

```
# imports
from google.colab import drive
drive.mount('/content/drive')
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True)

```
# create pandas dataframe
df = pd.read_csv("/content/drive/MyDrive/songs_normalize.csv")
df.head()
```

	artist	song	duration_ms	explicit	year	popularity	danceability	energy	key	loudness	mode	speechiness
0	Britney Spears	Oops!...I Did It Again	211160	False	2000	77	0.751	0.834	1	-5.444	0	0.0437
1	blink-182	All The Small Things	167066	False	1999	79	0.434	0.897	0	-4.918	1	0.0488
2	Faith Hill	Breathe	250546	False	1999	66	0.529	0.496	7	-9.007	1	0.0290
3	Bon Jovi	It's My Life	224493	False	2000	78	0.551	0.913	0	-4.063	0	0.0466
4	*NSYNC	Bye Bye Bye	200560	False	2000	65	0.614	0.928	8	-4.806	0	0.0516

Code for Data Cleanup:

```
# Count of duplicates
print("Dropped", len(df[df.duplicated(keep='first')]), "duplicate rows!")
# Drop duplicates:
df.drop_duplicates(inplace=True)
```

```
# Data is complete
# print(df.isnull().sum())
```

Dropped 59 duplicate rows!

```
print("Dropped", len(df[df['genre'].isin(['set()'])]), "rows with genre 'set()'")
# Drop rows with genre = 'set()'
df = df[~df['genre'].isin(['set()'])]
```

```
# Reset the index if needed
df = df.reset_index(drop=True)
```

Dropped 22 rows with genre 'set()'.

```
# Create list of unique genres
df['genre'] = df['genre'].str.split(', ')
# Find unique genres (flatten the list of lists and convert to a set)
unique_genres = list(set(genre for genres in df['genre'] for genre in genres))
```

```
print("Unique Genres:", unique_genres)
print("Number of Unique Genres:", len(unique_genres))
```

Unique Genres: ['hip hop', 'easy listening', 'country', 'classical', 'metal', 'rock', 'blues', 'R&B', 'latin', 'jazz', 'World']
Number of Unique Genres: 14

```
# # Map every genre to a number
# genre_mapping = {genre: idx for idx, genre in enumerate(unique_genres)}
# print(genre_mapping)
```

```
# # Replace each genre with the genre mapping assigned
# df['genre'] = df['genre'].apply(lambda genres: sorted([genre_mapping[genre] for genre in genres]))
```

```
# Replace each boolean explicit with a numerical value
df['explicit'] = df['explicit'].apply(lambda explicit: 1 if (explicit == True) else 0)
```

```
# Flatten the 'genre' column
df_genre_column = df.explode('genre')
genre_counts = {}
# Loop through the rows and update genre counts
for index, row in df_genre_column.iterrows():
    if row['genre'] in genre_counts:
        genre_counts[row['genre']] += 1
    else:
        genre_counts[row['genre']] = 1

genre_counts = {k: genre_counts[k] for k in sorted(genre_counts)}
```

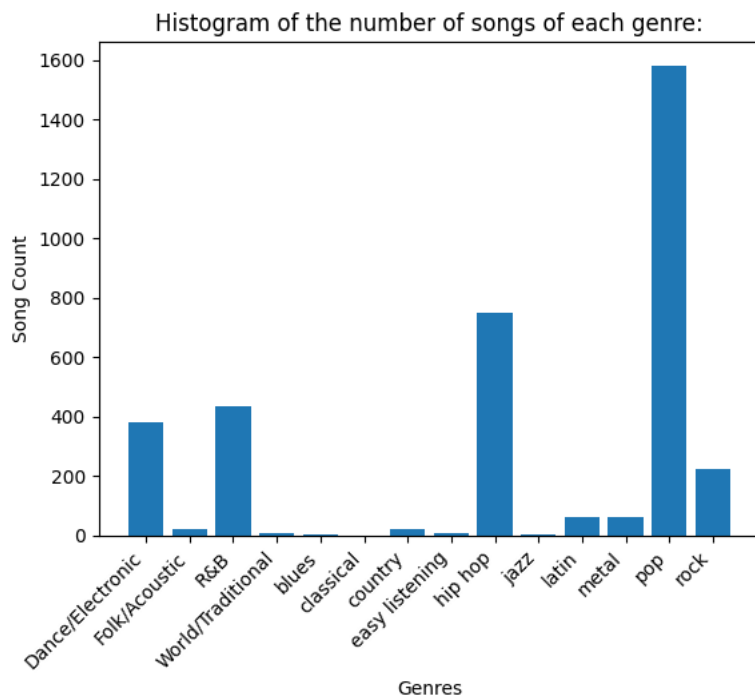
```
# Create a histogram
plt.bar(genre_counts.keys(), genre_counts.values())
```

```
# Label the axes and add a title
plt.xlabel('Genres')
plt.ylabel('Song Count')
plt.title('Histogram of the number of songs of each genre:')
```

```
# Adjust the appearance of x-axis labels
plt.xticks(rotation=45, ha='right') # Rotate labels by 45 degrees
```

```
# Show the histogram
plt.show()
```

```
{'Dance/Electronic': 380, 'Folk/Acoustic': 20, 'R&B': 437, 'World/Traditional': 10, 'blues': 4, 'classical': 1, 'country': 20, 'easy listening': 1, 'hip hop': 750, 'jazz': 1, 'latin': 60, 'metal': 60, 'pop': 1580, 'rock': 220}
```



We can drop the songs which solely belong to genres **Folk/Acoustic**, **World/Traditional**, **blues**, **classical**, **country**, **easy listening** and **jazz** because we have very little data about songs from these genres as you can see above (song count <= 20). Our model would consider these to be outliers or provide wrong predictions for these genres. Therefore, we will work with the remaining 7 genres.

```
# List of the genres to drop
genres_to_drop = ['Folk/Acoustic', 'World/Traditional', 'blues', 'classical', 'country', 'easy listening', 'jazz']
```

```
# Loop through the DataFrame and change the 'genre' column
for index, row in df.iterrows():
    modified_genre = [genre for genre in row['genre'] if genre not in genres_to_drop]
    df.at[index, 'genre'] = modified_genre
```

```
# Filter out songs with empty genre lists
df = df[df['genre'].apply(len) > 0]

# Reset the index if needed
df = df.reset_index(drop=True)

# Flatten the 'genre' column
df_genre_column = df.explode('genre')
genre_counts = {}
# Loop through the rows and update genre counts
for index, row in df_genre_column.iterrows():
    if row['genre'] in genre_counts:
        genre_counts[row['genre']] += 1
    else:
        genre_counts[row['genre']] = 1

genre_counts = {k: genre_counts[k] for k in sorted(genre_counts)}
unique_genres = list(genre_counts.keys())
print(genre_counts)

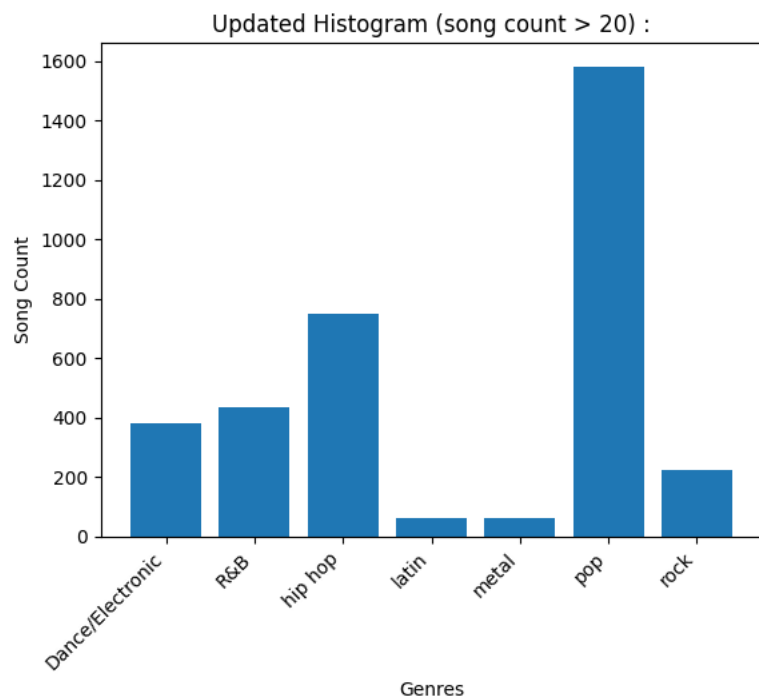
# Create a histogram
plt.bar(genre_counts.keys(), genre_counts.values())

