Machine Learning

Final Homework Report

Akıncan Kılıç - 190315044

---

0) Preparation / Project Layout

First, I had to decide which programming language that I was going to use in order to train my networks and prepare my data.

I decided to use **Python** since it is one of the leading programming languages for Machine Learning and Data Preparation and Visualisation.

For most of my Data visualizations in the homework I have used the **Seaborn** library and **MatPlotLib** in Python.

For training and selecting my models I have used the **scikit-learn** library, I have done classifications using all 5 methods that were said on the homework:

- Support Vector Machine
- Logistic Regression
- K-Nearest Neighbours
- Decision Tree
- Artificial Neural Network

I have written my code completely in an ***Object Oriented*** manner. I have created a class just for my data, because I didn't want to repeat the same data preparation actions for every model, and also wanted my code to be organised.

So, I have created a new Class called **MusicData()** and I wrote this on a new python file called **my_data.py**

I have created a file called **utility.py** and wrote my utility codes there such as keeping track of time and printing how long a model took to learn, and also to generate the heatmap plot for the model predictions.

I also have written every model on a new python file in this manner: **model_name.py**

I have two sub-directories called ***plots*** and ***dataset***. I keep my plots on the plots subfolder for organisation and I have my dataset under the dataset subfolder.

---

1) Data Exploration

First of all I had to do some Data Exploration

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50005 entries, 0 to 50004
Data columns (total 18 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   instance_id       50000 non-null  float64
 1   artist_name       50000 non-null  object 
 2   track_name        50000 non-null  object 
 3   popularity        50000 non-null  float64
 4   acousticness      50000 non-null  float64
 5   danceability      50000 non-null  float64
 6   duration_ms       50000 non-null  float64
 7   energy            50000 non-null  float64
 8   instrumentalness  50000 non-null  float64
 9   key               50000 non-null  object 
 10  liveness          50000 non-null  float64
 11  loudness          50000 non-null  float64
 12  mode              50000 non-null  object 
 13  speechiness       50000 non-null  float64
 14  tempo             50000 non-null  object 
 15  obtained_date     50000 non-null  object 
 16  valence           50000 non-null  float64
 17  music_genre       50000 non-null  object 
dtypes: float64(11), object(7)
memory usage: 6.9+ MB
```

We have 50005 entries on our dataset and 18 columns.

Here are some information about numerical columns:

	instance_id	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	liveness	loudness	speechiness	valence
count	50000.000000	50000.000000	50000.000000	50000.000000	5.000000e+04	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000
mean	55888.396360	44.220420	0.306383	0.558241	2.212526e+05	0.599755	0.181601	0.193896	-9.133761	0.093586	0.456264
std	20725.256253	15.542008	0.341340	0.178632	1.286720e+05	0.264559	0.325409	0.161637	6.162990	0.101373	0.247119
min	20002.000000	0.000000	0.000000	0.059600	-1.000000e+00	0.000792	0.000000	0.009670	-47.046000	0.022300	0.000000
25%	37973.500000	34.000000	0.020000	0.442000	1.748000e+05	0.433000	0.000000	0.096900	-10.860000	0.036100	0.257000
50%	55913.500000	45.000000	0.144000	0.568000	2.192810e+05	0.643000	0.000158	0.126000	-7.276500	0.048900	0.448000
75%	73863.250000	56.000000	0.552000	0.687000	2.686122e+05	0.815000	0.155000	0.244000	-5.173000	0.098525	0.648000
max	91759.000000	99.000000	0.996000	0.986000	4.830606e+06	0.999000	0.996000	1.000000	3.744000	0.942000	0.992000

I went through each column and identified their purposes and came to a conclusion about what to do with them:

Instance ID

This is just the index for the dataset, we do not need this column so I will **drop** this feature.

Artist Name

```
data['artist_name'].describe()
```

	artist_name
count	50000
unique	6863
top	empty_field
freq	2489

Here we see that we have 6863 unique artist names in our dataset, also the most repeating value is “empty_field”.

Using this feature would help us predict the music genre greatly since one artist tends to stick to one genre for their career, but I will still remove this feature from the dataset because if I keep this in I will have to do One Hot Encoding to be able to use this feature, which will result in 6863 extra columns in the dataset!

The other problem is that if I provide the artist name, since it’s heavily related with the Music Genre, my model could easily **overfit** and only rely on this feature to predict the genres, so when we encounter a new artist on the test set, it will fail to predict because it hasn’t learned how to do predictions without this feature.

- ★ Moreover, I want to be able to predict a song genre without having any clue about the artist, just by looking at the song statistics, this way my models are more generalised.

Track Name

```
data['track_name'].describe()
```

	track_name
count	50000
unique	41699
top	Home
freq	16

Unlike the artist_name which would have helped us predict genres but would cause many problems, this feature is useless to us, since track names are not related at all with the genre of the music or any other features. This is just an arbitrary name the artist picks for their song. So I will **drop** this feature.

Obtained Date

```
data['obtained_date'].describe()
```

	obtained_date
count	50000
unique	5
top	4-Apr
freq	44748

```
data['obtained_date'].unique()
```

	0
0	4-Apr
1	3-Apr
2	5-Apr
3	1-Apr
4	NaN
5	0/4

This feature only gives us when the data was obtained, not useful for our prediction model so I will **drop** it.

Popularity

```
data['popularity'].describe()
```

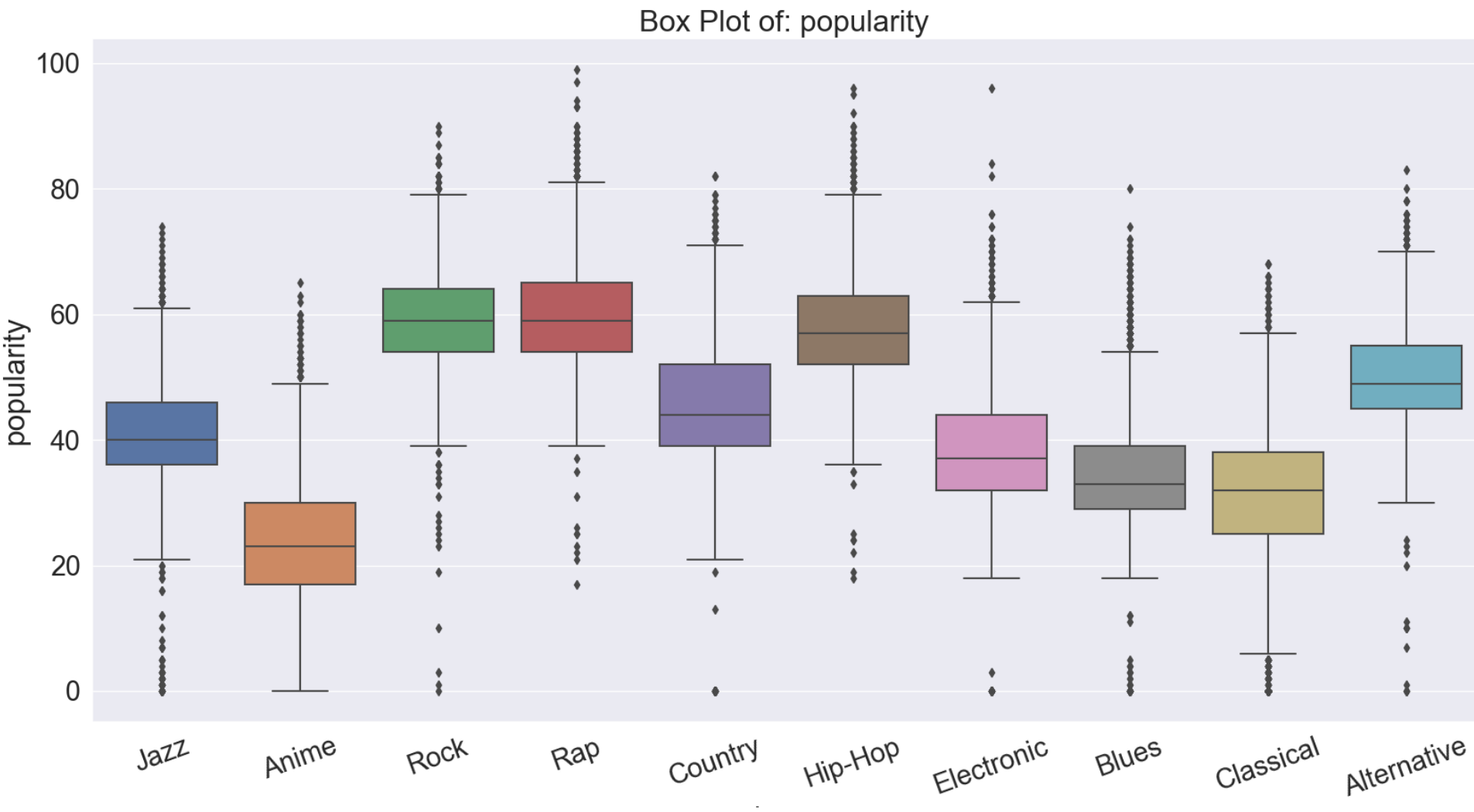
	popularity
count	50000.000000
mean	44.220420
std	15.542008
min	0.000000
25%	34.000000
50%	45.000000
75%	56.000000
max	99.000000

We see that this features values ranges from 0 to 99.

I wanted to see the relation of this feature to the music genre so,

+ I have written a code to generate Box Plots of features according to the Music Genre.
+ I have used this code to generate future Box Plots whenever I wanted.

```
def get_box_plots(self, features):
    for i, feature in enumerate(features):
        fig = plt.figure(figsize=(20, 10))
        sns.set(style="darkgrid", font_scale=2)
        sns.boxplot(data=self.df, x='music_genre', y=feature)
        plt.title(f"Box Plot of: {feature}")
        plt.xticks(rotation=20)
        plt.savefig(f'./plots/box_plots/{feature}.png')
        plt.close()
```



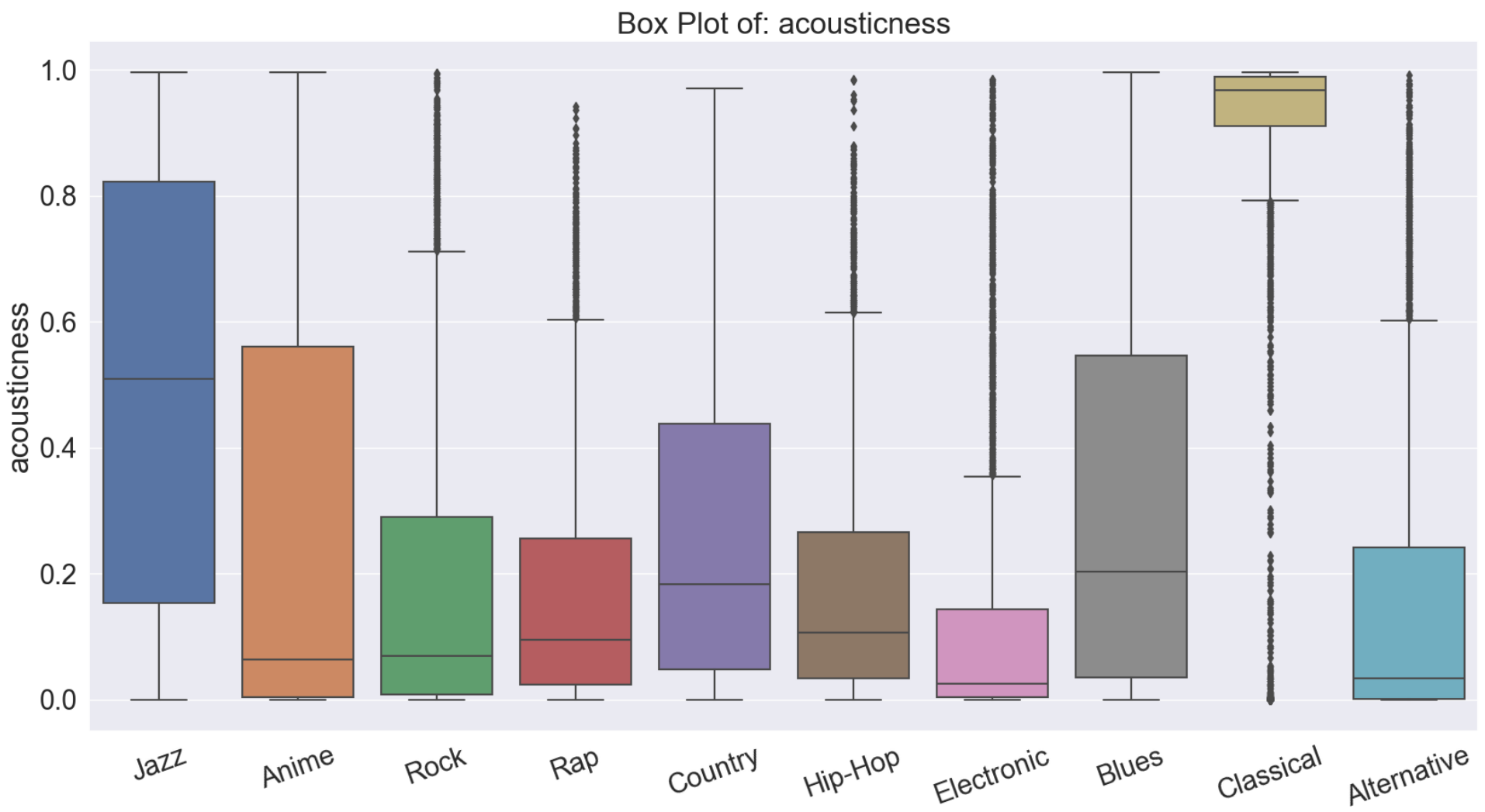
- From this box plot we can see that the popularity feature is very diverse across genres and it will help us predict the music genres.
- We can observe that Anime, Classical, Blues genres are the least popular ones while Hip-Hop, Rap, Rock are the leading genres in popularity.

Acousticness:

```
data['acousticness'].describe()
```

	acousticness
count	50000.000000
mean	0.306383
std	0.341340
min	0.000000
25%	0.020000
50%	0.144000
75%	0.552000
max	0.996000

Our values are between 0 and 1.



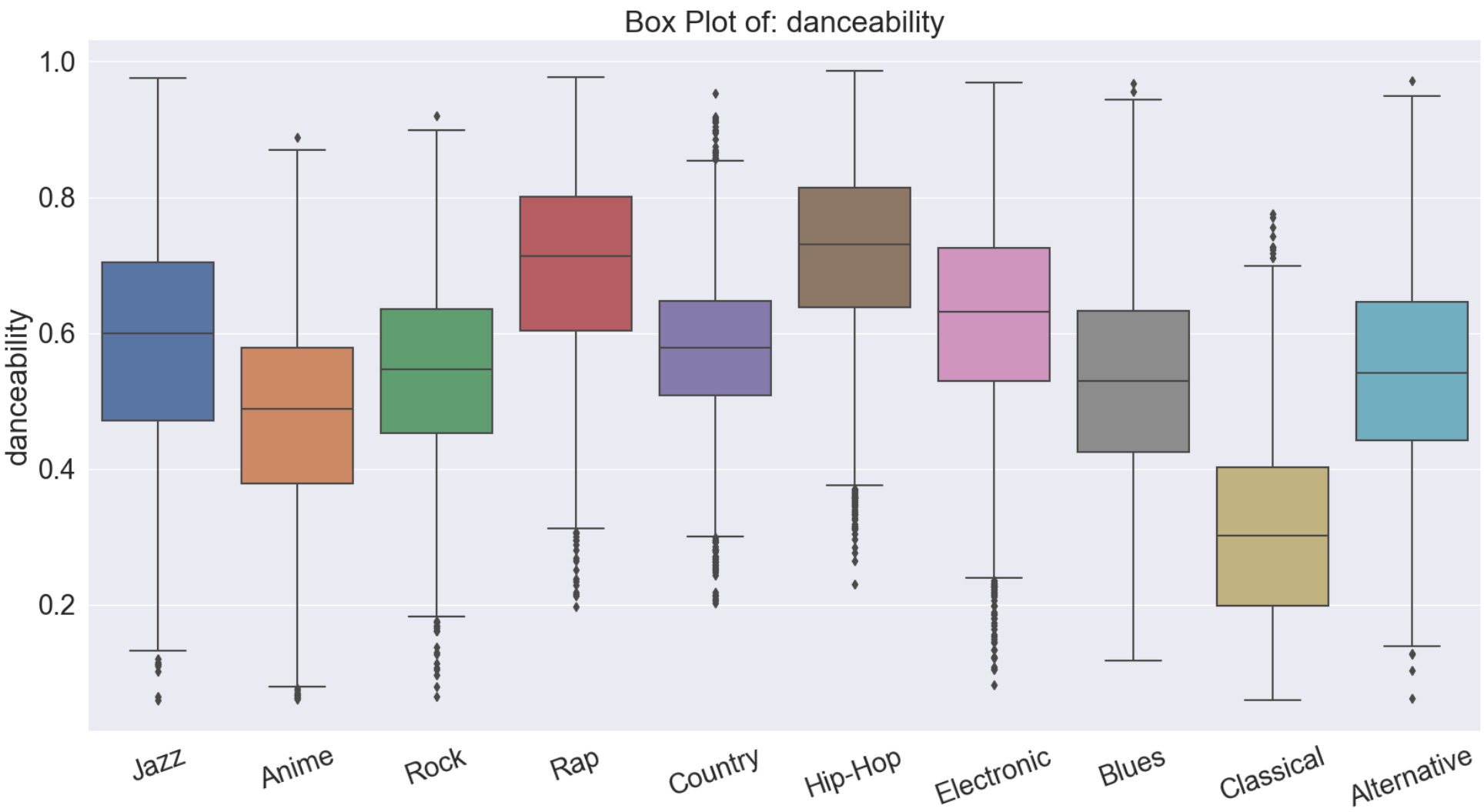
- This is another amazing feature that will help us greatly at predicting our genres since we have much diversity.
- We can see that Classical genre has a large amount of acousticness meanwhile genres such as Electronic, Hip-Hop, Alternative have almost no acousticness.

Danecability

```
data['danceability'].describe()
```

	danceability
count	50000.000000
mean	0.558241
std	0.178632
min	0.059600
25%	0.442000
50%	0.568000
75%	0.687000
max	0.986000

Our values are between 0.05 and 1.



- Again we observe the Classical genre being distinctive while Rap and Hip-Hop genres seem pretty similar.
- This is another great feature for us to use for our prediction models.

Duration MS

```
data['duration_ms'].describe()
```

	duration_ms
count	5.000000e+04
mean	2.212526e+05
std	1.286720e+05
min	-1.000000e+00
25%	1.748000e+05
50%	2.192810e+05
75%	2.686122e+05
max	4.830606e+06

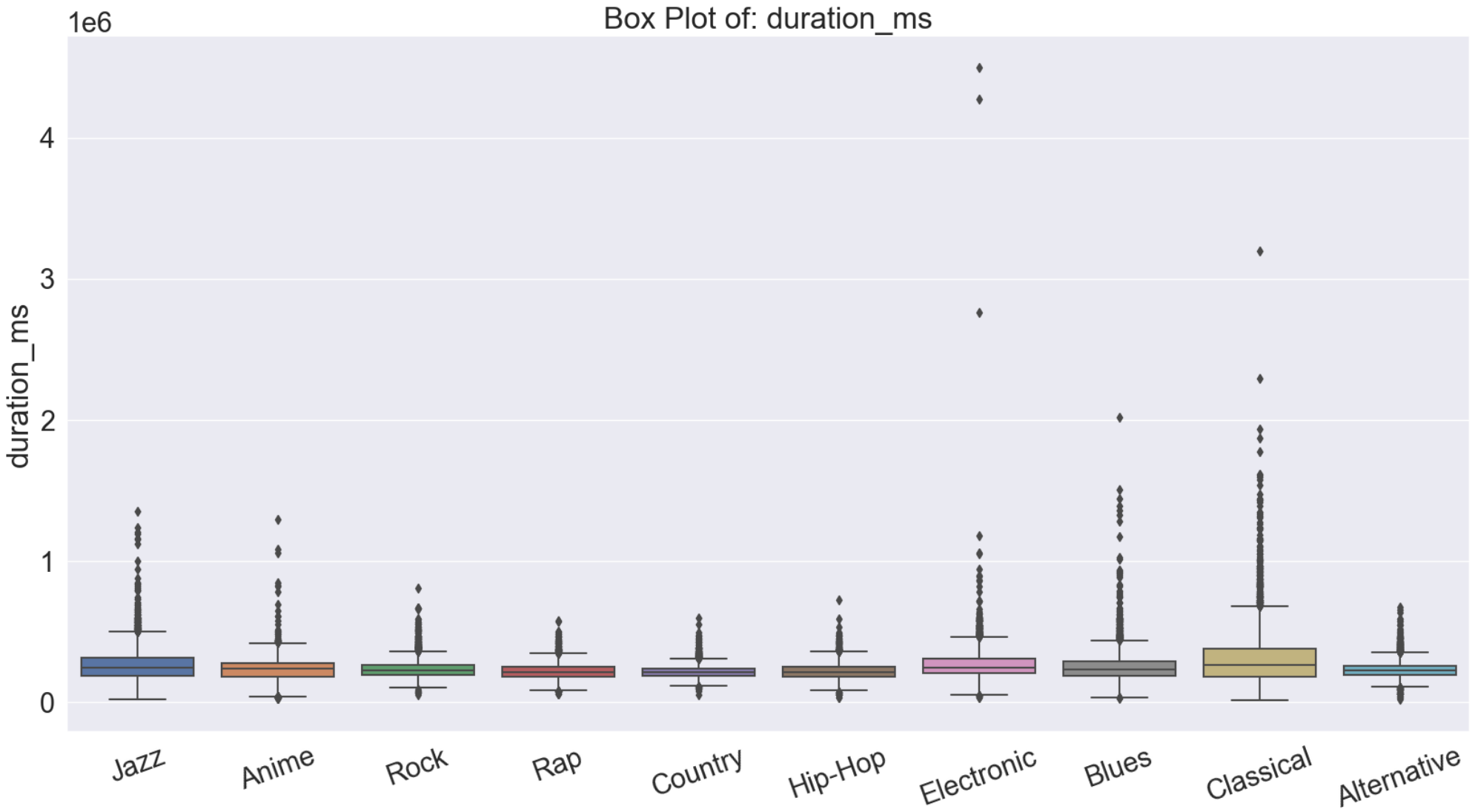
Here I observed an issue about the dataset, from the “min” we can see that we have “-1” millisecond durations for a song which is not possible.

