647	False	0.0236	0.502000	1.3

In [137…
```
df_music_data_sm = df_music_small.iloc[:,2:]
#, 'duration_in_ms', 'key', 'mode'
X = df_music_data_sm.drop(['Class', 'key', 'mode', 'time_signature'], axis=1)
Y = df_music_data_sm['Class']

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2)
```

In [143…
```
gb = GradientBoostingClassifier(n_estimators=50, max_depth=5, learning_rate=0.1
gb.fit(X_train, y_train)
```

```
y_pred = gb.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
```

```
Accuracy: 0.7012987012987013
```

In [144…
```
rfc = RandomForestClassifier(n_estimators=500)
rfc.fit(X_train, y_train)
y_pred = rfc.predict(X_test)
accuracy_score(y_test, y_pred)
```

Out[144]:
```
0.7012987012987013
```

Feature Importance Results

Although we tried to use feature importance to trip our data down a prioritize the important columns, our accuracy went down. Because of this we will reset the data to its previous state.

In [145…
```
#resets the x and y splits

df_music_data_sm = df_music_small.iloc[:,2:]
X = df_music_data_sm.drop(['Class'], axis=1)
Y = df_music_data_sm['Class']
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2)
```

Parameter Search

Since the Gradient Boosting Classifier has been our best model so far, we are now going to run a parameter search on it. This will try a ton of different parameter combinations and return the best one.

In [146…
```
# param_grid = {
#     'learning_rate': [0.01, 0.1, 1],
#     'n_estimators': [50, 100, 500],
#     'max_depth': [3, 5, 7]
# }
#
# # Initialize a GradientBoostingClassifier
# gb = GradientBoostingClassifier()
#
# # Initialize the GridSearchCV object
# grid_search = GridSearchCV(gb, param_grid, cv=5, verbose=1)
#
# # Fit the GridSearchCV object to the data
# grid_search.fit(X_train, y_train)
#
# # Print the best parameters and the corresponding score
# print("Best parameters: ", grid_search.best_params_)
# print("Best score: ", grid_search.best_score_)
```

Results:

Running this took a very long time run, so we have commented out the actual grid search and will be reporting the output from our other file here:

Fitting 5 folds for each of 27 candidates, totalling 135 fits Best parameters: {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 50} Best score: 0.6415748961278517

So we have now run a grid search for the best parameters to use with the best model we have tested. Our final model is the Gradient Boosting Classifier with a learning rate of 0.1, a max depth of 5, and n estimators of 50.

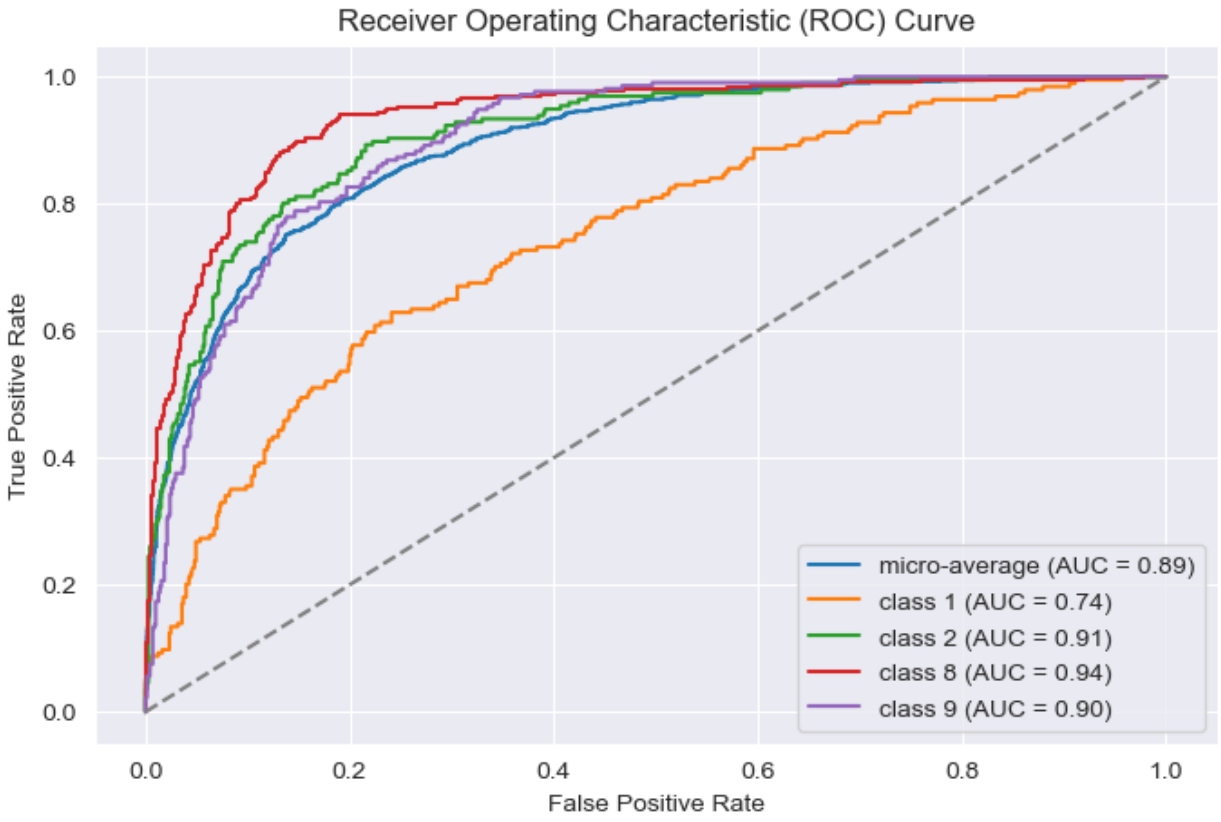
To see the results from our modeling file look in Modeling_2.ipynb.