# Label the axes and add a title
plt.xlabel('Genres')
plt.ylabel('Song Count')
plt.title('Updated Histogram (song count > 20) :')

# Adjust the appearance of x-axis labels
plt.xticks(rotation=45, ha='right') # Rotate labels by 45 degrees

# Show the histogram
plt.show()
```

{'Dance/Electronic': 380, 'R&B': 437, 'hip hop': 751, 'latin': 63, 'metal': 64, 'pop': 1582, 'rock': 225}



We can drop the **artist**, **song**, **duration_ms**, **year**, **popularity**, **genre** columns:

artist, song: Both of these columns do not provide any information in terms of the audio quality or sound characteristics and hence are not required for genre classification.

duration_ms: The duration of the song does not give us any significant insight for genre classification.

year, popularity: Year of release and popularity ranking of the song may be important information regarding historic music trends and genre popularity but these features are not required for genre classification.

```
try:
    df = df.drop(['artist','song','duration_ms','year','popularity'], axis=1)
except:
    pass
# We need to remove the genre column later to apply our unsupervised learning models.
df.head()
```

	explicit	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	liveness	valence	tempo
0	0	0.751	0.834	1	-5.444	0	0.0437	0.3000	0.000018	0.3550	0.894	95.053
1	0	0.434	0.897	0	-4.918	1	0.0488	0.0103	0.000000	0.6120	0.684	148.726
2	0	0.529	0.496	7	-9.007	1	0.0290	0.1730	0.000000	0.2510	0.278	136.859
3	0	0.551	0.913	0	-4.063	0	0.0466	0.0263	0.000013	0.3470	0.544	119.992
4	0	0.614	0.928	8	-4.806	0	0.0516	0.0408	0.001040	0.0845	0.879	172.656

We will now **Normalize** all the entries in the dataframe using MinMaxScaler which will transform all the entries to values between -1 and 1 based on the min and max values of the column.

```
from sklearn.preprocessing import MinMaxScaler

# Initializing the MinMaxScaler
scaler = MinMaxScaler(feature_range=(-1, 1))
# We have to scale every column except the last one
df_to_scale = df.iloc[:, :-1]
# Normalizing all the entries in the DataFrame
df_normalized = pd.DataFrame(scaler.fit_transform(df_to_scale), columns=df_to_scale.columns)
df_normalized['genre'] = df['genre']

df_normalized.head()
```

	explicit	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	liveness	valence
0	-1.0	0.470449	0.650461	-0.818182	0.489278	-1.0	-0.925832	-0.385273	-0.999964	-0.197835	0.830998
1	-1.0	-0.278960	0.783921	-1.000000	0.541259	1.0	-0.907381	-0.978932	-1.000000	0.420325	0.381752
2	-1.0	-0.054374	-0.065565	0.272727	0.137168	1.0	-0.979016	-0.645524	-1.000000	-0.447986	-0.486790
3	-1.0	-0.002364	0.817816	-1.000000	0.625754	-1.0	-0.915340	-0.946145	-0.999973	-0.217078	0.082255
4	-1.0	0.146572	0.849592	0.454545	0.552327	-1.0	-0.897250	-0.916431	-0.997888	-0.848467	0.798909

```
# Explode the 'genre' column to create a row for each genre
expanded_df = df_normalized.explode('genre')
# Calculate the mean values for each genre
df_genre_means = pd.pivot_table(expanded_df, index='genre', values=list(expanded_df.columns), aggfunc='mean').reset_index()
print("Mean values of each feature for every genre in the dataframe.\n".center(150, " "))
df_genre_means
```

Mean values of each feature for every genre in the dataframe.

	genre	acousticness	danceability	energy	explicit	instrumentalness	key	liveness	loudness	mode	speechi
0	Dance/Electronic	-0.778955	0.273373	0.511615	-0.768421	-0.922069	0.042105	-0.607016	0.518938	0.015789	-0.71
1	R&B	-0.690498	0.325500	0.286296	-0.432494	-0.986500	-0.032245	-0.657804	0.440895	0.038902	-0.66
2	hip hop	-0.761480	0.393342	0.366361	0.105193	-0.987489	0.017068	-0.597393	0.472718	0.083888	-0.56
3	latin	-0.688925	0.409134	0.555165	-0.809524	-0.998660	-0.004329	-0.582170	0.564235	0.174603	-0.76
4	metal	-0.916893	-0.103354	0.644470	-0.718750	-0.989436	-0.068182	-0.594284	0.547491	0.250000	-0.91
5	pop	-0.730154	0.281021	0.400245	-0.496839	-0.976530	-0.027238	-0.618143	0.481590	0.096081	-0.71
6	rock	-0.848472	0.009309	0.558512	-0.822222	-0.938355	-0.049697	-0.581506	0.518222	0.262222	-0.86

We have a list of genres for many songs but our unsupervised learning model would categorize the song to a particular genre. Hence, we have to provide one genre label to each song. We will do this by using the mean values we calculated above for each genre. By taking the euclidean distance between each song's audio feature values and the mean values from above, we will choose the genre that best represents a song. Consequently, we will remove all other genres from that song's genre list.

```
# Loop through the DataFrame and change the 'genre' column
for index, row in df_normalized.iterrows():
    dists = []
    for genre in row['genre']:
        genre_features = df_genre_means[df_genre_means['genre'] == genre].iloc[:,1:]
        song_features = df_normalized.iloc[index,:-1]
        # Create numpy arrays of feature values to find euclidean distance
        genre_values = genre_features[df_normalized.iloc[:, :-1].columns].values[0]
        song_values = song_features[df_normalized.iloc[:, :-1].columns].values
        # Calculate the Euclidean distance
        euclidean_dist = np.linalg.norm(song_values - genre_values)
        # Append to the dists array
        dists.append((genre, euclidean_dist))
    if index < 5:
        print("Array of Euclidean Distances: ", dists)
        print("Selected Genre: ", min(dists, key=lambda x: x[1])[0])
# Get the genre with the minimum distance and update the 'genre' column of the dataframe
df_normalized.at[index, 'genre'] = min(dists, key=lambda x: x[1])[0]
df_normalized.head()
```

Array of Euclidean Distances: [('pop', 1.7752394429909757)]
 Selected Genre: pop
 Array of Euclidean Distances: [('rock', 1.6859303351942319), ('pop', 1.9699797398924441)]
 Selected Genre: rock
 Array of Euclidean Distances: [('pop', 1.4540178813055917)]
 Selected Genre: pop
 Array of Euclidean Distances: [('rock', 1.6622885702272139), ('metal', 1.6580877882930478)]
 Selected Genre: metal
 Array of Euclidean Distances: [('pop', 1.7332948990314008)]
 Selected Genre: pop


	explicit	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	liveness	valence	
0	-1.0	0.470449	0.650461	-0.818182	0.489278	-1.0	-0.925832	-0.385273	-0.999964	-0.197835	0.830998	-0
1	-1.0	-0.278960	0.783921	-1.000000	0.541259	1.0	-0.907381	-0.978932	-1.000000	0.420325	0.381752	0
2	-1.0	-0.054374	-0.065565	0.272727	0.137168	1.0	-0.979016	-0.645524	-1.000000	-0.447986	-0.486790	0
3	-1.0	-0.002364	0.817816	-1.000000	0.625754	-1.0	-0.915340	-0.946145	-0.999973	-0.217078	0.082255	-0
4	-1.0	0.146572	0.849592	0.454545	0.552327	-1.0	-0.897250	-0.916431	-0.997888	-0.848467	0.798909	0

We need to reduce the number of the classes for our unsupervised learning models and therefore we will use the euclidean distances between genre feature means to find similar genres.