```
data[data['duration_ms'] == -1].shape
```

```
(4939, 18)
```

We see that there are 4939 rows that have the duration of -1 milliseconds.

I could fill these values with the median of the column but I have instead decided to remove them entirely since there are many outliers in this feature. I could be lowering the accuracy of my model by providing wrong information about a genre.



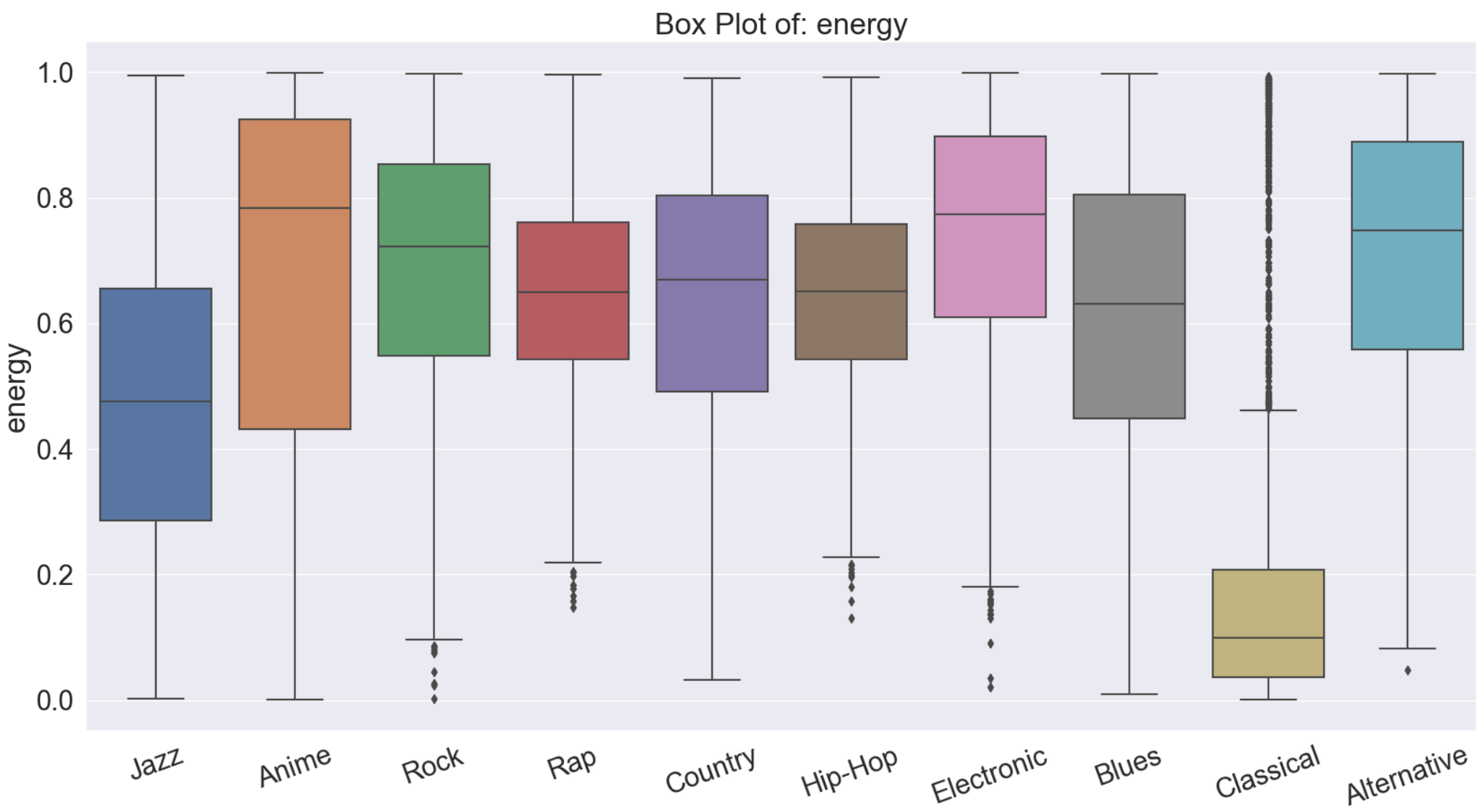
- From the box plot we can see that Electronic music and Classical music have extreme outliers and the others have many outliers as well.
- This is why I didn't want to fill the "-1" durations with medians.

Energy

```
data['energy'].describe()
```

	energy
count	50000.000000
mean	0.599755
std	0.264559
min	0.000792
25%	0.433000
50%	0.643000
75%	0.815000
max	0.999000

We see our energy feature values are between 0 and 1 here.



We observe many similar things again.

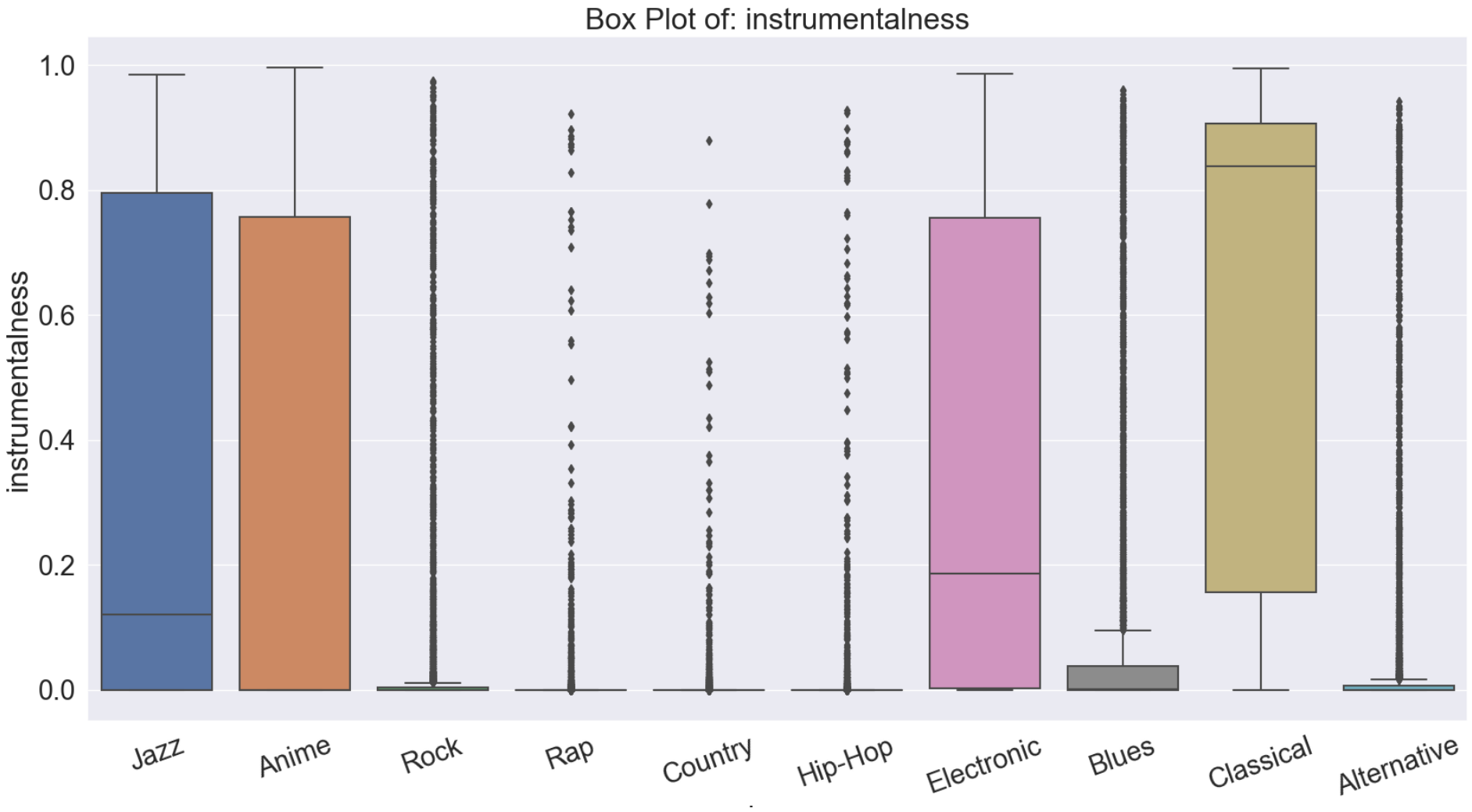
- The Classical genre is the heavy outlier while Rap and Hip-Hop are almost identical.
- We also observe Alternative and Jazz acting a bit diverse compared to the other features.
- This is another great feature we can use at prediction.

Instrumentalness

```
data['instrumentalness'].describe()
```

	instrumentalness
count	50000.000000
mean	0.181601
std	0.325409
min	0.000000
25%	0.000000
50%	0.000158
75%	0.155000
max	0.996000

Our values are between 0 and 1 again.



This box plot looks very interesting, it seems to have so many 0.0 values that everything else seems like an outlier. This seems suspicious. So I checked the 0 values.

```
data[data['instrumentalness'] == 0].shape
```

```
(15001, 18)
```

We have 15001 amounts of 0 values on instrumentalness. This is surely an issue with the dataset.

This feature will still be useful to us, so I want to use it. But I can't remove 15.000 rows from my dataset just to be able to use this feature. So instead, I will fill these missing features as the median of the column.

```
data['instrumentalness'].replace(0, np.nan, inplace=True)
data['instrumentalness'].fillna((data['instrumentalness'].median()), inplace=True)

data[data['instrumentalness'] == 0].shape
```

```
(0, 18)
```

Here I have removed the 0 values and replaced them with the median. This won't be an issue since if a song has real data about its instrumentalness it's likely going to be slightly higher than 0 instead of being spot on 0.0.

So by replacing 0's I am not losing valuable information.

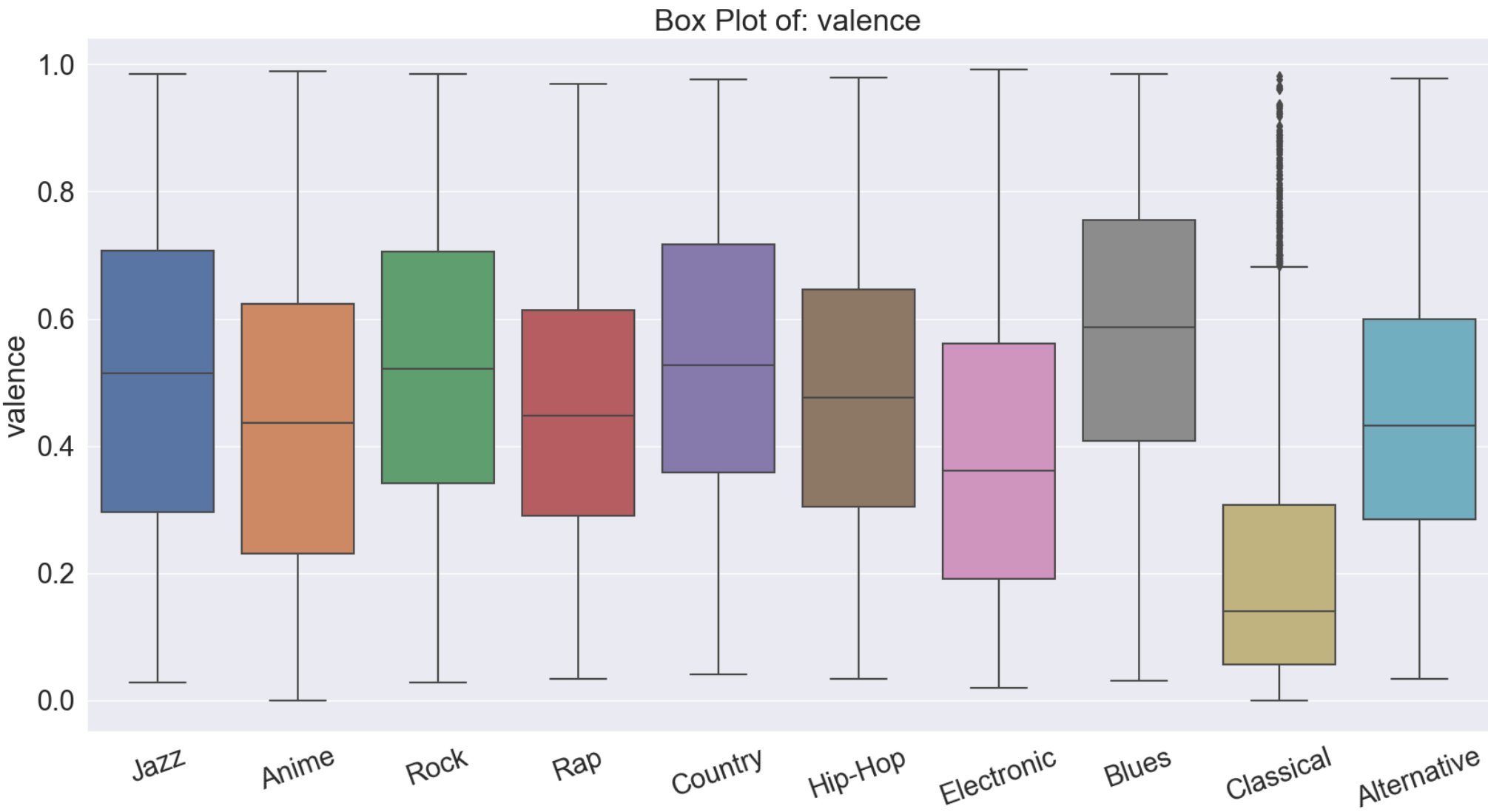
Valence

```
data['valence'].describe()
```

	valence
count	50000.000000
mean	0.456264
std	0.247119
min	0.000000
25%	0.257000
50%	0.448000
75%	0.648000
max	0.992000

We see our values are between 0 and 1.

+ Valence describes the musical positivity conveyed by the track.
+ Tracks with high valence sound more positive (happy, cheerful, euphoric).
+ Tracks with low valence sound more negative (sad, depressed, angry).



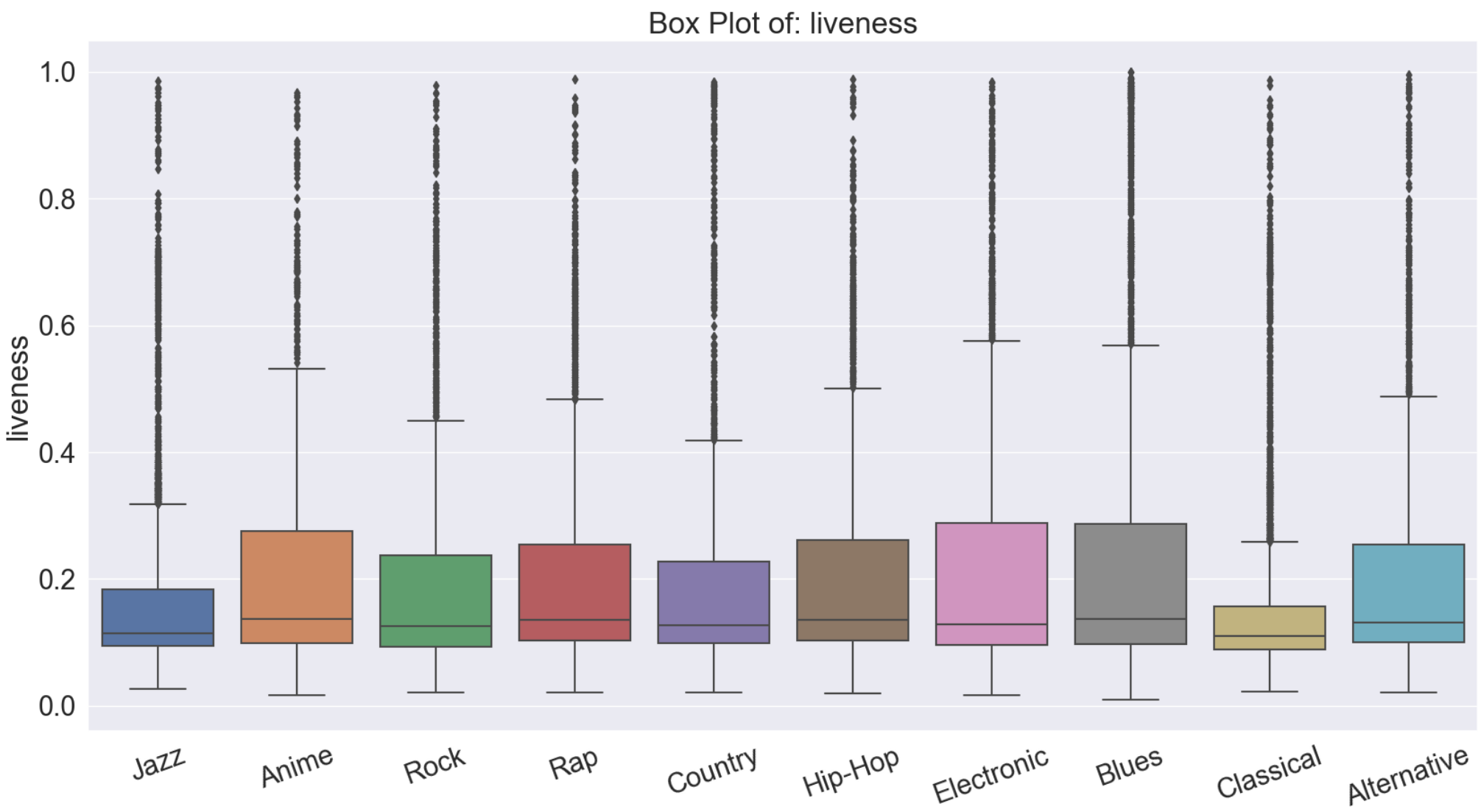
- To no surprise, classical is still the heavy outlier. We can finally see a little bit of difference between Rap and Hip-Hop.
- In general Hip-Hop seems to be the more “happier” type of genre.
- The other genres seem similar, nonetheless this is still a great feature for us to use in our predictions. It could help us to find the difference between Rap and Hip-Hop.

Liveness