In [152…
```
gb = GradientBoostingClassifier(n_estimators=50, max_depth=5, learning_rate=0.1
gb.fit(X_train, y_train)

y_pred = gb.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
```

```
Accuracy: 0.6731601731601732
```

In [153…
```
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
from sklearn.multiclass import OneVsRestClassifier


# Predict the probabilities for each class
y_pred_prob = gb.predict_proba(X_test)

# Convert the true labels to binary format
y_test_bin = label_binarize(y_test, classes=[1, 2, 8, 9])

# Create a One-vs-Rest Classifier
ovr = OneVsRestClassifier(gb)

# Fit the OvR Classifier on the training data
ovr.fit(X_train, y_train)

# Predict the probabilities for each class using the OvR Classifier
y_pred_prob_ovr = ovr.predict_proba(X_test)

# Compute the ROC curve and AUC for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(len(classes)):
    fpr[i], tpr[i], _ = roc_curve(y_test_bin[:, i], y_pred_prob[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and AUC
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_bin.ravel(), y_pred_prob_ovr.r
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Plot the ROC curve for each class and micro-average
plt.figure(figsize=(8, 5))
plt.plot(fpr["micro"], tpr["micro"], label='micro-average (AUC = {0:0.2f})'.for
for i in range(len(classes)):
```

```
    plt.plot(fpr[i], tpr[i], label='class {0} (AUC = {1:0.2f})'.format(classes[
plt.plot([0, 1], [0, 1], linestyle='--', color='grey')
plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```

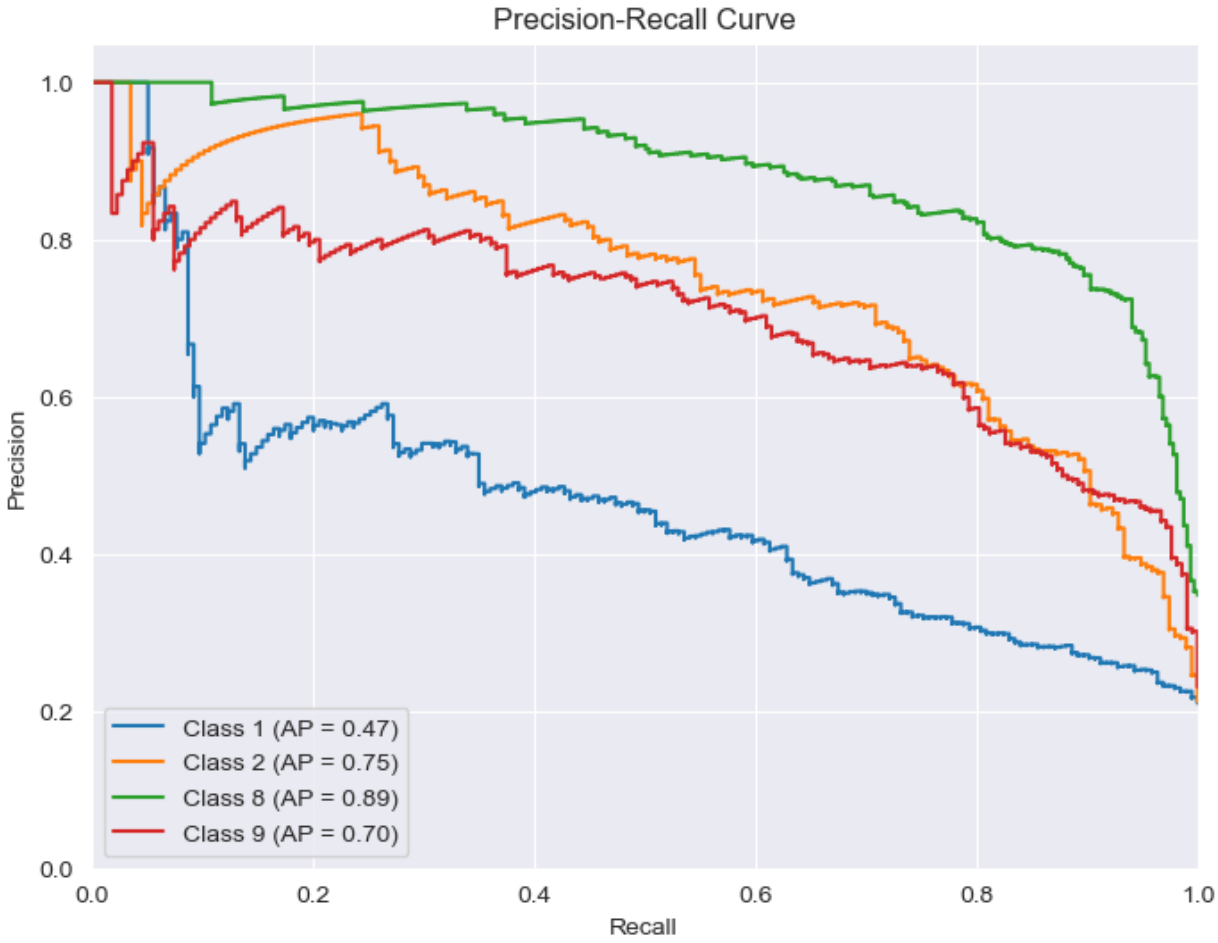


For our ROC curve we can see that all of our classes performed quite well. Each of the classes that we were predicting on had an AUC of at least 0.76, with the rest being all 0.90 or above. From this ROC curve we can see that it clearly indicates that our model has high discrimination power, which means our model is making accurate predictions and has a good balance between true positive and false positive rates.

In [154…