```
from itertools import combinations

# Calculate distances between all pairs of rows
distances = []
for pair in combinations(df_genre_means.iterrows(), 2):
    row1, row2 = pair[0][1], pair[1][1]
    genre1_data = row1[df_genre_means.iloc[:,1:].columns].values
    genre2_data = row2[df_genre_means.iloc[:,1:].columns].values
    dist = np.linalg.norm(genre1_data - genre2_data)
    if dist < 0.35:
        distances.append((row1['genre'], row2['genre'], dist))

# Create a DataFrame to store the distances
distance_df = pd.DataFrame(distances, columns=['Genre1', 'Genre2', 'Euclidean Distance'])
distance_df
```

	Genre1	Genre2	Euclidean Distance	
0	Dance/Electronic	pop	0.339293	
1	R&B	pop	0.182204	
2	metal	rock	0.268807	

The genres above have similar mean values but I am not sure if the euclidean distance is a good measure to eliminate labels because due to many features we have multidimensional data.

```
# # Code to find similarity of features for every genre based on feature means

# # Create a dictionary to map genres to colors
```

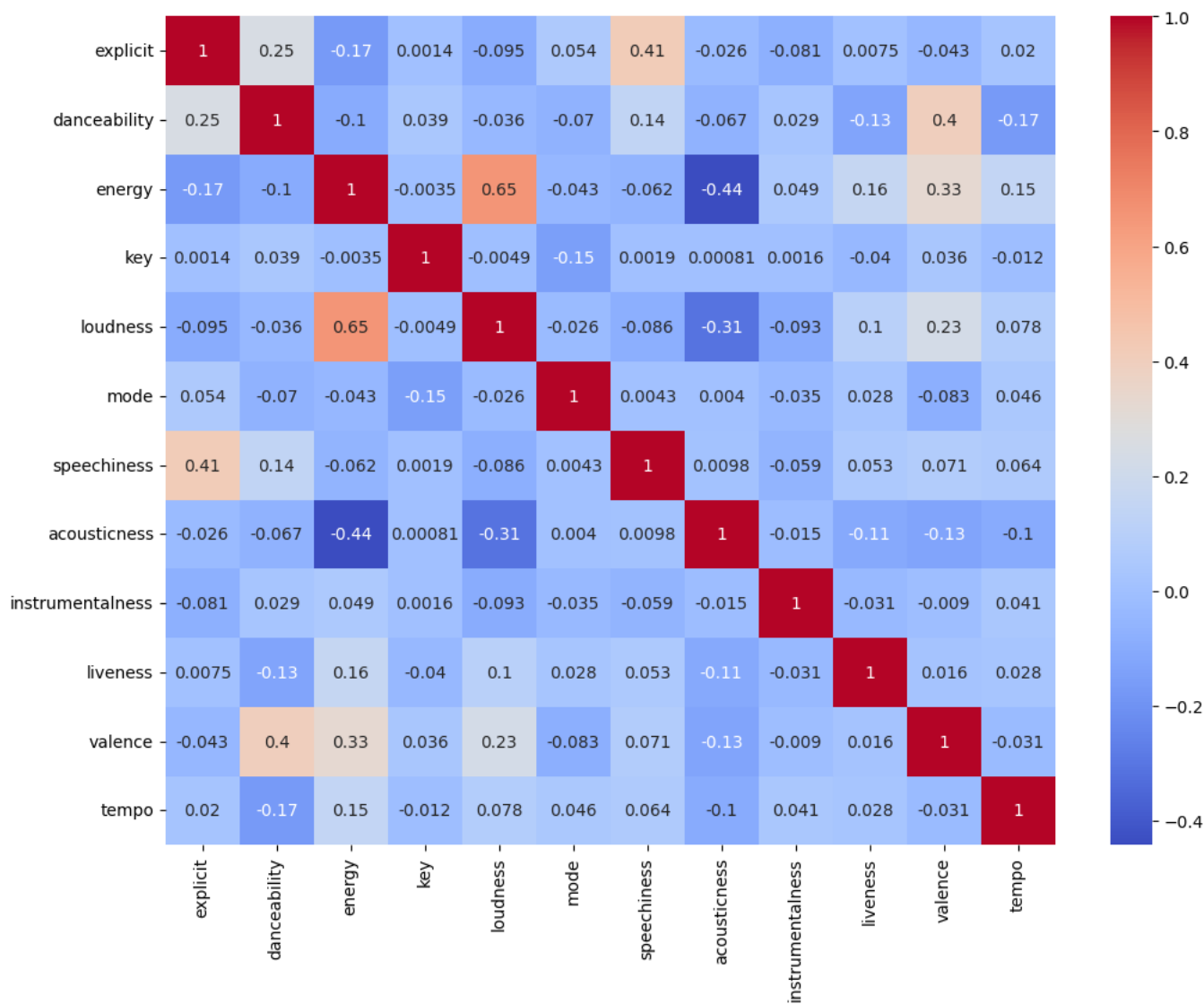
```
# genre_colors = {'Dance/Electronic': 'red', 'Folk/Acoustic': 'blue', 'R&B': 'green', 'country': 'black', 'hip hop': 'yellow', \
#                 'latin': 'grey', 'metal': 'violet', 'pop': 'indigo', 'rock': 'orange'}

# # Plot the points with different colors and labels
# for label, color in genre_colors.items():
#     subset = df_genre_means[df_genre_means['genre'] == label]
#     plt.scatter(subset['acousticness'], subset['explicit'], c=color, label=label)

# # Add labels and legend
# plt.xlabel('acousticness')
# plt.ylabel('explicit')
# plt.legend()
# # Show the plot
# plt.show()

corr_mat = df_normalized.iloc[:, :-1].corr()
plt.subplots(figsize=(12,9))
plt.title('\nCorrelation between Features:\n', fontsize = 20)
sns.heatmap(corr_mat, cmap='coolwarm', annot=True)
plt.show()
```

Correlation between Features:



```
# Code to print out highly correlated features
thresholds = [0.8, 0.6, 0.4] # Adjust the threshold as needed

# Took the upper triangular part of the correlation matrix
```

```
upper_triangular = np.triu(corr_mat, k=1)

# Highly correlated feature pairs
for threshold in thresholds:
    high_corr_pairs = [(corr_mat.columns[i], corr_mat.columns[j]) for i in range(upper_triangular.shape[0]) \
                        for j in range(i+1, upper_triangular.shape[1]) if abs(upper_triangular[i, j]) > threshold]

print(threshold, high_corr_pairs)

0.8 []
0.6 [('energy', 'loudness')]
0.4 [('explicit', 'speechiness'), ('danceability', 'valence'), ('energy', 'loudness'), ('energy', 'acousticness')]
```

The correlation doesn't seem to be high between features and so we will move forward with all the features.

We will now split the data into the training (80%) and the test set (20%). The genre column will be the class labels to classify the songs to.