```
data['liveness'].describe()
```

	liveness
count	50000.000000
mean	0.193896
std	0.161637
min	0.009670
25%	0.096900
50%	0.126000
75%	0.244000
max	1.000000

Our values are between 0 and 1.



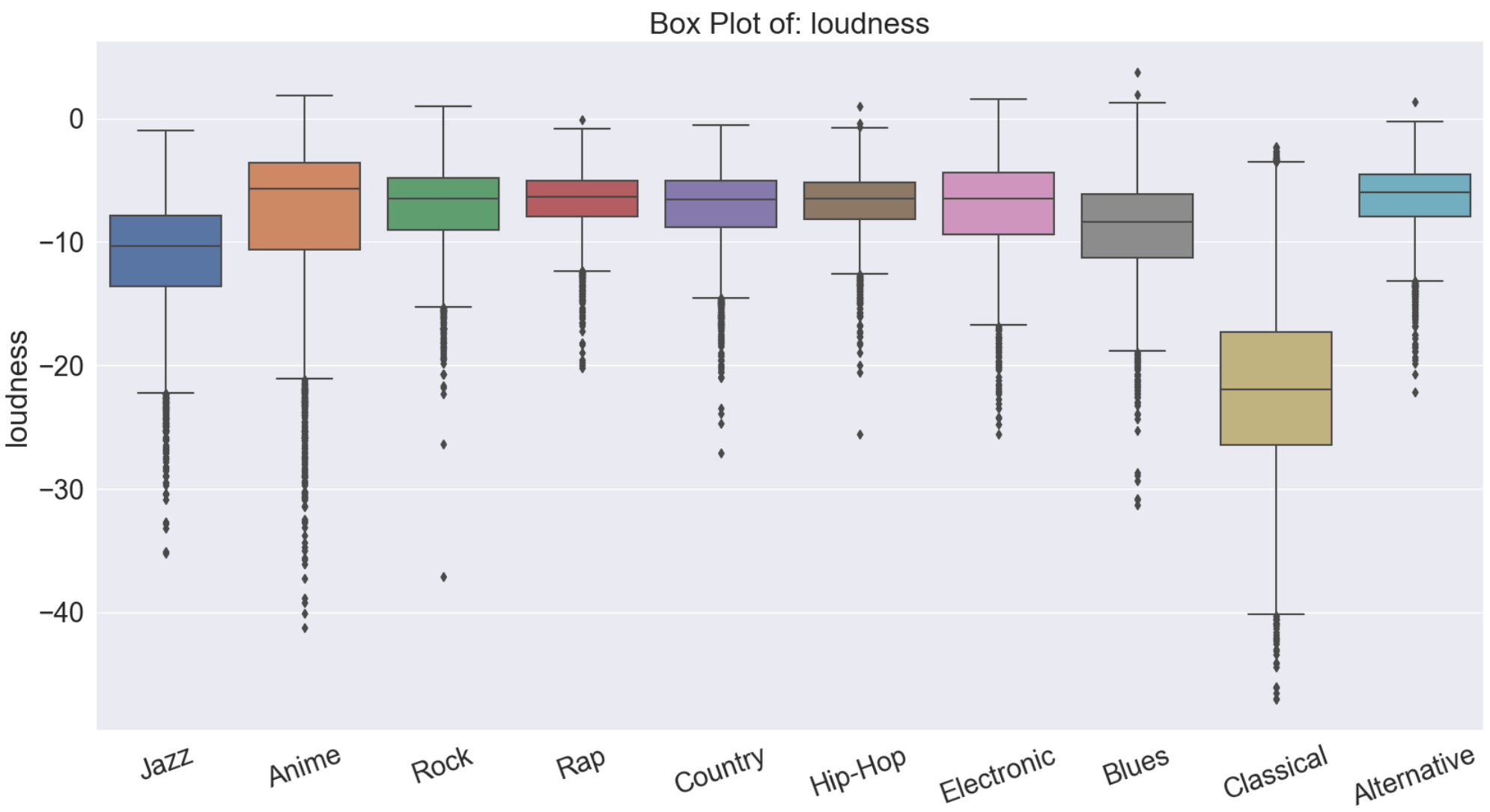
- All features here seem similar, yet there are still some differences between classical and jazz. I will keep this feature in my dataset. It can only help.

Loudness

```
data['loudness'].describe()
```

	loudness
count	50000.000000
mean	-9.133761
std	6.162990
min	-47.046000
25%	-10.860000
50%	-7.276500
75%	-5.173000
max	3.744000

We have negative and positive values here, showing that we should normalize our dataset before using it so that every feature is at the same number scale and not misinterpreted by our machine learning algorithm.

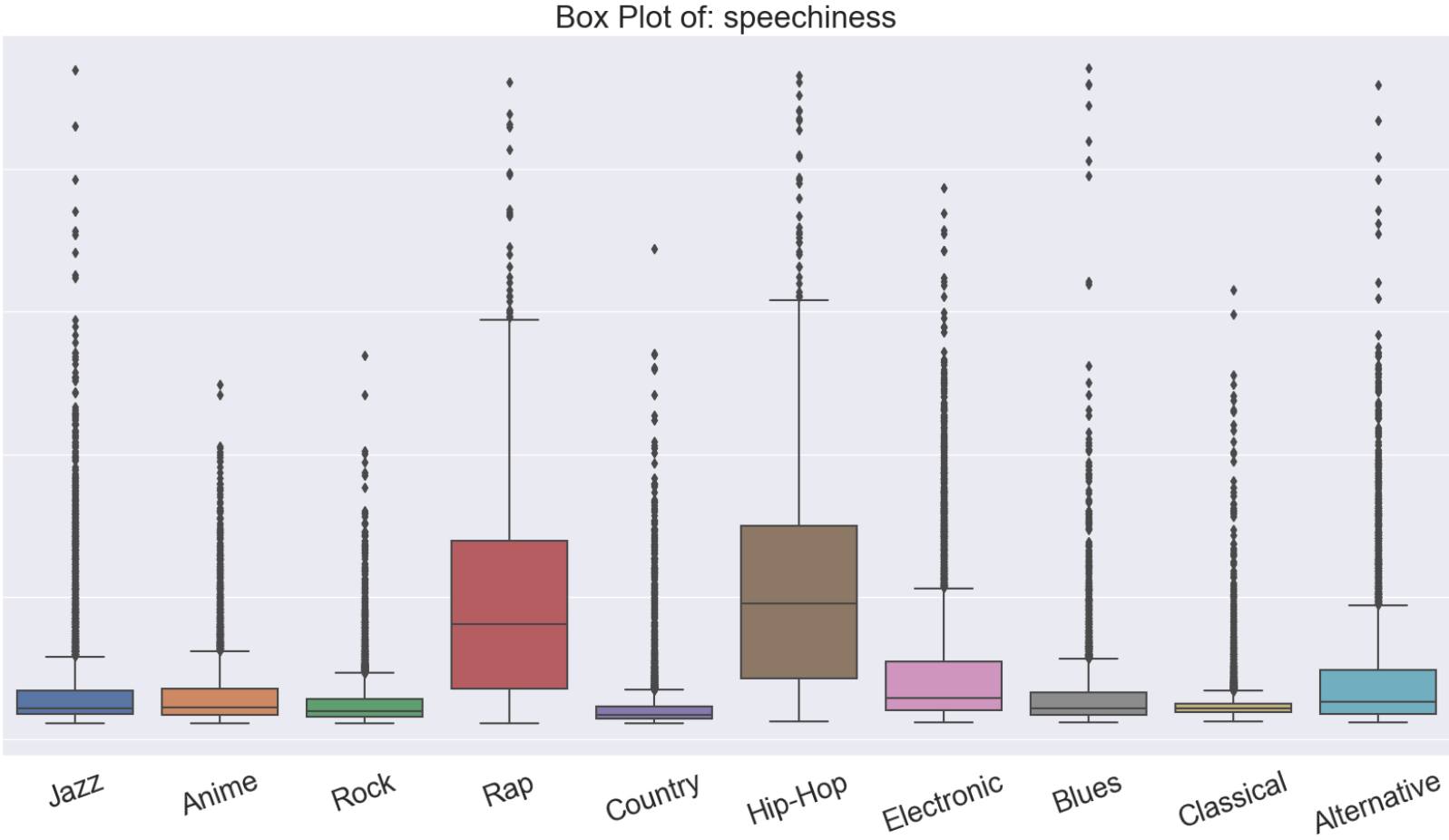

Box Plot of: loudness
loudness
0
−10
−20
−30
−40
Jazz
Anime
Rock
Rap
Country
Hip-Hop
Electronic
Blues
Classical
Alternative

Speechiness:

```
data['speechiness'].describe()
```

	speechiness
25%	0.036100
50%	0.048900
75%	0.098525
count	50000.000000
max	0.942000
mean	0.093586
min	0.022300
std	0.101373

Box Plot of: speechiness



- Here there is a huge difference between Rap, Hip-Hop and the other genres, this feature is essential for us to be able to predict these genres easily.
- So I will keep this feature in.

Tempo

```
data['tempo'].describe()
```

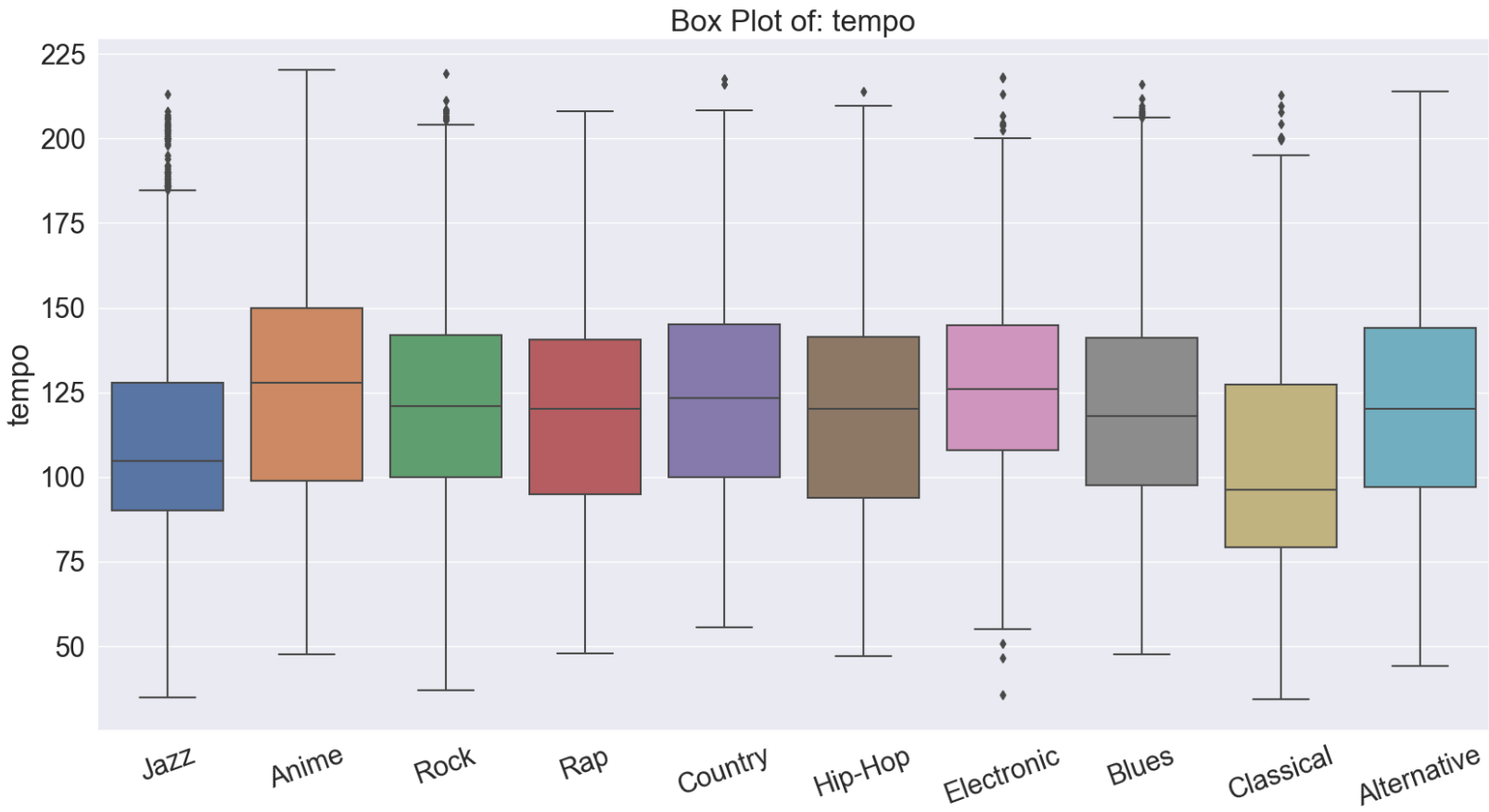
	tempo
count	50000
unique	29394
top	?
freq	4980

Here we see our top feature as “?”, this is a numerical column. This is an issue with the dataset, so I have marked all these “?” columns as NaN and dropped them from the dataset.

```
data[data['tempo'] == '?'].shape
```

```
(4980, 18)
```

- This meant that I have removed 4980 rows from my dataset, but I had to do this since this is an important feature that I cannot dismiss.
- Instead of filling median values, removing them was the better choice, 4980 rows aren't that significant on a 50005 row dataset.



Categorical Column Preprocessing:

- There are 3 total categorical columns left at the dataset.
- These columns are: Mode, Music Genre, Key.

To be able to use these categorical columns in my prediction models first I had to turn them into numerical values.

There were 2 ways that I could use in order to turn these categorical values into numerical.

First was to create a dictionary to map these values to a numerical value that I wanted, for example I could say for Mode column that: {Minor: 0, Major:1} or for the Key column I could say that: {A: 0, A#:1, B:2, C:3, C#:4, … , F#:10, G:11 G#:12} but doing it this way has one huge problem: **This way the algorithm will think that the key G is *more important* than the key A, since G has a higher numerical value.**

This is terrible for our machine learning algorithm. So, in order to prevent this I used the method of **One Hot Encoding (OHE)**.

Shortly **OHE** is the process of turning categorical values into numerical without having "*this data is more important*" factor. It does this by *converting each categorical value into a new column* and assigning binary values of 1 or 0 to these columns.

We should use OHE if our categorical column has no ranking in itself. In our dataset all our categorical values have no ranking amongst them, so I used OHE for every one of them.

Key

```
data['key'].describe()
```

	key
count	50000
unique	12
top	G
freq	5727

- This is a categorical value in our dataset, we have 12 distinct key values.

I have done some research about the Key of a music to figure out what it really does:

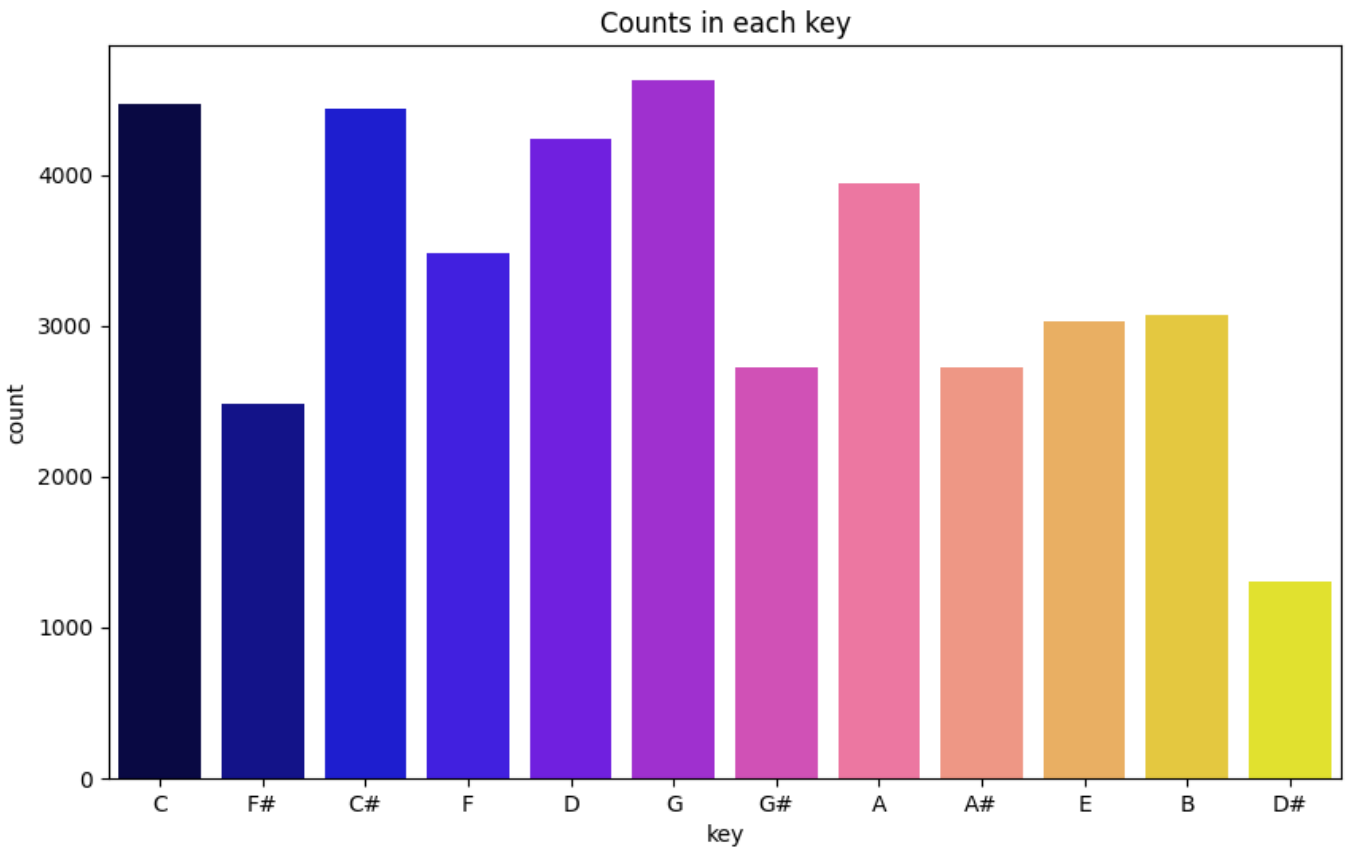
A key is the main collection of pitches, or notes, that contain the harmonic foundation of a piece of music. It determines the major or minor scale that a piece of music revolves around. A song in a major key is built on a major scale. A minor key song is built on a minor scale.

A song in the 'key of C major' focuses around the seven notes of the C major scale - C, D, E, F, G, A, and B. The song's melody, chords, and bassline are all generated from that collection of notes in the key.

Key column might not have much effect on predictions on our model, since the Key of the song is a personal taste. I will elaborate this later on my report.

+ I generated a count plot for how many entries are in the database according to the key.
+ Since I was going to generate more count plots in the future I wrote this into a function just like the box plot one.

```
def get_count_plot(self, feature, order=None):
    plt.figure(figsize=(10, 6))
    sns.countplot(x=feature, data=self.df, palette="gnuplot2", order=order)
    plt.title(f"Counts in each {feature}")
    plt.savefig(f'./plots/count/{feature}.png')
    plt.close()
```



We see that most songs are in C, C#, G keys while the least popular key to use is D#

Mode:

```
data['mode'].describe()
```

	mode
count	50000
unique	2
top	Major
freq	32099

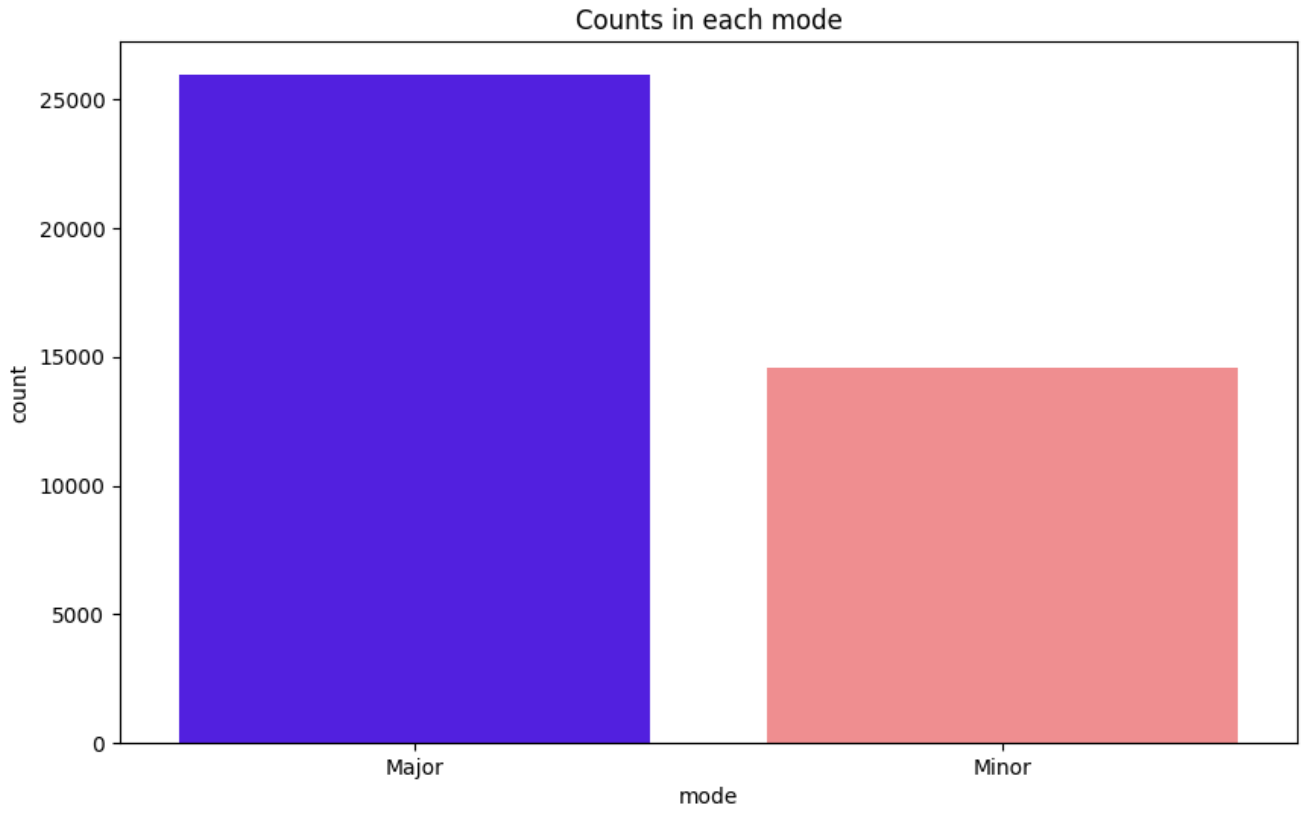
```
data['mode'].unique()
```

	0
0	Minor
1	Major

- The second categorical column in our dataset is the Mode.
- We have 2 unique modes.

The key and the mode are heavily related with each other since when you want to mention the key of a song, you almost always specify whether it is Major or Minor.

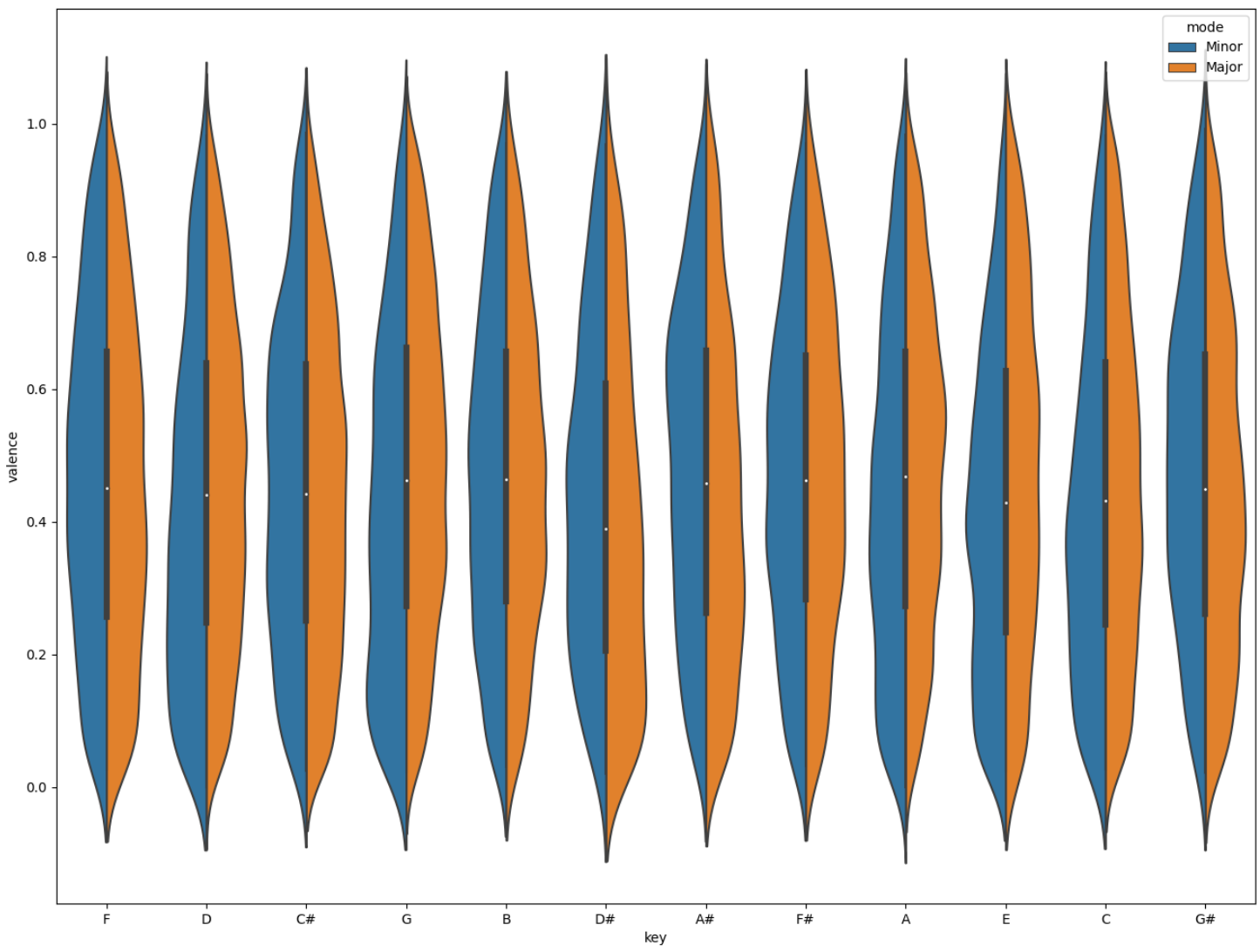
💡For this reason I will use the Key and Mode columns in combination.



We see most of our data is in the Major key.

I have generated this Violin plot to see if the key and the mode together mattered when determining the songs Valence (Happiness).

```
    def get_violinplot(self):
        plt.figure(figsize=(16, 12))  # Increase figure size to fit the heatmap.
        sns.set_context(font_scale=1.8)  # Increase font size.
        sns.violinplot(data=self.df,
                       x='key',
                       y='valence',
                       hue='mode',
                       split=True, )
        plt.savefig('./plots/violin.png')  # Save heatmap result to location as png.
        plt.close()
```



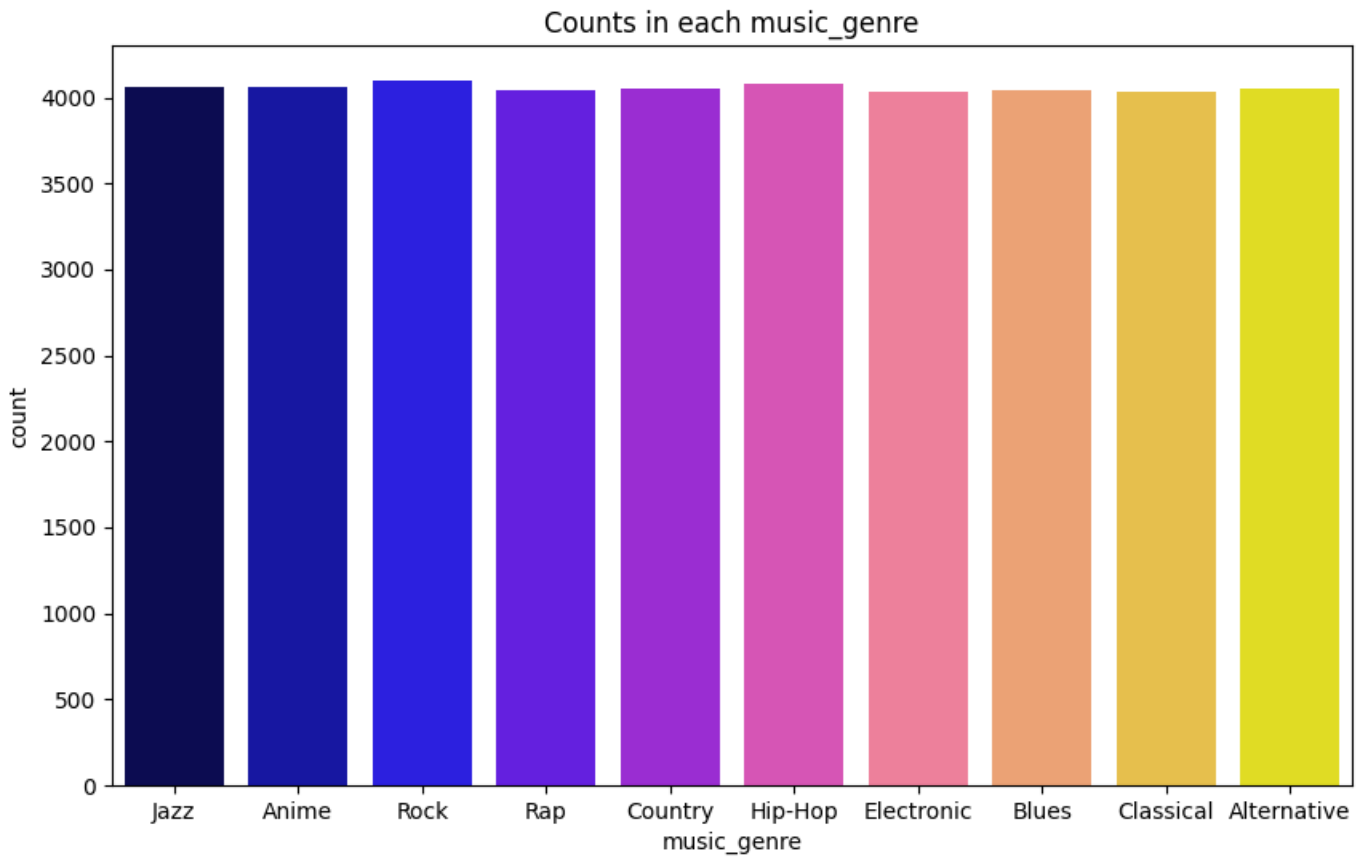
Since Valence shows the mood of the track, I wanted to see if the song being Minor or Major affected the mood of the song, and I also wanted to see the effect of the “Key” column on the mode of the song.

Here we can see that no matter what the key is the plots look similar, this shows us that the key column doesn’t have much effect on our prediction.

If we closely observe we can see that when our Valence values are low, the **blue part** of the Violin plot which demonstrates the Minor mode is **thicker**. Showing us that if the song is in the Minor mode, it will sound more depressing, sad, angry thus having a low Valence value.

Music Genre:

This is our final categorical column in the dataset and it is also the feature that we want to predict.



Here we see that we have equal amounts of entries on our dataset about genres.

There are 10 genres to be predicted:

- Jazz
- Anime
- Rock
- Rap
- Country
- Hip-Hop
- Electronic
- Blues
- Classical
- Alternative

The code for this pre-processing part:

- Reads csv file
- Marks “?” and “-1” values on duration columns as NaN (Not a Number)
- Removes NaN’s.
- Removes duplicate rows.
- Resets the indices after doing drop operations to have ordered data.
- **Shuffles the whole dataset so we have unorganized data for better training.**

```
  def load_dataset(self):
        # Load the dataset and mark ?'s as null.
        self.df = pd.read_csv(f'dataset/music_genre.csv', na_values="?")
        self.df['duration_ms'].replace(to_replace=-1, value=np.nan, inplace=True)
        self.df.dropna(inplace=True)  # Drop all null rows.
        self.df.drop_duplicates(inplace=True)  # Drop duplicate rows.
        self.df.reset_index(inplace=True)  # Reset indices since we deleted some rows.
        self.df = shuffle(self.df, random_state=1)  # Shuffle the dataset.
        return self.df

text_columns = ['instance_id', 'track_name', 'obtained_date', 'artist_name', 'index']
        self.df.drop(text_columns, axis=1, inplace=True)
```

```
categorical_columns = ['mode', 'music_genre', 'key']
self.df[categorical_columns] = self.df[categorical_columns].astype('category')

self.df = pd.get_dummies(self.df, columns=categorical_columns)  # OHE
```

PS: Code looks a little different on the source code, this is a more simplified version for the sake of the report.

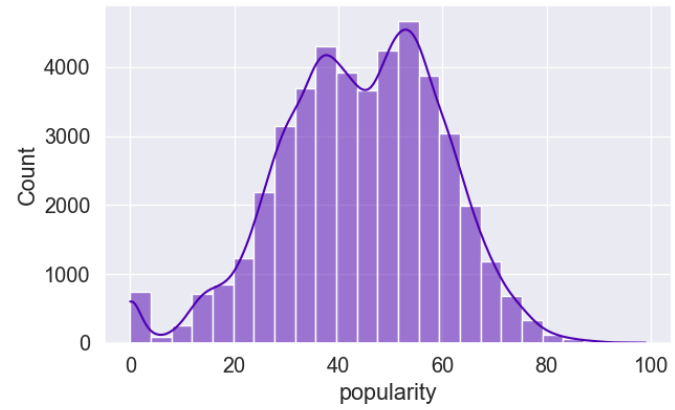
Plotting general information about the dataset:

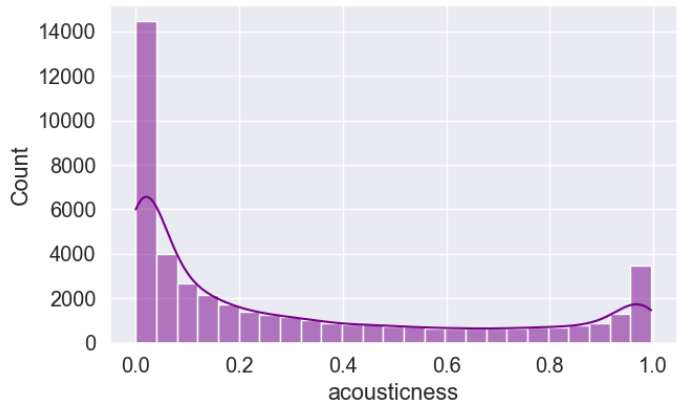
Histogram of every column:

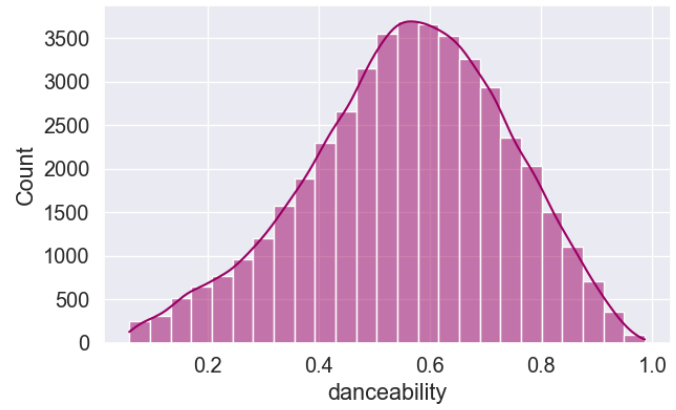
We can see the distribution of the data clearly here, we see that there are many songs that have an Acousticness of 0, these could be the genre of Anime, Electronic.

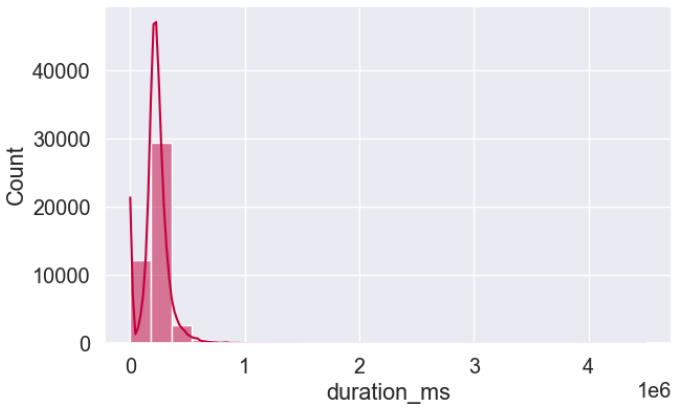
Also, we see that many songs have a short duration while only a few have a long duration time, these could be Classical genre songs since they are usually longer.

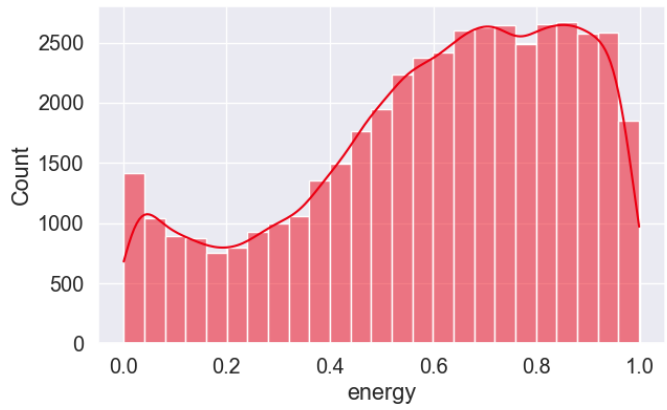
```
def get_histograms(self, features, rows, columns):
    fig = plt.figure(figsize=(20, 20))
    colors = sns.color_palette("brg", len(features))
    sns.set(style="darkgrid", font_scale=1.3)

    for i, feature in enumerate(features):
        if i == 0:
            pass # Skip index value
        else:
            ax = fig.add_subplot(rows, columns, i)
            sns.histplot(self.df[feature], bins=25, kde=True, ax=ax, color=colors[i])

    fig.tight_layout()
    plt.savefig('./plots/histograms.png')
    plt.close()
```

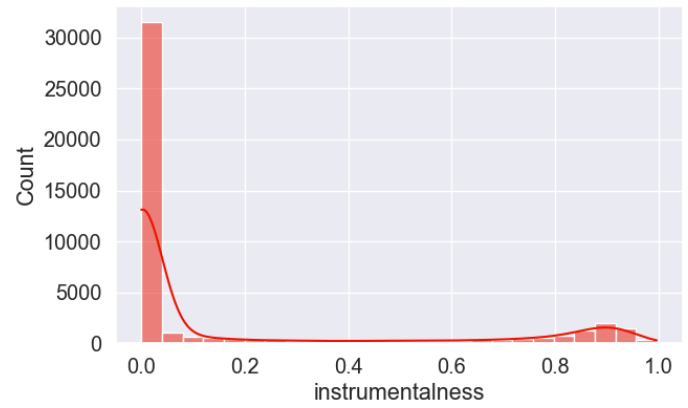

Count
popularity

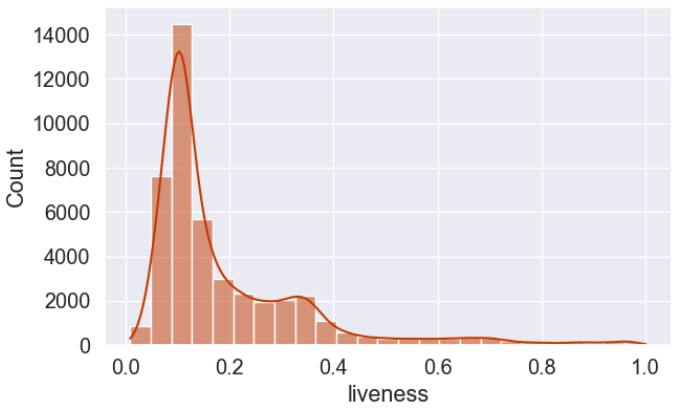

Count
acousticness

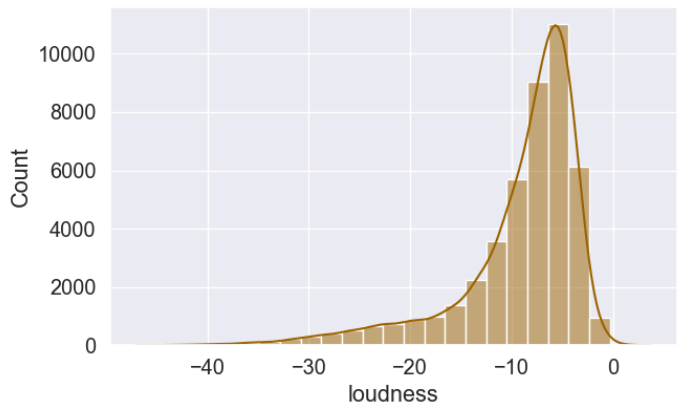

Count
danceability

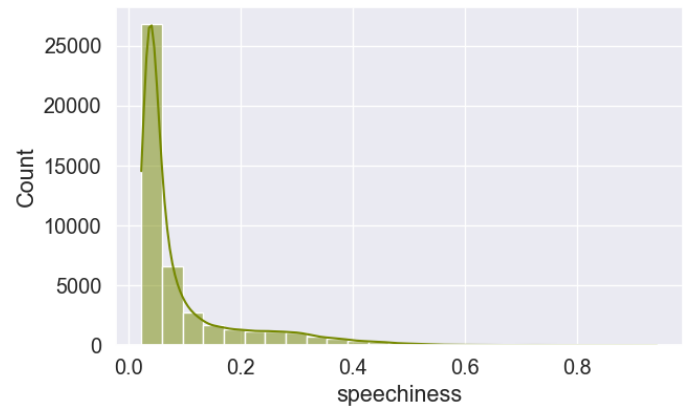

Count
duration_ms
1e6

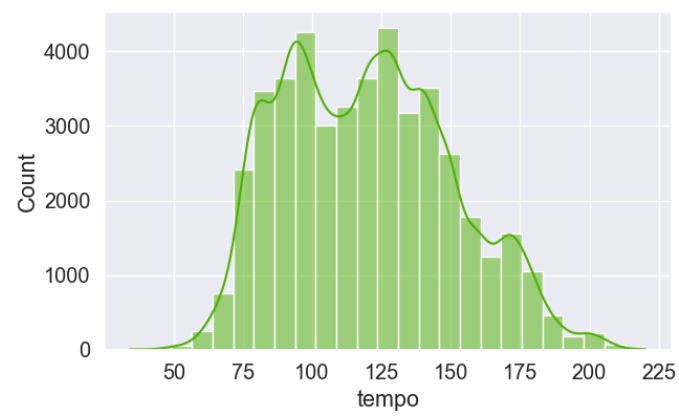

Count
energy

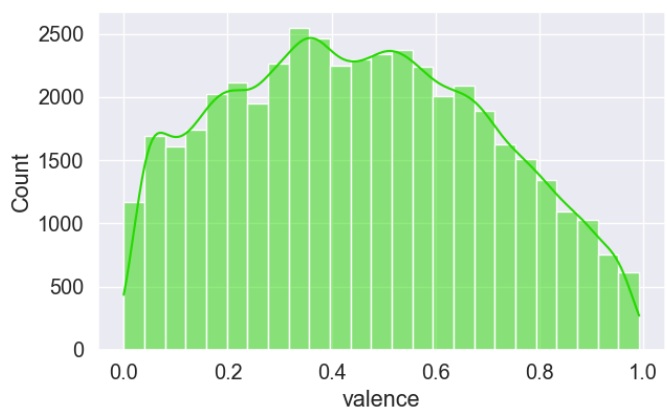

Count
instrumentalness


Count
liveness

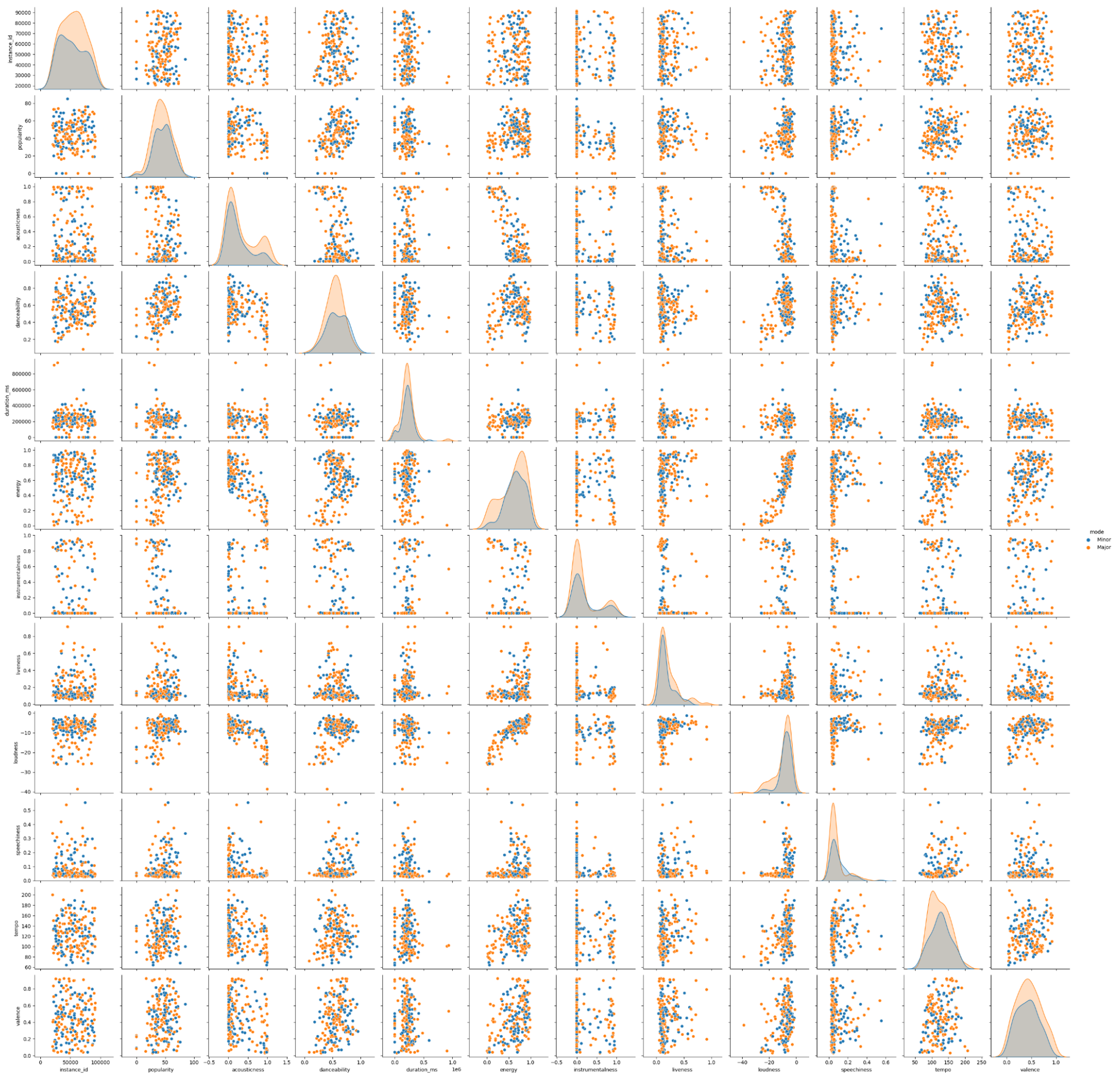

Count
loudness


Count
speechiness


Count
tempo

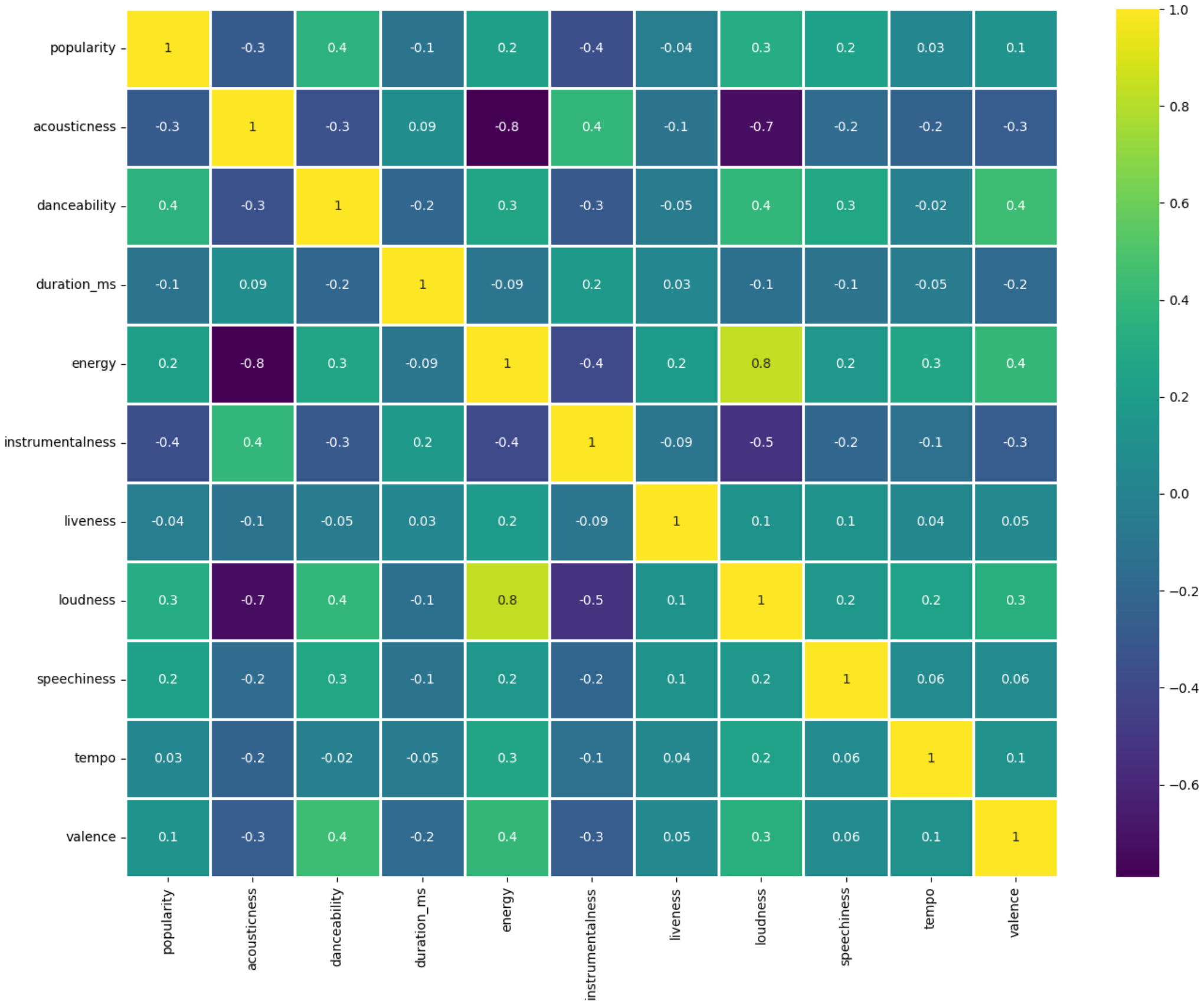

Count
valence

Pairplot of every column grouped by **Mode**: This plot is also too large in size to fit to the report, below is a preview. To be able to read and see the full quality run the code and look at the generated PNG result in the project folder.



To see the correlation between the data, I have created a Heatmap of the dataset. This way I would see which columns are related to which and I wanted to have a stronger understanding about the dataset.

```
def get_heatmap(self, filename, nums=True, linewidth=0, figsize=(16, 12)):
        """ Returns a heatmap of the dataset. """
        matplotlib.rc_file_defaults()
        plt.figure(figsize=(16, 12))  # Increase figure size to fit the heatmap.
        plt.xticks(rotation=90)
        sns.set_context(font_scale=1)  # Increase font size.
        sns.heatmap(self.df.corr(), annot=nums, fmt=".1g", cmap='viridis', linecolor='white',
                    linewidth=linewidth)  # Draw heatmap.
        plt.savefig(f'./plots/heatmap/{filename}.png')  # Save heatmap result to location as png.
        plt.close()
```



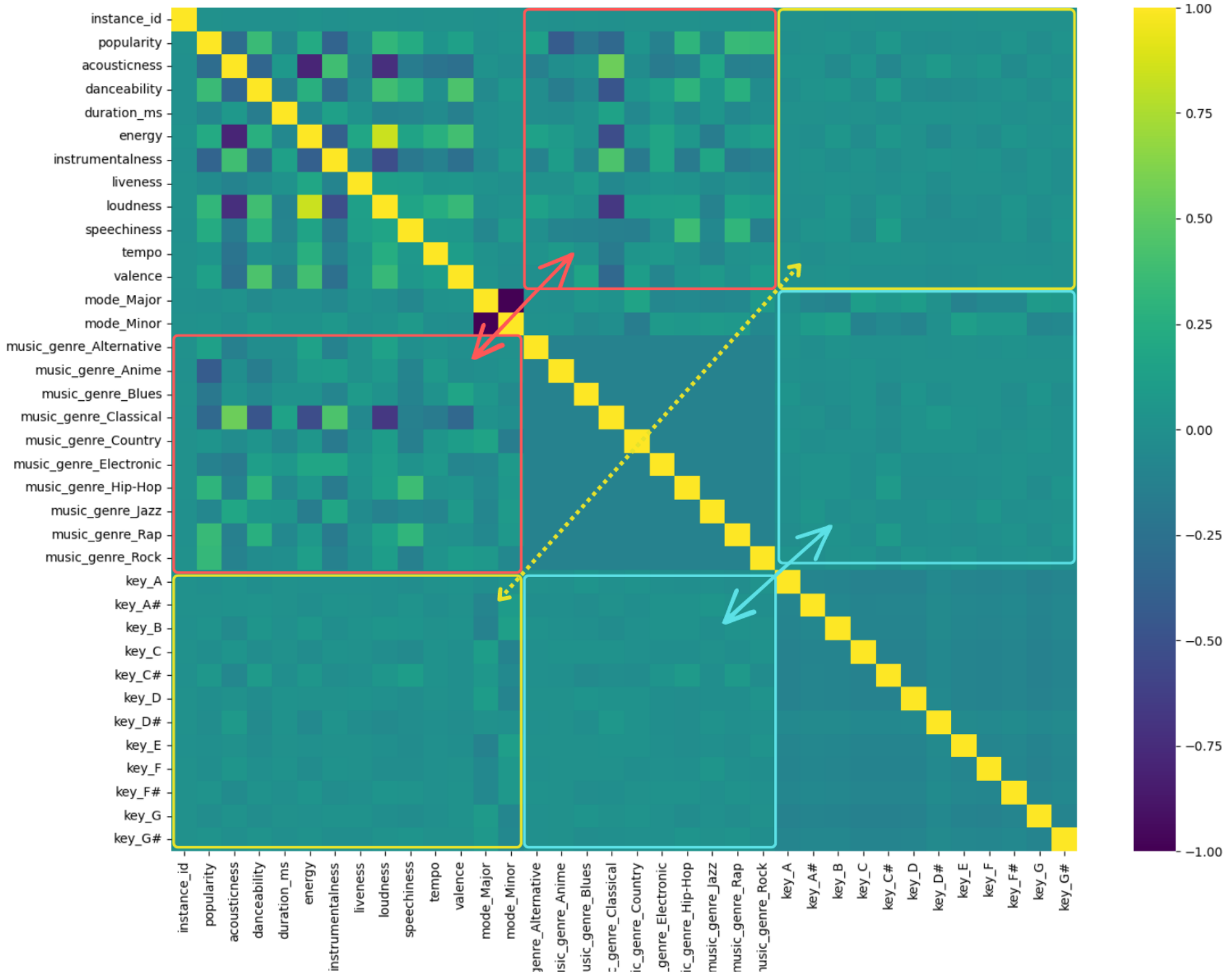
By analyzing the heatmap we see the correlations:

+ Energy vs Loudness = 0.8
+ Energy vs Acousticness: -0.8
+ Acousticness vs Loudness: -0.7
+ Loudness vs Instrumentalness = -0.5
+ Loudness vs Danceability = 0.4

These values show a strong relationship between these columns, so I wanted to add these to my dataset as new columns. This would explicitly tell my machine learning algorithm that these parameters were important.

★ I could use these new relations to improve my accuracy if I wasn't satisfied with the old accuracy of my models.

After performing One Hot Encoding to my categorical columns I have generated another heatmap:



I have analyzed this heatmap and made some coloured frames to explain my observations:

- ★ **Red Frame:** Shows us that our target columns (the values that we are trying to predicts which are the Music Genre) have some relation to our dataset, if we observe we can see some strong values shown in purple or light green colors. For example, if we look at music_genre_Classical column we can see that it has strong correlation with loudness and acousticness of the music etc. This shows that we can effectively use our dataset in order to predict our music genres.
- ★ **Yellow Frame:** Shows us that the key of the song doesn't really have any correlation with the dataset, since we can see that all the values are around 0.00.
- ★ **Cyan Frame:** Shows us that the key of the song also doesn't have a relation with the Music Genre. This is understandable since keys are arbitrary and they only alter the pitch of the song, this doesn't tell anything about our music genre. Hearing them as happy or sad is purely personal.

---

2) Final Touches on Data:

Summary of data pre-processing so far:

Features to be used:

- ♦ **Popularity** → Untouched.
- ♦ **Acousticness** →Untouched.
- ♦ **Danceability** → Untouched.
- ♦ **Duration_ms** → Some entries had missing values, they were dropped.
- ♦ **Energy** → Untouched.
- ♦ **Instrumentalness** → Had missing values, they were filled by median.
- ♦ **Valence** → Untouched.
- ♦ **Liveness** → Untouched.
- ♦ **Loudness** → Untouched.

- **Speechiness** → Untouched.
- **Tempo** → Had missing values, they were dropped.
- **Key** → Performed **One Hot Encoding**
- **Mode** → Performed **One Hot Encoding**
- **Music Genre** **(Prediction Feature)** → Performed **One Hot Encoding**

Features that were removed:

- **Instance_id** → Just an index.
- **Obtained Date** → When the data was obtained is not important.
- **Artist Name** → Decided not to use it to not overfit.
- **Track Name** → Every track has an unique name, no relation to the dataset.

Normalizing the features:

One of the essential parts of preparing my data was to normalize the features, I wrote a method on my MusicData class for this purpose:

```
   def normalize_features(self, df):
        scaler = RobustScaler()
        scaler.fit(df)
        scaled = scaler.transform(df)
        scaled_df = pd.DataFrame(scaled, columns=df.columns)
        return scaled_df
```

Here I normalize every column on the Dataframe by using the RobustScaler class from sklearn library.

- Robust Scaler algorithms scale features that are robust to outliers.
- Since we have an extreme amount of outliers in our dataset I picked this normalization method.
- The method it follows is almost similar to the Min-Max Scaler but it uses the interquartile range (rather than the min-max used in Min-Max Scaler). This scaling algorithm removes the data's median and scales based on the quantile range.

RobustScaler follows this formula:

$$\frac{x_i - Q_1(x)}{Q_3(x) - Q_1(x)}$$

- This way every data is on the same scale and can be understood way better for the machine algorithm.
- I could have also used Standard Scaler but since I have One-Hot Encoded columns, applying Standardizing to the One-Hot encoded features would mean assigning a distribution to categorical features.
- We don't want to do that; the OHE values should be binary.

---

3) Splitting the Data

```
self.df = self.normalize_features(self.df)

def split_feature_as_x_y(self, prediction_feature):
        prediction_feature = 'music_genre'
        filter_col = [col for col in self.df if col.startswith(prediction_feature)]
        X = self.df.drop(filter_col, axis=1)
        y = self.df[filter_col]
        return X, y

   def get_x_y_split(self):
        return self.X_train, self.X_test, self.y_train, self.y_test

self.X, self.y = self.split_feature_as_x_y('music_genre')
self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(self.X, self.y,test_size=0.20, random_state=1)
```

- Here again I am using the scikit-learn library and importing the train_test_split() method, which splits my X and Y data as train and test sets.
- I have picked a ratio of %20 test and 80% training. These seemed to be the correct values for a dataset with 50k entries.

- random_state = 1, makes it so that we get the same random values every time, this way I can see if the changes I make actually improve my models so it's not based on my luck at randomizing.

---

4) Models

I have provided the minified code for all my models here. Minified here means that I have removed any Console related stuff, such as printing how long it took to run or some comments since those are already provided in the source code I will attach with this project report. I kept my codes simple here on the report to explain the essential lines of the algorithm.

Support Vector Machine:

Support Vector Machines (SVMs) are:

- Effective in high dimensional spaces.
- Still effective in cases where the number of dimensions is greater than the number of samples.
- Is memory efficient.
- Different Kernel functions can be specified for the decision function and the problem at hand.

The minified code:

```
def support_vector_machine(kernel):
    data = MusicData()
    X_train, X_test, _, _ = data.get_x_y_split()
    y_train, y_test = data.get_y_categorical()
    y_train = y_train.values.ravel()  # Turns Y into an 1 Dimensional array.

    clf = svm.SVC(kernel=kernel)
    clf.fit(X_train, y_train)  # Train the model.

    y_pred = clf.predict(X_test)  # Make predictions with the trained SVM model.

    accuracy = accuracy_score(y_test, y_pred) * 100

    print(classification_report(y_test, y_pred))
    prediction_heatmap(y_test, y_pred, data.get_y_names(), "support_vector_machine")
```

I call the methods of my MusicData() class to easily get my already prepared, split and ready to use data.

`data.get_y_categorical()` Returns Music Genres as categories without any One Hot Encoding or numerical values, since Support Vector Machines accept categorical Y values.

`clf = svm.SVC(kernel=kernel)`The "svm" is the function I imported from scikit-learn. This is the Support Vector Machine model itself.

I have also passed here a parameter called "kernel" this decides which kernel to use on our Support Vector Machine, such as: Linear, Polynomial, RBF

- ➢ Linear is kernel good for linear data as its name suggest, it does categorization by splitting the input space by a line.
- ➢ Polynomial kernel is good for linear and curved input space. It can do classification by splitting the input space in curves.
- ➢ Radial Basis Function Kernel (RBF) can map the input space in **infinite dimensions**. For this reason I have decided to go with RBF.

`clf.fit(X_train, y_train)` → Trains the model on our train data.

`y_pred = clf.predict(X_test)`→ Makes predictions with the trained model on our test data.

`accuracy_score(y_test, y_pred) * 100` → accuracy_score() is a function I imported from the scikit-learn library, it compares the 2 arrays given to it, here they are the y_test and the y_pred arrays. And it returns a percentage on how accurate the predictions were based on those 2 arrays. Then I multiply it by 100 to get the actual percentage out of 100.

`classification_report(y_test, y_pred)`→ Gives detailed information about how well the model predicted for each Music Genre.

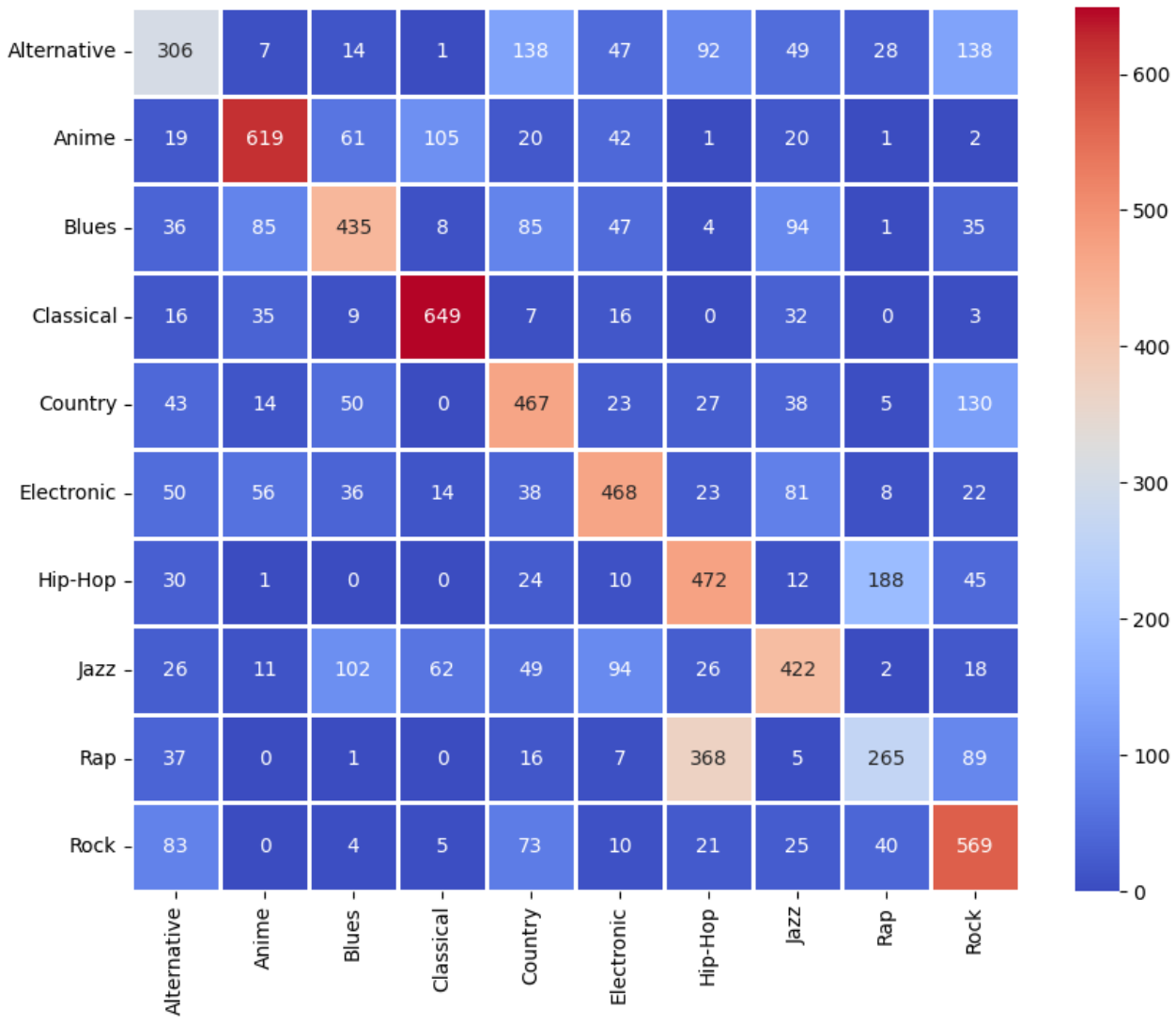
`prediction_heatmap(y_test, y_pred, data.get_y_names()`→ This is a utility function that I have written in my utility class. It plots the prediction accuracy for each genre in a heatmap.

```
Training the SVM model with 'rbf' kernel...
🕐 Timer | Trained the SVM Model : Executed in 71.0831s
----------------------------------------------------------------
The Accuracy for the Test Set is 57.59368836291914
              precision    recall  f1-score   support

 Alternative       0.47      0.37      0.42       820
       Anime       0.75      0.70      0.72       890
       Blues       0.61      0.52      0.56       830
   Classical       0.77      0.85      0.81       767
     Country       0.51      0.59      0.54       797
  Electronic       0.61      0.59      0.60       796
     Hip-Hop       0.46      0.60      0.52       782
        Jazz       0.54      0.52      0.53       812
         Rap       0.49      0.34      0.40       788
        Rock       0.54      0.69      0.60       830

    accuracy                           0.58      8112
   macro avg       0.58      0.58      0.57      8112
weighted avg       0.58      0.58      0.57      8112

🕐 Timer | Predicted results : Executed in 41.1257s
----------------------------------------------------------------
```

This is the console output when I run this class.

- ★ Here we can see that we have an accuracy of 57.60 % for the test set!
- ★ It took 71 seconds to train our model and 41 seconds for it to make the predictions on the test set.

Confusion Matrix visualized as a heatmap.



On the diagonal here we see the correctly made classifications. The more “red” the box gets the better percentage for that class.

From here we can also see that we have predicted Rap instead of Hip-Hop many times.

We have very strong predictions on Classical, Anime and Rock genres.

Logistic Regression:

Difference from Linear Regression is that we use Logistic Regression for classification problems while we use Linear regression for regression problems. Here since our problem at hand is a multi-classification problem, we can use Multinomial Logistic Regression.

There are 3 types of Logistic Regression:

- Binary Logistic Regression → The most common type, used for binary classifications (2 Classes. [Yes / No])
- Multinomial Logistic Regression → The one I used here. Able to make multi-classifications.
- Ordinal Logistic Regression → Makes classification predictions on ordered categorical data. Since our data can't be ordered and are independent of each other, I didn't use this.

The minified code:

```
def logistic_regression(iterations):
    data = MusicData()
    X_train, X_test, _, _ = data.get_x_y_split()
    y_train, y_test = data.get_y_categorical()
    y_train = y_train.values.ravel()


    model = LogisticRegression(max_iter=iterations)
    model.fit(X_train, y_train)

    training_time = print_time(t1, t2, "Trained the Logistic Regression Model")

    y_pred = model.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred) * 100
    print(f"The Accuracy for the Test Set is {accuracy}")
    print(classification_report(y_test, y_pred))
    prediction_heatmap(y_test, y_pred, data.get_y_names(), "logistic_regression")
```

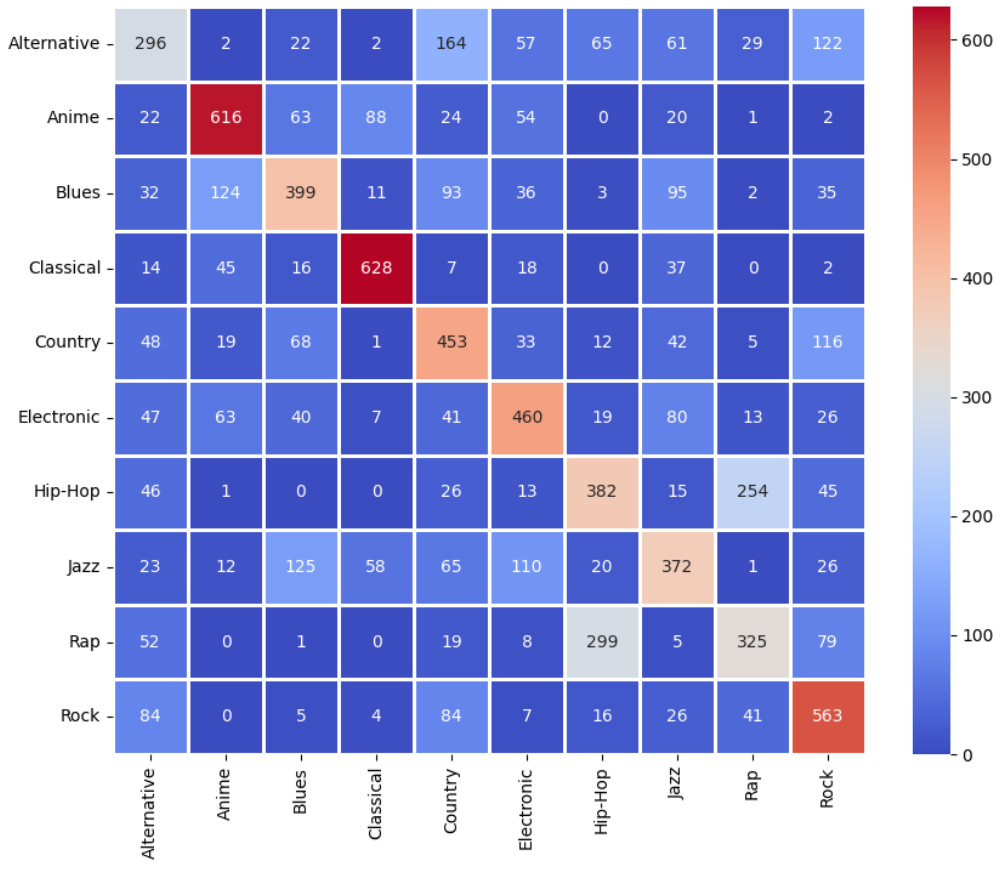
```
🕐 Timer | Trained the Logistic Regression Model : Executed in 16.4150s
---------------------------------------------------------------------
The Accuracy for the Test Set is 55.399408284023664
              precision    recall  f1-score   support

 Alternative       0.45      0.36      0.40       820
       Anime       0.70      0.69      0.70       890
       Blues       0.54      0.48      0.51       830
   Classical       0.79      0.82      0.80       767
     Country       0.46      0.57      0.51       797
  Electronic       0.58      0.58      0.58       796
     Hip-Hop       0.47      0.49      0.48       782
        Jazz       0.49      0.46      0.48       812
         Rap       0.48      0.41      0.45       788
        Rock       0.55      0.68      0.61       830

    accuracy                           0.55      8112
   macro avg       0.55      0.55      0.55      8112
weighted avg       0.55      0.55      0.55      8112
```

★ Here we can see that we have an accuracy of 55.40 % for the test set!
★ It took 16 seconds to train our model.

Confusion Matrix visualized as a heatmap.

On our confusion matrix below, we see similar results to the Support Vector Machine. Again the miss-classification between Rap and Hip-Hop is very large and is actually the main issue.

Other than that we see the Logistic Regression had a bit more trouble trying to predict the Jazz values, often miss-classifying it as other genres.



K-Nearest Neighbors:

Some information about this model:

- The KNN algorithm assumes that the new data and existing data are similar and places the new data in the category that is most similar to the existing categories.
- The KNN algorithm stores all available data and classifies a new data point based on its similarity to the existing data. This means that new data can be quickly classified into a suitable category. **This is where our K neighbor number comes into play.** It tells the algorithm for how many neighbors to control that is next to the data.
- The KNN algorithm can be used for both regression and classification, but it is most commonly used for classification problems.
- During the training phase, the KNN algorithm simply stores the dataset, and when it receives new data, it classifies it into a category that is very similar to the new data.

The minified code:

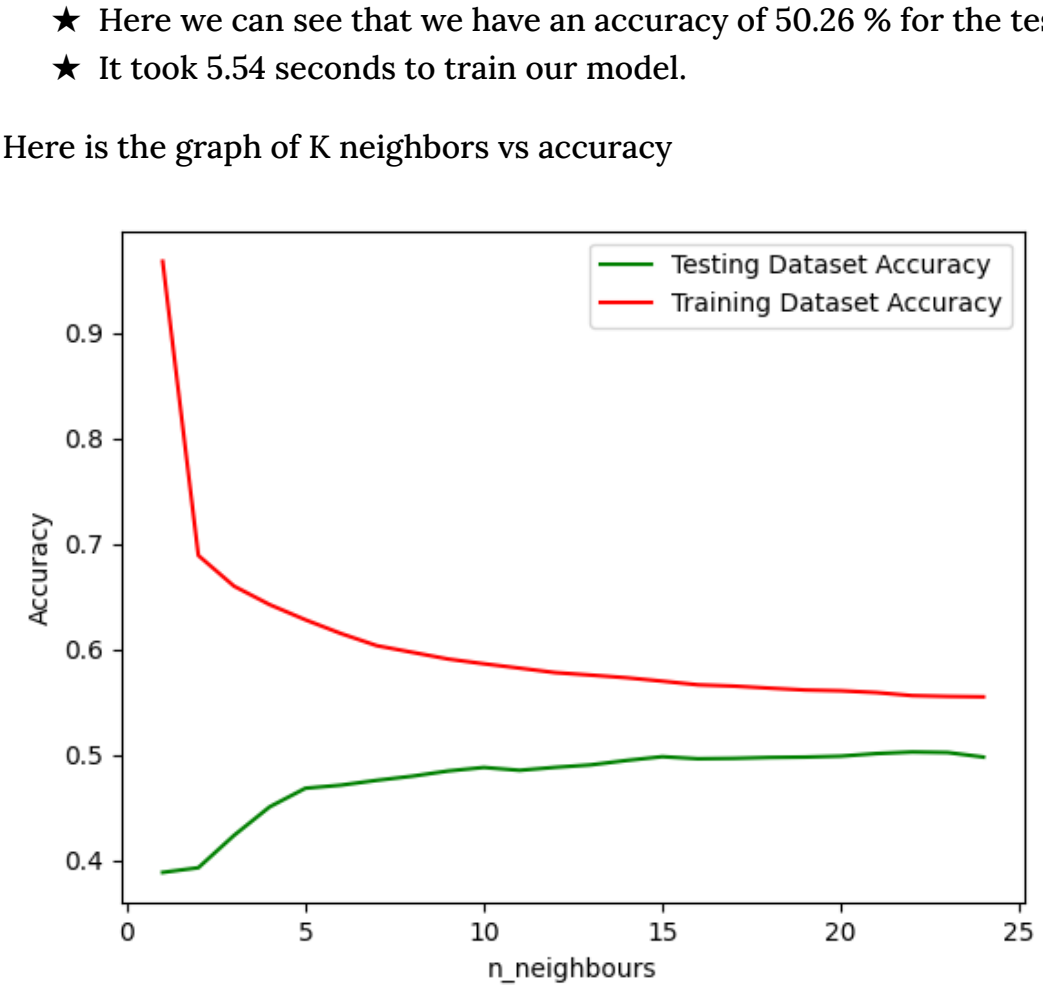
```
def knn(k):
    data = MusicData()
    X_train, X_test, _, _ = data.get_x_y_split()
    y_train, y_test = data.get_y_categorical()
    y_train = y_train.values.ravel()  # Turns Y into an 1 Dimensional array.

    neighbors = np.arange(1, k)  # Iterate over k neighbours.
    train_accuracy = np.empty(len(neighbors))
    test_accuracy = np.empty(len(neighbors))

    # Loop over K values
    for i, k in enumerate(neighbors):
        print(f"Fitting KNN for n={k}...")

        # Training the KNN.
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X_train, y_train)

        # Compute training and test data accuracy
        train_accuracy[i] = knn.score(X_train, y_train)
        test_accuracy[i] = knn.score(X_test, y_test)

        # Print results
        print(f"Train Accuracy of KNN for k={k} is: {train_accuracy[i]}")
        print(f" Test Accuracy of KNN for k={k} is: {test_accuracy[i]}")

    best_index = test_accuracy.argmax()
    print(f"Best accuracy was for k={best_index + 1}, with the test accuracy of: {test_accuracy[best_index] * 100}")
```

```python
# Generating the plot
    print(f"Generating the plot for the accuracies of the KNN...")
    plt.plot(neighbors, test_accuracy, label='Testing Dataset Accuracy', color='green')
    plt.plot(neighbors, train_accuracy, label='Training Dataset Accuracy', color='red')

    plt.legend()
    plt.xlabel('n_neighbours')
    plt.ylabel('Accuracy')
    plt.savefig('./plots/training/knn.png')
    plt.close()
    print(f"Plot generated successfully at location: ./plots/training/knn.png")
```

```
Fitting KNN for n=1...
Train Accuracy of KNN for k=1 is: 0.9683185404339251
 Test Accuracy of KNN for k=1 is: 0.38831360946745563
🕐 Timer | Fitted KNN and calculated scores : Executed in 5.2130s
---------------------------------------------------------------------
Fitting KNN for n=2...
Train Accuracy of KNN for k=2 is: 0.6891333826429981
 Test Accuracy of KNN for k=2 is: 0.3927514792899408
🕐 Timer | Fitted KNN and calculated scores : Executed in 4.7440s
---------------------------------------------------------------------
Fitting KNN for n=3...
Train Accuracy of KNN for k=3 is: 0.6600098619329389
 Test Accuracy of KNN for k=3 is: 0.4234467455621302
🕐 Timer | Fitted KNN and calculated scores : Executed in 5.7790s
---------------------------------------------------------------------
.
.
.
Fitting KNN for n=24...
Train Accuracy of KNN for k=24 is: 0.5549494575936884
 Test Accuracy of KNN for k=24 is: 0.4977810650887574
🕐 Timer | Fitted KNN and calculated scores : Executed in 6.3980s
---------------------------------------------------------------------
Generating the plot for the accuracies of the KNN...
Plot generated successfully at location: ./plots/training/knn.png
🕐 Timer | Trained all KNN : Executed in 129.8757s
---------------------------------------------------------------------
Best accuracy was for k=22, with the test accuracy of: 50.25887573964497
```

★ Here we can see that we have an accuracy of 50.26 % for the test set!
★ It took 5.54 seconds to train our model.

Here is the graph of K neighbors vs accuracy



Decision Tree:

The minified code:

```
def dtree():
    data = MusicData(dtree=True)
    X_train, X_test, y_train, y_test = data.get_x_y_split()

    best_depth = 3
    old_accuracy = 0
    best_tree = None

    for depth in range(1, 13):
        dtree = DecisionTreeClassifier(criterion="entropy", max_depth=depth)
        dtree = dtree.fit(X_train, y_train)

        y_pred = dtree.predict(X_test)
        accuracy = metrics.accuracy_score(y_test, y_pred) * 100
        print(f"DEPTH = {depth} | Accuracy: {accuracy}%")
        if accuracy > old_accuracy:
            print(f"Found better accuracy at this depth! Old Acc: {old_accuracy} new Acc: {accuracy}")
            old_accuracy = accuracy
            best_depth = depth
            best_tree = dtree

    print(f"Best accuracy was found at depth {best_depth}. Generating graph...")
    visualize_tree(best_tree, f"dtree_best")
```

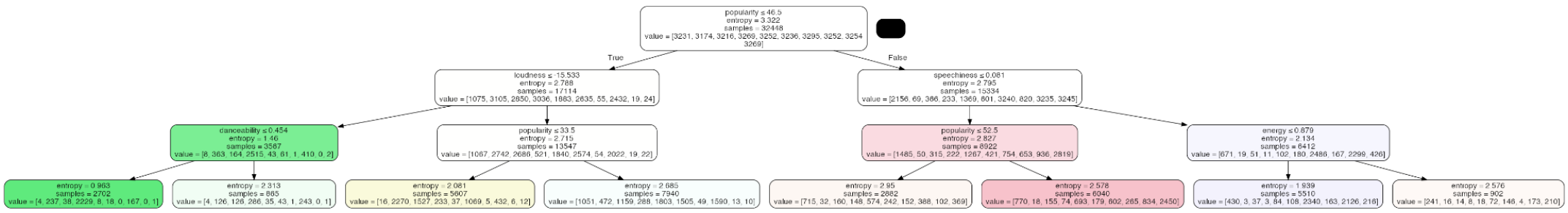
```
   def visualize_tree(decision_tree, filename):
        """ Draws the tree graph"""
        dot_data = StringIO()
        export_graphviz(decision_tree, out_file=dot_data,
                        filled=True, rounded=True,
                        special_characters=True, feature_names=X_train.columns)
        graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
        graph.write_png(f'./plots/training/{filename}.png')
        Image(graph.create_png())
```

```
🕐 Timer | Trained Decision Tree : Executed in 0.3873s
------------------------------------------------------------------
DEPTH = 1 | Accuracy: 20.944280078895464%
Found better accuracy at this depth! Old Acc: 0 new Acc: 20.944280078895464
🕐 Timer | Trained Decision Tree : Executed in 0.1741s
------------------------------------------------------------------
DEPTH = 2 | Accuracy: 32.519723865877715%
Found better accuracy at this depth! Old Acc: 20.944280078895464 new Acc: 32.519723865877715
🕐 Timer | Trained Decision Tree : Executed in 0.2348s
------------------------------------------------------------------
DEPTH = 3 | Accuracy: 37.9930966469428%
Found better accuracy at this depth! Old Acc: 32.519723865877715 new Acc: 37.9930966469428
🕐 Timer | Trained Decision Tree : Executed in 0.3307s
------------------------------------------------------------------
DEPTH = 4 | Accuracy: 43.68836291913215%
Found better accuracy at this depth! Old Acc: 37.9930966469428 new Acc: 43.68836291913215
🕐 Timer | Trained Decision Tree : Executed in 0.4262s
------------------------------------------------------------------
DEPTH = 5 | Accuracy: 46.21548323471401%
Found better accuracy at this depth! Old Acc: 43.68836291913215 new Acc: 46.21548323471401
🕐 Timer | Trained Decision Tree : Executed in 0.5280s
------------------------------------------------------------------
DEPTH = 6 | Accuracy: 47.53451676528599%
Found better accuracy at this depth! Old Acc: 46.21548323471401 new Acc: 47.53451676528599
🕐 Timer | Trained Decision Tree : Executed in 0.6122s
------------------------------------------------------------------
DEPTH = 7 | Accuracy: 50.09861932938856%
```

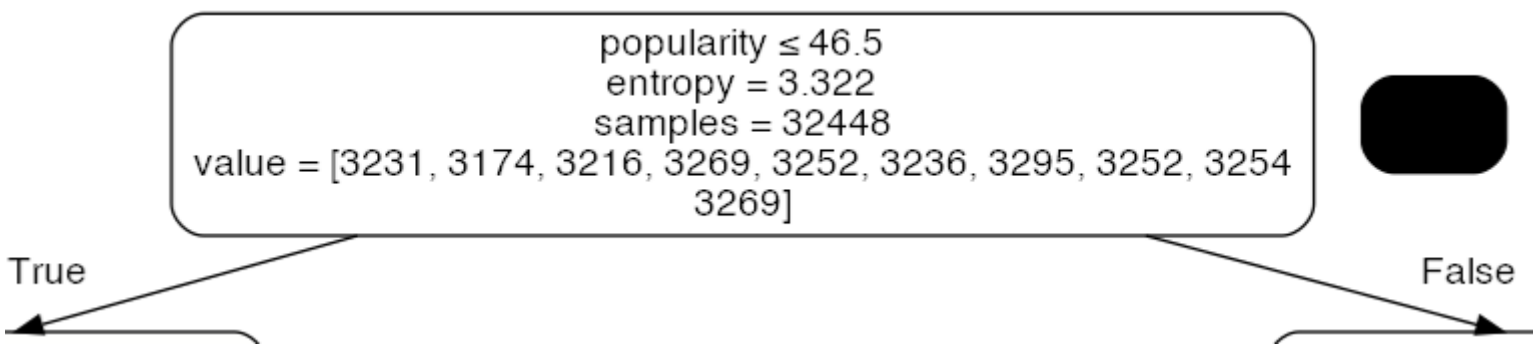
```
Found better accuracy at this depth! Old Acc: 47.53451676528599 new Acc: 50.09861932938856
🕐 Timer | Trained Decision Tree : Executed in 0.7253s
----------------------------------------------------------------
DEPTH = 8 | Accuracy: 51.947731755424066%
Found better accuracy at this depth! Old Acc: 50.09861932938856 new Acc: 51.947731755424066
🕐 Timer | Trained Decision Tree : Executed in 0.8260s
----------------------------------------------------------------
DEPTH = 9 | Accuracy: 52.40384615384615%
Found better accuracy at this depth! Old Acc: 51.947731755424066 new Acc: 52.40384615384615
🕐 Timer | Trained Decision Tree : Executed in 1.0107s
----------------------------------------------------------------
DEPTH = 10 | Accuracy: 52.1819526627219%
🕐 Timer | Trained Decision Tree : Executed in 1.0445s
----------------------------------------------------------------
DEPTH = 11 | Accuracy: 51.79980276134122%
🕐 Timer | Trained Decision Tree : Executed in 1.1206s
----------------------------------------------------------------
DEPTH = 12 | Accuracy: 51.04783037475345%
Best accuracy was found at depth 9. Generating graph...
```

★ Here we can see that we have an accuracy of **52.40 %** for the test set!
★ It took 1.01 seconds to train our model.

Here is the Decision Tree visualized at Depth level 3, this is still really hard to read so opening the actual png from the project folder is a good idea.



Here is a zoomed in version of the root node:



Samples: Here we see how many samples reached this node. Since this is our root node we have all the training samples which are 32k after splitting the dataset as training and testing.

Entropy: Entropy is a measure of impurity or uncertainty in a set of observations used in information theory. It determines how data is split by a decision tree. We are trying to reduce this as we go deeper and deeper into the tree.

Value: These are how many samples were predicted as what genres, we have 10 genres so there are 10 values. For example 3231 shows how many would be classified as “Anime” at this node.

And the Decision Tree visualized at it’s best accuracy, depth 9:

This image is incredibly big, you can view its full size from the project folder.

Artificial Neural Network:

Artificial neural networks are a type of machine-learning algorithm that is based on the structure of the human brain. I am very hopeful about this algorithm, since it can be used anywhere instead of a specific task like some of the other models. Also it can learn very complex relations since it literally acts like a human brain. We can provide as many parameters as we want to it as neurons.

The minified code:

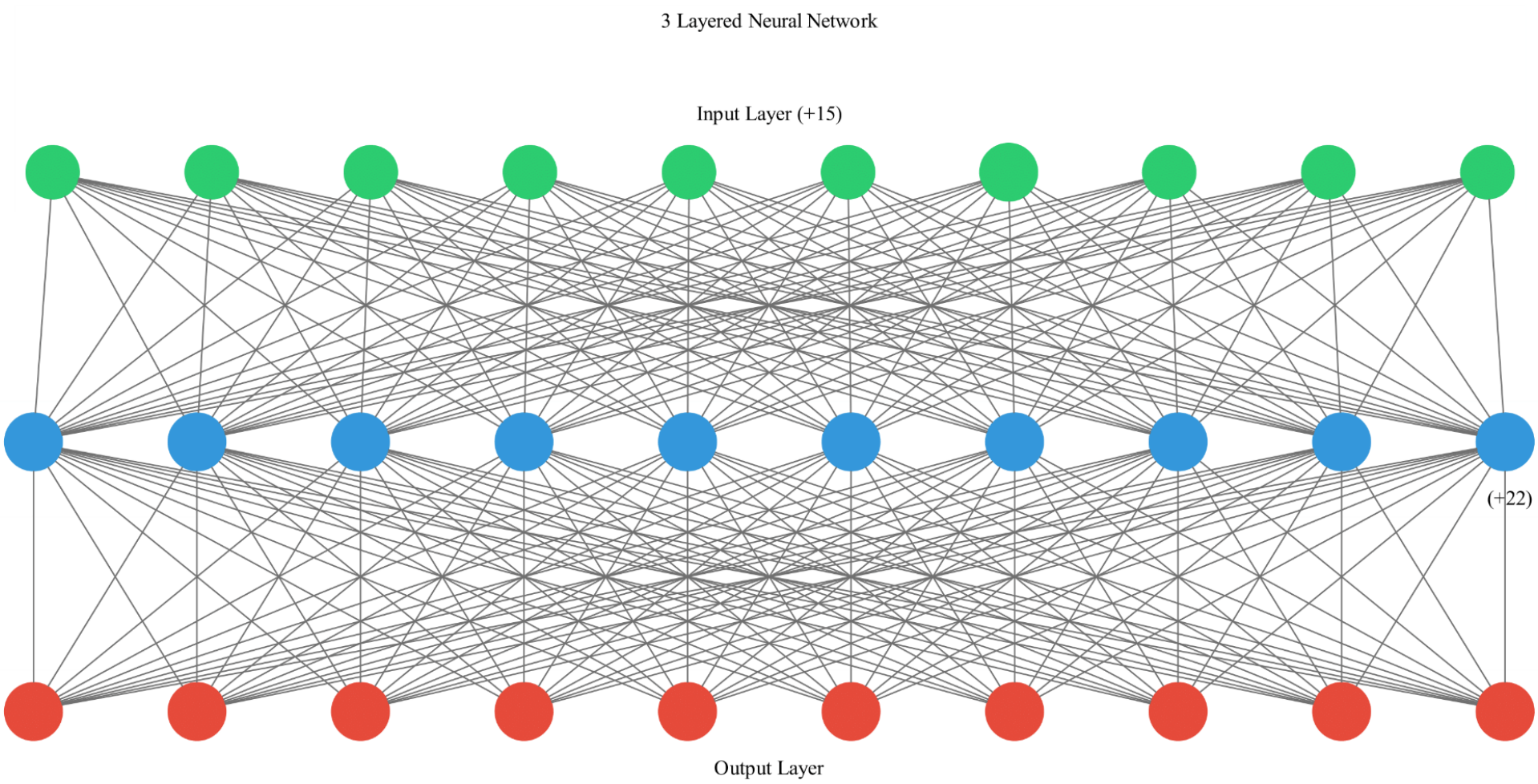
```
def ann(loss, optimizer, epochs, batch_size):
    data = MusicData()
    X_train, X_test, y_train, y_test = data.get_x_y_split()

    model = Sequential()
# This defines 2 layers. The input layer and the first hidden layer.
    model.add(Dense(32, input_dim=len(X_train.columns), activation='relu'))
    model.add(Dense(10, activation='softmax'))  # 3rd Layer (Output)

    model.compile(loss=loss, optimizer=optimizer, metrics=['accuracy'])
    model.summary()  # Print a summary of the Keras model:

    history = model.fit(X_train, y_train, epochs=epochs, batch_size=bath_size, verbose=1)

    _, accuracy = model.evaluate(X_test, y_test)
    print('Accuracy: %.2f' % (accuracy * 100))
```

★ **Epoch:** The entire dataset being passed through the Neural Network model is one epoch.
★ **Batch Size:** The number of samples that will be passed through to the network at one time.
★ **Loss:** I've gone with Binary Crossentropy.
  - The binary cross entropy is useful for training a model to solve multiple classification problems at once if each classification can be reduced to a binary choice.
  - Which in our case I have performed One Hot Encoding to all my target variables. So Binary Crossentropy is the perfect loss function to go for.
★ **Optimizer:** I have chosen **ADAM**. The results of the Adam optimizer are generally better than every other optimization algorithm, have faster computation time, and require fewer parameters for tuning.
★ **Activation Function:** I have chosen **RELU** for the 1st and 2nd layer, and used **Softmax** for my output layer.
  - The reason that I didn't choose Sigmoid is because of Vanishing Gradients.
  - The bigger the input the smaller the gradient of the sigmoid function.
  - The derivative of the sigmoid function will always be smaller than one, so when I have more and more layers and it becomes a deep learning problem, the gradients of Sigmoid go to zero because everytime we take the gradient on a layer we get an even smaller value.
  - So picking RELU was the correct choice here instead of Sigmoid.
  - I used a Softmax on the output layer to ensure our network output is between 0 and 1 and easy to map to either a probability of class or a hard classification of either class.

I have given *epoch as* 30 and 64 *as my batch size*,

Firstly, I started with a fairly small ANN model.
+ It had only 1 hidden layer.
+ In total 3 layers,
  - VariedSize- Input layer being the size of my feature columns.
  - 32- Hidden Layer
  - 10- Output

I have visualized the model as follows:




```
ann('binary_crossentropy', 'adam', 30, 64)
```

Results were as follows:

```
🕐 Timer | Trained the Artificial Neural Network Model : Executed in 22.4369s
------------------------------------------------------------------
254/254 [==============================] - 1s 2ms/step - loss: 0.1853 - accuracy: 0.5668
Accuracy: 56.68
```

I got an accuracy of `56.68%` this didn't satisfy me as I expected more from the Neural Network model, so I have changed my ANN model and added many more parameters to improve accuracy.

+ Added 3 more layers so in total I had 6 layers.
+ Then the layers follow as: 256, 128, 64, 32 and finally 10 as the output for each genre.

The reason I made it like this is, I thought by starting with a large number of neurons on the first layer, I would make many relations between the data itself.

Then as my neuron counts got smaller on each layer, it would combine those many features into more meaningful ones and finally make out which genre the music is.

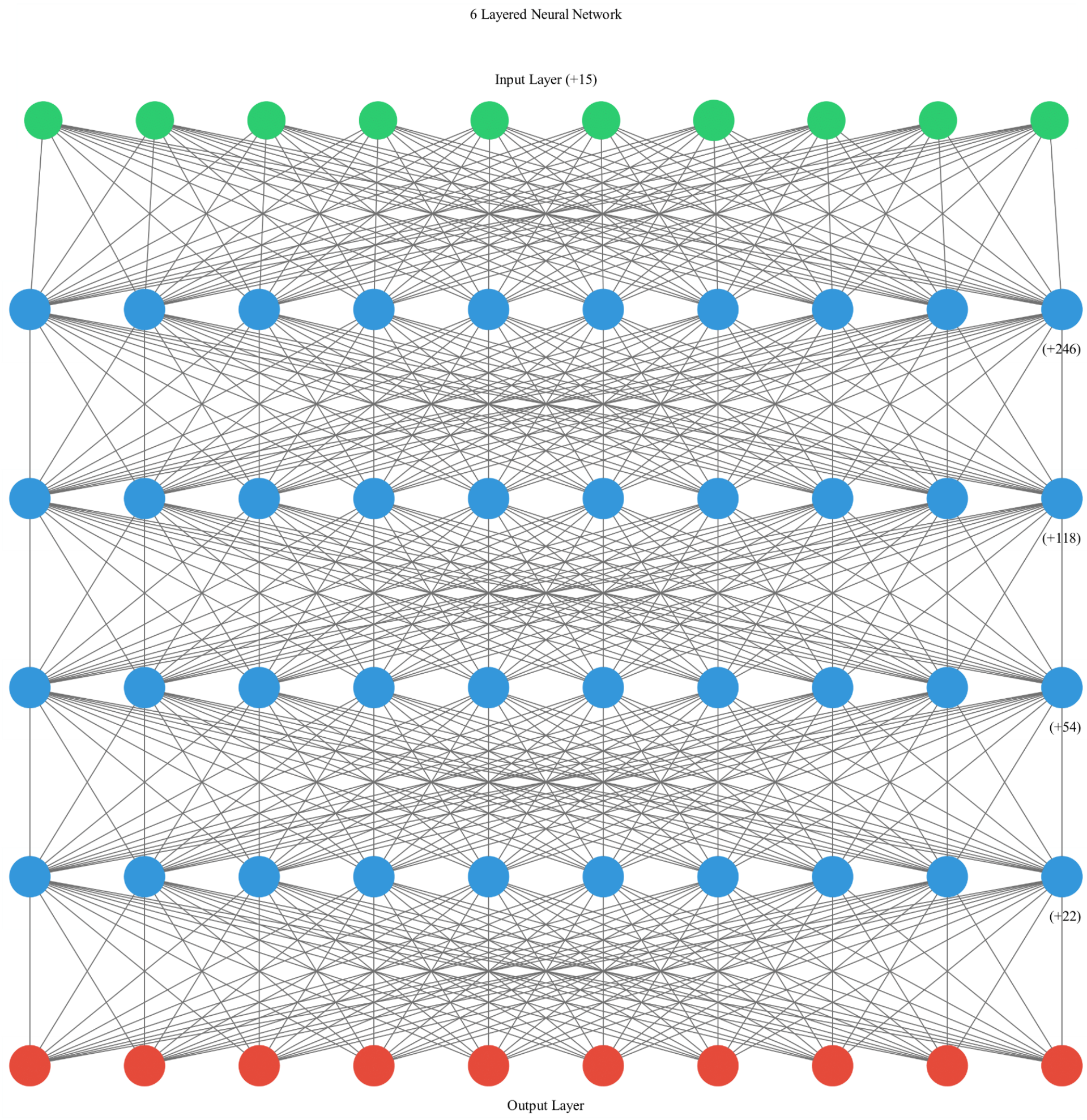
+ This gives me a total of 50,218 parameters on my neural network.

+ All layers except the output layer are activated by RELU and the final output layer is activated by Softmax for the reason I have mentioned above.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 256)               6656

 dense_1 (Dense)             (None, 128)               32896

 dense_2 (Dense)             (None, 64)                8256

 dense_3 (Dense)             (None, 32)                2080

 dense_4 (Dense)             (None, 10)                330
```

```
=================================================================
Total params: 50,218
Trainable params: 50,218
Non-trainable params: 0
_________________________________________________________________
```

Here is the drawing of the model:



The new code:

```
model = Sequential()
model.add(Dense(256, input_dim=len(X_train.columns), activation='relu'))  # This defines 2 layers. The input layer
and the first hidden layer.
model.add(Dense(128, activation='relu'))  # 3rd Layer
model.add(Dense(64, activation='relu'))  # 4th Layer
model.add(Dense(32, activation='relu'))  # 5th Layer
model.add(Dense(10, activation='softmax'))  # 6th Layer (Output)
```

This time I ran it with less epochs but a much smaller batch size.

The runtime increased but the accuracy was better, which was the goal.

```
ann('binary_crossentropy', 'adam', 10, 8)
```

```
Output:
Epoch 1/10
4056/4056 [==============================] - 10s 2ms/step - loss: 0.2037 - accuracy: 0.5158
Epoch 2/10
4056/4056 [==============================] - 7s 2ms/step - loss: 0.1844 - accuracy: 0.5635
Epoch 3/10
4056/4056 [==============================] - 7s 2ms/step - loss: 0.1787 - accuracy: 0.5742
Epoch 4/10
4056/4056 [==============================] - 7s 2ms/step - loss: 0.1747 - accuracy: 0.5838
Epoch 5/10
4056/4056 [==============================] - 7s 2ms/step - loss: 0.1714 - accuracy: 0.5922
Epoch 6/10
4056/4056 [==============================] - 7s 2ms/step - loss: 0.1689 - accuracy: 0.5948
Epoch 7/10
4056/4056 [==============================] - 8s 2ms/step - loss: 0.1662 - accuracy: 0.6044
Epoch 8/10
4056/4056 [==============================] - 7s 2ms/step - loss: 0.1635 - accuracy: 0.6088
Epoch 9/10
4056/4056 [==============================] - 7s 2ms/step - loss: 0.1614 - accuracy: 0.6146
Epoch 10/10
4056/4056 [==============================] - 7s 2ms/step - loss: 0.1586 - accuracy: 0.6222
🕐 Timer | Trained the Artificial Neural Network Model : Executed in 74.6719s
------------------------------------------------------------------
254/254 [==============================] - 1s 2ms/step - loss: 0.1784 - accuracy: 0.5820
Accuracy: 58.20

Process finished with exit code 0
```

- ★ It took 74 seconds to train our model.
- ★ Here we can see that we have an accuracy of `58.20 %` for the test set!
- ★ This is the best accuracy out of all the models I have tested!

---

5) Comparing Models

I have compared all 5 models that I have tested, and plotted some charts to visualize the difference.

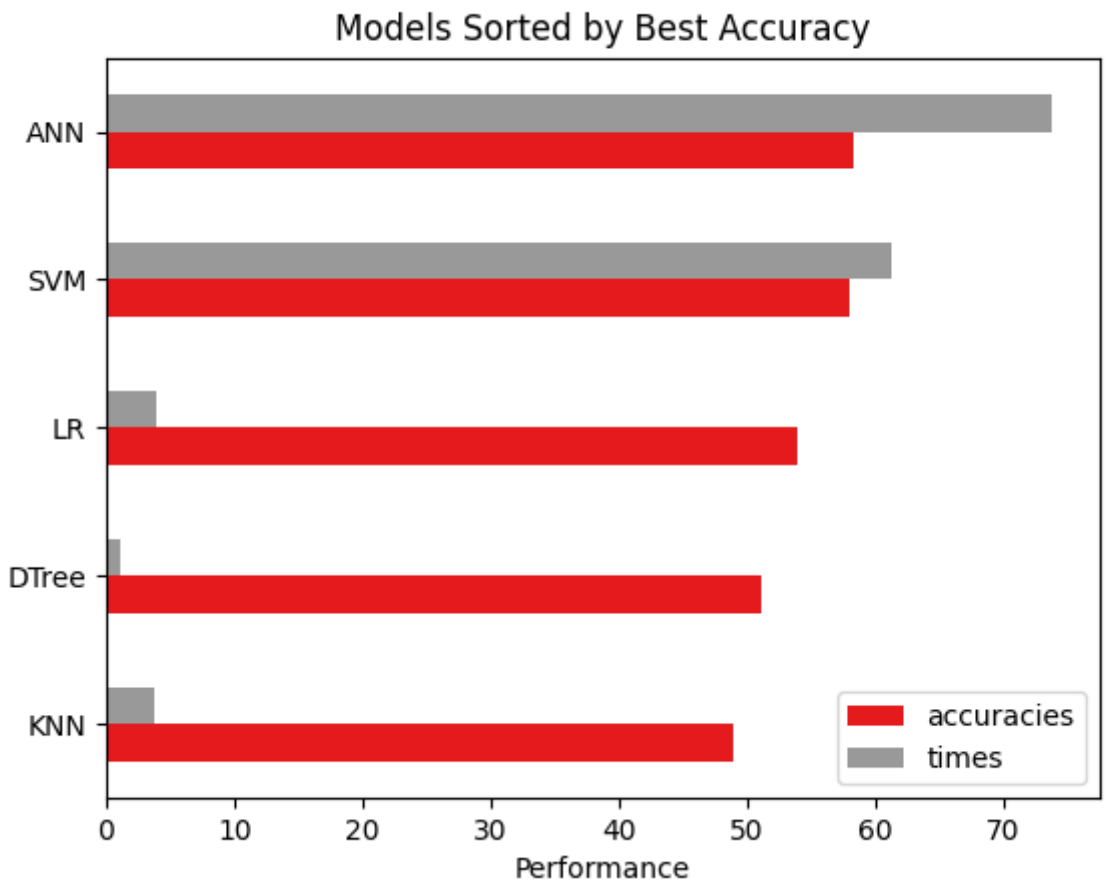
At the “Plotting general information about the dataset” part of my report, I have mentioned that I could provide extra features by combining some features with themselves and potentially increase the accuracy of my models. I have written a method to populate the dataset:

```
def populate_dataset(self, dtree=False):
        self.df['eng_loud'] = self.df['energy'] * self.df['loudness']  # 0.8 Relation
        self.df['eng_acstc'] = self.df['energy'] * self.df['acousticness']  # -0.8 Relation
        self.df['acstc_loud'] = self.df['acousticness'] * self.df['loudness']  # -0.7 Relation
        self.df['loud_instr'] = self.df['loudness'] * self.df['instrumentalness']  # -0.5 Relation
        self.df['loud_dance'] = self.df['loudness'] * self.df['danceability']  # 0.4 Relation

        if not dtree:
            self.df['minor_valence'] = self.df['mode_Minor'] * self.df['valence']
            self.df['major_valence'] = self.df['mode_Major'] * self.df['valence']

        self.df['energy^2'] = self.df['energy'] ** 2
        self.df['popularity^2'] = self.df['popularity'] ** 2
        self.df['valence^2'] = self.df['valence'] ** 2
        self.df['sin(energy)'] = np.sin(self.df['energy'])
        self.df['duration_cbrt'] = np.cbrt(self.df['duration_ms'])
        self.df['liveness_cos'] = np.sqrt(self.df['liveness'])
```

After doing my tests without using this method, I have done some using this method as well to see if any of these columns made any changes. I have added the related columns from the heatmap and also I have added squares, square roots, cubic roots of some features to see if they had provided any extra help.

Firstly I have plotted the results with RobustScaler normalization method and without the dataset being populated by the extra features I talked about previously on the project report.

Robust Scaler | Non-Populated



RobustScaler Non Populated		
	accuracies	times
KNN	48.915187	3.728407
DTree	51.084813	1.125275
LR	53.846154	3.957327
SVM	57.988166	61.200604
ANN	58.222389	73.825566

Here we see that our ***scoring*** order is:

1. **Artificial Neural Network → 58.22 %**
2. **Support Vector Machine → 57.99 %**
3. **Logistic Regression → 53.85 %**
4. **Decision Tree → 51.08 %**
5. **K-Nearest Neighbor → 48.92 %**

As I have expected, the Neural Network has the best accuracy.

The KNN model has the lowest accuracy.

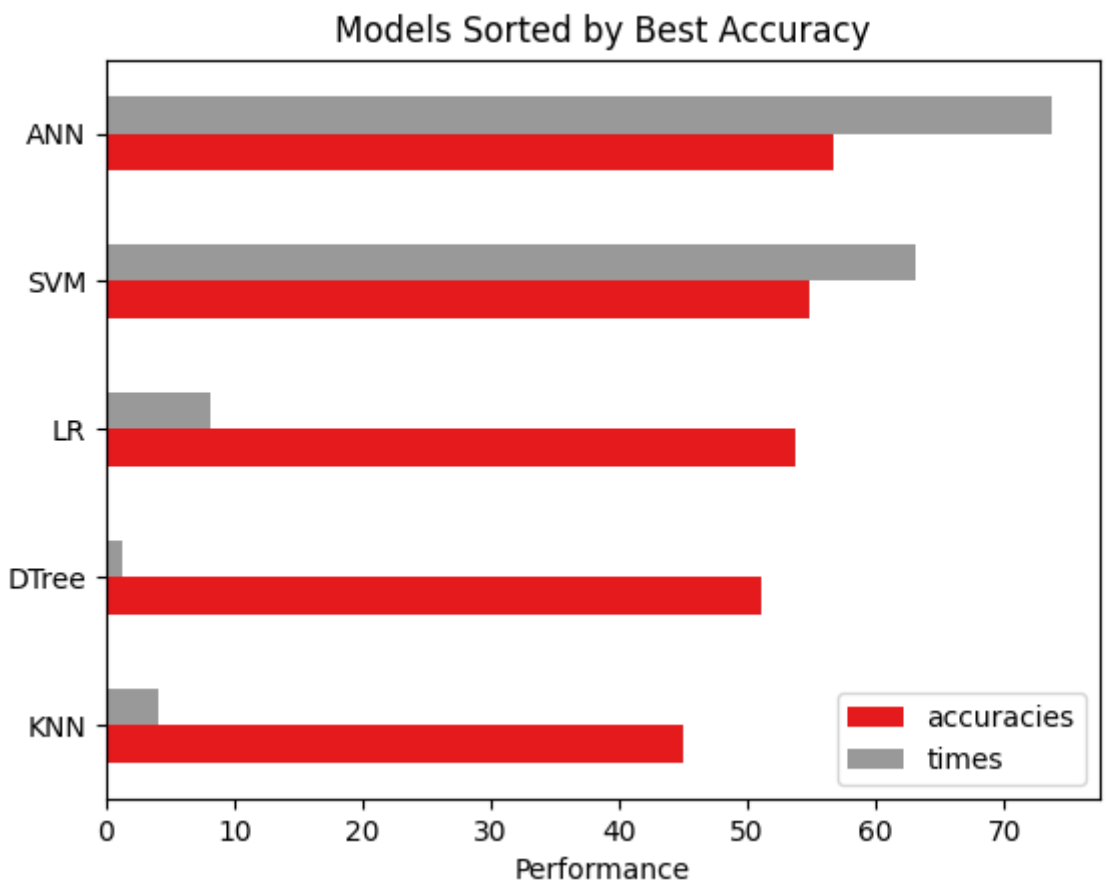
If we look at the ***timings*** we see that

1. **Decision Tree → 1.13s**
2. **K-Nearest Neighbor → 3.73s**
3. **Logistic Regression → 3.96s**
4. **Support Vector Machine → 61.20s**
5. **Artificial Neural Network→ 73.83s**

So even though ANN took the longest time to train, it provided the best results. Meanwhile we should note the speed of the decision tree and its respected accuracy.

Below I have tried to see the differences my Normalization method would make on the accuracies. And finally I have tried this same test with an engineered featured (populated) dataset.

Min-Max Scaler | Non-Populated



MinMaxScaler Non Populated	accuracies	times
KNN	44.995069	4.013001
DTree	51.023176	1.204003
LR	53.759862	8.193000
SVM	54.857002	63.022998
ANN	56.669134	73.772000

Here we see when we used Min-Max scaler instead, we got a lower accuracy overall. I have explained why this was happening eariler in my report. Hence why I had selected my normalization method as RobustScaler instead of Min-Max scaler.

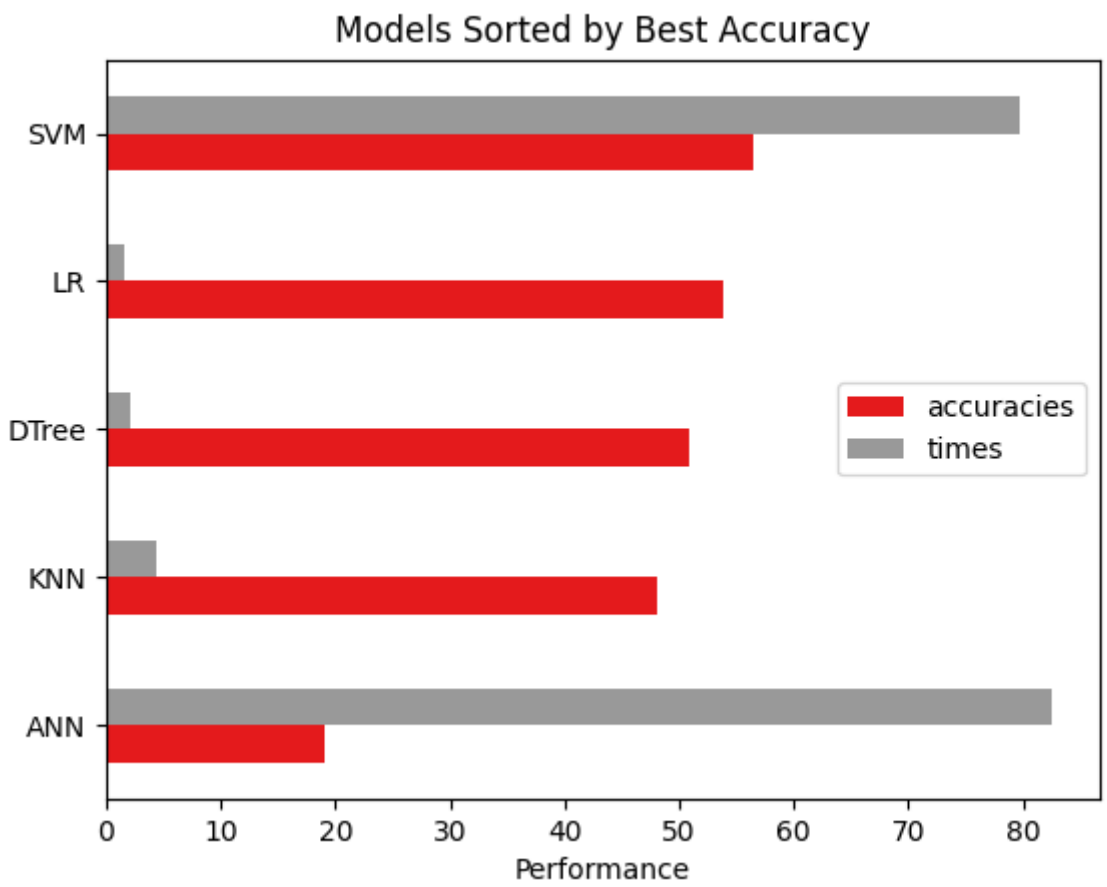
Our scoring order has not changed and it still is:

1. **Artificial Neural Network → 56.67 %**
2. **Support Vector Machine → 54.86 %**
3. **Logistic Regression → 53.76 %**
4. **Decision Tree → 51.02 %**
5. **K-Nearest Neighbor → 45.00 %**

If we look at the ***timings*** we see that nothing has changed just like the scoring:

1. **Decision Tree → 1.20s**
2. **K-Nearest Neighbor → 4.01s**
3. **Logistic Regression → 8.20s**
4. **Support Vector Machine → 63.02s**
5. **Artificial Neural Network → 73.78s**

Standart Scaler | Non-Populated




```
StandardScaler Non Populated
       accuracies      times
ANN     19.045858  82.564559
KNN     48.089250   4.357453
DTree   50.936884   2.031972
LR      53.870809   1.592998
SVM     56.508876  79.743391
```

Like I had mentioned before while selecting my normalization method, doing standardization is not a good idea with One Hot Encoded variables. We can see that our ANN performs terribly.

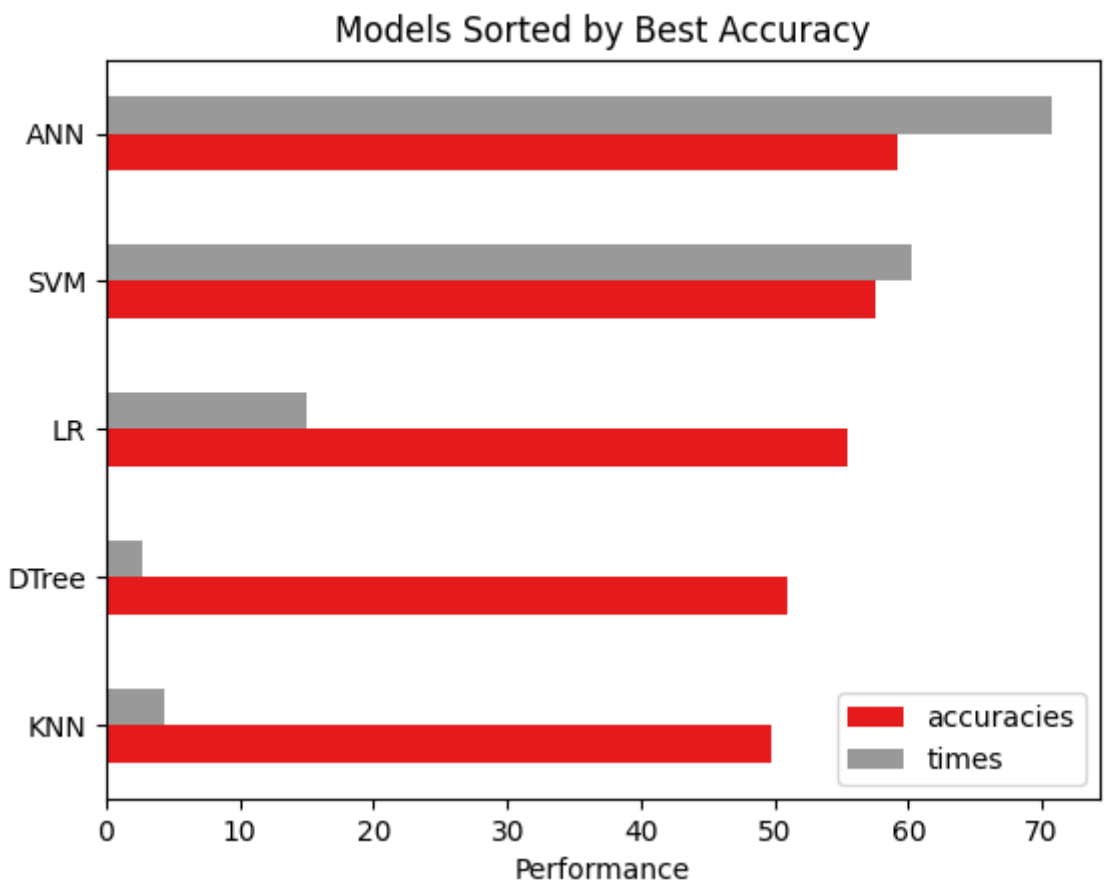
Our scoring order has changed, now the ANN is at the **bottom** of the list!

1. **Support Vector Machine → 56.51 %**
2. **Logistic Regression → 53.87 %**
3. **Decision Tree → 50.94 %**
4. **K-Nearest Neighbor → 48.09 %**
5. **Artificial Neural Network → 19.05 %**

If we look at the timings we see that the previous top 3 has changed:

1. **Logistic Regression → 1.59s**
2. **Decision Tree → 2.03s**
3. **K-Nearest Neighbor → 4.36s**
4. **Support Vector Machine → 79.74s**
5. **Artificial Neural Network → 82.56s**

Robust Scaler | Populated



	RobustScaler Populated	
	accuracies	times
KNN	49.815089	4.329000
DTree	50.936884	2.771217
LR	55.399408	14.931998
SVM	57.593688	60.287024
ANN	59.134614	70.788965

★ As we can see the **best performance** was here when the Dataset was populated with extra features that I made out from the correlation heatmap.
★ ANN achieved a performance of **59.13%** which is the best result I have gotten.

It is hard to increase this number, since genres like Hip-Hop and Rap are almost identical, one way of increasing this result would be to group some Music Genres together. This would improve the prediction accuracy massively but would provide weaker results. Since It would be predicting 2-3 genres instead of one, also the homework instructions are to predict 10 genres. So I didn't want to do any manipulation to the genres.

6) Conclusion

1. This homework has taught me a lot of stuff, I had spent almost 80% of my time trying to prepare my dataset.
2. It has made me learn how to properly prepare, visualize a dataset and do feature selection/engineering.
3. The learning part was the most fun to code and work with.
4. After the data was prepared and ready, coding the learning algorithms were really fun and seeing their results were quite interesting. As I have assumed the Artificial Neural Network performed the best.
5. My function that Populated the Dataset worked quite well, it showed me that my understanding of the dataset proved to be mostly correct.
6. I have learned which models are good for what type of classification. I had to do a lot of research before actually applying the models.

Overall I enjoyed this project, and I am happy with the 60% prediction rate of the Artificial Neural Network.

In the future I could do some of the suggestions I did to improve the accuracy to above 95%, such as using Artist names, Combining some Music Genres as 1 genre, using the Track names length to predict if a song is classical(since they always tend to have longer names) etc.

To run the code yourself, open the project and start running main.py | Make sure you have installed GraphViz, Tensorflow and imported the libraries. There is also a Jupyter notebook for the data exploration parts called notebook.ipynb

AKINCAN KILIÇ 190315044