```
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import average_precision_score


# compute the precision and recall for each class
precision = dict()
recall = dict()
average_precision = dict()
for i in range(len(classes)):
    precision[i], recall[i], _ = precision_recall_curve(y_test_bin[:, i], y_pre
    average_precision[i] = average_precision_score(y_test_bin[:, i], y_pred_pro

# Plot the precision/recall curve for each class
plt.figure(figsize=(8,6))
plt.step(recall[0], precision[0], where='post', label='Class 1 (AP = {:.2f})'.f
```

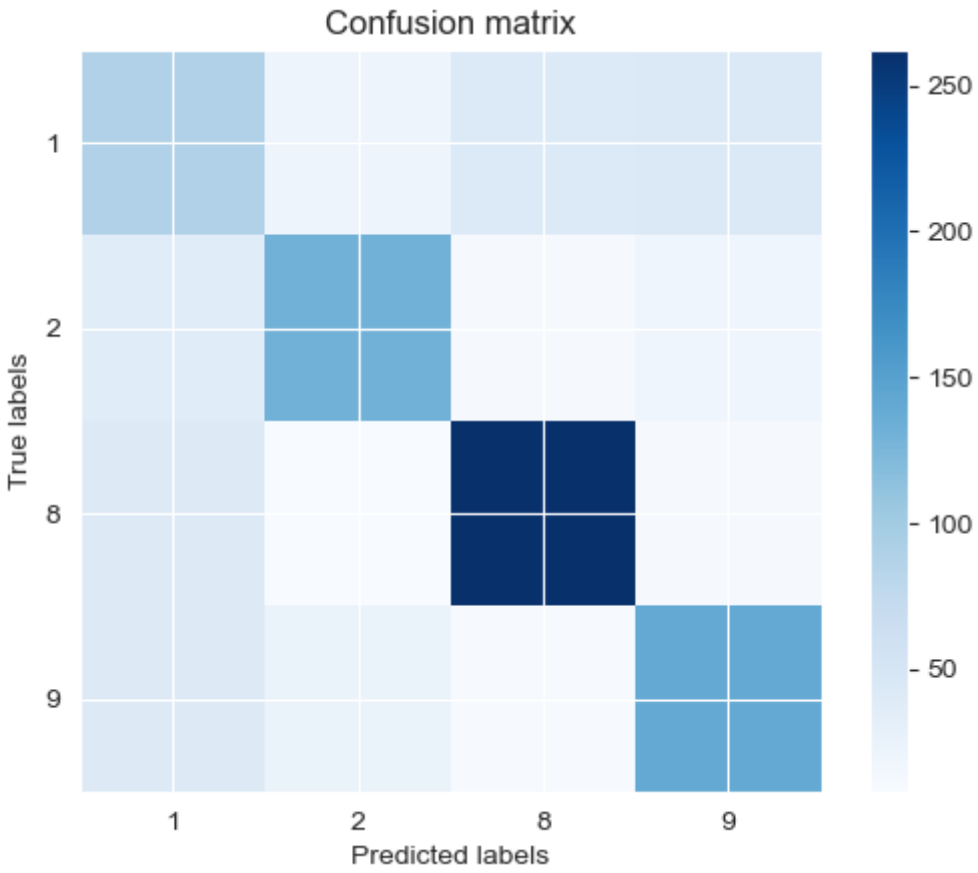
```
plt.step(recall[1], precision[1], where='post', label='Class 2 (AP = {:.2f})'.f
plt.step(recall[2], precision[2], where='post', label='Class 8 (AP = {:.2f})'.f
plt.step(recall[3], precision[3], where='post', label='Class 9 (AP = {:.2f})'.f
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Precision-Recall Curve')
plt.legend()
plt.show()
```



Our Precision-Recall curve also seems to indicate good results. With all but one class having an AP score of over 0.70, this shows that overall most of our classes have a solid precision-recall tradeoff.

In [155…