```
from sklearn.model_selection import train_test_split

X = df_normalized.iloc[:, :-1]
y = df_normalized.iloc[:, -1:]

xtrain, xtest, ytrain, ytest = train_test_split(X, y, test_size=0.2, random_state=42, shuffle=True)
cols = xtrain.columns

# To deal with outliers, we will use robust scaler to scale our data
xtrain = scaler.fit_transform(xtrain)
xtest = scaler.transform(xtest)

xtrain = pd.DataFrame(xtrain, columns = cols)
xtest = pd.DataFrame(xtest, columns = cols)

xtrain.describe()
```

	explicit	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	liveness
count	1526.000000	1526.000000	1526.000000	1526.000000	1526.000000	1526.000000	1526.000000	1526.000000	1526.000000	1526.000000
mean	-0.433814	0.264940	0.408104	-0.020732	0.515861	0.106160	-0.711363	-0.739947	-0.969047	-0.617000
std	0.901298	0.335521	0.325470	0.654970	0.199157	0.994675	0.345957	0.349806	0.180313	0.339533
min	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000
25%	-1.000000	0.056738	0.199767	-0.636364	0.415976	-1.000000	-0.940937	-0.970531	-1.000000	-0.844370
50%	-1.000000	0.289598	0.442856	0.090909	0.537676	1.000000	-0.865051	-0.887742	-1.000000	-0.753450
75%	1.000000	0.491726	0.658405	0.454545	0.653560	1.000000	-0.624457	-0.639377	-0.999851	-0.467820
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

```
xtest.describe()
```

	explicit	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	liveness
count	382.000000	382.000000	382.000000	382.000000	382.000000	382.000000	382.000000	382.000000	382.000000	382.000000
mean	-0.460733	0.307626	0.426807	-0.021894	0.525660	0.089005	-0.682319	-0.740411	-0.973066	-0.617000
std	0.888703	0.318667	0.313753	0.663250	0.182761	0.997337	0.362766	0.352723	0.155671	0.339533
min	-1.000000	-0.879433	-0.563394	-1.000000	-0.233134	-1.000000	-0.997467	-0.999997	-1.000000	-0.988450
25%	-1.000000	0.120567	0.220422	-0.636364	0.434131	-1.000000	-0.930807	-0.975500	-1.000000	-0.844370
50%	-1.000000	0.321513	0.460862	-0.090909	0.550319	1.000000	-0.846599	-0.879443	-1.000000	-0.753450
75%	1.000000	0.538416	0.673763	0.454545	0.647492	1.000000	-0.529486	-0.670627	-0.999899	-0.467820
max	1.000000	0.985816	0.957632	1.000000	1.046930	1.000000	0.667149	0.952868	0.524873	0.874920

The LabelEncoder converts all the data into numeric values, making it computationally easy to use.

```
from sklearn.preprocessing import LabelEncoder
```

```

LE = LabelEncoder()

ytrain = LE.fit_transform(ytrain)
ytest = LE.transform(ytest)

X = pd.concat([xtrain, xtest], axis=0)
y = pd.concat([pd.DataFrame(ytrain), pd.DataFrame(ytest)], axis=0)

y_train = LE.inverse_transform(ytrain)
y_test = LE.inverse_transform(ytest)

y_original = pd.concat([pd.DataFrame(y_train), pd.DataFrame(y_test)], axis=0)

/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_label.py:116: DataConversionWarning: A column-vector y was passed as a 1D array, which has been converted to a 1D array in the background. This behavior may change in the future. Recommended: y = column_or_1d(y, warn=True)
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_label.py:134: DataConversionWarning: A column-vector y was passed as a 1D array, which has been converted to a 1D array in the background. This behavior may change in the future. Recommended: y = column_or_1d(y, dtype=self.classes_.dtype, warn=True)

```

Now we will be running PCA on the dataframe in order to reduce the number of dimensions and remove highly correlated data while retaining most information.

PCA uses an orthogonal transformation to convert highly correlated data into a lower dimensional linear space that:

- Maximizes variance of projected data
- Minimizes mean squared distance between data point and projections

The newly obtained set of values have low correlation and are called principal components. There are three common approaches to finding the number of principal components:

- Eigenvalue criterion
- Proportion of variance explained criterion
- Scree plot criterion

We will be using the Scree Plot criterion. A Scree Plot is a simple line segment plot that shows the eigenvalues for each individual PC. It shows the eigenvalues on the y-axis and the number of factors on the x-axis. It always displays a downward curve. The scree plots generally start high on the left, and fall rather quickly eventually flattening out. The first component explains high variance, the next few components explain moderate variance, and the rest components explain low variance. The scree plot criterion looks for the "elbow" in the curve and selects all components right before the line flattens out.

```
from sklearn.decomposition import PCA
```

```
cov_matrix_PCA = PCA(n_components=len(X.columns))
cov_matrix_PCA.fit(X)
```

```

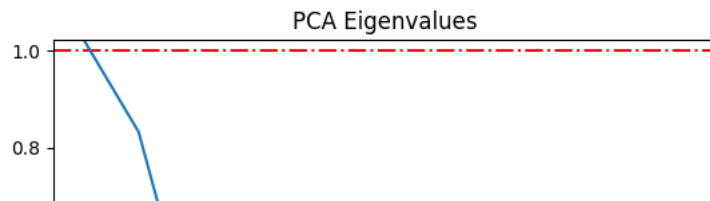
PCA
PCA(n_components=12)

```

```

plt.xlabel('Number of Features')
plt.ylabel('Eigenvalues')
plt.title('PCA Eigenvalues')
plt.ylim(0, max(cov_matrix_PCA.explained_variance_))
plt.style.context('seaborn-whitegrid')
plt.axhline(y=1, color='r', linestyle='-.')
plt.plot(cov_matrix_PCA.explained_variance_)
plt.show()

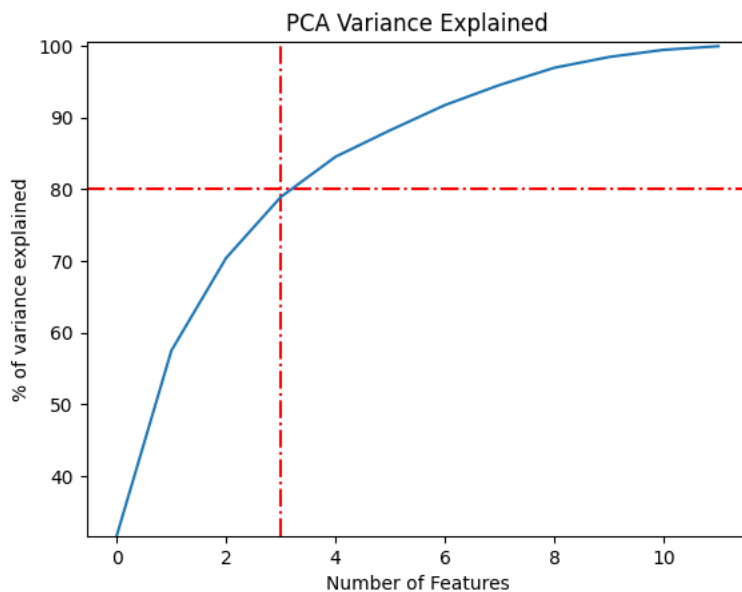
```

Kaiser Rule: Select principal components with eigenvalues of at least 1.

This rule provides a simple and straightforward approach to selecting PCAs. However, for our case it is telling us to select only 1 principal component which may not be ideal. This probably implies that our data has varying variances across dimensions.

```
var = cov_matrix_PCA.explained_variance_ratio_
var = np.cumsum(np.round(cov_matrix_PCA.explained_variance_ratio_, decimals=3)*100)
plt.xlabel('Number of Features')
plt.ylabel('% of variance explained')
plt.title('PCA Variance Explained')
plt.ylim(min(var), 100.5)
plt.style.context('seaborn-whitegrid')
plt.axhline(y=80, color='r', linestyle='-.')
plt.axvline(x=3, color='r', linestyle='-.')
plt.plot(var)
plt.show()
```



Proportion of Variance Plot: This approach involves looking at the cumulative proportion of variance explained by each PC and selecting a number of PCs that can collectively describe 80% of the total variance. This method allows us to capture a substantial amount of information while potentially reducing dimensionality effectively. It is more flexible and shows that we should select 3 principal components. Thus we will move forward with **3 PCs**.

```
# Run PCA with 3 principal components
pca = PCA(n_components=3)
X_PCA = pca.fit_transform(X, y)

# Create a 3D scatter plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

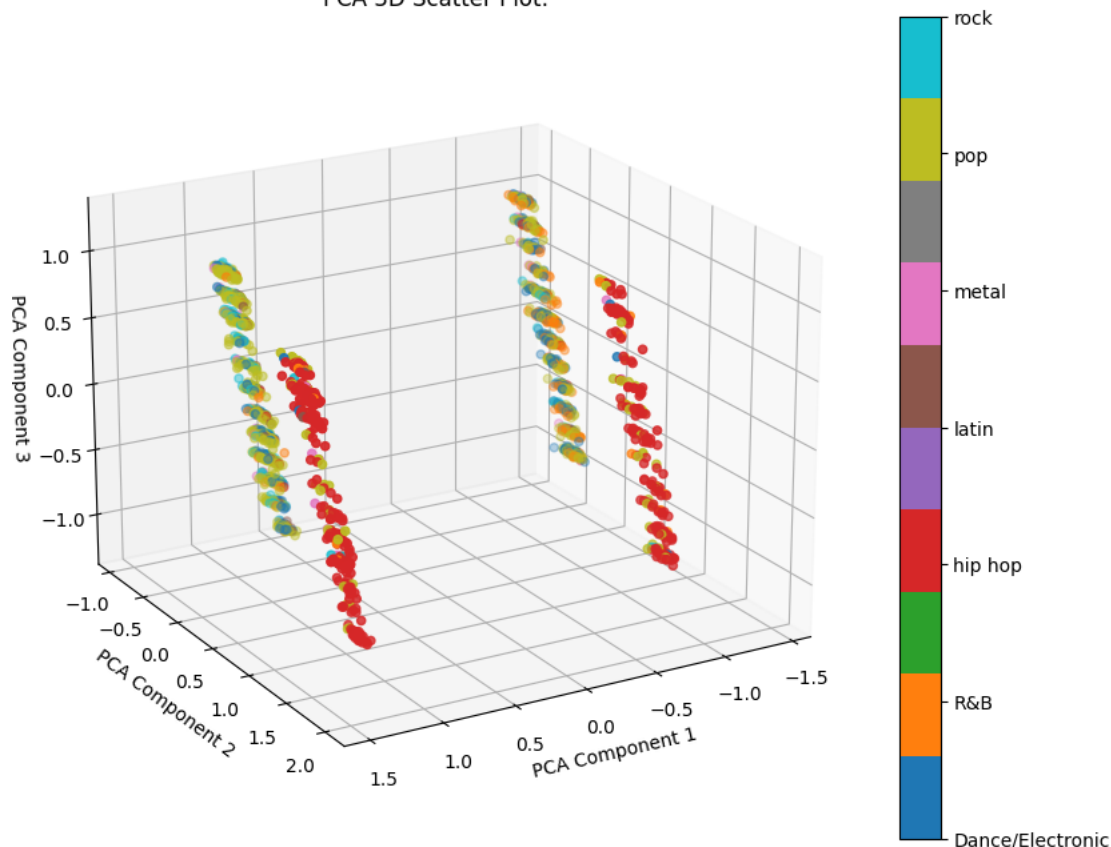
# Defined a colormap for assigning colors based on labels
cmap = plt.get_cmap('tab10')
# Scatter plot with the first three PCA components
scatter = ax.scatter(X_PCA[:, 0], X_PCA[:, 1], X_PCA[:, 2], c=y, cmap=cmap)

# Created a custom colorbar with labels based on unique classes
cbar = plt.colorbar(scatter, ax=ax, ticks=np.unique(y))
cbar.set_ticklabels(np.unique(y_original))
cbar.set_label('Predicted Labels')
```

```
# Add labels and title
ax.set_xlabel('PCA Component 1')
ax.set_ylabel('PCA Component 2')
ax.set_zlabel('PCA Component 3')
plt.title('PCA 3D Scatter Plot:')
ax.view_init(elev=20, azim=60)
```

```
plt.show()
```

PCA 3D Scatter Plot:



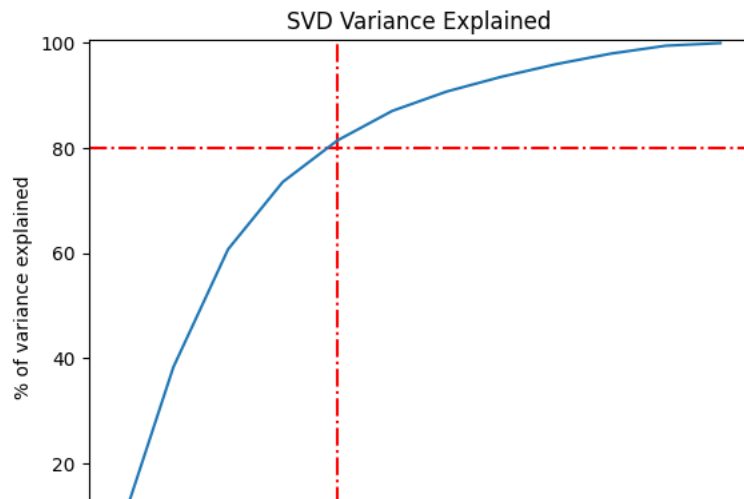
Now we will be running **SVD** on the dataframe which provides us with a **reduced-rank optimization** of the data. We will perform linear dimensionality reduction by means of truncated singular value decomposition.

```
from sklearn.decomposition import TruncatedSVD
```

```
cov_matrix_SVD = TruncatedSVD(n_components=len(X.columns))
cov_matrix_SVD.fit(X)
```

```
▼ TruncatedSVD
TruncatedSVD(n_components=12)
```

```
var = cov_matrix_SVD.explained_variance_ratio_
var = np.cumsum(np.round(cov_matrix_SVD.explained_variance_ratio_, decimals=3)*100)
plt.xlabel('Number of Features')
plt.ylabel('% of variance explained')
plt.title('SVD Variance Explained')
plt.ylim(min(var), 100.5)
plt.style.context('seaborn-whitegrid')
plt.axhline(y=80, color='r', linestyle='-.')
plt.axvline(x=4, color='r', linestyle='-.')
plt.plot(var)
plt.show()
```



Proportion of Variance Plot: This approach involves looking at the cumulative proportion of variance explained by each component and selecting the number of components that can collectively describe 80% of the total variance. This method allows us to capture a substantial amount of information while potentially reducing dimensionality effectively. It is more flexible and shows that we should select 4 components. Thus we will move forward with **4 components**.

```
# Run Truncated SVD with 4 components
svd = TruncatedSVD(n_components=4)
X_SVD = svd.fit_transform(X, y)

# Create a 3D scatter plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

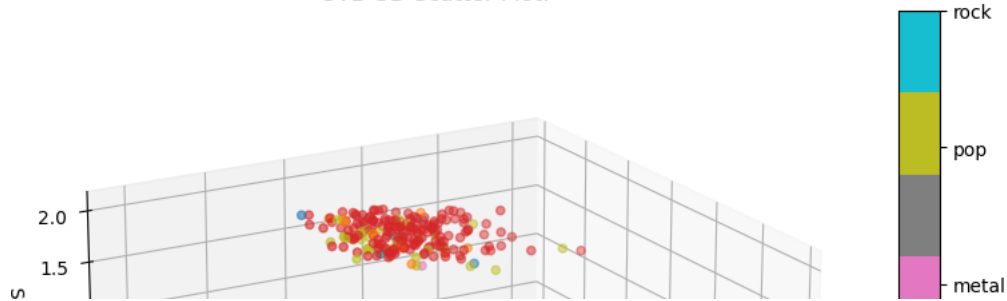
# Defined a colormap for assigning colors based on labels
cmap = plt.get_cmap('tab10')
# Scatter plot with the first three SVD components
scatter = ax.scatter(X_SVD[:, 0], X_SVD[:, 1], X_SVD[:, 2], c=y, cmap=cmap)

# Created a custom colorbar with labels based on unique classes
cbar = plt.colorbar(scatter, ax=ax, ticks=np.unique(y))
cbar.set_ticklabels(np.unique(y_original))
cbar.set_label('Predicted Labels')

# Add labels and title
ax.set_xlabel('SVD Component 1')
ax.set_ylabel('SVD Component 2')
ax.set_zlabel('SVD Component 3')
plt.title('SVD 3D Scatter Plot:')
ax.view_init(elev=20, azim=60)

plt.show()
```

SVD 3D Scatter Plot:



This scatterplot represents 3 of the 4 Truncated SVD components.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

cov_matrix_LDA = LinearDiscriminantAnalysis(n_components=len(np.unique(y_original)) - 1)
X_LDA = cov_matrix_LDA.fit_transform(X, y.values.ravel())

# Create a 3D scatter plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Defined a colormap for assigning colors based on labels
cmap = plt.get_cmap('tab10')
# Scatter plot with the first three LDA components
scatter = ax.scatter(X_LDA[:, 0], X_LDA[:, 1], X_LDA[:, 2], c=y, cmap=cmap)

# Created a custom colorbar with labels based on unique classes
cbar = plt.colorbar(scatter, ax=ax, ticks=np.unique(y))
cbar.set_ticklabels(np.unique(y_original))
cbar.set_label('Predicted Labels')

# Add labels and title
ax.set_xlabel('LDA Component 1')
ax.set_ylabel('LDA Component 2')
ax.set_zlabel('LDA Component 3')
plt.title('LDA 3D Scatter Plot:')
ax.view_init(elev=20, azim=60)

plt.show()
```

LDA 3D Scatter Plot:



```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=3)
X_TSNE = tsne.fit_transform(X, y)

# Create a 3D scatter plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

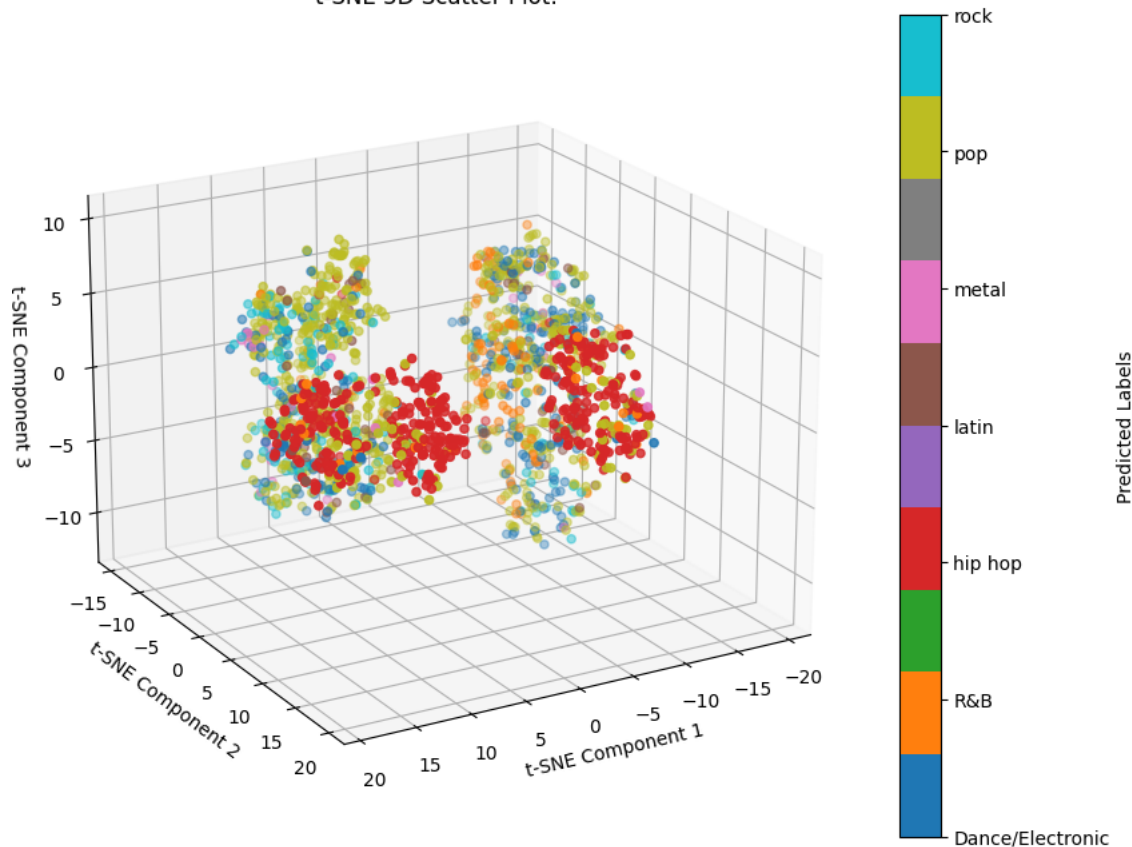
# Defined a colormap for assigning colors based on labels
cmap = plt.get_cmap('tab10')
# Scatter plot with the first three t-SNE components
scatter = ax.scatter(X_TSNE[:, 0], X_TSNE[:, 1], X_TSNE[:, 2], c=y, cmap=cmap)

# Create a custom colorbar with labels based on unique classes
cbar = plt.colorbar(scatter, ax=ax, ticks=np.unique(y))
cbar.set_ticklabels(np.unique(y_original))
cbar.set_label('Predicted Labels')

# Add labels and title
ax.set_xlabel('t-SNE Component 1')
ax.set_ylabel('t-SNE Component 2')
ax.set_zlabel('t-SNE Component 3')
plt.title('t-SNE 3D Scatter Plot:')
ax.view_init(elev=20, azim=60)

plt.show()
```

t-SNE 3D Scatter Plot:



Now we are ready to run our models. Since we are dealing with a classification problem, we will try these models:

- Random Forest
- Naive Bayes
- Stochastic Gradient Descent Classifier
- Logistic Regression

```

from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.cluster import KMeans, DBSCAN
from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import SGDClassifier
from sklearn.multiclass import OneVsOneClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score, cross_val_predict
from sklearn.preprocessing import label_binarize
from sklearn.metrics import precision_score, recall_score, f1_score
import warnings

# We can add parameters as required.
models = [['Naive Bayes', GaussianNB()], ['SGD', OneVsOneClassifier(SGDClassifier())], ['Logistic', LogisticRegression(multi_class=

# Random Forest has to be done separately because it needs binary encoded labels.
RF_clf = RandomForestClassifier(random_state=42, min_samples_split = 5)

res = []
kfold = StratifiedKFold(n_splits=10, random_state=13, shuffle=True)
bin_encoded_ytrain = label_binarize(ytrain, classes=np.unique(ytrain))

RF_cross_val_score = cross_val_score(RF_clf, xtrain, bin_encoded_ytrain, cv=10, scoring='accuracy')
print("Random Forest ->")
print("CrossVal Score Mean:    ", RF_cross_val_score.mean())
print("CrossVal Score Std:    ", RF_cross_val_score.std())

for name, model in models:
    cv_score = cross_val_score(model, xtrain, y_train, cv = kfold, scoring = 'accuracy')
    res.append(cv_score)
    print("")
    print(name,"->")
    print("CrossVal Score Mean:    ", cv_score.mean())
    print("CrossVal Score Std:    ", cv_score.std())

    Random Forest ->
    CrossVal Score Mean:    0.44494324045407635
    CrossVal Score Std:    0.01684579452116015

    Naive Bayes ->
    CrossVal Score Mean:    0.44028637770897827
    CrossVal Score Std:    0.06299859936903071

    SGD ->
    CrossVal Score Mean:    0.5655228758169935
    CrossVal Score Std:    0.045086671245315006

    Logistic ->
    CrossVal Score Mean:    0.5995700034399725
    CrossVal Score Std:    0.02942973985870683

```

All of our models have very low accuracy because we have a lack of data. Moreover, the current data is not a good representation of the different genre classes. It appears that Logistic Regression has the highest accuracy (59.6%). We will now calculate the recall and precision values. Then, we will calculate the F1 score based on that.

```

res_precision_recall = []

y_pred_RF = cross_val_predict(RF_clf, xtrain, bin_encoded_ytrain, cv = 10)
res_precision_recall += [['Random Forest', precision_score(bin_encoded_ytrain, y_pred_RF, average = "micro"), \
    recall_score(bin_encoded_ytrain, y_pred_RF, average = "micro")]]

RF_precision = precision_score(bin_encoded_ytrain, y_pred_RF, average = "micro")
RF_recall = recall_score(bin_encoded_ytrain, y_pred_RF, average = "micro")
RF_F1 = 2 * (RF_precision * RF_recall) / (RF_precision + RF_recall)
print('%s -> %s: %f    %s: %f    %s: %f' % ('Random Forest', 'Precision Score', RF_precision, 'Recall Score', RF_recall, 'F1 Score'))

for name, model in models:
    y_pred = cross_val_predict(model, xtrain, y_train, cv = kfold)
    precision = precision_score(y_train, y_pred, average = "micro")
    recall = recall_score(y_train, y_pred, average = "micro")
    # storing the precision and recall values
    res_precision_recall += [[name, precision, recall]]
    model_f1_score = 2 * (precision * recall) / (precision + recall)

```

```

print("")
print('%s -> %s: %f      %s: %f      %s: %f' % (name, 'Precision Score', precision, 'Recall Score', recall, 'F1 Score', model_f1_
Random Forest -> Precision Score: 0.723110      Recall Score: 0.444954      F1 Score: 0.550913
Naive Bayes -> Precision Score: 0.440367      Recall Score: 0.440367      F1 Score: 0.440367
SGD -> Precision Score: 0.549148      Recall Score: 0.549148      F1 Score: 0.549148
Logistic -> Precision Score: 0.599607      Recall Score: 0.599607      F1 Score: 0.599607

```

Logistic Regression has the best F1 score (59.6) followed by Random Forest Classifier, then SGD and Naive Bayes. Now we will run these models on the test data.

```

precision_scores = []
recall_scores = []
F1_scores = []

RF_clf.fit(xtrain, ytrain)
RF_y_pred = RF_clf.predict(xtest)
precision_scores.append(precision_score(ytest, RF_y_pred, average='weighted', zero_division=1))
recall_scores.append(recall_score(ytest, RF_y_pred, average='weighted'))
F1_scores.append(f1_score(ytest, RF_y_pred, average='weighted'))
print("Random Forest Precision Score: ", precision_scores[0])
print("Random Forest Recall Score: ", recall_scores[0])
print('Random Forest F1 Score: ', F1_scores[0])

for name, model in models:
    model_pred = (model.fit(xtrain, ytrain)).predict(xtest)
    precision_scores.append(precision_score(ytest, model_pred, average='weighted', zero_division=1))
    recall_scores.append(recall_score(ytest, model_pred, average='weighted'))
    F1_scores.append(f1_score(ytest, model_pred, average='weighted'))
    print("")
    print(name, "Precision Score: ", precision_scores[-1])
    print(name, "Recall Score: ", recall_scores[-1])
    print(name, "F1 Score: ", F1_scores[-1])

width = 0.2
x = np.arange(4)

plt.bar(x - width, F1_scores, width, label='F1 Score')
plt.bar(x, precision_scores, width, label='Precision')
plt.bar(x + width, recall_scores, width, label='Recall')

plt.xlabel("Models")
plt.ylabel("Scores")
plt.title("Comparison of Model Performance")
plt.xticks(x, ['Random Forest', 'Naive Bayes', 'SGD', 'Logistic'])
plt.legend()
plt.show()

```

Random Forest Precision Score: 0.6629803828380919
 Random Forest Recall Score: 0.6204188481675392
 Random Forest F1 Score: 0.571231343329644

Naive Bayes Precision Score: 0.5269295384862275
 Naive Bayes Recall Score: 0.40575916230366493
 Naive Bayes F1 Score: 0.42457614276887556

SGD Precision Score: 0.588996202472187
 SGD Recall Score: 0.5785340314136126
 SGD F1 Score: 0.4955796059685683

Logistic Precision Score: 0.6073080972690835
 Logistic Recall Score: 0.5759162303664922
 Logistic F1 Score: 0.5034886291294335

Comparison of Model Performance

```
# kmeans = KMeans(n_clusters=len(np.unique(y)), random_state=42, init='k-means++', n_init="auto").fit(xtrain)
# y_pred = kmeans.predict(xtest)
```

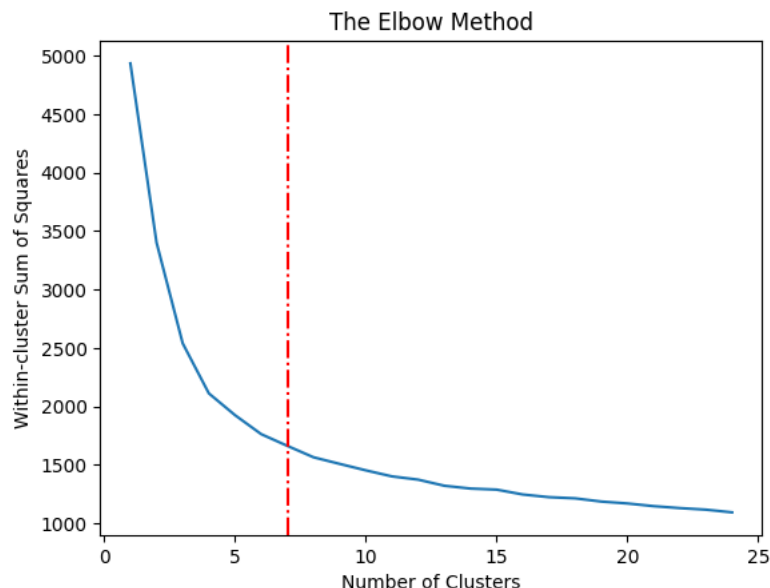
```
wcss = []
# Suppress the FutureWarning
with warnings.catch_warnings():
    warnings.filterwarnings("ignore", category=FutureWarning)
```

```
for i in range(1, 25):
    kmeans = KMeans(i)
    kmeans.fit(xtrain)
    wcss.append(kmeans.inertia_)
```

```
# print("Accuracy:", accuracy_score(ytest, y_pred)*100, "%")
# cm = confusion_matrix(ytest, y_pred)
# sns.heatmap(cm, annot=True)
```

0.1 ↓     |

```
num_clusters = range(1,25)
plt.plot(num_clusters, wcss)
plt.title("The Elbow Method")
plt.xlabel("Number of Clusters")
plt.ylabel("Within-cluster Sum of Squares")
plt.axvline(x=7, color='r', linestyle='-.')
plt.show()
print("This suggests that we can move forward with 7 clusters for KMeans.")
```



This suggests that we can move forward with 7 clusters for KMeans.

```
# kfold = KFold(n_splits=5, shuffle=True, random_state=42)
# kfold.get_n_splits(X)

# ADA_BC_clf = AdaBoostClassifier(n_estimators=250, random_state=0)
# scores = []
# for i, (train_index, test_index) in enumerate(kfold.split(X)):
#     X_train, X_test = X.iloc[train_index,:], X.iloc[test_index,:]
```



```
# y_train, y_test = y.iloc[train_index], y[test_index]
# # Training and evaluating on each fold
# ADA_BC_clf.fit(xtrain, ytrain)
# y_pred = ADA_BC_clf.predict(xtest)
# scores.append(ADA_BC_clf.score(xtest, ytest))
# print(scores)

with warnings.catch_warnings():
    warnings.filterwarnings("ignore", category=FutureWarning)
    kmeans = KMeans(n_clusters=7, init='k-means++', max_iter=100, random_state=42).fit(xtrain)
    print(kmeans.labels_, ytrain)

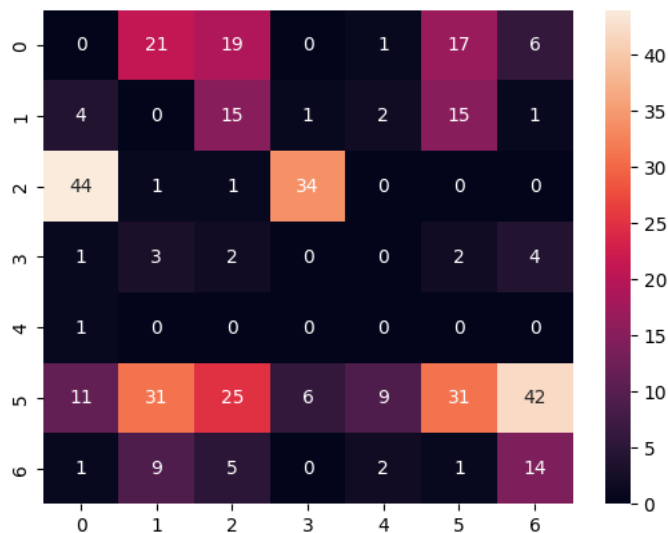
y_pred = kmeans.predict(xtest)

print("Accuracy:", accuracy_score(ytest, y_pred)*100, "%")
cm = confusion_matrix(ytest, y_pred)
sns.heatmap(cm, annot=True)
```

```
[0 5 1 ... 6 1 5] [2 5 5 ... 5 5 6]
```

```
Accuracy: 12.041884816753926 %
```

```
<Axes: >
```



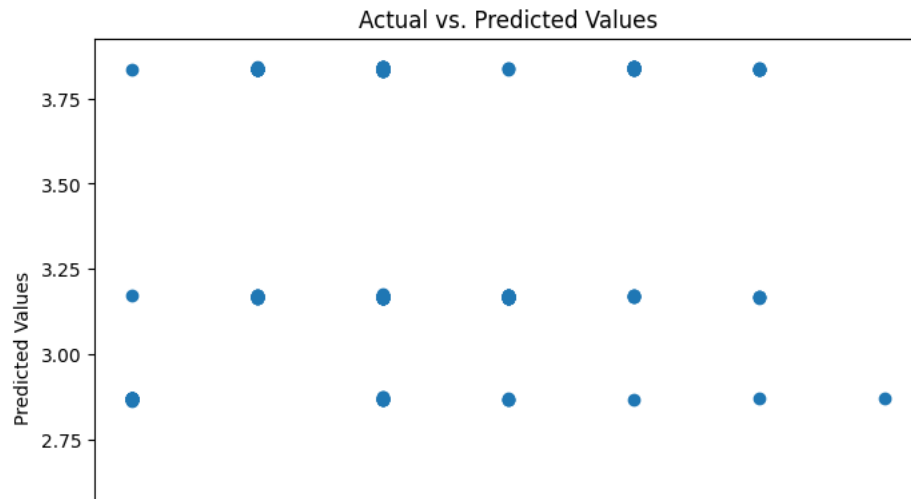
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Lasso
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler

# Create a Lasso model with a specific alpha (regularization strength)
lasso = Lasso(alpha=0.1)

# Fit the Lasso model to the training data
lasso.fit(xtrain, ytrain)
# Make predictions on the test data
y_pred = lasso.predict(xtest)

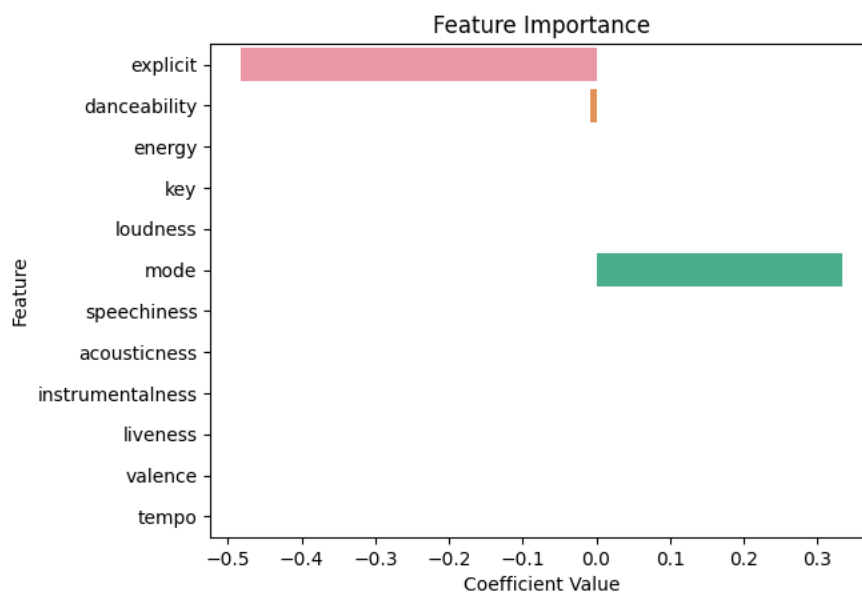
# Evaluate the model's performance (e.g., using Mean Squared Error)
mse = mean_squared_error(ytest, y_pred)
print(f"Mean Squared Error: {mse}")
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred)
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Actual vs. Predicted Values")
plt.show()
```

Mean Squared Error: 4.110939282210901



```
# Get feature coefficients from the Lasso model
feature_importance = lasso.coef_
```

```
# Create a bar plot to show feature importance
sns.barplot(x=feature_importance, y=X.columns)
plt.xlabel("Coefficient Value")
plt.ylabel("Feature")
plt.title("Feature Importance")
plt.show()
```



```
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
import plotly.express as px

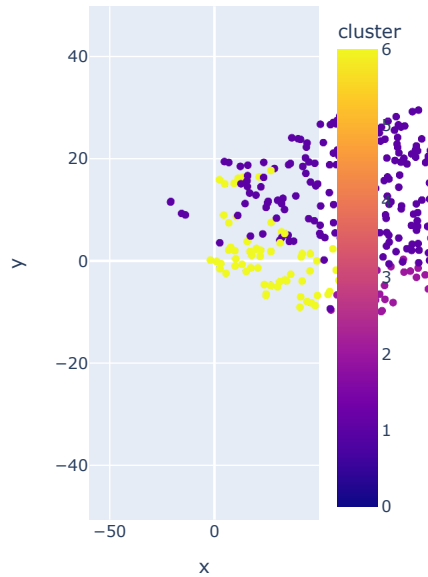
dataf = df_normalized
with warnings.catch_warnings():
    warnings.filterwarnings("ignore", category=FutureWarning)
    cluster_pipeline = Pipeline([('scaler', StandardScaler()), ('kmeans', KMeans(n_clusters=7))])
    A = dataf.select_dtypes(np.number)
    cluster_pipeline.fit(A)
    dataf['cluster'] = cluster_pipeline.predict(A)

tsne_pipeline = Pipeline([('scaler', StandardScaler()), ('tsne', TSNE(n_components=2, verbose=1))])
genre_embedding = tsne_pipeline.fit_transform(A)
projection = pd.DataFrame(columns=['x', 'y'], data=genre_embedding)

projection['genre'] = dataf['genre']
projection['cluster'] = dataf['cluster']
```

```
fig = px.scatter(projection, x='x', y='y', color='cluster', hover_data=['x', 'y', 'genre'])  
fig.show()
```

```
[t-SNE] Computing 91 nearest neighbors...  
[t-SNE] Indexed 1908 samples in 0.002s...  
[t-SNE] Computed neighbors for 1908 samples in 0.122s...  
[t-SNE] Computed conditional probabilities for sample 1000 / 1908  
[t-SNE] Computed conditional probabilities for sample 1908 / 1908  
[t-SNE] Mean sigma: 1.015253  
[t-SNE] KL divergence after 250 iterations with early exaggeration: 72.023224  
[t-SNE] KL divergence after 1000 iterations: 1.370458
```



Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.