```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, y_pred)

plt.imshow(cm, cmap=plt.cm.Blues)
plt.xlabel("Predicted labels")
plt.ylabel("True labels")
plt.xticks(np.arange(len(classes)), classes)
plt.yticks(np.arange(len(classes)), classes)
plt.title('Confusion matrix')
plt.colorbar()
plt.show()
```



From this confusion matrix we can see that the best performing class was class 8, which makes sense because it is the largest class in terms of total entries. Our worst performing class is class 1, and classes 2 and 9 appear to perform adequately.

In [156…

```
from sklearn.metrics import classification_report

# Generate classification report and convert to a pandas dataframe
report = classification_report(y_test, y_pred, output_dict=True)
classification = pd.DataFrame(report).transpose()

# Filter dataframe to show only precision, recall, and f1-score for each class
class_metrics = classification.loc[classification.index.isin(['1', '2', '8', '9

# Rename index to 'Class x' for readability
class_metrics.index = ['Class 1', 'Class 2', 'Class 8', 'Class 9']

# Print the dataframe
print(class_metrics)
```

```
         precision    recall  f1-score
Class 1   0.429952  0.458763  0.443890
Class 2   0.715847  0.668367  0.691293
Class 8   0.811146  0.816199  0.813665
Class 9   0.663507  0.657277  0.660377
```

Lastly our prediction performance table further validates our previous analysis again showing that class 8 performs the best compared to all the other classes, while class 1 has the worst performance.

Discussion

Throughout the process of creating this project, we struggled with finding the right model to classify our genres. At first all of the models we ran had accuracy around 30%, this indicated to us that maybe trying to classify over 10 different genres with high accuracy was not something completely feasible with the data we had. So from there we decided to lower the amount of genres we were attempting to classify to four.

From there we were able to get better results using gradient boosting and random forest models. From our data preparation, we created a correlation matrix and attempted to utilize that to pick different features to use to classify our genres. This helped us identify important features that would help us create the most accurate model and understand which ones were not useful. The features that we found to be most correlated were `energy` and `loudness`, as well as `valence` and `danceability`. This was further portrayed later in our data preparation through looking at the distribution of these points compared over all the different genres.

In terms of the hyperparameters for both gradient boosting and random forest models, the two models have a parameter in common: n_estimators. In the end, for the gradient boosting classifier we used n_estimators equal to 50 and random forests equal to 500.

Conclusion and Summary of our Findings

After working hard on this project I think we can safely say that, although this was a fun project to work on, this dataset seems to be a little too subjective for actual music classification. We found this dataset on Kaggle and it has been used for Hackathons and for things like this before, however, there was never any explanation of how some of the column values were decided. We don't know how whever created this dataset decided on each song's dancability or speechiness. This is part of what made it so interesting to work with though.

EDA: The EDA for this dataset showed us from the start how convoluted this classification project would be. There were many columns and some correlation between them and the classes, but there was never anything nearly as clear as the iris dataset we worked on in lab. There was also an uneven distribution in the numbers of rows for each class with Hip-Hop having almost a third of the data alone. It was these results from our EDA that led us to make the decisions we did when choosing what data to use in our modeling.

Modeling: The modeling part of this project started off poorly with us get about 30% accuracy every time. It wasn't long before we realized the model must just be classifying the hip-hop class. This is what led us to cut down on the number of classes we were looking at. After this we tried a number of different models and got the best results from Random Forrest and Gradient Boost Classifiers. Eventually we went with the Gradient Boost Classifier as our final model because it tended to have better results, although we dif try to use the Random Forrest's feature importance to cut down on the number of columns we were looking at. Finally, we ran a param search and now found the best parameters for the job.

We finished with a Gradient Boosting Classifier with a learning rate of 0.1, a max depth of 5, n estimators of 50, and an accuracy of 67%.

In [ ]: